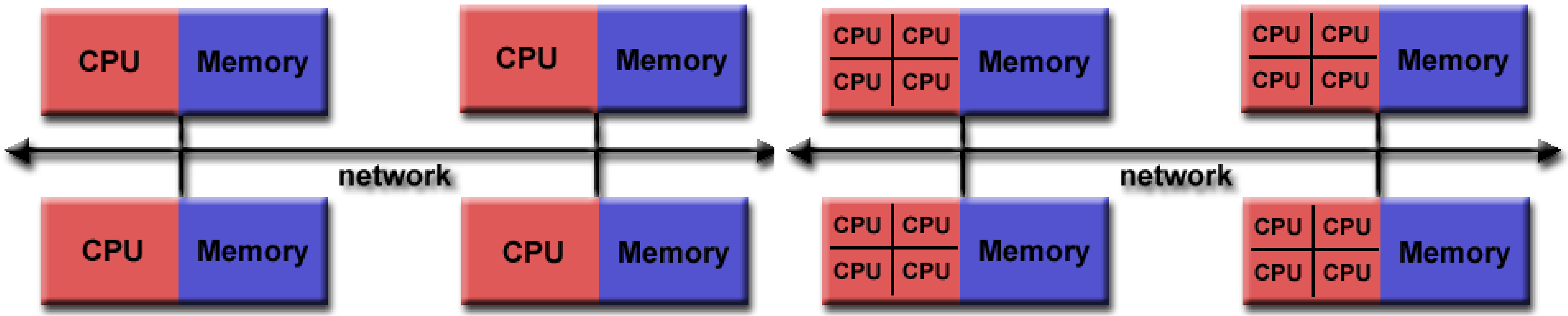
EPL372
Lab Exercise 5: (MPI Lecture) Introduction to MPI

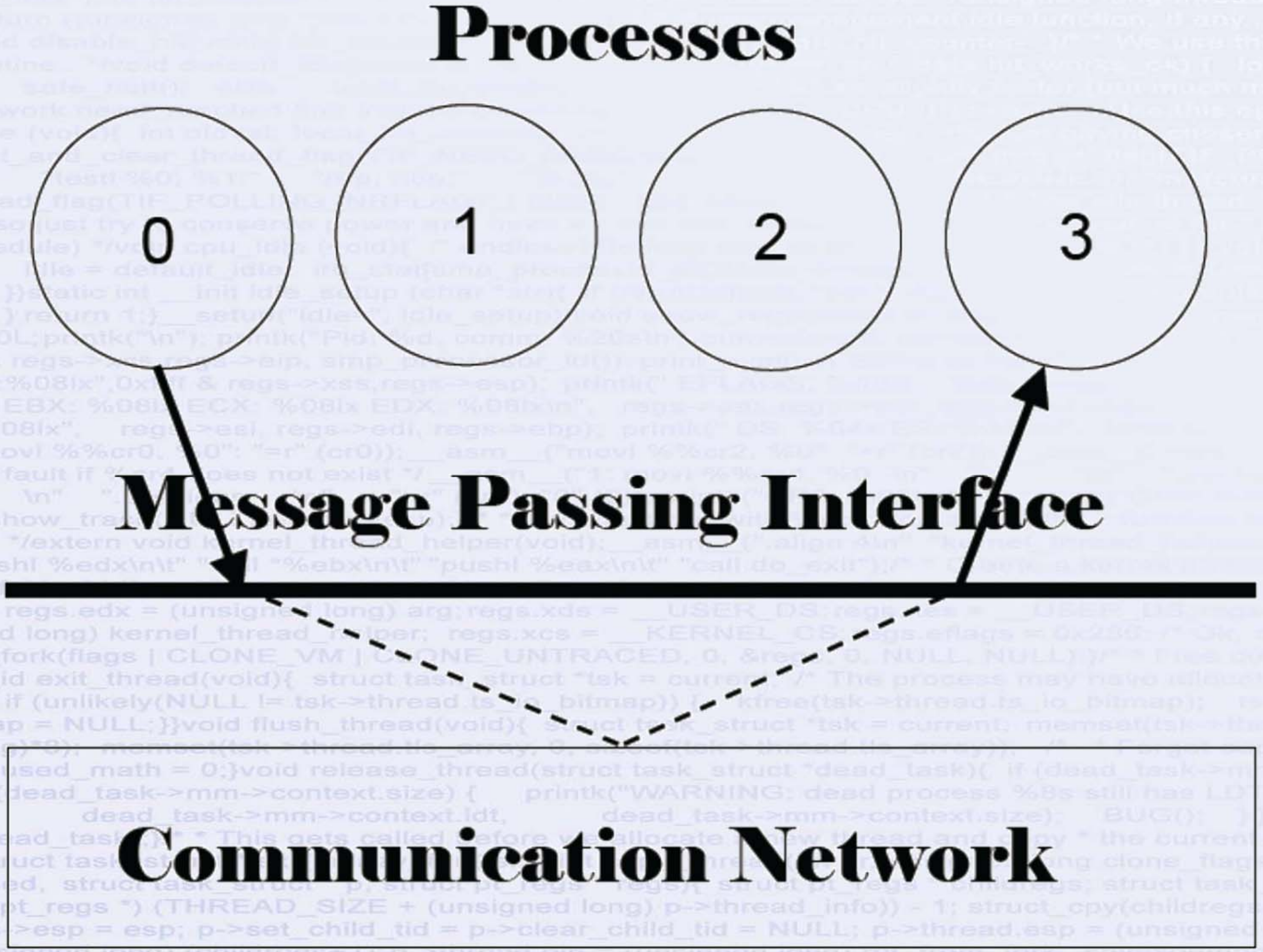
References:
https://computing.llnl.gov/tutorials/mpi/
https://computing.llnl.gov/tutorials/mpi/exercise.html
http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf

Why MPI?

- □ It is NOT a library - but rather the specification of what such a library should be.
- □ An MPI library is the most important piece of software in parallel programming.
- □ All the world’s largest supercomputers are programmed using MPI



Message Passing Programming Paradigm



Processes

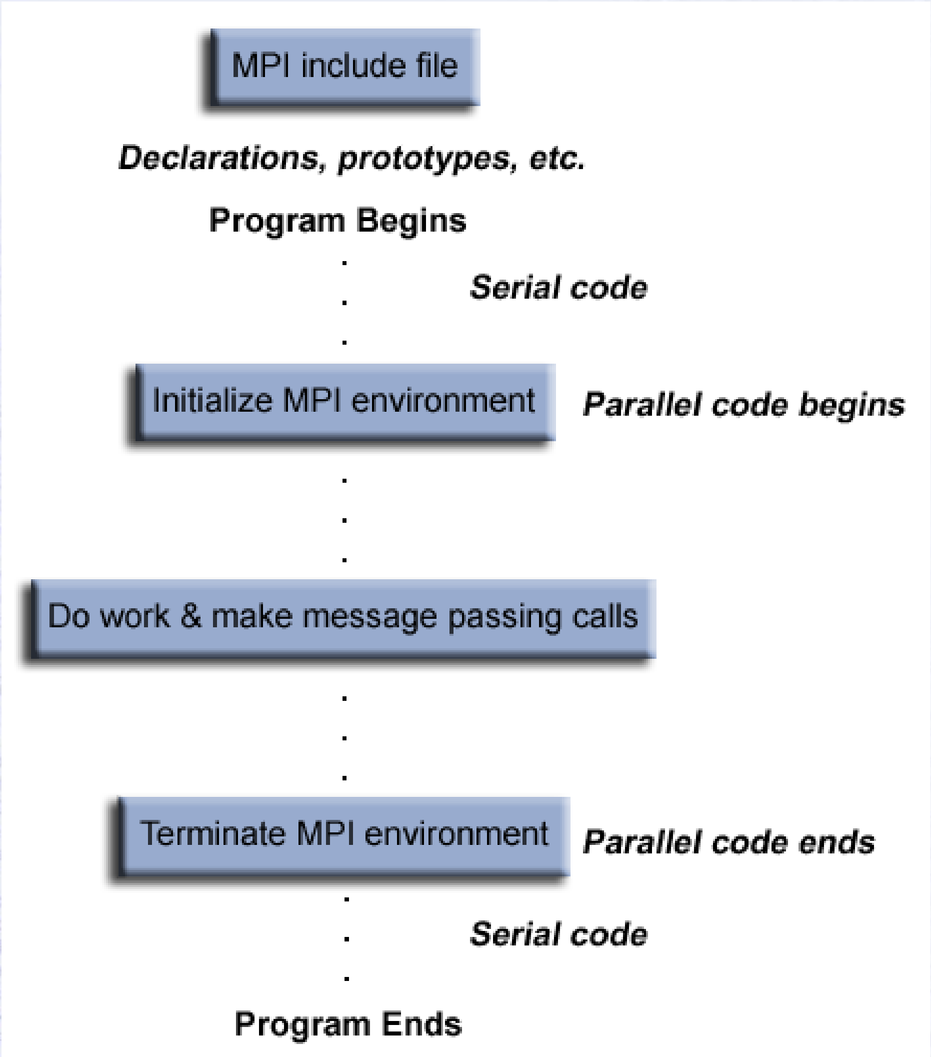
Message Passing Interface

Communication Network

Message Passing Programming Paradigm

- All variables are private
- Processes communicate with messages using:
  - Special subroutine calls
- Typically:
  - A single program is running on each processor

General MPI Program Structure



MPI include file

Declarations, prototypes, etc.

Program Begins

Serial code

Initialize MPI environment

Parallel code begins

Do work & make message passing calls

Terminate MPI environment

Parallel code ends

Serial code

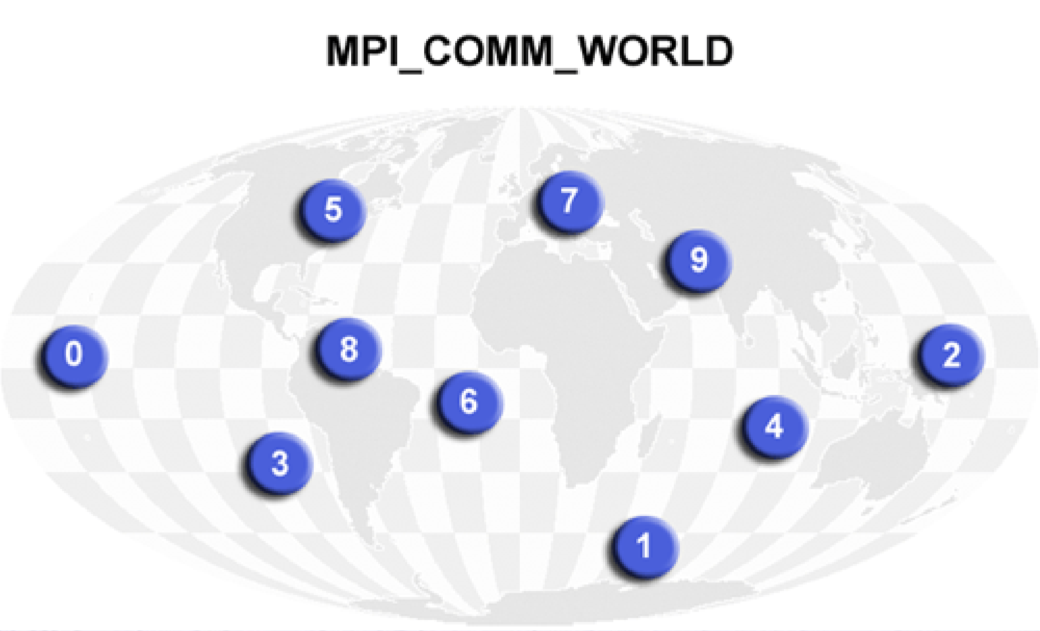
Program Ends

Communicators and Groups

MPI uses objects called **communicators** and **groups** to define which collection of processes may communicate with each other.

