UNF
UNIVERSITY of NORTH FLORIDA.

Introduction to MPI

Presented By: Jonathan O'Berry, Kris Roker, and Poonkuzhali Vasudevan

What is MPI ?

UNF
UNIVERSITY of NORTH FLORIDA.

- MPI stands for Message Passing Interface
- MPI is a message-passing library specification that provides a de-facto "standard" message passing model for distributed and shared memory computers
- Designed to provide a portable parallel programming interface.

What is MPI? (cont.)

UNF
UNIVERSITY of NORTH FLORIDA.

- MPI is not endorsed by any standards organization, but it is considered the standard in industry.
  - Five versions of the MPI standard called [1]:
    - MPI-1 (1994): Provides over 115 functions and is fully implemented by all libraries.
    - MPI-2 (1996): Provides over 500 functions but is not fully implemented by all libraries.
    - MPI-3 (2014): Improves scalability and offers multicore and cluster support. Removed language bindings for C++
    - MPI-4 (2021): Introduced Partitioned Communications.
    - MPI-5: Still Under Development

Key Concepts [1]

UNF
UNIVERSITY of NORTH FLORIDA.

- Comm
- Color
- Key
- Newcomm
- Derived data types
- Point-to-point
- Collective basics
- One-sided

Where Do I Find a Version of MPI

UNF
UNIVERSITY of NORTH FLORIDA.

- Most vendors have their own implementations optimized for their architecture and communication subsystems
  - Microsoft (Microsoft MPI v10.1.13)[2]
  - Cray (Cray MPICH)[3]
  - IBM (IBM Spectrum MPI)[4]
  - Intel (Intel MPI Library)[5]
- Open-source
  - MPICH
  - OpenMPI
  - DeinoMPI[6]

MPI Languages and Compiling

UNF
UNIVERSITY of NORTH FLORIDA.

- MPI has language bindings for C and Fortran
- MPI 1 and 2 also have language bindings for C++
- Python has a package PyMPI

- To compile MPI in C MPICC must be used rather than GCC.
- MPICC is necessary because it invokes a series of complex GCC compiler and linker flags that are necessary for programs using MPI [7].

A Minimal MPI Program (C)

UNF
UNIVERSITY of NORTH FLORIDA.

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

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

Better Hello (C)

UNF
UNIVERSITY of NORTH FLORIDA.

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

int main( 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( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

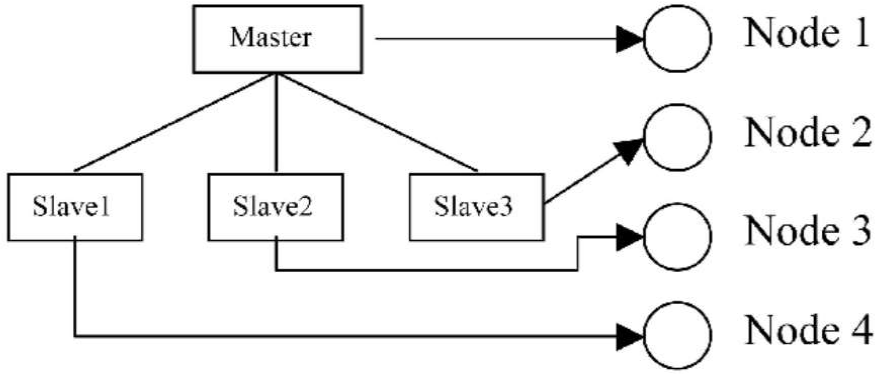
```
[n01445775@h01 ~]$ mpirun -n 20 ./bHello.out
I am 0 of 20
I am 1 of 20
I am 2 of 20
I am 4 of 20
I am 5 of 20
I am 6 of 20
I am 7 of 20
I am 8 of 20
I am 9 of 20
I am 10 of 20
I am 11 of 20
I am 12 of 20
I am 13 of 20
I am 14 of 20
I am 15 of 20
I am 16 of 20
I am 17 of 20
I am 18 of 20
I am 19 of 20
I am 3 of 20
```

Using SPMD (Single Program Multiple Data) Computational Model

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

int main (int argc, char *argv[])
{
  int myrank, mysize;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /*find process rank */
  MPI_Comm_size( MPI_COMM_WORLD, &mysize );
  if (myrank == 0) { /*Master*/
    printf( "I am %d of %d\n", myrank, mysize );
    printf("ZEEEERRRROOO\n\n\n\n");
  } else {/*Slave*/
    printf("Not Zero\n");
    printf( "I am %d of %d\n\n\n\n", myrank, mysize);
  }
  MPI_Finalize();
}
```

Master and Slave are executed by the Master and slave process respectively.



[8]

MPI Definitions of Blocking and Non-Blocking

- Blocking - return after their local actions complete, though the message transfer may not have been completed.
  - Return when "locally complete" - when location used to hold message can be used again or altered without affecting message being sent.
  - Blocking send will send message and return - does not mean that message has been received, just that process free to move on without adversely affecting message.
- Non-blocking - return immediately.

These terms may have different interpretations in other systems.

Message Buffers

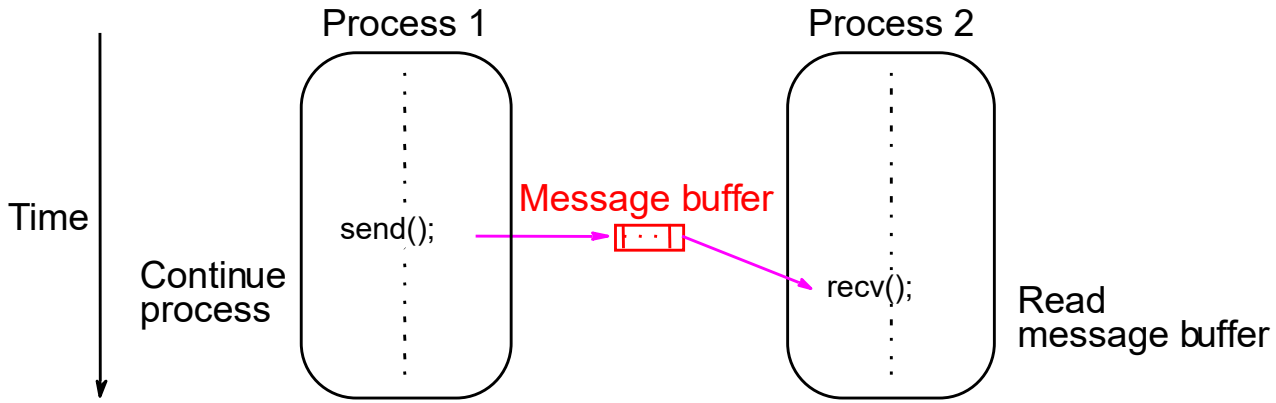
UNF
UNIVERSITY of NORTH FLORIDA.

An area of memory where data is stored in the process of being moved

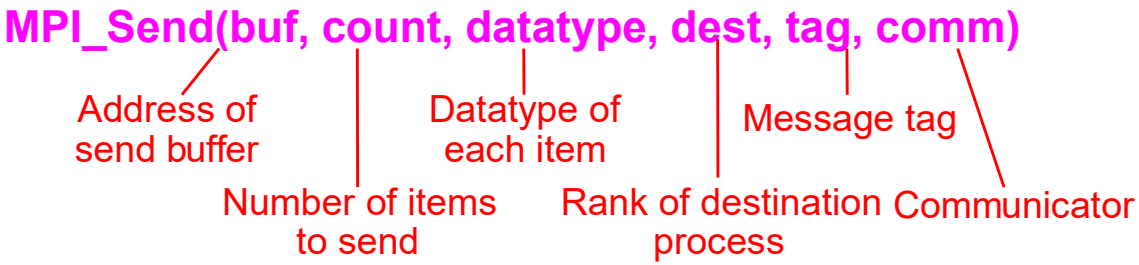
- Message will be added to buffer before sending in some modes
- User can specify a buffer to be used in sending messages
  - MPI_BUFFER_ATTACH(buffer_addr, size) to attach a defined buffer
  - MPI_BUFFER_DETACH(buffer_addr, size) to detach buffer from MPI
  - Where buffer is the address of the buffer being attached
  - Where size is the size of that buffer in bytes
- Only one buffer can be attached to a process at a time

How message-passing routines return before message transfer completed

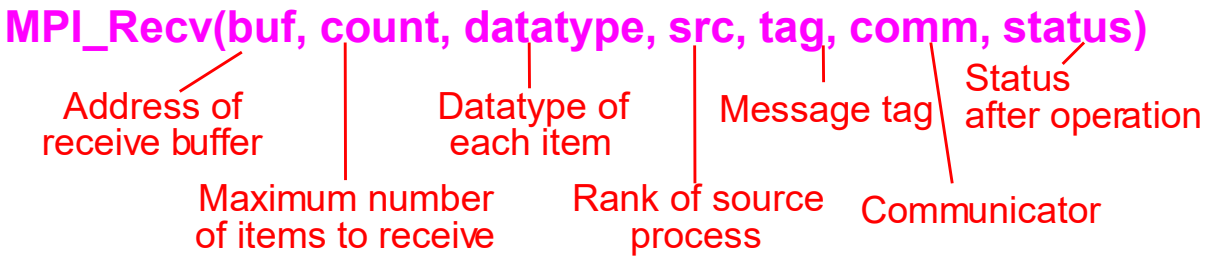
Message buffer needed between source and destination to hold message:



Parameters of Blocking Send



Parameters of Blocking Receive



Example

To send an integer from process 0 to process 1, do an operation on it, and send it back

