MPI Basics

Chris Kauffman

Last Updated:
Mon Feb 9 06:07:29 PM EST 2026

Logistics

Reading: Grama Ch 6 + 4

- Ch 6: MPI basics
- Ch 4: Communication patterns
- Zaratan Setup Link on Schedule (Week 1)

Assignments

- A1 Due Soon
  - On-time by Thu 12-Feb
  - Late through Sat 14-Feb
- **Questions?**
- A2 up next week: MPI Programming

Today

- Primitives for Distributed Memory Computing
- MPI Programming

Next Week

- Communication Patterns
- Thu 19-Feb: Mini-Exam 1

Announcements

Assignment Clarifications

Several clarifications regarding Assignment 1 discussed on Piazza posts, some of which are added to the assignment.

- Network embedding: Mesh into Ring example from last week
- Speedup definition
- Assumptions about perfect trees in networks
- Meaning of times

Coding Environment Setup

Zaratan Setup added to schedule in Week 1
`https://www.cs.umd.edu/~profk/416/setup-guide.html`

- Good idea to code your K-Means solution there to get acquainted as later codes MUST run there
- Provided test cases / `testy` run on Zaratan and most Linuxes, likely not on other platforms

Generic Send and Receive

- Distributed memory machines require explicit sharing of data
- Minimum required functionality is:

```
    send(void *sendbuf, int nelems, int dst_proc);
 receive(void *recvbuf, int nelems, int src_proc);
```

- Referred to as a "point-to-point" communication
- **Sample Use**

```
// P0 runs                // P1 runs
a = 100;                  receive(&a, 1, 0)
send(&a, 1, 1);           printf("%d\n", a);
a=50;
```

- Proc 0 sends a single integer to Proc 1
- Proc 0 then changes that integer
- Proc 1 receives and prints the integer

Programmatic Send/Receive

- Typically write this as a single program which every processor runs: Single Program, Multiple Data (SPMD)
- Assume availability of a function giving logical Proc Number
- Branching on proc number to take different actions

```c
void exchange(){
  int a = -1;
  int my_proc = get_processor_number();
  if(my_proc == 0){
    a = 100;
    send(&a, 1, 1);     // send data 100 to Proc 1
    a=50;
  }
  else if(my_proc == 1){
    receive(&a, 1, 0);  // receive data from Proc 0
    printf("%d\n", a);
  }
  printf("my_proc: %d  a: %d\n",my_proc,a);
}
```

Flavors of Send/Receive

- Hardware+OS may support copying message into a "buffer" space which allows sending program to proceed faster: copy to buffer, OS/Hardware handles the rest
- Functions usually available to do both Blocking `send()` and Nonblocking "immediate" `isend()` BUT without OS/Hardware support they are the same

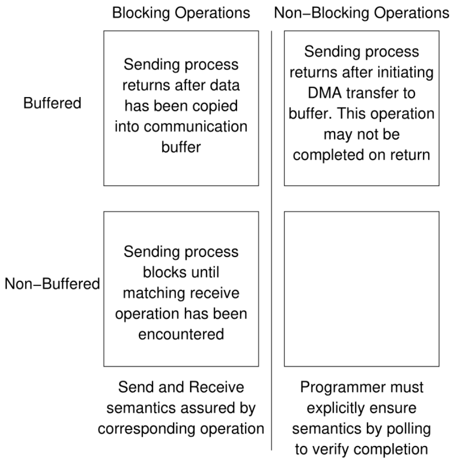
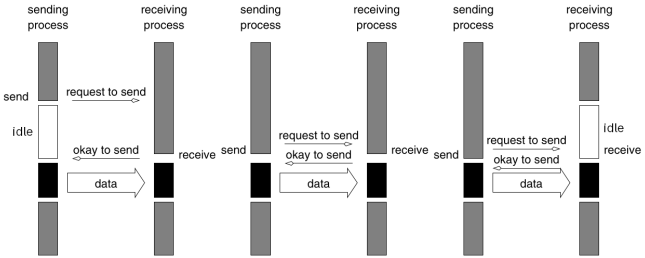

Blocking Operations
Non–Blocking Operations
Buffered
Sending process returns after data has been copied into communication buffer
Sending process returns after initiating DMA transfer to buffer. This operation may not be completed on return
Non–Buffered
Sending process blocks until matching receive operation has been encountered
Send and Receive semantics assured by corresponding operation
Programmer must explicitly ensure semantics by polling to verify completion

Figure 6.3 Space of possible protocols for send and receive operations.