Most MPI routines require you to specify a communicator as an argument.

MPI_COMM_WORLD is the predefined communicator that includes all of your MPI processes.



MPI Syntax

- □ Header file:
  - ■ `#include <mpi.h>`
- □ Function Format:
  - ■ `error = MPI_Xxxxx(parameter, ...);`
  - ■ `MPI_Xxxxx(parameter, ...);`

Rank

Within a communicator, every process has its own **unique, integer identifier** assigned by the system when the process initializes. A rank is sometimes also called a "task ID". Ranks are contiguous and begin at zero (MASTER).

Rank is used by the programmer to specify the source and destination of messages. Often used conditionally by the application to control program execution (if rank=0 do this / if rank=1 do that).

```
MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
```

MPI_Init

Initializes the MPI execution environment. This function must be called in every MPI program, must be called before any other MPI functions and must be called only once in an MPI program.

For C programs, MPI_Init may be used to pass the command line arguments to all processes, although this is not required by the standard and is implementation dependent.

```
MPI_Init(&argc, &argv);
```

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(MPI_COMM_WORLD, &numtasks);
```

MPI_Get_processor_name

Returns the processor name. Also returns the length of the name. The buffer for "name" must be at least MPI_MAX_PROCESSOR_NAME characters in size. What is returned into "name" is implementation dependent - may not be the same as the output of the "hostname" or "host" shell commands.

```
MPI_Get_processor_name(hostname, &len);
```

MPI_Finalize

Terminates the MPI execution environment. This function should be the last MPI routine called in every MPI program - no other MPI routines may be called after it.

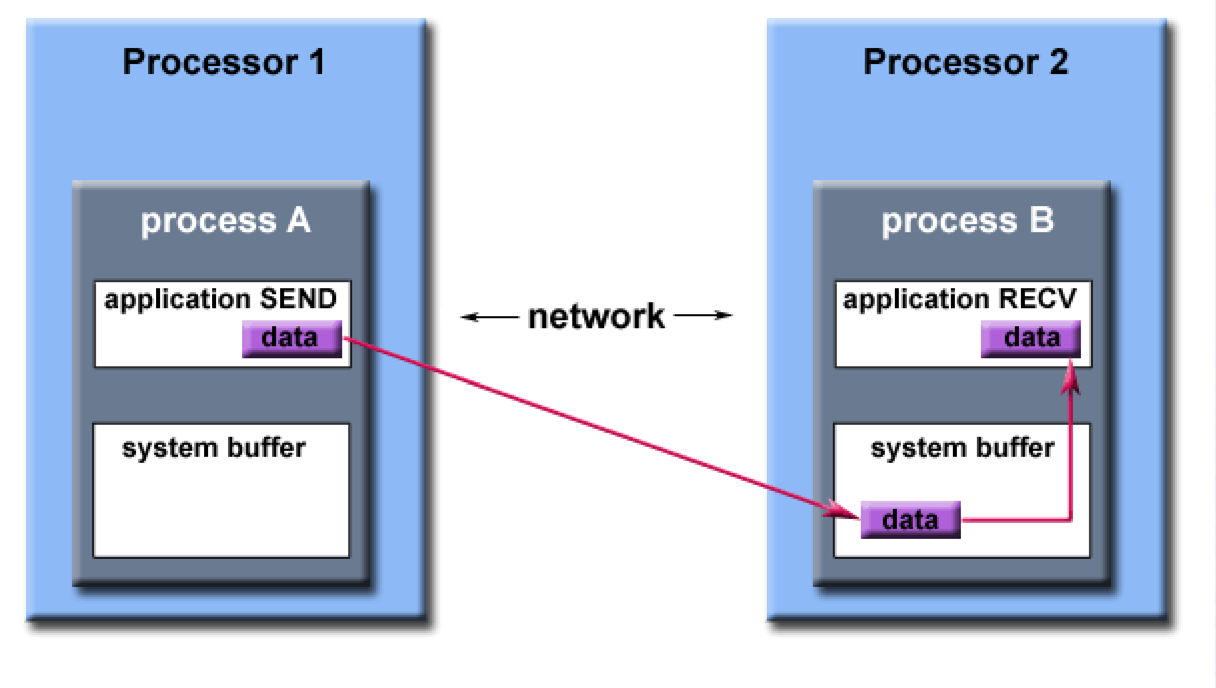
```
MPI_Finalize();
```

Environment Management Routines Example

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

int main(int argc, char *argv[]) {
int  numtasks, rank, len, rc;
char hostname[MPI_MAX_PROCESSOR_NAME];

rc = MPI_Init(&argc,&argv);
if (rc != MPI_SUCCESS) {
  printf ("Error starting MPI program. Terminating.\n");
  MPI_Abort(MPI_COMM_WORLD, rc);
  }

MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Get_processor_name(hostname, &len);
printf ("Number of tasks= %d My rank= %d Running on %s\n",
numtasks,rank,hostname);

/*******  do some work *******/

MPI_Finalize();
}
```

Sending Messages

A **system buffer** area is reserved to hold data in transit



Path of a message buffered at the receiving process

Point-to-Point Operations - Blocking vs. Non-blocking:

Blocking:

- A blocking send routine will only "return" after it is safe to modify the application buffer (your send data) for reuse. Safe means that modifications will not affect the data intended for the receive task. Safe does not imply that the data was actually received - it may very well be sitting in a system buffer.

- A blocking send can be synchronous which means there is handshaking occurring with the receive task to confirm a safe send.

- A blocking send can be asynchronous if a system buffer is used to hold the data for eventual delivery to the receive.

- A blocking receive only "returns" after the data has arrived and is ready for use by the program.

Point-to-Point Operations - Blocking vs. Non-blocking:

Non-blocking:

- Non-blocking send and receive routines behave similarly - they will return almost immediately. They do not wait for any communication events to complete, such as message copying from user memory to system buffer space or the actual arrival of message.
- Non-blocking operations simply "request" the MPI library to perform the operation when it is able. The user can not predict when that will happen.
- It is unsafe to modify the application buffer (your variable space) until you know for a fact the requested non-blocking operation was actually performed by the library. There are "wait" routines used to do this.
- Non-blocking communications are primarily used to overlap computation with communication and exploit possible performance gains.

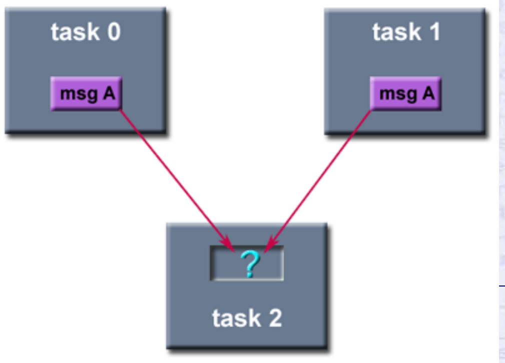
Order and Fairness

Order:

- MPI guarantees that messages will not overtake each other.
- If a sender sends two messages (Message 1 and Message 2) in succession to the same destination, and both match the same receive, the receive operation will receive Message 1 before Message 2.
- If a receiver posts two receives (Receive 1 and Receive 2), in succession, and both are looking for the same message, Receive 1 will receive the message before Receive 2.
- Order rules do not apply if there are multiple threads participating in the communication operations.

Fairness:

- MPI does not guarantee fairness - it's up to the programmer to prevent "operation starvation".
- Example: task 0 sends a message to task 2. However, task 1 sends a competing message that matches task 2's receive. Only one of the sends will complete.



MPI Message Passing Routine Arguments

Blocking sends	`MPI_Send(buffer,count,type,dest,tag,comm)`
Non-blocking sends	`MPI_Isend(buffer,count,type,dest,tag,comm,request)`
Blocking receive	`MPI_Recv(buffer,count,type,source,tag,comm,status)`
Non-blocking receive	`MPI_Irecv(buffer,count,type,source,tag,comm,request)`

Buffer

Program (application) address space that references the data that is to be sent or received. In most cases, this is simply the variable name that is be sent/received. For C programs, this argument is passed by reference and usually must be prepended with an ampersand: **`&var1`**

Data Count

Indicates the number of data elements of a particular type to be sent.

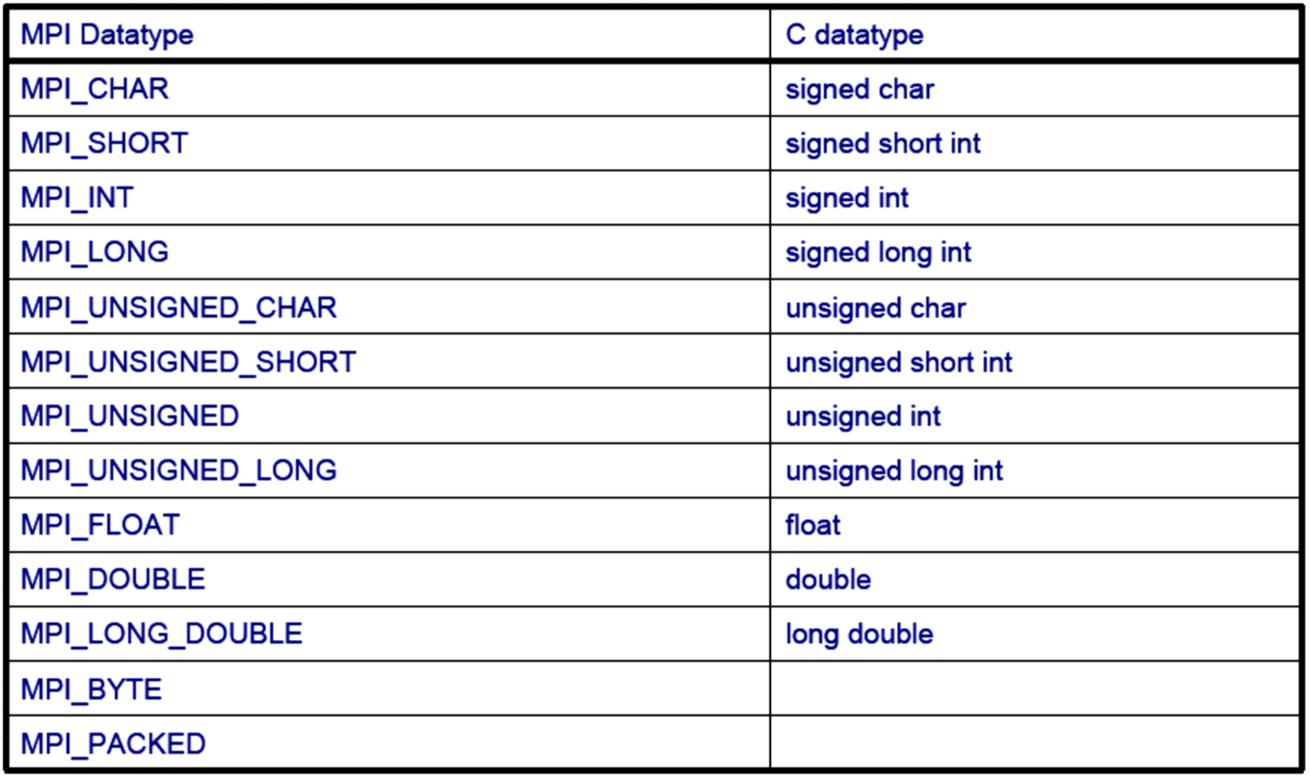
Data Type

For reasons of portability, MPI predefines its elementary data types. The table below lists those required by the standard. (i.e **`MPI_INT`**)

https://computing.llnl.gov/tutorials/mpi/#Routine_Arguments

MPI Message Passing Routine Arguments

Blocking sends	`MPI_Send(buffer,count,type,dest,tag,comm)`
Non-blocking sends	`MPI_Isend(buffer,count,type,dest,tag,comm,request)`
Blocking receive	`MPI_Recv(buffer,count,type,source,tag,comm,status)`
Non-blocking receive	`MPI_Irecv(buffer,count,type,source,tag,comm,request)`



MPI Datatype	C datatype
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_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

MPI Message Passing Routine Arguments

Blocking sends	`MPI_Send(buffer,count,type,dest,tag,comm)`
Non-blocking sends	`MPI_Isend(buffer,count,type,dest,tag,comm,request)`
Blocking receive	`MPI_Recv(buffer,count,type,source,tag,comm,status)`
Non-blocking receive	`MPI_Irecv(buffer,count,type,source,tag,comm,request)`

Destination

An argument to send routines that indicates the process where a message should be delivered. Specified as the rank of the receiving process.

