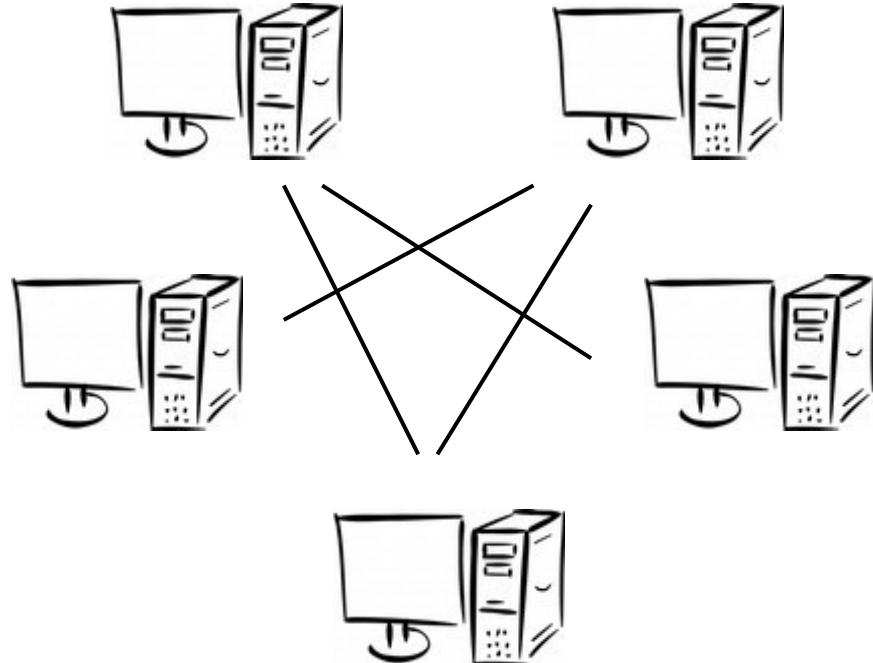


# CS 470

## Spring 2017

Mike Lam, Professor



## Distributed Programming & MPI

# MPI paradigm

- Single program, multiple data (**SPMD**)
  - One program, multiple **processes (ranks)**
  - Processes communicate via **messages**
    - An MPI *message* is a collection of fixed-size data elements
    - Underlying mechanism (e.g., sockets) is implementation-dependent
  - Multiple processes may run on the same node
    - They do NOT share an address space!
    - But intra-node communication will be faster than inter-node
  - Processes are grouped into **communicators**
    - May be in multiple communicators simultaneously
    - Default communicator: **`MPI_COMM_WORLD`** (all processes)

# Message-Passing Interface (MPI)

- MPI is a standardized library interface
  - Available online: <http://www.mpi-forum.org/docs/>
  - MPI-1 released in 1994 after Supercomputing '93
  - MPI-2 (1996) added one-sided operations and parallel I/O
  - MPI-3 (2012) improved non-blocking and one-sided operations
    - Also added tooling interface
  - Latest version (MPI-3.1) approved June 2015
  - Working groups currently designing MPI-4.0
- Several widely-used implementations
  - OpenMPI (on our cluster)
  - MPICH / MVAPICH / MVAPICH2 (higher performance)

# MPI-3.1 support

## Status of MPI-3.1 Implementations

	MPICH	MVAPICH	Open MPI	Cray MPI	Tianhe MPI	Intel MPI	IBM BG/Q MPI <sup>1</sup>	IBM PE MPICH <sup>2</sup>	IBM Platform	SGI MPI	Fujitsu MPI	MS MPI	MPC	NEC MPI
NBC	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Nbrhood collectives	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✓
RMA	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	Q2'17	✓
Shared memory	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	*	✓
Tools Interface	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	* Q4'16	✓
Comm-creat group	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	*	✓
F08 Bindings	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓	✗	✗	✗	✓
New Datatypes	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓
Large Counts	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	Q2'16	✓
Matched Probe	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	Q2'16	✓
NBC I/O	✓	Q3'16	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	Q4'16	✓

