MPI Basics

Chris Kauffman

Last Updated:
Thu Feb 3 09:45:08 AM CST 2022

Logistics

Reading: Grama Ch 6 + 4

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

Assignments

- A1 Due Soon
  - On-time submission Wed 2/2
  - Late Submission Fri 2/4
- **Questions?**
- A2 up later this week: MPI Programming

Today

- Primitives for Distributed Memory Computing
- MPI Programming

Next Week

- Comm Patterns
- Thu 2/10: Mini-Exam 1

Generic Send and Receive

Minimum required functionality to do distributed memory parallel computing:

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

Sample Use

```
P0                       P1

a = 100;                 receive(&a, 1, 0)
send(&a, 1, 1);          printf("%d\n", a);
a=0;
```

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

More typical appearance

Will typically write this as a single program which every processor runs.

```
void exchange(){
  int a = 100;
  int my_proc = get_processor_number();
  if(my_proc == 0){
    send(&a, 1, 1);
    a=0;
  }
  else if(my_proc == 1){
    receive(&a, 1, 0);
    printf("%d\n", a);
  }
}
```

- Function to identify proc number
- Branching on proc number to take different actions

Flavors Send/Receive

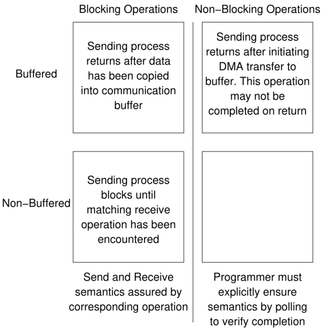
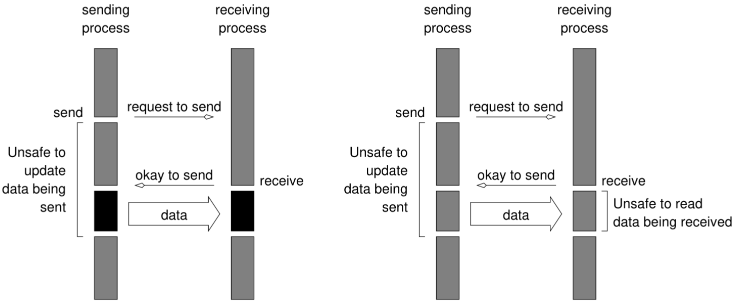


Figure 6.3 Space of possible protocols for send and receive operations.

- Hardware/OS support for buffered communication tends to make things run faster
- Usually have function calls available to do `send()` (blocking) and `send_nonblocking()` but must have some hardware support for it

Blocked and Unbuffered



(a) Without hardware support (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.

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

Ordering of Send Receive

```
P0                     P1

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

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

Blocking with Buffers

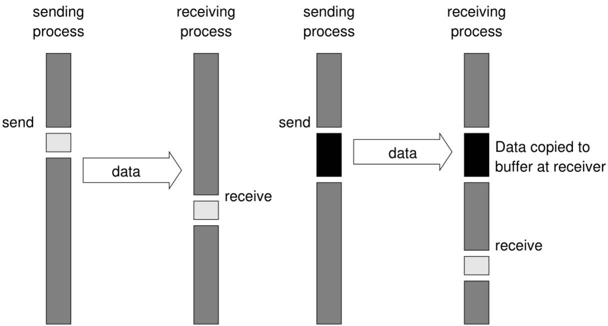


Figure 6.2 Blocking buffered transfer protocols: (a) in the presence of communication hardware with buffers at send and receive ends; and (b) in the absence of communication hardware, sender interrupts receiver and deposits data in buffer at receiver end.

Hardware buffer support, sender and receiver have a memory minion

No buffer support: sender interrupts receiver

The Danger Continues

```
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?