Blocking + Unbuffered Send/Receive



(a) Sender comes first; idling at sender
(b) Sender and receiver come at about the same time; idling minimized
(c) Receiver comes first; idling at receiver

Figure 6.1 Handshake for a blocking non-buffered send/receive operation. It is easy to see that in cases where sender and receiver do not reach communication point at similar times, there can be considerable idling overheads.

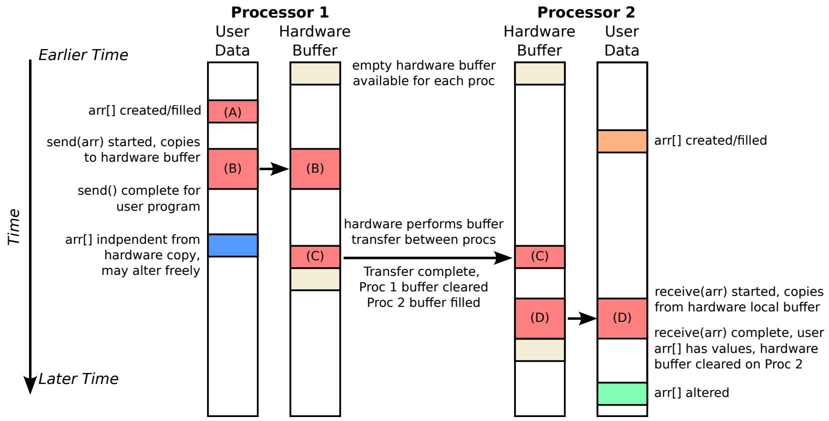
Blocking/Unbuffered: no extra buffer available to hold pending sends/receives so must wait until message is sent to proceed
Blocked processors are idle, do no work, which cuts into speedup

Improper Send/Receive Order May Deadlock

```
// P0                      // P1
send(&a, 1, 1);            send(&a, 1, 0);  // both blocked
receive(&b, 1, 1);         receive(&b, 1, 0);
```

- Above ordering is a bug: both processors block until the other receives a message
- They may both wait forever, a form of Deadlock
- On some platforms for some message sizes, the above may NOT produce deadlock due to `send()` being **buffered** by OS/Hardware

Blocking + Buffered Send/Receive


Processor 1
User Data
Hardware Buffer
Processor 2
Hardware Buffer
User Data
Earlier Time
Time
Later Time
empty hardware buffer available for each proc
arr[] created/filled
(A)
send(arr) started, copies to hardware buffer
(B)
(B)
send() complete for user program
arr[] indpendent from hardware copy, may alter freely
hardware performs buffer transfer between procs
(C)
(C)
Transfer complete, Proc 1 buffer cleared Proc 2 buffer filled
arr[] created/filled
receive(arr) started, copies from hardware local buffer
(D)
(D)
receive(arr) complete, user arr[] has values, hardware buffer cleared on Proc 2
arr[] altered

Blocking + Buffered Send/Receive

Commonly finite-sized hardware communications buffers are available that offload data transfer work from the main processor; decouples `send()/receive()` operations on processors

Buffers are no Panacea: Deadlocks Still Possible

Receive always Blocks

```
// P0                      // P1
receive(&a, 1, 1);         receive(&a, 1, 0);
send(&b, 1, 1);            send(&b, 1, 0);
```

- `receive()` always blocks until message is obtained
- Does the above code work even in the buffered setting?

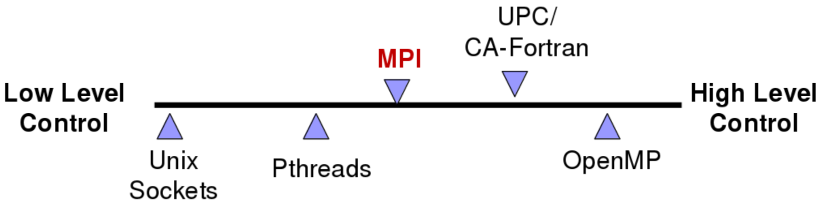
Finite Buffer Sizes

```
// P0                      // P1
send(&a, N, 1);            send(&a, N, 0);
receive(&b, N, 1);         receive(&b, N, 0);
```