Source: <http://mpi-forum.org/mpi31-impl-status-Jun16.pdf>

# MPI development

- MPI is more than a library (unlike pthreads)
  - Compiler wrapper (`mpicc` / `mpiccc` / `mpif77`)
    - Still need to `#include <mpi.h>`
  - Program launcher (`mpirun`)
  - Job management integration (`salloc` / `sbatch`)
    - SLURM *tasks* = MPI *ranks / processes*
- System admins use **modules** to ease setup
  - Command: `module load mpi`
  - Populates your shell environment w/ MPI paths
  - **Spack** is a user-mode system for installing new software and modules
    - Must install Spack and set it up first (see cluster guide on website)
    - To use MPICH (faster!): `spack install mpich` / `spack load mpich`

# MPI conventions

- Identifiers start with “MPI\_”
  - Also, first letter following underscore is uppercase
- MPI must be initialized and cleaned up
  - `MPI_Init` and `MPI_Finalize`
  - For `MPI_Init`, you can “pass through” argc and argv
  - No MPI calls before `MPI_Init` or after `MPI_Finalize`!
- Task parallelism is based on rank / process ID
  - `MPI_Comm_rank` and `MPI_Comm_size`
  - Usually, rank 0 is special (the "master" process)
- I/O is asymmetrical
  - All ranks may write to `stdout` (or `stderr`)
  - Usually, only rank 0 can read `stdin`

# Basic MPI functions

```
int MPI_Init (int *argc, char ***argv)  
int MPI_Finalize ()  
int MPI_Comm_size (MPI_Comm comm, int *size)  
int MPI_Comm_rank (MPI_Comm comm, int *rank)  
  
double MPI_Wtime ()  
int MPI_Barrier (MPI_Comm comm)
```

# MPI Hello World

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

int main(int argc, char **argv)
{
    int mpi_rank;
    int mpi_size;

    MPI_Init(&argc,&argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);

    printf("Hello from process %2d / %d (%s)!\n", mpi_rank+1, mpi_size);

    MPI_Finalize();

    return 0;
}
```

# Point-to-point messages

```
int MPI_Send (void *buf, int count, MPI_Datatype dtype,  
              int dest, int tag, MPI_Comm comm)  
  
int MPI_Recv (void *buf, int count, MPI_Datatype dtype,  
              int src,   int tag, MPI_Comm comm, MPI_Status *status)
```

# Point-to-point messages

```
int MPI_Send (void *buf, int count, MPI_Datatype dtype,  
              int dest, int tag, MPI_Comm comm)  
  
int MPI_Recv (void *buf, int count, MPI_Datatype dtype,  
              int src, int tag, MPI_Comm comm, MPI_Status *status)
```

Annotations for MPI\_Send:

- dest: must correspond
- tag: must match
- count: recv count must be equal to or higher than send count
- dtype: must match

Annotations for MPI\_Recv:

- src: must correspond
- tag: must match

# MPI Hello World 2

```
if (mpi_rank != 0) {  
  
    /* Create message and send to process 0 */  
  
    MPI_Send(message, strlen(message)+1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);  
  
} else {  
  
    /* Process 0: Print message then receive & print messages from other  
     * processes */  
  
    printf("Local: %s\n", message);  
    for (int i = 1; i < mpi_size; i++) {  
        MPI_Recv(message, MAX_MSG_SIZE, MPI_CHAR, i, 0, MPI_COMM_WORLD,  
                 MPI_STATUS_IGNORE);  
        printf("Recv: %s\n", message);  
    }  
}  
  