```
int number;
if (world_rank == 0) {
    // If we are rank 0, set the number to 13 and send it to process 1
    number = 13;
    MPI_Send(&number, 1, MPI_INT, 1, 0,  MPI_COMM_WORLD);
    // Now we wait to receive 14 back from process 1
    MPI_Recv(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Process 0 received number %d from process 1\n", number);

}
else if (world_rank == 1) {
    // Listen out for the number from process 0
    MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Process 1 received number %d from process 0\n", number);
    // Add 1 to it and send it back
    number ++;
    MPI_Send(&number, 1, MPI_INT, 0, 0,  MPI_COMM_WORLD);
}
```

```
[n01233188@h01 ~]$ mpirun -n=2 ./SendReceive.out
Process 1 received number 13 from process 0
Process 0 received number 14 from process 1
```

Blocking MPI Send Modes

UNF UNIVERSITY of NORTH FLORIDA.

- MPI_Send()
  - Standard blocking send
  - Will not return until either the send buffer is available or a matching receive is posted
- Two protocols
  - Eager
    - Entire message is sent assuming the receiver can get it
    - Only used for small messages
  - Rendezvous
    - Sender asks receiver to send data, and will send when receiver has a buffer for it
    - Higher synchronization delays; can lower performance

Blocking MPI Send Modes

UNF
UNIVERSITY of NORTH FLORIDA.

- MPI_BSend()
  - Buffered send
  - Will start and may complete before a matching receive is posted
  - Will buffer the message if no matching receive
  - User must define buffer for it
- MPI_SSend()
  - Synchronous send
  - Will start whether or not a matching receive was posted
  - Will only complete when that receive is posted
- MPI_RSend()
  - Ready send
  - Will only start if the matching receive has already been posted

MPI Nonblocking Routines

UNF
UNIVERSITY of NORTH FLORIDA.

- Nonblocking send - MPI_Isend() - will return "immediately" even before source location is safe to be altered.
  - Has similar send modes to blocking routines
    - MPI_Isend()
    - MPI_Ibsend()
    - MPI_Issend()
    - MPI_Irsend()
- Nonblocking receive - MPI_Irecv() - will return even if no message to accept.

Nonblocking Routine Formats

- `MPI_Isend(buf,count,datatype,dest,tag,comm,request)`
- `MPI_Irecv(buf,count,datatype,source,tag,comm, request)`
- Completion detected by **`MPI_Wait()`** and **`MPI_Test()`**.
  - **`MPI_Wait()`** waits until operation completed and returns then.
  - **`MPI_Test()`** returns with flag set indicating whether operation completed at that time.
  - Need to know whether particular operation completed.
  - Determined by accessing **`request`** parameter.

Example

To send an integer from process 0 to process 1 and allow process 0 to continue,

```
MPI_Request request;
int number;
if (world_rank == 0) {
    // If we are rank 0, set the number to 13 and send it to process 1
    number = 13;
    MPI_Isend(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &request);
    printf("Sent the number %d to process 1 \n", number);
    // Now do something else
    for(int i=0; i<500; i+=29){
        number = number + i;
        printf("Added %d, now the number is %d \n", i, number);
    }
    printf("Addition loop finished. \n");
    MPI_Wait(&request, MPI_STATUS_IGNORE);
}
else if (world_rank == 1) {
    // Listen out for the number from process 0
    MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Process 1 received number %d from process 0\n", number);
}
```

```
[n01233188@h01 ~]$ mpirun -n=2 ./IsendReceive.out
Sent the number 13 to process 1
Added 0, now the number is 13
Added 29, now the number is 42
Added 58, now the number is 100
Added 87, now the number is 187
Added 116, now the number is 303
Added 145, now the number is 448
Added 174, now the number is 622
Added 203, now the number is 825
Added 232, now the number is 1057
Added 261, now the number is 1318
Added 290, now the number is 1608
Added 319, now the number is 1927
Added 348, now the number is 2275
Added 377, now the number is 2652
Process 1 received number 13 from process 0
Added 406, now the number is 3058
Added 435, now the number is 3493
Added 464, now the number is 3957
Added 493, now the number is 4450
Addition loop finished.
[n01233188@h01 ~]$
```

Blocking vs Nonblocking

UNF
UNIVERSITY of NORTH FLORIDA.

- Blocking communication is beneficial for synchronized processes
  - Can cause performance bottlenecks when processes must stop until communication is done
  - This is not usually expected to be a problem
- Nonblocking communication allows processes to do something else while it waits for a send/receive
  - Provides performance increases when communication and computation can be done in parallel
  - Makes it difficult for processes to deadlock

Nonblocking sends can be used with blocking receives and vice-versa

Deadlocks

UNF
UNIVERSITY of NORTH FLORIDA.

- When two or more processes are blocked due to a circular dependency
- Happens under 4 conditions:
  - Mutual Exclusion
    - Some resources are useable by one process at a time
  - Hold and Wait
    - A blocked process is holding a resource and is waiting for another one
  - No Preemption
    - Held resources can only be released by the process holding it
  - Circular Wait
    - One process is waiting for another process to finish, which is waiting for the first process to finish

Safe Scenario

UNF UNIVERSITY of NORTH FLORIDA.

- Always succeeds, even if no buffering is done.

```
if(rank==0)
{
   MPI_Send(1,…);
   MPI_Recv(1,...);
}
else if(rank==1)
{
   MPI_Recv(0,...);
   MPI_Send(0,...);
}
```

Deadlock

UNF
UNIVERSITY of NORTH FLORIDA.

- Will always deadlock, no matter the buffering mode.

