MPI
(Message Passing Interface)

Mitsuhisa Sato
RIKEN R-CCS
and University of Tsukuba

(Original from Prof. Takahashi, University of Tsukuba and
Part of slides courtesy of Prof. Yuetsu Kodama, RIKEN)

How to make computer fast?

- Computer became faster and faster by
  - Device
  - Computer architecture
- Computer architecture to perform processing in parallel at several levels:
  - Inside of CPU (core)
  - Inside of Chip
  - Between chips (+GPU)
  - Between computer

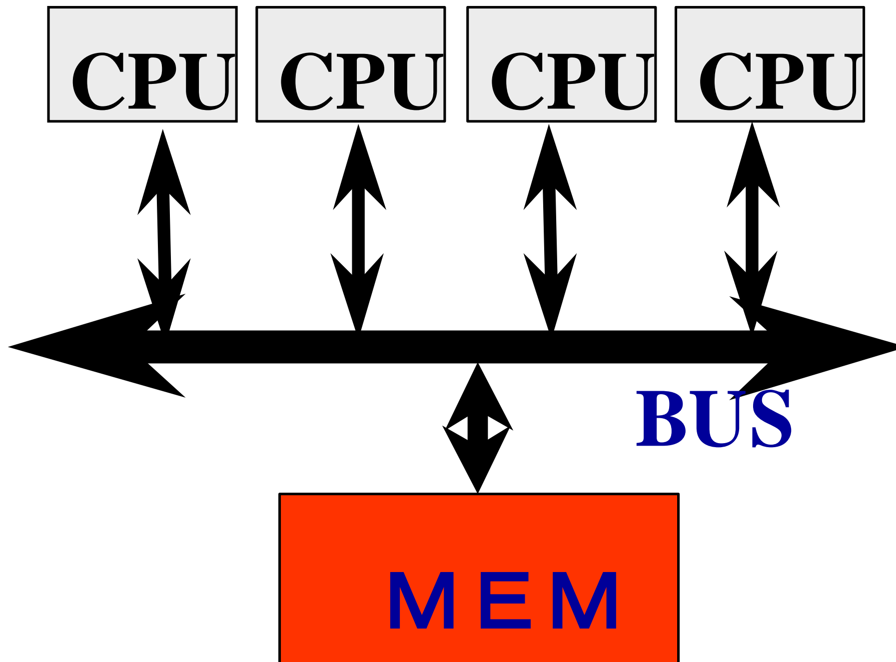
Pipeline
Superscalar

mulitcore

Shared memory multiprocessor

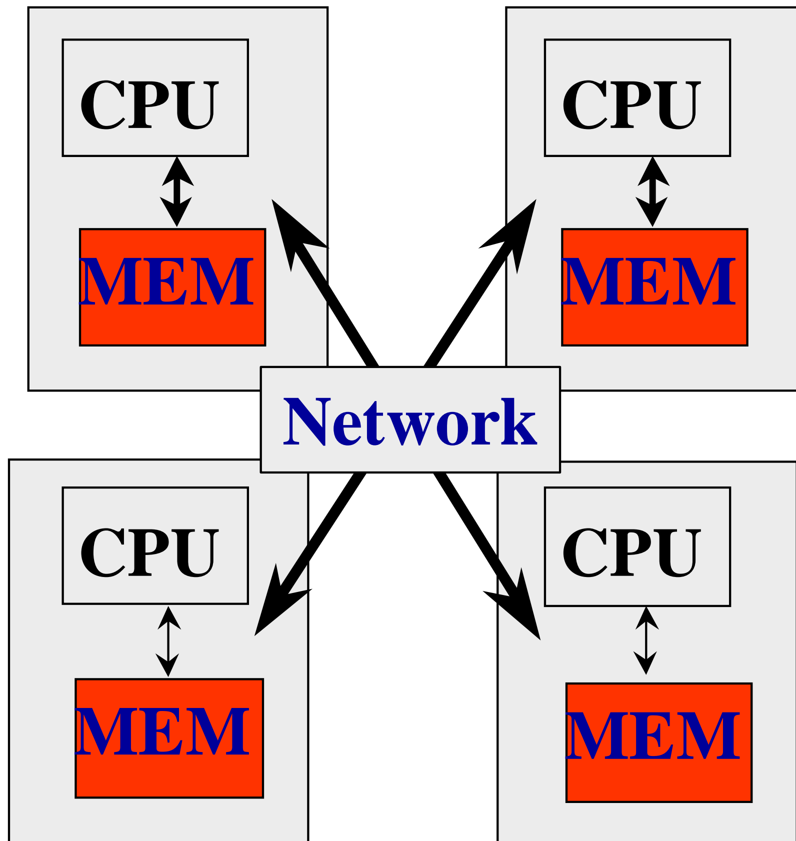
Distributed memory computer or Grid

Shared memory multi-processor system



- Multiple CPUs share main memory
- Threads executed in each core(CPU) communicate with each other by accessing shared data in main memory.
- Enterprise Server
- SMP Multi-core processors

Distributed memory multi-processor

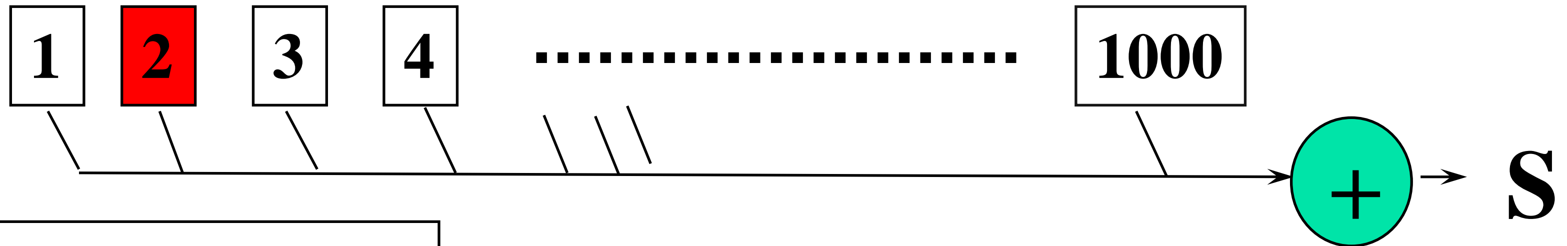


- **System with several computer of CPU and memory, connected by network.**
- **Thread executed in each computer communicate with each other by exchanging data (message) via network.タ**
- **PC Cluster**
- **AMP Multi-core processor**

Very simple example of parallel computing for high performance

```
for(i=0;i<1000; i++)
    S += A[i]
```

