moasys
Moasys Corporation

2013 국가슈퍼컴퓨팅
Summer School@UNIST

MPI Parallel Programming

CONTENTS
Part I.

I. An Introduction to MPI Parallel Programming with the Message-Passing Model

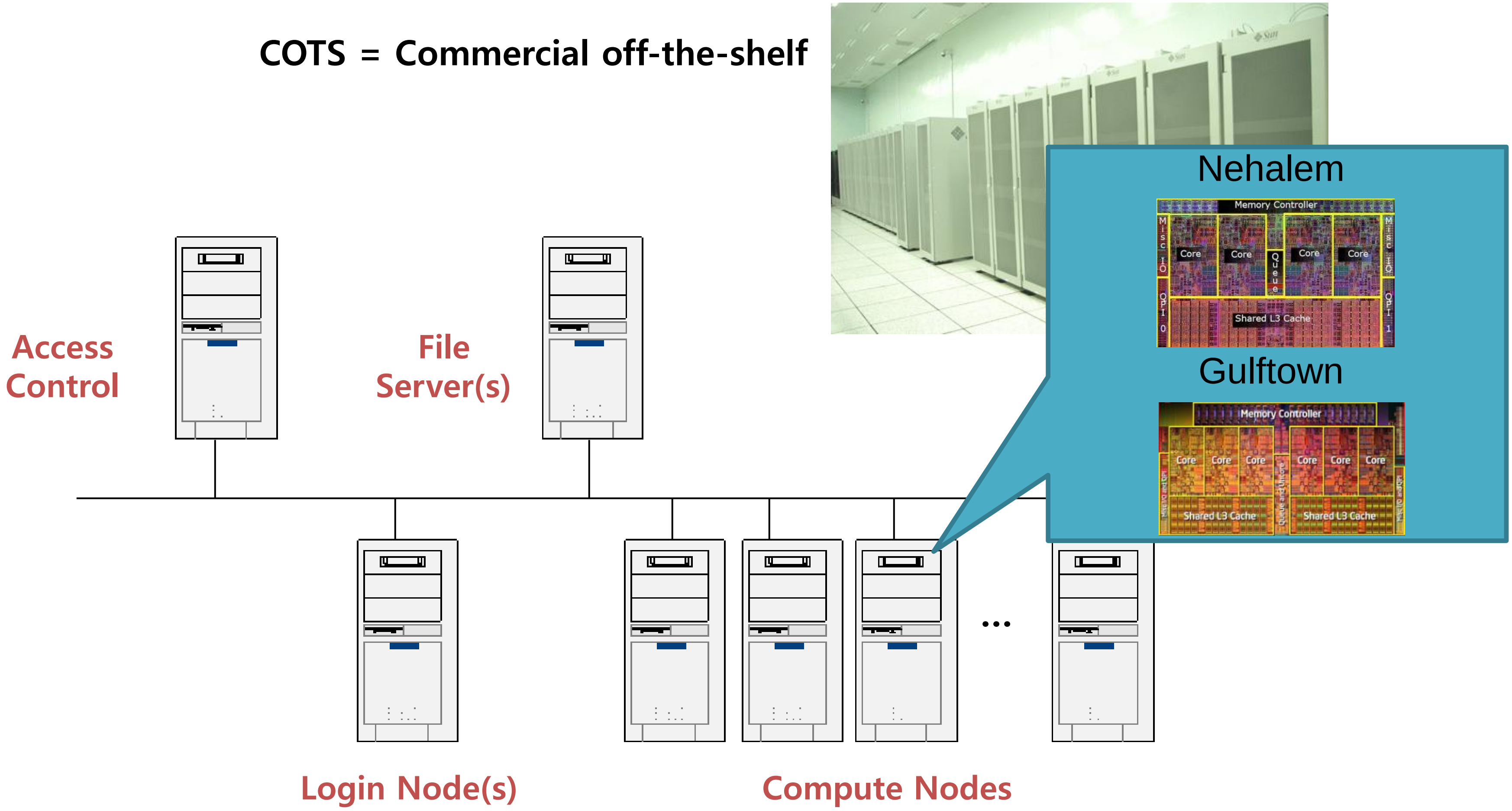
II. MPI Basic Send & Receive

III. Point to Point Communication Routines

I. An Introduction to MPI Parallel Programming with the Message-Passing Model

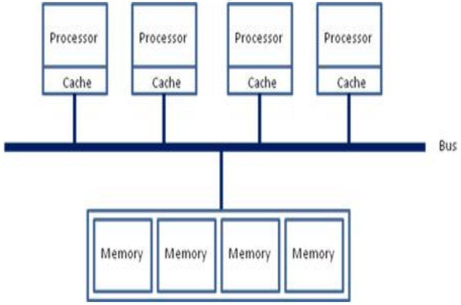
Current HPC Platforms : COTS-Based Clusters

COTS = Commercial off-the-shelf

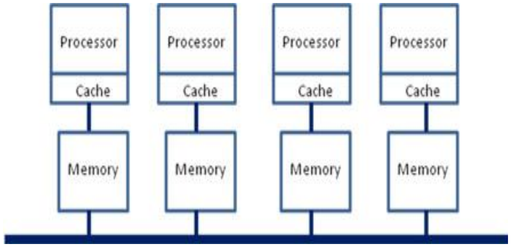


Memory Architectures

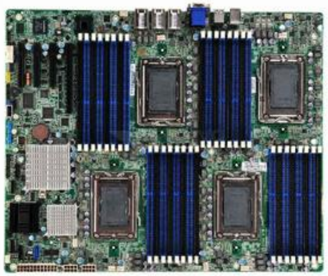
- **Shared Memory**
  - **Single address space for all processors**



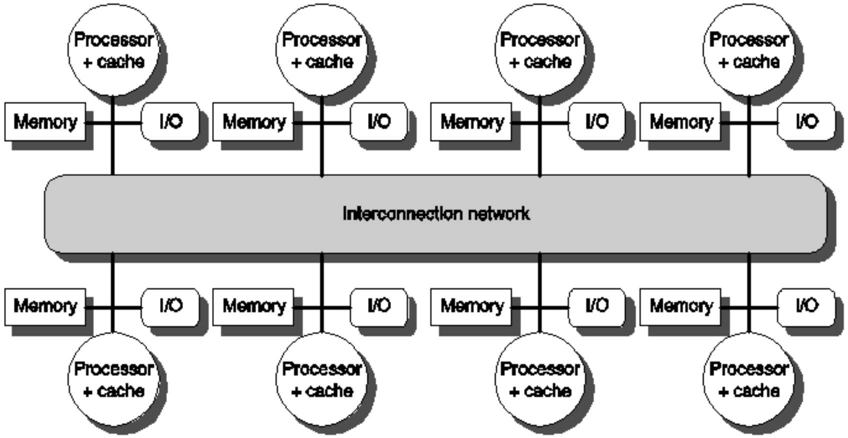
<UMA>

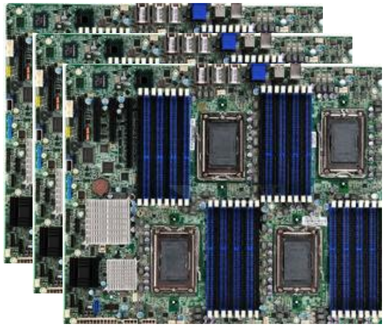


<NUMA>



- **Distributed Memory**





What is MPI?

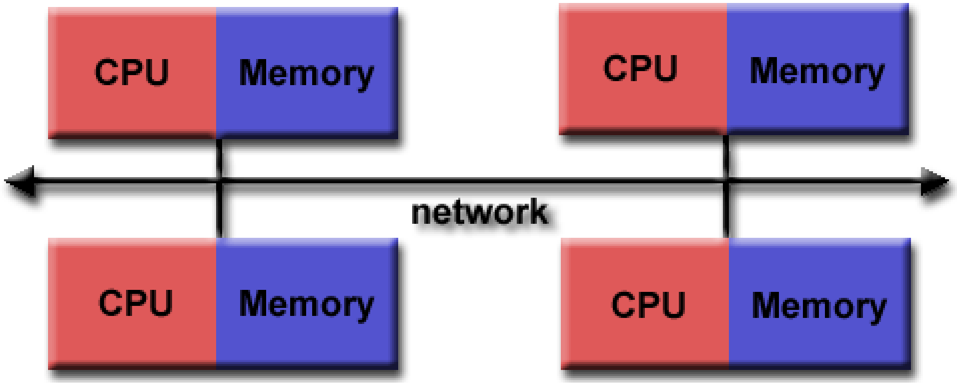
- MPI = Message Passing Interface
- MPI is a specification for the developers and users of message passing libraries. By itself, it is NOT a library – but rather the specification of what such a library should be.
- MPI primarily addresses the message-passing parallel programming model : data is moved from the address space of one process to that of another process through cooperative operations on each process.
- Simply stated, the goal of the message Passing Interface is to provide a widely used standard for writing message passing programs. The interface attempts to be :
    - Practical
    - Portable
    - Efficient
    - Flexible

What is MPI?

- The MPI standard has gone through a number of revisions, with the most recent version being MPI-3.
- Interface specifications have been defined for C and Fortran90 language bindings :
  - C++ bindings from MPI-1 are removed in MPI-3
  - MPI-3 also provides support for Fortran 2003 and 2008 features
- Actual MPI library implementations differ in which version and features of the MPI standard they support. Developers/users will need to be aware of this.

Programming Model

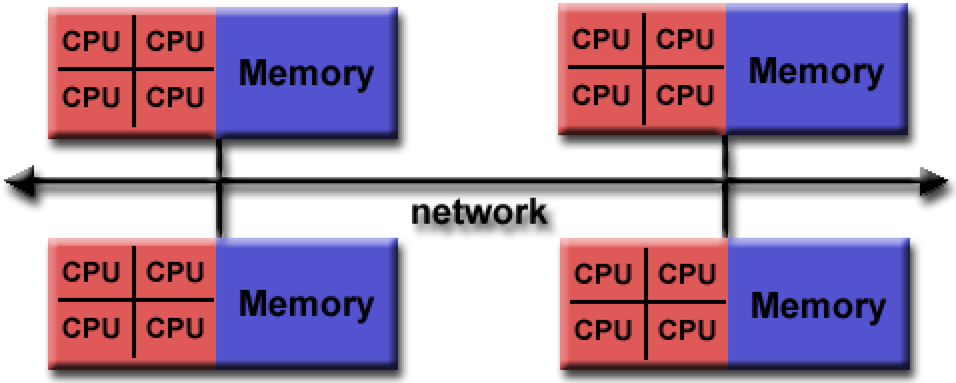
- Originally, MPI was designed for distributed memory architectures, which were becoming increasingly popular at time (1980s – early 1990s).



- As architecture trends changed, shared memory SMPs were combined over networks creating hybrid distributed memory/shared memory systems.

Programming Model

- MPI implementers adapted their libraries to handle both types of underlying memory architectures seamlessly. They also adapted/developed ways of handing different interconnects and protocols.



- Today, MPI runs on virtually any hardware platform :
  - Distributed Memory
  - Shared Memory
  - Hybrid
- The programming model clearly remains a distributed memory model however, regardless of the underlying physical architecture of the machine.

Reasons for Using MPI

- **Standardization**
  - **MPI is the only message passing library which can be considered a standard. It is supported on virtually all HPC platforms. Practically, it has replaced all previous message passing libraries.**
- **Portability**
  - **There is little or no need to modify your source code when you port your application to a different platform that supports (and is compliant with) the MPI standard.**
- **Performance Opportunities**
  - **Vendor implementations should be able to exploit native hardware features to optimize performance.**
- **Functionality**
  - **There are over 440 routines defined in MPI-3, which includes the majority of those in MPI-2 and MPI-1.**
- **Availability**
  - **A Variety of implementations are available, both vendor and public domain.**

History and Evolution

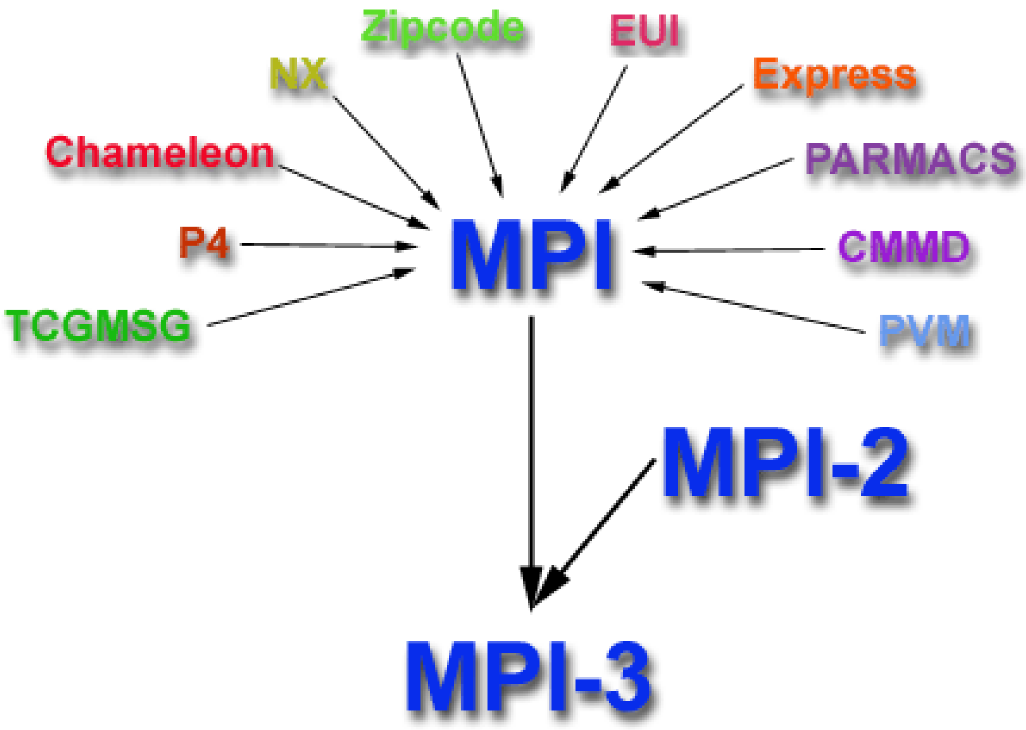
- MPI has resulted from the efforts of numerous individuals and groups that began in 1992.
- 1980s – early 1990s : Distributed memory, parallel computing develops, as do a number of incompatible soft ware tools for writing such programs – usually with tradeoffs between portability, performance, functionality and price. Recognition of the need for a standard arose.
- Apr 1992 : Workshop on Standards for Message Passing in a Distributed Memory Environment, Sponsored by the Center for Research on Parallel Computing, Williamsburg, Virginia. The basic features essential to a standard message passing interface were discussed, and a working group established to continue the standardization process. Preliminary draft proposal developed subsequently.

