ĐẠI HỌC BÁCH KHOA

25 YEARS ANNIVERSARY SOICT

HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

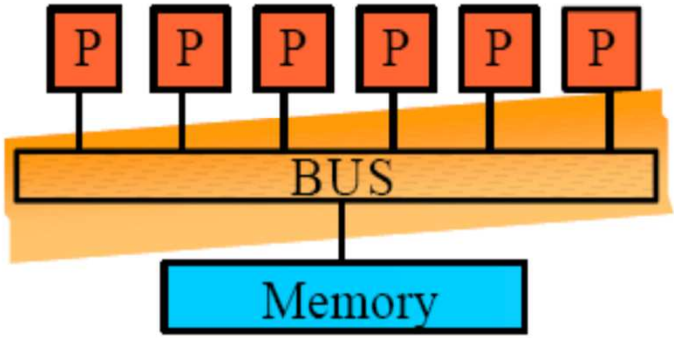
Message-Passing Programming (MPI)

References

- Michael J. Quinn. **Parallel Computing. Theory and Practice**. McGraw-Hill
- Albert Y. Zomaya. **Parallel and Distributed Computing Handbook**. McGraw-Hill
- Ian Foster. **Designing and Building Parallel Programs**. Addison-Wesley.
- Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar **.** **Introduction to Parallel Computing, Second Edition.** Addison Wesley.
- Joseph Jaja**. An Introduction to Parallel Algorithm.** Addison Wesley.
- Nguyễn Đức Nghĩa**. Tính toán song song.** Hà Nội 2003.

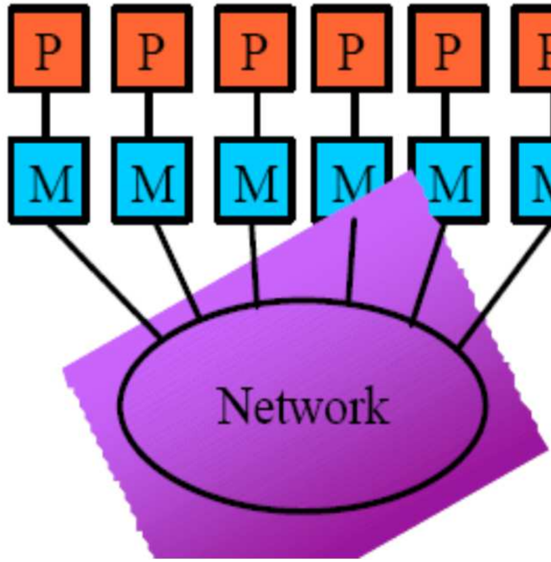
3.1 MPI Parallel Programming Model

Shared vs. Distributed Memory



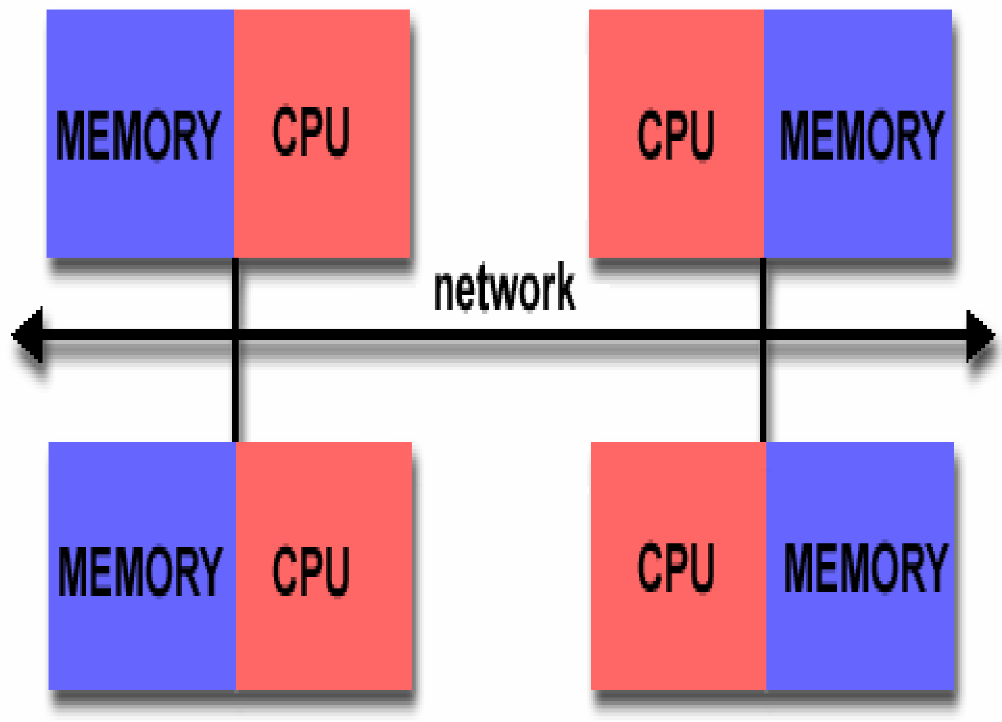
Shared memory - single address space. All processors have access to a pool of shared memory. (Ex: SGI Origin, Sun E10000)

Distributed memory - each processor has it's own local memory. Must do message passing to exchange data between processors. (Ex: CRAY T3E, IBM SP, clusters)



Distributed Memory

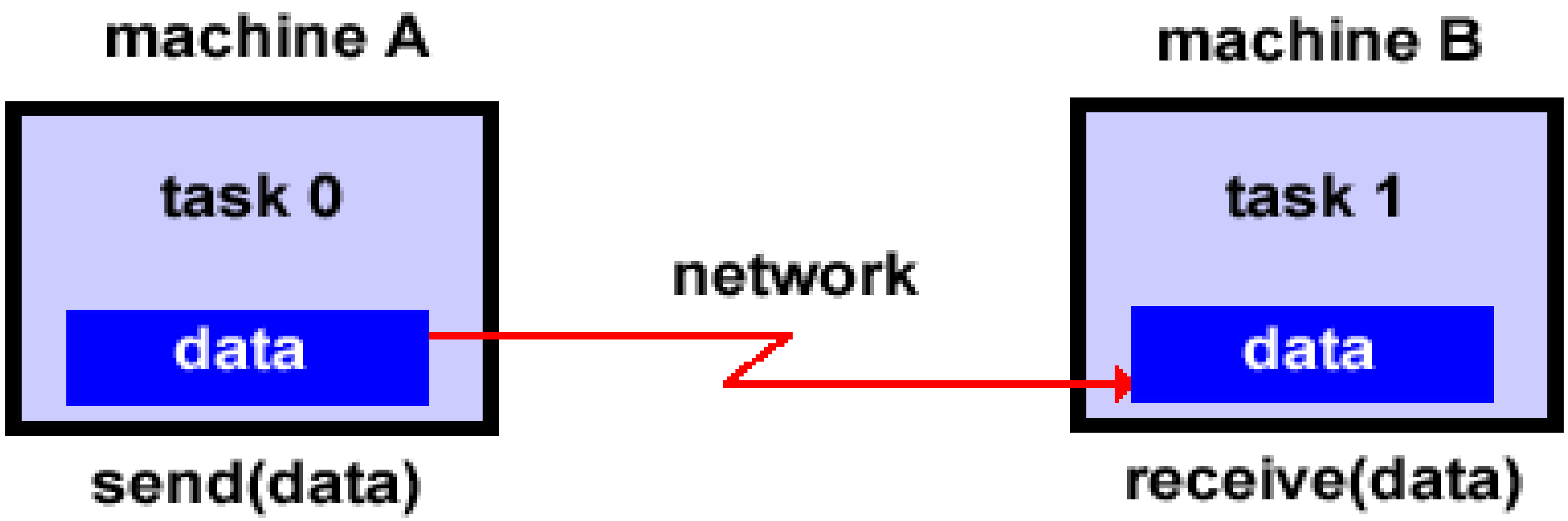
- Each CPU has private memory.
- CPU can not access to the other CPU's private memory