int MPI_Send  (void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm)  
int MPI_Recv  (void *buf, int count, MPI_Datatype dtype, int src,  int tag, MPI_Comm comm,  
              MPI_Status *status)
```

# MPI datatypes

C data type	MPI data type
char	MPI_CHAR
int	MPI_INT
long	MPI_LONG
long long	MPI_LONG_LONG
float	MPI_FLOAT
double	MPI_DOUBLE

# Generic receiving

- All parameters are required for `MPI_Send`
- `MPI_Recv` allows for some ambiguity
  - count is the *maximum* count (actual could be lower)
  - src can be `MPI_ANY_SOURCE` and tag can be `MPI_ANY_TAG`
- The status parameter provides this info
  - Pointer to `MPI_Status` struct that is populated by `MPI_Recv`
  - After receive, access members `MPI_SOURCE` and `MPI_TAG`
  - Use `MPI_Get_count` to calculate true count
  - If you don't need any of these, pass `MPI_IGNORE_STATUS`

**Postel's Law:** “Be conservative in what you do;  
be liberal in what you accept from others.”

# Blocking

- Exact blocking behavior is implementation-dependent
  - `MPI_Send` **may** block until the message is sent
    - Sometimes depends on the size of the message
    - `MPI_Ssend` will **always** block until the message is received
  - `MPI_Recv` will **always** block until the message is received
  - Know your implementation or be conservative
    - You can use `MPI_Ssend` to check your code
    - Use `MPI_SendRecv` if both sending and receiving

```
int MPI_Sendrecv (void *send_buf, int send_count, MPI_Datatype send_dtype, int dest, int send_tag,
                  void *recv_buf, int recv_count, MPI_Datatype recv_dtype, int src, int recv_tag,
                  MPI_Comm comm, MPI_Status *status)
```

# Non-blocking send/receive

- Some operations are guaranteed not to block
  - Point-to-point: `MPI_Isend` and `MPI_Irecv`
  - Includes some collectives (in `MPI-3`)
- These operations merely “request” some communication
  - `MPI_Request` variables can be used to track these requests
  - `MPI_Wait` blocks until an operation has finished
  - `MPI_Test` sets a flag if the operation has finished

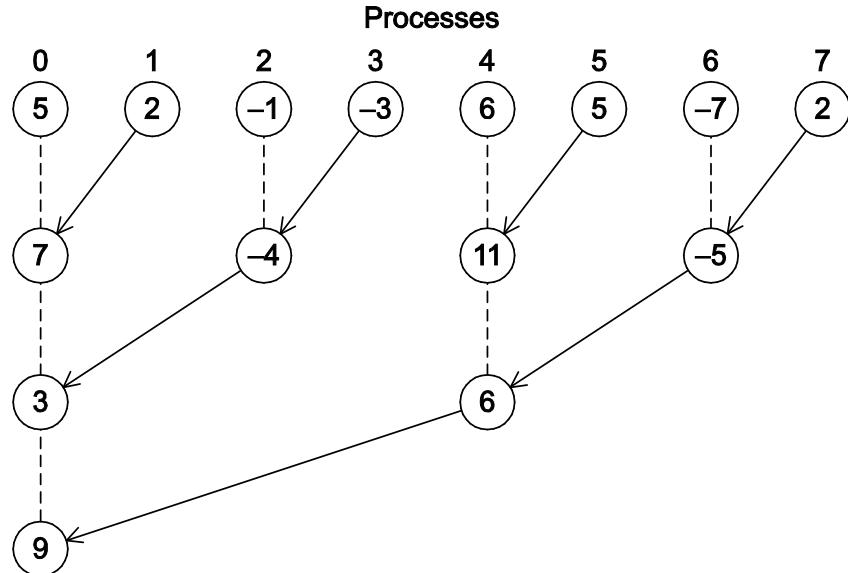
```
int MPI_Isend (void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Irecv (void *buf, int count, MPI_Datatype dtype, int src,  int tag, MPI_Comm comm, MPI_Request *request,
               MPI_Status *status)