Non-blocking Communication

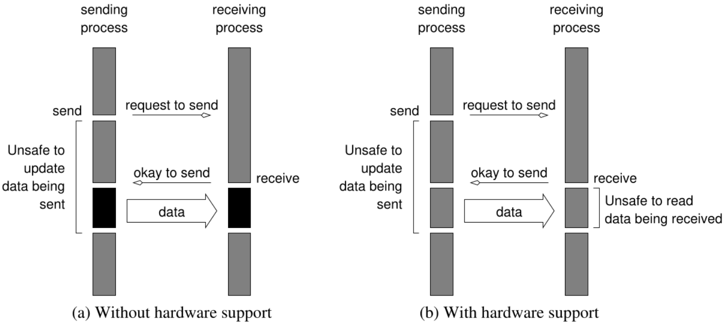
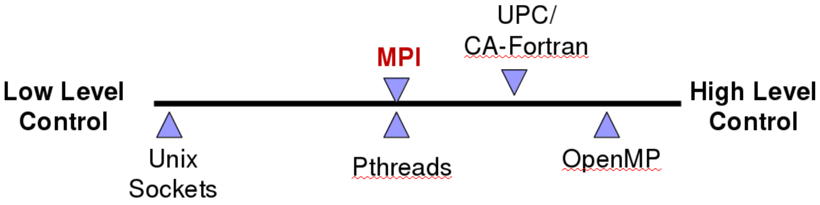


Figure 6.4 Non-blocking non-buffered send and receive operations (a) in absence of communication hardware; (b) in presence of communication hardware.

- 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
- Open source implementations: MPICH, Open MPI
- Proprietary: Intel, Platform, IBM, Platform, Cray
- Typically geared for particular architecture
- May exploit specifics of a particular machine



MPI In a Nutshell: 6 Essential Functions

```c
// Initializes MPI.
int MPI_Init(int *argc, char ***argv) ;

// Terminates MPI.
int MPI_Finalize() ;

// Determines the number of processes.
int MPI_Comm_size(MPI_Comm comm, int *size);

// Determines the label of the calling process.
int MPI_Comm_rank(MPI_Comm comm, int *rank);

// 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);
```

MPI Setup: Hello World

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

int main (int argc, char *argv[]){
  int rank, size;
  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
  for(int i=0; i<1; i++){
    printf("Hello world from process %d of %d\n",
           rank, size);
  }
  MPI_Finalize();
  return 0;
}
```

- Note the use of `MPI_COMM_WORLD` which is a predefined constant corresponding to all processors.
- Can also set up other communicators that correspond to subsets of processors

Compilation and Running

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

```
lila [test-code]% mpicc -o hello hello.c

lila [test-code]% ./hello
Hello world from process 0 of 1

lila [test-code]% mpirun hello
Hello world from process 0 of 4
Hello world from process 1 of 4
Hello world from process 2 of 4
Hello world from process 3 of 4

lila [test-code]% mpirun -np 2 hello
Hello world from process 0 of 2
Hello world from process 1 of 2

lila [test-code]% mpirun -np 8 hello
Hello world from process 7 of 8
Hello world from process 0 of 8
Hello world from process 2 of 8
Hello world from process 3 of 8
Hello world from process 4 of 8
Hello world from process 6 of 8
Hello world from process 1 of 8
Hello world from process 5 of 8
```

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 CSE Labs machines with MPI are not all 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

Warning Suppression in OpenMPI

```
csel-plate02 [examples]% mpirun -np 4 ./mpi_hello
--------------------------------------------------------------------------
[[8230,1],1]: A high-performance Open MPI point-to-point messaging module
was unable to find any relevant network interfaces:

Module: OpenFabrics (openib)
  Host: csel-plate02

Another transport will be used instead, although this may result in
lower performance.

NOTE: You can disable this warning by setting the MCA parameter
btl_base_warn_component_unused to 0.
--------------------------------------------------------------------------
P 0: Hello world from process 0 of 4 (host: csel-plate02)
Hello from the root processor 0 of 4 (host: csel-plate02)
P 1: Hello world from process 1 of 4 (host: csel-plate02)
P 2: Hello world from process 2 of 4 (host: csel-plate02)
P 3: Hello world from process 3 of 4 (host: csel-plate02)
[csel-plate02:235926] 3 more processes have sent help message help-mpi-btl-base.txt / btl:n
[csel-plate02:235926] Set MCA parameter "orte_base_help_aggregate" to 0 to see all help / e
csel-plate02 [examples]% mpirun --btl_base_warn_component_unused 0 -np 4 ./mpi_hello
mpirun: Error: unknown option "--btl_base_warn_component_unused"
Type 'mpirun --help' for usage.