- For small `N`, does not deadlock due to buffers
- For large `N`, does deadlock due to finite buffer size

MPI: Message Passing Interface

- Standardized library of functions for C/C++/Fortran
- Communicate between processors in a distributed memory machine
- First appearing around 1992
- MPI Version 1.x universally deployed, Version 2.x less so
- Open Source Versions: MPICH, Open MPI
- Proprietary Versions: Intel, Platform, IBM, Platform, Cray
- Typically vendor configures MPI for particular architecture / network of a large-scale machine



MPI In a Nutshell: 6 Essential Functions

```
// Initialize and Terminate MPI
int MPI_Init(int *argc, char ***argv);
int MPI_Finalize();

// Get total number of processors
int MPI_Comm_size(MPI_Comm comm, int *size);

// Get logical proc number of calling process
int MPI_Comm_rank(MPI_Comm comm, int *rank);

// Send a message to dest processor
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
             int dst_proc, int tag, MPI_Comm comm);

// Receive a message from source processor
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
             int src_proc, int tag, MPI_Comm comm,
             MPI_Status *status);
```

MPI Hello World

```c
// mpi_hello.c: C Example of hello world with MPI.  Compile and run as
// > mpicc -o mpi_hello mpi_hello.c
// > mpirun ./mpi_hello      # use number of processors equal to total machine
// > mpirun -np 2 mpi_hello  # use 2 processors
// > mpirun -np 8 mpi_hello  # use 8 processors

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

int main (int argc, char *argv[]){
  int rank;                         // the id of this processor
  int size;                         // the number of processors being used

  MPI_Init (&argc, &argv);                   // starts MPI
  MPI_Comm_rank (MPI_COMM_WORLD, &rank); // get current process id
  MPI_Comm_size (MPI_COMM_WORLD, &size); // get number of processes

  // Say hello from this proc
  printf( "Proc %d of %d says 'Hello world'\n", rank, size );

  MPI_Finalize();
  return 0;
}
```

Compiling and Running

- Demo using openmpi implementation
- `mpirun` for interactive running
- `mpirun -np 4 progr` sets number of “processors” to 4

```
>> cd 04-mpi-code/
>> mpicc -o mpi_hello mpi_hello.c
>> ./mpi_hello
...
Proc 0 of 1 says 'Hello world'

>> mpirun -np 2 mpi_hello
...
Proc 0 of 2 says 'Hello world'
Proc 1 of 2 says 'Hello world'

>> mpirun mpi_hello
...
Proc 2 of 4 says 'Hello world'
Proc 0 of 4 says 'Hello world'
Proc 1 of 4 says 'Hello world'
Proc 3 of 4 says 'Hello world'
```

MPI Implementations and OpenMPI Warnings

- Several Implementations of MPI:
  - OpenMPI and MPICH are free, open source, widely available
  - HPC Vendors like IBM and Cray provide their own tailored MPI versions
- Recent Versions of OpenMPI can complain a LOT about various items missing; usually options / environment vars exist to suppress these warnings
  - Example: `--mca btl_base_warn_component_unused 0` to warn about missing HPC network components during `mpirun`
  - Example: `--mca opal_warn_on_missing_libcuda 0` if not intending to use GPU libraries
- Exact nature of warnings/errors varies a lot, look at messages which often dictate how to disable them
- Zaratan does not warn a lot as it seems configured well but course staff may provide specific instructions if warnings emerge

Warning Suppression in OpenMPI

```
>> mpicc mpi_hello_plus.c

# On the below machine, warnings abound
>> mpirun -np 2 a.out
--------------------------------------------------------------------------
The library attempted to open the following supporting CUDA libraries,
but each of them failed.  CUDA-aware support is disabled.
libcuda.so.1: cannot open shared object file: No such file or directory
libcuda.dylib: cannot open shared object file: No such file or directory
/usr/lib64/libcuda.so.1: cannot open shared object file: No such file or directory
/usr/lib64/libcuda.dylib: cannot open shared object file: No such file or directory
If you are not interested in CUDA-aware support, then run with
--mca opal_warn_on_missing_libcuda 0 to suppress this message.  If you are interested
in CUDA-aware support, then try setting LD_LIBRARY_PATH to the location
of libcuda.so.1 to get passed this issue.
--------------------------------------------------------------------------
P0000 [val]: Hello world from process    0 of 2
P0001 [val]: Hello world from process    1 of 2
[val:558294] 1 more process has sent help message help-mpi-common-cuda.txt / dlopen failed
[val:558294] Set MCA parameter "orte_base_help_aggregate" to 0 to see all help / error messages

# in BASH, sourcing provided script sets some environment variables to quiet things down
>> source mpiopts.sh

>> mpirun $MPIOPTS -np 2 a.out
P0001 [val]: Hello world from process    1 of 2
P0000 [val]: Hello world from process    0 of 2
```