int MPI_Wait (MPI_Request *request,          MPI_Status *status)
int MPI_Test (MPI_Request *request, int *flag, MPI_Status *status)
```

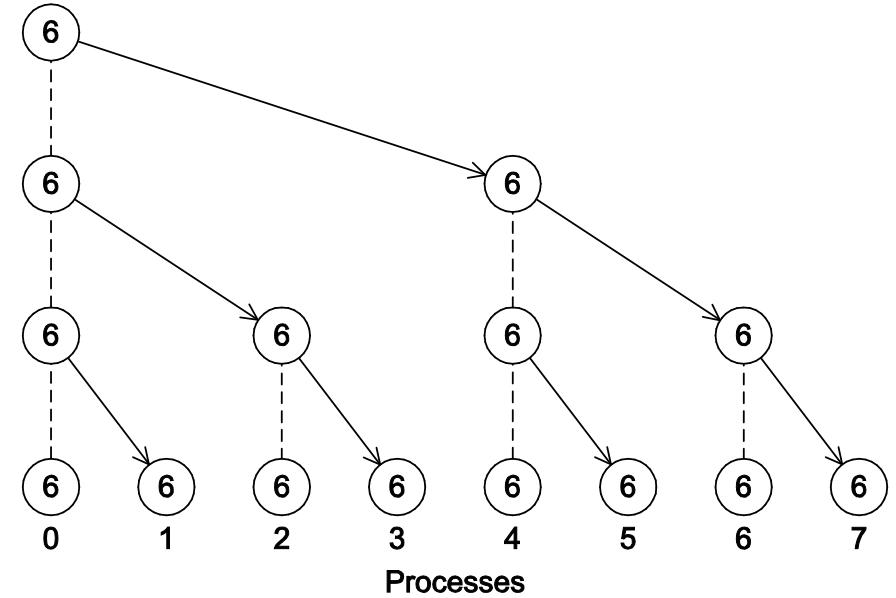
# Issues with point-to-point

- No global message order guarantees
  - Between any send/recv pair, messages are **nonovertaking**
    - If  $p_1$  sends  $m_1$  then  $m_2$  to  $p_2$ , then  $p_2$  must receive  $m_1$  first
  - No guarantees about global ordering
  - Communication between all processes can be tricky
- Process 0 reads input and collects results
  - Using point-to-point operations does not scale well
  - Need a more efficient method
- **Collective** operations provide *correct* and *efficient* built-in all-process communication

# Tree-structured communication



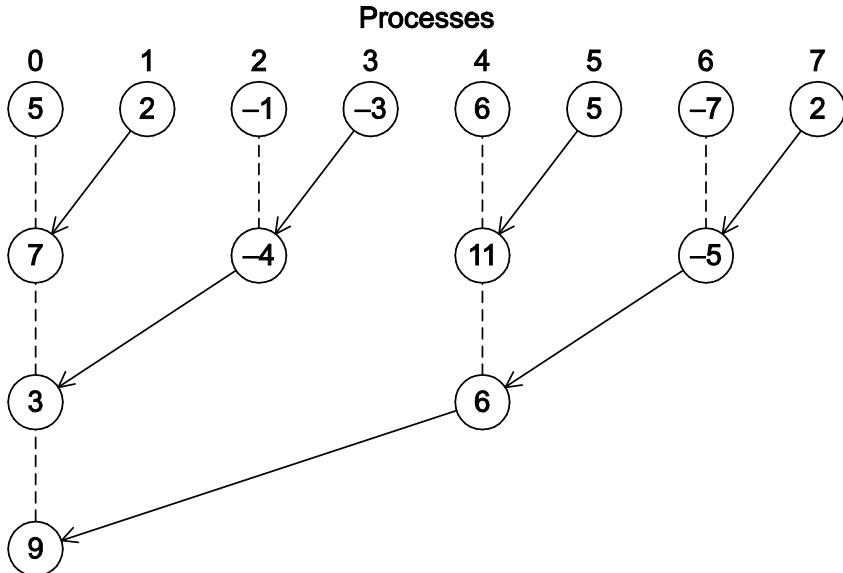
Reduction



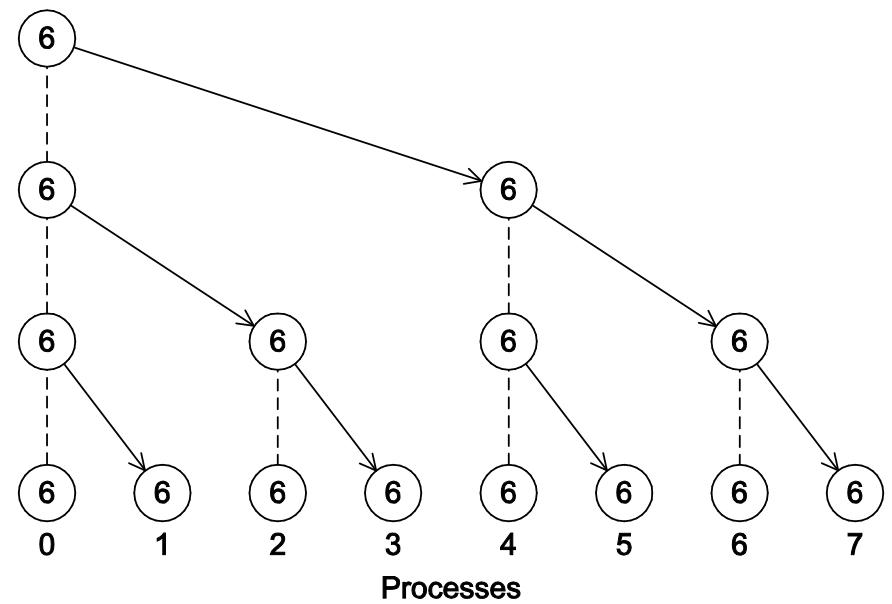
Broadcast