Source

An argument to receive routines that indicates the originating process of the message. Specified as the rank of the sending process. This may be set to the wild card MPI_ANY_SOURCE to receive a message from any task.

MPI Message Passing Routine Arguments

Blocking sends	`MPI_Send(buffer,count,type,dest,tag,comm)`
Non-blocking sends	`MPI_Isend(buffer,count,type,dest,tag,comm,request)`
Blocking receive	`MPI_Recv(buffer,count,type,source,tag,comm,status)`
Non-blocking receive	`MPI_Irecv(buffer,count,type,source,tag,comm,request)`

Tag

Arbitrary non-negative integer assigned by the programmer to uniquely identify a message. Send and receive operations should match message tags. For a receive operation, the wild card MPI_ANY_TAG can be used to receive any message regardless of its tag. The MPI standard guarantees that integers 0-32767 can be used as tags, but most implementations allow a much larger range than this.

Communicator

Indicates the communication context, or set of processes for which the source or destination fields are valid. Unless the programmer is explicitly creating new communicators, the predefined communicator **MPI_COMM_WORLD** is usually used.

MPI Message Passing Routine Arguments

Blocking sends	**`MPI_Send(buffer,count,type,dest,tag,comm)`**
Non-blocking sends	**`MPI_Isend(buffer,count,type,dest,tag,comm,request)`**
Blocking receive	**`MPI_Recv(buffer,count,type,source,tag,comm,status)`**
Non-blocking receive	**`MPI_Irecv(buffer,count,type,source,tag,comm,request)`**

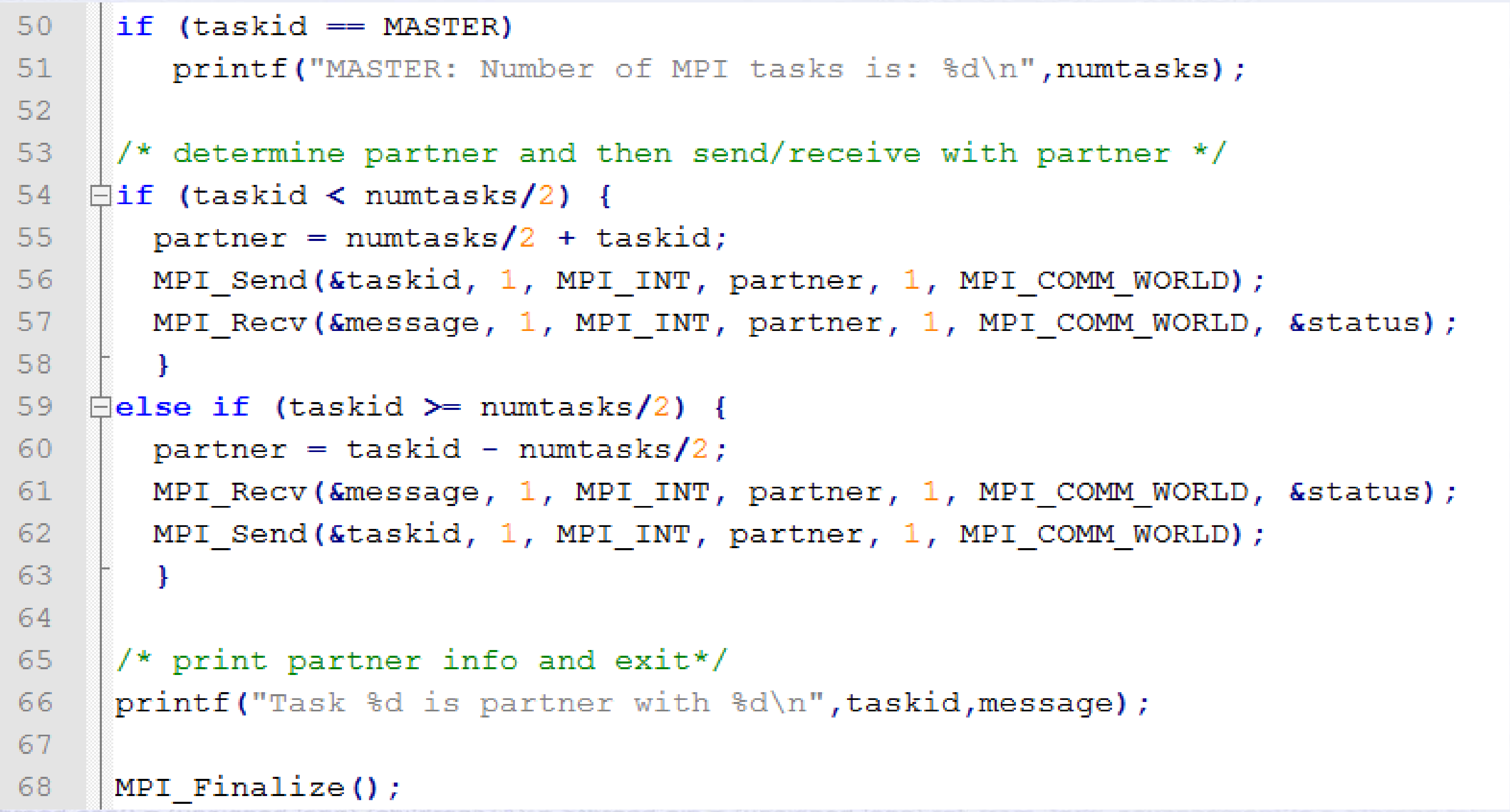
Status

For a receive operation, indicates the source of the message and the tag of the message. In C, this argument is a pointer to a predefined structure MPI_Status (ex. stat.MPI_SOURCE stat.MPI_TAG). Additionally, the actual number of bytes received are obtainable from Status via the MPI_Get_count routine.

Request

Used by non-blocking send and receive operations. Since non-blocking operations may return before the requested system buffer space is obtained, the system issues a unique "request number". The programmer uses this system assigned "handle" later (in a WAIT type routine) to determine completion of the non-blocking operation. In C, this argument is a pointer to a predefined structure MPI_Request.

Example of Blocked Send/Receive



```c
if (taskid == MASTER)
   printf("MASTER: Number of MPI tasks is: %d\n",numtasks);

/* determine partner and then send/receive with partner */
if (taskid < numtasks/2) {
  partner = numtasks/2 + taskid;
  MPI_Send(&taskid, 1, MPI_INT, partner, 1, MPI_COMM_WORLD);
  MPI_Recv(&message, 1, MPI_INT, partner, 1, MPI_COMM_WORLD, &status);
  }
else if (taskid >= numtasks/2) {
  partner = taskid - numtasks/2;
  MPI_Recv(&message, 1, MPI_INT, partner, 1, MPI_COMM_WORLD, &status);
  MPI_Send(&taskid, 1, MPI_INT, partner, 1, MPI_COMM_WORLD);
  }

/* print partner info and exit*/
printf("Task %d is partner with %d\n",taskid,message);

MPI_Finalize();
```

MPI_Waitall

MPI_Waitall - Waits for all given communications to complete

```
int MPI_Waitall(
     int count,
     MPI_Request array_of_requests[],
     MPI_Status array_of_statuses[] )
```

INPUT PARAMETERS

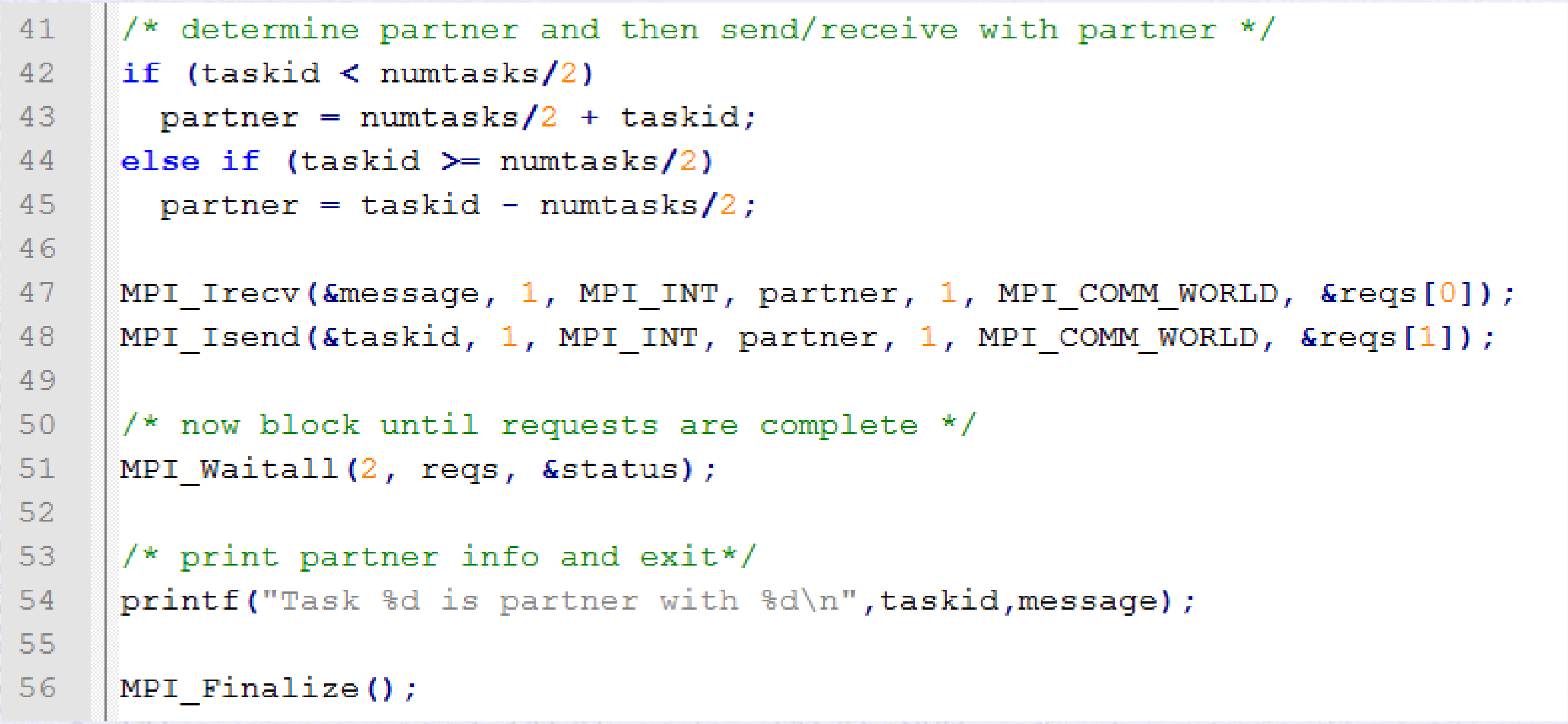
count - lists length (integer)

array_of_requests - array of requests (array of handles)

