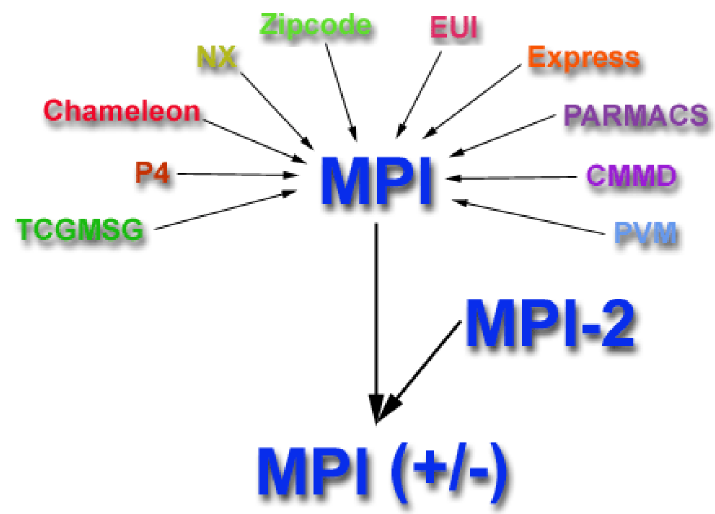
2110412 Parallel Comp Arch
Parallel Programming with MPI

Natawut Nupairoj, Ph.D.
Department of Computer Engineering, Chulalongkorn University

Overview

- MPI = Message Passing Interface
- Provide portable programming paradigm on existing development environments
  - Derived from several previous message-passing libraries
  - Versions for C/C++ and FORTRAN
  - Hide details of architecture (e.g. message passing, buffering)
  - Provides fundamental message management services



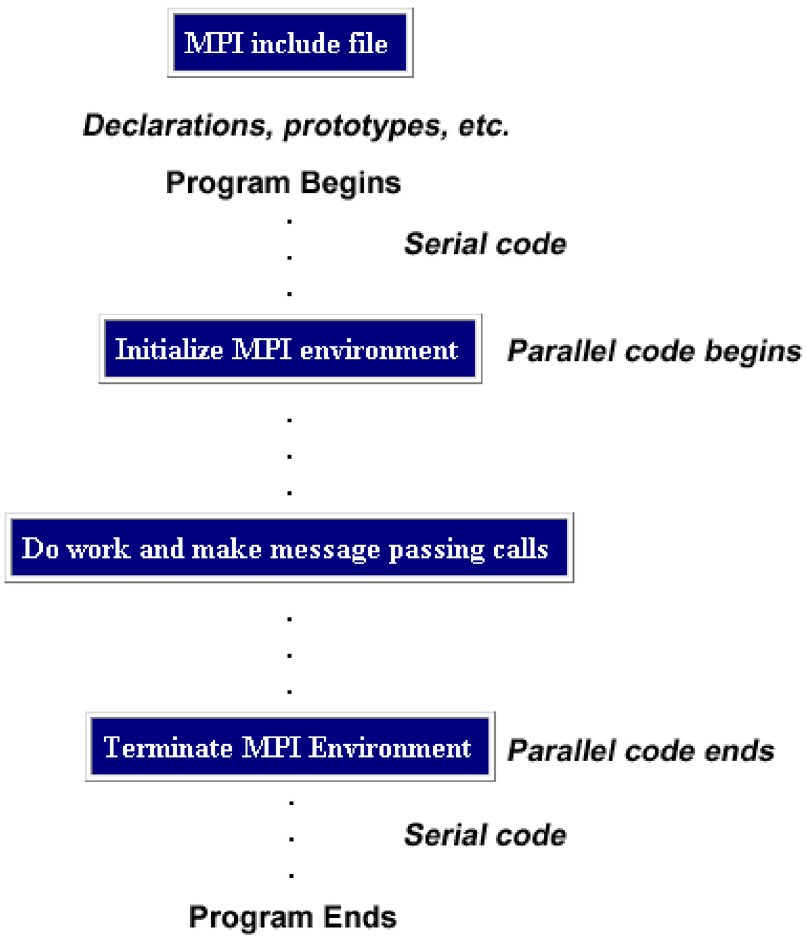
MPI History

- Late 1980s: vendors had unique libraries
- 1989: Parallel Virtual Machine (PVM) developed at Oak Ridge National Lab
- 1992: Work on MPI standard begun
- 1994: Version 1.0 of MPI standard
- 1997: Version 2.0 of MPI standard
- Today: MPI is dominant message passing library standard