History and Evolution

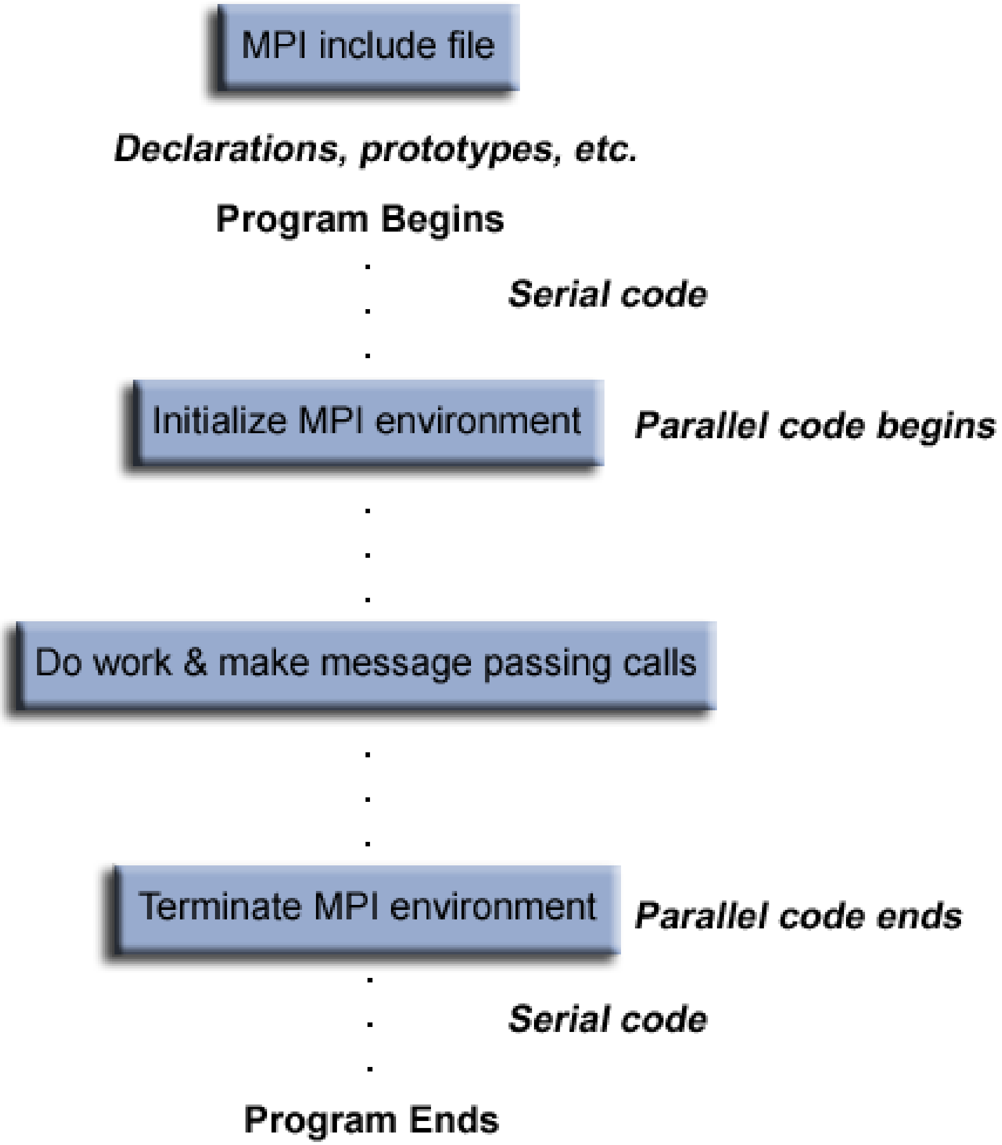
- Nov 1992 : Working group meets in Minneapolis. MPI draft proposal (MPI1) from ORNL presented. Group adopts procedures and organization to form the MPI Forum. It eventually comprised of about 175 individuals from 40 organizations including parallel computer vendors, software writers, academia and application scientists.
- Nov 1993 : Supercomputing 93 conference – draft MPI standard presented.
- May 1994 : Final version of MPI-1.0 released.
- MPI-1.0 was followed by versions MPI-1.1 (Jun 1995), MPI-1.2 (Jul 1997) and MPI-1.3 (May 2008).
- MPI-2 picked up where the first MPI specification left off, and addressed topics which went far beyond the MPI-1 specification. Was finalized in 1996.
- MPI-2.1 (Sep 2009), and MPI-2.2 (Sep 2009) followed.
- Sep 2012 : The MPI-3.0 standard was approved.

History and Evolution

- Documentation for all versions of the MPI standard is available at :
  - http://www.mpi-forum.org/docs/



A General Structure of the MPI Program



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

A Header File for MPI routines

- Required for all programs that make MPI library calls.

C include file	Fortran include file
#include “mpi.h”	include ‘mpif.h’

- With MPI-3 Fortran, the USE mpi_f80 module is preferred over using the include file shown above.

The Format of MPI Calls

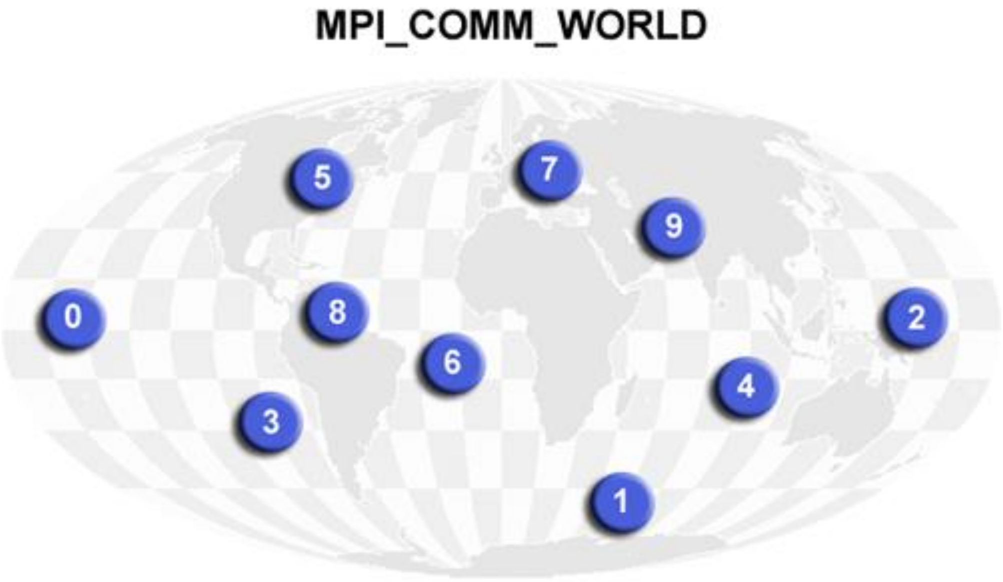
- C names are case sensitive; Fortran name are not.
- Programs must not declare variables or functions with names beginning with the prefix MPI_ or PMPI_ (profiling interface).

C Binding	
Format	rc = MPI_Xxxxx(parameter, ...)
Example	rc = MPI_Bsend(&buf, count, type, dest, tag, comm)
Error code	Returned as "rc", MPI_SUCCESS if successful.

Fortran Binding	
Format	CALL MPI_XXXXX(parameter, ..., ierr) call mpi_xxxxx(parameter, ..., ierr)
Example	call MPI_BSEND(buf, count, type, dest, tag, comm, ierr)
Error code	Returned as "ierr" parameter, MPI_SUCCESS if successful.

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.
- Communicators and groups will be covered in more detail later. For now, simply use MPI_COMM_WORLD whenever a communicator is required - it is the predefined communicator that includes all of your MPI processes.



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.
- 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).

Error Handling

- Most MPI routines include a return/error code parameter, as described in "Format of MPI Calls" section above.
- However, according to the MPI standard, the default behavior of an MPI call is to abort if there is an error. This means you will probably not be able to capture a return/error code other than MPI_SUCCESS (zero).
- The standard does provide a means to override this default error handler. You can also consult the error handing section of the MPI Standard located at http://www.mpi-forum.org/docs/mpi-11-html/node148.html .
- The types of errors displayed to the user are implementation dependent.

Environment Management Routines

- **MPI_Init**
  - **Initializes the MPI execution environment. This function must be called is 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.**

C	Fortran
MPI_Init(&argc, &argv)	MPI_INIT(ierr)

  - **Input parameters**
    - **argc : Pointer to the number of arguments**
    - **argv : Pointer to the argument vector**
  - **ierr : the error return argument**

Environment Management Routines

- **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.**

C	Fortran
MPI_Comm_size(comm, &size)	MPI_COMM_SIZE(comm, size, ierr)

- **Input parameters**
  - **comm : communicator (handle)**
- **Output parameters**
  - **size : number of processes in the group of comm (integer)**
- **ierr : the error return argument**

Environment Management Routines

- **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. This rank is often referred to as a task ID. If a process becomes associated with other communicators, it will have a unique rank within each of these as well.**

C	Fortran
MPI_Comm_rank(comm, &rank)	MPI_COMM_SIZE(comm, rank, ierr)

- **Input parameters**
  - **comm : communicator (handle)**
- **Output parameters**
  - **rank : rank of the calling process in the group of comm (integer)**
- **ierr : the error return argument**

Environment Management Routines

- **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.**

C	Fortran
MPI_Finalize()	MPI_FINALIZE(ierr)

  - **ierr : the error return argument**

Environment Management Routines

- **MPI_Abort**
  - **Terminates all MPI processes associated with the communicator. In most MPI implementations it terminates ALL processes regardless of the communicator specified.**

C	Fortran
MPI_Abort(comm, errorcode)	MPI_ABORT(comm, errorcode, ierr)

- **Input parameters**
  - **comm : communicator (handle)**
  - **errorcode : error code to return to invoking environment**
- **ierr : the error return argument**

Environment Management Routines

- MPI_Get_processor_name
  - Return 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.

C	Fortran
MPI_Get_processor_name(&name, &resultlength)	MPI_GET_PROCESSOR_NAME(name, resultlength, ierr)

  - Output parameters
    - name : A unique specifies for the actual (as opposed to virtual) node. This must be an array of size at least MPI_MAX_PROCESOR_NAME .
    - resultlen : Length (in characters) of the name.
  - ierr : the error return argument

Environment Management Routines

- **MPI_Get_version**
  - **Returns the version (either 1 or 2) and subversion of MPI.**

C	Fortran
MPI_Get_version(&version, &subversion)	MPI_GET_VERSION(version, subversion, ierr)

- **Output parameters**
  - **version : Major version of MPI (1 or 2)**
  - **subversion : Miner version of MPI.**
- **ierr : the error return argument**

Environment Management Routines

- **MPI_Initialized**
  - **Indicates whether MPI_Init has been called – returns flag as either logical true(1) or false(0).**

C	Fortran
MPI_Initialized(&flag)	MPI_INITIALIZED(flag, ierr)

- **Output parameters**
  - **flag : Flag is true if MPI_Init has been called and false otherwise.**
- **ierr : the error return argument**

Environment Management Routines

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

C	Fortran
MPI_Wtime()	MPI_WTIME()

  - **Return value**
    - **Time in seconds since an arbitrary time in the past.**

- **MPI_Wtick**
  - **Returns the resolution in seconds (double precision) of MPI_Wtime.**

C	Fortran
MPI_Wtick()	MPI_WTICK()

  - **Return value**
    - **Time in seconds of the resolution MPI_Wtime.**

Example: Hello world

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

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

          rc = MPI_Init(&argc, &argv);

          printf("Hello world.\n");

          rc = MPI_Finalize();

          return 0;
}
```

Example: Hello world

Execute a mpi program.

```
$ module load [compiler] [mpi]
$ mpicc hello.c
$ mpirun -np 4 -hostfile [hostfile] ./a.out
```

Make out a hostfile.

