Distributed Memory Programming

Dr. Hrachya Astsatryan,
Institute for Informatics and Automation Problems,
National Academy of Sciences of Armenia,
E-mail: hrach@sci.am

1

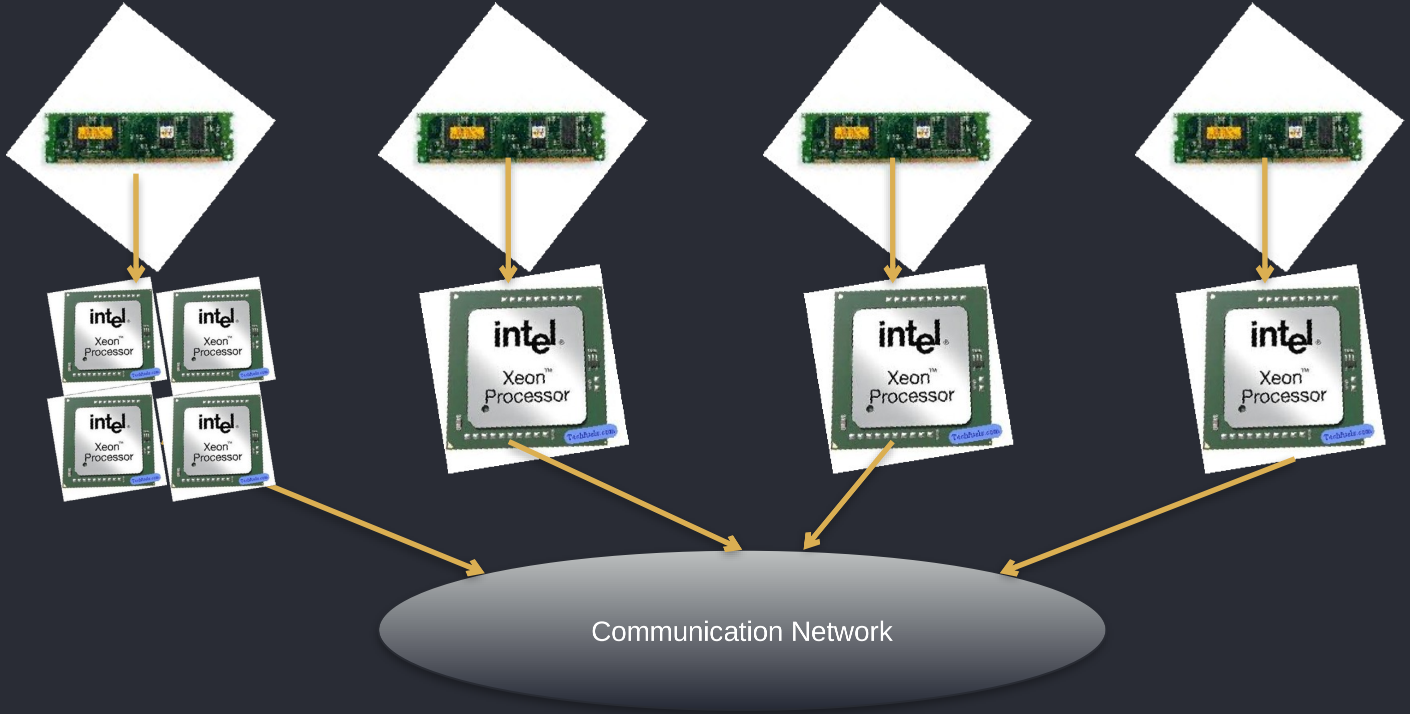
Distributed memory programming

Benefits

- Each processor has its memory, creating isolated memory spaces across the system.
- Data communication between processors occurs through a fabric, requiring explicit send and receive operations.
- Communication between processors involves manual coordination, with programmers specifying the data to be sent and received.
- Synchronization between processors is directly tied to communication, meaning data synchronization occurs automatically when a receive operation completes.

Distributed Memory Programming Paradigm

Each node has rapid access to its own local memory and access to the memory of other nodes via some sort of communications network



2

Message Passing Interface (MPI)

MPI

The MPI is a message passing library standard based on the consensus of the MPI Forum, which has over 40 participating organizations, including vendors, researchers, software library developers, and users.

intel

hp HEWLETT® PACKARD

CRAY RESEARCH

Sun microsystems

IBM®

nVIDIA.

JARA Jülich Aachen Research Alliance

OAK RIDGE National Laboratory

NSF National Science Foundation WHERE DISCOVERIES BEGIN

CALIFORNIA INSTITUTE OF TECHNOLOGY 1891

Myricom Products from CSPI

University of Stuttgart Germany

SiliconGraphics

HITACHI Inspire the Next

東京工業大学 Tokyo Institute of Technology

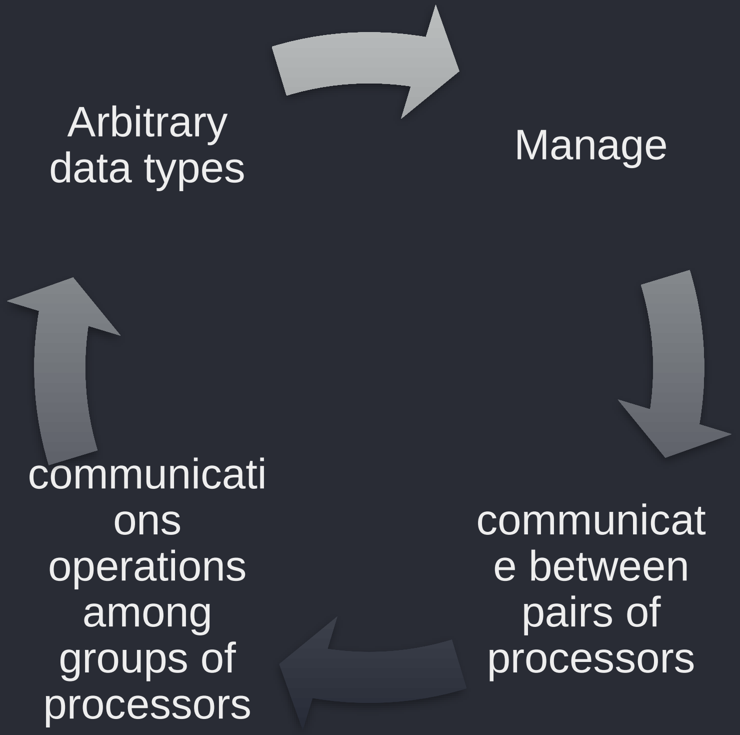
THE UNIVERSITY of TENNESSEE UT KNOXVILLE

Argonne NATIONAL LABORATORY

Usage benefits

- Supports wide variety of platforms including support for heterogeneous parallel architectures.
- Provides source code portability. MPI programs should compile and run as-is on any platform.
- Debugging
- Dynamic process management. More control over data location and flow within a parallel application
- A great deal of functionality, including a number of different types of functions

Basic futures



Arbitrary data types

Manage

communicate between pairs of processors

communications operations among groups of processors

MPI Implementations: OpenMPI

The Open MPI Project is an open source MPI implementation that is developed and maintained by a consortium of academic, research, and industry partners. Open MPI is therefore able to combine the expertise, technologies, and resources from all across the High Performance Computing community in order to build the best MPI library available.

