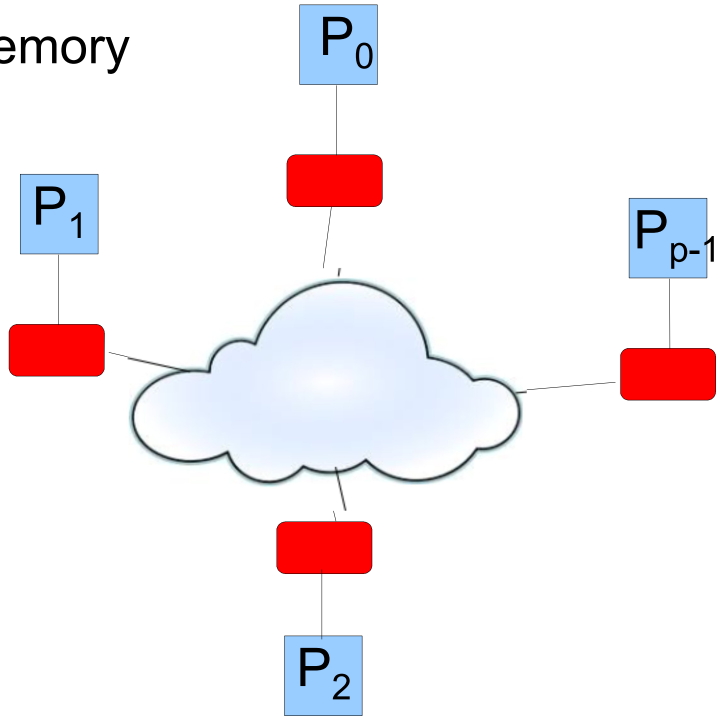
Parallel Computing Using Message Passing

Distributed Memory Computers

Single Program Multiple Data (SPMD)

- Each processor can only see and access its own memory
- Send and receive messages for accessing non-local memory
- Use process numbers for ID



API: Message Passing Interface (MPI)

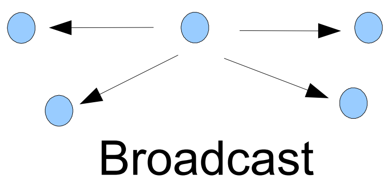
- Library routines called from C, C++, or Fortran

Compared to shared memory:
+ Easier to build large systems
+ More control (but also responsibility) for the programmer
+ Dedicated effort gives better performance

- Harder to program
- Cost of developing code is higher
- Systems (typically) have higher latency

Message Passing Interface (MPI)

- Standardized communication library (since 1994)
- Supports point-to-point as well as collective communication



- Gives portable code
- Available on most computers, (Open-MPI and MPICH)
- Can be run on networked workstations as well as on shared memory computers!

A Typical MPI Program

- Process 0 reads the data
- Distributes data to the other processes
- Each process sets up local data structures
- Performs local computations interlaced with communication
- Gather result on process 0
- Process 0 writes to disk or prints to screen

Main Challenges

- How to partition data between processors?
  - Want equal load between processors
  - How to partition unstructured data?

- Setting up distributed data structures
  - Must change from global to local numbering
  - Must know where different data is stored

- Larger systems/less tightly coupled → Higher latency than shared memory
  - Minimize number of communication operations
  - Overlap communication with computation

- No automatic synchronization, must be specified by the programmer

A Minimal MPI Program

hello.c:

```
#include <stdio.h>
#include <mpi.h>                        // MPI header file

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

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

  printf( "Hello world from process %d of %d\n", rank, size );

  MPI_Finalize();                       // End MPI, called by every processor
  return 0;
}
```

Compile with

`mpicc hello.c`   (can also use `gcc hello.c -lmpi`)