MPI Oversubscribing

Default OpenMPI config uses all processors on a single machine, fails for larger requests unless using option `--oversubscribe`

```
>> mpirun -np 2 a.out
...
P0001 [val]: Hello world from process    1 of 2
P0000 [val]: Hello world from process    0 of 2

>> mpirun -np 16 a.out
--------------------------------------------------------------------------
There are not enough slots available in the system to satisfy the 16
slots that were requested by the application:

  a.out

Either request fewer slots for your application, or make more slots
available for use.
...
Alternatively, you can use the --oversubscribe option to ignore the
number of available slots when deciding the number of processes to
launch.
--------------------------------------------------------------------------

>> mpirun  --oversubscribe -np 16 a.out
P0009 [val]: Hello world from process    9 of 16
...
P0014 [val]: Hello world from process   14 of 16
P0012 [val]: Hello world from process   12 of 16

# Newer MPI versions prefer `--map-by :OVERSUBSCRIBE` to `--oversubscribe`
```

Hostfiles

- For laptop/desktop MPI or on Zaratan, system config establishes the nodes with processors
- Some clusters instead use a **hostfile** in `mpirun` to indicate host names of other machines in cluster
- Simplest form of hostfile is a list of symbolic or IP addresses for machines to recruit for the run

```
myclust>> cat hostfile.txt
node01.mycluster.umd.edu
node02.mycluster.umd.edu
node03.mycluster.umd.edu
node04.mycluster.umd.edu
node05.mycluster.umd.edu

myclust>> mpirun -hostfile hostfile.txt -np 100 ./a.out
...
P0003 [node01]: Hello world from process    3 of 100
...
P0089 [node02]: Hello world from process   39 of 100
...
P0190 [node05]: Hello world from process   89 of 100
```

- Again, **we won't need hostfiles as Zaratan is already configured for MPI**

MPI On Zaratan

- UMD's HPC cluster is Zaratan, access via

  ```
  >> ssh MYID@login.zaratan.umd.edu
  ```
- **Login Nodes** are `login-1 login-2 login-3` and are meant for development, compilation, data manipulation, short (<= 1min) tests
- Most nodes (including login) have 128 cores on them but are shared among all logged in useres
- Keep code runs on `login` short and sweet
- When experimenting with MPI, limit procs requested to **no more than 16**

  ```
  profk@login-1 [04-mpi-code]% mpicc -o mpi_hello_plus mpi_hello_plus.c
  profk@login-1 [04-mpi-code]% mpirun -np 16 mpi_hello_plus
  P0011 [login-1.zaratan.umd.edu]: Hello world from process   11 of 16
  P0013 [login-1.zaratan.umd.edu]: Hello world from process   13 of 16
  P0014 [login-1.zaratan.umd.edu]: Hello world from process   14 of 16
  ...
  ```
- To request longer jobs and more processors, use the **SLURM Scheduler** discussed in more detail later