Sequential computation



Parallel computation

1 2 250

251 500

501 750

751 1000

Processor 1

Processor 2

プProcessor 3

Processor 4

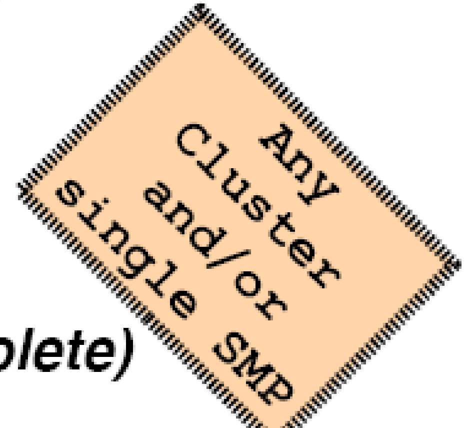
S

Parallel programming model

- Message passing programming model
  - Parallel programming by exchange data (message) between processors (nodes)
  - Mainly for distributed memory system (possible also for shared memory)
  - Program must control the data transfer explicitly.
  - Programming is sometimes difficult and time-consuming
  - Program may be scalable (when increasing number of Proc)
- Shared memory programming model
  - Parallel programming by accessing shared data in memory.
  - Mainly for shared memory system. (can be supported by software distributed shared memory)
  - System moves shared data between nodes (by sharing)
  - Easy to program, based on sequential version
  - Scalability is limited. Medium scale multiprocessors.

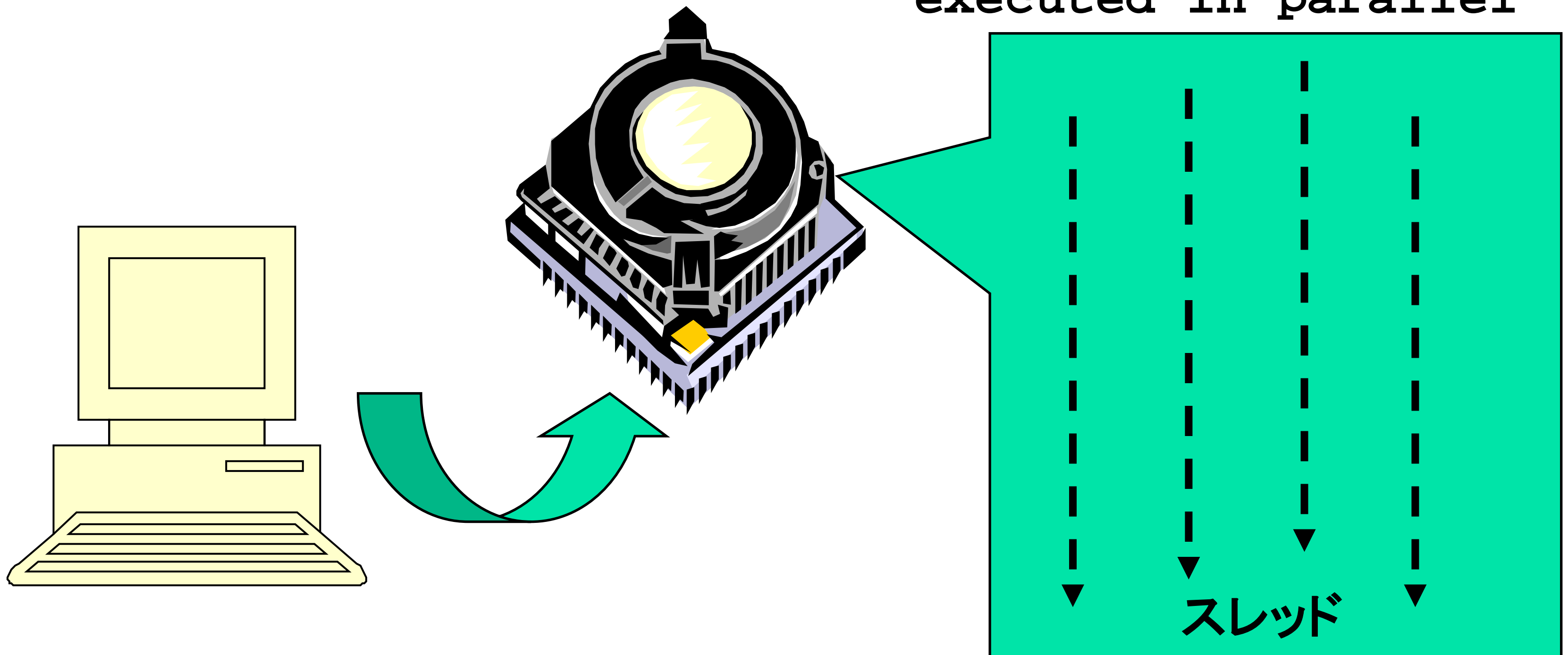
Parallel programming models

- *There are numerous parallel programming models*
- *The ones most well-known are:*
  - *Distributed Memory*
    - ✔ *Sockets (standardized, low level)*
    - ✔ *PVM - Parallel Virtual Machine (obsolete)*
    - ✔ *MPI - Message Passing Interface (de-facto std)*
  - *Shared Memory*
    - ✔ *Posix Threads (standardized, low level)*
    - ✔ *OpenMP (de-facto standard)*
    - ✔ *Automatic Parallelization (compiler does it for you)*



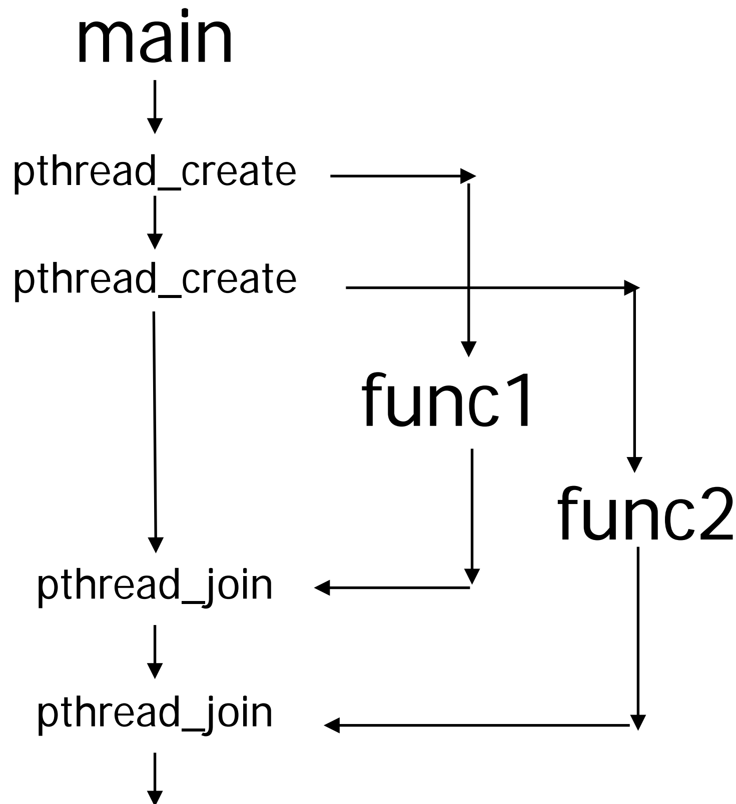
Multithread(ed) programming

- Basic model for shared memory
- Thread of execution = abstraction of execution in processors.
  - Different from process
    - Procss = thread + memory space
  - POSIX thread library = pthread