OUTPUT PARAMETER

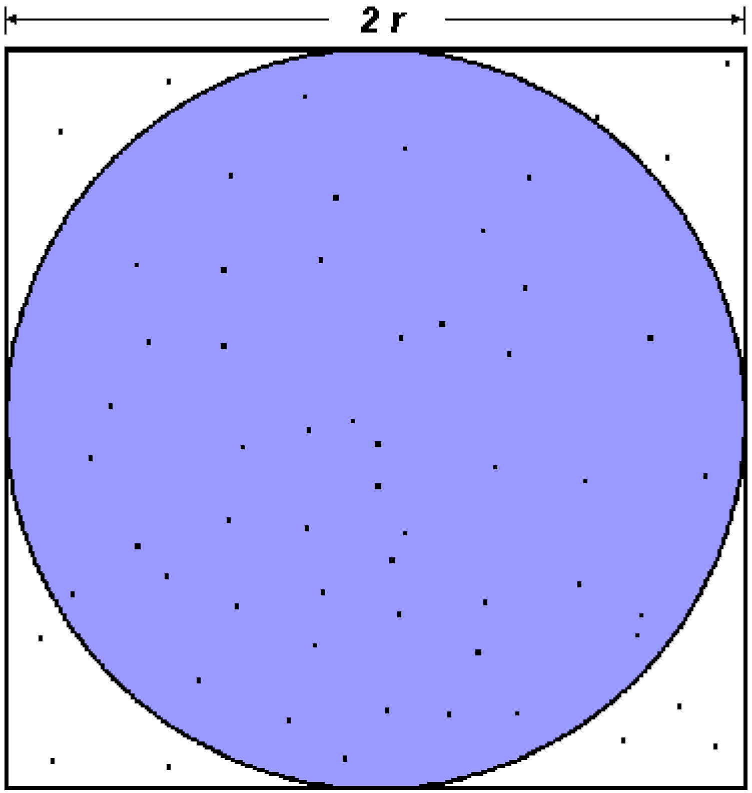
array_of_statuses - array of status objects (array of Status). May be MPI_STATUSES_IGNORE

Example of Non-Blocked Send/Receive



```c
/* determine partner and then send/receive with partner */
if (taskid < numtasks/2)
  partner = numtasks/2 + taskid;
else if (taskid >= numtasks/2)
  partner = taskid - numtasks/2;

MPI_Irecv(&message, 1, MPI_INT, partner, 1, MPI_COMM_WORLD, &reqs[0]);
MPI_Isend(&taskid, 1, MPI_INT, partner, 1, MPI_COMM_WORLD, &reqs[1]);

/* now block until requests are complete */
MPI_Waitall(2, reqs, &status);

/* print partner info and exit*/
printf("Task %d is partner with %d\n",taskid,message);

MPI_Finalize();
```

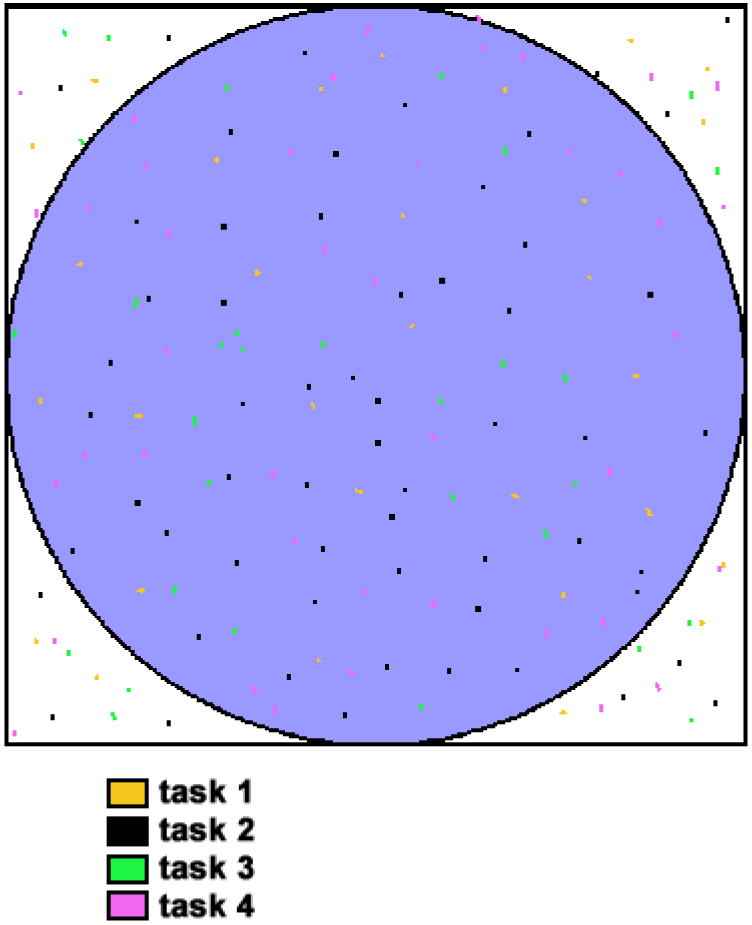
Calculating the Value of Pi



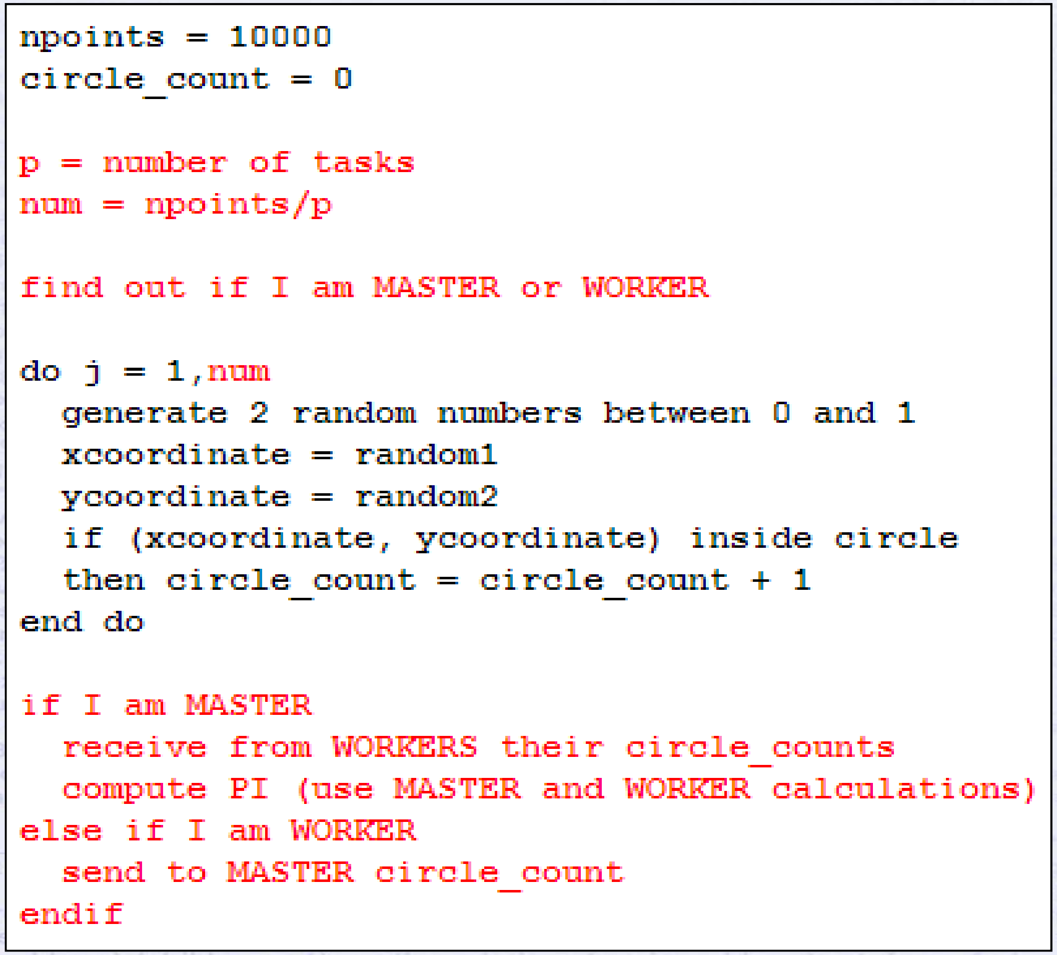
$$A_S = (2r)^2 = 4r^2$$

$$A_C = \pi r^2$$

$$\pi = 4 \text{ x } \frac{A_C}{A_S}$$

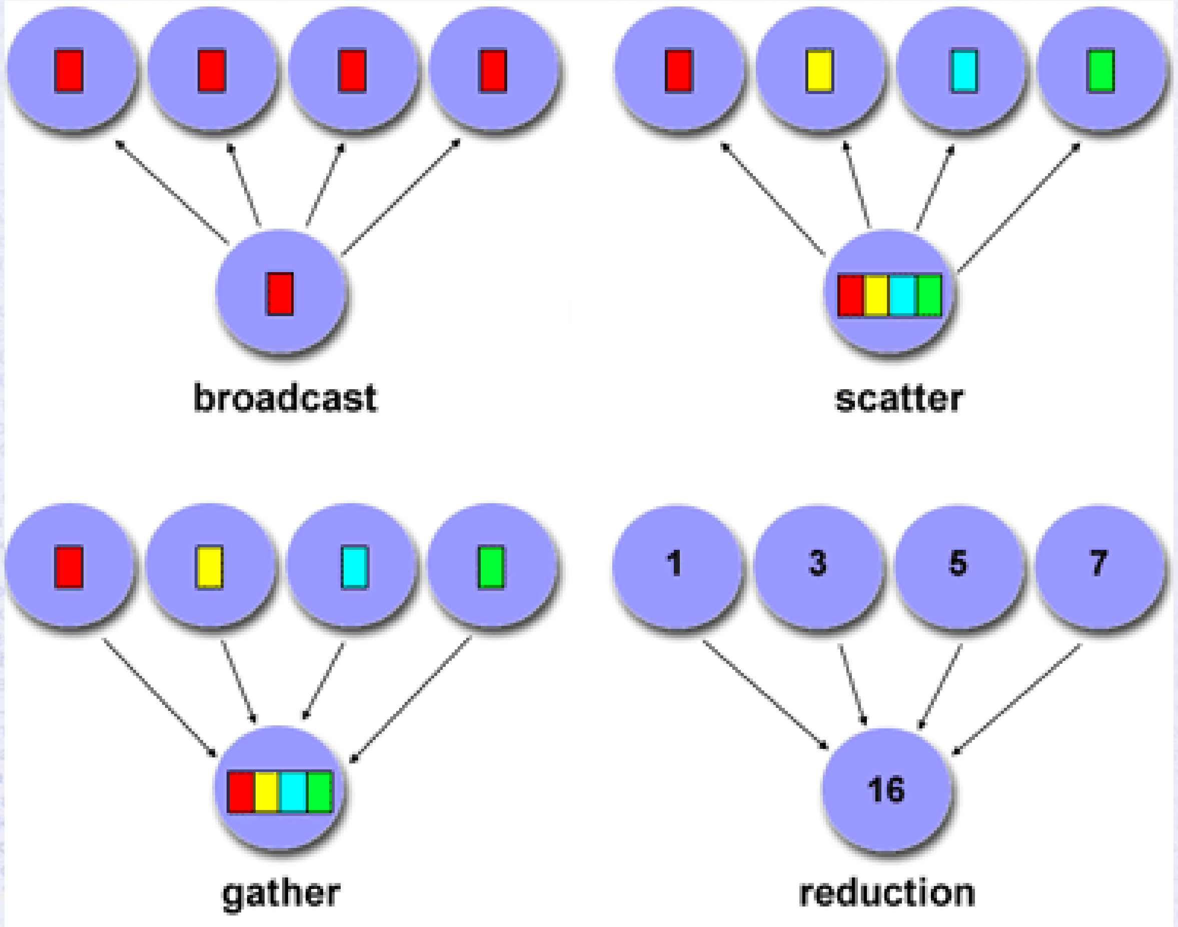


Pseudo code solution



```
npoints = 10000
circle_count = 0

p = number of tasks
num = npoints/p

find out if I am MASTER or WORKER

do j = 1,num
  generate 2 random numbers between 0 and 1
  xcoordinate = random1
  ycoordinate = random2
  if (xcoordinate, ycoordinate) inside circle
  then circle_count = circle_count + 1
end do

if I am MASTER
  receive from WORKERS their circle_counts
  compute PI (use MASTER and WORKER calculations)
else if I am WORKER
  send to MASTER circle_count
endif
```