```
ibs0001 slots=2
ibs0002 slots=2
ibs0003 slots=2
ibs0003 slots=2
…
```

Example : Environment Management Routine

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

int main(argc,argv)
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 *******/

rc = MPI_Finalize();

return 0;

}
```

Coffee break

II. MPI Basic Send & Receive

Types of Point-to-Point Operations

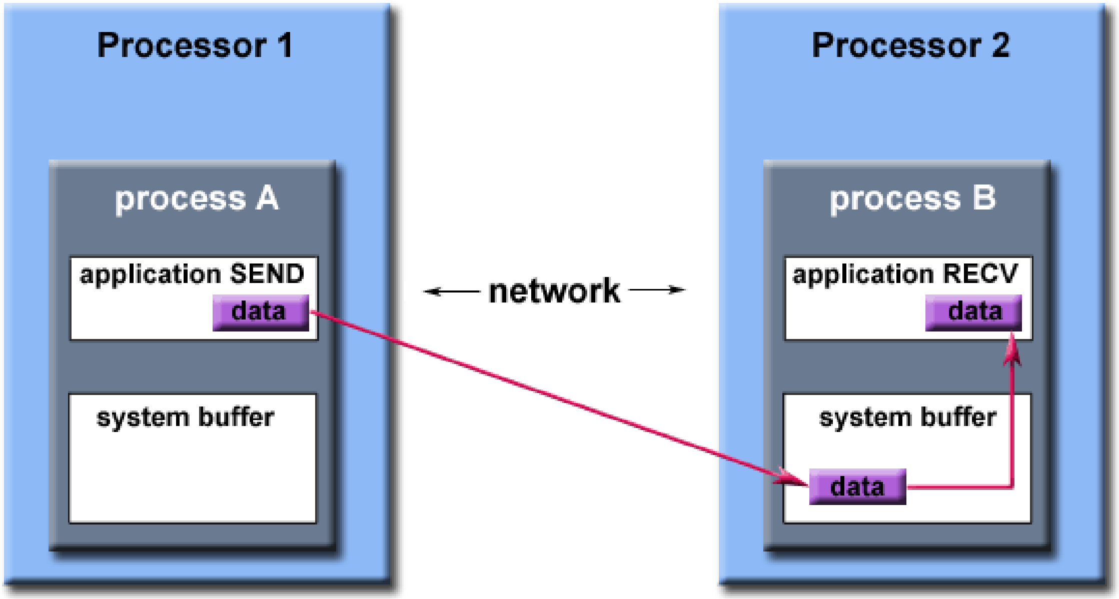
- **MPI point-to-point operations typically involve message passing between two, and only two, different MPI tasks. One task is performing a send operation and the other task is performing a matching receive operation.**
- **There are different types of send and receive routines used for different purposes.**
  - **Synchronous send**
  - **Blocking send/blocking receive**
  - **Non-blocking send/non-blocking receive**
  - **Buffered send**
  - **Combined send/receive**
  - **"Ready" send**
- **Any type of send routine can be paired with any type of receive routine.**
- **MPI also provides several routines associated with send – receive operations, such as those used to wait for a message's arrival or prove to find out if a message has arrived.**

Buffering

- In a perfect world, every send operation would be perfectly synchronized with its matching receive. This is rarely the case. Somehow or other, the MPI implementation must be able to deal with storing data when the two tasks are out of sync.
- Consider the following two cases
  - A send operation occurs 5 seconds before the receive is ready – where is the message while the receive is pending?
  - Multiple sends arrive at the same receiving task which can only accept one send at a ti me – what happens to the messages that are “backing up”?

Buffering

- The MPI implementation (not the MPI standard) decides what happens to data in these typ es of cases. Typically, a system buffer area is reserved to hold data in transit.



Path of a message buffered at the receiving process

Buffering

- System buffer space is :
  - Opaque to the programmer and managed entirely by the MPI library
  - A finite resource that can be easy to exhaust
  - Often mysterious and not well documented
  - Able to exist on the sending side, the receiving side, or both
  - Something that may improve program performance because it allows send – receive operations to be asynchronous.

Blocking vs. Non-blocking

- Most of the MPI point-to-point routines can be used in either blocking or non-blocking mode.

- 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 dose 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.

- 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 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 possibale performance gains.

III. Point to Point Communication Routines

MPI Message Passing Routine Arguments

- **MPI point-to-point communication routines generally have an argument list that takes one of the following formats :**

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.**

MPI Message Passing Routine Arguments

- **Data type**
  - **For reasons of portability, MPI predefines its elementary data types. The table below lists those required by the standard.**

C Data Types	
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_SIGNED_CHAR	signed char
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 Message Passing Routine Arguments

- **Destination**
  - **An argument to send routines that indicates the process where a message should be delivered. Specified as the rank of the receiving process.**
- **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 communicator, the predefined communicator MPI_COMM_WORLD is usually used.**

MPI Message Passing Routine Arguments

- 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 predefined structure MPI_Status (ex. stat.MPI_SOURCE, stat.MPI_TAG).
  - In Fortran, it is an integer array of size MPI_STATUS_SIZE (ex. stat(MPI_SOURCE), stat(MPI_TAG)).
  - Additionally, the actual number of bytes received are obtainable from Status via MPI_Get_out 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 pointer to predefined structure MPI_Request.
  - In Fortran, it is an integer.

Example : Blocking Message Passing Routine (1/2)

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

int main(argc,argv)
int argc;
char *argv[];  {
int numtasks, rank, dest, source, rc, count, tag=1;
char inmsg, outmsg='x';
MPI_Status Stat;

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

if (rank == 0) {
  dest = 1;
  source = 1;
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  }

else if (rank == 1) {
  dest = 0;
  source = 0;
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  }
```

Example : Blocking Message Passing Routine (2/2)

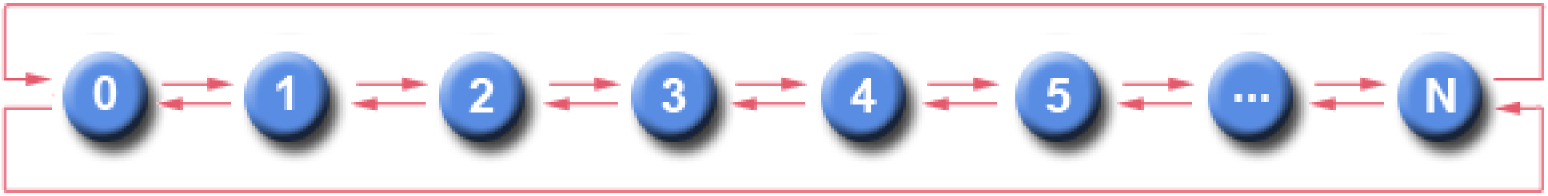
```
rc = MPI_Get_count(&Stat, MPI_CHAR, &count);
printf("Task %d: Received %d char(s) from task %d with tag %d \n",
       rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);

MPI_Finalize();

return 0;

}
```

Example : Dead Lock

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

int main(argc,argv)
int argc;
char *argv[];  {
int numtasks, rank, dest, source, rc, count, tag=1;
char inmsg, outmsg='x';
MPI_Status Stat;

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

if (rank == 0) {
  dest = 1;
  source = 1;
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  }

else if (rank == 1) {
  dest = 0;
  source = 0;
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
}
```

Example : Non-Blocking Message Passing Routine (1/2)

Nearest neighbor exchange in a ring topology



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

int main(argc,argv)
int argc;
char *argv[];  {
int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
MPI_Request reqs[4];
MPI_Status stats[2];

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

prev = rank-1;
next = rank+1;
if (rank == 0)  prev = numtasks - 1;
if (rank == (numtasks - 1))  next = 0;
```

Example : Non-Blocking Message Passing Routine (2/2)

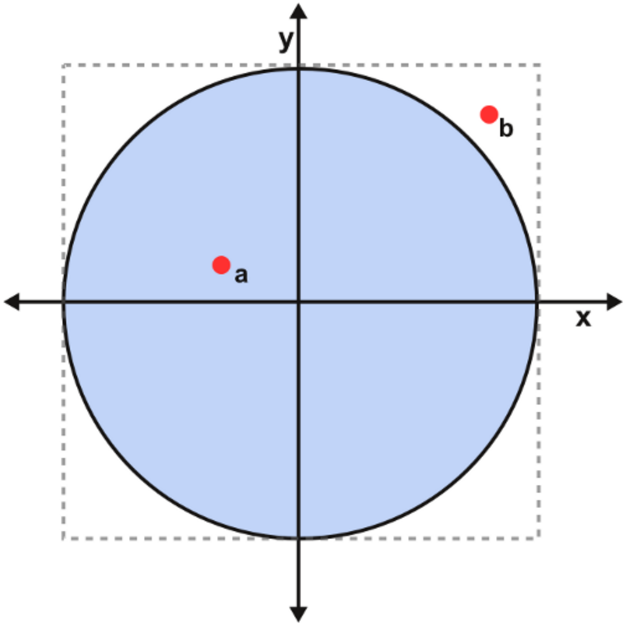
```
MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);

MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);

      {  do some work  }

MPI_Waitall(4, reqs, stats);

MPI_Finalize();

return 0;

}
```

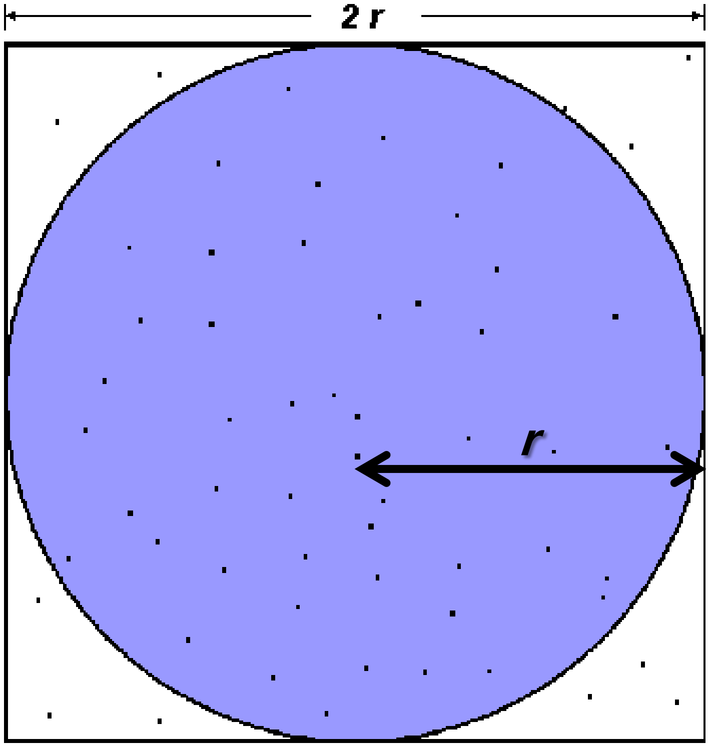
Coffee break

Advanced Example : Monte-Carlo Simulation

- <Problem>
  - Monte carlo simulation
  - Random number use
  - *PI = 4 x Ac/As*

- <Requirement>
  - N's processor(rank) use
  - P2p communication





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

$A_C = \pi r^2$

$\pi = 4 \times \frac{A_C}{A_S}$

Advanced Example : Monte-Carlo Simulation for PI

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

int main() {
const long num_step=100000000;
long i, cnt;
double pi, x, y, r;

printf(“------------------------------------------------------------\n”);
pi = 0.0;
cnt = 0;
r = 0.0;

for (i=0; i<num_step; i++) {
  x = rand() / (RAND_MAX+1.0);
  y = rand() / (RAND_MAX+1.0);
  r = sqrt(x*x + y*y);
  if (r<=1) cnt += 1;
}

pi = 4.0 * (double)(cnt) / (double)(num_step);
printf(“PI = %17.15lf (Error = %e)\n”, pi, fabs(acos(-1.0)-pi));
printf(“------------------------------------------------------------\n”);

return 0;

}
```

Advanced Example : Numerical integration for PI

<Problem>

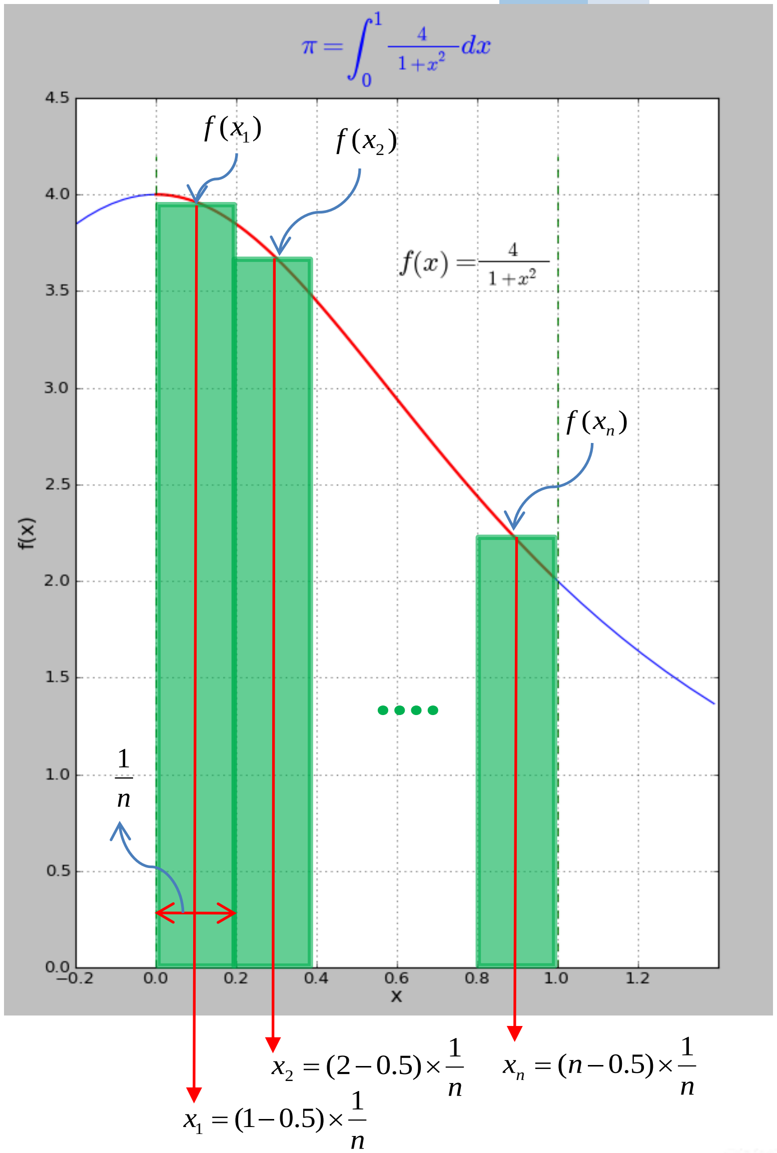
- Get PI using Numerical integration

$$\int_0^1 \frac{4.0}{(1+x^2)}\, dx = \pi$$

<Requirement>

- Point to point communication

$$\pi \approx \sum_{i=1}^{n} \frac{4}{1+((i-0.5)\times\frac{1}{n})^2} \times \frac{1}{n}$$