MPI Hello On Zaratan

```
# assumes correct environment configuration on Zaratan,
# that ~/.bashrc has lines below for mpicc / mpirun
#   source ~profk/bin/cmsc216-env
#   source ~profk/bin/cmsc416-env

>> ssh profk@login.zaratan.umd.edu
...
Last login: Thu Jan 29 17:18:51 2026
profk@login-1 [04-mpi-code]% cd 416/04-mpi-code/
profk@login-1 [04-mpi-code]% mpicc -o mpi_hello_plus mpi_hello_plus.c

profk@login-1 [04-mpi-code]% mpirun -np 16 mpi_hello_plus
P0011 [login-1.zaratan.umd.edu]: Hello world from process  11 of 16
P0013 [login-1.zaratan.umd.edu]: Hello world from process  13 of 16
P0014 [login-1.zaratan.umd.edu]: Hello world from process  14 of 16
P0010 [login-1.zaratan.umd.edu]: Hello world from process  10 of 16
P0000 [login-1.zaratan.umd.edu]: Hello world from process   0 of 16
P0002 [login-1.zaratan.umd.edu]: Hello world from process   2 of 16
P0004 [login-1.zaratan.umd.edu]: Hello world from process   4 of 16
P0005 [login-1.zaratan.umd.edu]: Hello world from process   5 of 16
P0006 [login-1.zaratan.umd.edu]: Hello world from process   6 of 16
P0007 [login-1.zaratan.umd.edu]: Hello world from process   7 of 16
P0009 [login-1.zaratan.umd.edu]: Hello world from process   9 of 16
P0012 [login-1.zaratan.umd.edu]: Hello world from process  12 of 16
P0015 [login-1.zaratan.umd.edu]: Hello world from process  15 of 16
P0001 [login-1.zaratan.umd.edu]: Hello world from process   1 of 16
P0003 [login-1.zaratan.umd.edu]: Hello world from process   3 of 16
P0008 [login-1.zaratan.umd.edu]: Hello world from process   8 of 16
```

MPI Send and Recieve

MPI's most basic functionality is point-to-point message transfer via `MPI_Send()` / `MPI_Recv()`

```c
{
  int count = 5;
  int a[count]={10,20,30,40,50};
  int b[count];
  int partner = 1;
  int tag = 1;

  // Send contents of a to partner proc with tag=1
  MPI_Send(a, count, MPI_INT, partner, tag, MPI_COMM_WORLD);

  // Receive message into b from partner proc
  MPI_Recv(b, count, MPI_INT, partner, tag, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);  // ignore status of receipt
}
```

- Analyze the codepack program `send_recv_demo.c`
  - Study code then try several runs
  - Pick a number of procs that will cause it problems
- Compare with `send_recv_bugs.c`
  - Vary the `ss` and `rr` param to demo stalls
  - Vary the size of buffers sent to check implementation
  - Note Laptop vs Zaratan buffer sizes differ

Tags Make Messages Unique

```
int a[10], b[10], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
  //////////////////////////  V : TAG to send
  MPI_Send(a, 10, MPI_INT, 1, 123, MPI_COMM_WORLD);
  MPI_Send(b, 10, MPI_INT, 1, 456, MPI_COMM_WORLD);
}
else if (myrank == 1) {
  //////////////////////////  V : TAG to receive
  MPI_Recv(b, 10, MPI_INT, 0, 456, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  MPI_Recv(a, 10, MPI_INT, 0, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

- Send/Recv have an integer Tag parameter which must match between sender and receiver
- Above code may deadlock if not buffered due to the misordering of Tags
- Mostly we will use `tag=1` for simplicity
- Alternatively `MPI_ANY_TAG` works for `MPI_Recv()`
- Receiver can capture the received status (last param) and query what tag was received (we won't bother with this)

MPI Data Types Supported

```
// Sends a message.
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
             int dst_proc, int tag, MPI_Comm comm);

// Receives a message.
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
             int src_proc, int tag, MPI_Comm comm,
             MPI_Status *status);
```

- Buffer is always untyped (`void* buf`)
- To strive for slightly better safety, MPI has standard datatypes

`MPI_CHAR`	signed char
`MPI_INT`	signed int
`MPI_LONG`	signed long int
`MPI_FLOAT`	float
`MPI_DOUBLE`	double
`MPI_BYTE`	Last two used for sending
`MPI_PACKED`	structure arrays

Unsigned types also available

Exercise: Heat Transfer in MPI

- Discuss conversion of the following A1 code to an MPI version
- How is data in `H[][]` divided up?
- Is communication required?
- How would one arrange `MPI_Send() / MPI_Recv()` calls?
- How much data needs to be transferred and between who?
- When the computation is finished, how can all data be printed / displayed / saved?