Message Passing Model

- The message passing model demonstrates the following characteristics:
  - A set of tasks that use their own local memory during computation. Multiple tasks can reside on the same physical machine as well across an arbitrary number of machines.
  - Tasks exchange data through communications by sending and receiving messages.
  - Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation.

Message Passing Model



Message Passing

- A process is a program counter and address space.

- Message passing is used for communication among processes.

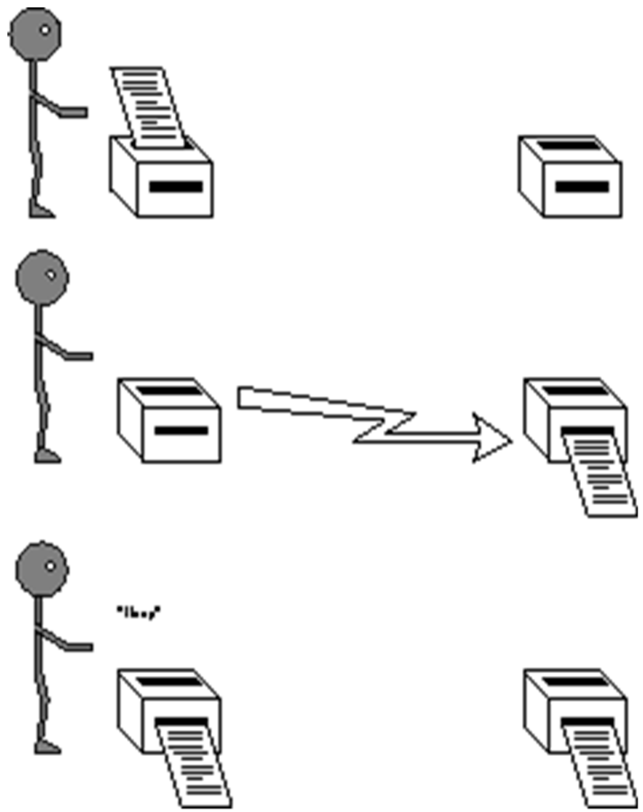
- Inter-process communication:
  - Type:
    Synchronous / Asynchronous
  - Movement of data from one process's address space to another's

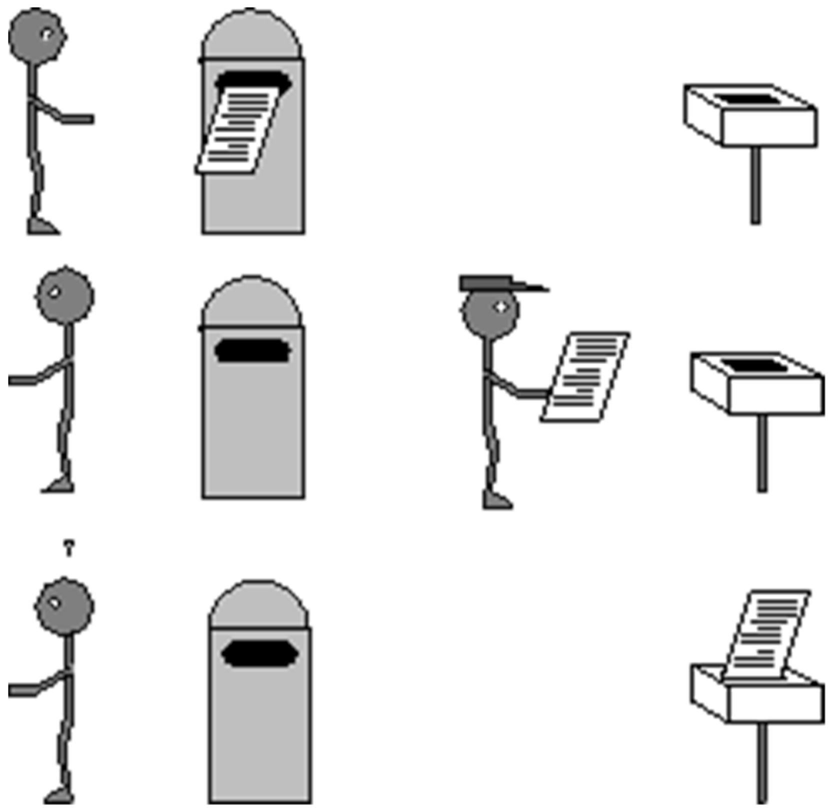
Synchronous Vs. Asynchronous

- A synchronous communication is not complete until the message has been received.

- An asynchronous communication completes as soon as the message is on the way.

Synchronous Vs. Asynchronous (cont.)





What is message passing?