POSIX thread library

- Create thread: thread_create
- Join threads: pthread_join
- Synchronization, lock




```
#include <pthread.h>

void func1( int x );   void func2( int x );

main() {
        pthread_t t1 ;
        pthread_t t2 ;
          pthread_create( &t1, NULL,
                           (void *)func1, (void *)1 );
          pthread_create( &t2, NULL,
                           (void *)func2, (void *)2 );
          printf("main()¥n");
          pthread_join( t1, NULL );
          pthread_join( t2, NULL );
}
void func1( int x ) {
       int i ;
        for( i = 0 ; i<3 ; i++ ) {
             printf("func1( %d ): %d ¥n",x, i );
        }
}
void func2( int x ) {
             printf("func2( %d ): %d ¥n",x);
}
```

Programming using POSIX thread

- Create threads

Pthread, Solaris thread

```
for(t=1;t<n_thd;t++){
  r=pthread_create(thd_main,t)
}
thd_main(0);
for(t=1; t<n_thd;t++)
       pthread_join();
```

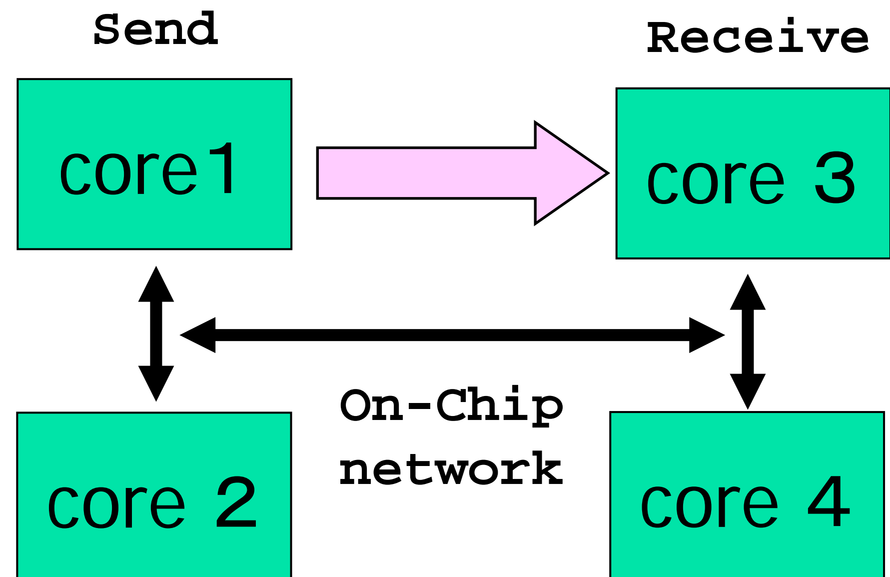
Thread =
Execution of program

- Divide and assign iterations of loop
- Synchronization for sum

```
int s;  /* global */
int n_thd; /* number of threads */
int thd_main(int id)
{ int c,b,e,i,ss;
  c=1000/n_thd;
  b=c*id;
  e=s+c;
  ss=0;
  for(i=b; i<e; i++) ss += a[i];
  pthread_lock();
  s += ss;
  pthread_unlock();
  return s;
}
```

Message passing programming

- General programming paradigm for distributed memory system.
  - Data exchange by "send" and "receive"
- Communication library, layer
  - POSIX IPC, socket
  - TIPC (Transparent Interprocess Communication)
  - LINX (on Enea's OSE Operating System)
  - MCAPI (Multicore Communication API)
  - MPI (Message Passing Interface)



Simple example of Message Passing Programming

- Sum up 1000 element in array

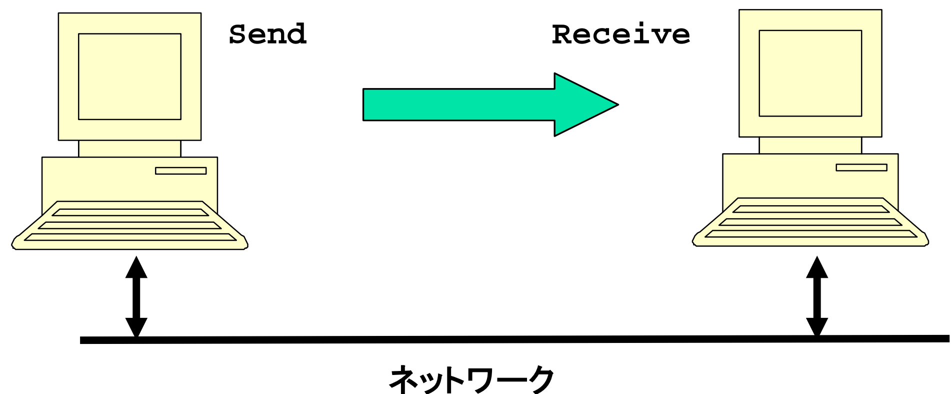
```
int a[250]; /* 250 elements are allocated in each node */

main(){     /* start main in each node */
    int i,s,ss;
    s=0;
    for(i=0; i<250;i++) s+= a[i]; /*compute local sum*/
    if(myid == 0){   /* if processor 0 */
        for(proc=1;proc<4; proc++){
            recv(&ss,proc);  /* receive data from others*/
            s+=ss;   /*add local sum to sum*/
        }
    } else {  /* if processor 1,2,3 */
        send(s,0);  /* send local sum to processor 0 */
    }
}
```

Parallel programming using MPI

- MPI (Message Passing Interface)
- Mainly, for High performance scientific computing
- Standard library for message passing parallel programming in high-end distributed memory systems.
  - Required in case of system with more than 100 nodes.
  - Not easy and time-consuming work
    - "assembly programming" in distributed programming
- Communication with message
  - Send/Receive
- Collective operations
  - Reduce/Bcast
  - Gather/Scatter

Over-specs for Embedded system Programming?!