Advanced Example : Numerical integration for PI

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

int main() {
const long num_step=100000000;
long i;
double sum, step, pi, x;
step = (1.0/(double)num_step);
sum=0.0;

printf(“------------------------------------------------------------\n”);

for (i=0; i<num_step; i++) {
x = ((double)i - 0.5) * step;
sum += 4.0/(1.0+x*x);
}

pi = step * sum;

printf(“PI = %5lf (Error = %e)\n”, pi, fabs(acos(-1.0)-pi));
printf(“------------------------------------------------------------\n”);

return 0;

}
```

Any questions?

CONTENTS
Part II.

I. Introduction to Collective Operations in MPI

II. Collective Communication Routines

III. Advanced Features of the MPI

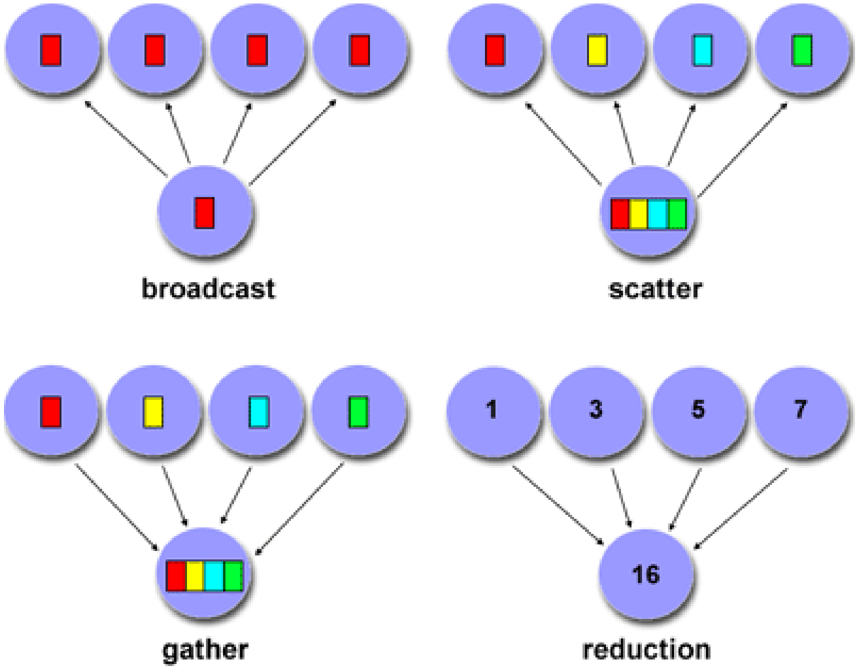
I. Introduction to Collective Operations in MPI

Scope

- Collective communication routines must involve all processes within the scope of a communicator.
  - All processes are by default, members in the communicator MPI_COMM_WORLD.
  - Additional communicator can be defined by the programmer. See the Group and Communicator Management Routine section for details.
- Unexpected behavior, including program failure, can occur if even one task in the communicator doesn't participate.
- It is the programmer's responsibility to ensure that all processes within a communicator participate in any collective operations.

Type of Collective Operations

- Synchronization
  - processes wait until all members of the group have reached the synchronization point.
- Data Movement
  - broadcast, scatter/gather, all to all.
- Collective Computation (reductions)
  - one member of the group collects data from the other members and performs an operation (min, max, add, multiply, etc.) on that data.



Programming Considerations and Restrictions

- With MPI-3, collective operations can be blocking or non-blocking. Only blocking operation s are covered in this tutorial.
- Collective communication routines do not take message tag arguments.
- Collective operations within subset of processes are accomplished by first partitioning the s ubsets into new groups and then attaching the new groups to new communicators.
- Con only be used with MPI predefined datatypes – not with MPI Derived Data Types.
- MPI-2 extended most collective operations to allow data movement between intercommuni cators (not covered here).

II. Collective Communication Routine

Collective Communication Routines

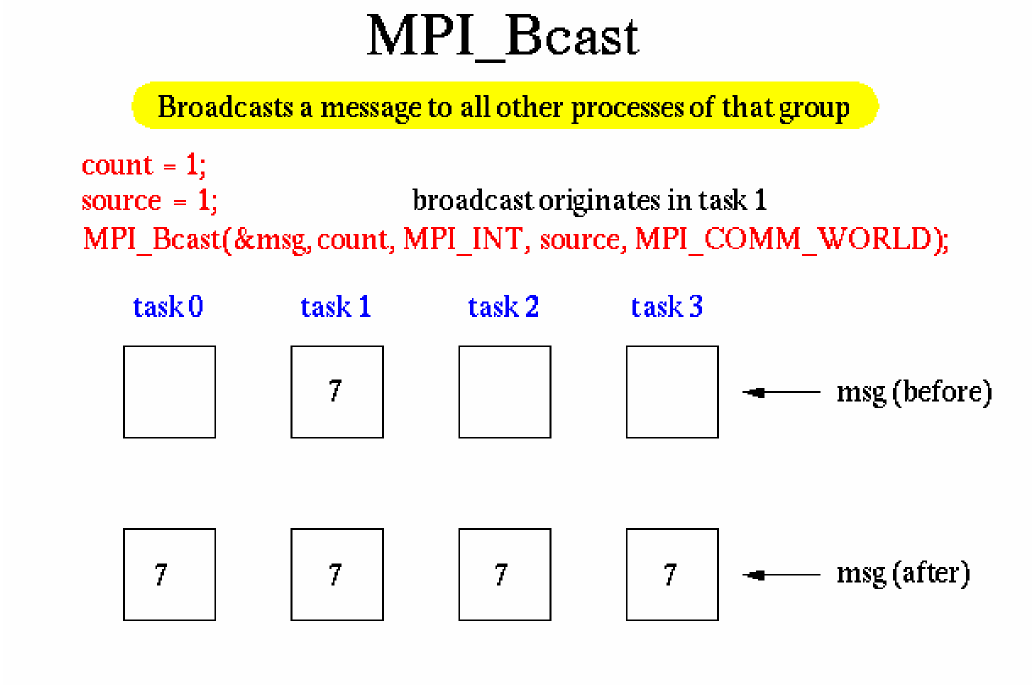
- **MPI_Barrier**
  - **Synchronization operation. Creates a barrier synchronization in a group. Each task, when reaching the MPI_Barrier call, blocks until all tasks in the group reach the same MPI_Barrier call. Then all tasks are free to proceed.**

C	Fortran
MPI_Barrier(comm)	MPI_BARRIER(comm, ierr)

Collective Communication Routines

- **MPI_Bcast**
  - **Data movement operation. Broadcasts (sends) a message from the process with rank "root" to all other processes in the group.**

C	Fortran
MPI_Bcast(&buffer, count, datatype, root, comm)	MPI_BCAST (buffer,count,datatype,root,comm,ierr)



MPI_Bcast

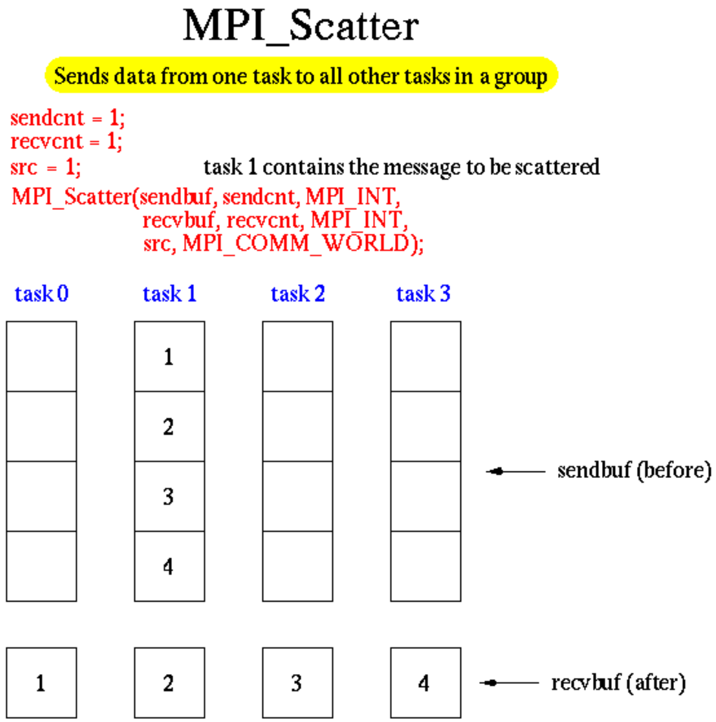
Broadcasts a message to all other processes of that group