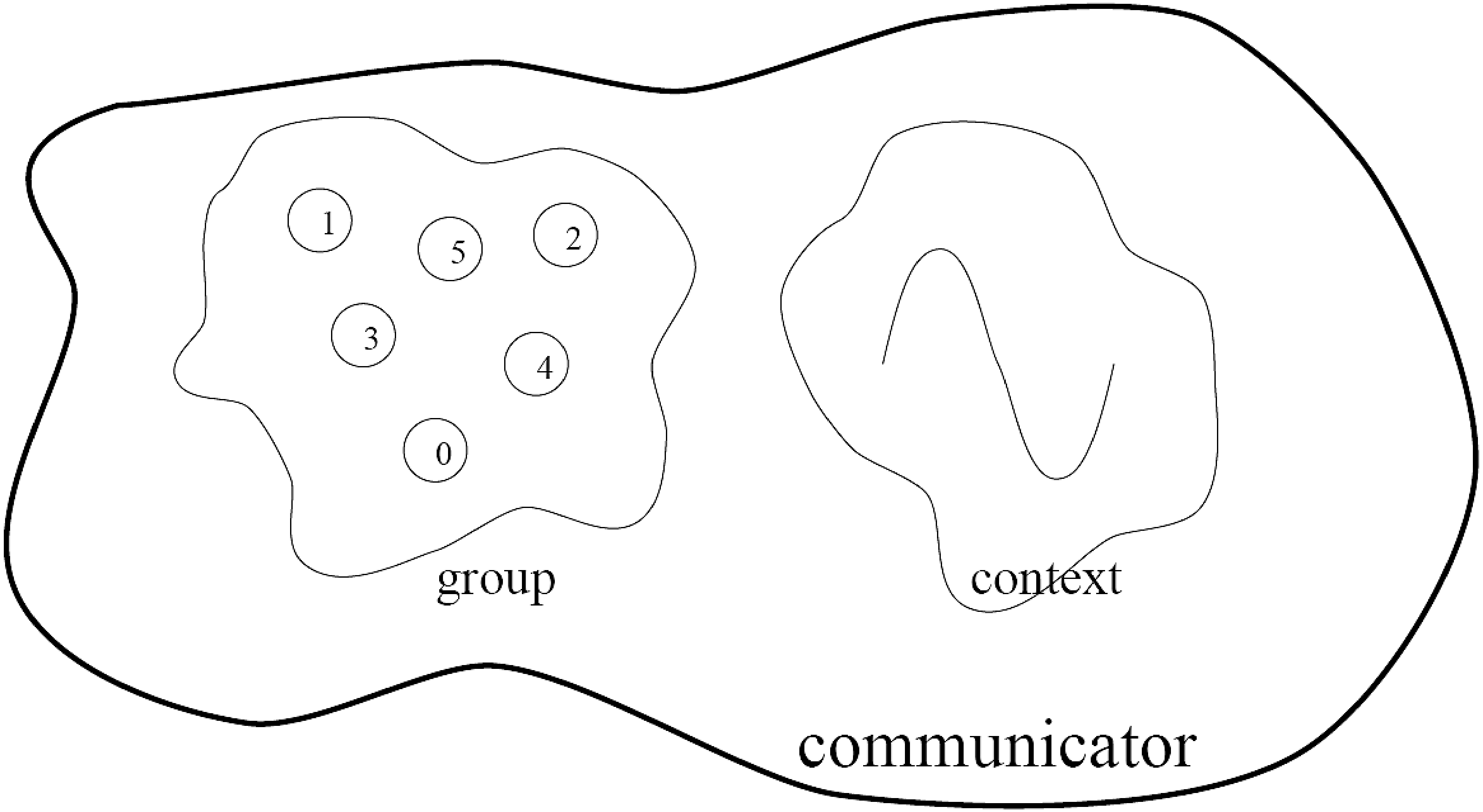
- Data transfer.
- Requires cooperation of sender and receiver
- Cooperation not always apparent in code

What is MPI Libs?

- A message-passing library specifications:
  - Extended message-passing model
  - Not a language or compiler specification
  - Not a specific implementation or product
- For parallel computers, clusters, and heterogeneous networks.
- Communication modes: *standard*, *synchronous*, *buffered,* and *ready*.
- Designed to permit the development of parallel software libraries.
- Designed to provide access to advanced parallel hardware for
  - End users
  - Library writers
  - Tool developers

3.2 Synchronization and Communication

Group and Context



Group and Context (cont.)

- Are two important and indivisible concepts of MPI.
- Group: is the set of processes that communicate with one another.
- Context: it is somehow similar to the frequency in radio communications.
- Communicator: is the central object for communication in MPI. Each communicator is associated with a group and a context.

Communication Modes

- Based on the type of send:
  - Synchronous: Completes once the acknowledgement is received by the sender.
  - Buffered send: completes immediately, unless if an error occurs.
  - Standard send: completes once the message has been sent, which may or may not imply that the message has arrived at its destination.
  - Ready send: completes immediately, if the receiver is ready for the message it will get it, otherwise the message is dropped silently.

Blocking vs. Non-Blocking

- Blocking, means the program will not continue until the communication is completed.

- Non-Blocking, means the program will continue, without waiting for the communication to be completed.

Features of MPI

- General
  - Communications combine context and group for message security.
  - Thread safety can’t be assumed for MPI programs.

Features of MPI (2)

- Communicator Information
- Point to Point communication
- Collective Communication
- Topology Support
- Error Handling

Features that are NOT part of MPI

- Process Management
- Remote memory transfer
- Threads
- Virtual shared memory

MPI Programming Structure

- Asynchronous
  - Hard to reason
  - Non-deterministic behavior
- Loosely synchronous
  - Synchronize to perform interactions
  - Easier to reason
- SPMD
  - **S**ingle **P**rogram **M**ultiple **D**ata

Why to use MPI?

- MPI provides a powerful, efficient, and portable way to express parallel programs.

- MPI was explicitly designed to enable libraries which may eliminate the need for many users to learn (much of) MPI.

- Portable !!!!!!!!!!!!!!!!!!!!!!!!!!

- Good way to learn about subtle issues in parallel computing

How big is the MPI library?

- Huge (125 Functions).

- Basic (6 Functions).

Blocking Communication

Six Golden MPI Functions

The minimal set of MPI routines.

`MPI_Init`	Initializes MPI.
`MPI_Finalize`	Terminates MPI.
`MPI_Comm_size`	Determines the number of processes.
`MPI_Comm_rank`	Determines the label of the calling process.
`MPI_Send`	Sends a message.
`MPI_Recv`	Receives a message.

Skeleton MPI Program

```
#include <mpi.h>

main( int argc, char** argv )
{
    MPI_Init( &argc, &argv );

    /* main part of the program */

  /*
    Use MPI function call depend on your data
  partitioning and the parallelization
  architecture
  */

    MPI_Finalize();
}
```

Initializing MPI

- The initialization routine MPI_INIT is the first MPI routine called.

- MPI_INIT is called once

```
int mpi_Init( int *argc, char **argv );
```

A minimal MPI program(c)

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
  MPI_Init(&argc, &argv);
  printf("Hello, world!\n");
  MPI_Finalize();
  Return 0;
}
```

A minimal MPI program(c) (cont.)

- #include “mpi.h” provides basic MPI definitions and types.
- MPI_Init starts MPI
- MPI_Finalize exits MPI
- Note that all non-MPI routines are local; thus “printf” run on each process
- Note: MPI functions return error codes or MPI_SUCCESS

Compile and run the code

- Compile using:

  mpicc –o pi pi.c

  Or

  mpic++ –o pi pi.cpp
- mpirun –np # of procs –machinefile XXX pi

- -machinefile tells MPI to run the program on the machines of XXX.

Error handling

- By default, an error causes all processes to abort.
- The user can have his/her own error handling routines.
- Some custom error handlers are available for downloading from the net.

Improved Hello (c)

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
   int rank, size;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   printf("I am %d of %d\n", rank, size);
   MPI_Finalize();
   return 0;
}
```

Some concepts

- The default communicator is the `MPI_COMM_WORLD`

- A process is identified by its rank in the group associated with a communicator.