MPI Programming Model

- Focus on distributed memory system
- Explicit parallelism
  - MPI provides standard message passing API (about 115 functions in MPI-1)
  - Programmer must identify the parallelism and call MPI functions to implement the parallel program
  - Program must follow MPI programming structure
- Number of tasks is static
  - Not dynamically spawn during run-time in MPI-1
  - MPI-2 supports dynamic tasks

MPI Programming Structure



- Start by including the “mpi.h” (standard header file)
- Initialize MPI environment with MPI_Init
- Call MPI functions to communicate between parallel tasks
- Terminate MPI environment with MPI_Finalize

MPI Initialize and Terminate

- Statement needed in every program before any other MPI code

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

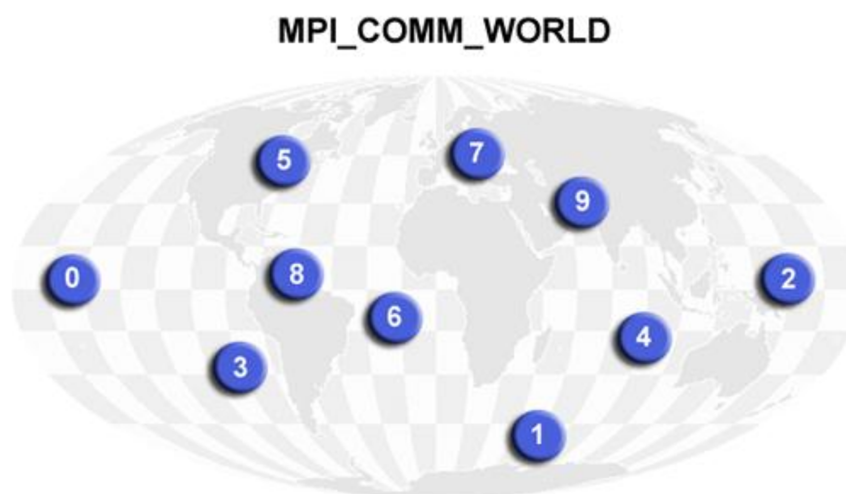
- Last statement of MPI code must be

```
MPI_Finalize();
```