Programming in MPI

```
#include "mpi.h"
#include <stdio.h>
#define MY_TAG 100
double A[1000/N_PE];
int main( int argc, char *argv[])
{
    int n, myid, numprocs, i;
    double sum, x;
    int  namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    MPI_Status status;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Get_processor_name(processor_name,&namelen);
    fprintf(stderr,"Process %d on %s¥n", myid, processor_name);

    ....
```

Programming in MPI

```
    sum = 0.0;
    for (i = 0; i < 1000/N_PE; i++){
       sum+ = A[i];
    }

    if(myid == 0){
       for(i = 1; i < numprocs; i++){
          MPI_Recv(&t,1,MPI_DOUBLE,i,MY_TAG,MPI_COMM_WORLD,&status
           sum += t;
        }
    } else
           MPI_Send(&t,1,MPI_DOUBLE,0,MY_TAG,MPI_COMM_WORLD);
    /* MPI_Reduce(&sum, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM
    MPI_Barrier(MPI_COMM_WORLD);
    ...
    MPI_Finalize();
    return 0;
}
```

MPI parallel programming

- MPI（Message Passing Interface）is a parallel programming model for distributed memory systems.
- MPI is not a new programming language, but a library for message-passing called from C or Fortran.
- MPI is proposed as a standard by a broadly based committee of vendors, implementers, and users.
- MPI2.1 is released in 2008, and MPI3.0 is released in 2012, with additional features such as one-sided communication, etc, but in this lecture features in MPI1.0 are introduced.

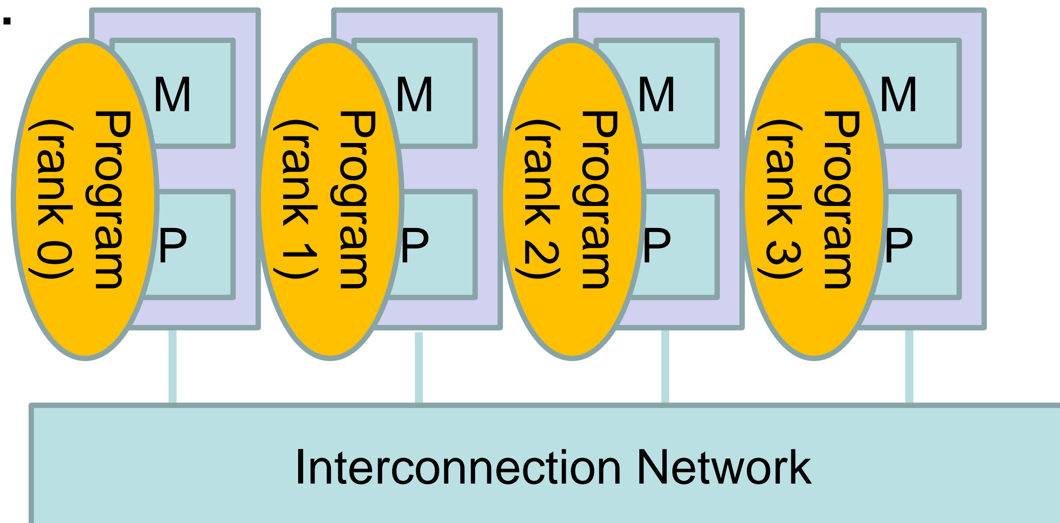
http://www.mpi-forum.org/

Parallel programming model

- Parallel programming model is categorized to the following two models.
  - SPMD（Single Program Multiple Data）
  - MPMD（Multiple Program Multiple Data）
- In SPMD model, same programs are executed in each node. (Ex. MPI)
- In MPMD model, different programs are executed in each node (Ex. master/worker pattern)

Execution model of MPI

- Same programs(processes) run on multiple processors
  - A process does not synchronize to other processes without communication.
- Each process has own ID (rank).
- Process communicates to other processes using MPI functions.



Structure of MPI program

```
#include "mpi.h"
#include <stdio.h>
#define N 1000

int main( int argc, char *argv[])
{
   int myid, nprocs, sendbuf[N], recvbuf[N];
   MPI_Status status;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
   MPI_Comm_rank(MPI_COMM_WORLD, &myid);
…
   MPI_Send(sendbuf, N, MPI_INTEGER, (myid + 1) % nprocs, MPI_COMM_WORLD);
   MPI_Recv(recvbuf, N, MPI_INTEGER, (myid + 1) % nprocs, 0,
                  MPI_COMM_WORLD, &status);
…
   MPI_Finalize();
   return 0;
}
```

Steps of MPI programming

(1) Include a header file: #include “mpi.h”

(2) Call MPI_Init() to initialize the MPI runtime environment

(3) Call MPI_Comm_size() to get the number of processes

(4) Call MPI_Comm_rank() to get the self process ID

(5) Call MPI_Send() and MPI_Recv() to communicate with other processes.

(6) Call MPI_Finalize() to complete the MPI runtime environment

MPI functions

- There are more than one hundred of functions in MPI, and classified to followings:
  - Point-to-point communication
  - Collective communication
  - Groups, Contexts, Communicators
  - Process Topologies
  - Derived datatypes and MPI_Pack/Unpack
  - MPI Environmental Management
- You can write a MPI program with about 20 MPI functions in usual.
  - Frequently used functions are about 10.