Data Types

- The data message which is sent or received is described by a triple (address, count, datatype).
- The following data types are supported by MPI:
  - Predefined data types that are corresponding to data types from the programming language.
  - Arrays.
  - Sub blocks of a matrix
  - User defined data structure.
  - A set of predefined data types

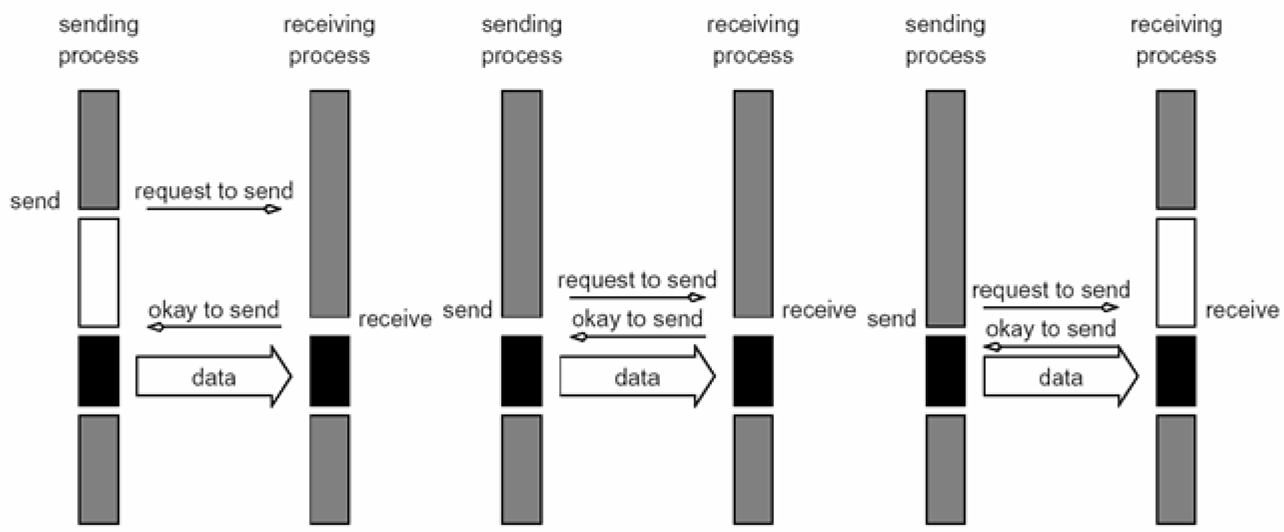
Basic MPI types

MPI datatype	**C datatype**
MPI_CHAR	signed char
MPI_SIGNED_CHAR	signed char
MPI_UNSIGNED_CHAR	unsigned char
MPI_SHORT	signed short
MPI_UNSIGNED_SHORT	unsigned short
MPI_INT	signed int
MPI_UNSIGNED	unsigned int
MPI_LONG	signed long
MPI_UNSIGNED_LONG	unsigned long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

Why defining the data types during the send of a message?

Because communications take place between heterogeneous machines. Which may have different data representation and length in the memory.

Blocking Non-Buffered Communication



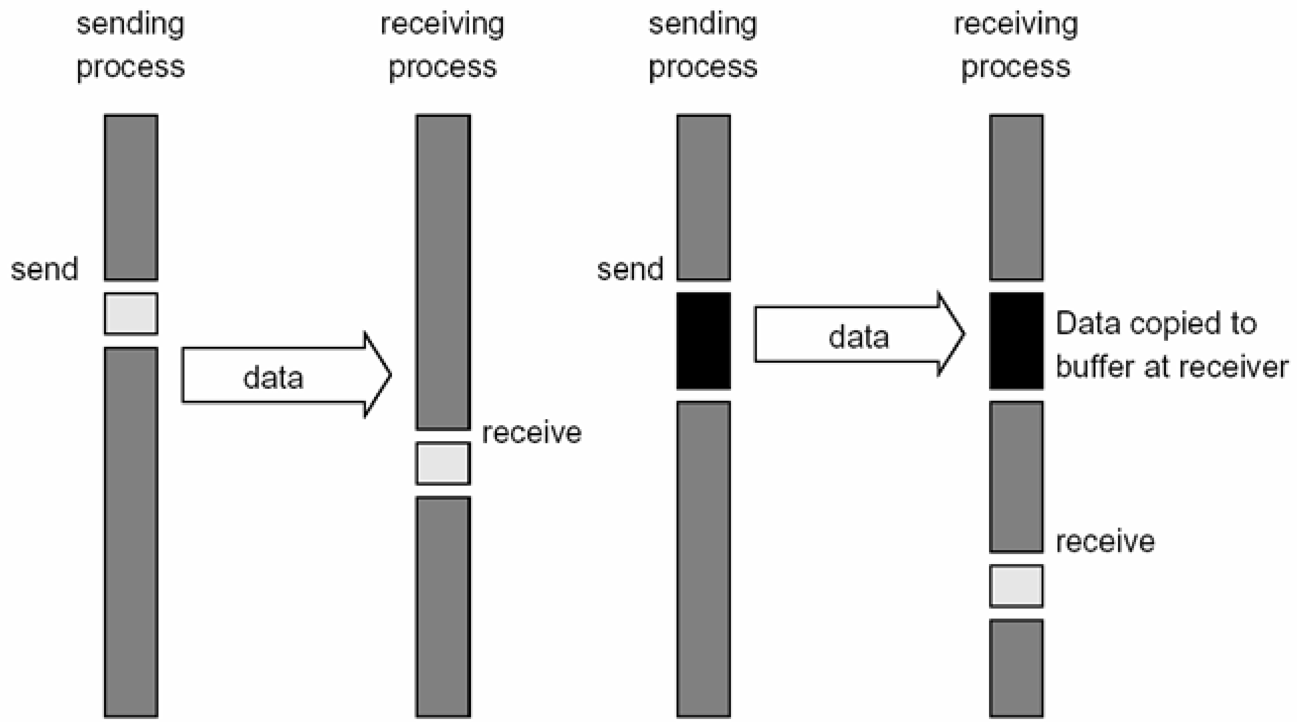
(a) Sender comes first; idling at sender

(b) Sender and receiver come at about the same time; idling minimized

(c) Receiver comes first; idling at receiver

Handshake for a blocking non-buffered send/receive operation. It is easy to see that in cases where sender and receiver do not reach communication point at similar times, there can be considerable idling overheads.

Blocking Buffered Communication



Blocking buffered transfer protocols: (a) in the presence of communication hardware with buffers at send and receive ends; and (b) in the absence of communication hardware, sender interrupts receiver and deposits data in buffer at receiver end.

MPI blocking send

```
MPI_SEND(void *start, int
  count,MPI_DATATYPE datatype, int dest,
  int tag, MPI_COMM comm)
```

- The message buffer is described by (start, count, datatype).
- dest is the rank of the target process in the defined communicator.
- tag is the message identification number.