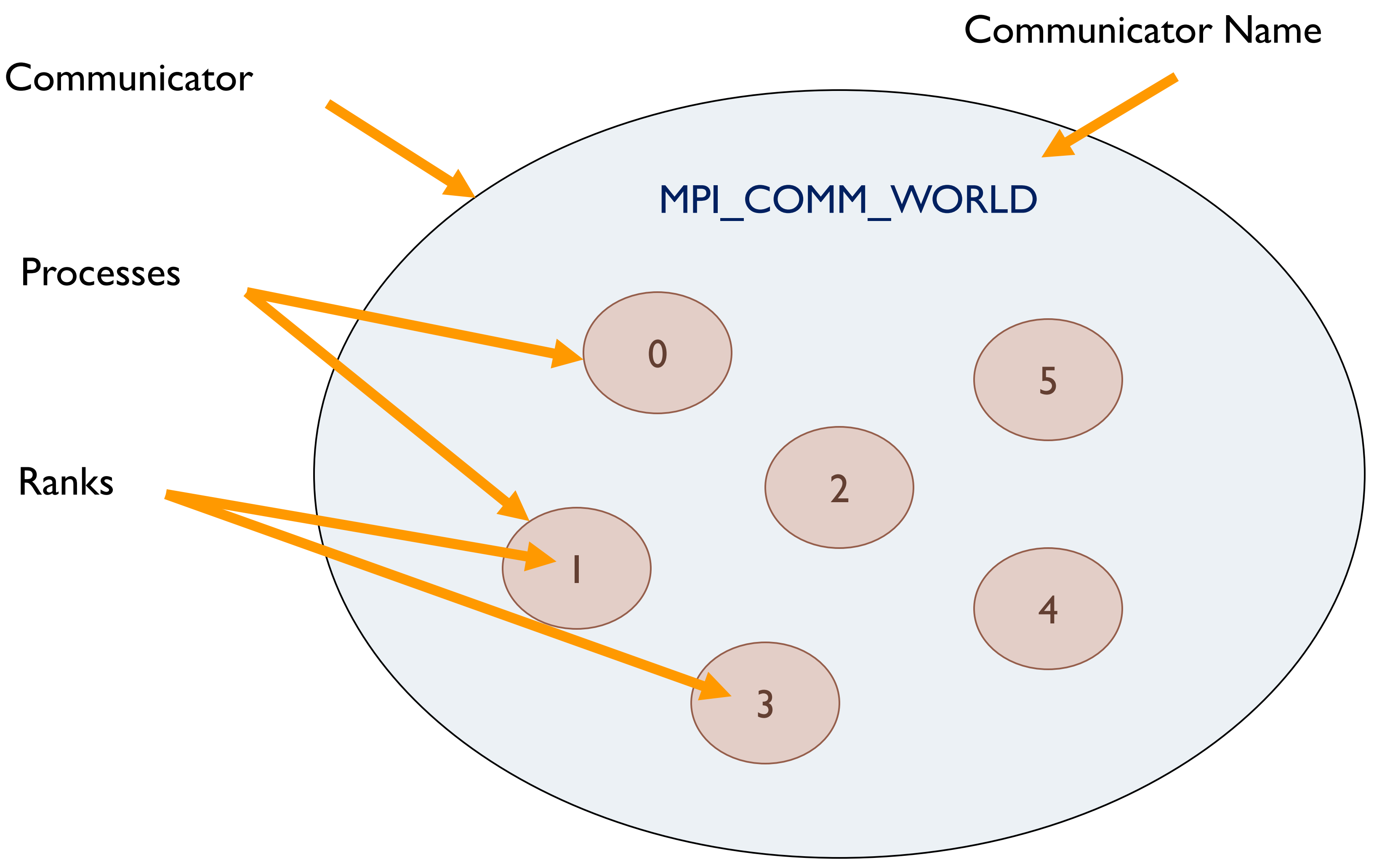
  - Program will not terminate without this statement

MPI Communication Model

- When process communicates, it must refer to communicator
- Communicator
  - Collection of processes
  - Determines scope to which messages are relative
  - identity of process (rank) is relative to communicator
  - scope of global communications (broadcast, etc.)
- MPI_COMM_WORLD = all processes