MPI Implementations: MPICH

MPICH is a high performance and widely portable implementation of MPI standard too. MPICH and its derivatives form the most widely used implementations of MPI in the world. They are used exclusively on nine of the top 10 supercomputers (June 2015 ranking), including the world’s fastest supercomputer: Tianhe-2.

MPI Implementations

- **OpenMPI Java -** provides Java bindings for MPI, allowing Java programmers to develop parallel and distributed applications using the MPI standard for communication and synchronization.

- **MPI4Py -** Python interface to the MPI standard, enabling Python developers to create parallel and distributed applications using MPI functionality, such as point-to-point communication, collective operations, and process management.

- others

Alternatives

- **OpenSHMEM** - programming model designed for partitioned global address space (PGAS) architectures.
- **Coarray Fortran** - extends Fortran with a SPMD parallel programming model, allowing for simple syntax for parallelism without requiring explicit message passing.
- **Unified Parallel C (UPC)** - extension of the C programming language that provides a shared memory programming model for distributed memory architectures, facilitating parallel programming.
- **Chapel** - parallel programming language developed by Cray, designed for productivity and performance on large-scale systems, featuring a multithreaded execution model and support for task parallelism.
- **X10** - high-performance programming language designed for parallel computing, featuring constructs for parallelism, distribution, and asynchrony.

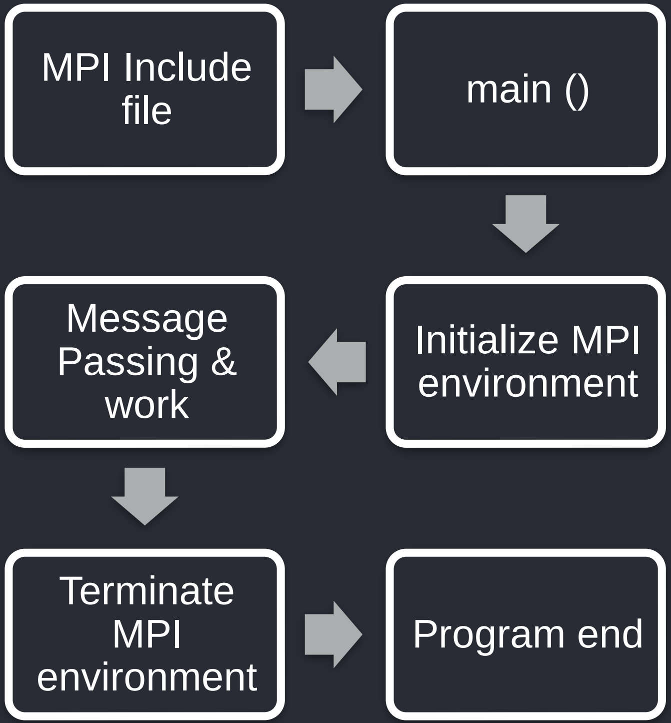
3

Starting MPI

Serial C: Hello

```
#include <stdio.h>
main (int argc, char *argv[])
{
printf("Hello world!\n");
}
```

MPI Structure



MPI Include file

main ()

Initialize MPI environment

Message Passing & work

Terminate MPI environment

Program end

MPI: Include library

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

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

printf("Hello world!\n");
return 0;

}
```

Every C/C++ MPI program must include the MPI header file (which contains the MPI function type declarations)

MPI: Initialize and Terminate

Statement needed in every program before any other MPI code. accepts the argc and argv variables that are provided as arguments to main

- MPI_Init (&argc, &argv); - initializes MPI environment (first statement of the program)

Last statement of MPI code must be. Program will not terminate without this statement

- MPI_Finalize();   - cleans up the MPI environment. No other MPI routine can be called after this call

MPI: C

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

int main (int argc, char *argv[])
{
int error;
error = MPI_Init(&argc, &argv);

printf("Hello world!\n");
error = MPI_Finalize();
return 0;
}
```

MPI: Compilation

```
mpicc -o first first.c
```

Compiler	Language	Script Name
GNU	C	mpicc
Intel	C	mpiicc
PGI	C	mpipgcc
GNU	C++	mpiCC
Intel	C++	mpiicpc
PGI	C++	mpipgCC
GNU	Fortran	mpif77
Intel	Fortran	mpiifort
PGI	Fortran	Mpipgf77/mpipgf90

MPI execution

mpirun, and mpiexec execute both serial and parallel jobs. (/usr/lib64/mpich/bin/)

- mpirun [options] <program> [<args>]

- -np (-n in case of mpiexec) - run this many copies of the program on the given nodes

- -hosts wn1,wn2, etc..
- -hostfile
- wn1:2
- wn2:2

4

Identifying processors

Groups

A group is an ordered set of processes

- Each process is associated with a rank
- Ranks are contiguous and start from zero

- Groups allow collective operations to work on a subset of processes.

Ranks also used to specify source and destination of communications.