MPI blocking receive

```
MPI_RECV(void *start, int count, MPI_DATATYPE
datatype, int source, int tag, MPI_COMM comm,
MPI_STATUS *status)
```

- Source is the rank of the sender in the communicator.
- The receiver can specify a wildcard value for souce (MPI_ANY_SOURCE) and/or a wildcard value for tag (MPI_ANY_TAG), indicating that any source and/or tag are acceptable
- Status is used for exrtra information about the received message if a wildcard receive mode is used.
- If the count of the message received is less than or equal to that described by the MPI receive command, then the message is successfully received. Else it is considered as a buffer overflow error.

MPI_STATUS

- Status is a data structure
- In C:

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

More info

- A receive operation may accept messages from an arbitrary sender, but a send operation must specify a unique receiver.

- Source equals destination is allowed, that is, a process can send a message to itself.

Why MPI is simple?

- Many parallel programs can be written using just these six functions, only two of which are non-trivial;
    - MPI_INIT
    - MPI_FINALIZE
    - MPI_COMM_SIZE
    - MPI_COMM_RANK
    - MPI_SEND
    - MPI_RECV

Simple full example

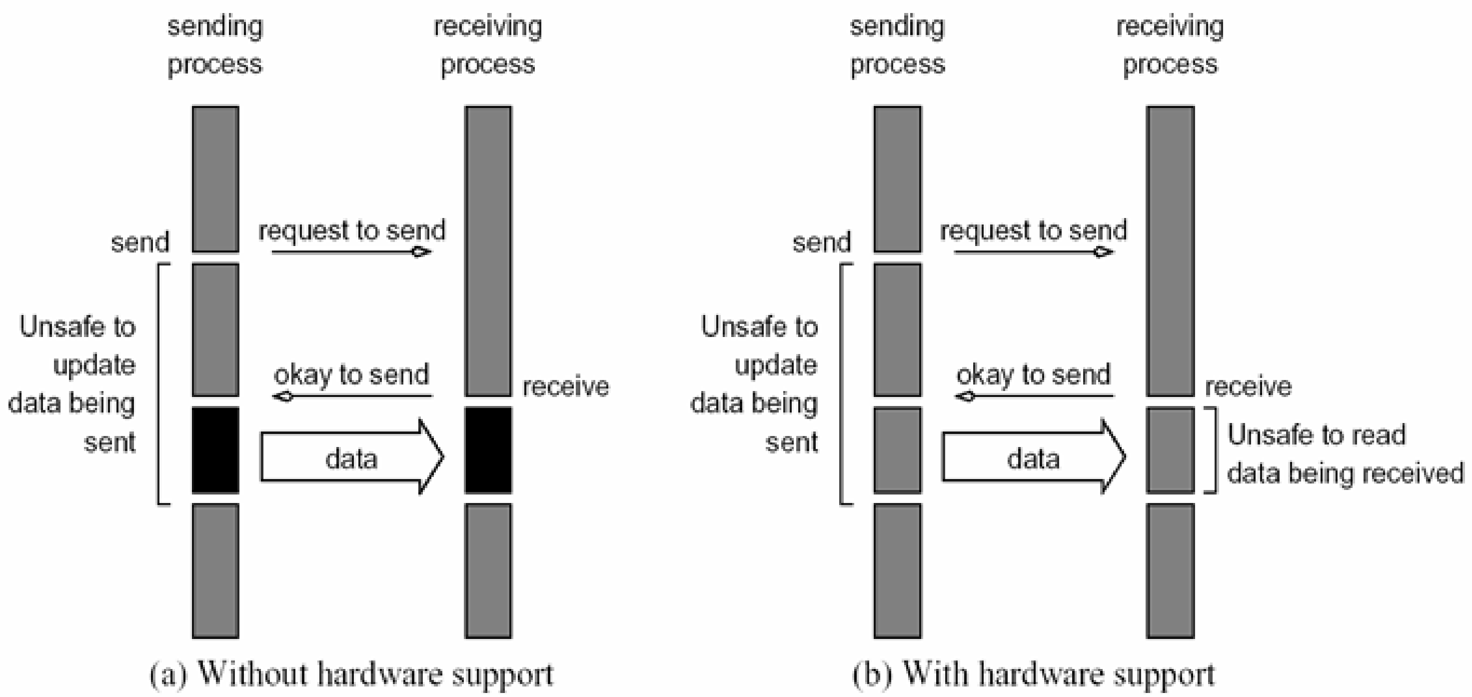
```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char *argv[])
{
  const int tag = 42;         /* Message tag */
  int id, ntasks, source_id, dest_id, err, i;
  MPI_Status status;
  int msg[2]; /* Message array */

  err = MPI_Init(&argc, &argv); /* Initialize MPI */
  if (err != MPI_SUCCESS) {
    printf("MPI initialization failed!\n");
    exit(1);
  }
  err = MPI_Comm_size(MPI_COMM_WORLD, &ntasks); /* Get nr of tasks */
  err = MPI_Comm_rank(MPI_COMM_WORLD, &id);  /* Get id of this process */
  if (ntasks < 2) {
    printf("You have to use at least 2 processors to run this program\n");
    MPI_Finalize();    /* Quit if there is only one processor */
    exit(0);
  }
```

Simple full example (Cont.)

```
if (id == 0) {  /* Process 0 (the receiver) does this */
    for (i=1; i<ntasks; i++) {
      err = MPI_Recv(msg, 2, MPI_INT, MPI_ANY_SOURCE, tag, MPI_COMM_WORLD, \
                     &status);          /* Receive a message */
      source_id = status.MPI_SOURCE; /* Get id of sender */
      printf("Received message %d %d from process %d\n", msg[0], msg[1], \
             source_id);
    }
  }
  else {    /* Processes 1 to N-1 (the senders) do this */
    msg[0] = id; /* Put own identifier in the message */
    msg[1] = ntasks;        /* and total number of processes */
    dest_id = 0; /* Destination address */
    err = MPI_Send(msg, 2, MPI_INT, dest_id, tag, MPI_COMM_WORLD);
  }

  err = MPI_Finalize();          /* Terminate MPI */
  if (id==0) printf("Ready\n");
  exit(0);
  return 0;
}
```

Non-Blocking Communication

Non-Blocking Non-Buffered Communication



Non-blocking non-buffered send and receive operations (a) in absence of communication hardware; (b) in presence of communication hardware.

Non-Blocking Send and Receive

```
MPI_ISEND(buf, count, datatype, dest, tag, comm,
  request)
MPI_IRECV(buf, count, datatype, dest, tag, comm,
  request)
```

- request is a request handle which can be used to query the status of the communication or wait for its completion.

Non-Blocking Send and Receive (Cont.)

- A non-blocking send call indicates that the system may start copying data out of the send buffer. The sender must not access any part of the send buffer after a non-blocking send operation is posted, until the complete-send returns.
- A non-blocking receive indicates that the system may start writing data into the receive buffer. The receiver must not access any part of the receive buffer after a non-blocking receive operation is posted, until the complete-receive returns.

