

MPI Basics

Chris Kauffman

*Last Updated:
Tue Feb 7 01:45:04 PM CST 2023*

Logistics

Reading: Grama Ch 6 + 4

- ▶ Ch 6: MPI basics
- ▶ Ch 4: Communication patterns

Assignments

- ▶ A1 Due Soon
 - ▶ On-time by Thu 02-Feb
 - ▶ Late through Sat 04-Feb
- ▶ **Questions?**
- ▶ A2 up next week:
MPI Programming

Today

- ▶ Primitives for Distributed Memory Computing
- ▶ MPI Programming

Next Week

- ▶ Comm Patterns
- ▶ Thu 09-Feb: Mini-Exam 1

Generic Send and Receive

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

```
send(void *sendbuf, int nelems, int dest)
receive(void *recvbuf, int nelems, int source)
```

- ▶ Referred to as a “point-to-point” communication

- ▶ **Sample Use**

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

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

More typical appearance

- ▶ 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

```
1 void exchange(){
2     int a = 100;
3     int my_proc = get_processor_number();
4     if(my_proc == 0){
5         send(&a, 1, 1);    // send data 100 to Proc 1
6         a=50;
7     }
8     else if(my_proc == 1){
9         receive(&a, 1, 0); // receive data from Proc 0
10        printf("%d\n", a);
11    }
12 }
```

Flavors Send/Receive

- ▶ Hardware+OS may support copying message into a “buffer” space to make allowing program to proceed faster
- ▶ Functions usually available to do both blocking `send()` and `send_nonblocking()` with hardware support BUT without OS/hardware support they are the same

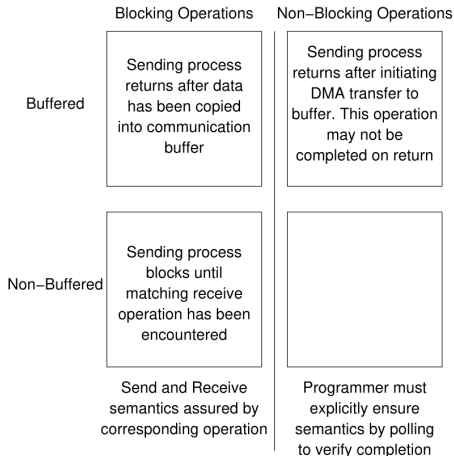


Figure 6.3 Space of possible protocols for send and receive operations.

Blocked and Unbuffered

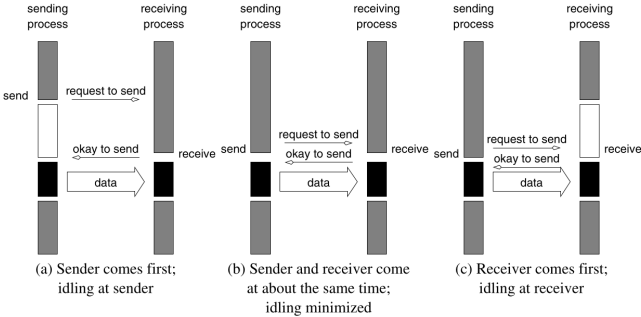


Figure 6.1 Handshake for a blocking non-buffered send/receive operation. It is easy to see that in cases where sender and receiver do not reach communication point at similar times, there can be considerable idling overheads.

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

Ordering of Send Receive

1	P0		P1
2			
3	<code>send(&a, 1, 1);</code>		<code>send(&a, 1, 0);</code>
4	<code>receive(&b, 1, 1);</code>		<code>receive(&b, 1, 0);</code>

Assuming send/receive blocked/unbuffered, what's wrong with the above code?

Blocking with Buffers

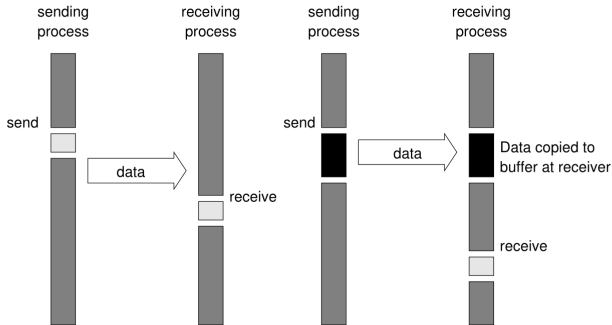


Figure 6.2 Blocking buffered transfer protocols: (a) in the presence of communication hardware with buffers at send and receive ends; and (b) in the absence of communication hardware, sender interrupts receiver and deposits data in buffer at receiver end.

