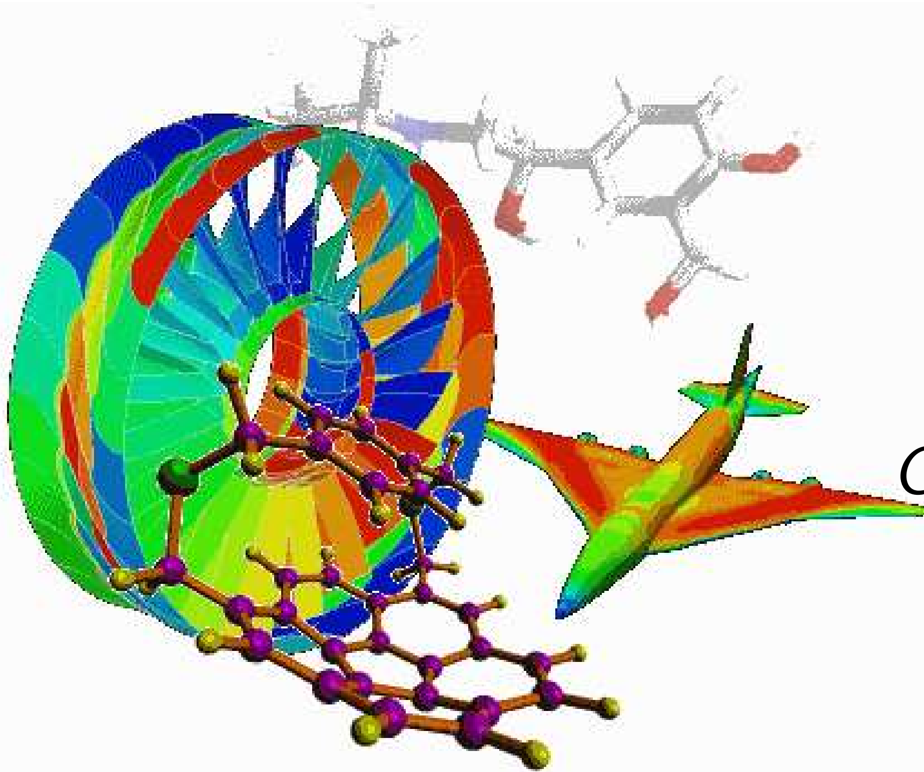
Introduction to MPI

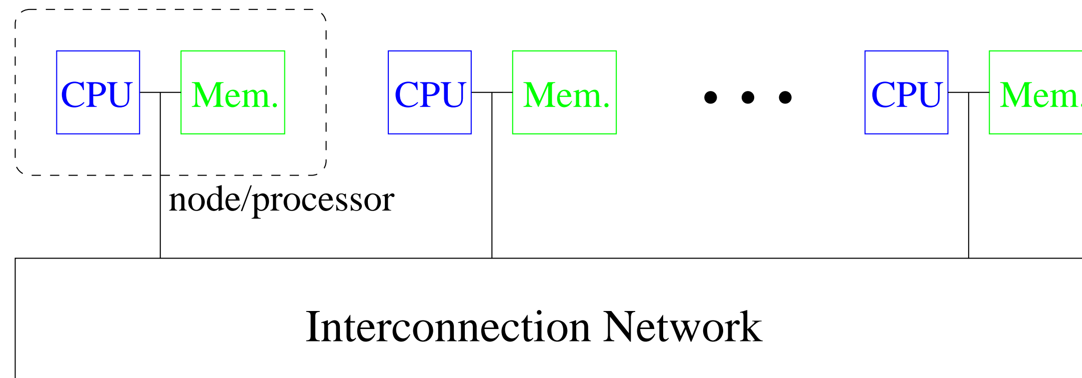


CME342 / AA220 / CS238
Lecture 3
April 4, 2005

Outline

- MPI.
- Some simple codes in FORTRAN, C, C++.
- Initialization and simple communicators.
- Compiling on `junior.stanford.edu`
- Pt-to-pt communication.
- Collective communication.
- Derived datatypes.
- Communicators & topology.
- Examples.

Message Passing Programming



- Each processor has its own private memory and address space.
- The processors communicate with one another through network.
- Ideally, each node is directly connected to every other node $\longrightarrow$ too expensive to build.
- A compromise is to use crossbar switches connecting the processors.
- Use simple topology: e.g. linear array, ring, mesh, hypercube.
- Comm. time is the bottleneck of message passing programming.