Non-Blocking Send and Receive (Cont.)

```
MPI_WAIT (request, status)
MPI_TEST (request, flag, status)
```

- The MPI_WAIT will block your program until the non-blocking send/receive with the desired request is done.

- The MPI_TEST is simply queried to see if the communication has completed and the result of the query (TRUE or FALSE) is returned immediately in flag.

Deadlocks in blocking operations

- What happens with

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

- Send a large message from process 0 to process 1
  - If there is insufficient storage at the destination, the send must wait for the user to provide the memory space(through a receive)
- This is called "unsafe" because it depends on the availability of system buffers.

Some solutions to the "unsafe" problem

- Order the operations more carefully

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

Use non-blocking operations:

Process 0	Process 1
ISend(1)	ISend(0)
IRecv(1)	IRecv(0)
Waitall	Waitall

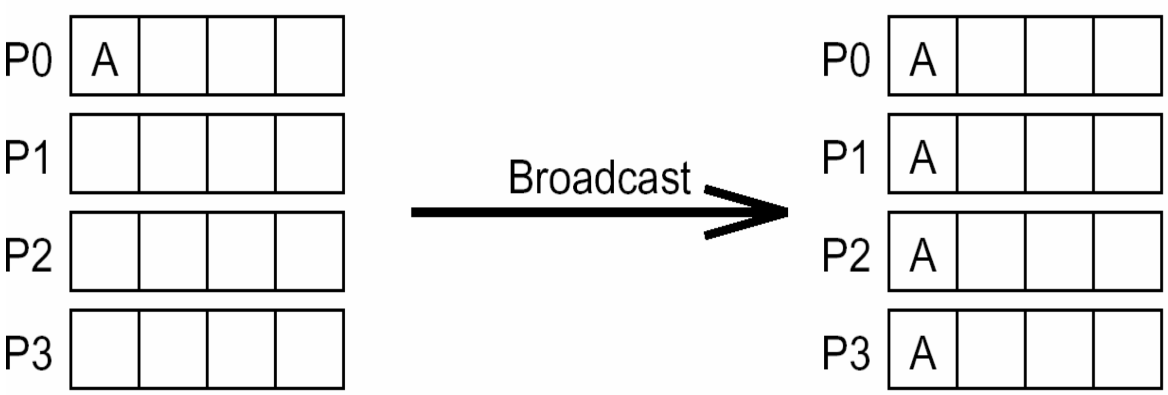
MPI Functions: Synchronization

```
int MPI_Barrier(MPI_Comm comm)
```

Collective Communications

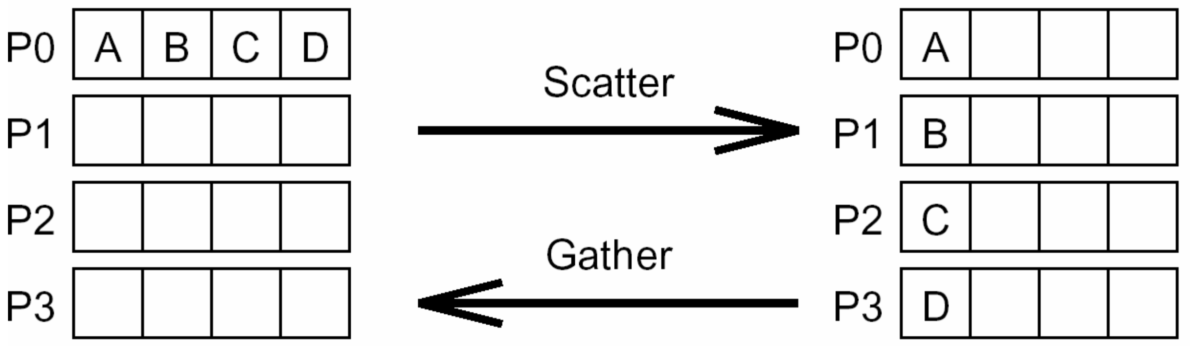
- One-to-All Broadcast
- All-to-One Reduction
- All-to-All Broadcast & Reduction
- All-Reduce & Prefix-Sum
- Scatter and Gather
- All-to-All Personalized

MPI Functions: Broadcast



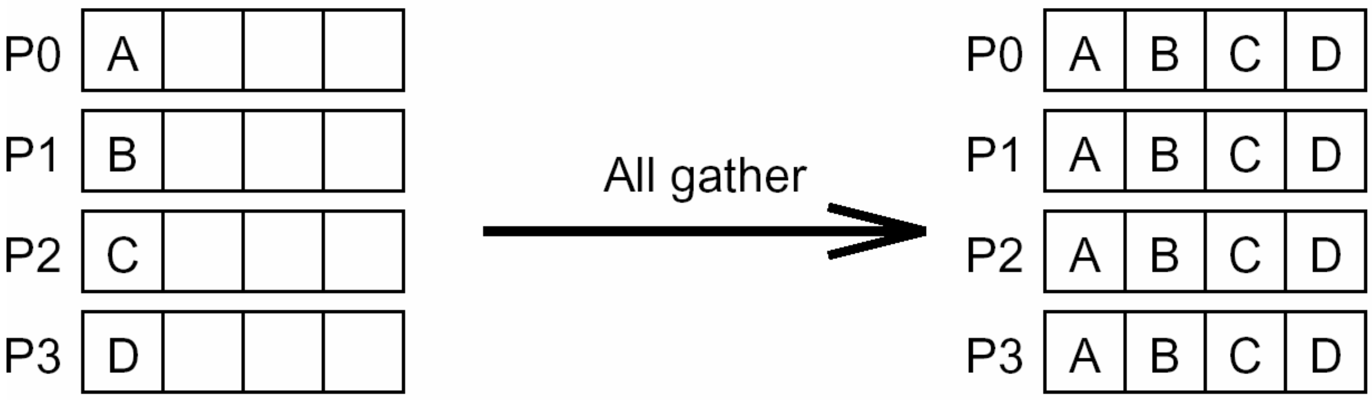

```
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype,
        int source, MPI_Comm comm)
```