```
int MPI_Bcast      (void      *buf,  
                     MPI_Datatype dtype,  
                     int count,  
                     int root, MPI_Comm comm)  
  
int MPI_Reduce    (void *send_buf, void *recv_buf, int count,  
                     MPI_Datatype dtype, MPI_Op op,   int root, MPI_Comm comm)
```

# Tree-structured communication



# Reduction



# Broadcast

## int MPI\_Bcast

```
(void *buf,  
 MPI_Datatype dtype,
```

```
int count,  
int root(MBT_Comm comm)
```

## int MPI\_Reduce

`(void *send_buf, void *recv_buf`  
`MPI_Datatype dtype, MPI_Op op,`

```
int count, rank 0  
int root, MPI_Comm comm)
```

# Collective reductions

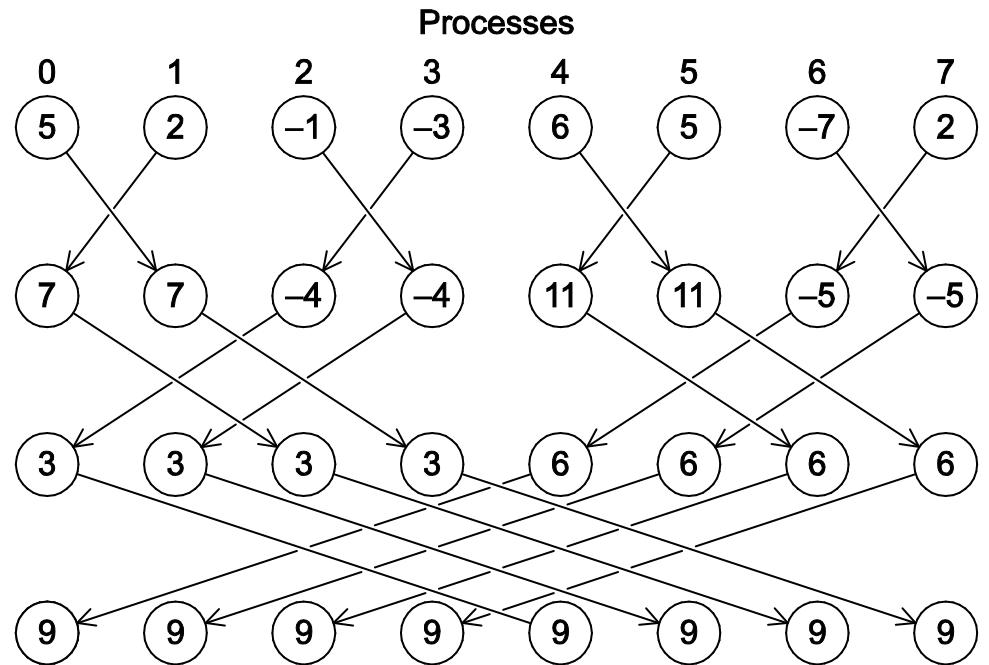
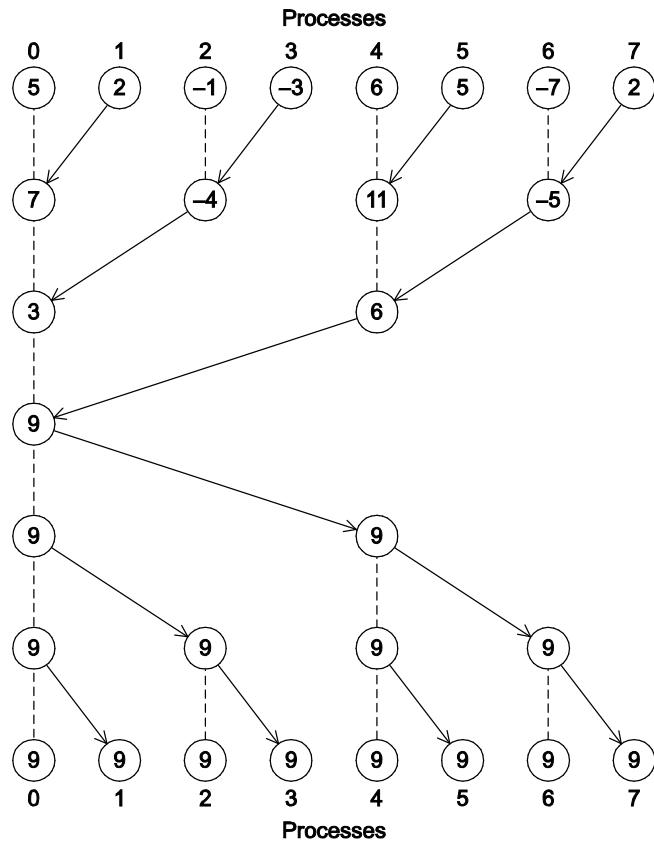
- Reduction operations
  - `MPI_SUM`, `MPI_PROD`, `MPI_MIN`, `MPI_MAX`
