MPI for Beginners

HONGSUK YI
(hsyi@kisti.re.kr)

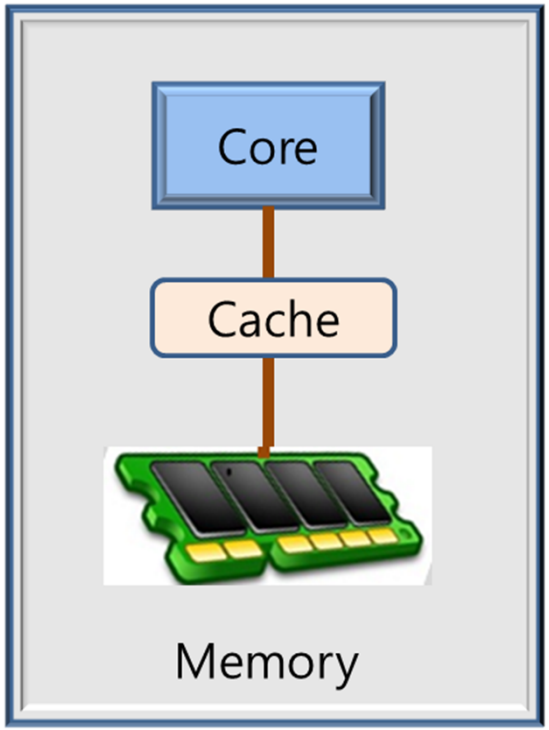
KISTI

Agenda

- What is MPI?
  - How to write a parallel program in MPI
  - Basic MPI
  - Point-to-point communication
- Slightly more intermediated topics:
  - Non-blocking communication in MPI
  - Collective communication in MPI
  - Lab2
- Summary and final Q&A

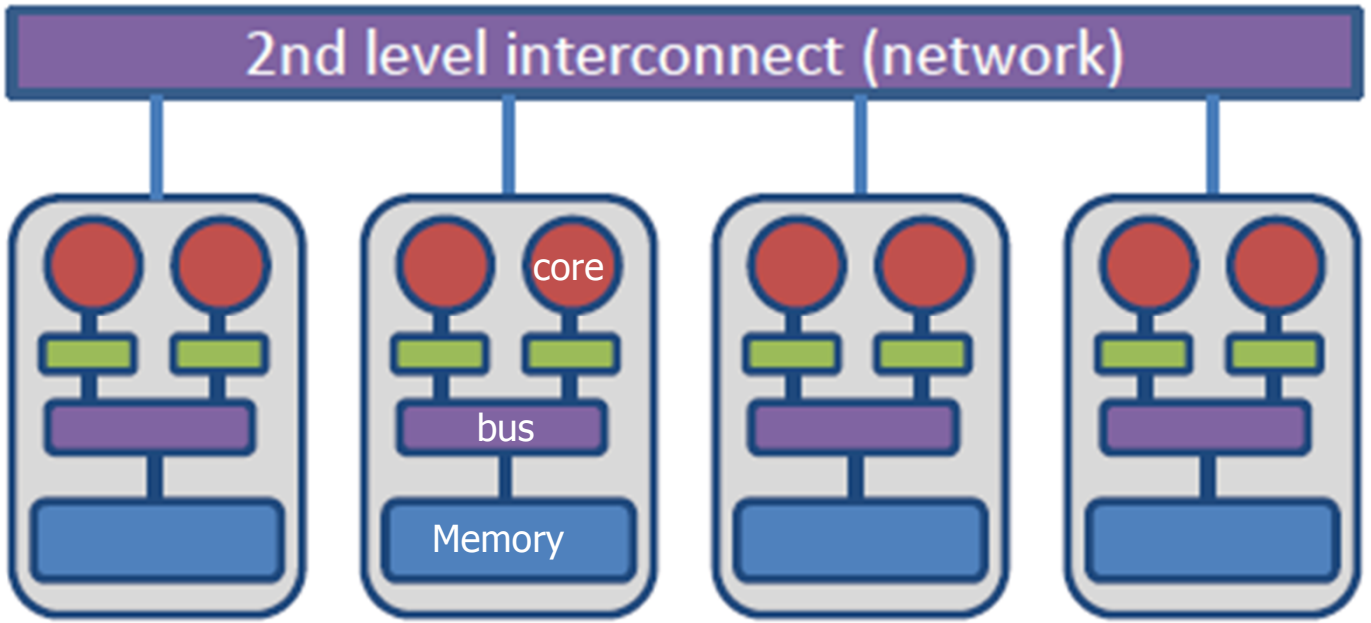
Single Processor System

- Processor
  - Core, Thread, CPU, task
  - Fetch program from memory
  - Load data from memory
  - Write results back to memory
  - Execute program instructions
  - Process data
- Main Memory
  - Store program and data



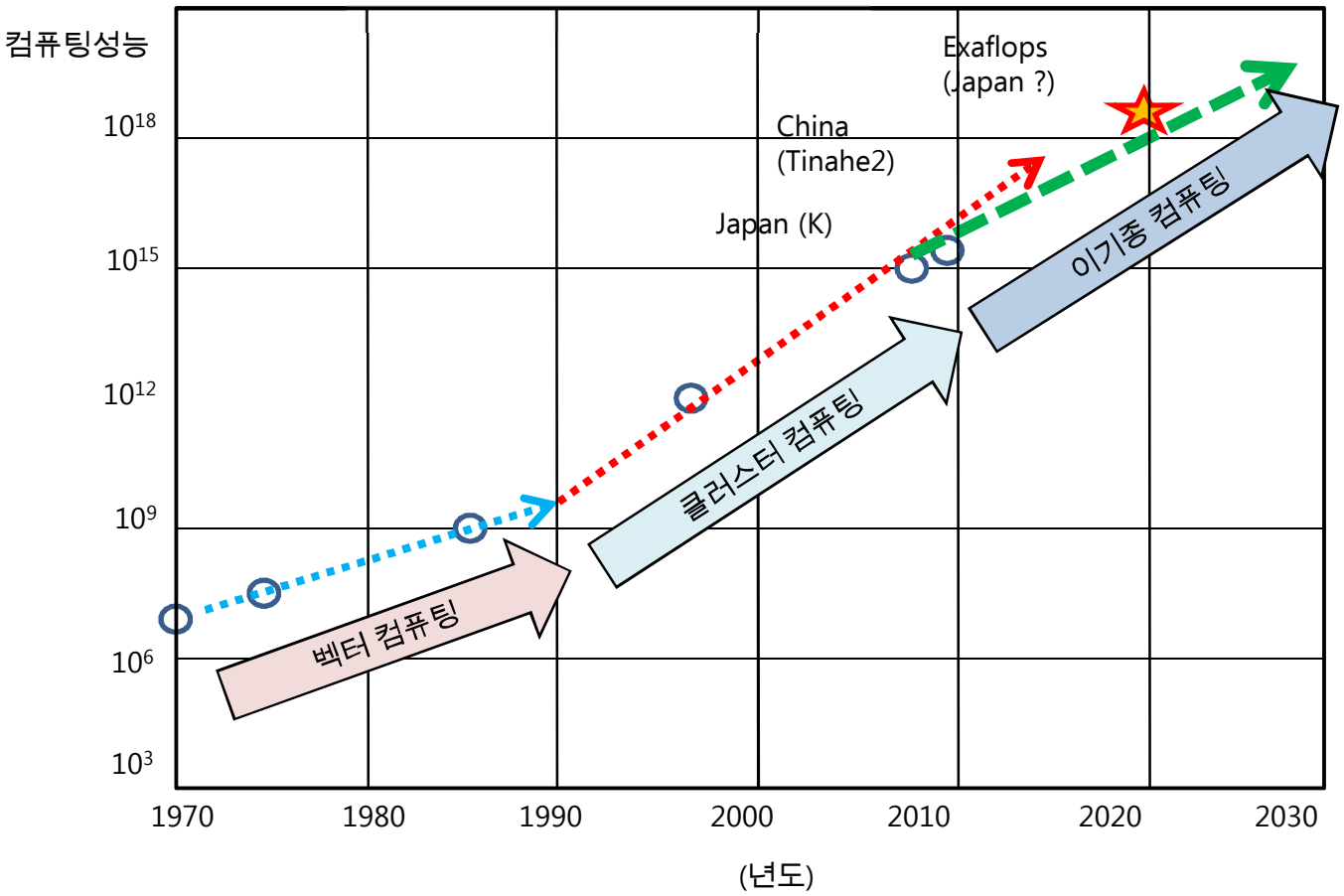
Distributed-Memory Parallelism

- Second level interconnect is not cache coherent
  - Typically used in HPC: InfiniBand (BW ~ 1.2 GB/s)