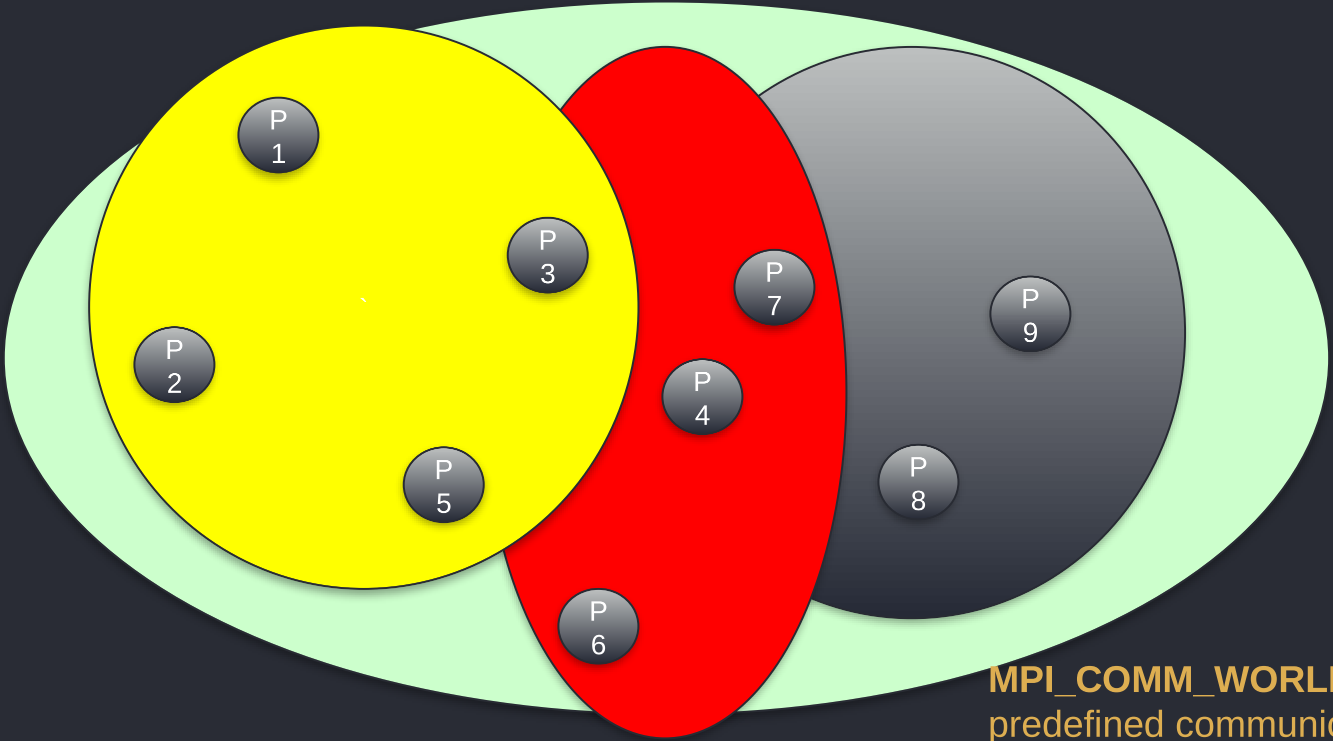
Communicator

- A communicator can be thought of as a handle to an object (group attribute) that describes a group of processes

- An intracommunicator is used for communication within a single group

- An intercommunicator is used for communication between 2 disjoint groups

Communicator



MPI_COMM_WORLD - predefined communicator that includes all MPI processes

MPI_Comm_rank

- Returns the rank of the calling MPI process within the specified communicator.

- Initially, each process will be assigned a unique integer rank between 0 and number of tasks - 1 within the communicator MPI_COMM_WORLD.

- If a process becomes associated with other communicators, it will have a unique rank within each of these as well.

```
MPI_Comm_rank (comm,rank);
```

MPI_Comm_rank

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

int main (int argc, char *argv[])
{
int error, rank;
error = MPI_Init(&argc, &argv);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);

printf(”The current process ID is %d \n”, rank);

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

MPI_Comm_size

- Returns the total number of MPI processes in the specified communicator, such as MPI_COMM_WORLD. If the communicator is MPI_COMM_WORLD, then it represents the number of MPI tasks available to your application.

```
MPI_Comm_size (comm,rank);
```

MPI_Comm_size

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

int main (int argc, char *argv[])
{
int error, rank, size;
error = MPI_Init(&argc, &argv);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
MPI_Comm_size (MPI_COMM_WORLD, &size);
printf(”The current process ID is %d from %d\n”,rank, size);

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

MPI Com. & Groups

Returns the version and subversion of the MPI standard that's implemented by the library.

- MPI_Get_version (&version,&subversion)

- MPI_Get_processor_name obtains the actual name of the processor on which the process is executing.

- MPI_Get_processor_name(processor_name, &name_len);

MPI_Comm_size

```
#include <mpi.h>
#include <stdio.h>
int main (int argc, char *argv[]) {
int error, rank, size, version, subversion;
char processor_name[MPI_MAX_PROCESSOR_NAME];
int name_len;
error = MPI_Init(&argc, &argv);
MPI_Get_version (&version,&subversion);
MPI_Get_processor_name(processor_name, &name_len);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
MPI_Comm_size (MPI_COMM_WORLD, &size);
printf ("version=%d, subversion=%d\n", version, subversion);
printf(”The current process ID is %d from %d from processor %s\n”,
rank, size, processor_name);
error = MPI_Finalize(); return 0;
}
```

MPI Com. & Groups: MPI_Wtime ()

- MPI_Wtime () - Returns an elapsed wall clock time in seconds (double precision) on the calling processor.

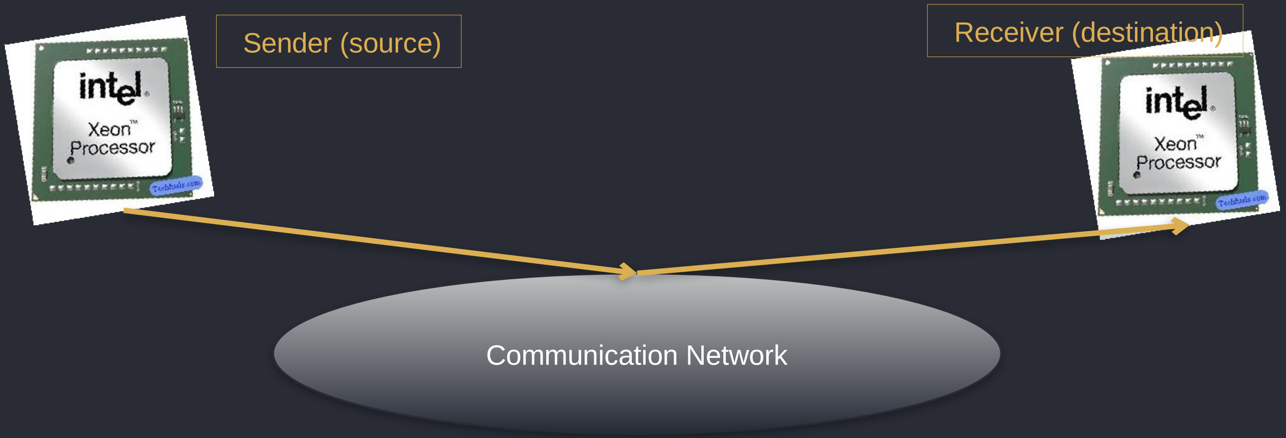
MPI Com. & Groups: MPI_Wtime ()

```
#include <mpi.h>
#include <stdio.h>
main (int argc, char *argv[]) {
int error, rank, size;
double starttime, endtime;
error = MPI_Init(&argc, &argv);
starttime = MPI_Wtime();
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
MPI_Comm_size (MPI_COMM_WORLD, &size);
printf(”The current process ID is %d from %d\n”,rank, size);
endtime   = MPI_Wtime();
printf("The execution is took %f seconds\n",endtime-
starttime);
error = MPI_Finalize();
}
```

5

First communications: point to point

Source and Destination



Sender (source)

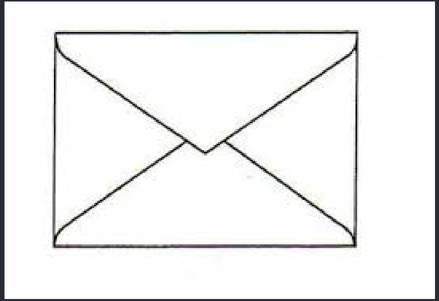
Receiver (destination)

Communication Network

Pending messages challenges ?