- Collective operations are matched based on ordering
  - Not on source / dest or tag
  - Try to keep code paths as simple as possible

Time	Process 0	Process 1	Process 2
0	<code>a = 1; c = 2</code>	<code>a = 1; c = 2</code>	<code>a = 1; c = 2</code>
1	<code>MPI_Reduce (&amp;a, &amp;b, ...)</code>	<code>MPI_Reduce (&amp;c, &amp;d, ...)</code>	<code>MPI_Reduce (&amp;a, &amp;b, ...)</code>
2	<code>MPI_Reduce (&amp;c, &amp;d, ...)</code>	<code>MPI_Reduce (&amp;a, &amp;b, ...)</code>	<code>MPI_Reduce (&amp;c, &amp;d, ...)</code>

**NOTE:** Reductions with count > 1 operate on a per-element basis

# MPI\_Allreduce

- Combination of MPI\_Reduce and MPI\_Broadcast

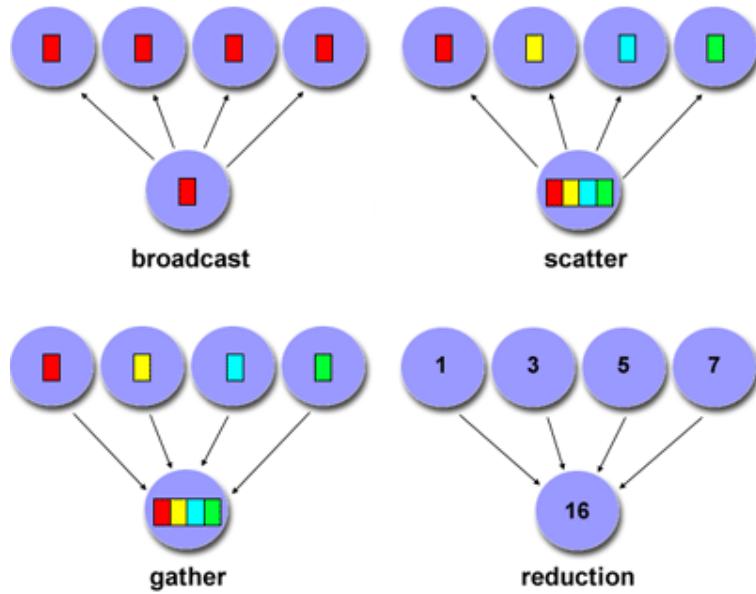


# Data distribution

- MPI\_Scatter and MPI\_Gather