Message Passing Programs

- Separate processors.
- Separate address spaces.
- Processors execute independently and concurrently.
- Processors transfer data cooperatively.
- Single Program Multiple Data (SPMD)
  - ▷ All processors are executing the same program, but act on different data.
- Multiple Program Multiple Data (MPMD)
  - ▷ Each processors may be executing a different program.
- Common software tools: PVM, MPI.

What is MPI?

- Message-passing library specification (IEEE Standard)
  - ▷ Message-passing model
  - ▷ Not a compiler specification
  - ▷ Not a specific product
- For parallel computers, clusters & heterogeneous networks.
- Designed to permit the development of parallel software libraries.
- Designed to provide access to advanced parallel hardware for:
  - ▷ End users.
  - ▷ Library writers.
  - ▷ Tool developers.

Who designed MPI?

- Broad group of participants.
- Vendors:
  IBM, Intel, TMC, Meiko, Cray, Convex, nCube.
- Library developers:
  PVM, p4, Zipcode, TCGMSG, Chameleon, Express, Linda.
- Application specialists and consultants.
  - ▷ Companies: ARCO, KAI, NAG, Parasoft, Shell, ...
  - ▷ Labs: ANL, LANL, LLNL, ORNL, SNL, ...
  - ▷ Universities: almost 20.

Why use MPI?

- Standardization:
  The only message passing library which can be considered a standard.
- Portability:
  There is no need to modify the source when porting codes from one platform to another.
- Performance:
  Vendor implementations should be able to exploit native hardware to optimize performance.
- Availability:
  A variety of implementations are available, both vendor and public domain, e.g. MPICH implemented by ANL.
- Functionality:
  It provides around 200 subroutine/function calls.

Features of MPI

- General
  - Communicators combine context and group for message security.
  - Thread safety.

- Point-to-point communication:
  - Structured buffers and derived datatypes, heterogeneity.
  - Modes: standard, synchronous, ready (to allow access to fast protocols), buffered.

- Collective communication:
  - Both built-in & user-defined collective operations.
  - Large number of data movement routines.
  - Subgroup defined directly or by topology.

Is MPI Large or Small?

- MPI is large – around 200 functions
  Extensive functionality requires many functions/subroutines.

- MPI is small – 6 functions:
  - ▷ `MPI_Init`: Initialize MPI.
  - ▷ `MPI_Comm_size`: Find out how many processes there are.
  - ▷ `MPI_Comm_rank`: Find out which process I am.
  - ▷ `MPI_Send`: Send a message.
  - ▷ `MPI_Recv`: Receive a message.
  - ▷ `MPI_Finalize`: Terminate MPI.

- MPI is just right
  - ▷ One can use its flexibility when it is required.
  - ▷ One need not master all parts of MPI to use it.

Examples of Large Scale Codes. TFLO2000 and FLO107-MB

- MPI Standard has around 200 functions/subroutines for just about anything that you may think of.
- FLO107-MB: A parallel, multiblock, Navier-Stokes flow solver for external aerodynamic configurations uses around 20-25 MPI SUBROUTINES.
- TFLO2000: Main ASC Turbomachinery code uses around 35 MPI SUBROUTINES.
- TFLO2000 uses some of the advanced features of inter-communicators and, to deal with sliding mesh interfaces, it requires more barriers and different types of communication

- Normal codes are likely to use only a *few* MPI calls.

Example 1: Hello, world!

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

int main(argc, argv)
int argc;
char **argv;
{
  MPI_Init(&argc, &argv);
  printf("Hello, world!\n");
  MPI_Finalize();
  return 0;
}
```

- `#include "mpi.h"` provides basic MPI definitions and types.
- `MPI_Init` starts MPI.
- `MPI_Finalize` exits MPI.
- Note that all non-MPI routines are local; thus `printf` runs on each process.

Example 2: "Advanced" Hello, world!

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

int main(argc, argv)
int argc;
char **argv;
{
  int rank, size;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  printf("Hello, world! I am %d of %d\n", rank, size);
  MPI_Finalize();
  return 0;
}
```

- `MPI_Comm_rank` determines the proc id (0 to $nproc - 1$).
- `MPI_Comm_size` determines # of procs.
- Note: for some parallel systems, only a few designated procs can do I/O. MPI-1.2 Standard defines API for parallel I/O (extensions in MPI-2.0).
- What does the output look like?