Collective Communication Routines



https://computing.llnl.gov/tutorials/mpi/

MPI_Barrier

Synchronization operation. Creates a barrier synchronization in a group. Each task, when reaching the MPI_Barrier call, blocks until all tasks in the group reach the same MPI_Barrier call. Then all tasks are free to proceed.

```
MPI_Barrier (MPI_COMM_WORLD)
```

https://www.open-mpi.org/doc/v1.8/man3/MPI_Barrier.3.php

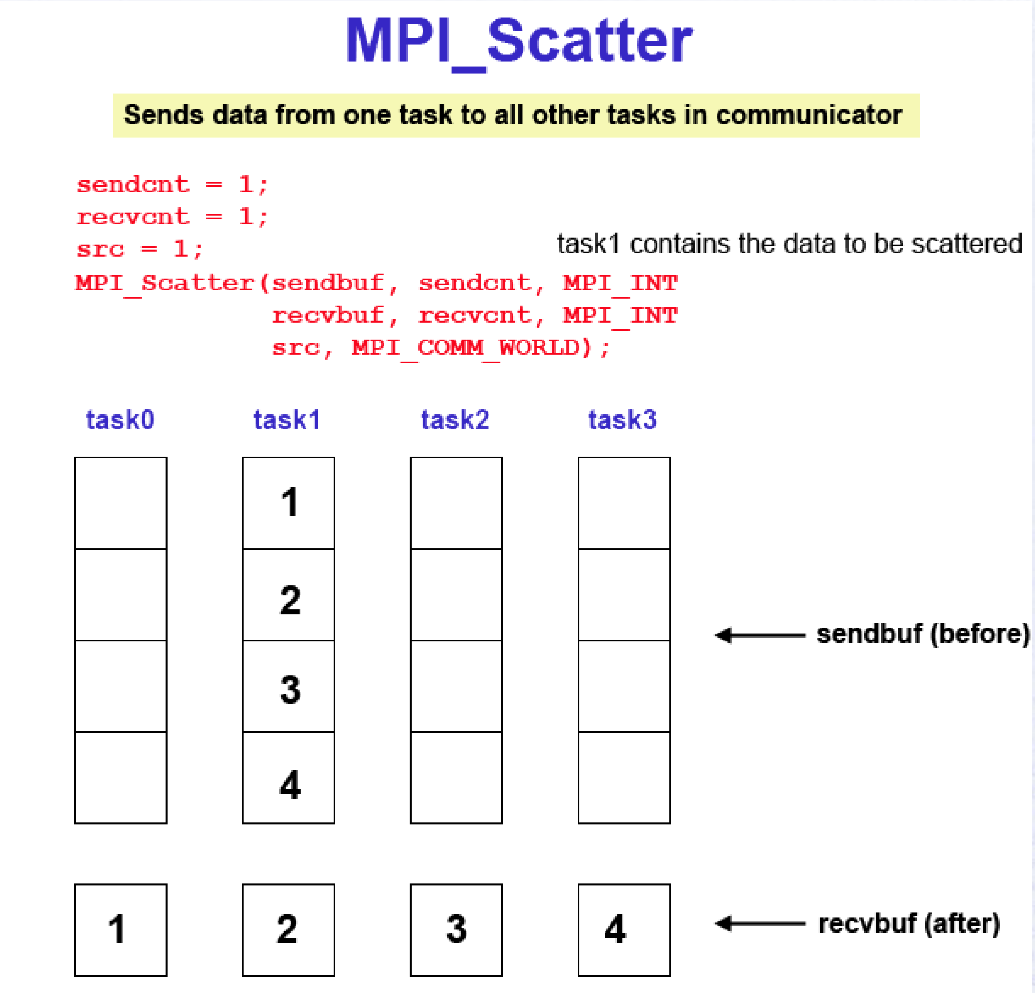
Collective Communication Routines

MPI_Scatter

Data movement operation. Distributes distinct messages from a single source task to each task in the group.

```
MPI_Scatter (&sendbuf,sendcnt,sendtype,&recvbuf,
recvcnt,recvtype,root,comm)
```

MPI_Scatter



MPI_Scatter

Sends data from one task to all other tasks in communicator

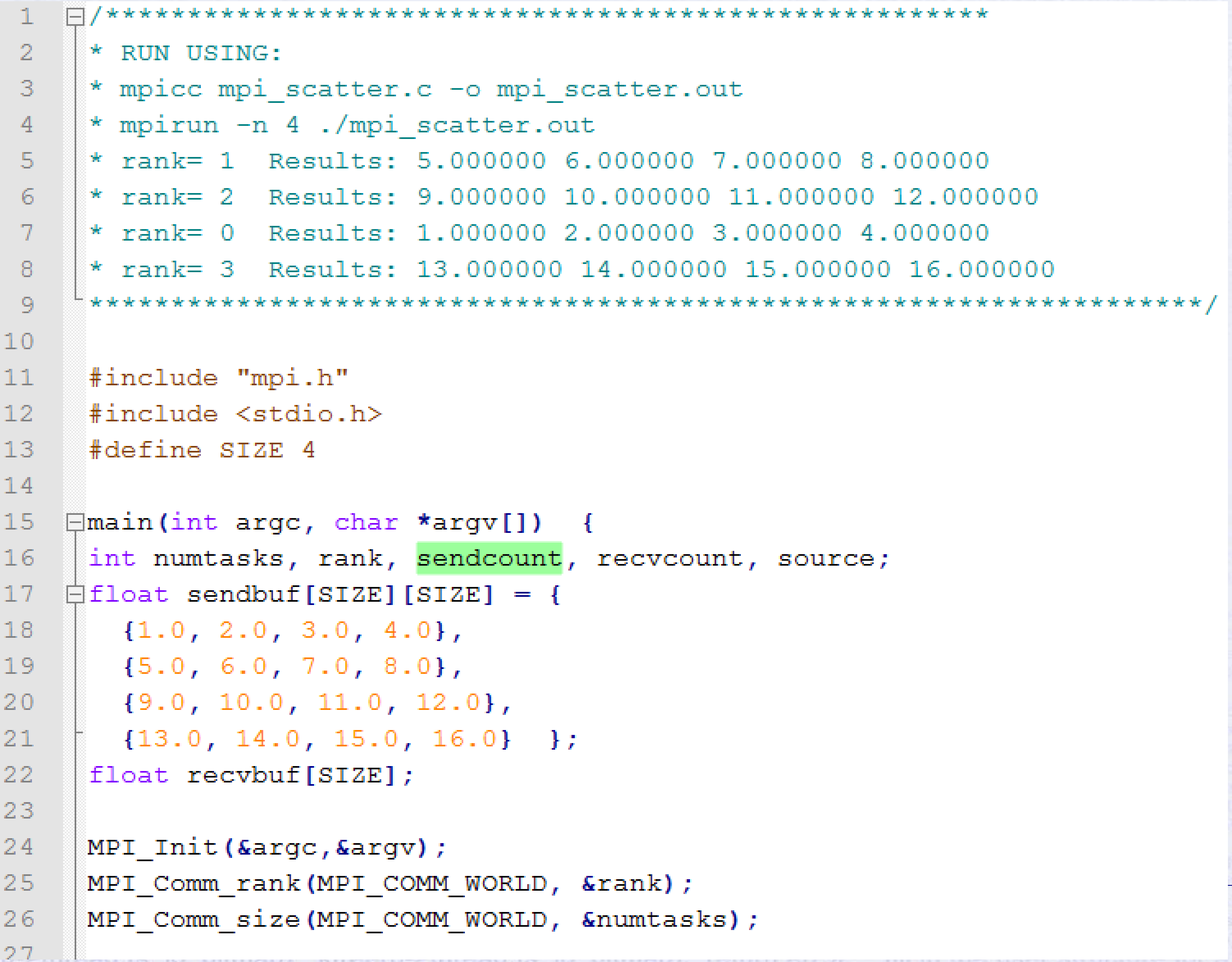
```
sendcnt = 1;
recvcnt = 1;
src = 1;
MPI_Scatter(sendbuf, sendcnt, MPI_INT
            recvbuf, recvcnt, MPI_INT
            src, MPI_COMM_WORLD);
```