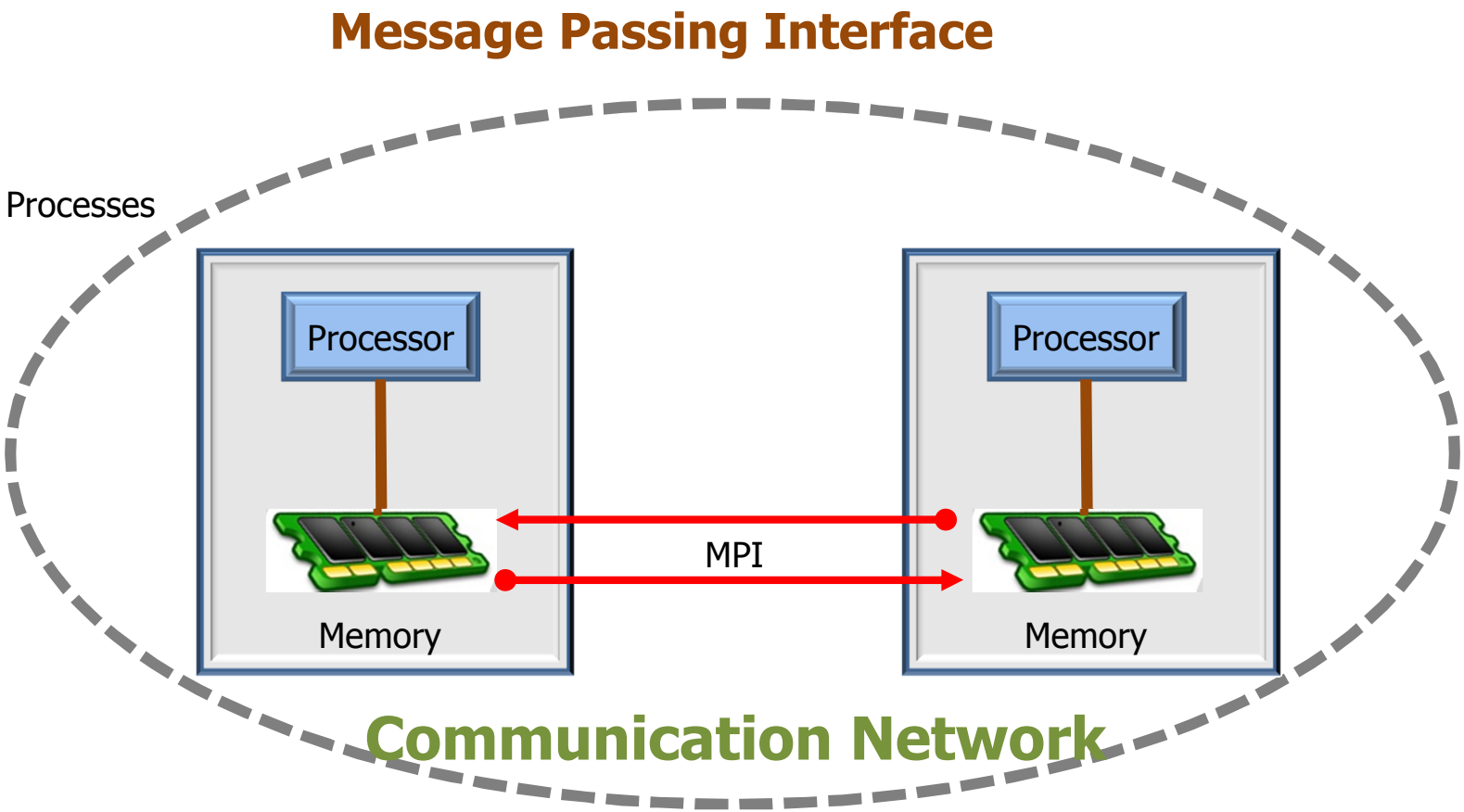


Distributed Memory System

MPI + ?



Parallel Programming Paradigm

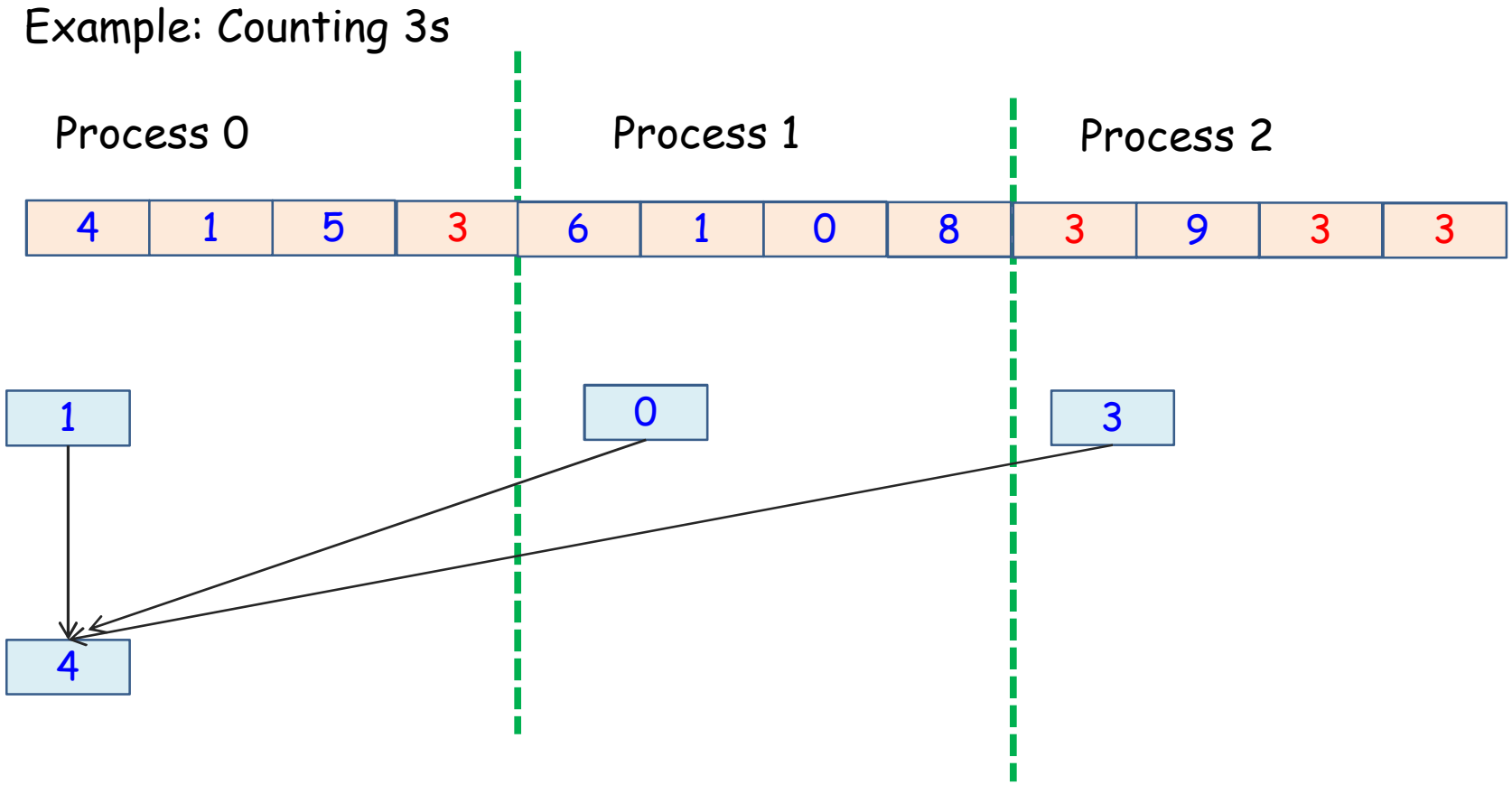


A Message-Passing Program

if (rank== 0) then k = k + 1182

Process 0	Process 1
k: 18 pid: 0 k: 1200	k: 19 pid: 1 k: 19

The Message-Passing Model (Example)



What is MPI?

- MPI: Message Passing Interface
  - Maintained by the MPI Forum since 1992
    - http://www.mpi-forum.org/
  - Incorporates the best ideas in a "standard" way
    - Many concrete implementations of the MPI standard
- MPI is not
  - A language or compiler specification
  - A specific implementation or product

MPI History

- Version 1.0 (1994): FORTRAN 77 and C bindings
- Version 1.1 (1995): Minor corrections and clarifications
- Version 1.2 (1997): Further corrections and clarifications
- Version 2.0 (1997): Major enhancements
  - One-sided communication
  - Parallel I/O
  - Dynamic process creation
  - Fortran 90 and C++ bindings
  - Thread safety
  - Language interoperability
- Version 2.1 (2008): Merger of MPI-1 and MPI-2
- Version 2.2 (2009): Minor corrections and clarifications
  - C++ bindings deprecated
- Version 3.0 (2012): Major enhancements
  - Non-blocking collective operations; C++ deleted from the standard

What is in MPI-1

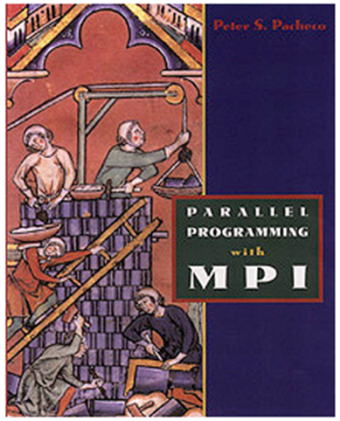
- Basic functions for communication (100+ functions)
  - Blocking sends, receives
  - Nonblocking sends and receives
    - Variants of above
  - Rich set of collective communication functions
    - Broadcast, scatter, gather, etc
    - Very important for performance; widely used
  - Datatypes to describe data layout
  - Process topologies
  - C, C++ and Fortran bindings
  - Error codes and classes

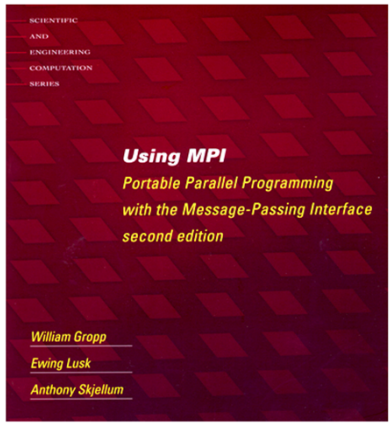
More information and documentation

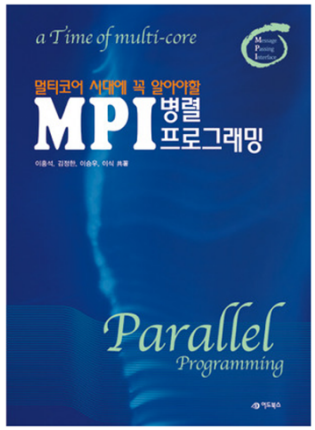
- The MPI Forum document archive (free standards)
  - http://www.mpi-forum.org/docs/
  - All MPI official releases, in both PDF and HTML
- Other information on Web : tutorials, a FAQ, MPI pages
  - http://www-unix.mcs.anl.gov/mpi/
  - http://www.mcs.anl.gov/research/projects/mpi/www/
  - http://www.open-mpi.org/
  - RWTH AACHEN University
    - http://www.rz.rwth-aachen.de/mpi/