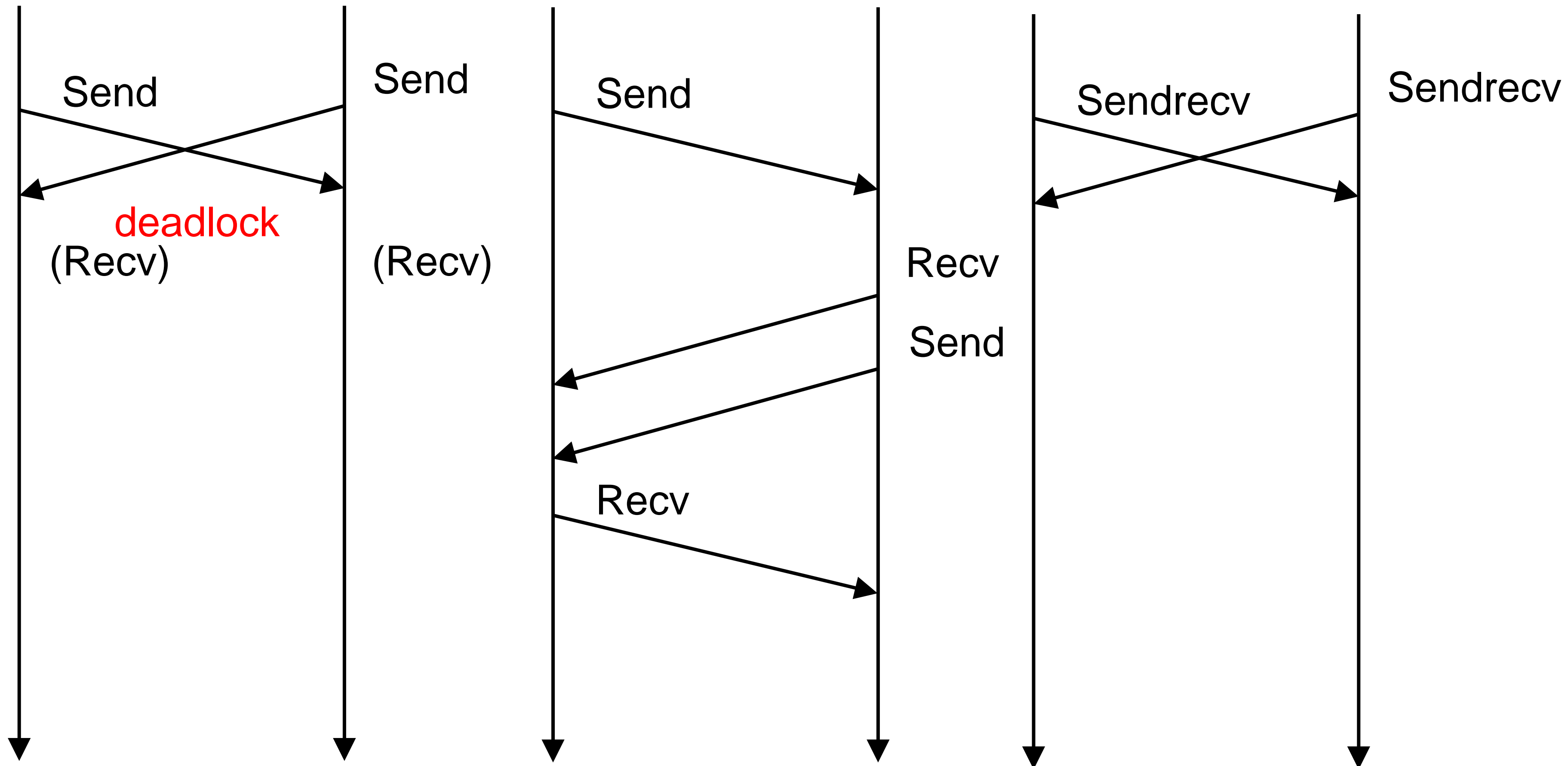
Communicator

- A communicator specifies the process group that can send and receive messages to each other.
- A predefined communicator MPI_COMM_WORLD is provided by MPI. It allows communication with all processes that are accessible after MPI initialization and processes are identified by their rank in it. Usually using only MPI_COMM_WORLD is enough.
- Users may define new communicators if necessary.

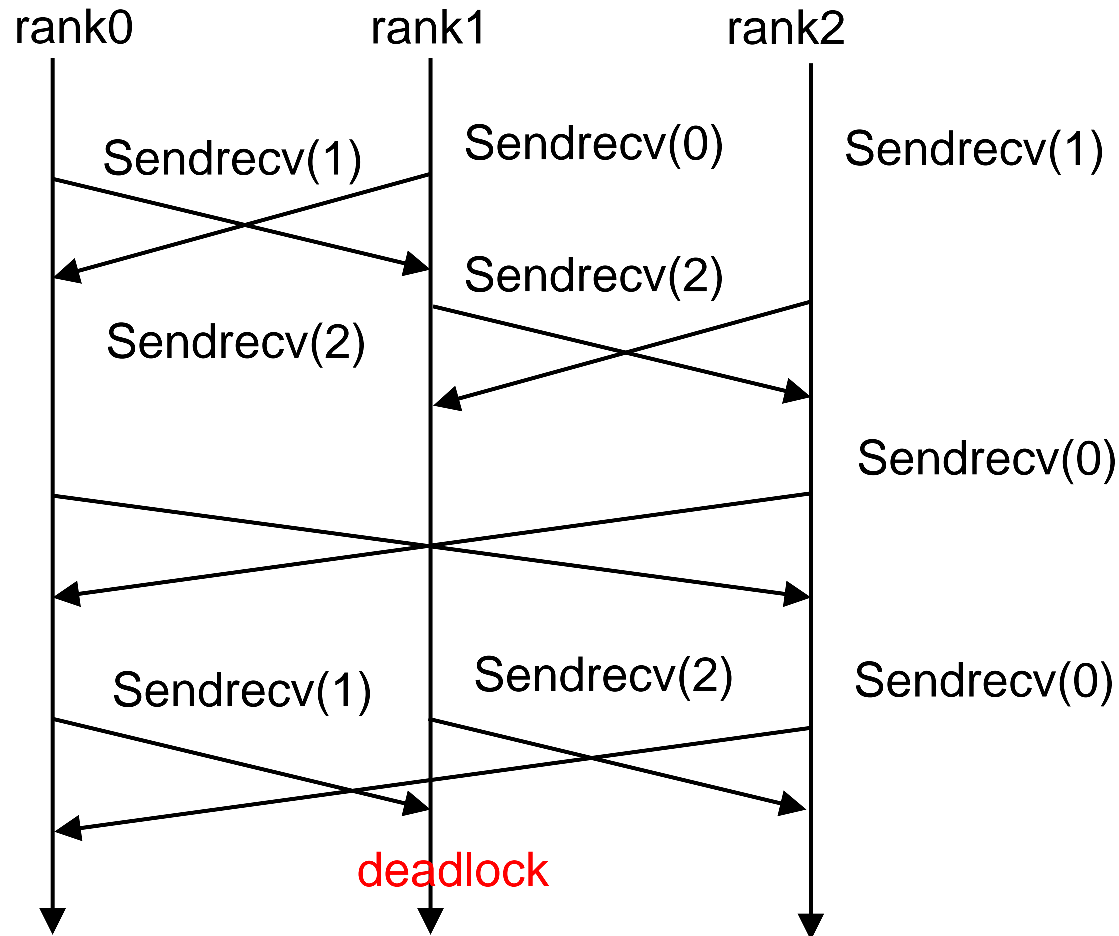
Point-to-Point Communication

- Examples of point-to-point Communication
  - Blocking Communication（MPI_Send，MPI_Recv）
    - MPI_Send may block until the message is received by the destination process.
    - MPI_send/recv specifies the buffer area for communication, and after MPI_send/recv returns, the buffer can be modified..
  - Nonblocking Communication（MPI_Isend，MPI_Irecv, MPI_Wait）
    - they can improve performance by overlapping communication and computation.
  - Bi-directional Communication（MPI_Sendrecv）
    - It prevent cyclic dependencies that may lead to deadlock.