Where is the data?
What type of data?
How much data is sent?
To whom is the data sent?
How does the receiver know which data to collect?

Message



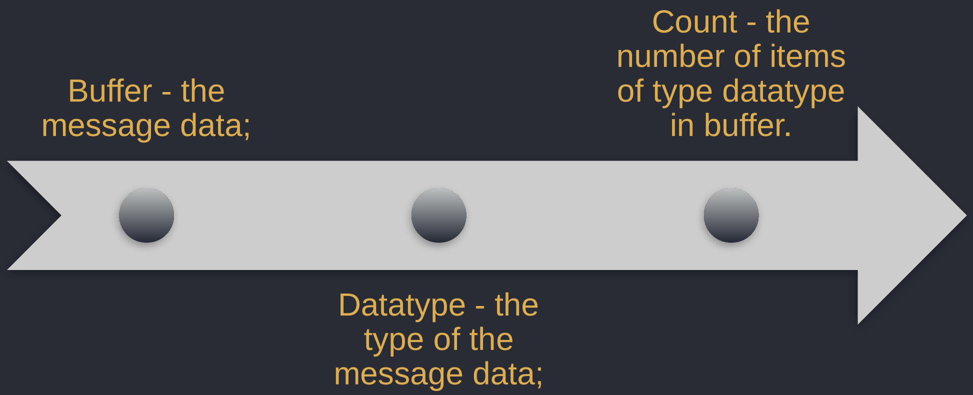
envelope

+

message body

- **source** - the sending process
- **destination** - the receiving process
- **communicator** - specifies a group of processes to which both source and destination belong
- **tag** - used to classify messages. For example, one tag value can be used for messages containing data and another tag value for messages containing status information

Message body



Buffer - the message data;

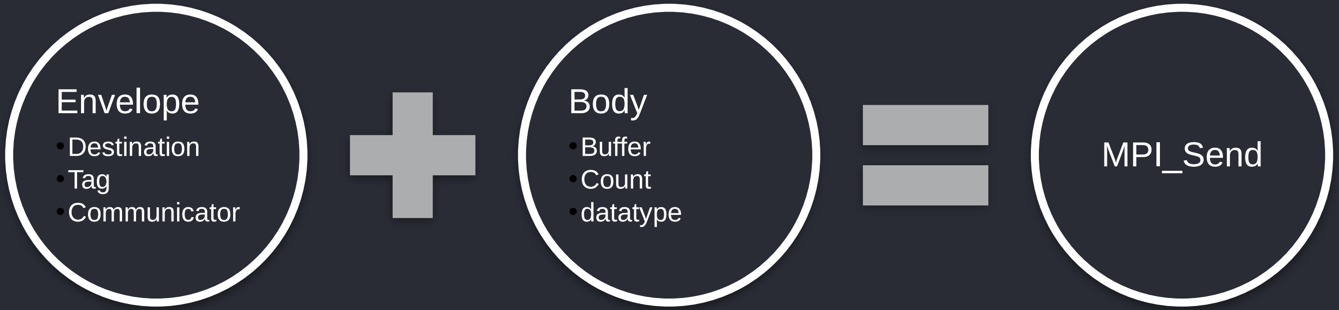
Datatype - the type of the message data;

Count - the number of items of type datatype in buffer.

- For example the buffer can be an array, where the dimension is given by count, and the type of the array elements is given by datatype.

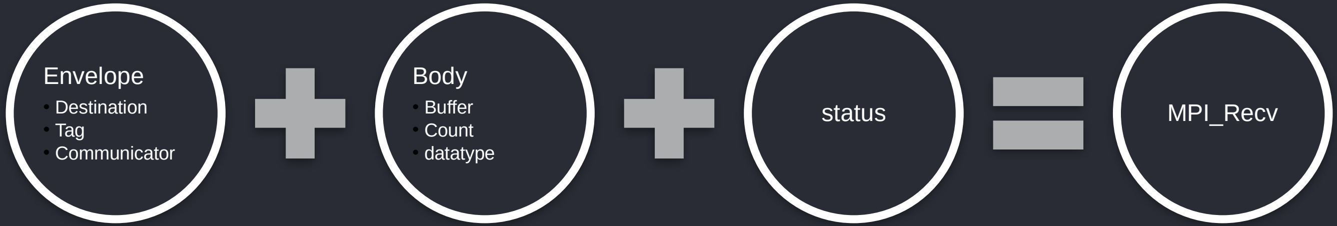
- Using datatypes and counts, rather than bytes and bytecounts, allows structured data and noncontiguous data to be handled smoothly.

Send Message



- int MPI_Send(void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm);
- All arguments are input arguments. An error code is returned by the function.

Receive Message



- int MPI_Recv(void *buf, int count, MPI_Datatype dtype, int source, int tag, MPI_Comm comm, MPI_Status *status);
- buf and status are output arguments; the rest are inputs. An error code is returned by the function.
- * - for the source (accept a message from any process) and the tag (accept a message with any tag value). If wildcards are not used, the call can accept messages from only the specified sending process, and with only the specified tag value. Communicator wildcards are not available.