MPI Reference

- MPI: The Complete Reference Vol. 1 The MPI Core (1998)
- MPI: The Complete Reference Vol. 2 The MPI Extensions (1998)
- Using MPI (1999)
- Using MPI-2 (2000)
- Parallel Programming with MPI (1996)







Agenda

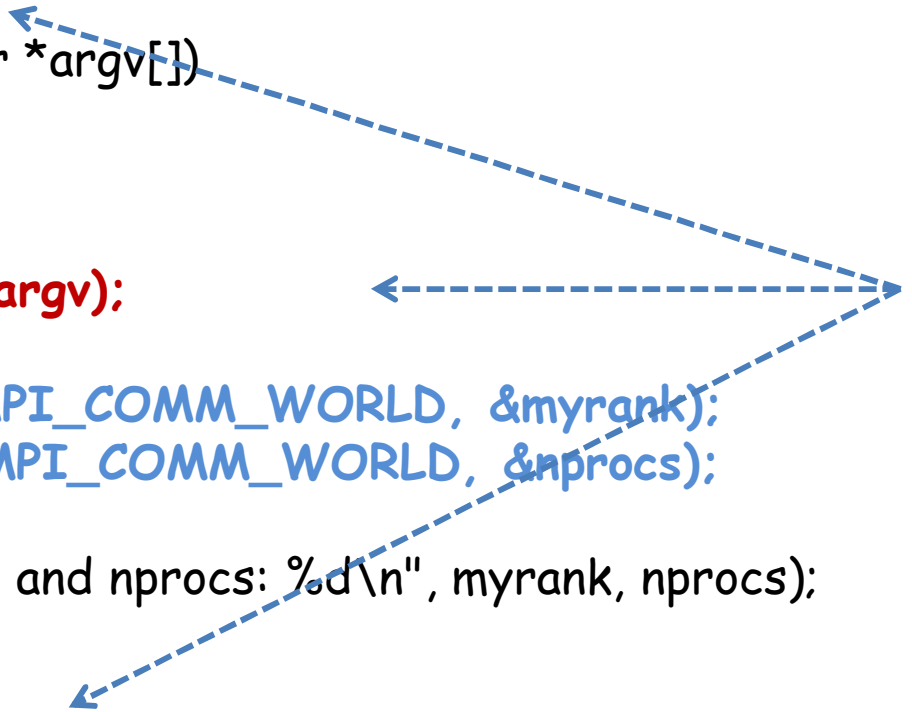
- What is MPI?
  - How to write a parallel program in MPI
  - Basic MPI
  - Point-to-point communication
- Slightly more intermediated topics:
  - Non-blocking communication in MPI
  - Collective communication in MPI
  - Lab2
- Summary and final Q&A

Six Functions for Simplified MPI

- **Six functions**
  - **MPI_INIT**
  - **MPI_COMM_SIZE**
  - **MPI_COMM_RANK**
  - **MPI_SEND**
  - **MPI_RECV**
  - **MPI_FINALIZE**

Simple MPI program : C

```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char *argv[])
{
   int myrank, nprocs;

   MPI_Init(&argc, &argv);

   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

   printf(" myrank: %d and nprocs: %d\n", myrank, nprocs);

   MPI_Finalize();

   return 0;
}
```



Basic requirements for an MPI program

```
$mpicc -o a.out myrank.c
$mpirun -np 4 ./a.out
```

Simple MPI program : Fortran

```
PROGRAM hello
  IMPLICIT NONE
  INCLUDE 'mpif.h'

  INTEGER myrank, nprocs, ierr

  CALL MPI_Init(ierr)
  CALL MPI_Comm_size(MPI_COMM_WORLD, nprocs, ierr)
  CALL MPI_Comm_rank(MPI_COMM_WORLD, myrank, ierr)

  WRITE (*, *) 'myrank:',myrank, 'and nprocs=', nprocs

  CALL MPI_FINALIZE(ierr)
END PROGRAM hello
```

```
mpif90 -o a.out myrank.f90
mpirun -np 4 ./a.out
```

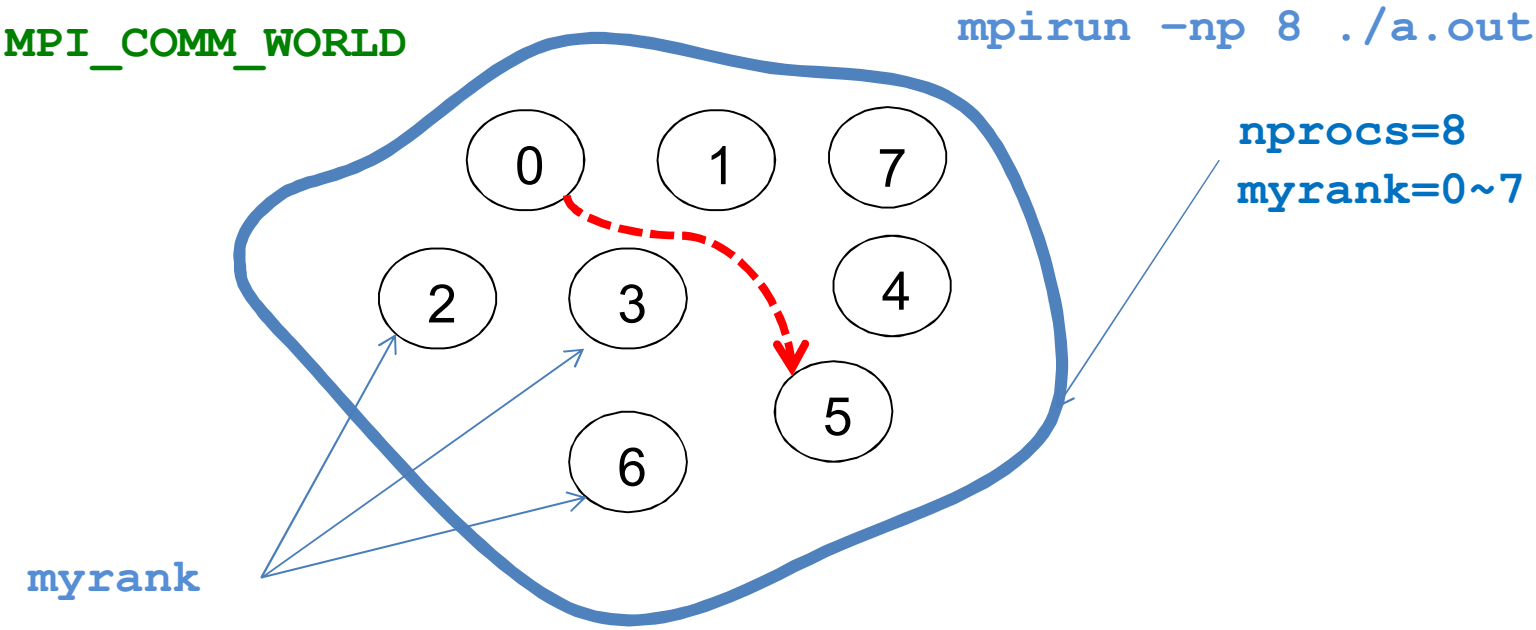
```
myrank: 1 and nprocs=  4
myrank: 2 and nprocs=  4
myrank: 3 and nprocs=  4
myrank: 0 and nprocs=  4
```

Compiling and Running MPI codes

- MPI is a library
  - Applications can be written in C, C++ or Fortran
- Compilation:
  - mpicc test.c -o a.out
- Execution:
  - mpirun –np 4 ./a.out
  - mpiexec –np 4 ./a.out

MPI Communicator and Rank

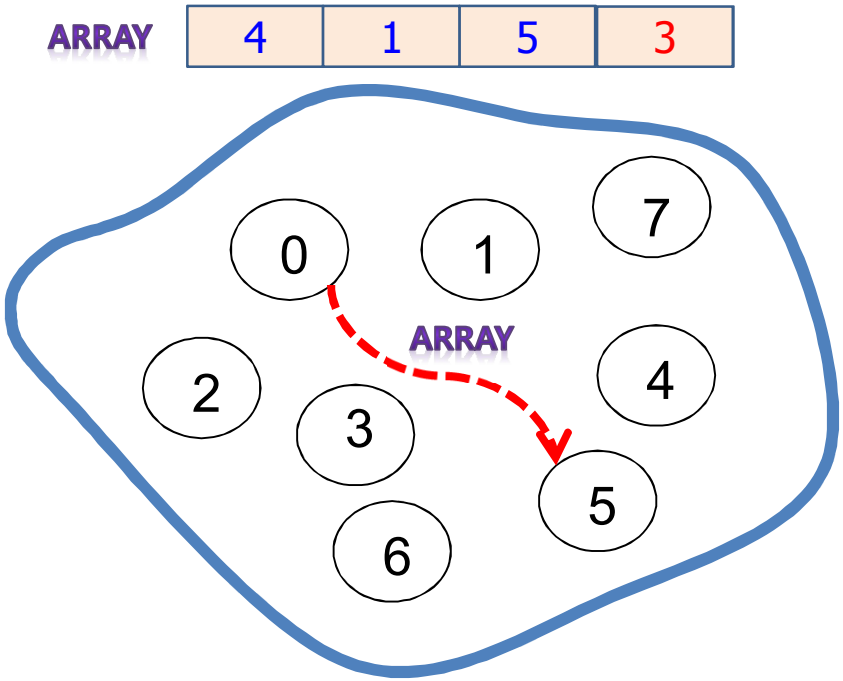
- A group of processes that communicate with each other
  - All MPI communication calls have a communicator argument
  - Most often you will use MPI_COMM_WORLD



MPI_Send and MPI_Recv

- MPI_Send(buf, count, datatype, dest, tag, comm)
  - ‘buf’, ‘count’, ‘datatype’
  - ‘dest’, ‘tag’, ‘comm’
- MPI_Recv(buf, count, datatype, dest, tag, comm, status)
  - ‘buf’, ‘count’, ‘datatype’
  - ‘dest’, ‘tag’, ‘comm’ ,’status’

MPI Message and Tag



buf = array a[]
count = 4
datatype = int (8 byte for DP)