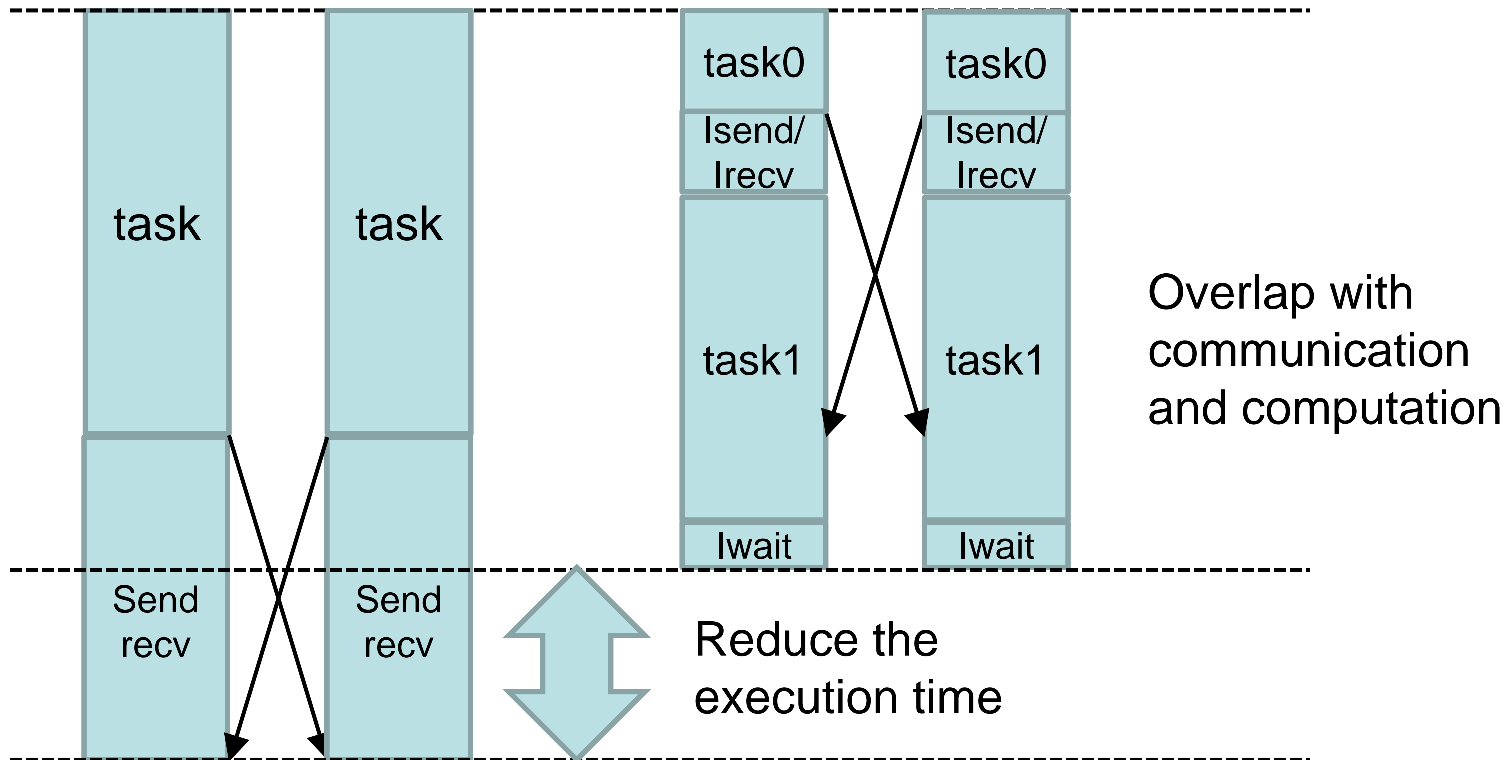
Example of send/recv



Example of sendrecv



Example of Isend/Irecv



P2P Comm. functions

- int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
- int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
  - blocking send/receive operation
    - buf: initial address of send buffer
    - count: number of elements in send buffer
    - datatype: datatype of each send buffer element
    - dest: rank of destination
    - source: rank of source
    - tag: message tag
    - comm: communicator
    - status: status object (structure MPI_Status)

Predefined MPI datatypes

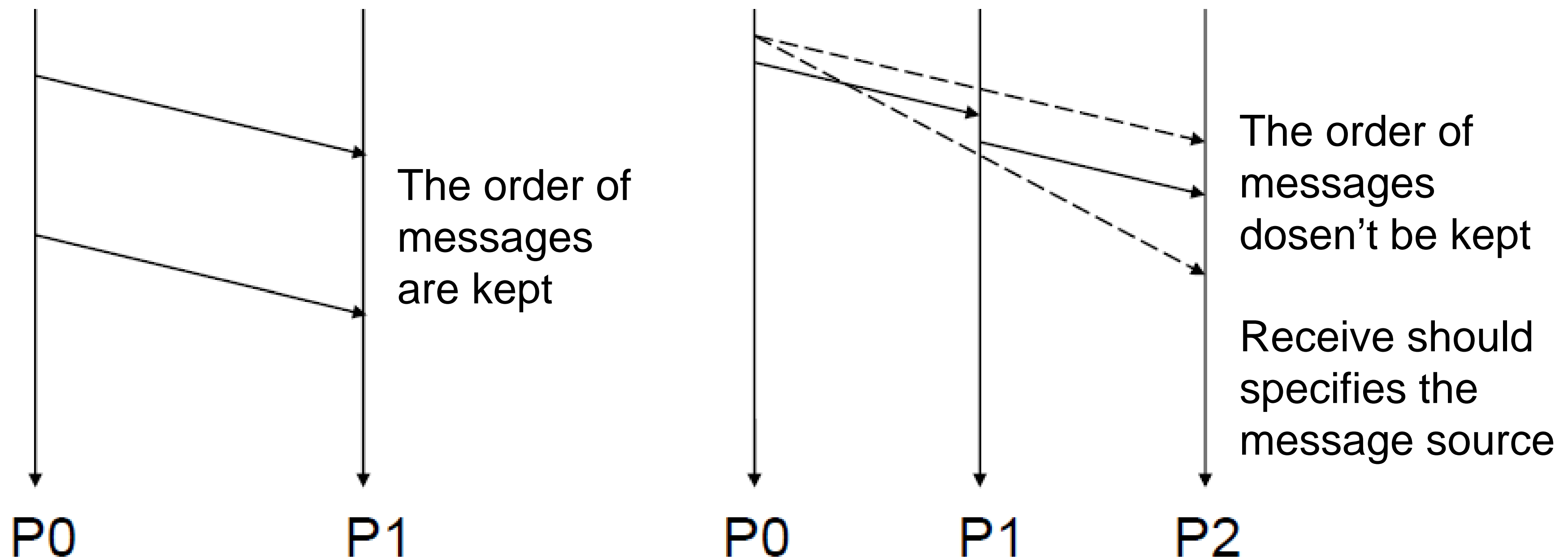
MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double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

Message tag

- Integer to distinguish different types of messages
  - User can define message tag freely for each message type in program.
  - A message can be received if it matches the source, tag and comm values specified by the receive operation.
- The receiver may specify a wildcard MPI_ANY_SOURCE and/or MPI_ANY_TAG indicating that any source and/or tag are acceptable.