MPI Functions: Scatter & Gather



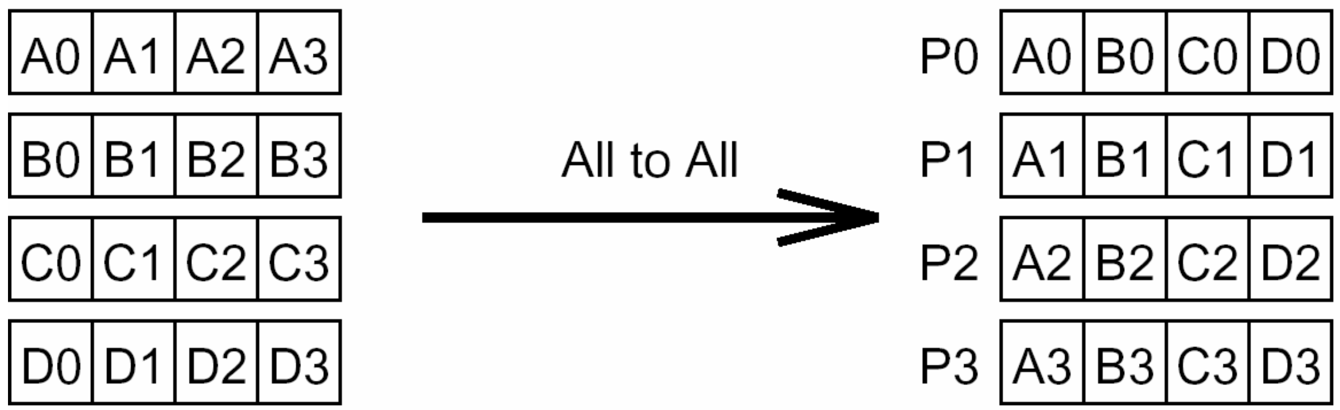

```
int MPI_Scatter(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf, int recvcount,
        MPI_Datatype recvdatatype, int source, MPI_Comm comm)

int MPI_Gather(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf, int recvcount,
        MPI_Datatype recvdatatype, int target, MPI_Comm comm)
```

MPI Functions: All Gather



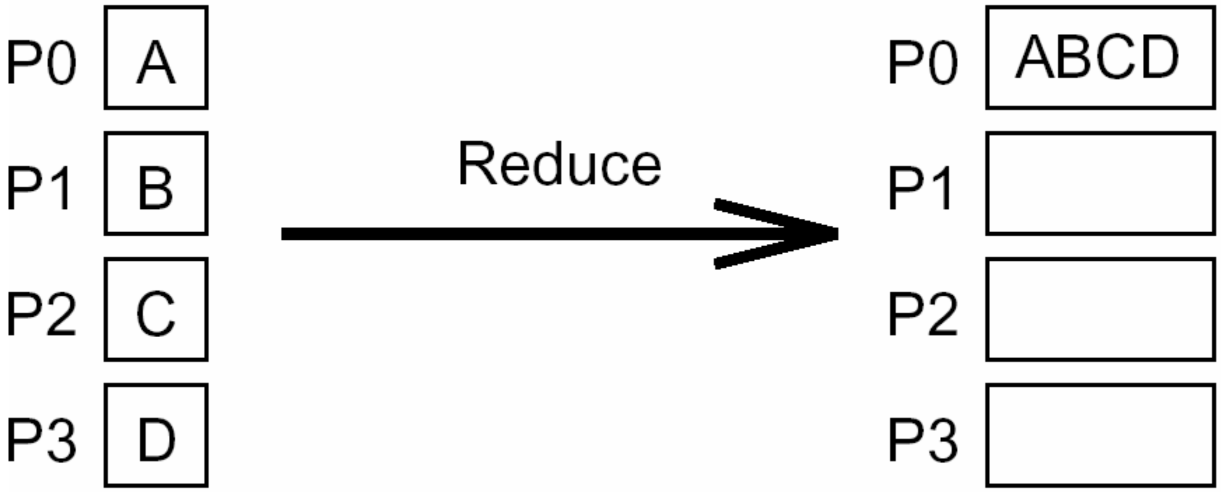

```
int MPI_Allgather(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf, int recvcount,
        MPI_Datatype recvdatatype, MPI_Comm comm)
```

MPI Functions: All-to-All Personalized




```
int MPI_Alltoall(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf, int recvcount,
        MPI_Datatype recvdatatype, MPI_Comm comm)
```

MPI Functions: Reduction




```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
        MPI_Datatype datatype, MPI_Op op, int target,
        MPI_Comm comm)
```

MPI Functions: Operations

Predefined reduction operations.

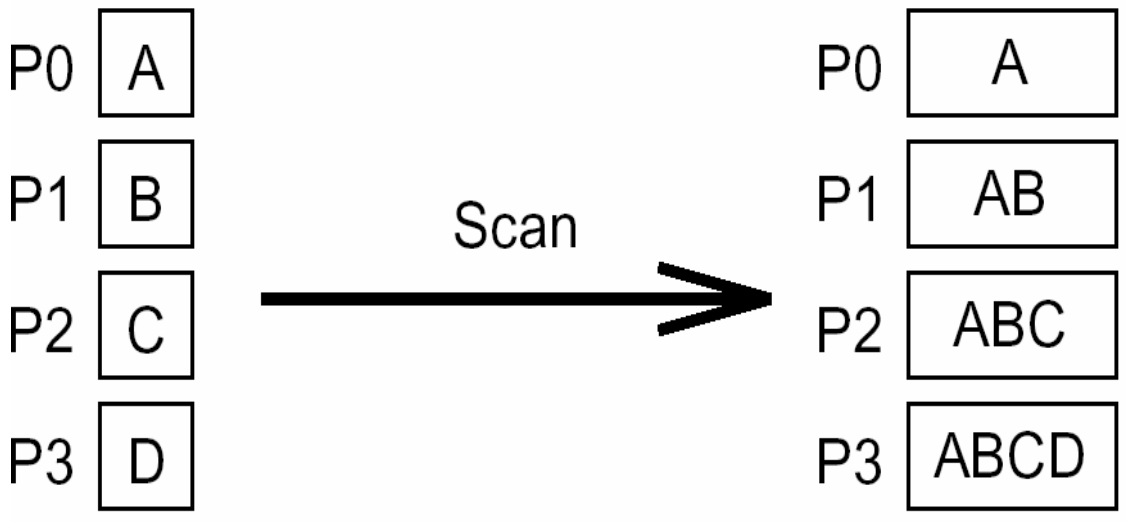
Operation	Meaning	Datatypes
MPI_MAX	Maximum	C integers and floating point
MPI_MIN	Minimum	C integers and floating point
MPI_SUM	Sum	C integers and floating point
MPI_PROD	Product	C integers and floating point
MPI_LAND	Logical AND	C integers
MPI_BAND	Bit-wise AND	C integers and byte
MPI_LOR	Logical OR	C integers
MPI_BOR	Bit-wise OR	C integers and byte
MPI_LXOR	Logical XOR	C integers
MPI_BXOR	Bit-wise XOR	C integers and byte
MPI_MAXLOC	max-min value-location	Data-pairs
MPI_MINLOC	min-min value-location	Data-pairs

MPI Functions: All-reduce

- Same as MPI_Reduce, but all processes receive the result of MPI_Op operation.

```
int MPI_Allreduce(void *sendbuf, void *recvbuf, int count,
        MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

MPI Functions: Prefix Scan




```
int MPI_Scan(void *sendbuf, void *recvbuf, int count,
        MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

MPI Names

MPI names of the various operations

Operation	MPI Name
One-to-all broadcast	MPI_Bcast
All-to-one reduction	MPI_Reduce
All-to-all broadcast	MPI_Allgather
All-to-all reduction	MPI_Reduce_scatter
All-reduce	MPI_Allreduce
Gather	MPI_Gather
Scatter	MPI_Scatter
All-to-all personalized	MPI_Alltoall