```
if(rank==0)
{
   MPI_Recv(1,...);
   MPI_Send(1,...);
}
else if(rank==1)
{
   MPI_Recv(0,...);
   MPI_Send(0,...);
}
```

Unsafe Scenario

UNF
UNIVERSITY of NORTH FLORIDA.

- Only succeeds if sufficient buffering is present -- unsafe!

```
if(rank==0)
{
   MPI_Send(1,...);
   MPI_Recv(1,...);
}
else if(rank==1)
{
   MPI_Send(0,...);
   MPI_Recv(0,...);
}
```

Some Solutions to the "unsafe" Problem

- Order the operations more carefully:

Process 0	Process 1
`Send(1)`	`Recv(0)`
`Recv(1)`	`Send(0)`

- Use non-blocking operations:

Process 0	Process 1
`Isend(1)`	`Isend(0)`
`Irecv(1)`	`Irecv(0)`
`Waitall`	`Waitall`

Multiple Recv

UNF
UNIVERSITY of NORTH FLORIDA.

```
if(rank==0)
{
  MPI_Recv(1,...);
  MPI_Recv(2,...);
}
else if(rank==1)
{
  MPI_Send(0,...);
} else if (rank==2) {
  MPI_Send(0,...);
}
```

Collective Communication of MPI

Point to Point vs Collective Communication

UNF UNIVERSITY of NORTH FLORIDA.

- Point-to-point communication involves direct data exchange between two individual processes.
  - Example: MPI_Send and MPI_Recv
- Collective communication involves data exchange among a group of processes (usually all the processes in a communicator). The operation is defined to involve all processes in a communicator, ensuring synchronization.
  - Example: MPI_Bcast, MPI_Reduce

Characteristics

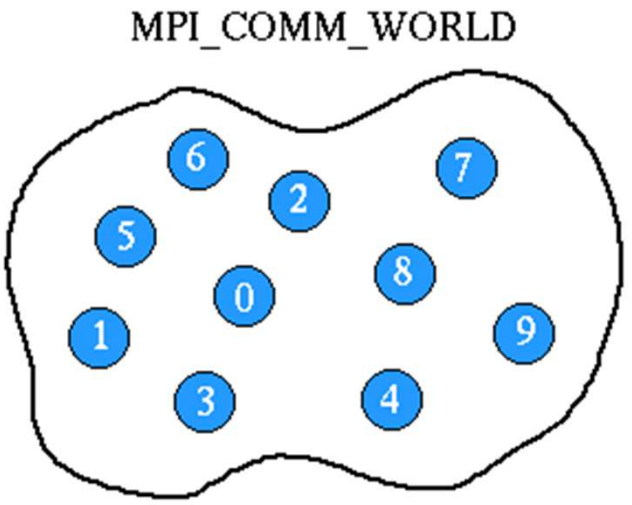
UNF
UNIVERSITY of NORTH FLORIDA.

- **Group Coordination:** Collective routines involve coordinated communication within a group of processes (identified by an MPI communicator).
- **Simplifies Complex Communication:** Serve as substitutes for a complex sequence of point-to-point calls.
- **Blocking Behavior:** Must block until they complete locally (for blocking calls).
- **Synchronization:** May or may not use synchronized communication (depends on implementation).
- **Root Process:** Some routines specify a root process to originate or collect all data.
- **Data Matching:** Must match data amounts between senders and receivers exactly.
- **No Message Tags:** Collective routines do not use message tags.
- **Variations:** Include many variations within basic categories (e.g., MPI_Bcast, MPI_Reduce, MPI_Gather).
- **Efficiency:** Built upon point-to-point routines but optimized for group communication.

Communicators

UNF
UNIVERSITY of NORTH FLORIDA.

- An MPI Communicator is an object that defines a group of processes that can communicate with each other in an MPI program.
- MPI_COMM_WORLD includes all the processes started by the MPI program.
- Each process in a communicator has a unique rank (ID).



Three Types of Operations

UNF
UNIVERSITY of NORTH FLORIDA.

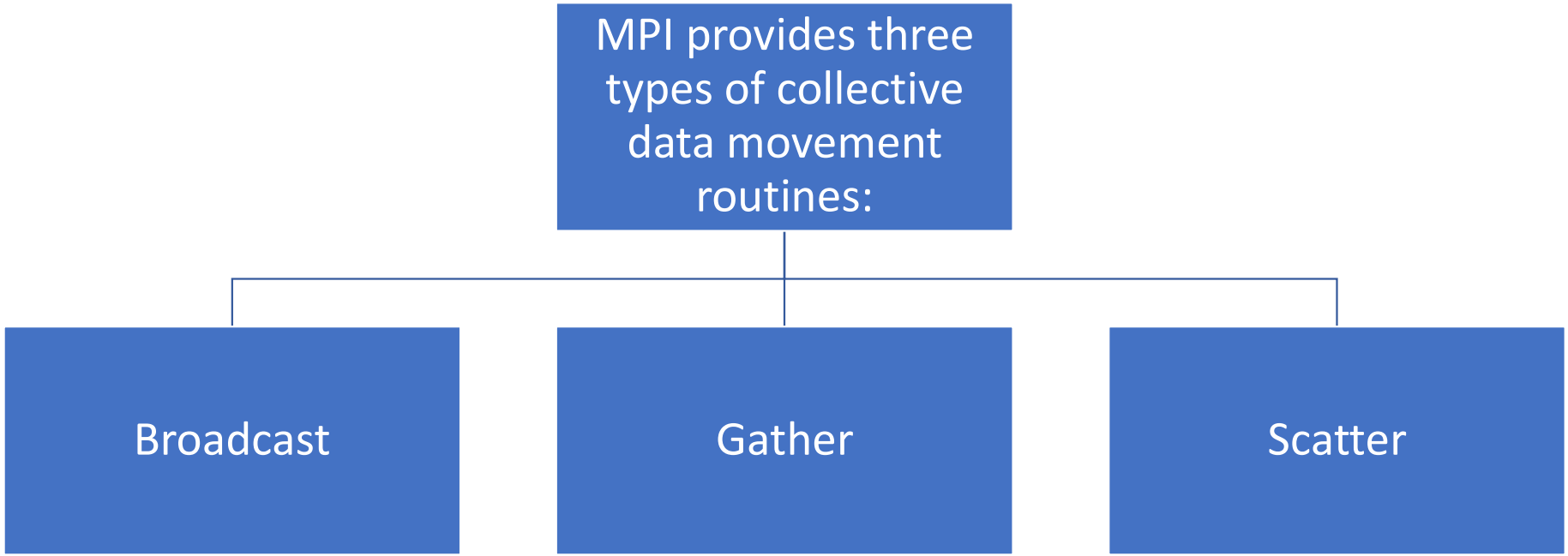
Synchronization:	Data Movement:	Collective Computation (Reductions):
• Processes wait until all members of the group have reached the synchronization point. • Function: MPI_Barrier	• broadcast, scatter/gather, all to all.	• One member of the group collects data from other members and performs an operation (min, max, add, multiply, etc.) on that data.

Barrier Synchronization

UNF
UNIVERSITY of NORTH FLORIDA.

- A collective operation where all processes in a communicator wait until each process reaches the synchronization point.
- Ensures that all processes align before any of them proceed to the next section of code.
  - Function: int MPI_Barrier(MPI_Comm comm)
    - comm: The communicator containing the group of processes.
    - return value is an integer value error code.
- Blocking: The function call blocks the processes until every member of the communicator has called MPI_Barrier.

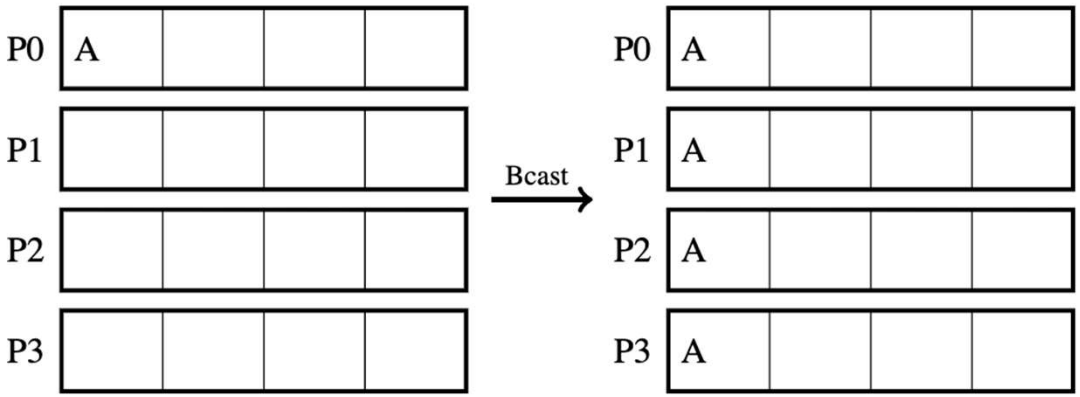
Data Movement



Broadcast

A collective operation where one process (the root) sends data to all other processes in the communicator.