Run with

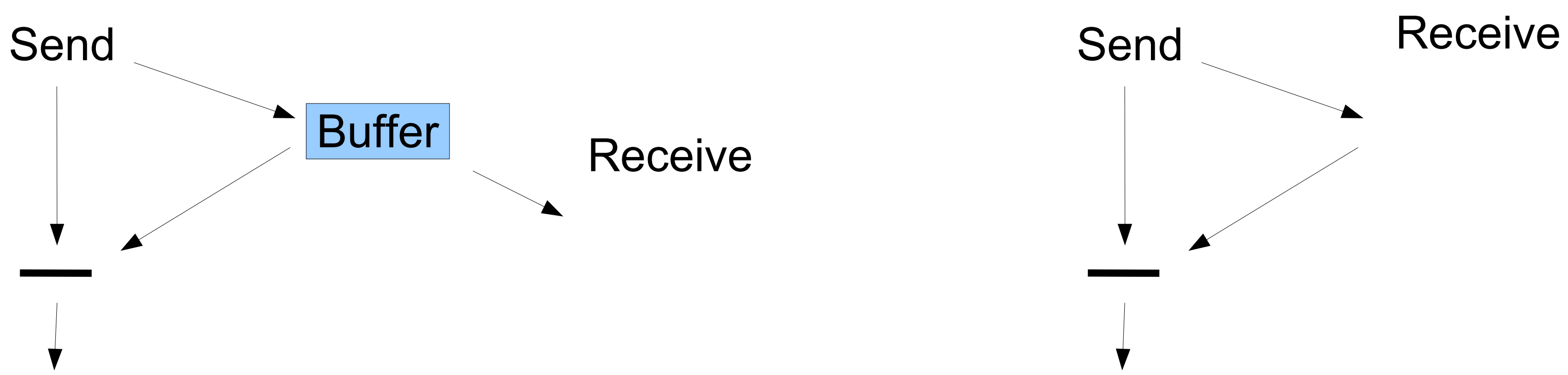
`mpirun -np 2 a.out`

Point-to-point Communication

Two main modes: blocking or non-blocking

Differs in ease of use and in speed

Blocking: Send does not return until sending variable is either sent or copied to a safe buffer. Which mode is used can depend on size of message. Receive operation returns when all data has been received.



Sender is free to reuse sending variable (or array) immediately after return from Send.

Receiver can use data immediately after return from Receive operation

`MPI_Send(....)`   `MPI_Recv(....)`

Blocking Send and Receive Routines

```
int MPI_Send(buf, count, type, dest, tag, comm);
```

`buf`: address of data to be sent
`count`: Number of entries
`type`: Type of data, `MPI_INT`, `MPI_FLOAT`, `MPI_CHAR` etc.
`dest`: ID of recipient process, in the range 0 through p-1.
`tag`: User defined tag, use a number to identify the message
`comm`: Communicator: "ID of communication space", `MPI_COMM_WORLD`

returns error code

```
int MPI_Recv(buf, count, type, source, tag, comm, &status);
```

`MPI_Status status;` Object with information:
- Where did the message come from
- What is the tag
- Errors...

Wildcards: `MPI_ANY_TAG` Receive message with any tag
`MPI_ANY_SOURCE` Receive message from any source

Communication from processor 0 to every other processor

com.c

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

int main(argc,argv)
int argc;
char **argv;
{
  int myrank;
  int np;
  int x, i;
  MPI_Status status;

  MPI_Init(&argc,&argv);                  // Start MPI, get myrank and np
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &np);
  printf("I am %d out of %d processors \n",myrank,np);

  if (myrank == 0) {
    x = 11;
    for(i=1;i<np;i++)                     // Note, does not send to itself
      MPI_Send(&x,1,MPI_INT,i,1,MPI_COMM_WORLD); }  // Send one integer
  else {
    MPI_Recv(&x,1,MPI_INT,0,1,MPI_COMM_WORLD,&status);
    printf("Processor %d got %d \n",myrank,x); }

  MPI_Finalize();
  return(0);
}
```

Communication in a ring starting from processor 0

ring.c