```
count = 1;
source = 1;                      broadcast originates in task 1
MPI_Bcast(&msg, count, MPI_INT, source, MPI_COMM_WORLD);
```

Collective Communication Routines

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

C	Fortran
MPI_Scatter (&sendbuf,sendcnt,sendtype,&recvbuf, recvcnt,recvtype,root,comm)	MPI_SCATTER (sendbuf,sendcnt,sendtype,recvbuf, recvcnt,recvtype,root,comm,ierr)



MPI_Scatter

Sends data from one task to all other tasks in a group

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

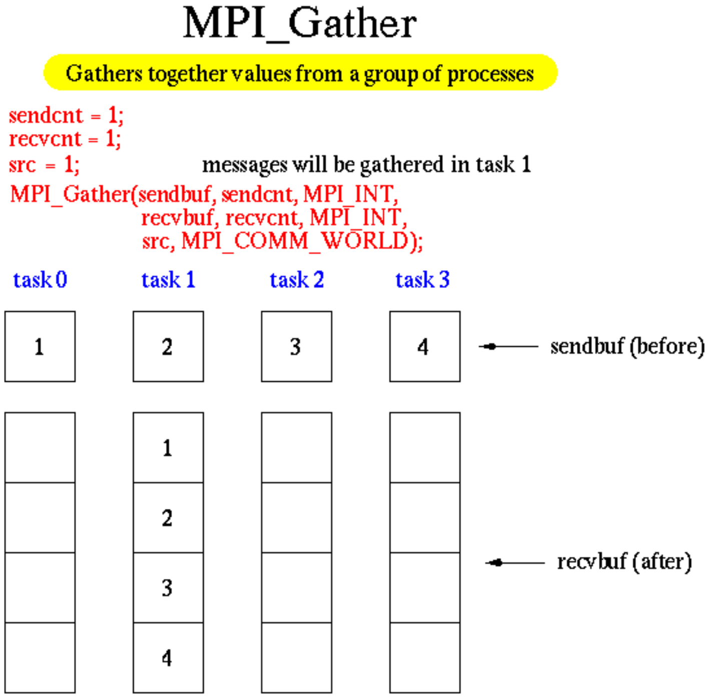
task 1 contains the message to be scattered

Collective Communication Routines

MPI_Gather

- Data movement operation. Gathers distinct messages from each task in the group to a single destination task. This routine is the reverse operation of MPI_Scatter.

C	Fortran
MPI_Gather (&sendbuf,sendcnt,sendtype,&recvbuf, recvcount,recvtype,root,comm)	MPI_GATHER (sendbuf,sendcnt,sendtype,recvbuf, recvcount,recvtype,root,comm,ierr)



MPI_Gather

Gathers together values from a group of processes

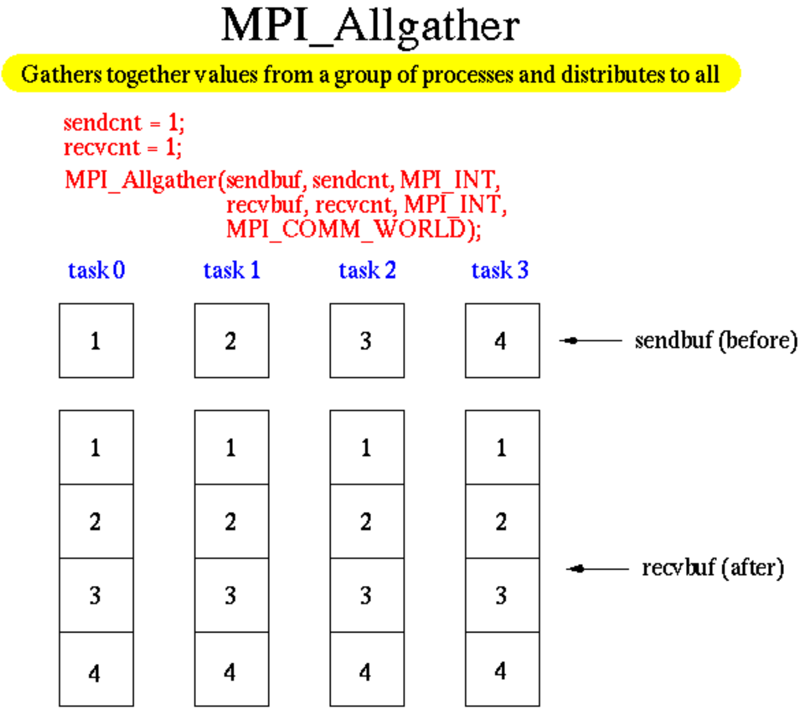
```
sendcnt = 1;
recvcnt = 1;
src  = 1;
MPI_Gather(sendbuf, sendcnt, MPI_INT,
           recvbuf, recvcnt, MPI_INT,
           src, MPI_COMM_WORLD);
```

messages will be gathered in task 1

Collective Communication Routines

- **MPI_Allgather**
  - **Data movement operation. Concatenation of data to all tasks in a group. Each task in the group, in effect, performs a one-to-all broadcasting operation within the group.**

C	Fortran
MPI_Allgather (&sendbuf,sendcount,sendtype,&recvbuf, recvcount,recvtype,comm)	MPI_ALLGATHER (sendbuf,sendcount,sendtype,recvbuf, recvcount,recvtype,comm,info)



MPI_Allgather

Gathers together values from a group of processes and distributes to all

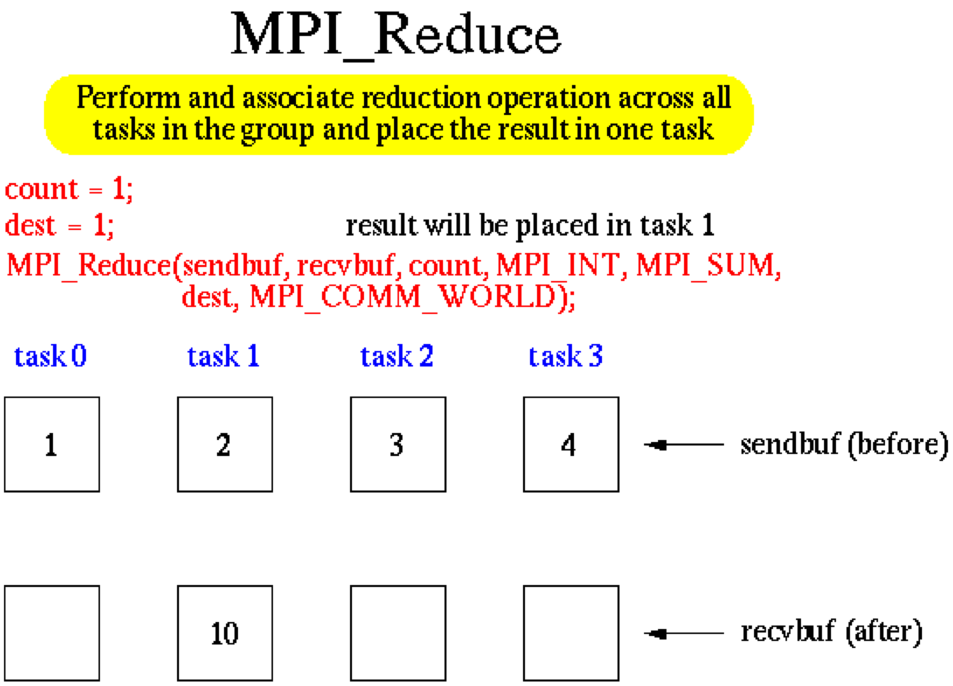
```
sendcnt = 1;
recvcnt = 1;
MPI_Allgather(sendbuf, sendcnt, MPI_INT,
              recvbuf, recvcnt, MPI_INT,
              MPI_COMM_WORLD);
```

Collective Communication Routines

MPI_Reduce

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

C	Fortran
MPI_Reduce (&sendbuf,&recvbuf,count,datatype, op,root,comm)	MPI_REDUCE (sendbuf,recvbuf,count,datatype,op, root,comm,ierr)



MPI_Reduce

Perform and associate reduction operation across all tasks in the group and place the result in one task

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

result will be placed in task 1

Collective Communication Routines

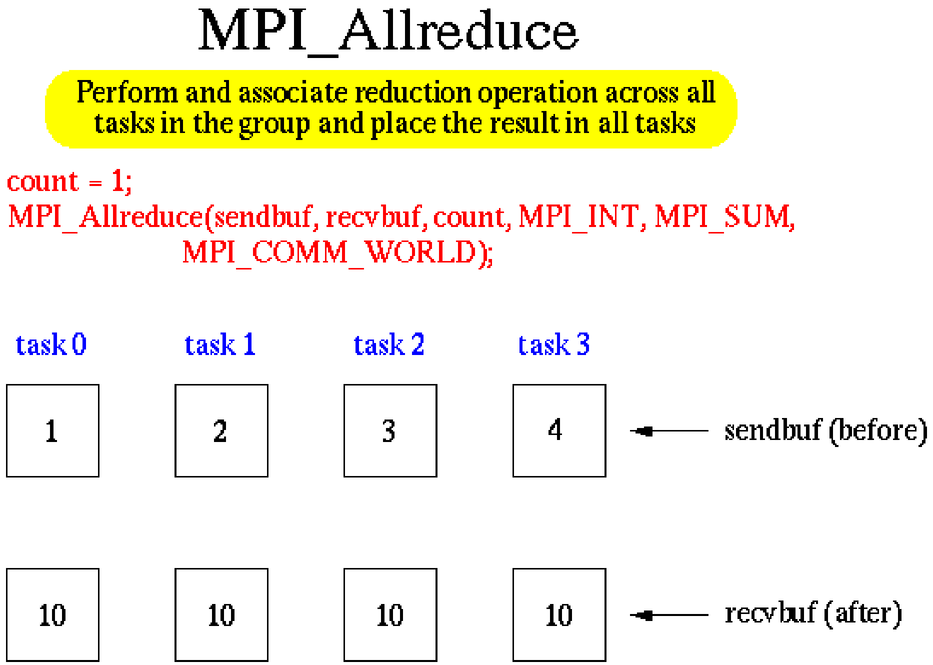
- **The predefined MPI reduction operations appear below. Users can also define their own reduction functions by using the MPI_Op_create routine.**

MPI Reduction Operation		C Data Types
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	float, double and long double
MPI_MINLOC	min value and location	float, double and long double

Collective Communication Routines

- **MPI_Allreduce**
  - **Collective computation operation + data movement. Applies a reduction operation and places the result in all tasks in the group. This is equivalent to an MPI_Reduce followed by an MPI_Bcast.**

C	Fortran
MPI_Allreduce (&sendbuf,&recvbuf,count,datatype, op,comm)	MPI_ALLREDUCE (sendbuf,recvbuf,count,datatype,op, comm,ierr)



MPI_Allreduce

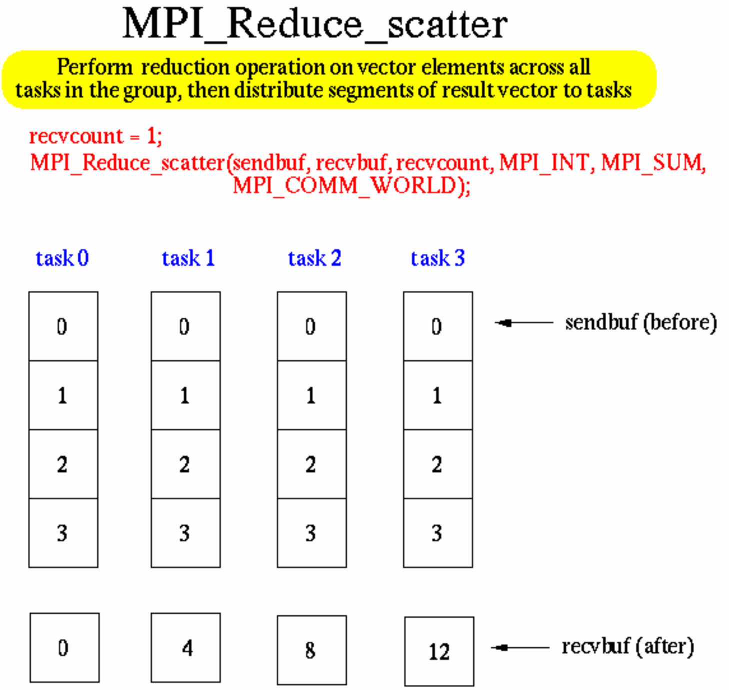
Perform and associate reduction operation across all tasks in the group and place the result in all tasks

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

Collective Communication Routines

- MPI_Reduce_scatter
  - Collective computation operation + data movement. First does an element-wise reduction on a vector across all tasks in the group. Next, the result vector is split into disjoint segments and distributed across the tasks. This is equivalent to an MPI_Reduce followed by an MPI_Scatter operation.

C	Fortran
MPI_Reduce_scatter (&sendbuf,&recvbuf,recvcount,datatype, op,comm)	MPI_REDUCE_SCATTER (sendbuf,recvbuf,recvcount,datatype, op,comm,ierr)



MPI_Reduce_scatter

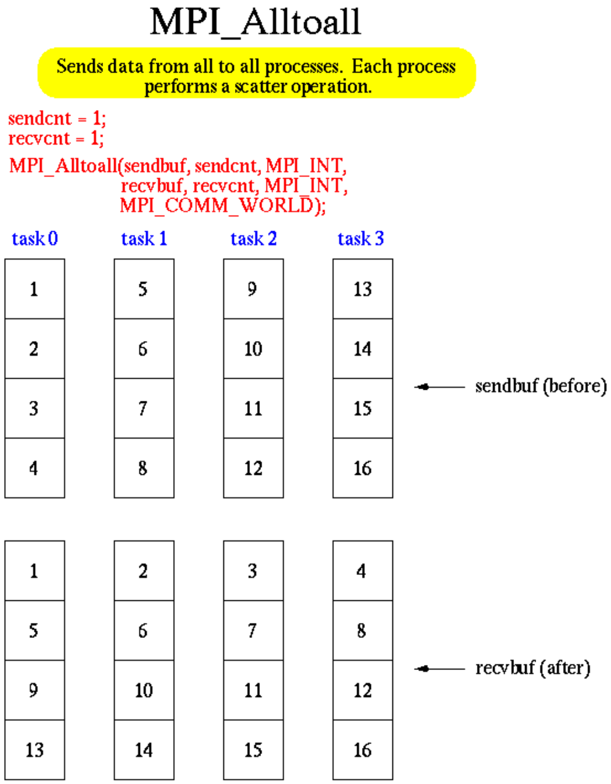
Perform reduction operation on vector elements across all tasks in the group, then distribute segments of result vector to tasks