Compiling MPI Program on `junior.stanford.edu`

You have two main options:

1. SUN optimized MPI implementation.
   - `/opt/SUNWhpc/bin/mpf90 -o mycode mycode.f -lmpi`
   - `/opt/SUNWhpc/bin/mpfcc -o mycode mycode.c -lmpi`
   - `mpcc`, `mpCC`, `mpf77`, `mpf90`, `mpf95` are available.
   - `/opt/SUNWhpc/bin/mprun -np 4 mycode`

2. ANL MPICH publicly available implementation.
   - `/afs/ir/class/cs238/mpich/bin/mpif90 -o mycode mycode.f`
   - `/afs/ir/class/cs238/mpich/bin/mpicc -o mycode mycode.f`
   - `mpicc`, `mpiCC`, `mpif77`, `mpif90` are available.

- `/afs/ir/class/cs238/mpich/bin/mpirun -np 4 mycode`

SUN's implementation is best because it has been optimized for that particular computer and can achieve higher bandwidths and lower latencies.

SUN's implementation depends on additional software running on the machine. If it fails, use MPICH.

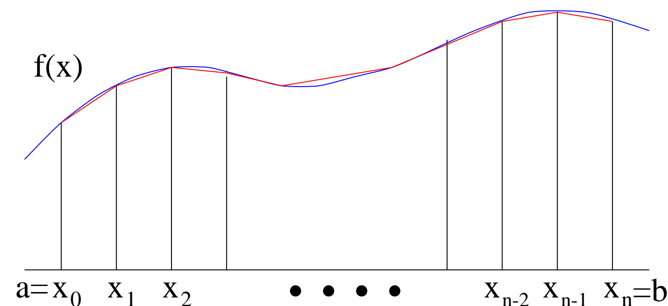
MPICH should always run properly but will have lower performance than SUN's version. For development purposes it is just fine. For benchmarking purposes, you will have to use the version from SUN.

Example: Calculate π

- Well-known formula:

$$\int_0^1 \frac{4}{1+x^2}\,dx = \pi.$$

- Numerical integration (Trapezoidal rule):



$$\int_a^b f(x)\,dx \approx h[\frac{1}{2}f(x_0) + f(x_1) + \cdots + f(x_{n-1}) + \frac{1}{2}f(x_n)].$$

$$x_i = a + ih, \quad h = (b-a)/n, \quad n = \#\text{ of subintervals.}$$

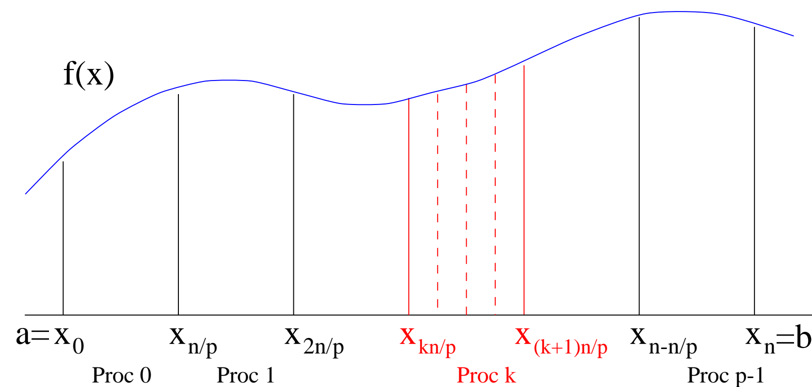
Example: Calculate π (cont.)

- A sequential function `Trap(a,b,n)` approx the integral from a to b of $f(x)$ using trap. rule with n subintervals:

```
double Trap(a,b,n) {

  h = (b-a)/n;
  integral = (f(a)+f(b))/2;
  for (i=1; i<=n-1; i++) {
    x = a+i*h;
    integral = integral+f(x);
  }
  integral = h*integral;
  return integral;
}
```

Parallelizing `Trap`



- Divide the interval $[a, b]$ into p equal subintervals.
- Each proc calculates the local approx. integral using trap. rule simultaneously.
- Finally, combine the local values to obtain the total integral.