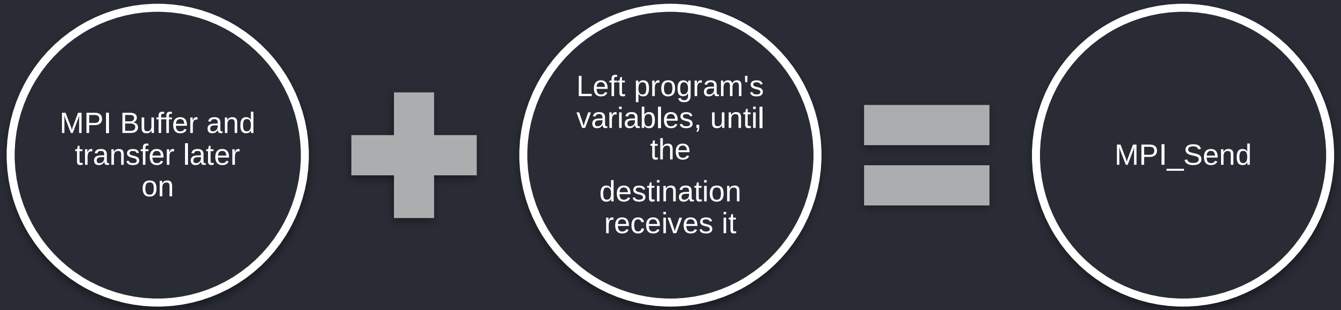
send/recv example

```
#include <mpi.h>
#include <stdio.h>
int main (int argc, char *argv[])  {
int error, rank,i;
MPI_Status status;
double myarray[80];
for (i=0;i<80;i=i+1) myarray[i]=i;
error = MPI_Init(&argc, &argv);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
if( rank == 0 ) MPI_Send(myarray, 80, MPI_DOUBLE, 1, 11,
MPI_COMM_WORLD);
else if( rank == 1 )  {
MPI_Recv(myarray, 80, MPI_DOUBLE, 0, 11, MPI_COMM_WORLD,
&status);
printf ("%f\n",myarray[50]); }
error = MPI_Finalize();
return 0;
}
```

send/recv deadlock

```
#include <mpi.h>
#include <stdio.h>
int main (int argc, char *argv[])  {
int error, rank,i;
MPI_Status status;
double myarray[80];
for (i=0;i<80;i=i+1) myarray[i]=i;
error = MPI_Init(&argc, &argv);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
if( rank == 0 ) {MPI_Recv(myarray, 80, MPI_DOUBLE, 0, 11,
MPI_COMM_WORLD, &status);
MPI_Send(myarray, 80, MPI_DOUBLE, 1, 11, MPI_COMM_WORLD);}
else if( rank == 1 )  {
MPI_Recv(myarray, 80, MPI_DOUBLE, 0, 11, MPI_COMM_WORLD,
&status);
printf ("%f\n",myarray[50]);  }
error = MPI_Finalize();
return 0;
```

Send Message

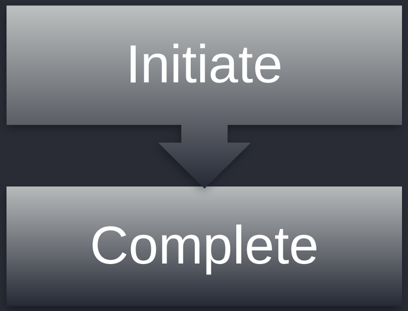


MPI Buffer and transfer later on

Left program's variables, until the

destination receives it

MPI_Send



Initiate

Complete

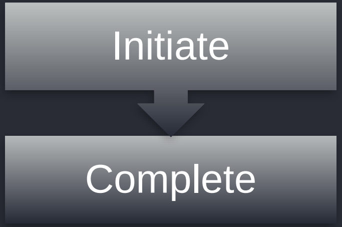
Memory usage/speed up?

send nonblocking

- int MPI_Isend(void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm, MPI_Request *request);

- The request handle identifies the send operation that was posted. The request handle can be used to check the status of the posted send or to wait for its completion.

- None of the arguments passed to MPI_ISEND should be read or written until the send operation it invokes is completed.



Initiate

Complete

receive nonblocking

int MPI_Irecv(void *buf, int count, MPI_Datatype dtype, int source, int tag, MPI_Comm comm, MPI_Request *request);

The request handle identifies the send operation that was posted. The request handle can be used to check the status of the posted recv or to wait for its completion.

None of the arguments passed to MPI_Irecv should be read or written until the send operation it invokes is completed.

Initiate

Complete

Complete

MPI_ISEND or MPI_IRECV can subsequently wait for the posted operation to complete by calling MPI_WAIT

```
int MPI_Wait( MPI_Request *request, MPI_Status *status );
```

a request handle (returned when the send or receive was posted)

for receive, information on the message received; for send, may contain an error code

Initiate
Complete

Send modes

Standard

- MPI internal buffer and transferred asynchronously to the destination process, or the source and destination processes synchronize on the message.

Synchronous

- the sending process may assume the destination process has begun receiving the message. The destination process need not be done receiving the message, but it must have begun receiving the message.

Ready

- a matching receive has already been posted at the destination process before ready mode send is called. If a matching receive has not been posted at the destination, the result is undefined

Buffered

- requires MPI to use buffering

A receiving process can use the same call to MPI_RECV or MPI_IRECV, regardless of the send mode used to send the message.