MPI Functions: Topology

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims,
        int *periods, int reorder, MPI_Comm *comm_cart)
```

```
int MPI_Cart_rank(MPI_Comm comm_cart, int *coords, int *rank)
int MPI_Cart_coord(MPI_Comm comm_cart, int rank, int maxdims,
        int *coords)
```

Performance Evaluation

- Elapsed (wall-clock) time

```
double t1, t2;
t1 = MPI_Wtime();
...
t2 = MPI_Wtime();
printf( "Elapsed time is %f\n", t2 - t1 );
```

Matrix/Vector Multiply

Program 6.4 Row-wise Matrix-Vector Multiplication

```
RowMatrixVectorMultiply(int n, double *a, double *b, double *x,
                        MPI_Comm comm)
{
  int i, j;
  int nlocal;          /* Number of locally stored rows of A */
  double *fb;          /* Will point to a buffer that stores the entire vector b */
  int npes, myrank;
  MPI_Status status;

  /* Get information about the communicator */
  MPI_Comm_size(comm, &npes);
  MPI_Comm_rank(comm, &myrank);

  /* Allocate the memory that will store the entire vector b */
  fb = (double *)malloc(n*sizeof(double));

  nlocal = n/npes;

  /* Gather the entire vector b on each processor using MPI's ALLGATHER operation */
  MPI_Allgather(b, nlocal, MPI_DOUBLE, fb, nlocal, MPI_DOUBLE,
      comm);

  /* Perform the matrix-vector multiplication involving the locally stored submatrix */
  for (i=0; i<nlocal; i++) {
    x[i] = 0.0;
    for (j=0; j<n; j++)
      x[i] += a[i*n+j]*fb[j];
  }

  free(fb);
}
```

3.3 OpenMPI Installation

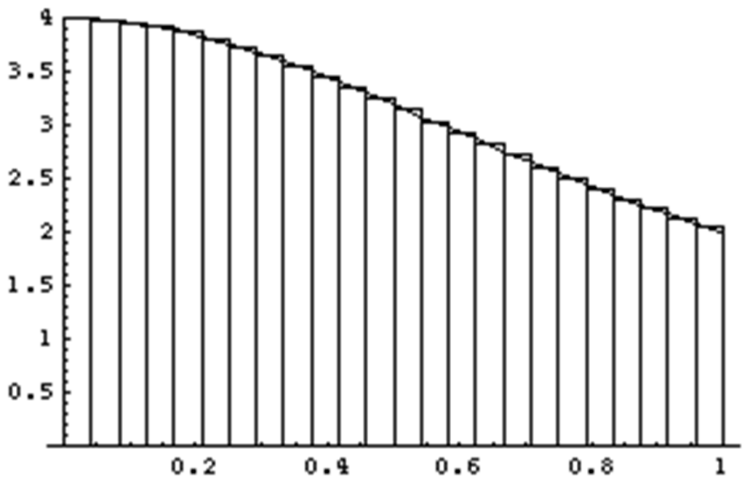
OpenMPI Installation - Cluster

- https://www.open-mpi.org
- Step 1 https://youtu.be/-t4k6IwmtFI
- Step 2 https://youtu.be/zXgwahyZxAw
- Step 3 https://youtu.be/WLVWNLZ2Lw8
- Step 4 https://youtu.be/HLTm5-bVt7c

3.4 Examples

Example: Compute PI (0)

$$\pi = \int_0^1 \frac{4}{1+x^2}\, dx$$



Example: Compute PI (1)

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

int main(int argc, char *argv[])
{
  int done = 0, n, myid, numprocs, I, rc;
  double PI25DT = 3.141592653589793238462643;
  double mypi, pi, h, sum, x, a;
  MPI_INIT(&argc, &argv);
  MPI_COMM_SIZE(MPI_COMM_WORLD, &numprocs);
  MPI_COMM_RANK(MPI_COMM_WORLD, &myid);
  while (!done)
  {
        if (myid == 0)
        {
        printf("Enter the number of intervals: (0 quits) ");
                scanf("%d", &n);
        }
        MPI_BCAST(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if (n == 0)
}
```

Example: Compute PI (2)

```
    h = 1.0 / (double)n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs)
    {
        x = h * ((double)i - 0.5);
        sum += 4.0 / (1.0 + x * x);
    }
    mypi = h * sum;
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);

    if (myid == 0) printf("pi is approximately %.16f, Error is
%.16f\n", pi, fabs(pi - PI25DT));

    MPI_Finalize();
    return 0;
}
```

Example 2: Compute Prime Number (0)

```c
# include <math.h>
# include <mpi.h>
# include <stdio.h>
# include <stdlib.h>
# include <time.h>

int main ( int argc, char *argv[] );
int prime_number ( int n, int id, int p );
void timestamp ( );

/****************************************************************************/

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

/****************************************************************************/
```

Example 2: Compute Prime Number (1)

```
{
  int I, id, ierr, n,n_factor,n_hi,n_lo,p,primes,primes_part;
  double wtime;
  n_lo = 1;
  n_hi = 1048576;
  n_factor = 2;

ierr = MPI_Init ( &argc, &argv );
ierr = MPI_Comm_size ( MPI_COMM_WORLD, &p );
ierr = MPI_Comm_rank ( MPI_COMM_WORLD, &id );
```

Example 2: Compute Prime Number (2)

```
if ( id == 0 )
 {
   timestamp ( );
   printf ( "\n" );
   printf ( "PRIME_MPI\n" );
   printf ( "  C/MPI version\n" );
   printf ( "\n" );
   printf ( "  An MPI example program to count the number of primes.\n"
);
   printf ( "  The number of processes is %d\n", p );
   printf ( "\n" );
   printf ( "         N        Pi          Time\n" );
   printf ( "\n" );
 }
```

Example 2: Compute Prime Number (3)

```
n = n_lo;

  while ( n <= n_hi )
  {
    if ( id == 0 )
    {
      wtime = MPI_Wtime ( );
    }
    ierr = MPI_Bcast ( &n, 1, MPI_INT, 0, MPI_COMM_WORLD );

    primes_part = prime_number ( n, id, p );

    ierr = MPI_Reduce ( &primes_part, &primes, 1, MPI_INT, MPI_SUM, 0,
      MPI_COMM_WORLD );

    if ( id == 0 )
    {
      wtime = MPI_Wtime ( ) - wtime;
      printf ( "  %8d  %8d  %14f\n", n, primes, wtime );
    }
    n = n * n_factor;
  }
```

Example 2: Compute Prime Number (4)

```
/*
  Terminate MPI.
*/
  ierr = MPI_Finalize ( );
/*
  Terminate.
*/
  if ( id == 0 )
  {
    printf ( "\n");
    printf ( "PRIME_MPI - Master process:\n");
    printf ( "  Normal end of execution.\n");
    printf ( "\n" );
    timestamp ( );
  }

  return 0;
}
```

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Thank you for your attentions !