Communicator

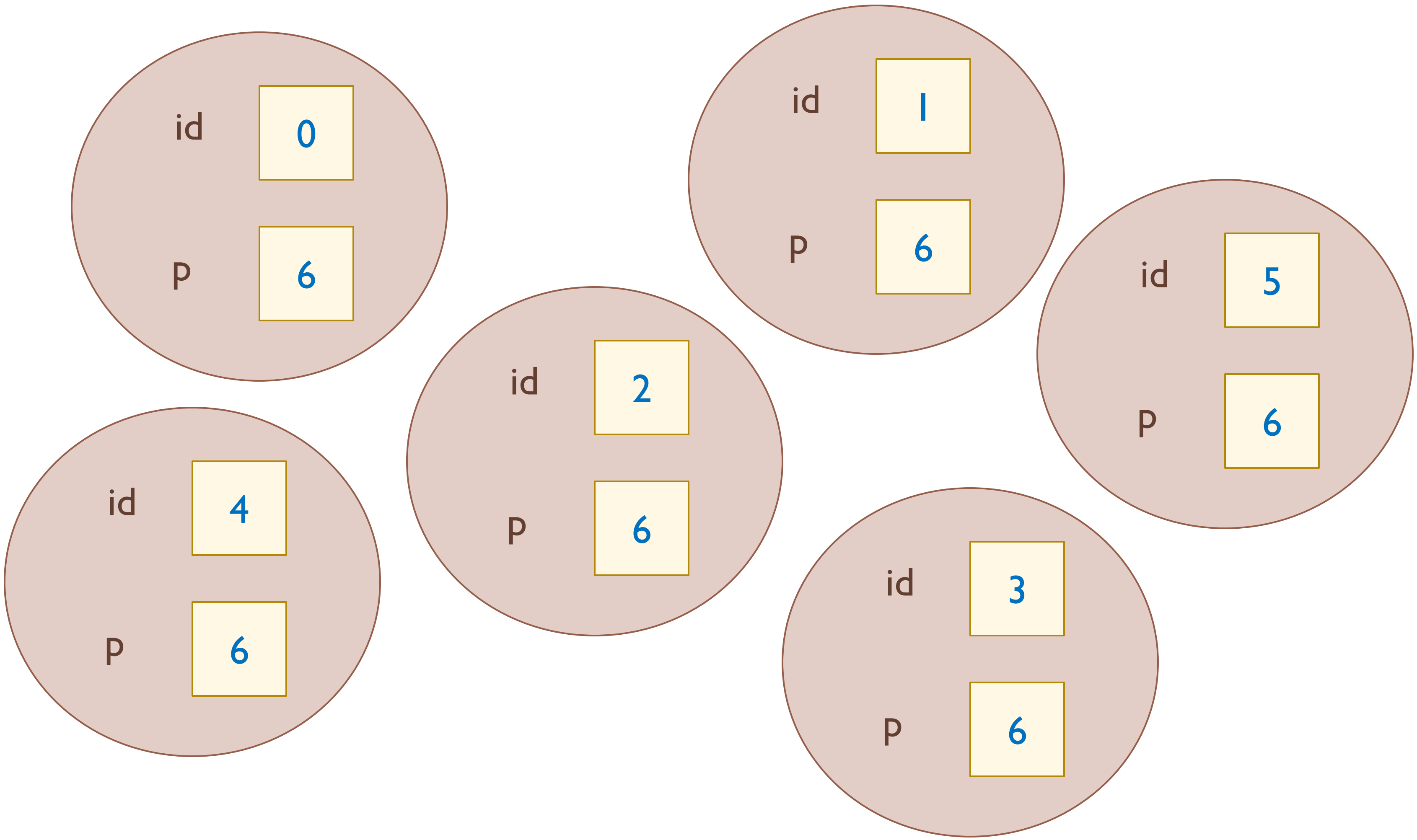


Process Rank and Size

- Unique, integer identifier assigned by the system to each process
- For specifying the source and destination of messages
- Contiguous and begin at zero
- 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, &id);

MPI_Comm_size (MPI_COMM_WORLD, &p);
```

Replication of Automatic Variables



Example – Simple MPI program

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

int main(argc,argv)
int argc;
char *argv[];
{
  int numtasks, rank;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  printf ("Number of tasks= %d My rank= %d\n",
       numtasks,rank);
  MPI_Finalize();
}
```

MPI - SPMD Computational Model

```
int main(int argc,char *argv[])
{
  MPI_Init(&argc,&argv);
  ...
  MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  ...
  if (rank == 0)
      /* Do some master work here */
  else
      /* Do some slave work here */
  ...
  MPI_Finalize();
}
```

MPI Communication Model

- Point-to-Point Communication
  - Send and receive messages between 2 processes
  - Exchange information one-to-one
- Collective Communication
  - Send and receive messages between group of processes
  - Synchronization and collaboration

MPI - Sending a Message with MPI_Send

```
MPI_Send(msg, count, type, dest, tag,
         MPI_COMM_WORLD);
```

message contents	block of memory
count	number of items in message
message type	type of each item
destination	rank of processor to receive
tag	integer designator for message
communicator	the communicator within which the message is sent

MPI_Datatype Options

- MPI_CHAR
- MPI_DOUBLE
- MPI_FLOAT
- MPI_INT
- MPI_LONG
- MPI_LONG_DOUBLE
- MPI_SHORT
- MPI_UNSIGNED_CHAR
- MPI_UNSIGNED
- MPI_UNSIGNED_LONG
- MPI_UNSIGNED_SHORT

MPI - Receiving a Message with MPI_Recv

```
MPI_Recv(msg, MAXSIZE, type, src, tag,
         MPI_COMM_WORLD, &status);
```

message contents	block of memory
count	size of buffer
message type	type of each item
source	rank of processor sending
tag	integer designator for message
communicator	the communicator within which the message is received
status	information about message received

Message Passing Example

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"     /* includes MPI library code specs */