```
recvcount = 1;
MPI_Reduce_scatter(sendbuf, recvbuf, recvcount, MPI_INT, MPI_SUM,
                   MPI_COMM_WORLD);
```

Collective Communication Routines

- **MPI_Alltoall**
  - **Data movement operation. Each task in a group performs a scatter operation, sending a distinct message to all the tasks in the group in order by index.**

C	Fortran
MPI_Alltoall (&sendbuf,sendcount,sendtype,&recvbuf, recvcnt,recvtype,comm)	MPI_ALLTOALL (sendbuf,sendcount,sendtype,recvbuf, recvcnt,recvtype,comm,ierr)



MPI_Alltoall

Sends data from all to all processes. Each process performs a scatter operation.

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

task 0	task 1	task 2	task 3
1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

sendbuf (before)

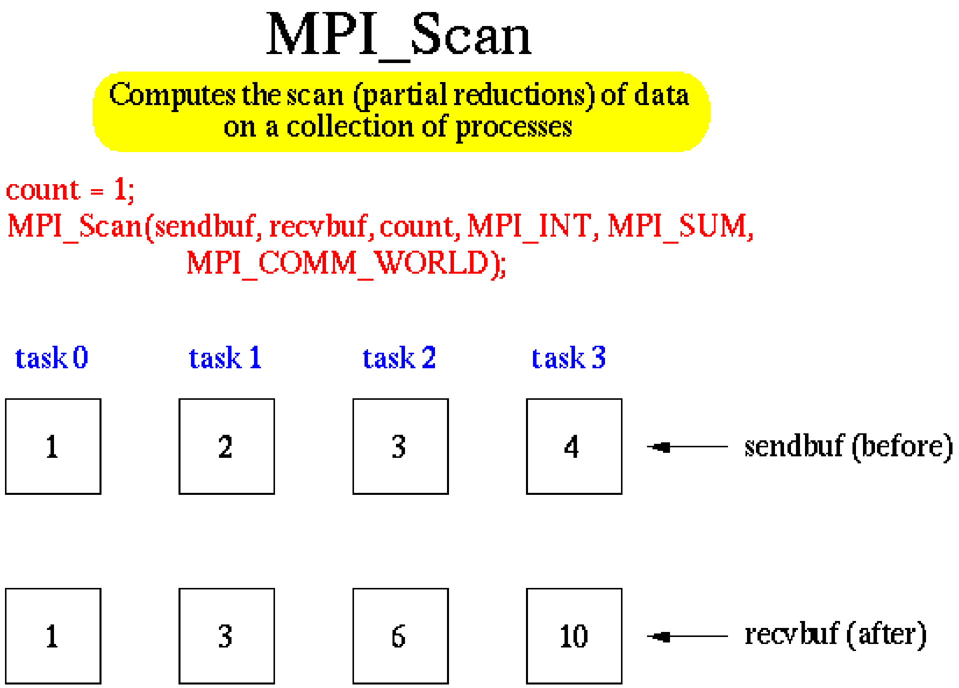
task 0	task 1	task 2	task 3
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

recvbuf (after)

Collective Communication Routines

- **MPI_Scan**
  - **Performs a scan operation with respect to a reduction operation across a task group.**

C	Fortran
MPI_Scan (&sendbuf,&recvbuf,count,datatype, op,comm)	MPI_SCAN (sendbuf,recvbuf,count,datatype,op, comm,ierr)



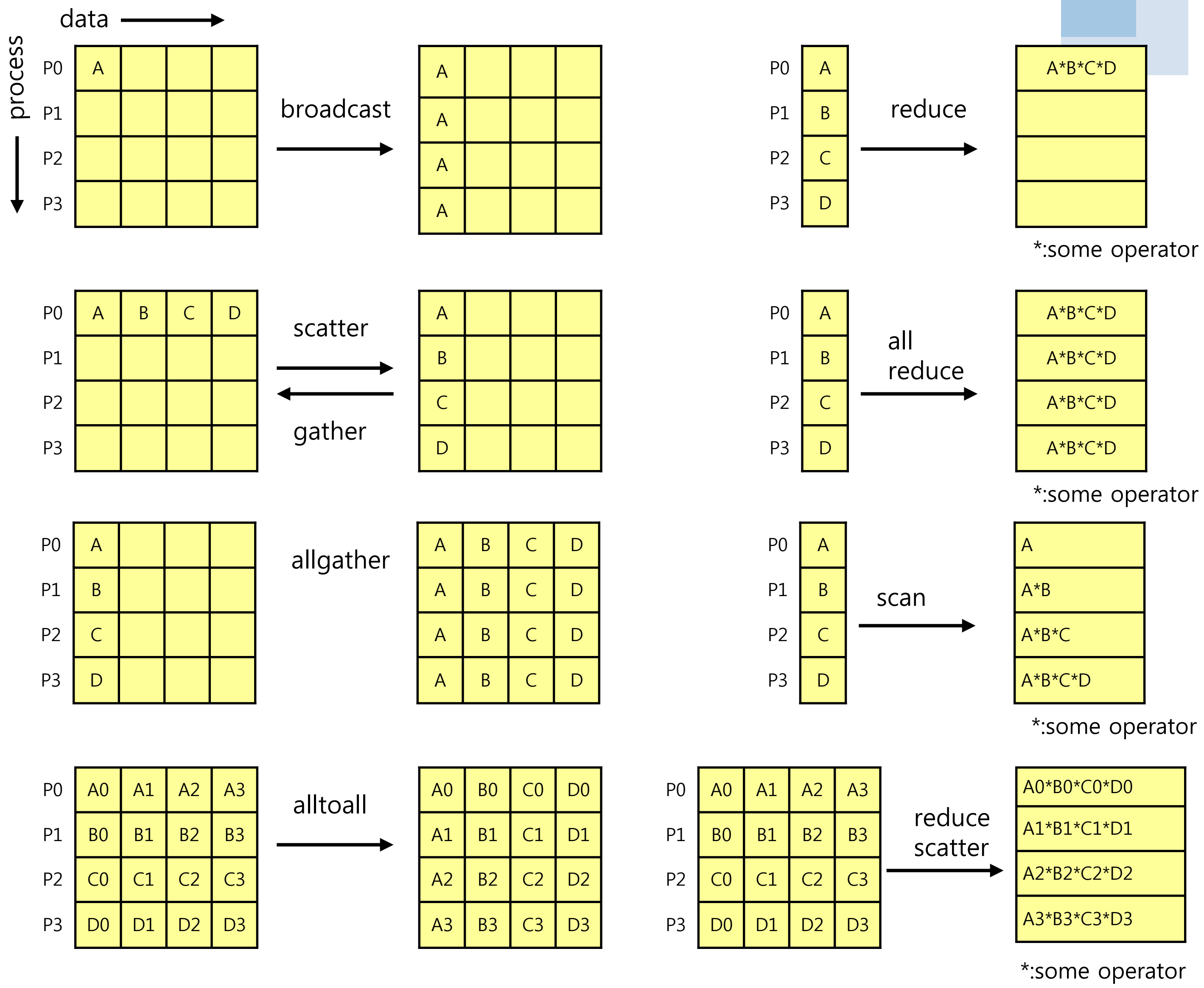
MPI_Scan

Computes the scan (partial reductions) of data on a collection of processes

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

task 0	task 1	task 2	task 3	
1	2	3	4	← sendbuf (before)
1	3	6	10	← recvbuf (after)

Collective Communication Routines



data →

process ↓

broadcast

P0	A			
P1				
P2				
P3				

A			
A			
A			
A			

reduce

P0	A
P1	B
P2	C
P3	D

A*B*C*D

*:some operator

scatter / gather

P0	A	B	C	D
P1				
P2				
P3				

A			
B			
C			
D			

all reduce

P0	A
P1	B
P2	C
P3	D

A*B*C*D
A*B*C*D
A*B*C*D
A*B*C*D

*:some operator

allgather

P0	A			
P1	B			
P2	C			
P3	D			

A	B	C	D
A	B	C	D
A	B	C	D
A	B	C	D

scan

P0	A
P1	B
P2	C
P3	D

A
A*B
A*B*C
A*B*C*D

*:some operator

alltoall

P0	A0	A1	A2	A3
P1	B0	B1	B2	B3
P2	C0	C1	C2	C3
P3	D0	D1	D2	D3

A0	B0	C0	D0
A1	B1	C1	D1
A2	B2	C2	D2
A3	B3	C3	D3

reduce scatter

P0	A0	A1	A2	A3
P1	B0	B1	B2	B3
P2	C0	C1	C2	C3
P3	D0	D1	D2	D3

A0*B0*C0*D0
A1*B1*C1*D1
A2*B2*C2*D2
A3*B3*C3*D3

*:some operator

Example : Collective Communication (1/2)

- **Perform a scatter operation on the rows of an array**

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

int main(argc,argv)
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);
```

Example : Collective Communication (2/2)

```
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();

return 0;

}
```

Advanced Example : Monte-Carlo Simulation for PI

- Use the collective communication routines!

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

int main() {
const long num_step=100000000;
long i, cnt;
double pi, x, y, r;

printf(“-----------------------------------------------------------\n”);
pi = 0.0;
cnt = 0;
r = 0.0;

for (i=0; i<num_step; i++) {
  x = rand() / (RAND_MAX+1.0);
  y = rand() / (RAND_MAX+1.0);
  r = sqrt(x*x + y*y);
  if (r<=1) cnt += 1;
}

pi = 4.0 * (double)(cnt) / (double)(num_step);
printf(“PI = %17.15lf (Error = %e)\n”, pi, fabs(acos(-1.0)-pi));
printf(“-----------------------------------------------------------\n”);

return 0;

}
```

Advanced Example : Numerical integration for PI

» **Use the collective communication routines!**

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

int main() {
const long num_step=100000000;
long i;
double sum, step, pi, x;
step = (1.0/(double)num_step);
sum=0.0;

printf(“-----------------------------------------------------------\n”);

for (i=0; i<num_step; i++) {
x = ((double)i - 0.5) * step;
sum += 4.0/(1.0+x*x);
}

pi = step * sum;

printf(“PI = %5lf (Error = %e)\n”, pi, fabs(acos(-1.0)-pi));
printf(“-----------------------------------------------------------\n”);

return 0;

}
```

Coffee break

III. Advanced Features of the MPI

Derived Data Types

- MPI also provides facilities for you to define your own data structures based upon sequences of the MPI primitive data types. Such user defined structures are called derived data types.
- Primitive data types are contiguous. Derived data types allow you to specify non-contiguous data in a convenient manner and to treat it as though it was contiguous.
- MPI provides several methods for constructing derived data types:
  - Contiguous
  - Vector
  - Indexed
  - Struct

Derived Data Type Routines

- MPI_Type_contiguous
  - The simplest constructor. Produces a new data type by making count copies of an exist ing data type.
    - MPI_Type_contiguous (count,oldtype,&newtype)

- MPI_Type_vector
  - Similar to contiguous, but allows for regular gaps (stride) in the displacements. MPI_Ty pe_hvector is identical to MPI_Type_vector except that stride is specified in bytes.
    - MPI_Type_vector (count,blocklength,stride,oldtype,&newtype)

- MPI_Type_indexed
  - An array of displacements of the input data type is provided as the map for the new d ata type. MPI_Type_hindexed is identical to MPI_Type_indexed except that offsets are s pecified in bytes.
    - MPI_Type_indexed (count,blocklens[],offsets[],old_type,&newtype)

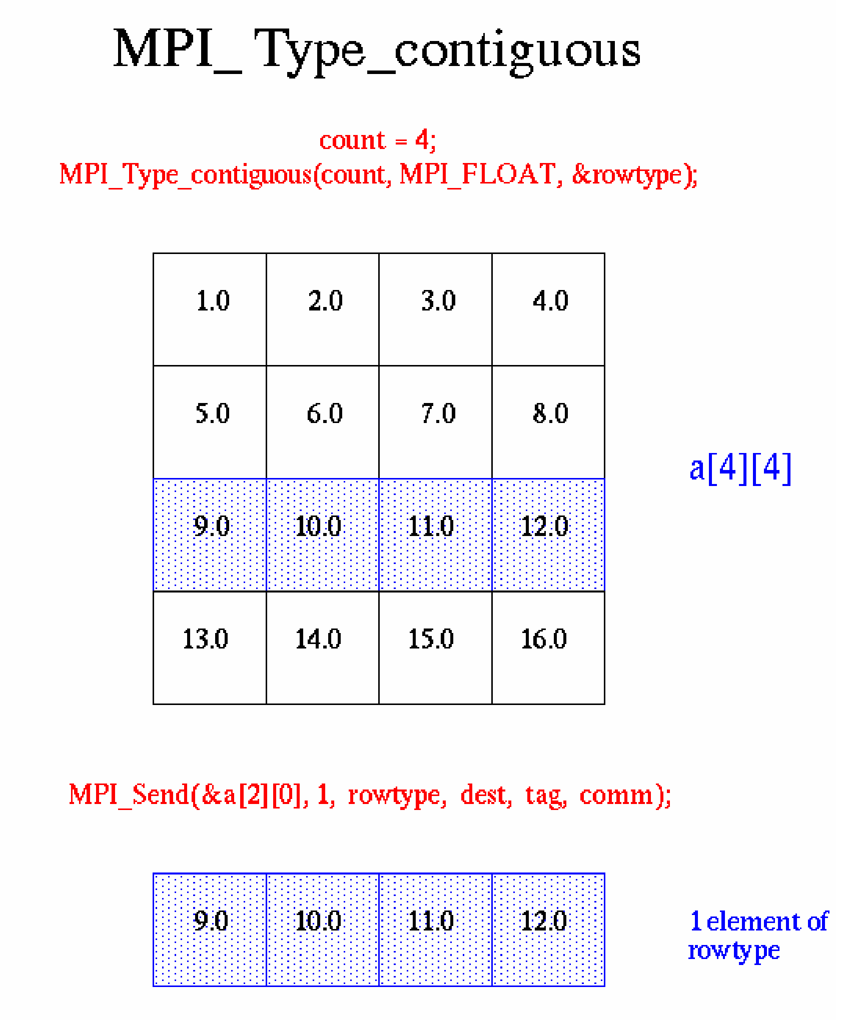
- MPI_Type_struct
  - The new data type is formed according to completely defined map of the component d ata types.
    - MPI_Type_struct (count,blocklens[],offsets[],old_types,&newtype)

Derived Data Type Routines

- MPI_Type_extent
  - Returns the size in bytes of the specified data type. Useful for the MPI subroutines tha t require specification of offsets in bytes.
    - MPI_Type_extent (datatype,&extent)
- MPI_Type_commit
  - Commits new datatype to the system. Required for all user constructed (derived) datat ypes.
    - MPI_Type_commit (&datatype)
- MPI_Type_free
  - Deallocates the specified datatype object. Use of this routine is especially important to prevent memory exhaustion if many datatype objects are created, as in a loop.
    - MPI_Type_free (&datatype)

Example : Contiguous Derived Data Type

- Create a data type representing a row of an array and distribute a different row to all processes.