```
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int
source, MPI_Comm comm)
```

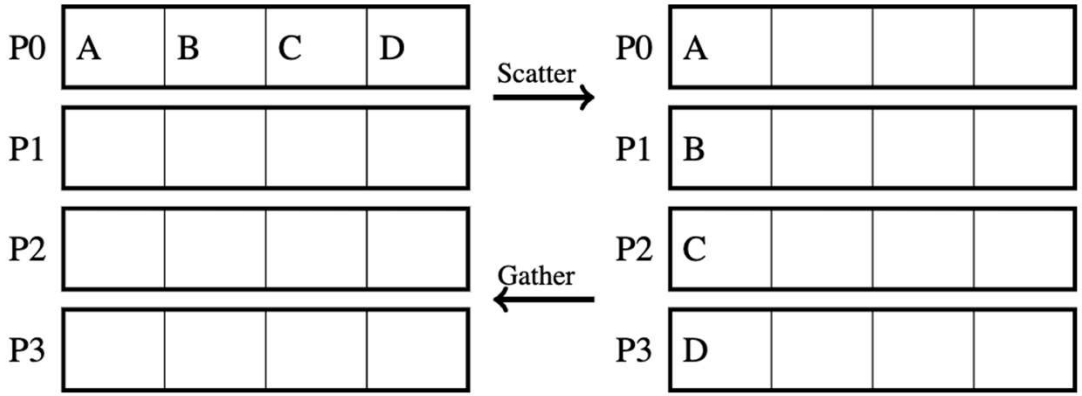


[9]

Gather

- Collects data from all processes in the communicator and gathers it into a single array at the root process.

```
int MPI_Gather(void *sbuf, int scount, MPI_Datatype stype, void *rbuf, int
rcount, MPI_Datatype rtype, int root, MPI_Comm comm )
```

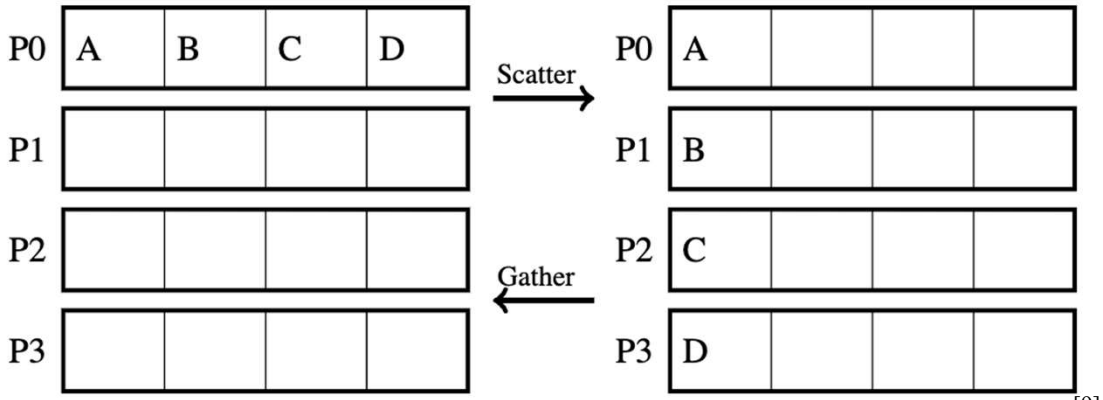


[9]

Scatter

- Divides data from the root process into equal parts and distributes them to all processes in the communicator.

```
int MPI_Scatter(void *sbuf, int scount, MPI_Datatype stype, void *rbuf, int
rcount, \ MPI_Datatype rtype, int root, MPI_Comm comm)
```

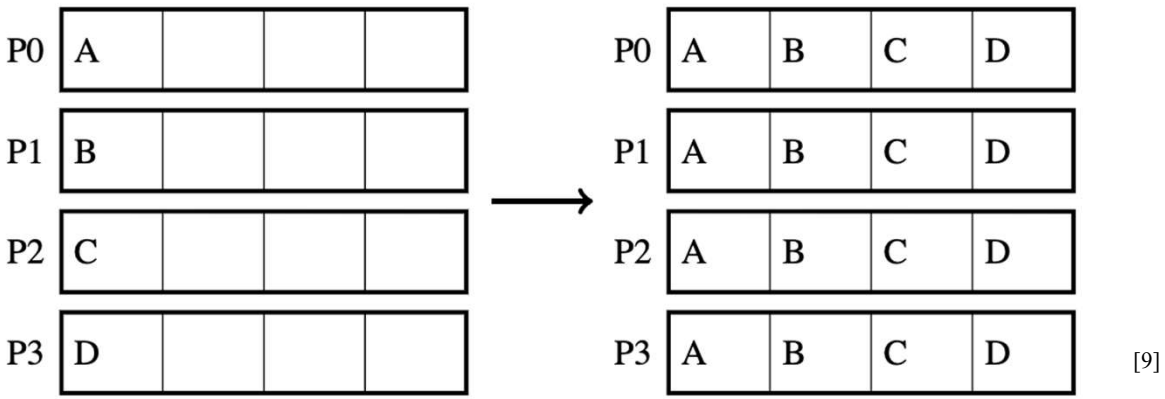


[9]

All gather

A collective operation where each process in the communicator sends its data to all other processes, collecting the data from all processes into a single array on every process.