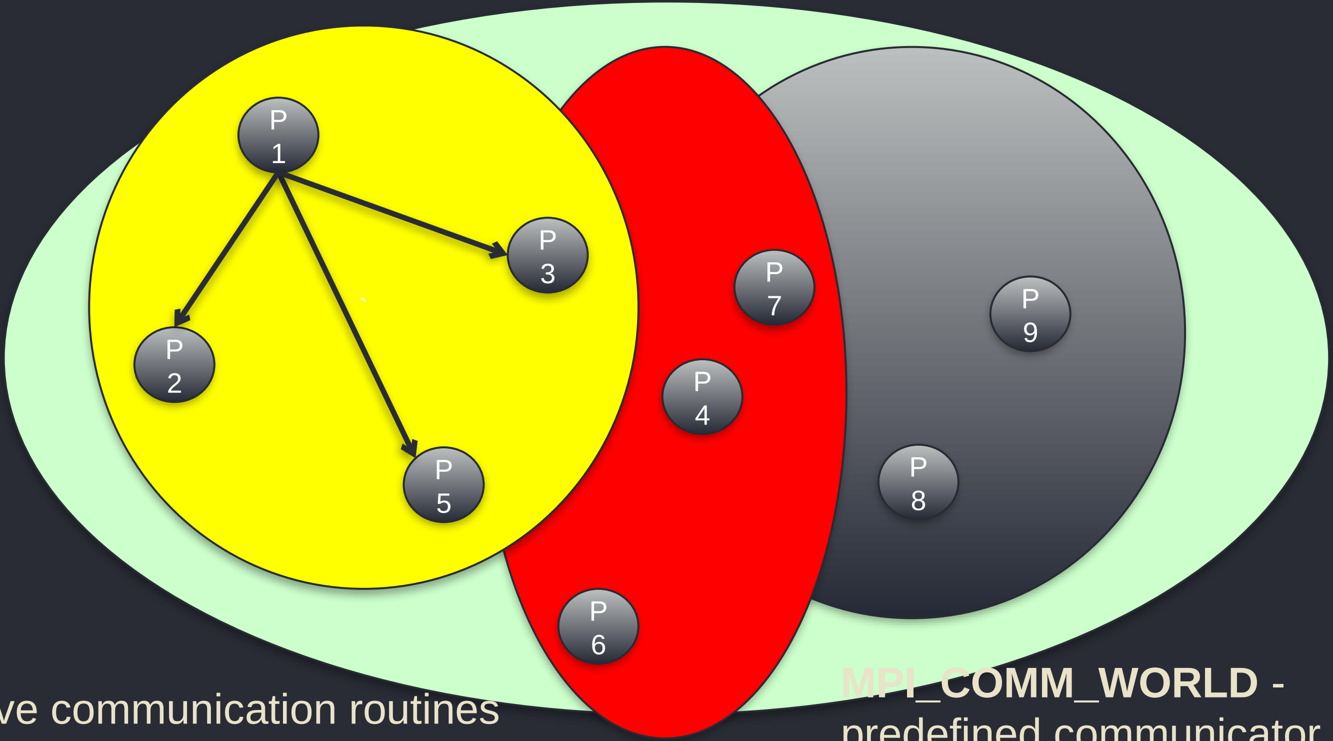
Send

Send Mode	Blocking Function	Nonblocking Function
Standard (will complete when buffer is available for use)	MPI_SEND	MPI_ISEND
Synchronous (will complete only until a matching receive has been posted and transfer has started)	MPI_SSEND	MPI_ISSEND
Ready (send starts only if a matching receive has been posted)	MPI_RSEND	MPI_IRSEND
Buffered (as soon as the user buffer is copied to the system buffer)	MPI_BSEND	MPI_IBSEND

6

Collective communications

Overview



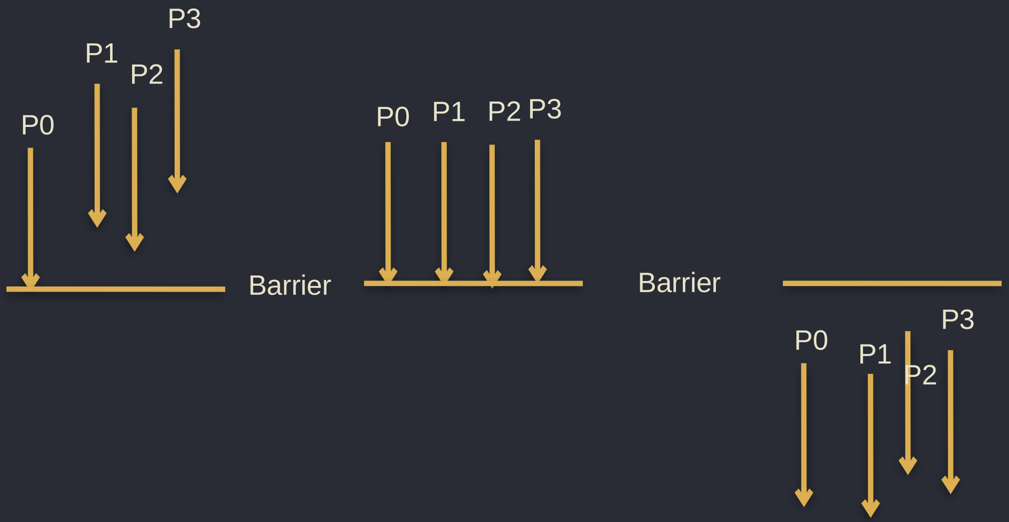
Collective communication routines transmit data among all processes in a group.

MPI_COMM_WORLD - predefined communicator that includes all MPI processes

Barrier

Stop processes until all processes within a communicator reach the barrier, which is useful for different cases, such as in measuring the performance

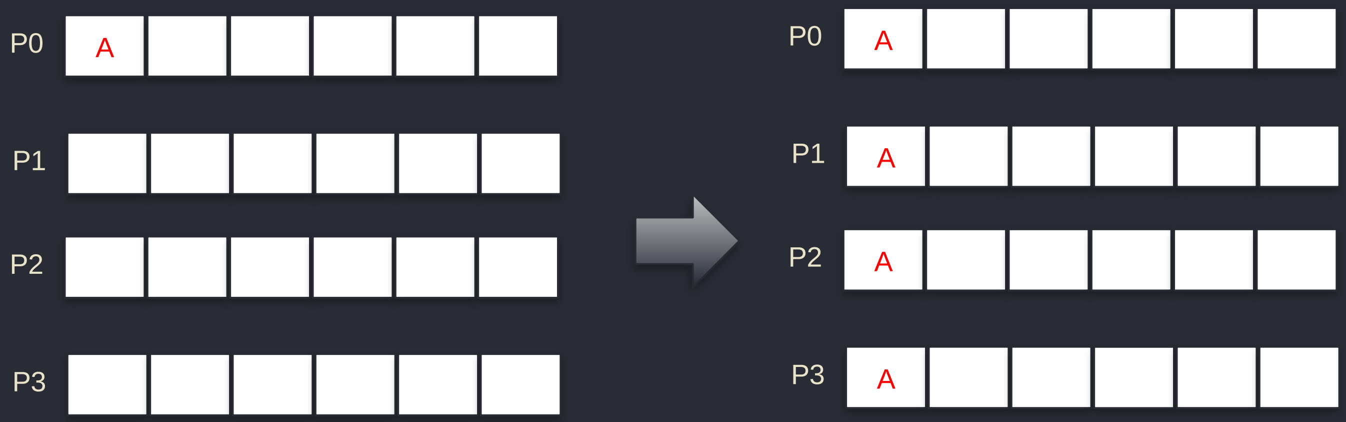
int MPI_Barrier (MPI_Comm comm)



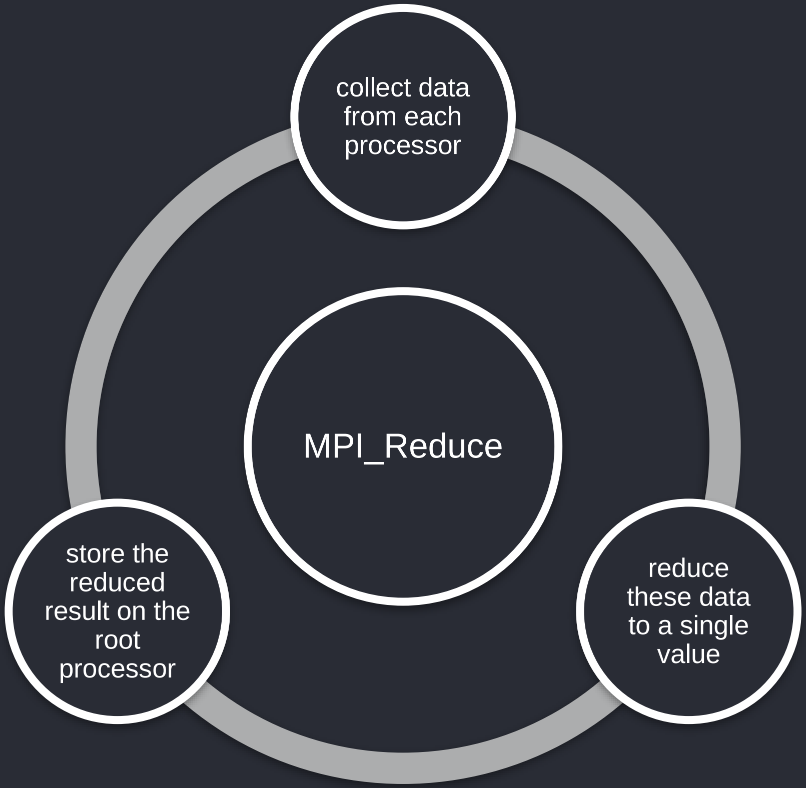
Broadcast

Enables to copy data from the memory of the root processor to the same memory locations for other processors in the communicator.