Hardware buffer support, sender and receiver have a memory minion

No buffer support: sender interrupts receiver

The Danger Continues

```
1 P0                P1
2
3 receive(&a, 1, 1);  receive(&a, 1, 0);
4 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?

Non-blocking Communication

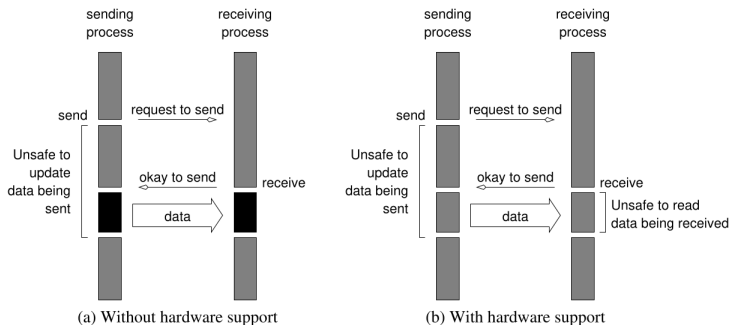
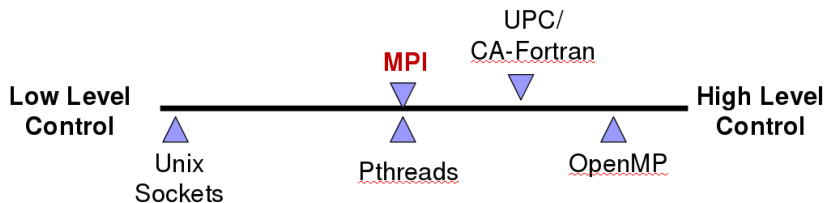


Figure 6.4 Non-blocking non-buffered send and receive operations (a) in absence of communication hardware; (b) in presence of communication hardware.

- ▶ Takes a bit more work on the programming side
- ▶ Must explicitly ensure that transaction completes with function calls
- ▶ `isend(data, dest, status)`: send w/o waiting
- ▶ `ireceive(data, dest, status)`: receive w/o waiting
- ▶ `wait(status)`: wait until a message has been sent or

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 implementations: MPICH, Open MPI
- ▶ Proprietary: 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 dest, int tag, MPI_Comm comm);

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

MPI Hello World

```
1 // mpi_hello.c: C Example of hello world with MPI.  Compile and run as
2 // > mpicc -o mpi_hello mpi_hello.c
3 // > mpirun ./mpi_hello      # use number of processors equal to total machine
4 // > mpirun -np 2 mpi_hello  # use 2 processors
5 // > mpirun -np 8 mpi_hello  # use 8 processors
6
7 #include <stdio.h>
8 #include <mpi.h>
9
10 int main (int argc, char *argv[]){
11     int rank;                // the id of this processor
12     int size;                // the number of processors being used
13
14     MPI_Init (&argc, &argv);          // starts MPI
15     MPI_Comm_rank (MPI_COMM_WORLD, &rank); // get current process id
16     MPI_Comm_size (MPI_COMM_WORLD, &size); // get number of processes
17
18     // Say hello from this proc
19     printf( "Proc %d of %d says 'Hello world'\n", rank, size );
20
21     MPI_Finalize();
22     return 0;
23 }
```

Compilation 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
- ▶ The many machines with MPI are not configured perfectly leading to additional errors
 - ▶ 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
- ▶ Provided `mpiopts.sh` script can be sourced to set suppress common errors

Warning Suppression in OpenMPI

```
>> mpicc mpi_hello_plus.c
```

```
>> 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
```

```
>> 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 `--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 8 a.out
```

```
-----
There are not enough slots available in the system to satisfy the 8
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.
```

```
-----
>> source mpiopts.sh
>> echo $MPIOPTS
--mca opal_warn_on_missing_libcuda 0 --oversubscribe
#
```

```
>> mpirun $MPIOPTS -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
```

Hostfiles

- ▶ For simple cluster configurations, can pass a **hostfile** to `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
- ▶ CSE Labs CUDA cluster¹ has the following machines which can be used for experimentation