task1 contains the data to be scattered

https://computing.llnl.gov/tutorials/mpi/

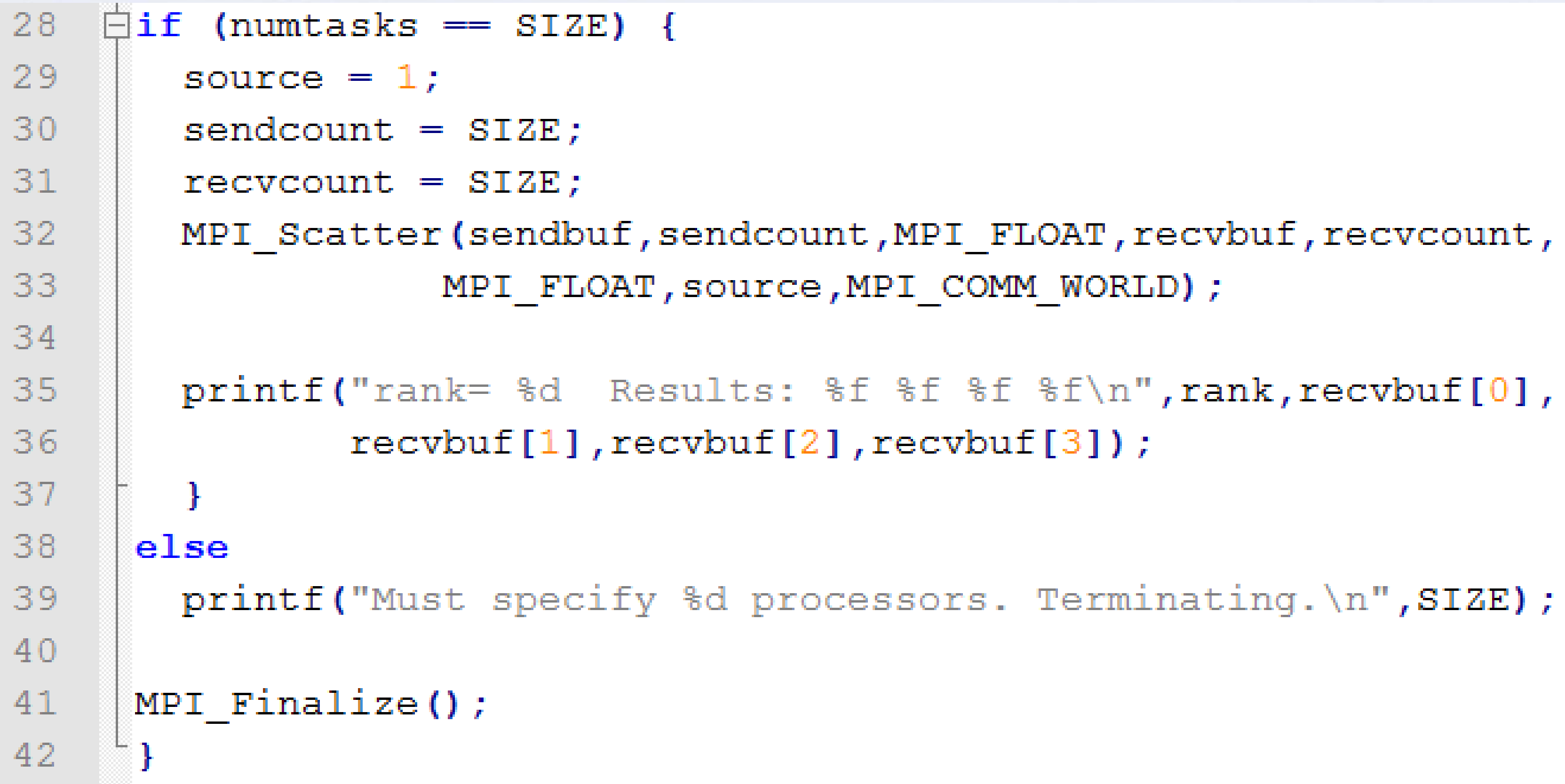
https://www.open-mpi.org/doc/v1.8/man3/MPI_Scatter.3.php

MPI_Scatter Example



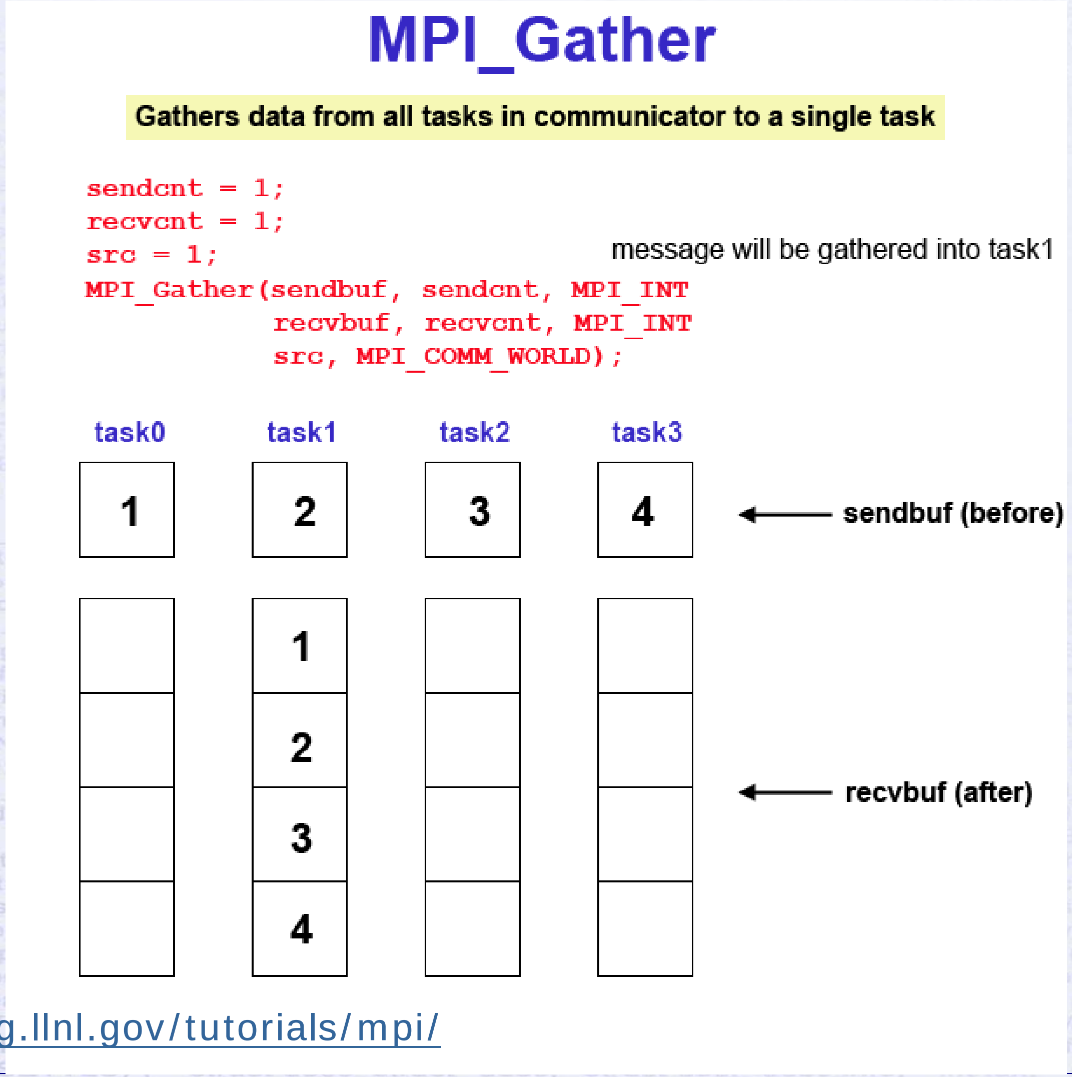
```
/*******************************************************
* RUN USING:
* mpicc mpi_scatter.c -o mpi_scatter.out
* mpirun -n 4 ./mpi_scatter.out
* rank= 1  Results: 5.000000 6.000000 7.000000 8.000000
* rank= 2  Results: 9.000000 10.000000 11.000000 12.000000
* rank= 0  Results: 1.000000 2.000000 3.000000 4.000000
* rank= 3  Results: 13.000000 14.000000 15.000000 16.000000
********************************************************************/

#include "mpi.h"
#include <stdio.h>
#define SIZE 4

main(int argc, char *argv[])  {
int numtasks, rank, sendcount, recvcount, source;
float sendbuf[SIZE][SIZE] = {
  {1.0, 2.0, 3.0, 4.0},
  {5.0, 6.0, 7.0, 8.0},
  {9.0, 10.0, 11.0, 12.0},
  {13.0, 14.0, 15.0, 16.0}  };
float recvbuf[SIZE];

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
```

MPI_Scatter Example



```
if (numtasks == SIZE) {
  source = 1;
  sendcount = SIZE;
  recvcount = SIZE;
  MPI_Scatter(sendbuf,sendcount,MPI_FLOAT,recvbuf,recvcount,
             MPI_FLOAT,source,MPI_COMM_WORLD);

  printf("rank= %d  Results: %f %f %f %f\n",rank,recvbuf[0],
         recvbuf[1],recvbuf[2],recvbuf[3]);
  }
else
  printf("Must specify %d processors. Terminating.\n",SIZE);

MPI_Finalize();
}
```

MPI_Gather



MPI_Gather

Gathers data from all tasks in communicator to a single task

```
sendcnt = 1;
recvcnt = 1;
src = 1;
MPI_Gather(sendbuf, sendcnt, MPI_INT
           recvbuf, recvcnt, MPI_INT
           src, MPI_COMM_WORLD);
```

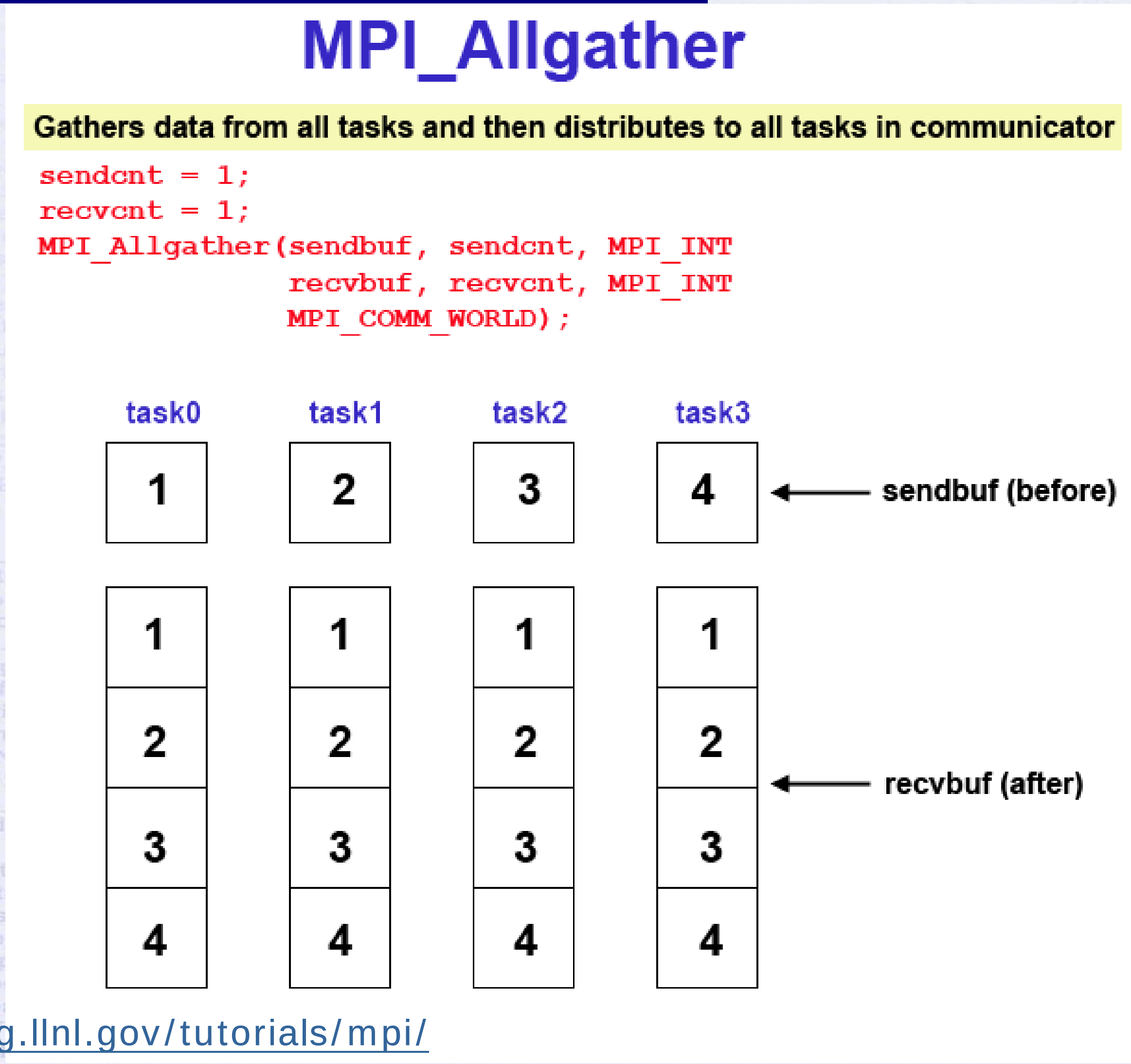
message will be gathered into task1

	task0	task1	task2	task3
sendbuf (before)	1	2	3	4
recvbuf (after)		1		
		2		
		3		
		4		