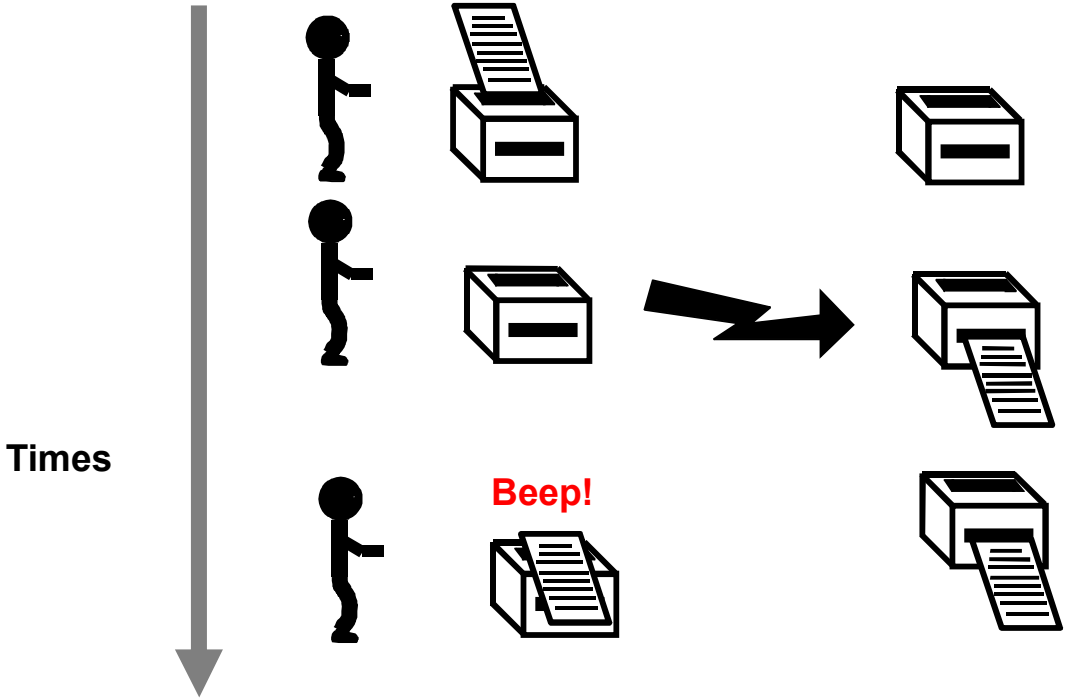
dest = rank(5)
tag = 0
comm= MPI_COMM_WORLD

MPI Data Types

MPI Data Type	C Data Type
MPI_CHAR – 1 Byte character	signed char
MPI_SHORT – 2 Byte integer	signed short int
MPI_INT – 4 Byte integer	signed int
MPI_LONG – 4 Byte integer	signed long int
MPI_UNSIGNED_CHAR – 1 Byte u char	unsigned char
MPI_UNSIGNED_SHORT – 2 Byte u int	unsigned short int
MPI_UNSIGNED – 4 Byte u int	unsigned int
MPI_UNSIGNED_LONG– 4 Byte u int	unsigned long int
MPI_FLOAT – 4 Byte float point	float
MPI_DOUBLE – 8 Byte float point	double
MPI_LONG_DOUBLE- 8 Byte float point	long double

Blocking Communication

- MPI_SEND/MPI_RECV are blocking communication
  - Only return from the MPI procedure call when the operation (whether it's send or receive) has completed



Simple Communication in MPI : C

```
/* Example name : pingpong */
int main(int argc, char *argv[])
{
   int myrank, nprocs, tag = 55, root = 0;
   int send = -1, recv = -1;
   MPI_Status status;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

   if (myrank == root) {
      printf("Before : myrank(%d) send = %d, recv = %d\n", myrank, send, recv);
      send = rank + 7;
      MPI_Send(&send, 1, MPI_INTEGER, 1, tag, MPI_COMM_WORLD);
   }
   else {
      MPI_Recv(&recv, 1, MPI_INTEGER, root, tag, MPI_COMM_WORLD, &status);
      printf("After  : myrank(%d) send = %d, recv = %d\n", myrank, send, recv);
   }
   MPI_Finalize();
   return 0;
}
```

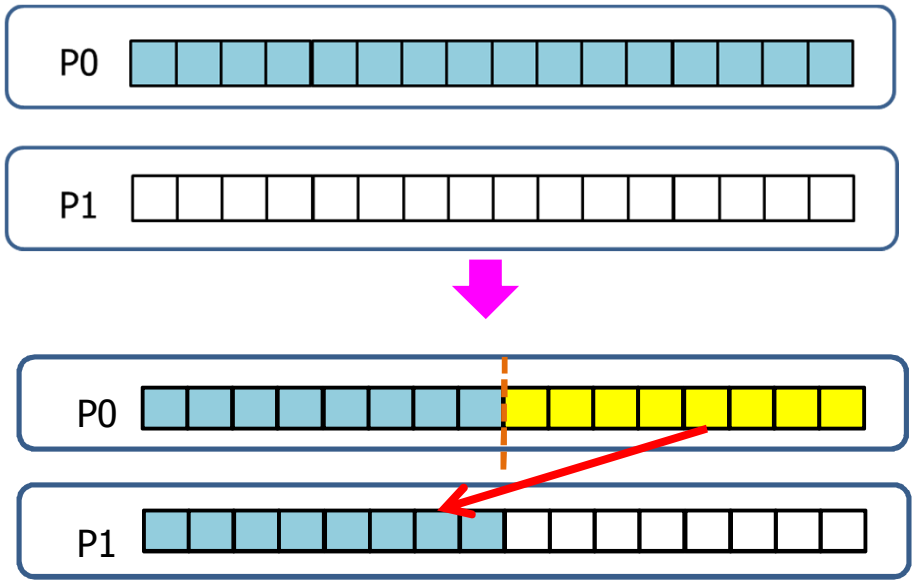
```
Before : myrank(0) send = -1, recv = -1
After  : myrank(1) send = -1, recv = 7
```

Simple Communication in MPI : Fortran

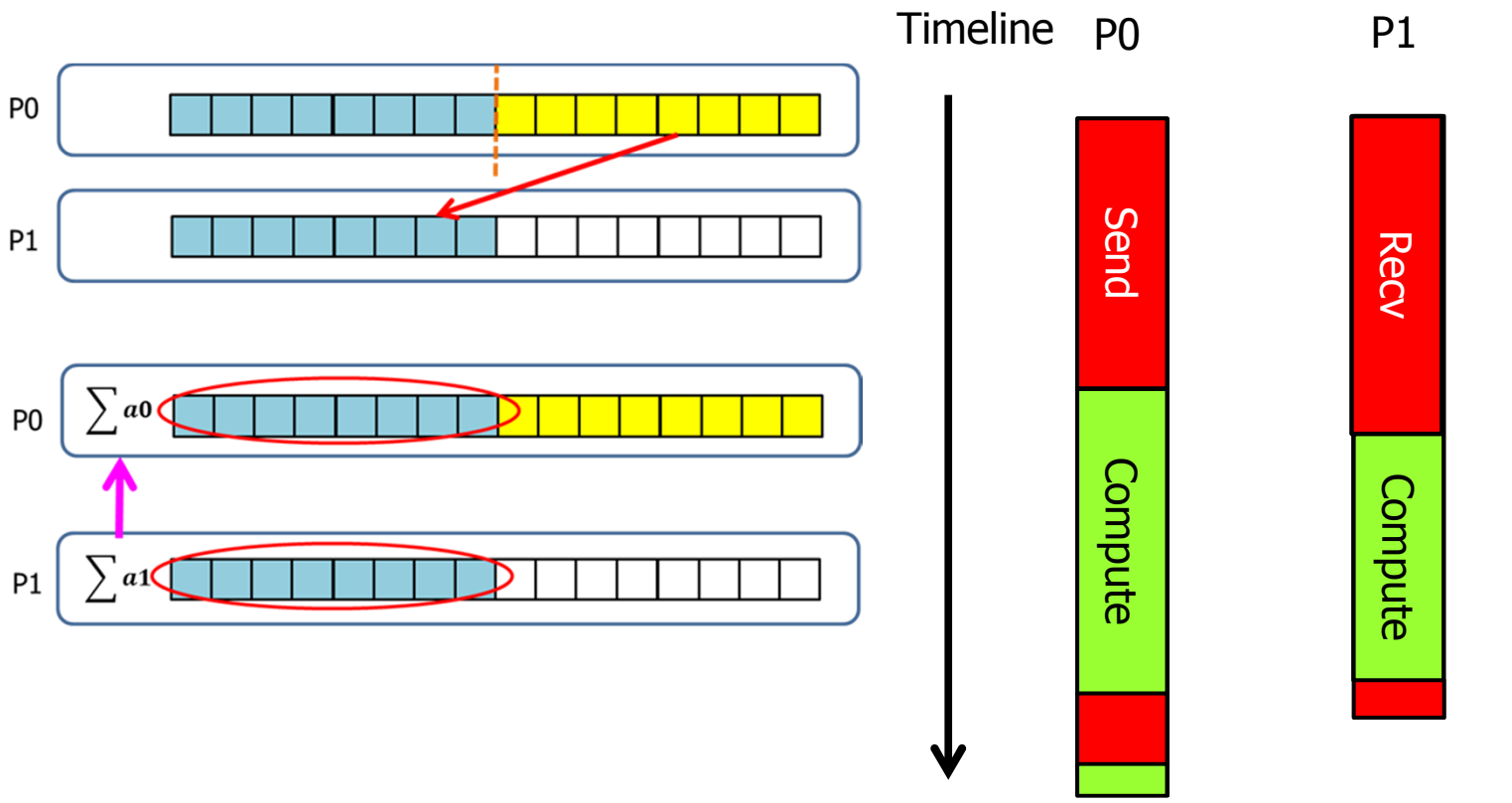
```
! Example name : p2p.f90
PROGRAM P2P
  IMPLICIT NONE
  INCLUDE 'mpif.h'
  INTEGER myrank, nprocs, ierr, status(MPI_STATUS_SIZE)
  INTEGER :: tag = 55, ROOT = 0
  INTEGER :: send = -1, recv = -1

  CALL MPI_INIT(ierr)
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

  IF (myrank == ROOT) THEN
    PRINT *, 'Before : myrank = ', myrank, 'send = ', send, 'recv = ', recv
    send = 7
    CALL MPI_SEND(send,1,MPI_INTEGER,1,tag, MPI_COMM_WORLD, ierr)
  ELSE
    CALL MPI_RECV(recv, 1, MPI_INTEGER,ROOT,tag, MPI_COMM_WORLD, status, ierr)
    PRINT *, 'After  : myrank = ', myrank, 'send = ',send, 'recv = ', recv
  END IF

  CALL MPI_FINALIZE(ierr)

END PROGRAM P2P
```