```
int MPI_Allgather(void *sbuf, int scount, MPI_Datatype stype, void *rbuf, int
rcount, MPI_Datatype rtype, MPI_Comm comm)
```

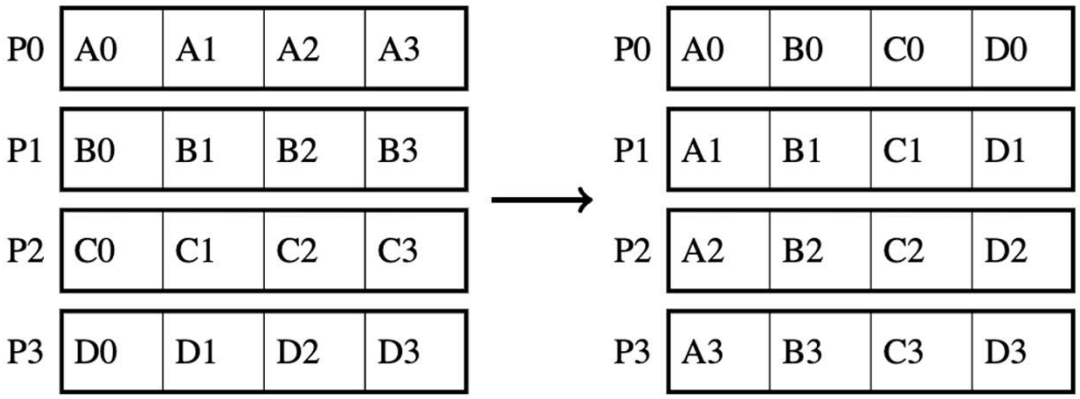


[9]

All to All

Every process sends data to and receives data from every other process in the communicator.

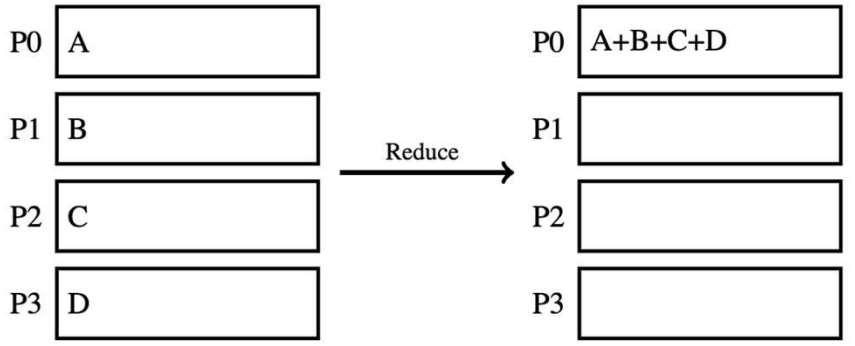
```
int MPI_Alltoall(void *sbuf, int scount, MPI_Datatype stype, void *rbuf, int
rcount, MPI_Datatype rtype, MPI_Comm comm)
```



[9]

Reduce

- A collective operation that combines data from all processes in a communicator using a specified operation (e.g., sum, minimum, maximum) and returns the result to one process (or all processes).
- Common Function:
  - MPI_Reduce: Reduces data to a single process (root).
  - MPI_Allreduce: Reduces data and distributes the result to all processes.



[9]

- int MPI_Reduce(void *sbuf, void *rbuf, int count, MPI_Datatype stype, MPI_Op op, int root, MPI_Comm comm)
- int MPI_Allreduce(void *sbuf, void *rbuf, int count, MPI_Datatype stype, MPI_Op op, MPI_Comm comm)

- MPI_Reduce_scatter: Combines data across all processes and scatters parts of the result back to each process.
- int MPI_Reduce_scatter(void *sbuf, void *rbuf, int *rcount, MPI_Datatype stype, MPI_Op op, MPI_Comm comm)

MPI predefined operations

UNF UNIVERSITY of NORTH FLORIDA.

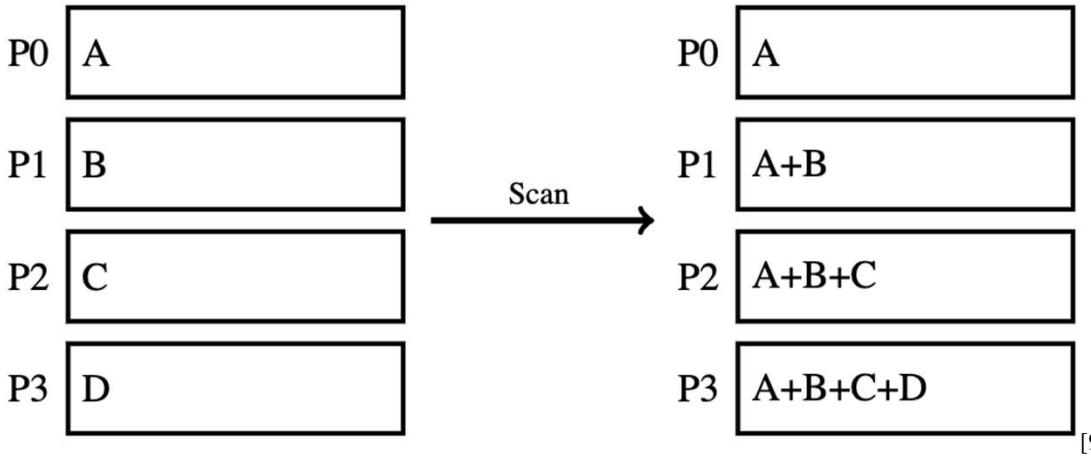
Name	Meaning	C type
MPI_MAX	maximum	integer, float
MPI_MIN	minimum	integer, float
MPI_SUM	sum	integer, float
MPI_PROD	product	integer, float
MPI_LAND	logical and	integer
MPI_BAND	bit-wise and	integer, MPI_BYTE
MPI_LOR	logical or	integer
MPI_BOR	bit-wise or	integer, MPI_BYTE
MPI_LXOR	logical xor	integer
MPI_BXOR	bit-wise xor	integer, MPI_BYTE
MPI_MAXLOC	max value and location	combination of int, float, double, and long double
MPI_MINLOC	min value and location	combination of int, float, double, and long double

[10]

Scan

- A collective operation that performs a prefix reduction (cumulative operation) across all processes in a communicator. Each process receives the result of the reduction up to its own rank.