int MPI_Bcast (void* buffer, int count, MPI_Datatype datatype, int rank, MPI_Comm comm)



Reduce



MPI_Reduce

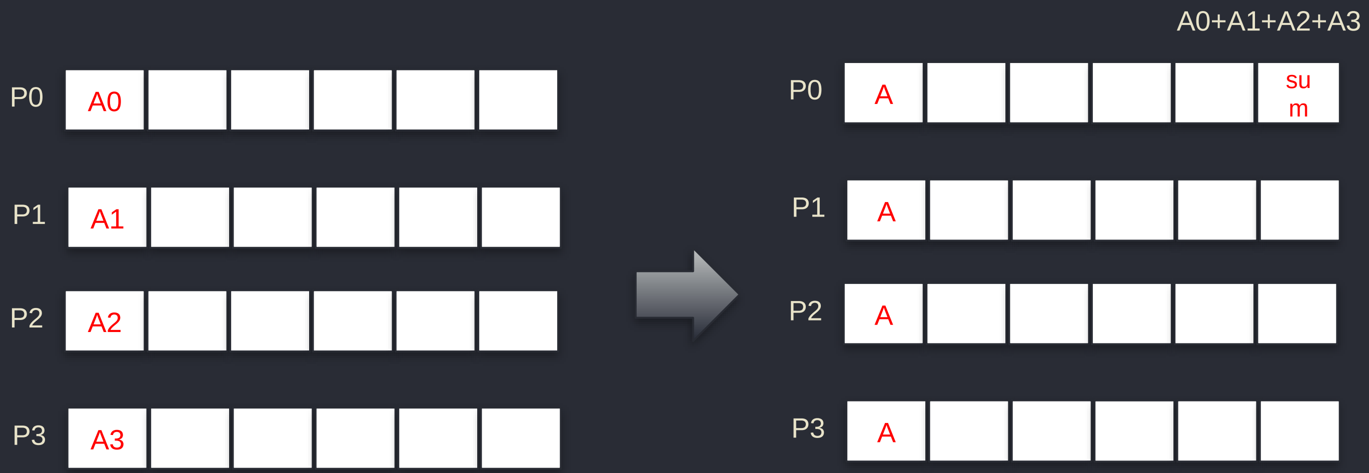
collect data from each processor

reduce these data to a single value

store the reduced result on the root processor

Reduce

MPI_REDUCE combines the elements provided in the send buffer, applies the specified operation (sum, min, max, ...), and returns the result to the receive buffer of the root process.



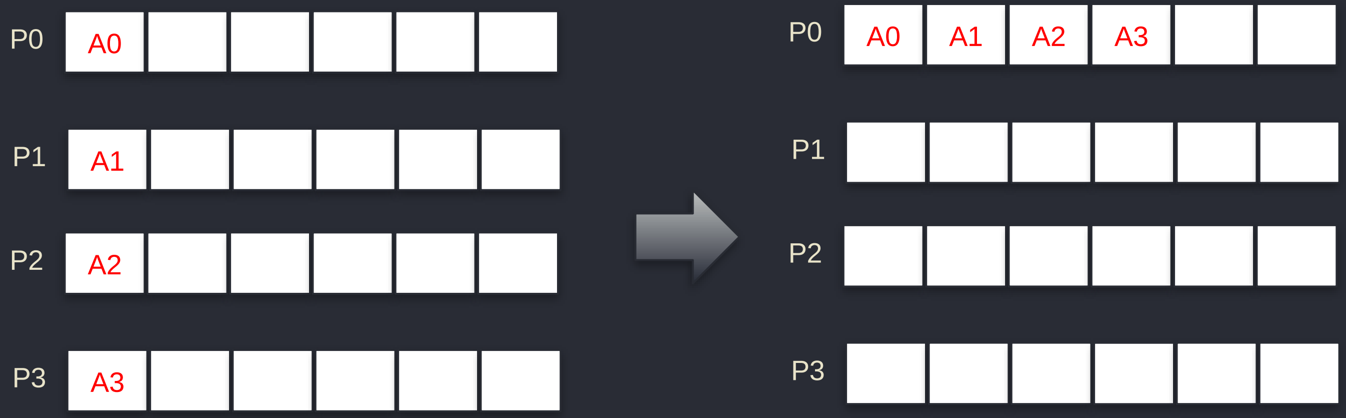
Reduce

Operation	Description
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bit-wise and
MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical xor
MPI_BXOR	logical xor
MPI_MINLOC	computes a global minimum and an index attached to the minimum value
MPI_MAXLOC	computes a global maximum and an index attached to the rank

Gather

Each process (including the root process) sends the contents of its send buffer to the root process. The root process receives the messages and stores them in rank order.

The gather also could be accomplished by each process calling MPI_SEND and the root process calling MPI_RECV ***N*** times to receive all of the messages.