MPI_ Type_contiguous

```
count = 4;
MPI_Type_contiguous(count, MPI_FLOAT, &rowtype);
```

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

a[4][4]

```
MPI_Send(&a[2][0], 1, rowtype, dest, tag, comm);
```

9.0	10.0	11.0	12.0

1 element of rowtype

Example : Contiguous Derived Data Type (1/2)

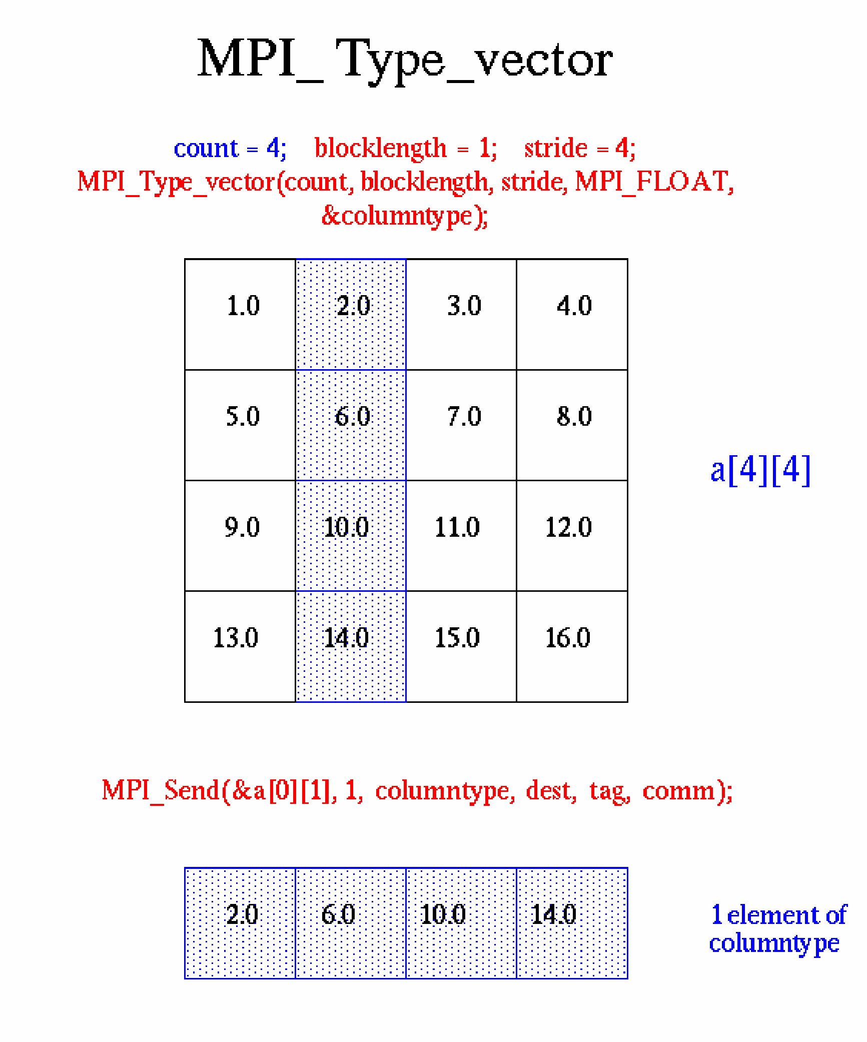
```
#include "mpi.h"
#include <stdio.h>
#define SIZE 4

int main(argc,argv)
int argc;
char *argv[];  {
int numtasks, rank, source=0, dest, tag=1, i;
float a[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 b[SIZE];

MPI_Status stat;
MPI_Datatype rowtype;

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

MPI_Type_contiguous(SIZE, MPI_FLOAT, &rowtype);
MPI_Type_commit(&rowtype);
```

Example : Contiguous Derived Data Type (2/2)

```
if (numtasks == SIZE) {
  if (rank == 0) {
     for (i=0; i<numtasks; i++)
       MPI_Send(&a[i][0], 1, rowtype, i, tag, MPI_COMM_WORLD);
     }

  MPI_Recv(b, SIZE, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &stat);
  printf("rank= %d  b= %3.1f %3.1f %3.1f %3.1f\n",
         rank,b[0],b[1],b[2],b[3]);
  }
else
  printf("Must specify %d processors. Terminating.\n",SIZE);

MPI_Type_free(&rowtype);
MPI_Finalize();

return 0;

}
```

Example : Vector Derived Data Type

- Create a data type representing a column of an array and distribute different columns to all processes.



MPI_ Type_vector

```
count = 4;   blocklength = 1;   stride = 4;
MPI_Type_vector(count, blocklength, stride, MPI_FLOAT,
&columntype);
```

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

a[4][4]

```
MPI_Send(&a[0][1], 1, columntype, dest, tag, comm);
```

2.0	6.0	10.0	14.0

1 element of columntype

Example : Vector Derived Data Type (1/2)

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

int main(argc,argv)
int argc;
char *argv[];  {
int numtasks, rank, source=0, dest, tag=1, i;
float a[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 b[SIZE];

MPI_Status stat;
MPI_Datatype columntype;

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

MPI_Type_vector(SIZE, 1, SIZE, MPI_FLOAT, &columntype);
MPI_Type_commit(&columntype);
```

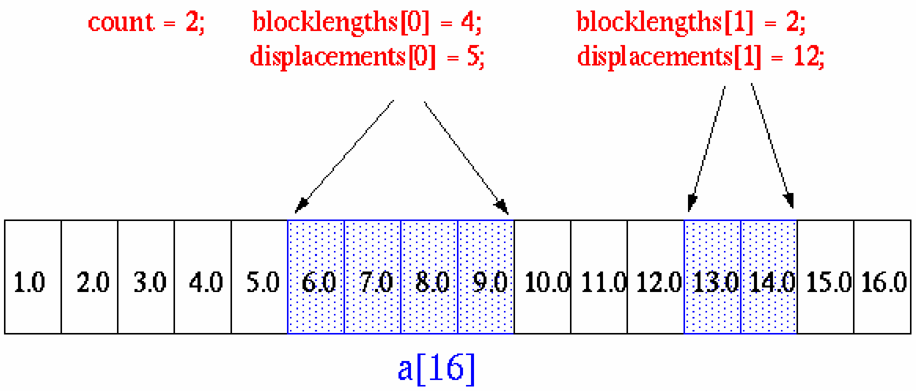
Example : Vector Derived Data Type (2/2)

```
if (numtasks == SIZE) {
  if (rank == 0) {
     for (i=0; i<numtasks; i++)
       MPI_Send(&a[0][i], 1, columntype, i, tag, MPI_COMM_WORLD);
        }

  MPI_Recv(b, SIZE, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &stat);
  printf("rank= %d  b= %3.1f %3.1f %3.1f %3.1f\n",
        rank,b[0],b[1],b[2],b[3]);
  }
else
  printf("Must specify %d processors. Terminating.\n",SIZE);

MPI_Type_free(&columntype);
MPI_Finalize();

return 0;

}
```

Example : Indexed Derived Data Type

- Create a datatype by extracting variable portions of an array and distribute to all tasks.

MPI_ Type_indexed



count = 2;

blocklengths[0] = 4;
displacements[0] = 5;

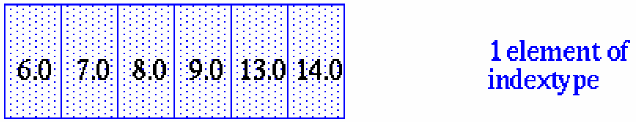
blocklengths[1] = 2;
displacements[1] = 12;

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

a[16]

```
MPI_Type_indexed(count, blocklengths, displacements, MPI_FLOAT, &indextype);
```

```
MPI_Send(&a, 1, indextype, dest, tag, comm);
```



6.0	7.0	8.0	9.0	13.0	14.0

1 element of indextype

Example : Indexed Derived Data Type (1/2)

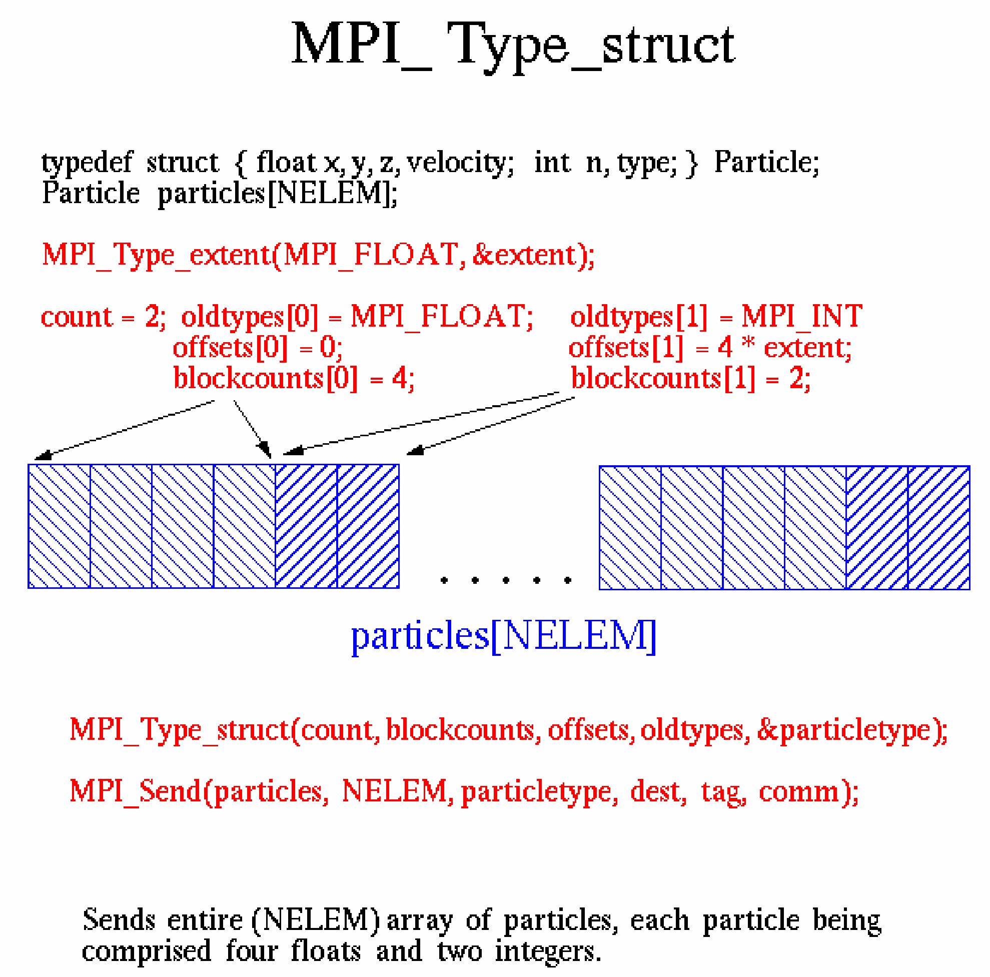
```
#include "mpi.h"
#include <stdio.h>
#define NELEMENTS 6

int main(argc,argv)
int argc;
char *argv[];  {
int numtasks, rank, source=0, dest, tag=1, i;
int blocklengths[2], displacements[2];
float a[16] =
  {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 b[NELEMENTS];

MPI_Status stat;
MPI_Datatype indextype;

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

blocklengths[0] = 4;
blocklengths[1] = 2;
displacements[0] = 5;
displacements[1] = 12;
```

Example : Indexed Derived Data Type (2/2)

```
MPI_Type_indexed(2, blocklengths, displacements, MPI_FLOAT, &indextype);
MPI_Type_commit(&indextype);

if (rank == 0) {
  for (i=0; i<numtasks; i++)
     MPI_Send(a, 1, indextype, i, tag, MPI_COMM_WORLD);
  }

MPI_Recv(b, NELEMENTS, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &stat);
printf("rank= %d  b= %3.1f %3.1f %3.1f %3.1f %3.1f %3.1f\n",
     rank,b[0],b[1],b[2],b[3],b[4],b[5]);

MPI_Type_free(&indextype);
MPI_Finalize();

return 0;

}
```

Example : Struct Derived Data Type

- Create a data type that represents a particle and distribute an array of such particles to all processes.



MPI_ Type_struct

```
typedef  struct  { float x, y, z, velocity;  int  n, type; }  Particle;
Particle  particles[NELEM];

MPI_Type_extent(MPI_FLOAT, &extent);

count = 2;  oldtypes[0] = MPI_FLOAT;     oldtypes[1] = MPI_INT
            offsets[0] = 0;              offsets[1] = 4 * extent;
            blockcounts[0] = 4;          blockcounts[1] = 2;
```

particles[NELEM]

```
MPI_Type_struct(count, blockcounts, offsets, oldtypes, &particletype);

MPI_Send(particles,  NELEM, particletype,  dest,  tag,  comm);
```

Sends entire (NELEM) array of particles, each particle being comprised four floats and two integers.

Example : Struct Derived Data Type (1/3)

```
#include "mpi.h"
#include <stdio.h>
#define NELEM 25

int main(argc,argv)
int argc;
char *argv[];  {
int numtasks, rank, source=0, dest, tag=1, i;

typedef struct {
  float x, y, z;
  float velocity;
  int  n, type;
  }          Particle;
Particle     p[NELEM], particles[NELEM];
MPI_Datatype particletype, oldtypes[2];
int          blockcounts[2];

/* MPI_Aint type used to be consistent with syntax of */
/* MPI_Type_extent routine */
MPI_Aint    offsets[2], extent;
```

Example : Struct Derived Data Type (2/3)

```
MPI_Status stat;

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

/* Setup description of the 4 MPI_FLOAT fields x, y, z, velocity */
offsets[0] = 0;
oldtypes[0] = MPI_FLOAT;
blockcounts[0] = 4;

/* Setup description of the 2 MPI_INT fields n, type */
/* Need to first figure offset by getting size of MPI_FLOAT */
MPI_Type_extent(MPI_FLOAT, &extent);
offsets[1] = 4 * extent;
oldtypes[1] = MPI_INT;
blockcounts[1] = 2;