Case study #1: Parallel Sum

- Array is originally on root process (P0)
- Parallel algorithm
  - Scatter (nprocs=2)
    - Half of the array is sent to process P1
  - Compute
    - P0 & P1 sum independently their segment
  - Reduction
    - Partial sum on P1 sent to P0



Parallel sum



P0 sums the partial sums

Parallel Sum (1/2)

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[]){
    int i,rank,nprocs;
    int N=100;
    int localN[2];
    double *array;
    double sum,psum;
    MPI_Status status;

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

    localN[0]=N/2;
    localN[1]=N-N/2;
    array=malloc(sizeof(double)*N);
    if(rank==0){
        for(i=0;i<N;i++)
            array[i]=1.0;
    }
```

Parallel Sum (2/2)

```
    if(rank==0)
       MPI_Send(&array[localN[0]],localN[1],MPI_DOUBLE,1,10,MPI_COMM_WORLD);
    if(rank==1)
       MPI_Recv(array,localN[1],MPI_DOUBLE,0,10,MPI_COMM_WORLD,&status);

    psum=0;
    for(i=0;i<localN[rank];i++){
      psum+=array[i];
    }
    if(rank==0)
       MPI_Recv(&sum,1,MPI_DOUBLE,1,11,MPI_COMM_WORLD,&status);
    if(rank==1)
       MPI_Send(&psum,1,MPI_DOUBLE,0,11,MPI_COMM_WORLD);

    if (rank==0){
       sum+=psum;
       printf("Sum is %g, partial sum %g\n",sum,psum);
    }
    free(array);
    MPI_Finalize();
}
```

Special Parameter Values

MPI_Send(buf, count, datatype,dest,tag, comm)

Information	API	
dest	MPI_PROC_NULL	Null destination, no operation takes place
comm	MPI_COMM_WORLD	Includes all processes
error	MPI_SUCCESS	Operation successful

Special Parameter Values

MPI_Recv(buf, count, datatype,source,tag, comm, status)

Info.	API	
source	MPI_PROC_NULL	No sender, no operation takes place
source	MPI_ANY_SOURCE	Receive from any sender tag
tag	MPI_ANY_TAG	Receive messages with any tag
status	MPI_STATUS_IGNORE	Do not store any status data
comm	MPI_COMM_WORLD	Includes all processes
error	MPI_SUCCESS	Operation successful

Status object

MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)

Information	Fortran	C
rank of the sender	status(MPI_SOURCE)	status.MPI_SOURCE
tag	status(MPI_TAG)	status.MPI_TAG
Error	Status(MPI_ERROR)	Status.MPI_ERROR
count	MPI_GET_COUNT()	MPI_Get_count()

Case study #2 : More Sum (1/2)

```
int main(int argc, char *argv[]){
   int i,rank,nprocs,N;
   double *array;
   double sum,psum;
   int *localN;
   MPI_Status status;
   int startIndex;

   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
   N=100*nprocs;
   localN=malloc(sizeof(int)*nprocs);
   if(rank==0){
      for(i=0;i<nprocs;i++)
         localN[i]=N/nprocs;
      for(i=0;i<N%nprocs;i++)
         localN[i]++;
   }
   array=malloc(sizeof(double)*N);
   if(rank==0){
      for(i=0;i<N;i++)
         array[i]=1.0;
   }
```

```
 if(rank==0)    {
       startIndex=localN[0];
       for(i=1;i<nprocs;i++){
          MPI_Send(&array[startIndex],localN[i],MPI_DOUBLE,i,10,MPI_COMM_WORLD);
          startIndex+=localN[i];
       }
    }
    else{
       MPI_Recv(array,N,MPI_DOUBLE,0,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
       MPI_Get_count(&status,MPI_DOUBLE,&(localN[rank]));
    }
    psum=0;
    for(i=0;i<localN[rank];i++){
       psum+=array[i];
    }
    if(rank==0) {
       sum=psum;
       for(i=1;i<nprocs;i++){
          MPI_Recv(&psum,1,MPI_DOUBLE,MPI_ANY_SOURCE,
                MPI_ANY_TAG,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
          sum+=psum;
       }
       printf("Sum is %g\n",sum);
    }
    else
       MPI_Send(&psum,1,MPI_DOUBLE,0,11,MPI_COMM_WORLD);
    free(array);    free(localN);    MPI_Finalize();
}
```

Wallclock Time

- Wallclock time (real time, elapsed time)
  - CPU, I/O, communication, and other jobs
  - gettimeofday() — C/C++
    - Resolution up to microseconds.
  - MPI_Wtime() — C/C++/Fortran
    - start = MPI_Wtime();
    - <code to be timed>
    - finish = MPI_Wtime();

Information About Messages

- MPI_Wait
  - Blocking until communication is completed.
  - Useful for both sender and receiver of *nonblocking* comm.
- MPI_Test
  - Non Blocking check for status of communication.
  - Useful for both sender and receiver of *nonblocking* comm.
- MPI_Probe
  - Receiver is notified (i.e., this is a **blocking** call) when messages from potentially *any* sender arrive and are ready to be processed.

Agenda

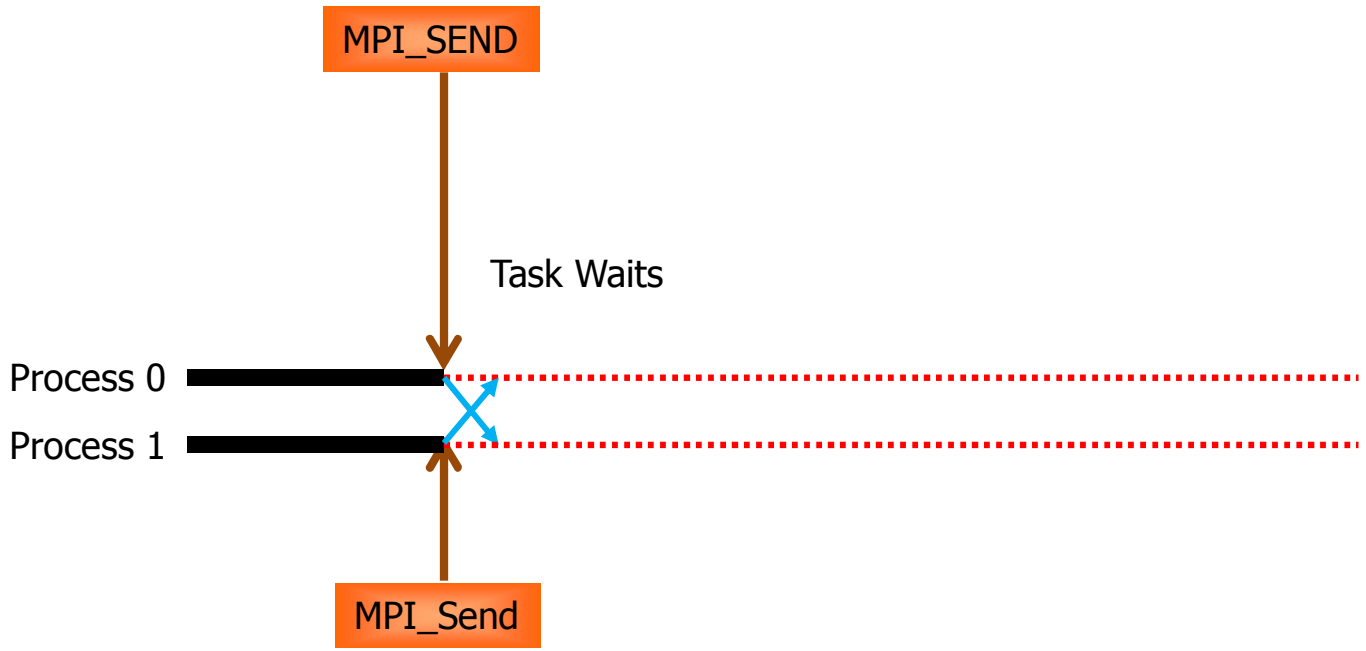
- What is MPI?
  - How to write a parallel program in MPI
  - Basic MPI
  - Point-to-point communication
- Slightly more intermediated topics:
  - Non-blocking communication in MPI
  - Collective communication in MPI
  - Lab2
- Summary and final Q&A

Blocking rountines and Deadlocks

- Blocking routines
  - Completion depends on other processes
  - Risk for deadlocks
    - the program is stuck forever
  - MPI_Send exits once the send buffer can be safely read and written to
  - MPI_Recv exits once it has received the message in the receive buffer

Deadlock

- All of the sends may block, waiting for a matching receive (will for large enough messages)



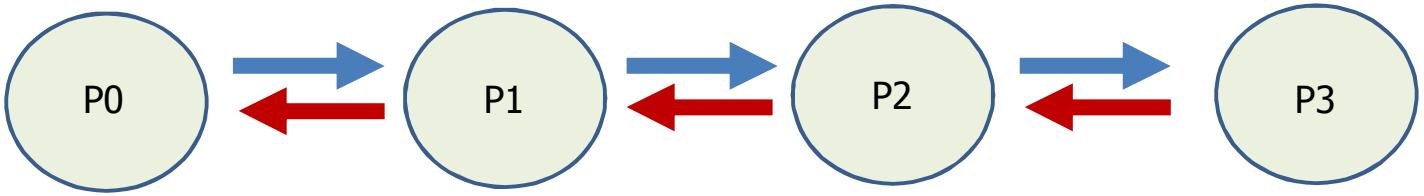
Combined send & receive

- Sends one message and receives another one, with one single command
  - Reduces risk for deadlocks
- Destination rank and source rank can be different, or same