https://computing.llnl.gov/tutorials/mpi/

https://www.open-mpi.org/doc/v1.8/man3/MPI_Gather.3.php

MPI_Allgather



MPI_Allgather

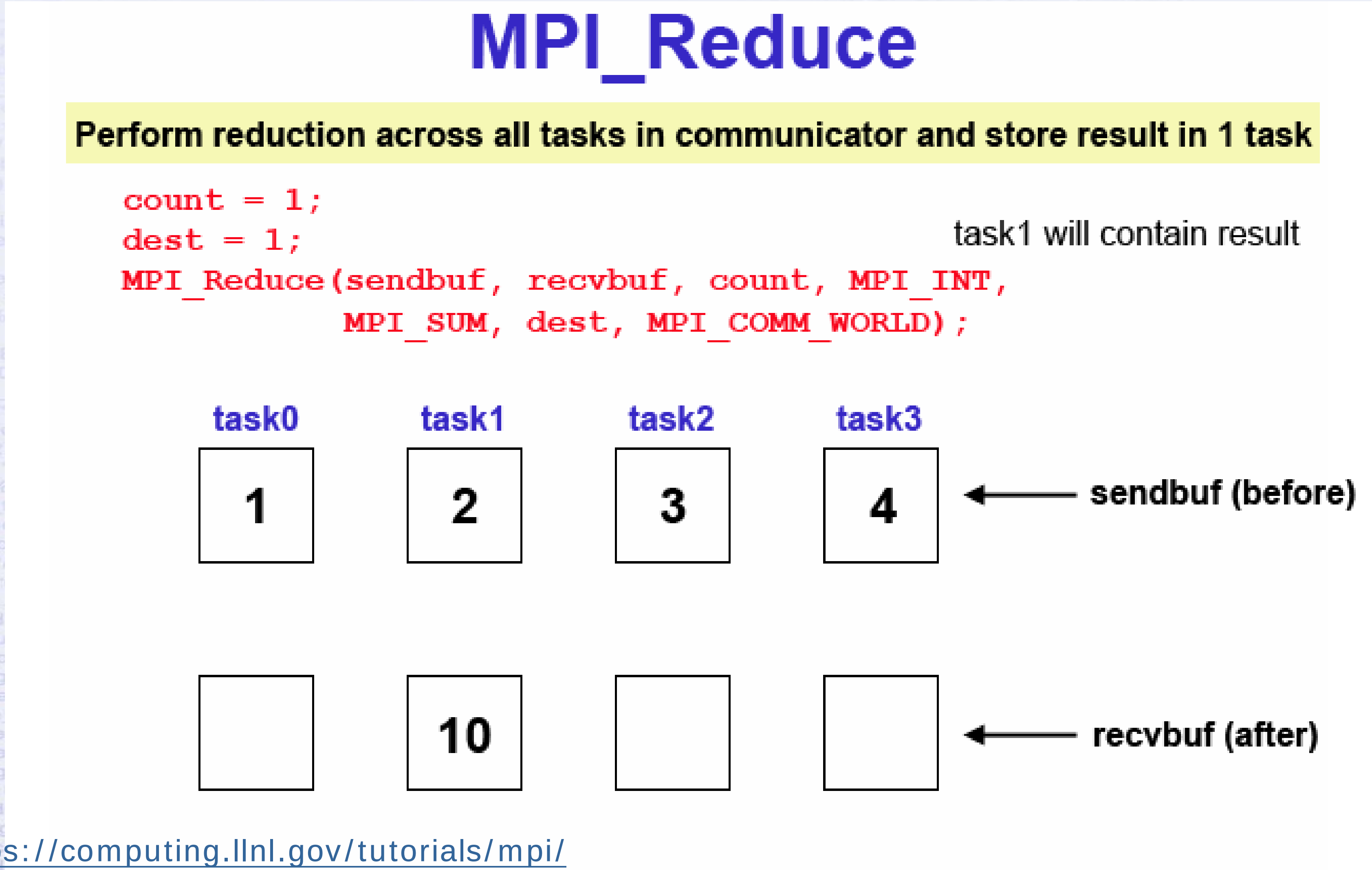
Gathers data from all tasks and then distributes to all tasks in communicator

```
sendcnt = 1;
recvcnt = 1;
MPI_Allgather(sendbuf, sendcnt, MPI_INT
              recvbuf, recvcnt, MPI_INT
              MPI_COMM_WORLD);
```

https://computing.llnl.gov/tutorials/mpi/

https://www.open-mpi.org/doc/v1.8/man3/MPI_Allgather.3.php

MPI_Reduce



MPI_Reduce

Perform reduction across all tasks in communicator and store result in 1 task

```
count = 1;
dest = 1;
MPI_Reduce(sendbuf, recvbuf, count, MPI_INT,
           MPI_SUM, dest, MPI_COMM_WORLD);
```

task1 will contain result

	task0	task1	task2	task3
sendbuf (before)	1	2	3	4
recvbuf (after)		10		

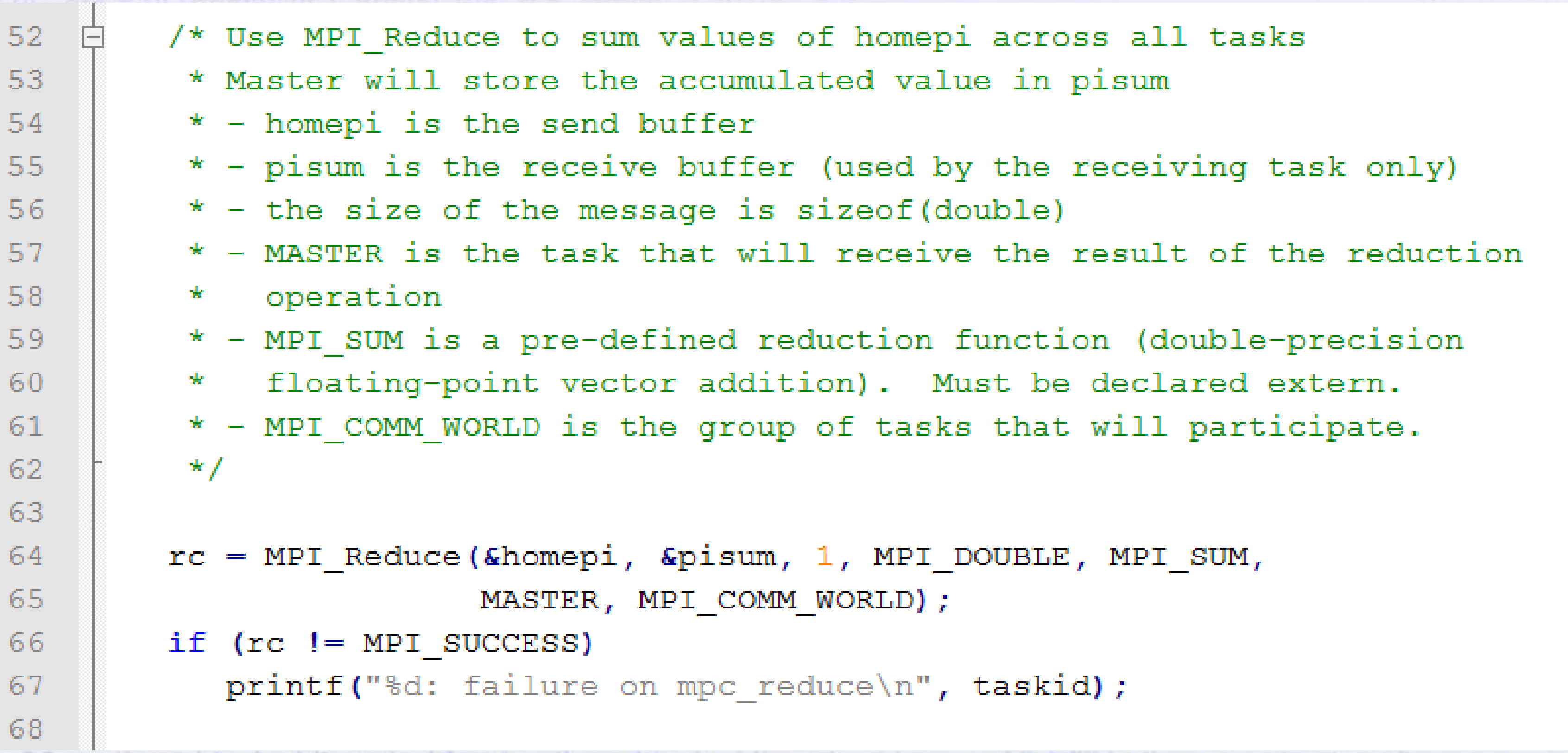
https://computing.llnl.gov/tutorials/mpi/

https://www.open-mpi.org/doc/v1.8/man3/MPI_Reduce.3.php

MPI_Reduce Example

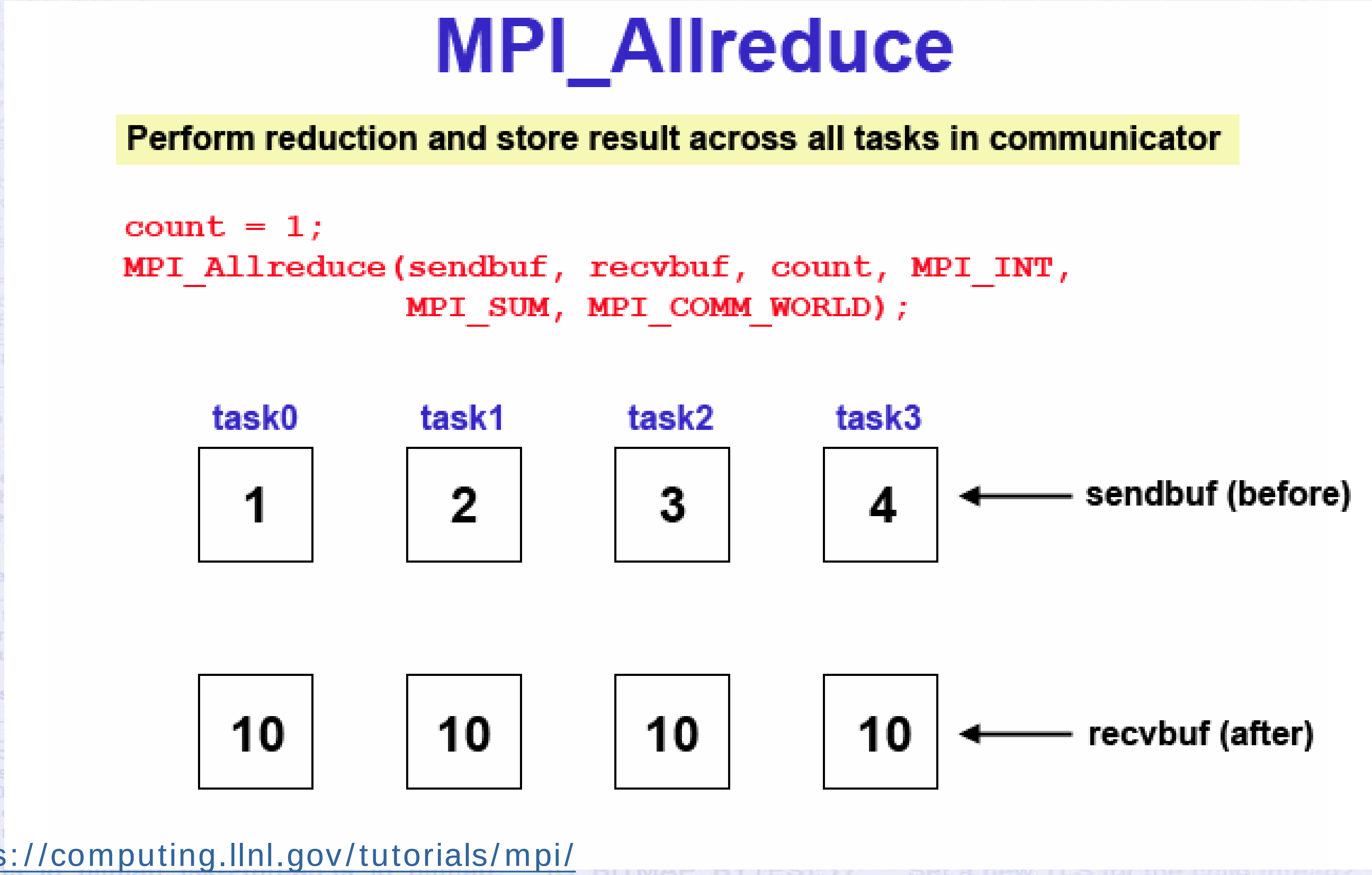
Collective computation operation. Applies a reduction operation on all tasks in the group and places the result in one task.

```
MPI_Reduce (&sendbuf,&recvbuf,count,datatype,op,root,comm)
```



```
/* Use MPI_Reduce to sum values of homepi across all tasks
 * Master will store the accumulated value in pisum
 * - homepi is the send buffer
 * - pisum is the receive buffer (used by the receiving task only)
 * - the size of the message is sizeof(double)
 * - MASTER is the task that will receive the result of the reduction
 *   operation
 * - MPI_SUM is a pre-defined reduction function (double-precision
 *   floating-point vector addition).  Must be declared extern.
 * - MPI_COMM_WORLD is the group of tasks that will participate.
 */

rc = MPI_Reduce(&homepi, &pisum, 1, MPI_DOUBLE, MPI_SUM,
                MASTER, MPI_COMM_WORLD);
if (rc != MPI_SUCCESS)
   printf("%d: failure on mpc_reduce\n", taskid);
```