  - MPI\_Allgather (gather + broadcast)
  - Provides efficient data movement
- Partitioning: block vs. cyclic
  - Usually application-dependent

Process	Components								Block-cyclic Blocksize = 2			
	Block				Cyclic							
0	0	1	2	3	0	3	6	9	0	1	6	7
1	4	5	6	7	1	4	7	10	2	3	8	9
2	8	9	10	11	2	5	8	11	4	5	10	11

# MPI collective summary



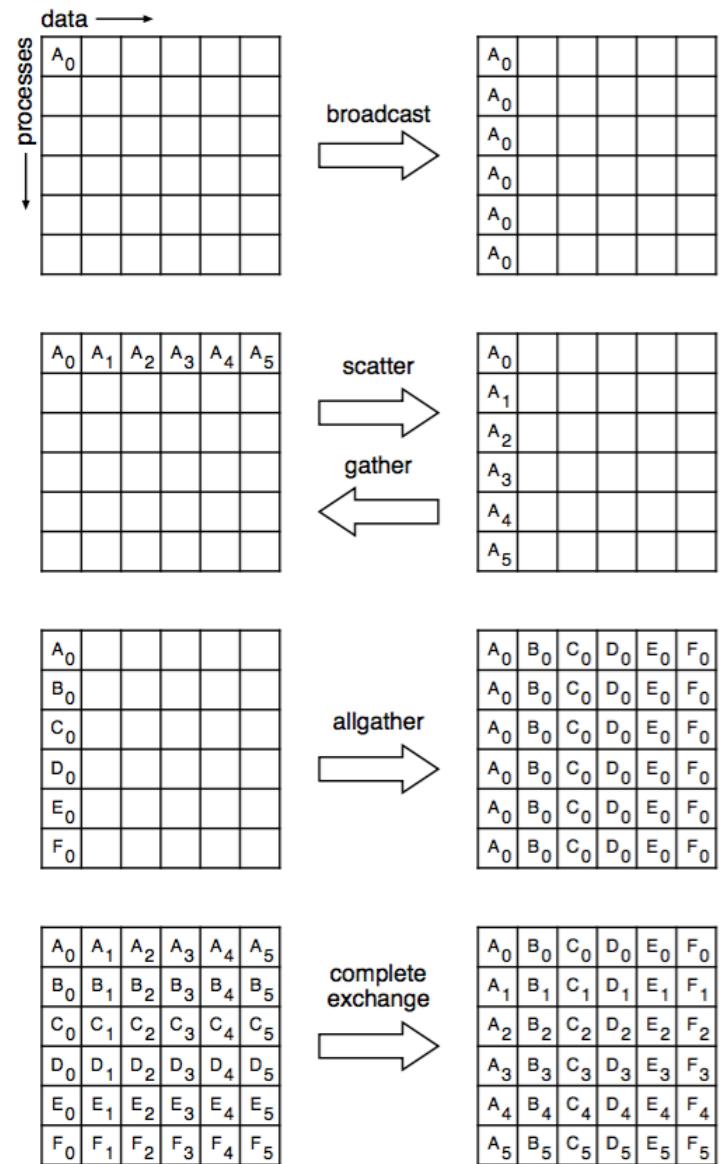
**`MPI_Bcast()`**  
**`MPI_Reduce()`**  
**`MPI_Allreduce()`**

Broadcast (one to all)  
Reduction (all to one)  
Reduction (all to all)

**`MPI_Scatter()`**  
**`MPI_Gather()`**  
**`MPI_Alltoall()`**  
**`MPI_Allgather()`**

Distribute data (one to all)  
Collect data (all to one)  
Distribute data (all to all)  
Collect data (all to all)

(these four include “\*v” variants for variable-sized data)



# MPI reference

## General