```
MPI_Sendrecv(sendbuf, sendcount, sendtype, dest,
sendtag, recvbuf, recvcount, recvtype, source, recvtag,
comm, status )
```

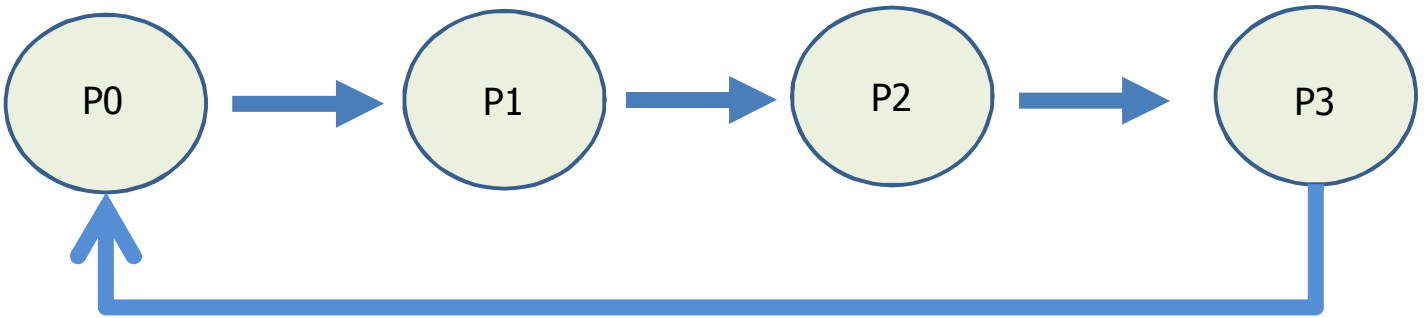
Sendrecv Communication Patterns

Pairwise exchange