Parallel Trap Program

```
/* Start up MPI */
MPI_Init(&argc, &argv);

/* Find out how many procs */
MPI_Comm_size(MPI_COMM_WORLD, &p);

/* Determine my proc id */
MPI_Comm_rank(MPI_COMM_WORLD, &my_id);

/* Apply Trap rule locally */
n = 128; a=0; b=1;
h = (b-a)/n;
nk = n/p;
ak = a+my_id*nk*h;
bk = ak+nk*h;
integral = Trap(ak,bk,nk);
```

Parallel Trap Program (cont.)

```
/* Sum up integrals */
if (my_id == 0) {
  total = integral;
  for (k=1; k<p; k++) {
    MPI_Recv(&integral, 1, MPI_DOUBLE, k,
             tag, MPI_COMM_WORLD,&status);
    total = total+integral;
  }
}
else {
  MPI_Send(&integral, 1, MPI_DOUBLE, 0,
           tag, MPI_COMM_WORLD);
}
if (mi_id == 0) {
  printf(``Estimated value of pi = %f'',total);
/* Close MPI */
```

```
MPI_Finalize();
```

- Can replace `MPI_Send` and `MPI_Recv` by `MPI_Reduce`.
- Embarrasingly parallel – no comm. needed during the computations of the local approx. integrals.

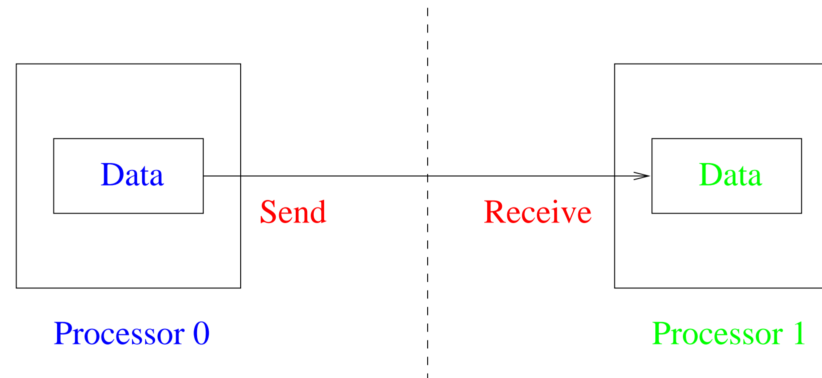
Timing

- MPI_Wtime() returns the wall-clock time.

```
double start, finish, time;

MPI_Barrier(MPI_COMM_WORLD);
start = MPI_Wtime();
      :
      :
MPI_Barrier(MPI_COMM_WORLD);
finish = MPI_Wtime();
time = finish - start;
```

Send & Receive



- Cooperative data transfer
- To (from) whom is data sent (received)?
- What is sent?
- How does the receiver identify it?

Message Passing: Send

```
MPI_Send(address, count, datatype, dest, tag, comm)
```

- `(address, count)` = a contiguous area in memory containing the message to be sent.
- `datatype` = Type of data, e.g. integer, double precision.
  ▷ message length = count*sizeof(datatype).
- `dest` = integer identifier representing the processor to send the message to.
- `tag` = nonnegative integer that the destination can use to selectively screen messages.
- `comm` = communicator = group of processors.

Message Passing: Receive

```
MPI_Recv(address, count, datatype, source, tag, comm, status)
```

- `(address, count)` = a contiguous area in memory reserved for the message to be received.
- `datatype` = Type of data, e.g. integer, double precision.
- `source` = integer identifier representing the processor that sent the message.
- `tag` = nonnegative integer that the destination can use to selectively screen messages.
- `comm` = communicator = group of processors.
- `status` = information about the message that is received.

Single Program Multiple Data (SPMD)

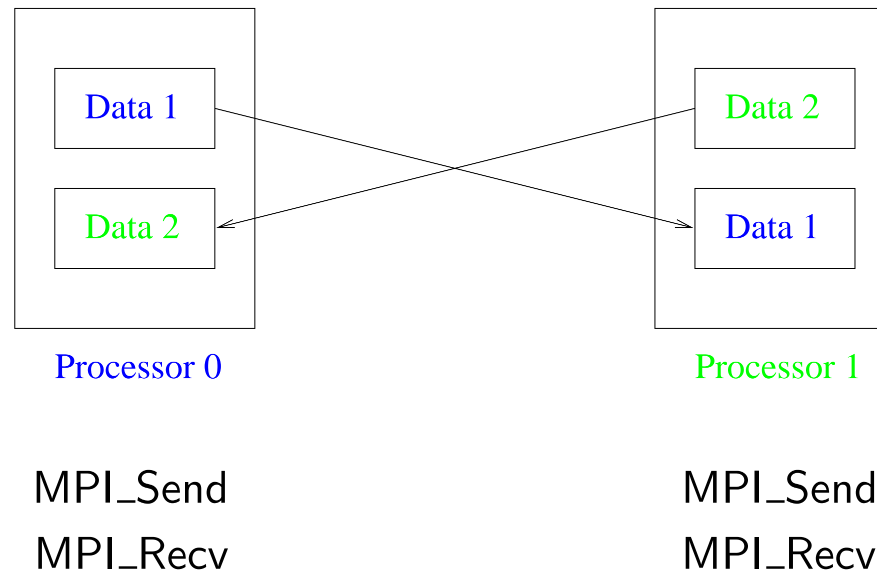
- Proc 0 & Proc 1 are actually performing different operations.
- However, not necessary to write separate programs for each processor.
- Typically, use conditional statement and proc id to define the job of each processor:

```
int a[10];

if (my_id == 0)
   MPI_Send (a,10,MPI_INT,1,0,MPI_COMM_WORLD);
else if (my_id == 1)
   MPI_Recv (a,10,MPI_INT,0,0,MPI_COMM_WORLD);
endif
```