```
int main(argc,argv)
int argc;
char **argv;
{
  int myrank, np;
  int x;
  MPI_Status status;

  MPI_Init(&argc,&argv);           // Set up MPI
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &np);
  printf("I am %d out of %d processors \n",myrank,np);

  if (myrank == 0) {  // Proc 0 sends to 1, then receives from last proc.
    x = 11;
    MPI_Send(&x,1,MPI_INT,(myrank+1)% np,1,MPI_COMM_WORLD);
    MPI_Recv(&x,1,MPI_INT,(myrank-1+np)% np,1,MPI_COMM_WORLD,&status);
    printf("Processor %d got %d from %d \n",myrank,x,(myrank-1)% np);
  }
  else {              // Receive from previous and send to next
    MPI_Recv(&x,1,MPI_INT,(myrank-1+np)% np,1,MPI_COMM_WORLD,&status);
    printf("Processor %d got %d from %d \n",myrank,x,(myrank-1)%np);
    MPI_Send(&x,1,MPI_INT,(myrank+1)% np,1,MPI_COMM_WORLD);
  }

  MPI_Finalize();
  return(0);
}
```

Documentation

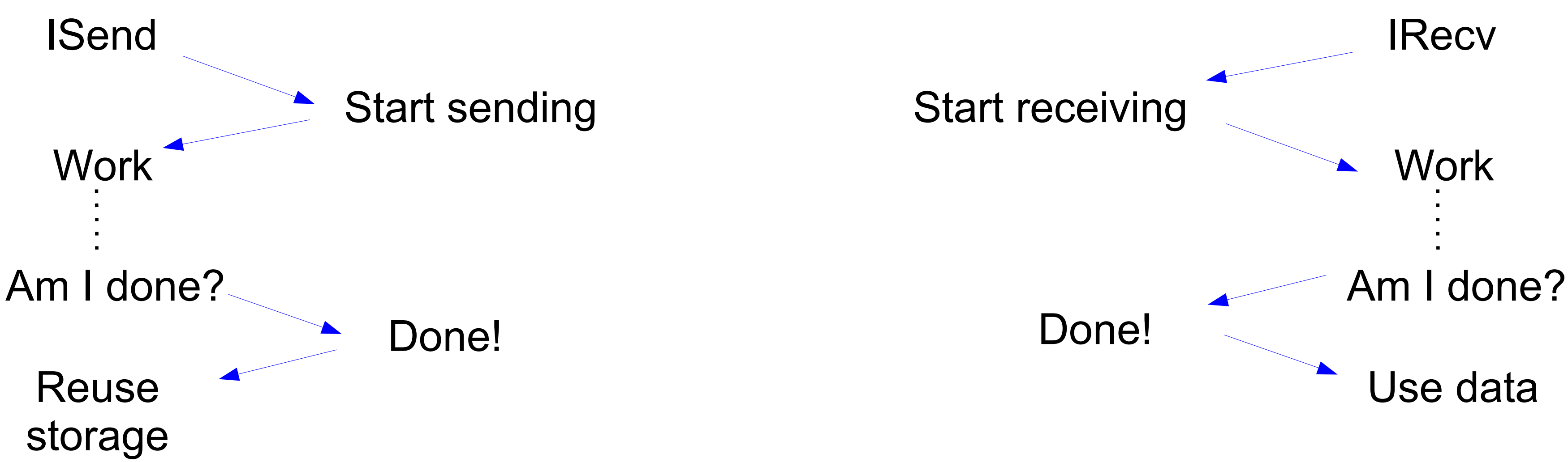
https://www.mpi-forum.org/ Standardization forum, contains documentation

https://www.open-mpi.org/ Open source MPI implementation, used in gcc

Also check videos for MPI tutorials on youtube.

Non-Blocking Communication

Non blocking: Send and receive routines returns immediately even if operation is not complete. Must check before writing/reading data.



- Can overlap sending and receiving with other work (if there is hardware support).
- Important for speed as communication is *much* slower than computation.
- Morale: Start Isend and Irecv as early as possible!

Use of MPI_Isend and MPI_Irecv

icom.c