```
MPI_Send(data,right); MPI_Recv(data,left)
MPI_Send(data,left);  MPI_Recv(data,right)
```

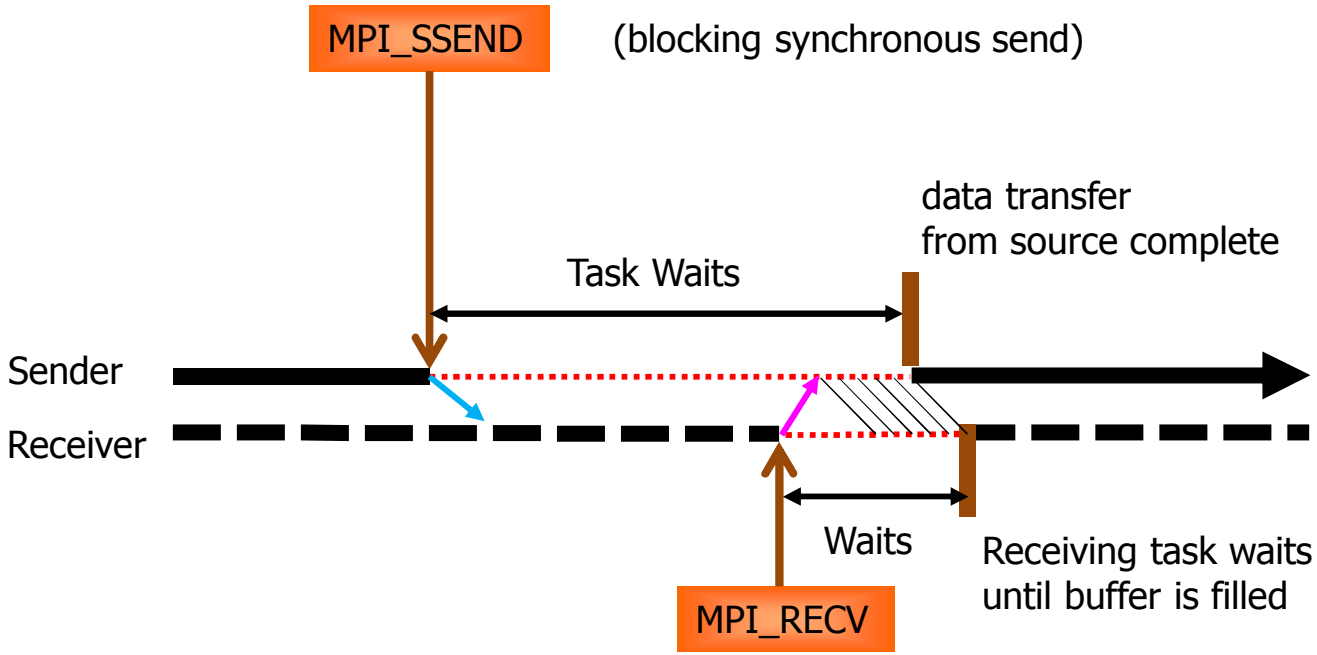
Pipe, a ring of processes exchanging data



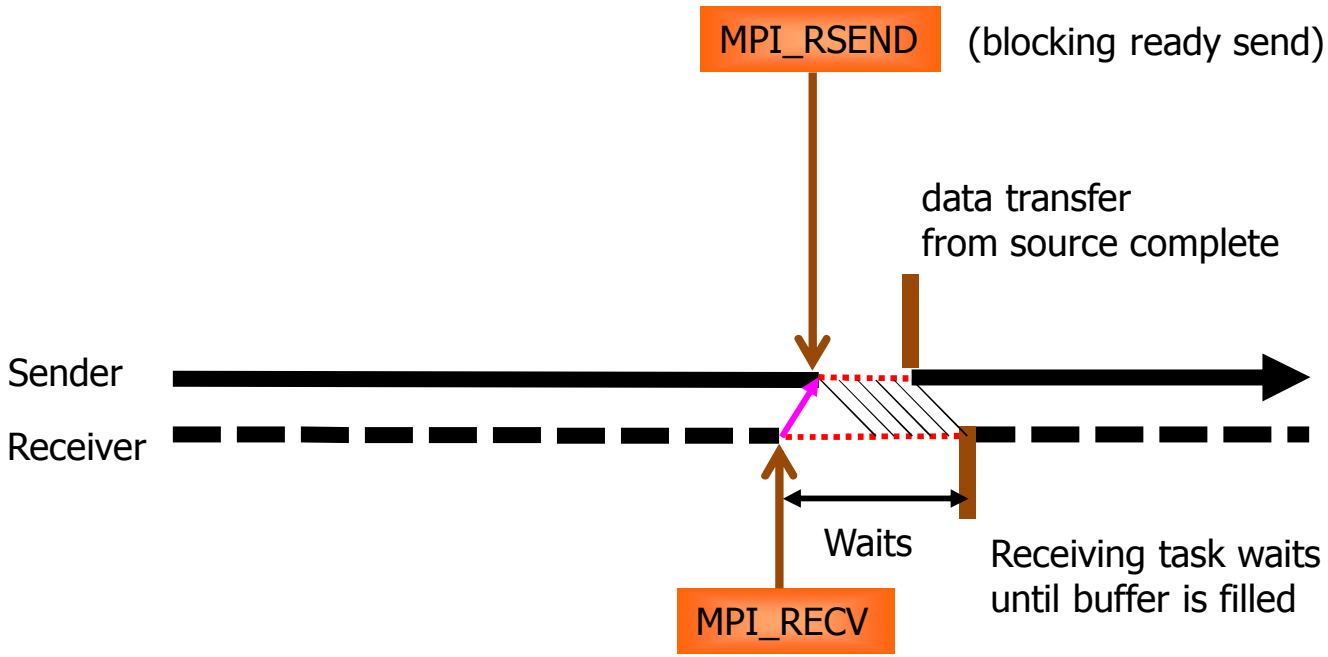
Communication Mode

Mode	MPI Call Routine	
	Blocking	Non Blocking
Synchronous	MPI_SSEND	MPI_ISSEND
Ready	MPI_RSEND	MPI_IRSEND
Buffer	MPI_BSEND	MPI_IBSEND
Standard	MPI_SEND	MPI_ISEND
Recv	MPI_RECV	MPI_IRECV

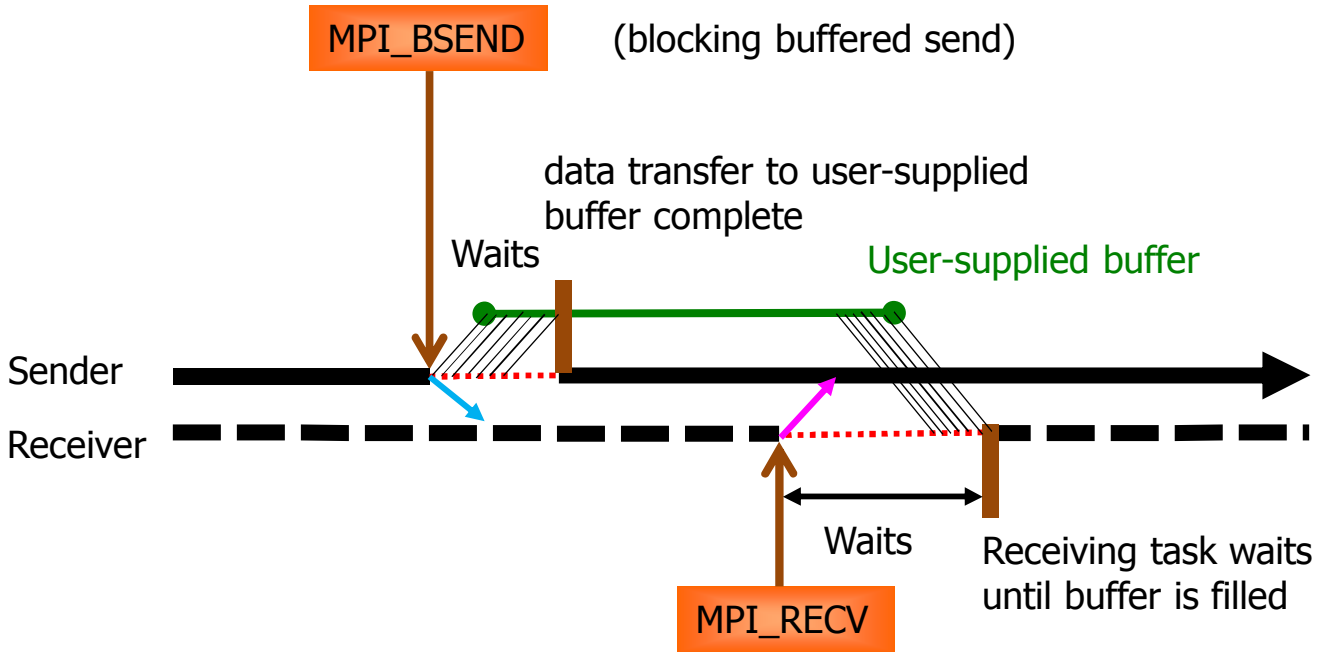
(Blocking) Synchronous Send



(Blocking) Ready Send



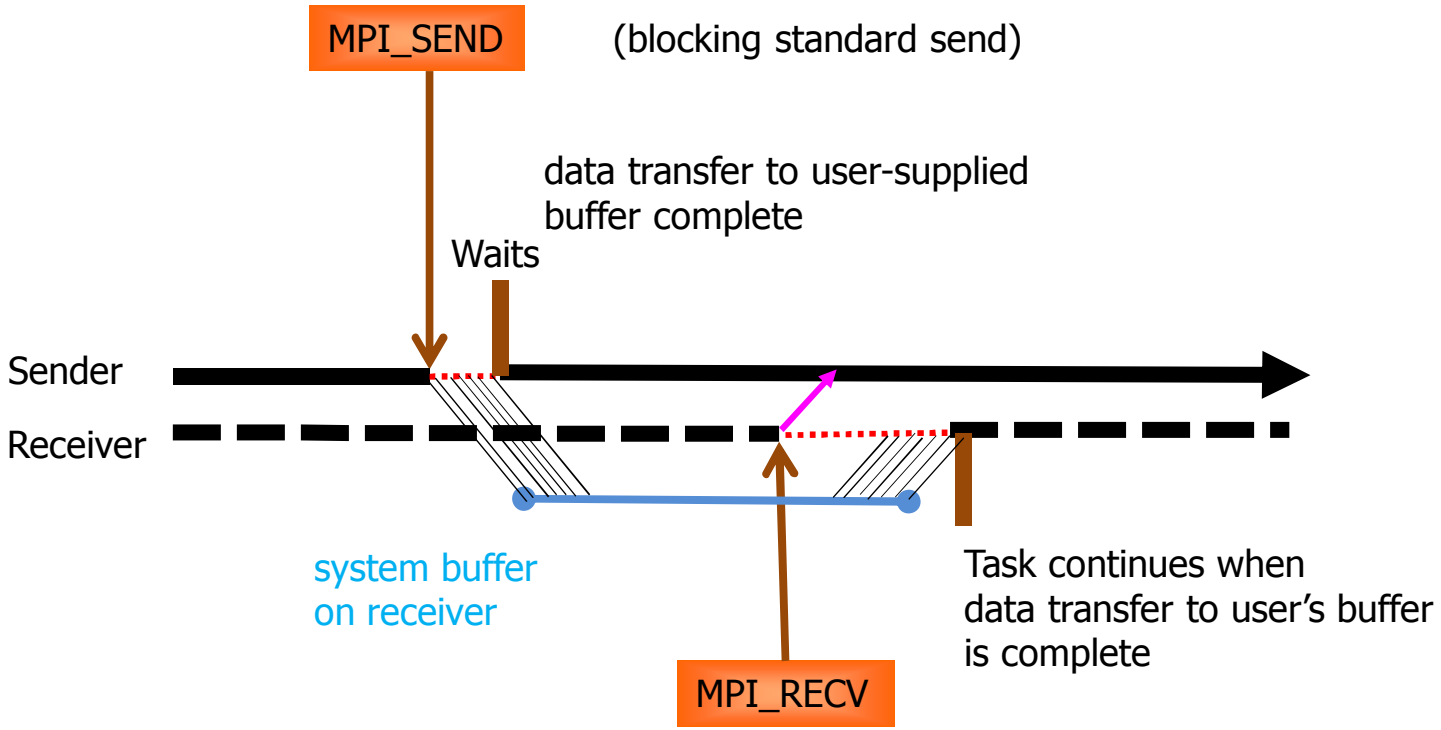
(Blocking) Buffered Send



MPI_Buffer_attach, MPI_Buffer_detach

(Blocking) Standard Send

CASE I : message size < threshold (~ kb)



CASE II : message size > threshold (~ kb) then takes Ssend mode

Non-blocking Communication

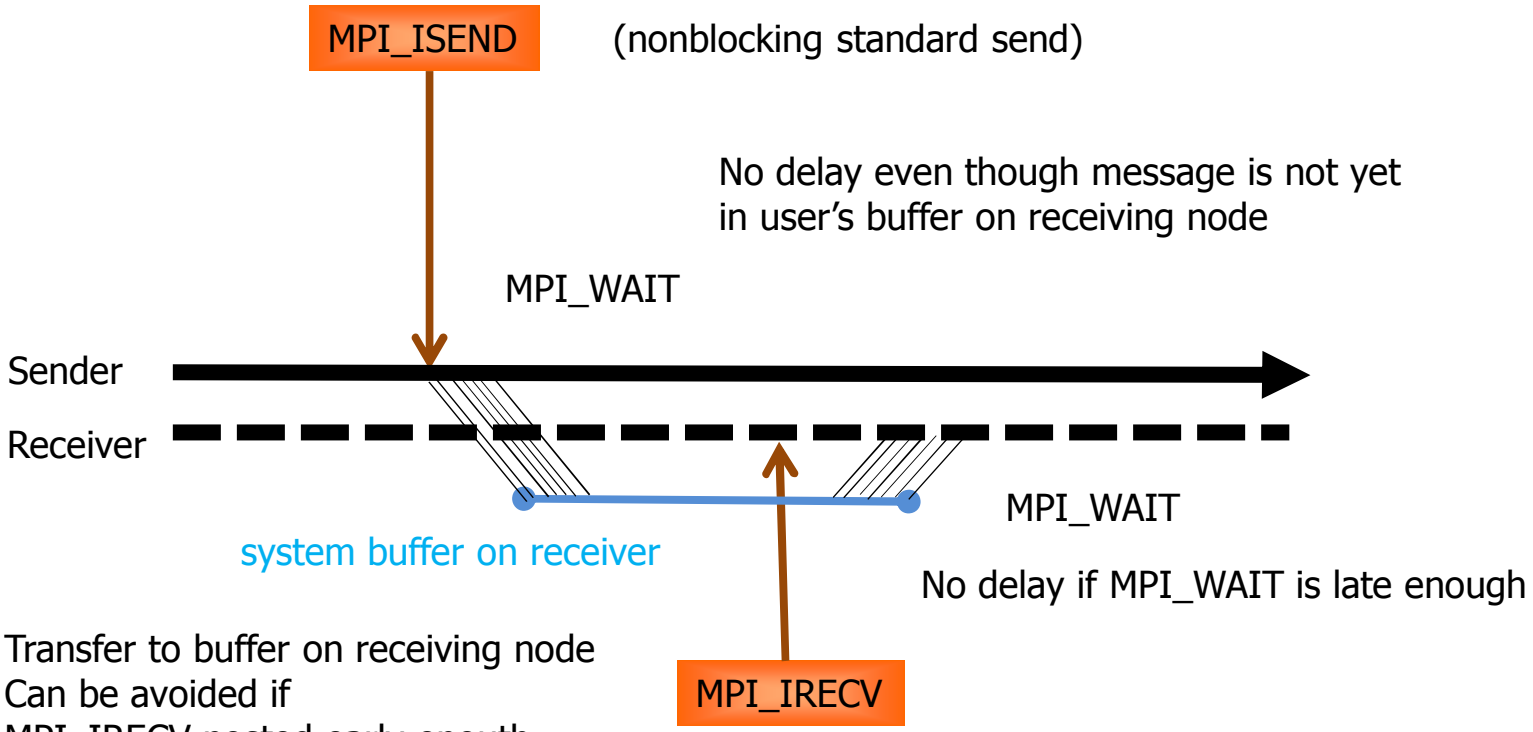
- Non-blocking (asynchronous) operations return (immediately) "request handles" that can be waited
  - MPI_ISEND(start, count, datatype, dest, tag, comm, request)
  - MPI_IRECV(start, count, datatype, src, tag, comm, request)