Deadlock

- Example: exchange data between 2 procs:



- `MPI_Send` is a synchronous operation. If no system buffering, it keeps waiting until a matching receive is posted.

- Both processors are waiting for each other → deadlock.
- However, OK if system buffering exists → unsafe program.
- Note: `MPI_Recv` is blocking and nonbuffered.
- A real deadlock:

Proc 0	Proc 1
MPI_Recv	MPI_Recv
MPI_Send	MPI_Send

- Fix by reordering comm.:

Proc 0	Proc 1
MPI_Send	MPI_Recv
MPI_Recv	MPI_Send

Buffered / Nonbuffered Comm.

- No-buffering (*phone calls*)
  - ▷ Proc 0 initiates the send request and rings Proc 1. It waits until Proc 1 is ready to receive. The transmission starts.
  - ▷ Synchronous comm. – completed only when the message was received by the receiving proc.
- Buffering (*beeper*)
  - ▷ The message to be sent (by Proc 0) is copied to a system-controlled block of memory (buffer).
  - ▷ Proc 0 can continue executing the rest of its program.
  - ▷ When Proc 1 is ready to receive the message, the system copies the buffered message to Proc 1.
  - ▷ Asynchronous comm. – may be completed even though the receiving proc has not received the message.

Buffered Comm. (cont.)

- Buffering requires system resources, e.g. memory, and can be slower if the receiving proc is ready at the time of requesting the send.
- Application buffer: address space that holds the data.
- System buffer: system space for storing messages. In buffered comm., data in application buffer is copied to/from system buffer.
- MPI allows comm. in buffered mode:
  `MPI_Bsend, MPI_Ibsend`.
- User allocates the buffer by:
  `MPI_Buffer_attach(buffer, buffer_size)`
- Free the buffer by `MPI_Buffer_detach`.

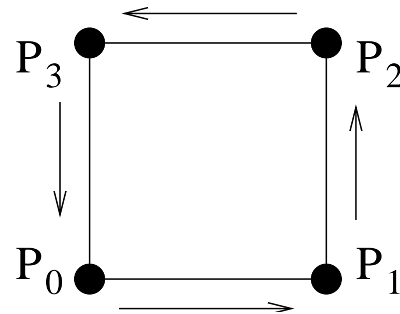
Blocking / Nonblocking Comm.

- Blocking Comm. (*McDonald's*)
  - ▷ The receiving proc has to wait if the message is not ready.
  - ▷ Different from synchronous comm.
  - ▷ Proc 0 may have already buffered the message to system and Proc 1 is ready, but the interconnection network is busy.
- Nonblocking Comm. (*In & Out*)
  - ▷ Proc 1 checks with the system if the message has arrived yet. If not, it continues doing other stuff. Otherwise, get the message from the system.
- Useful when computation and comm. can be performed at the same time.
- MPI allows both nonblocking send & receive:

`MPI_Isend`, `MPI_Irecv`.

- In nonblocking send, program identifies an area in memory to serve as a send buffer. Processing continues immediately without waiting for message to be copied out from application buffer.
- The program should not modify the application buffer until the nonblocking send has completed.
- Nonblocking comm. can combined with nonbuffering: `MPI_Issend`, or buffering: `MPI_Ibsend`.
- Use `MPI_Wait` or `MPI_Test` to determine if the nonblocking send or receive has completed.