```
// Simulate the temperature changes for internal cells
for(t=0; t<max_time-1; t++){
  for(p=1; p<width-1; p++){
    double left_diff  = H[t][p] - H[t][p-1];
    double right_diff = H[t][p] - H[t][p+1];
    double delta = -k*( left_diff + right_diff );
    H[t+1][p] = H[t][p] + delta;
  }
}
```

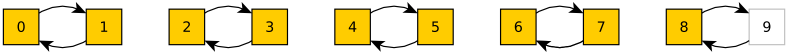
Some Patterns that occur in the problem

- Pair exchange of items: made easier with `MPI_Sendrecv()`
- Collecting final output for display: `MPI_Gather()`
  - Previewed here
  - Discussed in following lectures

Exchange: Sendrecv() for exchanging data between pairs

```
{
  double send[10], recv[10]; int partner;
  if(procid % 2 == 1 ){ // odd procs send left, receive left
    partner = procid-1;
    MPI_Send(send, 10, MPI_DOUBLE, partner, 1, MPI_COMM_WORLD);
    MPI_Recv(recv, 10, MPI_DOUBLE, partner, 1, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
  }
  else{                 // even procs receive right, send right
    partner = procid+1;
    MPI_Recv(recv, 10, MPI_DOUBLE, partner, 1, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
    MPI_Send(send, 10, MPI_DOUBLE, partner, 1, MPI_COMM_WORLD);
  }
}
{ // Sendrecv simplifies this pattern
  double send[10], recv[10]; int partner;
  partner = (procid % 2 == 1) ? procid-1 : procid+1;
  MPI_Sendrecv(send, 10, MPI_DOUBLE, partner, 1,
               recv, 10, MPI_DOUBLE, partner, 1,
               MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

Take Care: Pair exchange can hang

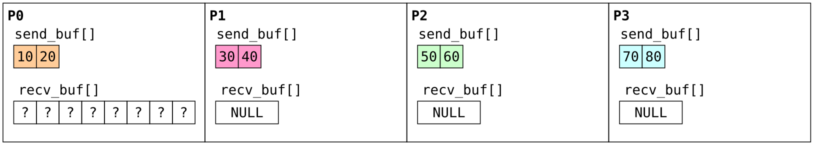


```
{
  double send[10], recv[10]; int partner;
  partner = (procid % 2 == 1) ? procid-1 : procid+1;
  MPI_Sendrecv(send, 10, MPI_DOUBLE, partner, 1,
               recv, 10, MPI_DOUBLE, partner, 1,
               MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

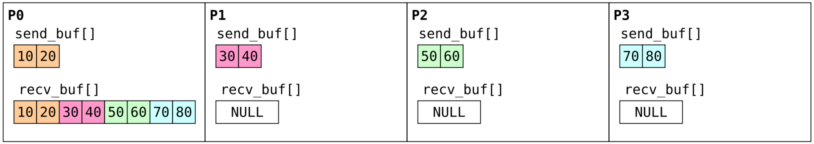
- With 9 processors, logic is broken
- Proc 8 will wait to communicate with absent partner
- Program never terminates
- Special checks for `procid == (nprocs-1)` can surmount this
- Heat problem is interesting as it involves communicating with a Left and Right partner, yet more complex and part of A2...

Getting Answers to Root Proc: Gather

Before Call Begins



```
MPI_Gather(send_buf,2,MPI_INT, recv_buf,2,MPI_INT, 0,MPI_COMM_WORLD);
```



After Call Completes

- At the end of Heat, every processor has computed some columns
- One processor needs to gather all of the data for printing / saving, usually “root” processor with ID/rank 0
- Everyone calls `MPI_Gather()` to get data to root proc

MPI_Gather Sample

Use of Gather

```
// Preamble for any code
MPI_Comm comm = MPI_COMM_WORLD;
int sendarray[100];
int procid, total_procs, *rbuf;
...;
// Only proc 0 needs space for to
// receive entirety of data
if(procid == 0) {
  rbuf = malloc(total_procs*100*
                sizeof(int));
}

// Everyone calls gather
// proc 0 gets all data eventually
MPI_Gather(sendarray, 100, MPI_INT,
           rbuf, 100, MPI_INT,
           0, comm);
```

Equivalent Non-Gather Code