```
cuda01.cselabs.umn.edu  
cuda02.cselabs.umn.edu  
cuda03.cselabs.umn.edu  
cuda04.cselabs.umn.edu  
cuda05.cselabs.umn.edu
```

- ▶ 40 physical cores per machine

¹CUDA cluster is present to support this class, thus MPI is set up for it. We will also use it later for GPU programming.

Extended Example on CUDA cluster 1/2

```
### log in to lab machines
>> ssh cuda01.cselabs.umn.edu
...
csel-cuda-01>> cat hostfile-cuda-full.txt
cuda01.cselabs.umn.edu
cuda02.cselabs.umn.edu
cuda03.cselabs.umn.edu
cuda04.cselabs.umn.edu
cuda05.cselabs.umn.edu

### compile + run mpi program
csel-cuda-01>> mpicc mpi_hello_plus.c
csel-cuda-01>> mpirun -hostfile hostfile-cuda-full.txt -np 32 ./a.out
No protocol specified
No protocol specified
P0023 [csel-cuda-01]: Hello world from process 23 of 32
P0024 [csel-cuda-01]: Hello world from process 24 of 32
...
P0002 [csel-cuda-01]: Hello world from process 2 of 32
P0013 [csel-cuda-01]: Hello world from process 13 of 32

### 40 processors per machine
csel-cuda-01>> mpirun -hostfile hostfile-cuda-full.txt -np 64 ./a.out
...
P0011 [csel-cuda-01]: Hello world from process 11 of 64
P0020 [csel-cuda-01]: Hello world from process 20 of 64
..
P0039 [csel-cuda-01]: Hello world from process 39 of 64
P0041 [csel-cuda-02]: Hello world from process 41 of 64
P0045 [csel-cuda-02]: Hello world from process 45 of 64
P0046 [csel-cuda-02]: Hello world from process 46 of 64
...
```

Extended Example on CUDA cluster 2/2

```
### utilize whole cluster
csel-cuda-01>> mpirun -hostfile hostfile-cuda-full.txt -np 200 ./a.out
...
P0003 [csel-cuda-01]: Hello world from process    3 of 200
...
P0089 [csel-cuda-03]: Hello world from process   89 of 200
...
P0190 [csel-cuda-05]: Hello world from process  190 of 200
P0123 [csel-cuda-04]: Hello world from process  123 of 200
...
P0077 [csel-cuda-02]: Hello world from process   77 of 200
```

```
### 200 processors total in CUDA cluster; going over this errors out
csel-cuda-01 [04-mpi-code]% mpirun -hostfile hostfile-cuda-full.txt -np 201 ./a.out
```

There are not enough slots available in the system to satisfy the 201 slots that were requested by the application:

./a.out

Either request fewer slots for your application, or make more slots available for use.

A "slot" is the Open MPI term for an allocatable unit where we can launch a process. The number of slots available are defined by the environment in which Open MPI processes are run:

1. Hostfile, via "slots=N" clauses (N defaults to number of processor cores if not provided)
2. The --host command line parameter, via a ":N" suffix on the hostname (N defaults to 1 if not provided)
3. Resource manager (e.g., SLURM, PBS/Torque, LSF, etc.)
4. If none of a hostfile, the --host command line parameter, or an RM is present, Open MPI defaults to the number of processor cores

Distributed Memory (MPI) Systems at UMN

CUDA Cluster

- ▶ `cse1-cuda01.cselabs.umn.edu` to `cse1-cuda05.cselabs.umn.edu`
- ▶ Have OpenMPI installed, honor hostfile
- ▶ Hostfile in `04-mpi-code.zip` as `hostfile-cuda-full.txt`
- ▶ Good for experimentation but not a true HPC system
- ▶ Requires setting up SSH keys / Known Hosts²

MSI Systems

- ▶ Will use MSI to evaluate scalability of program performance for A2
- ▶ Usually no need to use a hostfile as MPI jobs are run in batch and number of nodes is requested as part of job
- ▶ Requires use of job scheduling system SLURM, discuss later

²See [Accessing Unix/Linux Programming Environments Section 3](#) for instructions on setting up keys for password/Duo free login to CSE Labs

MPI Send and Recieve

Most basic functionality is point-to-point message transfer via
MPI_Send() / MPI_Recv()