The order of messages

- Between two nodes, the order of messages are kept.
- Among more than three, the order of messages may be changed.



Non blocking send/receive

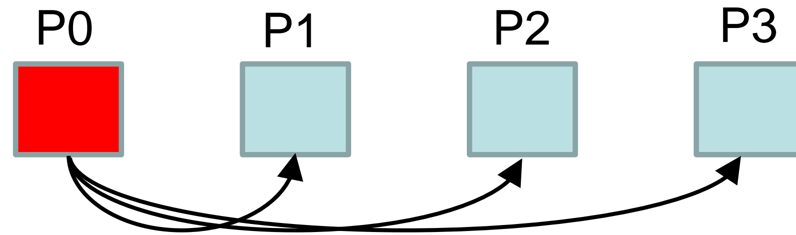
- MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_request *request)
- MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_request *request)
- MPI_Wait(MPI_request *request, MPI_status *status)
- MPI_Test(MPI_request *request, int *flag, MPI_status *status)

Collective Communication

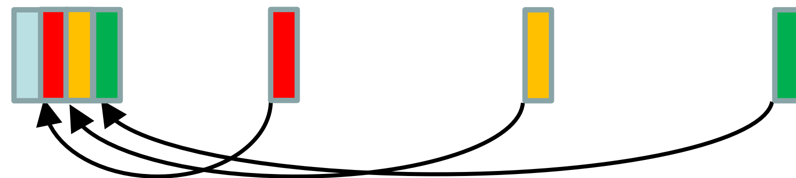
- Collective communication is defined as communication that involves a group of processes.
- Usually includes more than two processes.
- Examples of collective communication
  - Broadcast（MPI_Bcast）
  - Gather（MPI_Gather, MPI_Allgather）
  - Scatter（MPI_Scatter）
  - All-to-all（MPI_Alltoall）
  - Reduction（MPI_Reduce, MPI_Allreduce）