```
if(rank == 0){
  for(i=0; i<100; i++){
    rbuf[i] = sendarray[i];
  }
  for(i=1; i<total_procs; i++){
    int *rloc = &rbuf[i*100];
    MPI_Recv(rloc, 100,
             MPI_INT, i,
             tag, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
  }
}
else{
  MPI_Send(sendarray, 100,
           MPI_INT, 0,
           tag, MPI_COMM_WORLD);
}
```

Collective Communication Patterns Next

- `gather` is an example of a class of **Collective Communication Patterns**
- Will study more of these in subsequent lectures
- Using built-in collective comm. patterns simplifies programs
- MPI implementation typically uses the most efficient underlying communications for a particular network

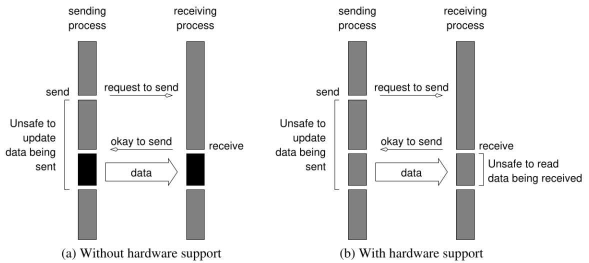
Sending Structs

Sending structs can be done via the MPI_BYTE type

```
{                               // from send_structs.c
  typedef struct {
    double x;
    int a, b;
  } dint_t;
  dint_t mine[10] = { {.x=1.23, .a=5, .b=7}, {.x=...}, ...}
  ...;
  // calculate data sizes "manually" just as is done in a malloc()
  MPI_Send(mine,  10*sizeof(dint_t), MPI_BYTE,
           partner, 1, MPI_COMM_WORLD);
}
```

- Simple and effective if all compute nodes **use the same binary layout** (the typical case)
- MPI also provides a (complex) method for situations where struct layout differs between nodes
- Must Dictate # of struct fields, types, and ordering into a `MPI_Datatype` and use `MPI_Type_create_struct()`
- Likely hurts performance if struct layout differs so will not discuss in detail

Non-Blocking Communication: “Immediate” Operations



sending process
receiving process
send
request to send
Unsafe to update data being sent
okay to send
receive
data
(a) Without hardware support
sending process
receiving process
send
request to send
Unsafe to update data being sent
okay to send
receive
data
Unsafe to read data being received
(b) With hardware support

Figure 6.4 Non-blocking non-buffered send and receive operations (a) in absence of communication hardware; (b) in presence of communication hardware.

- Non-Blocking calls return immediately
  - `isend(data,dest,request)`: send w/o waiting
  - `ireceive(data,dest,request)`: receive w/o waiting
- `wait(request)`: block until requested send/receive finishes, required to avoid race conditions
- Will revisit later if time permits:
  More Control = More Potential Speedup + More Potential Bugs

Non-Blocking Send / Receive in MPI

Effect	**Blocking**	**Non-Blocking**
Send	`MPI_Send(...)`	`MPI_Isend(..., &request)`
Receive	`MPI_Recv(...)`	`MPI_Irecv(..., &request)`
Sync	Automatic	`MPI_Wait(&request, ...)`

- Non-blocking calls trigger send/receive to be initiated but do not block process(or) to completion
- `MPI_Request` struct tracks whether operation has completed
- Block via `MPI_wait()` until send/recv completes
- Prior to blocking, unsafe to alter/use data in buffers
- Can pair `MPI_ISend() / MPI_Recv()` and vice versa
- Allows for more overlap of computation and communication at the cost of more complexity

Faux Example of MPI_Isend() / MPI_IRecv()

```
int data_a[100] = {...};
int data_b[100] = {...};
int partner = ...;
int tag = ...;

...; // compute data_a[]

MPI_Request request;
MPI_Isend(data_a, 100, MPI_INT, partner, tag, MPI_COMM_WORLD, &request);

...; // unsafe to alter data_a[] so compute data_b[]

MPI_Wait(&request, MPI_STATUS_IGNORE); // block until data_a[] has been sent
                                       // now safe to alter data_a[]
for(int i=0; i<100; i++){              // more computations on data_a[]
  data_a[i]++;
}
...;
MPI_Irecv(data_a, 100, MPI_INT, partner, tag, MPI_COMM_WORLD, &request);

...; // unsafe to do anything with data_a[], compute on data_b[]

MPI_Wait(&request, MPI_STATUS_IGNORE); // block until data_a[] has been sent
                                       // now safe to alter data_a[]
for(int i=0; i<100; i++){              // more computations on data_a[]
  data_a[i]++;
}
```