#define MAXSIZE 100

int main(int argc, char* argv[])
{
  int  myRank;        /* rank (identity) of process      */
  int  numProc;       /* number of processors            */
  int  source;        /* rank of sender                  */
  int  dest;          /* rank of destination             */
  int  tag = 0;       /* tag to distinguish messages     */
  char msg[MAXSIZE];  /* message (other types possible) */
  int  count;         /* number of items in message      */
  MPI_Status status;  /* status of message received      */
```

Message Passing Example

```
  MPI_Init(&argc, &argv);          /* start MPI   */

  /* get number of processes  */
  MPI_Comm_size(MPI_COMM_WORLD, &numProc);

  /* get rank of this process */
  MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

  /************************************************/
  /*  code to send, receive and process messages */
  /************************************************/

  MPI_Finalize();                  /* shut down MPI */
}
```

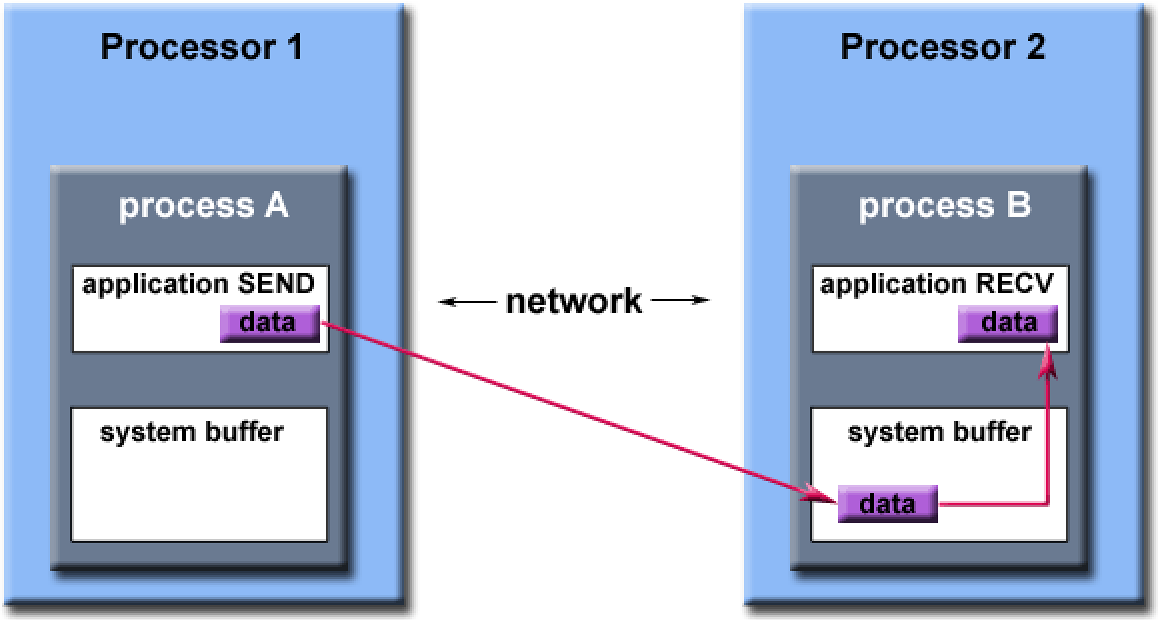
Message Passing Example

```
if (myRank != 0){/* all processes send to root        */

   /* create message */
   sprintf(msg, "Hello from %d", myRank);
   dest  = 0;                       /* destination is root  */
   count = strlen(msg) + 1;  /* include '\0' in message */

   MPI_Send(msg, count, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
 }
 else{/* root (0) process receives and prints messages */
      /* from each processor in rank order             */
   for(source = 1; source < numProc; source++){

     MPI_Recv(msg, MAXSIZE, MPI_CHAR,
              source, tag, MPICOMM_WORLD, &status);