```
int MPI_Scan(void *sbuf, void *rbuf, int count, MPI_Datatype datatype,
MPI_Op op, MPI_Comm comm)
```



[9]

Evaluating Parallel Programs

UNF
UNIVERSITY of NORTH FLORIDA.

Sequential execution time, t_s: Estimate by counting computational steps of best sequential algorithm.

Parallel execution time, t_p: In addition to number of computational steps, t_{comp}, need to estimate communication overhead, t_{comm}:

$$t_p = t_{comp} + t_{comm}$$

Communication Time

Many factors, including network structure and network contention. As a first approximation, use

$$t_{\text{comm}} = t_{\text{startup}} + nt_{\text{data}}$$

t_{startup} is startup time, essentially time to send a message with no data. Assumed to be constant.

t_{data} is transmission time to send one data word, also assumed constant, and there are n data words.

Benchmark Factors

With t_s, t_{comp}, and t_{comm}, can establish speedup factor and computation/communication ratio for a particular algorithm/implementation:

$$\text{Speedup factor} = \frac{t_s}{t_p} = \frac{t_s}{t_{comp} + t_{comm}}$$

$$\text{Computation/communication ratio} = \frac{t_{comp}}{t_{comm}}$$

Both functions of number of processors, p, and number of data elements, n.

- Speedup factor gives indication of scalability of parallel solution with increasing number of processors and problem size.
- Computation/communication ratio will highlight effect of communication with increasing problem size and system size.

Amdahl's Law

UNF
UNIVERSITY of NORTH FLORIDA.

- F is the fraction of a calculation that is sequential
- 1-F is the fraction that can be parallelized
- the maximum speed-up that can be achieved by using P processors is $\frac{1}{F+(1-F)/P}$
- Example: if 90% of a calculation can be parallelized (i.e. 10% is sequential) then the maximum speed-up which can be achieved on 5 processors is 1/(0.1+(1-0.1)/5) or roughly 3.6 (i.e. the program can theoratically run 3.6 times faster on five processors than on one)
- Amdahl's Law is a statement of the maximum theoretical speed-up you can ever hope to achieve. The actual speed-ups are always less than the speed-up predicted by Amdahl's Law.

!!!! Trivia Time !!!!

UNF
UNIVERSITY of NORTH FLORIDA.

https://www.flexiquiz.com/live

GitHub

UNF
UNIVERSITY of NORTH FLORIDA.

https://github.com/jonathanoberry/MPI_Pres_Files

References

[1] A. S. Gillis and S. J. Bigelow, “What is message passing interface (MPI)?,” Enterprise Desktop, https://www.techtarget.com/searchenterprisedesktop/definition/message-passing-interface-MPI (accessed Sep. 13, 2024).

[2] Download Microsoft MPI V10.1.3 from official Microsoft Download Center, https://www.microsoft.com/en-us/download/details.aspx?id=105289 (accessed Sep. 13, 2024).

[3] NERSC, “Cray MPICH,” Cray MPICH - NERSC Documentation, https://docs.nersc.gov/development/programming-models/mpi/cray-mpich/ (accessed Sep. 13, 2024).

[4] IBM, “IBM Spectrum MPI - Overview,” IBM, https://www.ibm.com/products/spectrum-mpi?utm_content=SRCWW&p1=Search&p4=43700067987454015&p5=p&p9=58700007548292905&gclid=CjwKCAjwxY-3BhAuEiwAu7Y6s5rb1xR9H6NwqLsTcAFoxi80pSOMxg87zUly_piX0KoPXaBqsFWRzhoC0qkQAvD_BwE&gclsrc=aw.ds (accessed Sep. 13, 2024).

[5] Intel, “Intel® MPI Library,” Intel, https://www.intel.com/content/www/us/en/developer/tools/oneapi/mpi-library.html#gs.ejds3u (accessed Sep. 13, 2024).

[6] “Deinompi,” DeinoMPI - High Performance Parallel Computing for Windows, https://mpi.deino.net/ (accessed Sep. 13, 2024).

[7] OpenMPI, “9.1. quick start: Building MPI applications,” 9.1. Quick start: Building MPI applications - Open MPI 5.0.x documentation, https://docs.open-mpi.org/en/v5.0.x/building-apps/quickstart.html (accessed Sep. 14, 2024).

References

[8] Xiao Qin, Hong Jiang, A. Manzanares, Xiaojun Ruan, and Shu Yin, “Communication-aware load balancing for parallel applications on clusters,” *IEEE Transactions on Computers*, vol. 59, no. 1, pp. 42–52, Jan. 2010. doi:10.1109/tc.2009.108

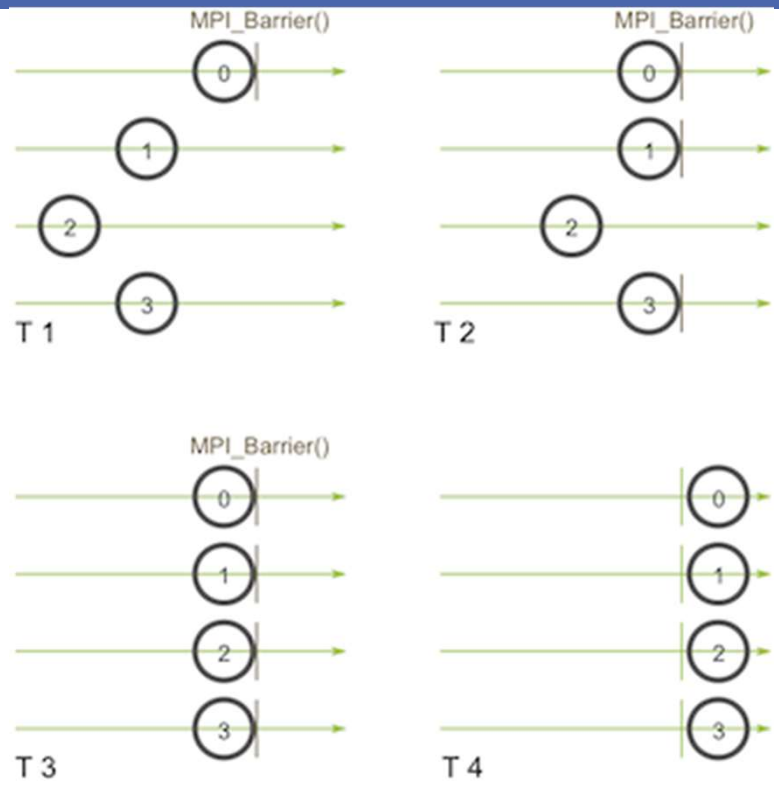
[9] D. Bindel, ”Applications of Parallel Computers, MPI Programming," Cornell University, Oct. 6, 2020. [Online]. Available: https://www.cs.cornell.edu/courses/cs5220/2020fa/lec/2020-10-06-mpi.html. [Accessed: Sep. 15, 2024].

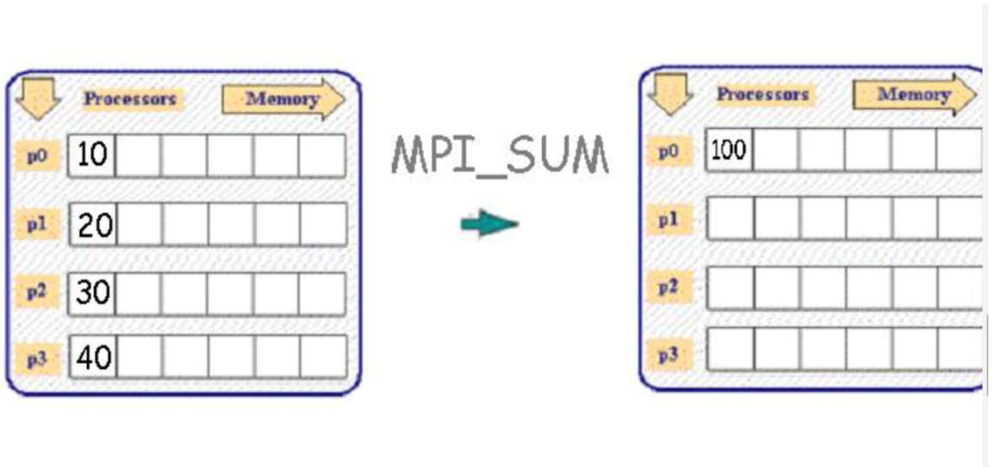
[10] Cornell Virtual Workshop, "Introduction to MPI Collective Communication," Cornell University. [Online]. Available: https://cvw.cac.cornell.edu/mpicc/intro/index. [Accessed: Sep. 15, 2024].

[11] NSA, “Capt. Grace Hopper on Future Possibilities: Data, Hardware, Software, and People (Part One, 1982),” YouTube, https://www.youtube.com/watch?v=si9iqF5uTFk (accessed Sep. 15, 2024).

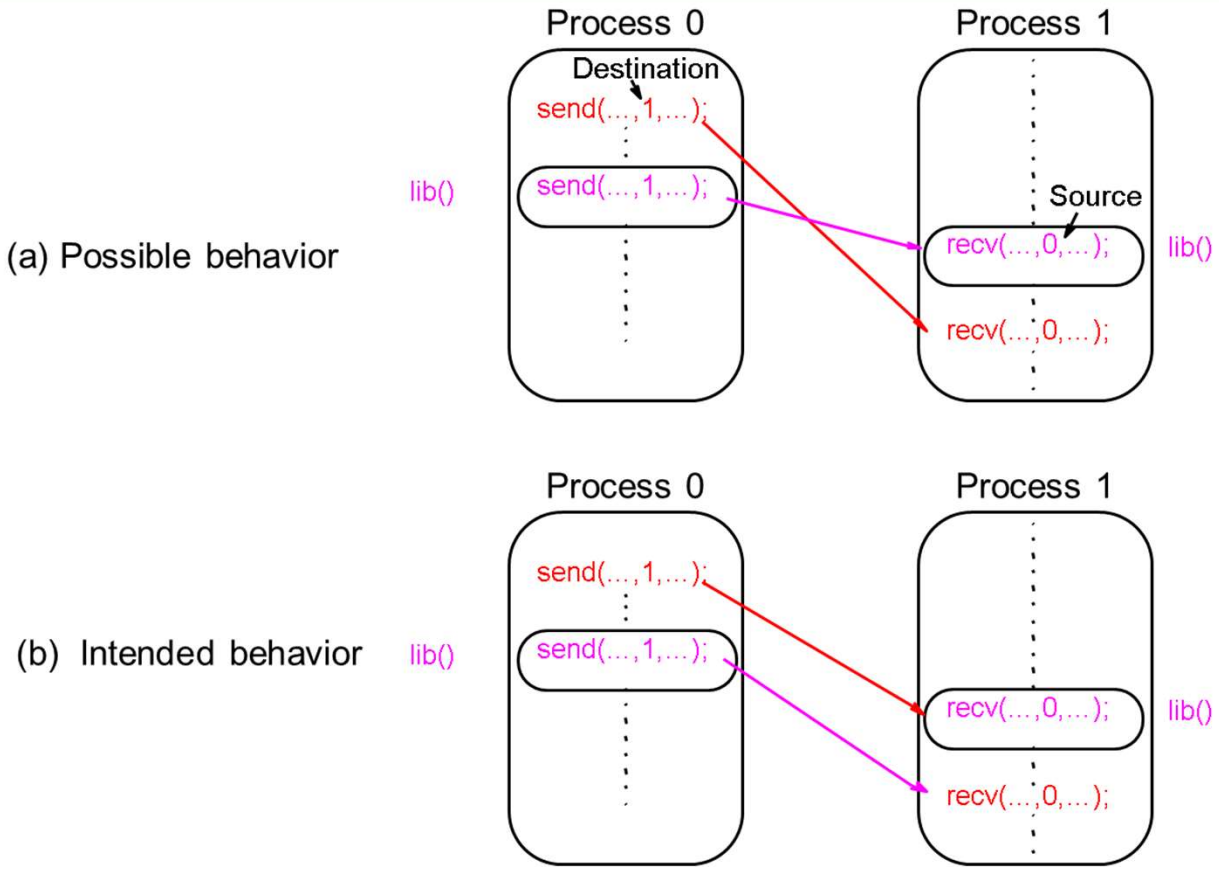
Thank You

Barrier


MPI_Barrier()
0
1
2
3
T 1
MPI_Barrier()
0
1
2
3
T 2
MPI_Barrier()
0
1
2
3
T 3
0
1
2
3
T 4


Processors
Memory
p0
10
p1
20
p2
30
p3
40
MPI_SUM
Processors
Memory
p0
100
p1
p2
p3

Message passing - Example



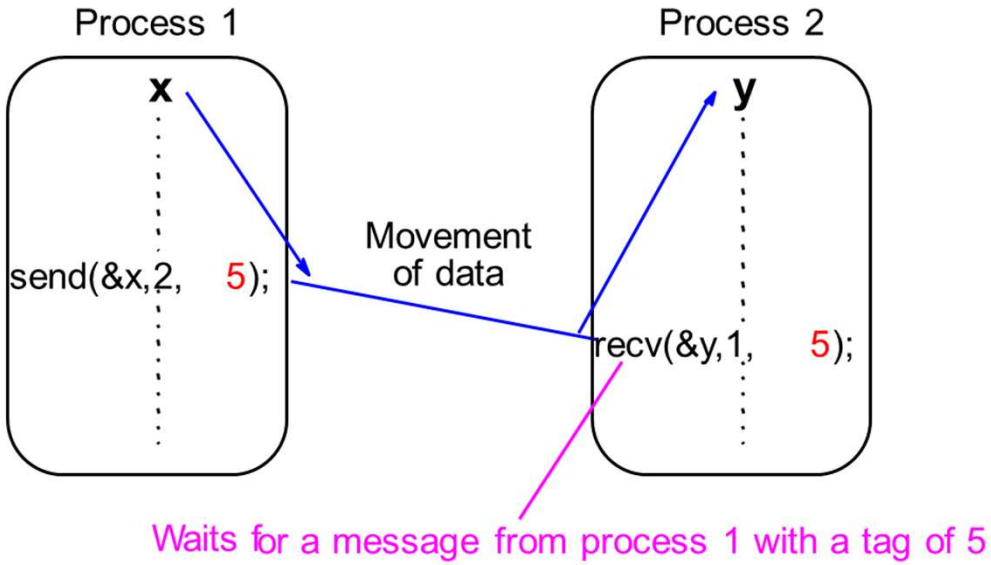
Message Tag

UNF
UNIVERSITY of NORTH FLORIDA.

- Used to differentiate between different types of messages being sent.
- Message tag is carried within message.
- If special type matching is not required, a wild card message tag, **`MPI_ANY_TAG`**, is used, so that the recv() will match with any send().

Message Tag Example

- To send a message, x, with message tag 5 from a source process, 1, to a destination process, 2, and assign to y:



Retrieving Further Information

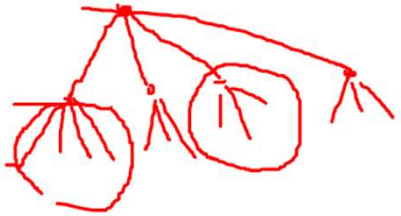
UNF UNIVERSITY of NORTH FLORIDA.

- **`Status`** is a data structure allocated in the user's program.
- In C:

```
int recvd_tag, recvd_from, recvd_count;
MPI_Status status;
MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ..., &status )
recvd_tag  = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
MPI_Get_count( &status, datatype, &recvd_count );
```

Collective Communication

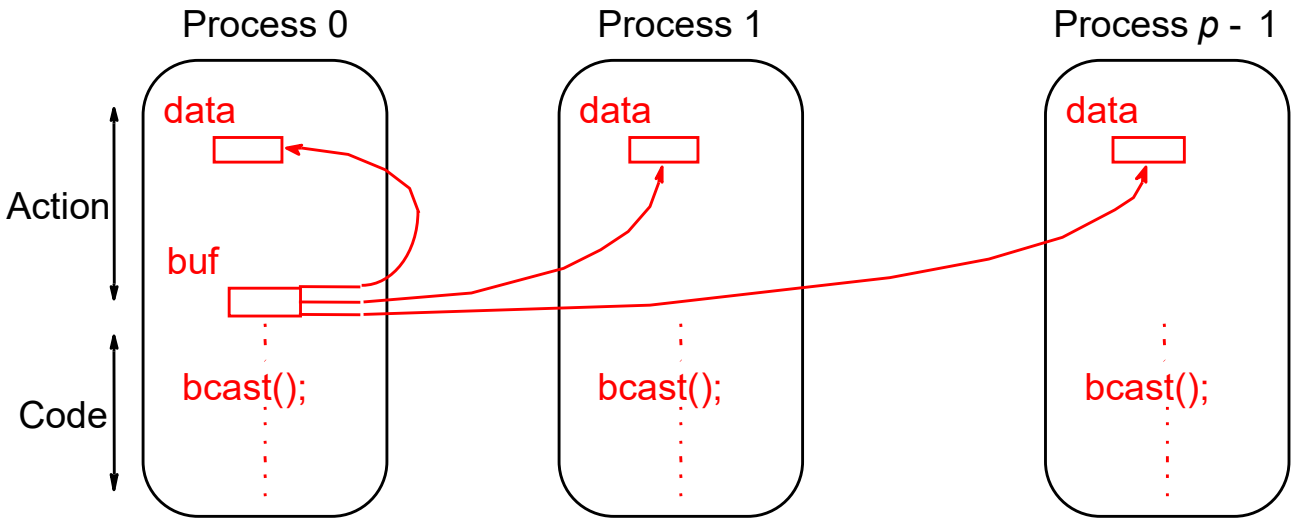
Involves set of processes, defined by a communicator. Message tags not present. Principal collective operations:



- **`MPI_Bcast()`** - Broadcast from root to all other processes
- **`MPI_Gather()`** - Gather values for group of processes
- **`MPI_Scatter()`** - Scatters buffer in parts to group of processes
- **`MPI_Alltoall()`** - Sends data from all processes to all processes
- **`MPI_Reduce()`** - Combine values on all processes to single value
- **`MPI_Reduce_scatter()`** - Combine values and scatter results
- **`MPI_Scan()`** - Compute prefix reductions of data on processes

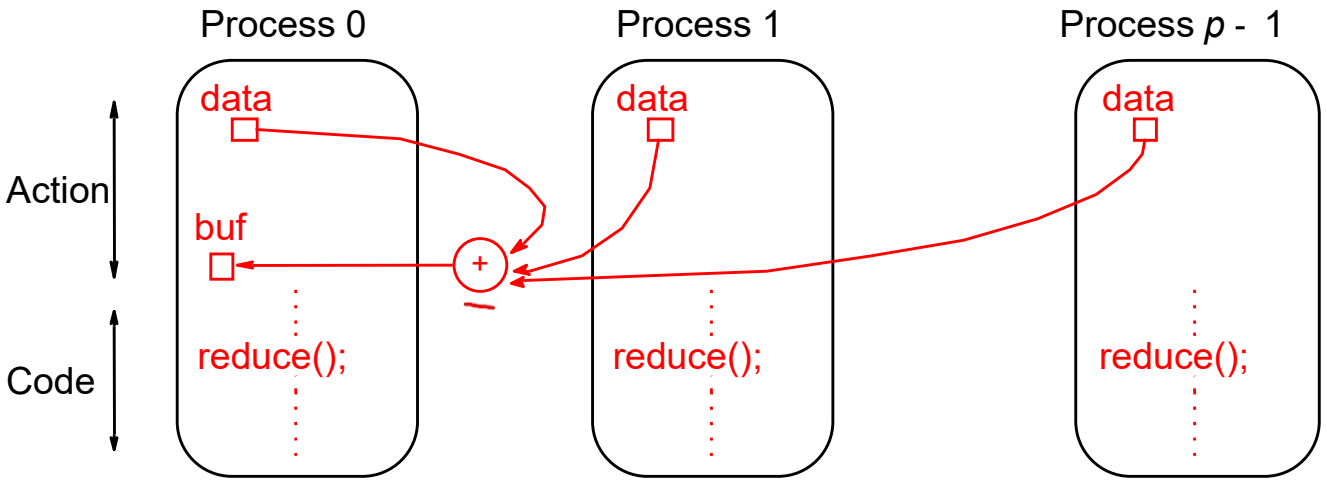
Broadcast

- Sending the same message to all processes.
  - int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)



Reduce

- Gather operation combined with specified arithmetic/logical operation.
  - Example: Values could be gathered and then added together by root:
  - int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)



MPI Operations for MPI_Reduce

UNF UNIVERSITY of NORTH FLORIDA.

- MPI operation Meaning
  MPI_MAX maximum, max
  MPI_MIN minimum, min
  MPI_SUM sum
  MPI_PROD product
  MPI_LAND logical and
  MPI_LOR logical or
  MPI_LXOR logical exclusive or
  MPI_BAND bitwise and
  MPI_BOR bitwise or
  MPI_BXOR bitwise exclusive or

Example: Factorial of n

UNF UNIVERSITY of NORTH FLORIDA.

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

int main(int argc, char** argv)
{
int MyRank, NumProcs, i, n, first, last, prod=1, result=1, Root=0;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);
MPI_Comm_size(MPI_COMM_WORLD, &NumProcs);

MPI_Bcast(&n, 1, MPI_INT, Root, MPI_COMM_WORLD);

first = (MyRank*n)/NumProcs;
last = ((MyRank+1)*n)/NumProcs;
//Last is the first element of the next processor

for(i=first+1; i<=last; i++)
       prod*=i;

MPI_Reduce(&prod, &result, 1, MPI_INT, MPI_PROD, Root, MPI_COMM_WORLD);

if(MyRank == Root)
    printf("\nFactorial %d is %d", n, result);

MPI_Finalize();
return 0;
}
```