```
int main(argc,argv)   // Note, only works when run with 2 processes
int argc;
char **argv;
{
  int myrank, np;
  int x;
  MPI_Status status;
  MPI_Request req;  // Variable used to hold status of communication operation

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &np);
  printf("I am %d out of %d processors \n",myrank,np);

  if (myrank == 0) {
    x = 11;
    MPI_Isend(&x,1,MPI_INT,1,1,MPI_COMM_WORLD,&req);
    MPI_Wait(&req,&status);  // Waiting for send to finish
  }
  else {
    MPI_Irecv(&x,1,MPI_INT,0,1,MPI_COMM_WORLD,&req);
    MPI_Wait(&req,&status);  // Waiting for receive to finish
    printf("Processor %d got %d \n",myrank,x);
  }

  MPI_Finalize();
  return(0);
}
```

Collective Operations

Collective operations can be implemented more efficiently (and conveniently) by the system

Examples: One-to-all, all-to-one, all-to-all

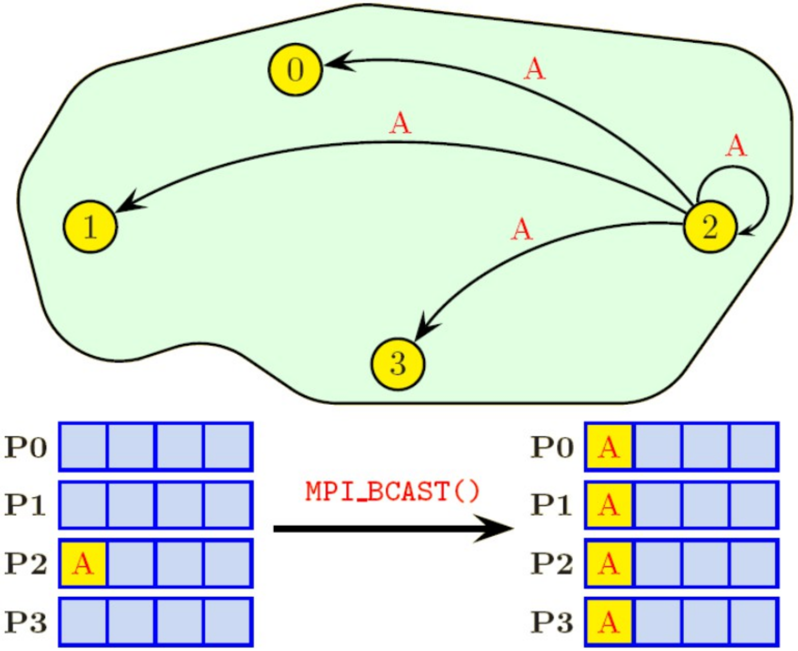
Broadcast:
`MPI_Bcast(data,size, type, source, comm)`

bcast.c:

```
.
.
if (myrank == 2) {
  x = 11;
}

MPI_Bcast(&x,1,MPI_INT,2,MPI_COMM_WORLD);