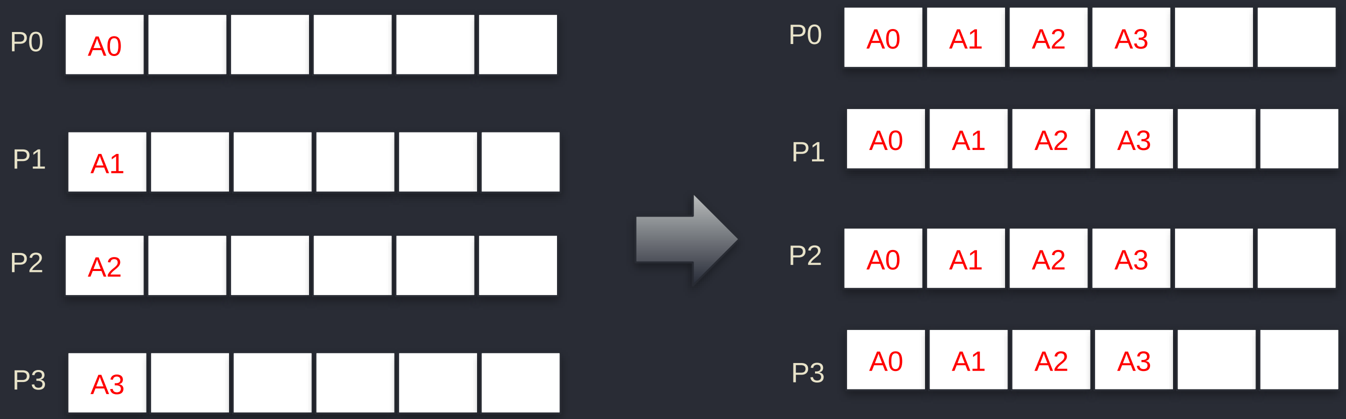
Gather

```
int MPI_Gather ( void* send_buffer, int send_count, MPI_datatype send_type, void* recv_buffer, int
recv_count, MPI_Datatype recv_type, int rank, MPI_Comm comm )
```

Parameter	In/Out	Description
send_buffer	in	starting address of send buffer
send_count	in	number of elements in send buffer
send_type	in	data type of send buffer elements
recv_buffer	out	starting address of receive buffer
recv_count	in	number of elements in receive buffer for a single receive
recv_type	in	data type of elements in receive buffer
recv_rank	in	rank of receiving process
comm	in	mpi communicator

AllGather

After the data are *gathered* into processor 0, you could then MPI_BCAST the gathered data to all of the other processors. It is more convenient and efficient to *gather* and *broadcast* with the single MPI_ALLGATHER operation.

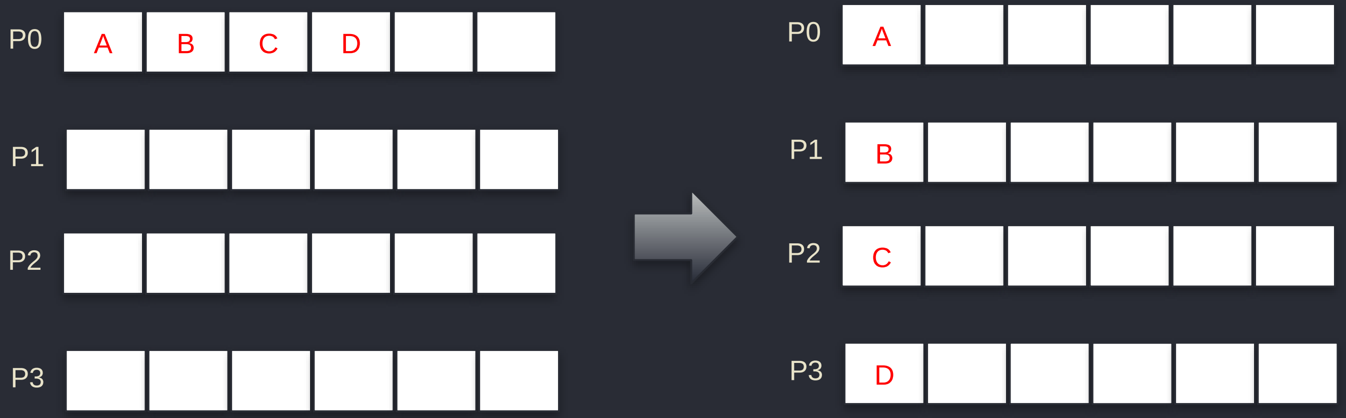


P0	A0					
P1	A1					
P2	A2					
P3	A3					

P0	A0	A1	A2	A3		
P1	A0	A1	A2	A3		
P2	A0	A1	A2	A3		
P3	A0	A1	A2	A3		

Scatter

The MPI_SCATTER routine is a *one-to-all* communication. Different data are sent from the root process to each process (in rank order). When MPI_SCATTER is called, the root process breaks up a set of contiguous memory locations into equal chunks and sends one chunk to each processor.



Summary

P0	P1	P2*	P3
a	b	c	d
a	b	c	d
		a,b,c,d	
a,b,c,d	e,f,g,h	i,j,k,l	m,n,o,p
		a	
SBuf	SBuf	SBuf	SBuf

Function
Gather
Allgather
Scatter
AlltoAll
Bcast
Memory

P0	P1	P2*	P3
		a,b,c,d	
a,b,c,d	a,b,c,d	a,b,c,d	a,b,c,d
a	b	c	d
a,e,i,m	b,f,j,n	c,g,k,o	d,h,l,p
a	a	a	a
RBuf	RBuf	RBuf	RBuf

Datatypes

MPI Datatype	**C Type**
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int

MPI Datatype	**C Type**
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	(none)

Datatypes

MPI Datatype	**C Type**
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int

MPI Datatype	**C Type**
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	(none)

Coll. Communications: example

```
#include <mpi.h>
#include <stdio.h>
int main (int argc, char *argv[]) {
int rank, buf;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if(rank == 0) { buf = 1; }
printf(”process %d: before Bcast buf is %d\n", rank, buf);
MPI_Bcast(&buf, 1, MPI_INT, 0, MPI_COMM_WORLD);
printf(”process %d: after Bcast buf is %d\n", rank, buf);
MPI_Finalize();
return 0;
}
```

Coll. Communications: example

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
int rank, buf, first, last, sum,i,n,result;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank== 0) N = atoi(argv[1]); MPI_Bcast(&N, 1, MPI_INT, 0,
MPI_COMM_WORLD);
first = N/numproc * rank + 1;
last = N/numproc * (rank+1);
for (i=first; i <= last; ++i) sum +=i; MPI_Reduce(&sum, &result, 1,
MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
if (rank==0) printf("Sum= %d\n", result);
MPI_Finalize();
return 0;
}
```

7

How to run MPI Job?

SLURM script

```
!/bin/bash -l
#SBATCH --job-name=mpi-job # Job name
#SBATCH --time=2:0:0 # Maximum runtime (1 hour)
#SBATCH --nodes=1 # Number of nodes requested
#SBATCH --ntasks-per-node=4 # Number of MPI tasks per node

mpiicc -o hello-mpi hello-mpi.c

mpirun -np 4 ./hello-mpi
```

Mixed MPI/OpenMP

```
!/bin/bash -l
#SBATCH --job-name=mpi-job # Job name
#SBATCH --time=2:0:0 # Maximum runtime (1 hour)
#SBATCH --nodes=2 # Number of nodes requested
#SBATCH --ntasks-per-node=1 # Number of MPI tasks per node
#SBATCH --cpus-per-task=4 # Number of CPU cores per task

mpiicc -o hello-mpi hello-mpi.c

mpirun -np 2 ./hello-mpi
```