```
1 int count = 5;
2 int a[count]={10,20,30,40,50};
3 int b[count];
4 int partner = 1;
5 int tag = 1;
6
7 // Send contents of a to partner proc with tag=1
8 MPI_Send(a, count, MPI_INT, partner, tag, MPI_COMM_WORLD);
9
10 // Receive message into b from partner proc
11 MPI_Recv(b, count, MPI_INT, partner, tag, MPI_COMM_WORLD,
12         MPI_STATUS_IGNORE); // ignore status of receipt
```

- ▶ Analyze the program `send_receive_test.c`
- ▶ **Compare** with `send_bugs.c` which demos stall problems
- ▶ Note `MPI_ANY_SOURCE` may be used for `recv`'s source

Tags Make Messages Unique

```
int a[10], b[10], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
    MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
    MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
    MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

- ▶ Tags must be honored on receive
- ▶ 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, possible to query what tag was received later on (though we won't have cause to do this)

MPI Data Types Supported

```
// Sends a message.
```

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,  
             int dest, int tag, MPI_Comm comm);
```

```
// Receives a message.
```

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,  
             int source, 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 displayed?

```
// 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);
}
```

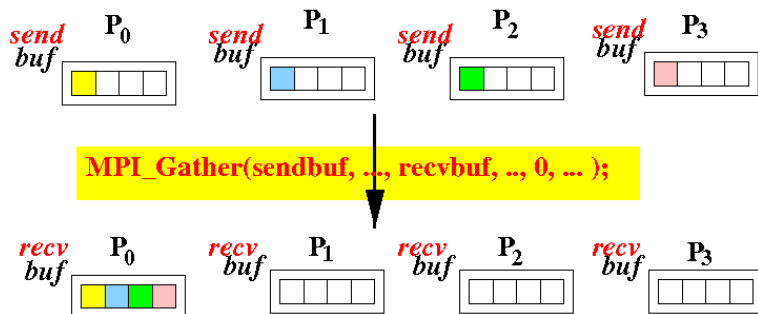
Take Care: Pair exchange can hang



```
{  
  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 a partner that doesn't exist
- ▶ Program never terminates

Gather Preview



Source: Shun Yan Cheung Notes on MPI

- ▶ Every processor has computed columns
- ▶ One processor (usually `procid 0`) needs to gather all of the data for printing / saving
- ▶ Everyone calls `MPI_Gather()`

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**
- ▶ 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 Send / Receive

BLOCKING	NON-BLOCKING
<code>MPI_Send(...)</code>	<code>MPI_Isend(..., &request)</code>
<code>MPI_Recv(...)</code>	<code>MPI_Irecv(..., &request)</code>
<code>SYNCRONIZE</code>	<code>MPI_Wait(&request, ...)</code>

- ▶ 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/rcv completes
- ▶ Prior to blocking, unsafe to alter/use data in buffers
- ▶ Can pair `MPI_Isend()` / `MPI_Irecv()` and vice versa

Faux Example of MPI_Isend() / MPI_IRecv()

```
1  int data_a[100] = {...};
2  int data_b[100] = {...};
3  int partner = ...;
4  int tag = ...;
5
6  ...; // compute data_a[]
7
8  MPI_Request request;
9  MPI_Isend(data_a, 100, MPI_INT, partner, tag, MPI_COMM_WORLD, &request);
10
11 ...; // unsafe to alter data_a[] so compute data_b[]
12
13 MPI_Wait(&request, MPI_STATUS_IGNORE); // block until data_a[] has been sent
14                                         // now safe to alter data_a[]
15 for(int i=0; i<100; i++){              // more computations on data_a[]
16     data_a[i]++;
17 }
18 ...;
19 MPI_Irecv(data_a, 100, MPI_INT, partner, tag, MPI_COMM_WORLD, &request);
20
21 ...; // unsafe to do anything with data_a[], compute on data_b[]
22
23 MPI_Wait(&request, MPI_STATUS_IGNORE); // block until data_a[] has been sent
24                                         // now safe to alter data_a[]
25 for(int i=0; i<100; i++){              // more computations on data_a[]
26     data_a[i]++;
27 }
```