Collective communication

- Broadcast

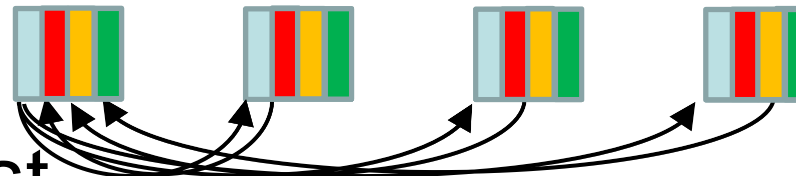


- Gather

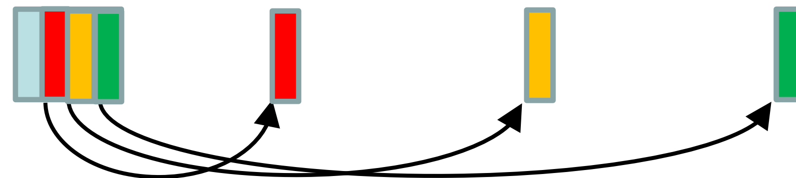


- Allgather

= Gather + Broadcast

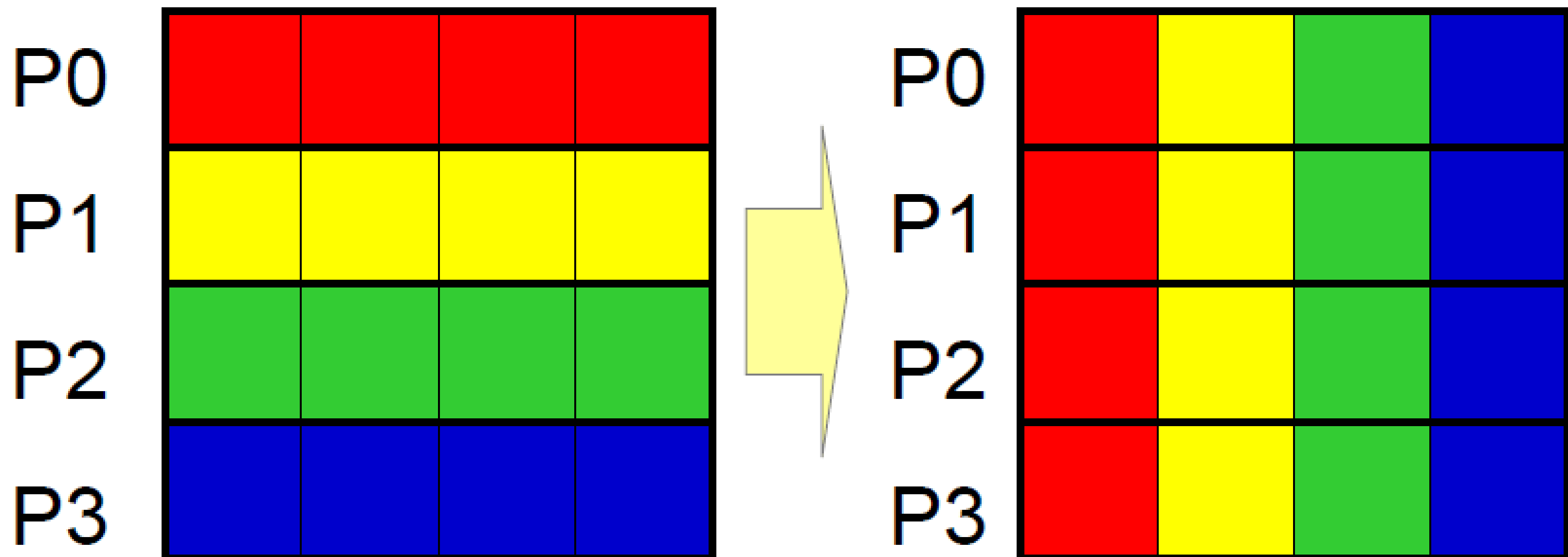


- Scatter



Alltoall

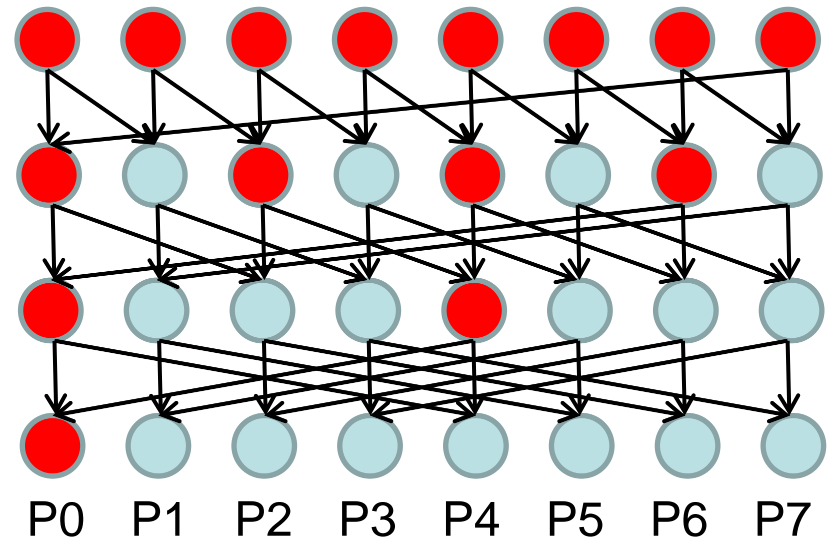
- Transpose array of distributed by row



Reduction

- int MPI_Reduce(void *sndbuf, void *rcvbuf, int count, MPI_Datatype datatype, MPI_Op operator, int root, MPI_Comm comm)
- int MPI_Allreduce(void *sndbuf, void *rcvbuf, int count, MPI_Datatype datatype, MPI_Op operator, MPI_Comm comm)

Operator	meaning
MPI_SUM	sum
MPI_PROD	product
MPI_MAX	maximum
MPI_MIN	minimum
MPI_LAND	logical and
MPI_BAND	bit-wise and



Communicator Management

- int MPI_Comm_size(MPI_Comm comm, int *size)
  - It indicates the number of processes involved in a communicator.
    - comm: communicator
    - size: number of processes in the group of comm
- int MPI_Comm_rank(MPI_Comm comm, int *rank)
  - It gives the rank of the process in the particular communicator's group.
    - comm: communicator
    - rank: rank of the calling process in group of comm

Both functions are local operations.

MPI Environmental Management

- int MPI_Init(int *argc, char **argv)
  - Initialize the MPI environment.
    - argc: number of arguments of command line
    - argv: arguments of command line
- int MPI_Finalize(void)
  - clean up all MPI state.
- double MPI_Wtime(void)
  - returns a floating-point number of seconds, representing elapsed wallclock time.

Example: Calculating the value of π

```
#include “mpi.h”
#include <stdio.h>
#include <math.h>
double f( double a ) { return (4.0 / (1.0 + a*a));}

int main( int argc, char *argv[])
{
        int done = 0, n, myid, numprocs, i;
        double PI25DT = 3.141592653589793238462643;
        double mypi, pi, h, sum, x;
        double startwtime, endwtime;
        int namelen;
        char processor_name[MPI_MAX_PROCESSOR_NAME];

        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
        MPI_Comm_rank(MPI_COMM_WORLD, &myid);
        MPI_Get_processor_name(processor_name, &namelen);

        startwtime = MPI_Wtime();
        if (myid == 0) n=atoi(argv[1]);
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        h = 1.0 / (double) n;
        sum = 0.0;
        for (i = myid + 1; i <=n; i += numprocs) {
            x = h * ((double) i - 0.5);
            sum += f(x);
        }
        mypi = h * sum;
        MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
        endwtime = MPI_Wtime();
        if (myid == 0) {
                printf(“pi :%.16f, Error : %.16f  n: %d, procs: %d, elaps:%.3f¥n”,
                           pi, fabs(pi  - PI25DT), n, numprocs, endwtime - startwtime);
        }
        MPI_Finalize();
        return 0;
}
```

Main part of calculating pi

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

Example of execution

```
$ mpicc pi.c -O -o pi
$ salloc -N 1 -p HPT -- mpirun --np 1 /home/COMP/daisuke/ahpc/mpi/pi 100000000
pi: 3.1415926535904264, Error: 0.0000000000006333 n: 100000000, procs: 1, elaps:2.004
$ salloc -N 1 -p HPT -- mpirun --np 2 /home/COMP/daisuke/ahpc/mpi/pi 100000000
pi: 3.1415926535900223, Error: 0.0000000000002292 n: 100000000, procs: 2, elaps:1.004
$ salloc -N 1 -p HPT -- mpirun --np 4 /home/COMP/daisuke/ahpc/mpi/pi 100000000
pi: 3.1415926535902168, Error: 0.0000000000004237 n: 100000000, procs: 4, elaps:0.503
$ salloc -N 1 -p HPT -- mpirun --np 8 /home/COMP/daisuke/ahpc/mpi/pi 100000000
pi: 3.1415926535896137, Error: 0.0000000000001794 n: 100000000, procs: 8, elaps:0.255
$ salloc -N 1 -p HPT -- mpirun /home/COMP/daisuke/ahpc/mpi/pi 100000000
pi: 3.1415926535897389, Error: 0.0000000000000542 n: 100000000, procs: 12, elaps:0.200
$ salloc -N 1 -p HPT -- mpirun /home/COMP/daisuke/ahpc/mpi/pi 1000000000
pi: 3.1415926535898397, Error: 0.0000000000000466 n: 1000000000, procs: 12, elaps:1.673
$ salloc -N 2 -p HPT -- mpirun /home/COMP/daisuke/ahpc/mpi/pi 1000000000
pi: 3.1415926535898517, Error: 0.0000000000000586 n: 1000000000, procs: 24, elaps:0.894
$ salloc -N 2 -n 12 -p HPT -- mpirun /home/COMP/daisuke/ahpc/mpi/pi 1000000000
pi: 3.1415926535898397, Error: 0.0000000000000466 n: 1000000000, procs: 12, elaps:1.721
```

Summary of MPI

- MPI is a parallel programming tool for distributed memory system.
- MPI is a library for message-passing.
  - Point to point communication
    - blocking: MPI_Send()/Recv()
    - Nonbloking: MPI_Isend()/Irecv()/Wait()
  - Colective communication
    - MPI_Bcast()/Gather()/Scatter()/AlltoAll/Reduce()
- MPI execution environment depends on the system that you use, ask to the system administrator.