printf("Processor %d got %d \n",myrank,x);
```



Scattering and Gathering of Data

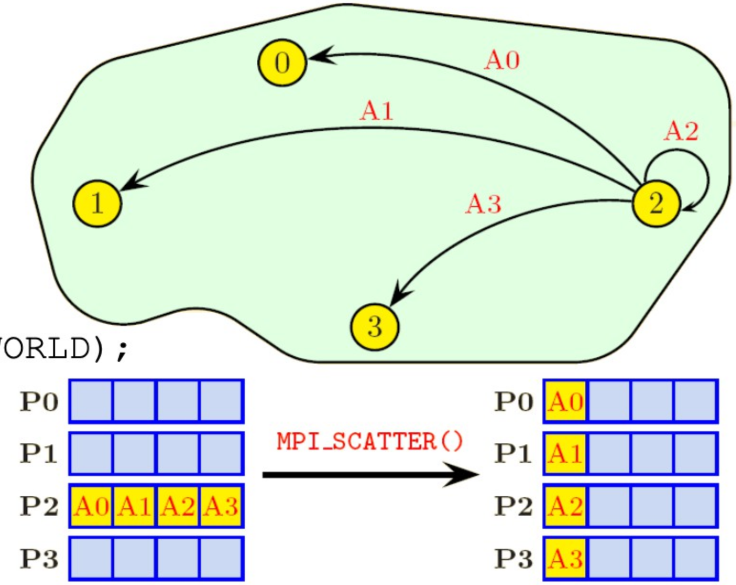
Scatter:
`MPI_Scatter(sendbuf,sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)`

Content of `sendbuf` is scattered to the processors, with `sendcount` consecutive elements per processor.
`root` is the ID of the sender.

```
.
.
if (myrank == 2) {
  for(i=0;i<nproc;i++)
    x[i] = i;
}

MPI_Scatter(x,1,MPI_INT,&y,1,MPI_INT,2,MPI_COMM_WORLD);

printf("Processor %d got %d \n",myrank,y);
```



Gather:
`MPI_Gather()` is the inverse operation of scatter.

Other Collective Operations

`MPI_Reduce()` : Combine values to one processor (sum, product, max)

reduce.c

```
MPI_Reduce(&myrank,&x1,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
MPI_Reduce(&myrank,&x2,1,MPI_INT,MPI_MAX,0,MPI_COMM_WORLD);

if (myrank == 0) {
  printf("Sum on processor %d: %d, should be %d \n",myrank,x1,(np-1)*np/2);
  printf("Max on processor %d: %d, should be %d \n",myrank,x2,np-1);
}
```

`MPI_Alltoall()`: Send individual message from every processor to every processor

How things can go wrong...

P_0:

```
MPI_Send(to proc 1,tag=1)
MPI_Send(to proc 1,tag=0)
```

P_1:

```
MPI_Recv(from proc 0, tag=0)
MPI_Recv(from proc 0, tag=1)
```

P_0:

```
MPI_Recv(from proc 1)
MPI_Send(to proc 1)
```

P_1:

```
MPI_Recv(from proc 0)
MPI_Send(to proc 0)
```

All processors execute:

```
for i=0,p-1
   MPI_Send(to proc i)

for i=0,p-1
   MPI_Recv(from proc i)
```

All processors execute:

```
if (myid != 0) {
   ...
   MPI_Barrier()
}
```

Measuring Time

`MPI_Wtime()`: Returns current time as a double value, used for determining how much time is being spent in different parts of the code.

```
start = MPI_Wtime();
….                    // Do some work
end = MPI_Wtime();
elapsed = end - start;
```

To perform accurate timing make sure all processors start together:

```
MPI_Barrier(MPI_COMM_WORLD);  // Wait for everyone
start = ...
…
end = …
```

Also good for debugging!
- Check that every processor gets to a certain point before moving on.

Programs to test: com_bcast.c

Evaluating Parallel Programs

t_p : Time to execute program on p processors

$t_p = f(n,p)$: Depends on both problem size (n) and number of processors (p)

$t_p = t_{comp} + t_{comm}$: Time for computation + time for communication

Time to send a message containing w bytes:

$$t_{comm} = \underset{\text{(latency)}}{t_{startup}} + \underset{\text{(bandwidth)}}{w\, t_{data}}$$

Typically $t_{startup} >> t_{data}$

How does latency and bandwidth compare?

$t_{comm} = \quad t_{startup} + w\, t_{data}$

For a short message $t_{startup}$ will dominate
run time2.c

For a long message $w\, t_{data}$ will dominate
run time1.c

Compare with time needed to perform one Flop
run band.c