```
int MPI_Init (int *argc, char ***argv)
int MPI_Finalize ()
int MPI_BARRIER (MPI_Comm comm)
double MPI_Wtime ()
```

```
int MPI_Comm_size (MPI_Comm comm, int *size)
int MPI_Comm_rank (MPI_Comm comm, int *rank)
Default communicator: MPI_COMM_WORLD
```

## Point-to-point Operations

```
int MPI_Send (void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm)
int MPI_Ssend (void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm)
int MPI_Recv (void *buf, int count, MPI_Datatype dtype, int src, int tag, MPI_Comm comm, MPI_Status *status)
    (maximum count)           (MPI_ANY_SOURCE / MPI_ANY_TAG)           (MPI_STATUS_IGNORE)
```

```
int MPI_Isend (void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Irecv (void *buf, int count, MPI_Datatype dtype, int src, int tag, MPI_Comm comm, MPI_Request *request,
               MPI_Status *status)
```

```
int MPI_Wait (MPI_Request *request,           MPI_Status *status)
int MPI_Test (MPI_Request *request, int *flag, MPI_Status *status)
int MPI_Get_count (MPI_Status *status, MPI_Datatype dtype, int *count)
```

```
int MPI_Sendrecv (void *send_buf, int send_count, MPI_Datatype send_dtype, int dest, int send_tag
                  void *recv_buf, int recv_count, MPI_Datatype recv_dtype, int src, int recv_tag,
                  MPI_Comm comm, MPI_Status *status)
```

## Collective Operations

```
int MPI_Bcast (void      *buf,           int count, MPI_Datatype dtype,           int root, MPI_Comm comm)
```

```
int MPI_Reduce (void *send_buf, void *recv_buf, int count, MPI_Datatype dtype, MPI_Op op, int root, MPI_Comm comm)
int MPI_Allreduce (void *send_buf, void *recv_buf, int count, MPI_Datatype dtype, MPI_Op op,           MPI_Comm comm)
```

```
int MPI_Scatter (void *send_buf,           void *recv_buf,           int send_count, MPI_Datatype send_dtype,
                  int recv_count, MPI_Datatype recv_dtype,           int root, MPI_Comm comm)
```

```
int MPI_Gather (void *send_buf,           void *recv_buf,           int send_count, MPI_Datatype send_dtype,
                  int recv_count, MPI_Datatype recv_dtype,           int root, MPI_Comm comm)
```

```
int MPI_Allgather (void *send_buf,           void *recv_buf,           int send_count, MPI_Datatype send_dtype,
                     int recv_count, MPI_Datatype recv_dtype,           MPI_Comm comm)
```

```
int MPI_Alltoall (void *send_buf,           void *recv_buf,           int send_count, MPI_Datatype send_dtype,
                   int recv_count, MPI_Datatype recv_dtype,           MPI_Comm comm)
```

# Distributed memory summary

- Distributed systems can scale massively
  - Hundreds or thousands of nodes, petabytes of memory
  - Millions/billions of cores, petaflops of computation capacity
- They also have significant issues
  - Non-uniform memory access (NUMA) costs
  - Requires explicit data movement between nodes
  - More difficult debugging and optimization
- Core design tradeoff: data distribution
  - How to partition, and what to send where (duplication?)
  - Goal: minimize data movement
  - Paradigm: computation is “free” but communication is not