/* Now define structured type and commit it */
MPI_Type_struct(2, blockcounts, offsets, oldtypes, &particletype);
MPI_Type_commit(&particletype);
```

Example : Struct Derived Data Type (3/3)

```
/* Initialize the particle array and then send it to each task */
if (rank == 0) {
  for (i=0; i<NELEM; i++) {
     particles[i].x = i * 1.0;
     particles[i].y = i * -1.0;
     particles[i].z = i * 1.0;
     particles[i].velocity = 0.25;
     particles[i].n = i;
     particles[i].type = i % 2;
     }
  for (i=0; i<numtasks; i++)
     MPI_Send(particles, NELEM, particletype, i, tag, MPI_COMM_WORLD);
  }

MPI_Recv(p, NELEM, particletype, source, tag, MPI_COMM_WORLD, &stat);

/* Print a sample of what was received */
printf("rank= %d   %3.2f %3.2f %3.2f %3.2f %d %d\n", rank,p[3].x,
     p[3].y,p[3].z,p[3].velocity,p[3].n,p[3].type);

MPI_Type_free(&particletype);
MPI_Finalize();

return 0;

}
```

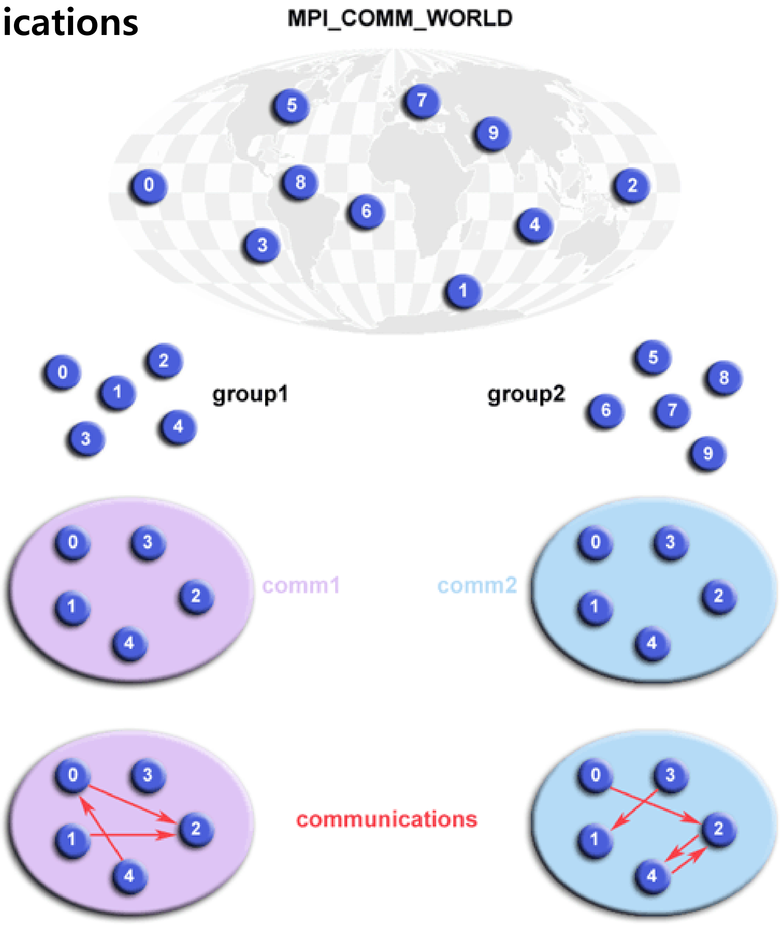
Coffee break

Groups vs. Communicators

- A group is an ordered set of processes. Each process in a group is associated with a unique integer rank. Rank values start at zero and go to N-1, where N is the number of processes in the group. In MPI, a group is represented within system memory as an object. It is accessible to the programmer only by a "handle". A group is always associated with a communicator object.
- A communicator encompasses a group of processes that may communicate with each other. All MPI messages must specify a communicator. In the simplest sense, the communicator is an extra "tag" that must be included with MPI calls. Like groups, communicators are represented within system memory as objects and are accessible to the programmer only by "handles". For example, the handle for the communicator that comprises all tasks is MPI_COMM_WORLD.
- From the programmer's perspective, a group and a communicator are one. The group routines are primarily used to specify which processes should be used to construct a communicator.

Primary Purposes of Group and Communicator Objects

- Allow you to organize tasks, based upon function, into task groups.
- Enable Collective Communications operations across a subset of related tasks.
- Provide basis for implementing user defined virtual topologies
- Provide for safe communications



Programming Considerations for Group/Communicator

- Groups/communicators are dynamic - they can be created and destroyed during program e xecution.
- Processes may be in more than one group/communicator. They will have a unique rank within each group/communicator.
- MPI provides over 40 routines related to groups, communicators, and virtual topologies.
- Typical usage:
  - Extract handle of global group from MPI_COMM_WORLD using MPI_Comm_group
  - Form new group as a subset of global group using MPI_Group_incl
  - Create new communicator for new group using MPI_Comm_create
  - Determine new rank in new communicator using MPI_Comm_rank
  - Conduct communications using any MPI message passing routine
  - When finished, free up new communicator and group (optional) using MPI_Comm_free and MPI_Group_free

Example : Group and Communicator Routine (1/2)

- **Create two different process groups for separate collective communications exchange. Requires creating new communicators also.**

```
#include "mpi.h"
#include <stdio.h>
#define NPROCS 8

int main(argc,argv)
int argc;
char *argv[];  {
int        rank, new_rank, sendbuf, recvbuf, numtasks,
           ranks1[4]={0,1,2,3}, ranks2[4]={4,5,6,7};
MPI_Group  orig_group, new_group;
MPI_Comm   new_comm;

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

if (numtasks != NPROCS) {
  printf("Must specify MP_PROCS= %d. Terminating.\n",NPROCS);
  MPI_Finalize();
  exit(0);
  }

sendbuf = rank;
```

Example : Group and Communicator Routine (2/2)

```
/* Extract the original group handle */
MPI_Comm_group(MPI_COMM_WORLD, &orig_group);

/* Divide tasks into two distinct groups based upon rank */
if (rank < NPROCS/2) {
  MPI_Group_incl(orig_group, NPROCS/2, ranks1, &new_group);
  }
else {
  MPI_Group_incl(orig_group, NPROCS/2, ranks2, &new_group);
  }

/* Create new new communicator and then perform collective communications */
MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
MPI_Allreduce(&sendbuf, &recvbuf, 1, MPI_INT, MPI_SUM, new_comm);

MPI_Group_rank (new_group, &new_rank);
printf("rank= %d newrank= %d recvbuf= %d\n",rank,new_rank,recvbuf);

MPI_Finalize();

return 0;

}
```

Virtual Topology

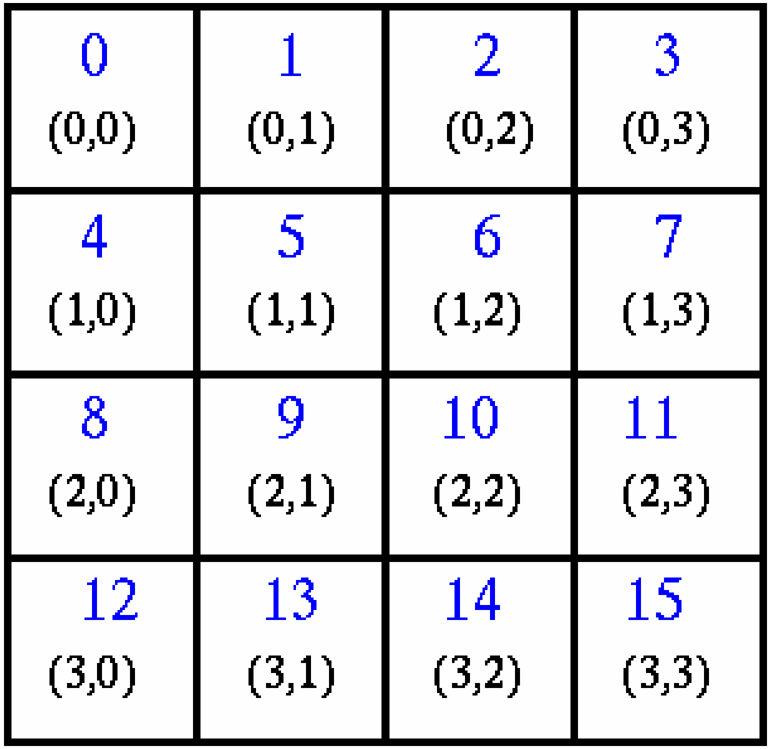
- In terms of MPI, a virtual topology describes a mapping/ordering of MPI processes into a g eometric "shape".
- The two main types of topologies supported by MPI are Cartesian (grid) and Graph.
- MPI topologies are virtual - there may be no relation between the physical structure of the parallel machine and the process topology.
- Virtual topologies are built upon MPI communicators and groups.
- Must be "programmed" by the application developer.

Why Use Virtual Topologies

- Convenience
  - Virtual topologies may be useful for applications with specific communication patterns - patterns that match an MPI topology structure.
  - For example, a Cartesian topology might prove convenient for an application that requires 4-way nearest neighbor communications for grid based data.
- Communication Efficiency
  - Some hardware architectures may impose penalties for communications between successively distant "nodes".
  - A particular implementation may optimize process mapping based upon the physical characteristics of a given parallel machine.
  - The mapping of processes into an MPI virtual topology is dependent upon the MPI implementation, and may be totally ignored.

Example : Cartesian Virtual Topology (1/3)

- A simplified mapping of processes into a Cartesian virtual topology appears below :



0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)
12 (3,0)	13 (3,1)	14 (3,2)	15 (3,3)

- Create a 4 x 4 Cartesian topology from 16 processors and have each process exchange its rank with four neighbors.

Example : Cartesian Virtual Topology (2/3)

```
#include "mpi.h"
#include <stdio.h>
#define SIZE 16
#define UP    0
#define DOWN  1
#define LEFT  2
#define RIGHT 3

int main(argc,argv)
int argc;
char *argv[];  {
int numtasks, rank, source, dest, outbuf, i, tag=1,
   inbuf[4]={MPI_PROC_NULL,MPI_PROC_NULL,MPI_PROC_NULL,MPI_PROC_NULL,},
   nbrs[4], dims[2]={4,4},
   periods[2]={0,0}, reorder=0, coords[2];

MPI_Request reqs[8];
MPI_Status stats[8];
MPI_Comm cartcomm;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

if (numtasks == SIZE) {
  MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder, &cartcomm);
  MPI_Comm_rank(cartcomm, &rank);
  MPI_Cart_coords(cartcomm, rank, 2, coords);
  MPI_Cart_shift(cartcomm, 0, 1, &nbrs[UP], &nbrs[DOWN]);
  MPI_Cart_shift(cartcomm, 1, 1, &nbrs[LEFT], &nbrs[RIGHT]);
```

Example : Cartesian Virtual Topology (3/3)

```
  printf("rank= %d coords= %d %d  neighbors(u,d,l,r)= %d %d %d %d\n",
         rank,coords[0],coords[1],nbrs[UP],nbrs[DOWN],nbrs[LEFT],
         nbrs[RIGHT]);

  outbuf = rank;

  for (i=0; i<4; i++) {
     dest = nbrs[i];
     source = nbrs[i];
     MPI_Isend(&outbuf, 1, MPI_INT, dest, tag,
               MPI_COMM_WORLD, &reqs[i]);
     MPI_Irecv(&inbuf[i], 1, MPI_INT, source, tag,
               MPI_COMM_WORLD, &reqs[i+4]);
     }

  MPI_Waitall(8, reqs, stats);

  printf("rank= %d                  inbuf(u,d,l,r)= %d %d %d %d\n",
         rank,inbuf[UP],inbuf[DOWN],inbuf[LEFT],inbuf[RIGHT]);   }
else
  printf("Must specify %d processors. Terminating.\n",SIZE);

MPI_Finalize();

return 0;

}
```

A Brief Word on MPI-2

- Intentionally, the MPI-1 specification did not address several "difficult" issues. For reasons of expediency, these issues were deferred to a second specification, called MPI-2 in 1997.
- MPI-2 was a major revision to MPI-1 adding new functionality and corrections.
- Key areas of new functionality in MPI-2:
  - Dynamic Processes - extensions that remove the static process model of MPI. Provides routines to create new processes after job startup.
  - One-Sided Communications - provides routines for one directional communications. Include shared memory operations (put/get) and remote accumulate operations.
  - Extended Collective Operations - allows for the application of collective operations to inter-communicators
  - External Interfaces - defines routines that allow developers to layer on top of MPI, such as for debuggers and profilers.
  - Additional Language Bindings - describes C++ bindings and discusses Fortran-90 issues.
  - Parallel I/O - describes MPI support for parallel I/O.

A Brief Word on MPI-3

- The MPI-3 standard was adopted in 2012, and contains signicant extensions to MPI-1 and

  MPI-2 functionality including:
  - Non-blocking Collective Operations - permits tasks in a collective to perform operation s without blocking, possibly offering performance improvements.
  - New one-sided communication operations - to better handle different memory models.
  - Neighborhood Collectives - Extends the distributed graph and Cartesian process topolo gies with additional communication power.
  - Fortran 2008 bindings - expanded from Fortran90 bindings
  - MPIT Tool Interface - This new tool interface allows the MPI implementation to expose certain internal variables, counters, and other states to the user (most likely performan ce tools).
  - Matched Probe - Fixes an old bug in MPI-2 where one could not probe for messages in a multi-threaded environment.

Any questions?