MPI_AllReduce



MPI_Allreduce

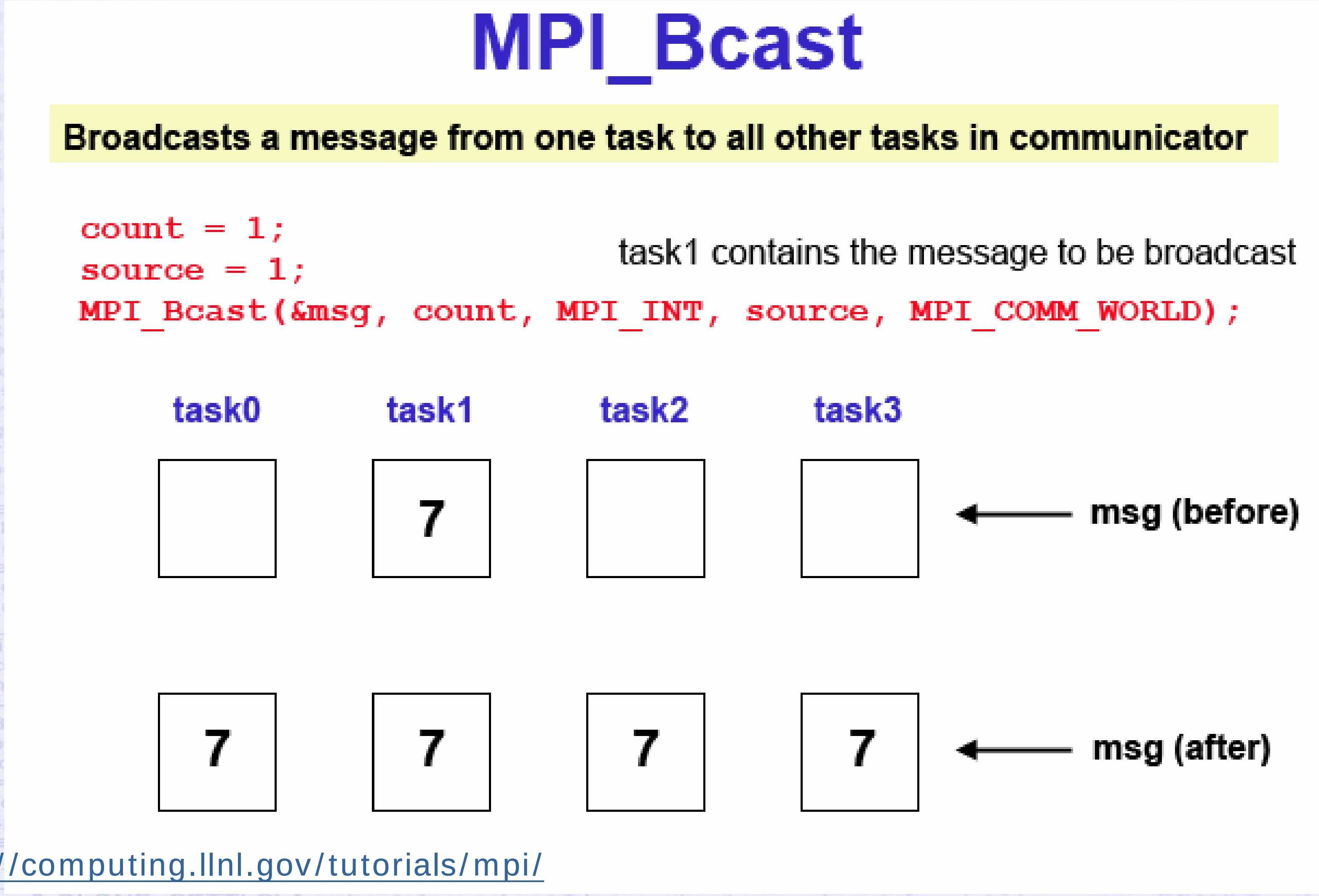
Perform reduction and store result across all tasks in communicator

```
count = 1;
MPI_Allreduce(sendbuf, recvbuf, count, MPI_INT,
              MPI_SUM, MPI_COMM_WORLD);
```

https://computing.llnl.gov/tutorials/mpi/

https://www.open-mpi.org/doc/v1.8/man3/MPI_Allreduce.3.php

MPI_Bcast on Matrix Multiplication



MPI_Bcast

Broadcasts a message from one task to all other tasks in communicator

```
count = 1;
source = 1;
MPI_Bcast(&msg, count, MPI_INT, source, MPI_COMM_WORLD);
```

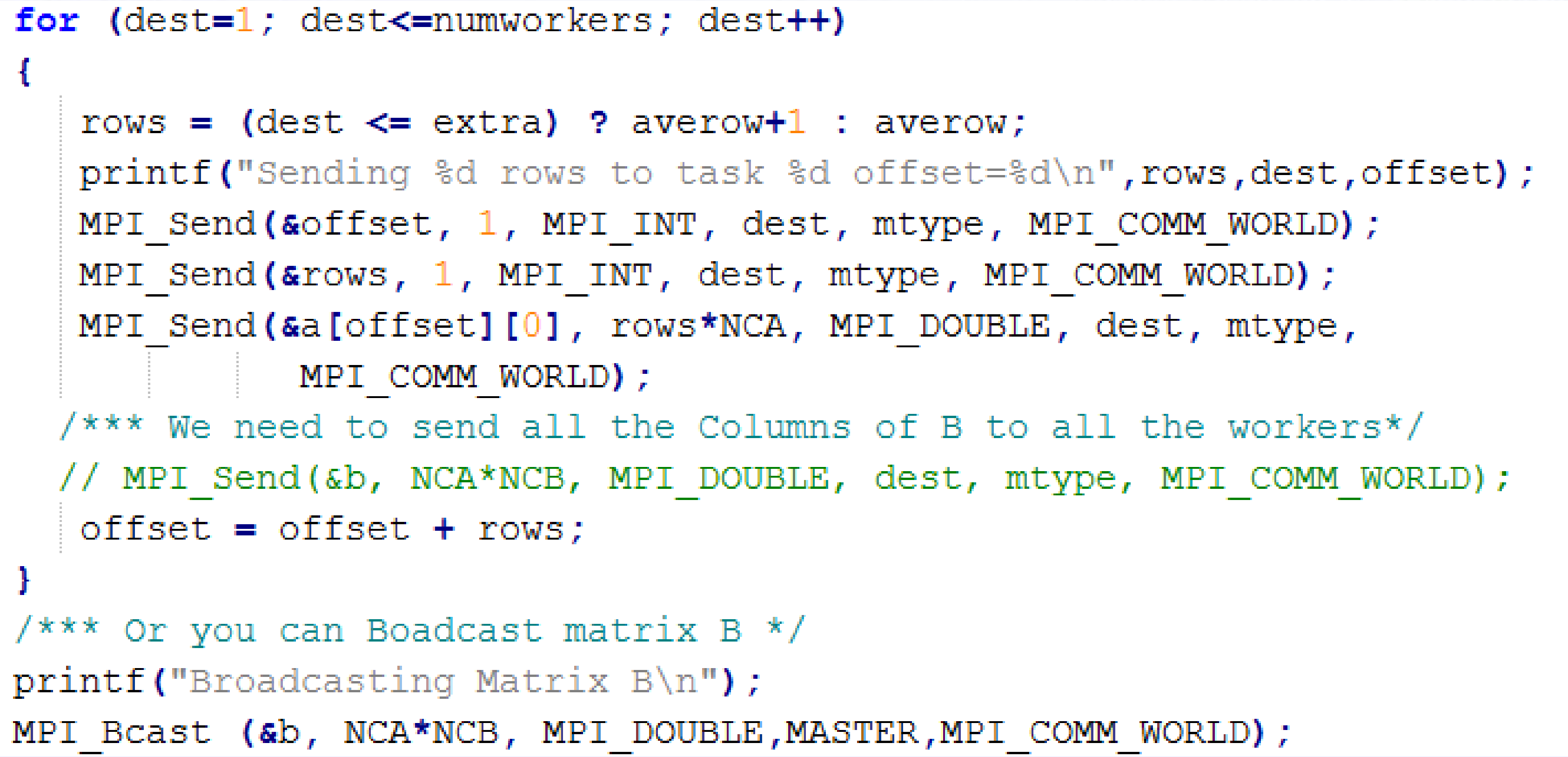
task1 contains the message to be broadcast

	task0	task1	task2	task3
msg (before)		7		
msg (after)	7	7	7	7

https://computing.llnl.gov/tutorials/mpi/

https://www.open-mpi.org/doc/v1.8/man3/MPI_Bcast.3.php

MPI_Bcast on Matrix Multiplication



```
for (dest=1; dest<=numworkers; dest++)
{
   rows = (dest <= extra) ? averow+1 : averow;
   printf("Sending %d rows to task %d offset=%d\n",rows,dest,offset);
   MPI_Send(&offset, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
   MPI_Send(&rows, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
   MPI_Send(&a[offset][0], rows*NCA, MPI_DOUBLE, dest, mtype,
            MPI_COMM_WORLD);
  /*** We need to send all the Columns of B to all the workers*/
  // MPI_Send(&b, NCA*NCB, MPI_DOUBLE, dest, mtype, MPI_COMM_WORLD);
   offset = offset + rows;
}
/*** Or you can Boadcast matrix B */
printf("Broadcasting Matrix B\n");
MPI_Bcast (&b, NCA*NCB, MPI_DOUBLE,MASTER,MPI_COMM_WORLD);
```

https://computing.llnl.gov/tutorials/mpi/