#                                 vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
csel-plate02 [examples]% mpirun --mca btl_base_warn_component_unused 0 -np 4 ./mpi_hello
P 0: Hello world from process 0 of 4 (host: csel-plate02)
Hello from the root processor 0 of 4 (host: csel-plate02)
P 1: Hello world from process 1 of 4 (host: csel-plate02)
P 2: Hello world from process 2 of 4 (host: csel-plate02)
```

MPI Hostfile

Default OpenMPI config is to use all processors on a single machine then start failing

```
val [04-mpi-code]% mpirun -np 2 ./mpi_hello_plus
P 0: Hello world from process 0 of 2 (host: val)
Hello from the root processor 0 of 2 (host: val)
P 1: Hello world from process 1 of 2 (host: val)

val [04-mpi-code]% mpirun -np 8 ./mpi_hello_plus
--------------------------------------------------------------------------
There are not enough slots available in the system to satisfy the 8
slots that were requested by the application:

  ./mpi_hello_plus
```

On some systems, like our lab machines, will can use a `hostfile.txt` which gives additional machines to harness - true distributed computation

```
csel-plate02 [examples]% mpirun -np 400 --hostfile hostfile-plate-ip.txt ./mpi_hello_plus
P18: Hello world from process 18 of 400 (host: csel-plate02)
P21: Hello world from process 21 of 400 (host: csel-plate02)
P141:Hello world from process 141 of 400 (host: csel-plate03)
P310:Hello world from process 310 of 400 (host: csel-plate04)
P149:Hello world from process 149 of 400 (host: csel-plate03)
...
```

MPI Systems on CSE Labs

3 primary systems we will use for coding assignments

- PLATE
  - `ssh csel-plate02.cselabs.umn.edu`
  - 3 nodes with LOTS 72 cores per, 288 HW threads
  - Likely use for MPI coding
- VEGGIE
  - `ssh csel-carrot.cselabs.umn.edu`
  - 5 nodes with multiple Cores + GPU
- PHI
  - `ssh phi01.cselabs.umn.edu`
  - Older hardware soon to retire
  - Several cores per node, 8-10 nodes

MPI Send and Recieve

```
int a[10], b[10];
int partner = 1;
...

// Send contents of a to partner proc with tag=1
MPI_Send(a, 10, MPI_INT, partner, 1, MPI_COMM_WORLD);

// Receive message into b from partner proc with tag=1,
// ignore status of receipt
MPI_Recv(b, 10, MPI_INT, partner, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE)
```

- 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
  MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD, MPI_STATUS
}
```

- 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, alt `MPI_ANY_TAG`

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);
```

- Type of buffer is always untyped (`void* buf`)
- To try to get at 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?

```c
// 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;
  }
}
```

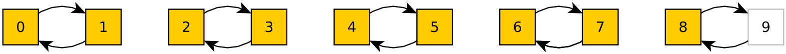
Some Patterns that occur in the problem

- 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);
}
```

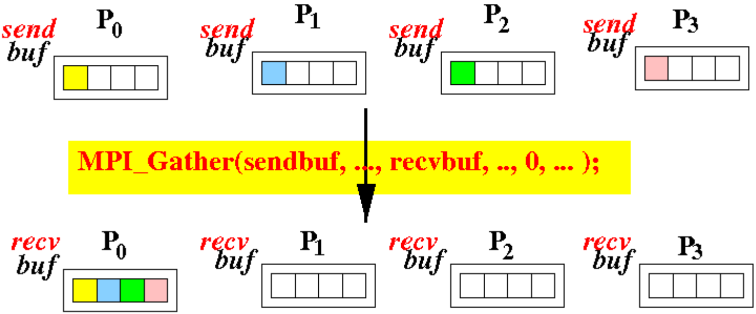
Take Care: Pair exchange can hang




```
{
  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
- Everyone calls `MPI_Gather()`

MPI_Gather Sample

Use of Gather

```c
// Preamble for any code
MPI_Comm comm = MPI_COMM_WORLD;
int sendarray[100];
int procid, total_procs, *rbuf;
...;
// Only proc 0 needs space for all
// 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

```c
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 and allows MPI implementation to exploit network as much as possible

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];
  // 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