Example: Data Exchange in a Ring Topology



- Blocking version:

```
for (i=0; i<p; i++) {
  send_offset = ((my_id-i+p) % p)*blksize;
  recv_offset = ((my_id-i-1+p) % p)*blksize;
  MPI_Send(y+send_offset, blksize, MPI_FLOAT,
           my_id+1, 0, ring_comm);
  MPI_Recv(y+recv_offset, blksize, MPI_FLOAT,
           my_id-1, 0, ring_comm, &status);
}
```

- Nonblocking version:

```
send_offset = my_id*blksize;
recv_offset = (my_id-1+p)*blksize;
for (i=0; i<p-1; i++) {
  MPI_Isend(y+send_offset, blksize, MPI_FLOAT,
            my_id+1, 0, ring_comm, &send_request);
  MPI_Irecv(y+recv_offset, blksize, MPI_FLOAT,
            my_id-1, 0, ring_comm, &recv_request);

  send_offset = ((my_id-i-1+p) % p)*blksize;
  recv_offset = ((my_id-i-2+p) % p)*blksize;

  MPI_Wait(&send_request, &status);
  MPI_Wait(&recv_request, &status);
}
```

- The comm. & computations of next offsets are overlapped.

Summary: Comm. Modes

- 4 comm. modes in MPI: standard, buffered, synchronous, ready. They can be either blocking or nonblocking.

- In standard modes (`MPI_Send, MPI_Recv, ...`), it is up to the system to decide whether messages should be buffered.

- In synch. mode, a send won't complete until a matching receive has been posted which has begun reception of the data.
  - ▷ `MPI_Ssend, MPI_Issend`.
  - ▷ No system buffering.

- In buffered mode, the completion of a send does not depend on the existence of a matching receive.

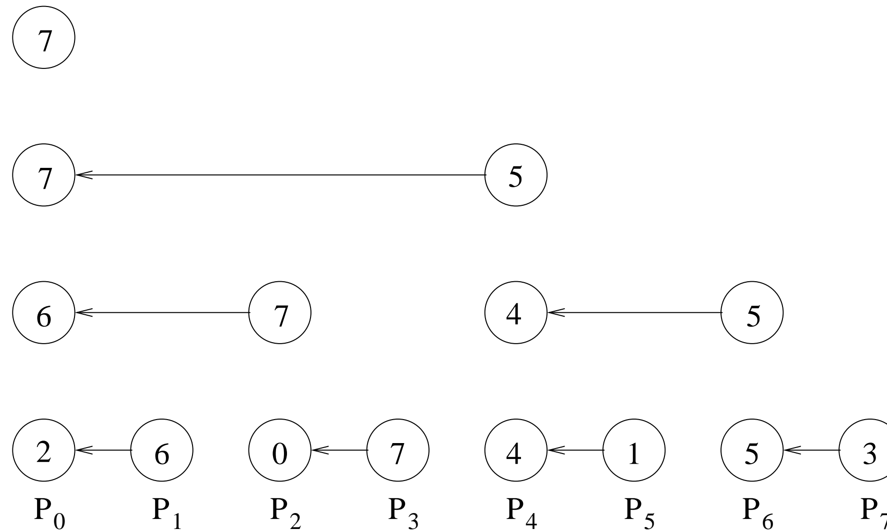
- ▷ `MPI_Bsend, MPI_Ibsend`.
- ▷ System buffering by `MPI_Buffer_attach` & `MPI_Buffer_detach`.

- Ready mode not discussed.

Collective Communications

- Comm. pattern involving all the procs; usually more than 2.
- `MPI_Barrier`: synchronize all procs.
- Broadcast (`MPI_Bcast`)
  - ▷ A single proc sends the same data to every proc.
- Reduction (`MPI_Reduce`)
  - ▷ All the procs contribute data that is combined using a binary operation.
  - ▷ Example: max, min, sum, etc.
  - ▷ One proc obtains the final answer.
- Allreduce (`MPI_Allreduce`)
  - ▷ Same as `MPI_Reduce` but every proc contains the final answer.
  - ▷ Effectively as `MPI_Reduce + MPI_Bcast`, but more efficient.

An Implementation



- Tree-structured communication:(find the max among procs)
- Only needs $\log_2 p$ stages of comm.
- Not necessary optimum on a particular architecture.

Other Collective Comm.

- Scatter (`MPI_Scatter`)
  - ▷ Split the data on proc *root* into p segments.
  - ▷ The 1st segement is sent to proc 0, the 2nd to proc 1, etc.
  - ▷ Similar to but more general than `MPI_Bcast`.
- Gather (`MPI_Gather`)
  - ▷ Collect the data from each proc and store the data on proc *root*.
  - ▷ Similar to but more general than `MPI_Reduce`.
- Can collect and store the data on *all* procs using `MPI_Allgather`.

Comparisons of Collective Comms.