     printf("%s\n", msg);
   }
 }
```

MPI Communication Mode

- Fully Synchronized (Rendezvous)
  - Send and Receive complete simultaneously
    - whichever code reaches the Send/Receive first waits
  - provides synchronization point (up to network delays)
- Buffered
  - Receive must wait until message is received
  - Send completes when message is moved to buffer clearing memory of message for reuse

MPI Communication Model



Path of a message buffered at the receiving process

MPI Communication Mode

- Asynchronous
  - Sending process may proceed immediately
    - does not need to wait until message is copied to buffer
    - must check for completion before using message memory
  - Receiving process may proceed immediately
    - will not have message to use until it is received
    - must check for completion before using message

MPI Send and Receive

- **MPI_Send/MPI_Recv** are synchronous, but buffering is unspecified
  - **MPI_Recv** suspends until message is received
  - **MPI_Send** may be fully synchronous or may be buffered
    - implementation dependent
- Variations allow synchronous or buffering to be specified
  - **MPI_Ssend**
  - **MPI_Bsend**
  - **MPI_Rsend**

Asynchronous Send and Receive

- **`MPI_Isend() / MPI_Irecv()`** are non-blocking. Control returns to program after call is made.
- Syntax is the same as for Send and Recv, except a MPI_Request* parameter is added to Isend and replaces the MPI_Status* for receive.

Detecting Completion

- **`MPI_Wait(&request, &status)`**
  - **`request`** matches request on **`Isend`** or **`Irecv`**
  - **`status`** returns status equivalent to

    status for **`Recv`** when complete
  - Blocks for send until message is buffered or sent so message variable is free
  - Blocks for receive until message is received and ready

Detecting Completion

- **`MPI_Test(&request, flag, &status)`**
  - **`request, status`** as for **`MPI_Wait`**
  - does not block
  - flag indicates whether message is sent/received
  - enables code which can repeatedly check for communication completion

Collective Communication

- Point-to-Point communication
  - single sender and single receiver
  - One-to-One
- Collective communication
  - multiple sender and/or multiple receiver
  - One-to-Many
  - Many-to-One
  - Many-to-Many

Broadcasting a message

- Broadcast: one sender, many receivers
- Includes all processes in communicator, all processes must make an equivalent call to MPI_Bcast
- Any processor may be sender (root), as determined by the fourth parameter
- First three parameters specify message as for MPI_Send and MPI_Recv, fifth parameter specifies communicator
- Broadcast serves as a global synchronization

MPI_Bcast() Syntax

```
MPI_Bcast(msg, count, MPI_INT, root,
          MPI_COMM_WORLD);
```

msg	pointer to message buffer
count	number of items sent
MPI_INT	type of item sent
root	sending processor
MPI_COMM_WORLD	communicator within which broadcast takes place

Note: count and type should be the same on all processors

Reduce

- All Processors send to a single processor, the reverse of broadcast
- Information must be combined at receiver
- Several combining functions available
  - MAX, MIN, SUM, PROD, LAND, BAND, LOR, BOR, LXOR, BXOR, MAXLOC, MINLOC

MPI_Reduce() syntax