A program to calculate the factorial of n numbers using p processors. The number n is input by the user. Root broadcasts the number to all the processes in MPI_COMM_WORLD. Each processor computes its share of the product, and the result is collected back in root using MPI_PROD as the reduce function.

Skip

Where Can You Get MPI?

- ▲ IBM
- ◆ Gateway
- ● GNU
- ■ Dell

The Correct Answer Is:

UNF
UNIVERSITY of NORTH FLORIDA.

IBM

What is the purpose of the Message Tag?

Skip

MPI
MUNICIPAL POLICE INSTITUTE

▲ Define the order of messages

◆ Identify an item as a message

● Show where a message starts and ends

■ Differentiate between types of message

The Correct Answer Is:

UNF
UNIVERSITY of NORTH FLORIDA.

- [x] Differentiate between types of message

Non-blocking returns after local actions have completed.

Skip

♦ True

▲ False

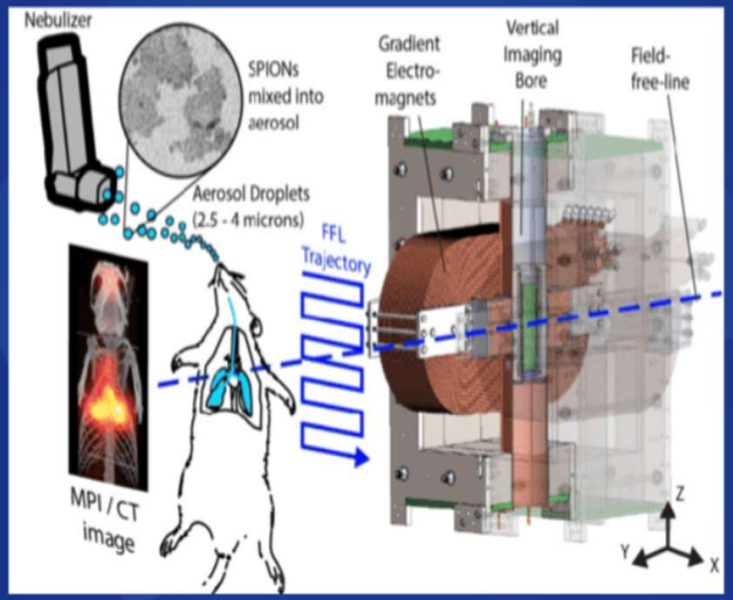
The Correct Answer Is:

UNF
UNIVERSITY of NORTH FLORIDA.

▲ False

Skip

A non-blocking routine completion is detected by:



▲ MPI Stop

◆ MPI Test

● MPI Wait

■ MPI Sleep

The Correct Answer Is:

UNF
UNIVERSITY of NORTH FLORIDA.