```
MPI_Reduce(&dataIn, &result, count,
           MPI_DOUBLE, MPI_SUM, root,
                MPI_COMM_WORLD);
```

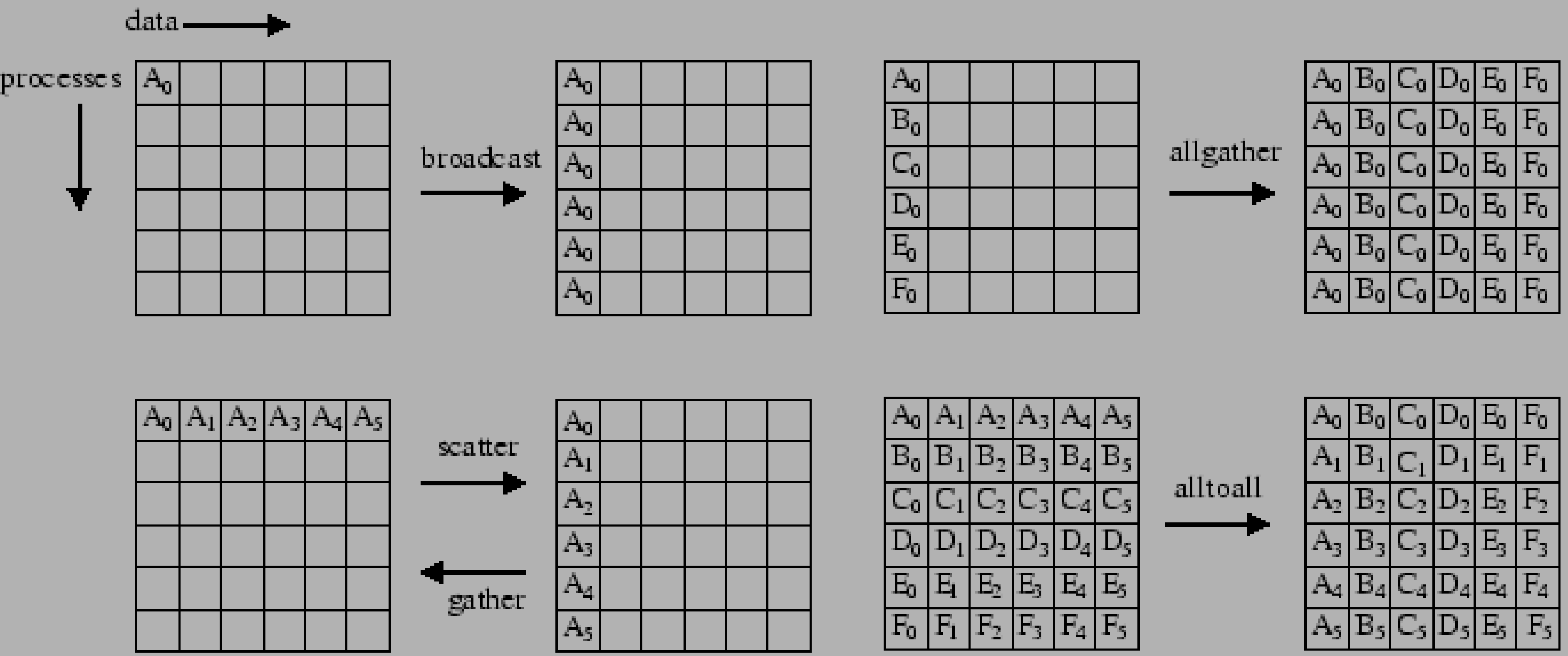
`dataIn`	data sent from each processor
`result`	stores result of combining operation
`count`	number of items in each of dataIn, result
`MPI_DOUBLE`	data type for **`dataIn`**, result
`MPI_SUM`	combining operation
`root`	rank of processor receiving data
`MPI_COMM_WORLD`	communicator

Example – Finding PI with MPI

- For simplicity, we will approximate PI with integral
  - PI = sum of "n" intervals
  - Each interval = (1/n)*4/(1+x*x)

$$PI = \int_{-\frac{1}{2}}^{\frac{1}{2}} \frac{4}{1+x^2}\,dx$$

- To implement in parallel
  - Rank 0 is the master process and others are the work processes
  - Master broadcasts "n" to all workers
  - Each process adds up "x" every n'th interval
    - (-1/2+rank/n, -1/2+rank/n+size/n,...).
  - Master sums all the results with reduction


data
processes
broadcast
allgather
scatter
gather
alltoall

MPI_Barrier()

```
MPI_Barrier(MPI_COMM_WORLD);
```

`MPI_COMM_WORLD` communicator within which broadcast takes place

provides for barrier synchronization without message of broadcast

Timing Programs

- MPI_Wtime()
  - returns a double giving time in seconds from a fixed time in the past
  - To time a program, record MPI_Wtime() in a variable at start, then again at finish, difference is elapsed time

```
startime = MPI_Wtime();
/* part of program to be timesd  */
stoptime = MPI_Wtime();
time = stoptime - starttime;
```

How to Build MPI on Windows XP

- Requirements
  - Microsoft Compute Cluster Pack SDK
    - http://www.microsoft.com/downloads/details.aspx?FamilyID=d8462378-2f68-409d-9cb3-02312bc23bfd&displaylang=en
  - Your favorite editor and C compiler
    - If you are using Visual Studio, please see http://www.cs.utah.edu/~delisi/vsmpi/
- Build your MPI program
- Running program
  - e.g. 3 tasks of test.exe

mpiexec –n 3 test

Assignment

- Writing an MPI program for Sorting "n" Number
  - Process rank 0 is the master, others are workers
  - Master accepts "n" from keyboard
  - Master randoms "n" integer numbers
  - Master coordinates with workers to sort these randomized numbers
  - You must measure the elapsed time for sorting
- Due date: 8 January 2010 at 18:00
- How to submit: sending email to "natawut.n@chula.ac.th"
- Note: I will use timestamp on your email