  - MPI_WAIT(request, status)
- One can also test without waiting using MPI_TEST
  - MPI_TEST(request, flag, status)
- Anywhere you use MPI_SEND or MPI_RECV, you can use the pair of MPI_ISEND/MPI_WAIT or MPI_IRECV/MPI_WAIT

Non-blocking communication

- Non-blocking behavior
  - Method returns immediately
  - Unsafe to use buffers before actual completion
  - Check status of call (Wait / Test)
- Allows overlapping of computation and communication.

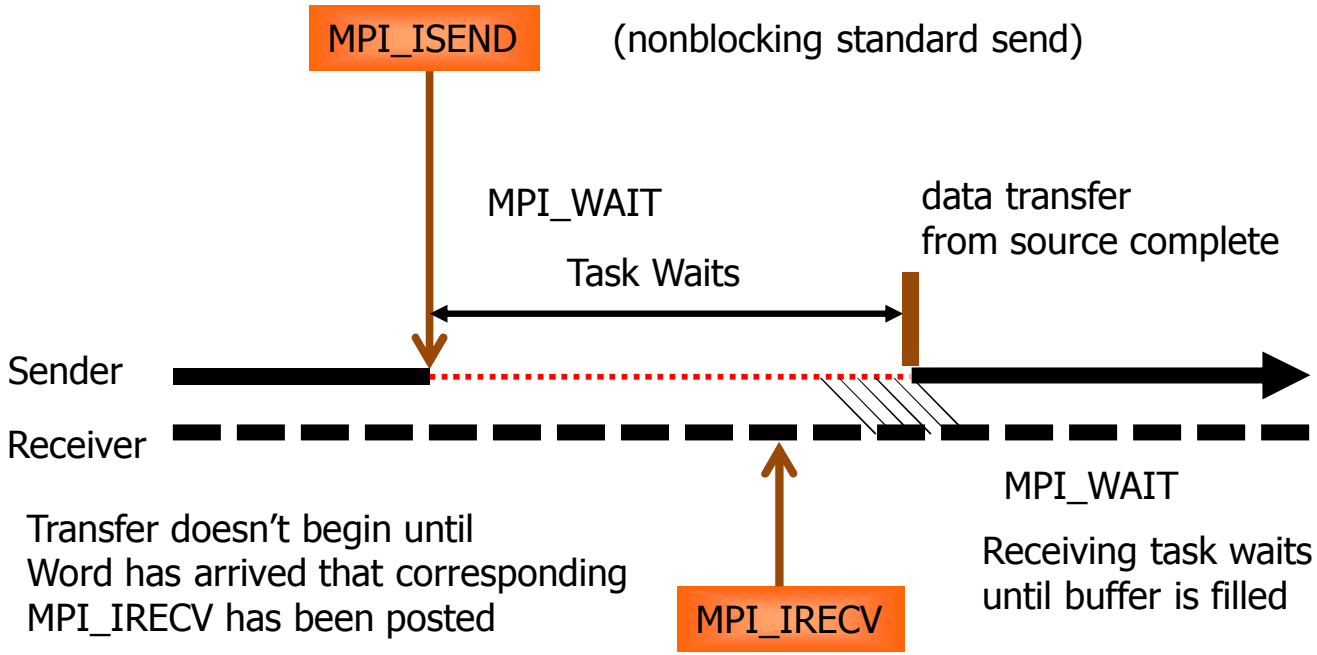
(non-blocking) Standard Send

CASE I : message size < threshold (~ kb)

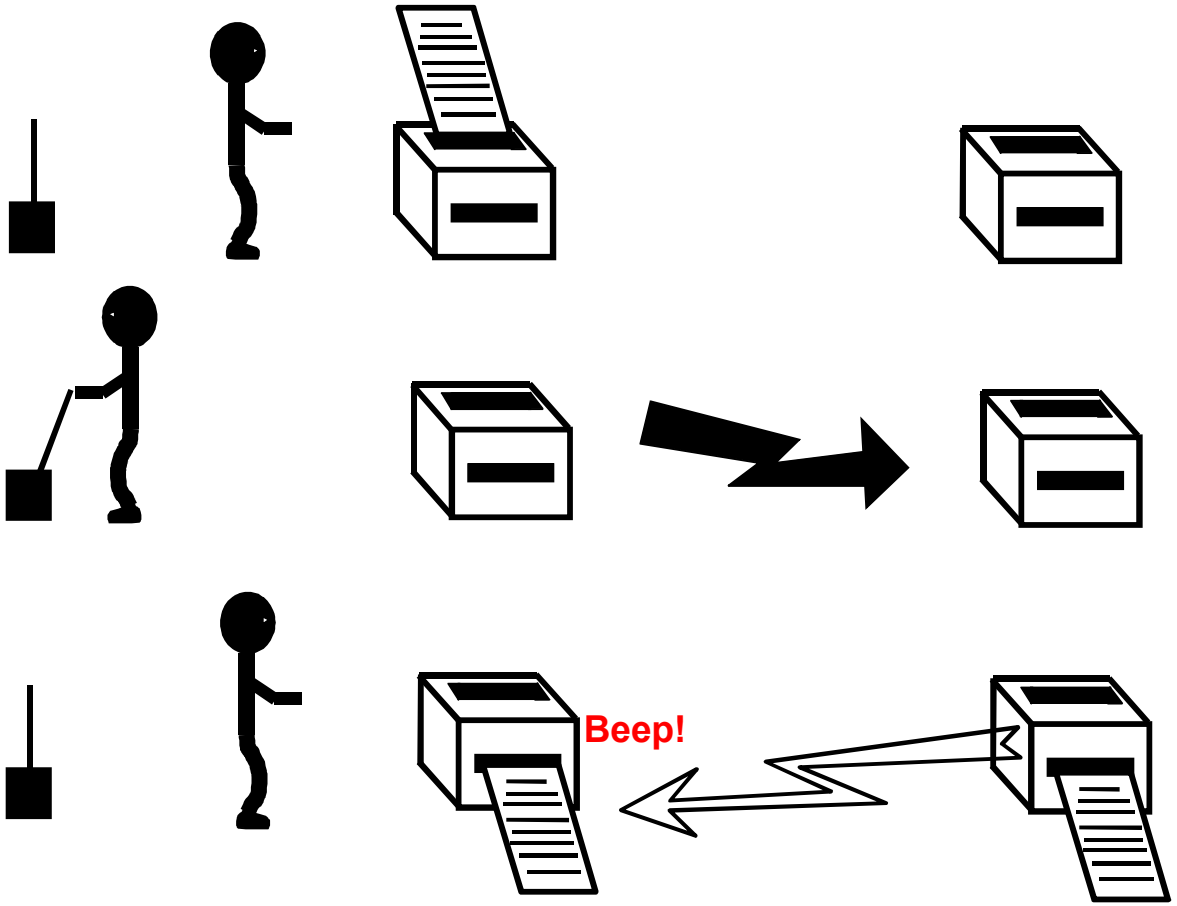


(non-blocking) Standard Send

CASE II : message size > threshold (~ kb)



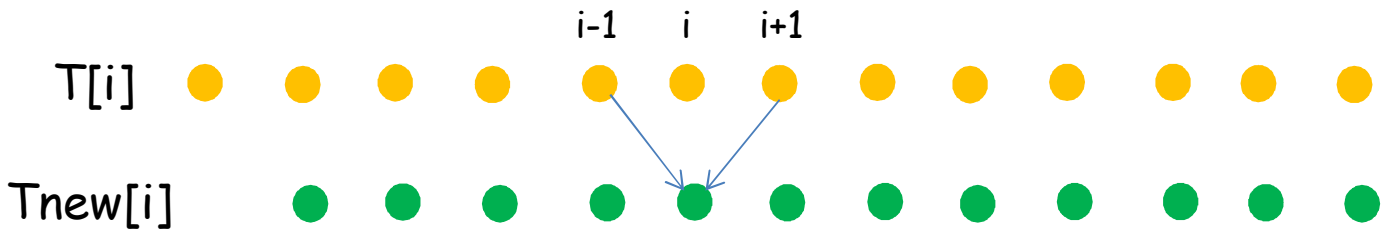
Synchronous Communicatoin



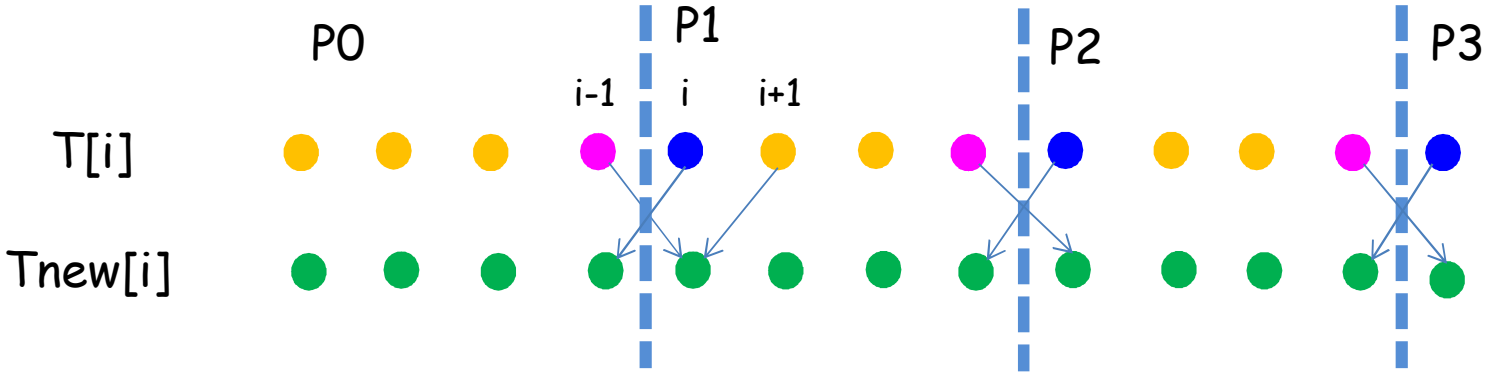
Simple parallelization example

Serial code:

```
main (intargc, char **argv) {
double T[N], Tnew[N];
T[0] = 1.0;

for (inti=1; i<N-1; i++)
    Tnew[i] = 0.5d0*(T[i-1] + T[i+1]);
}
```



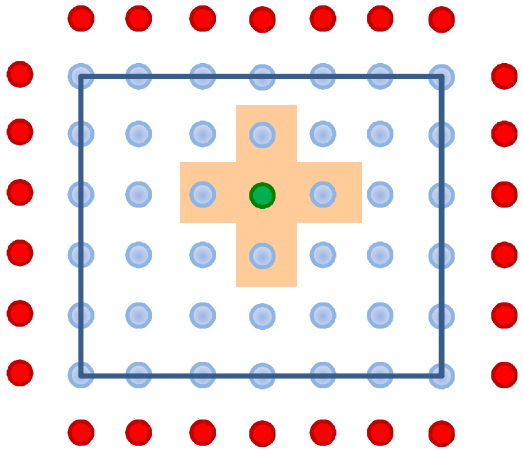
Simple parallelization example

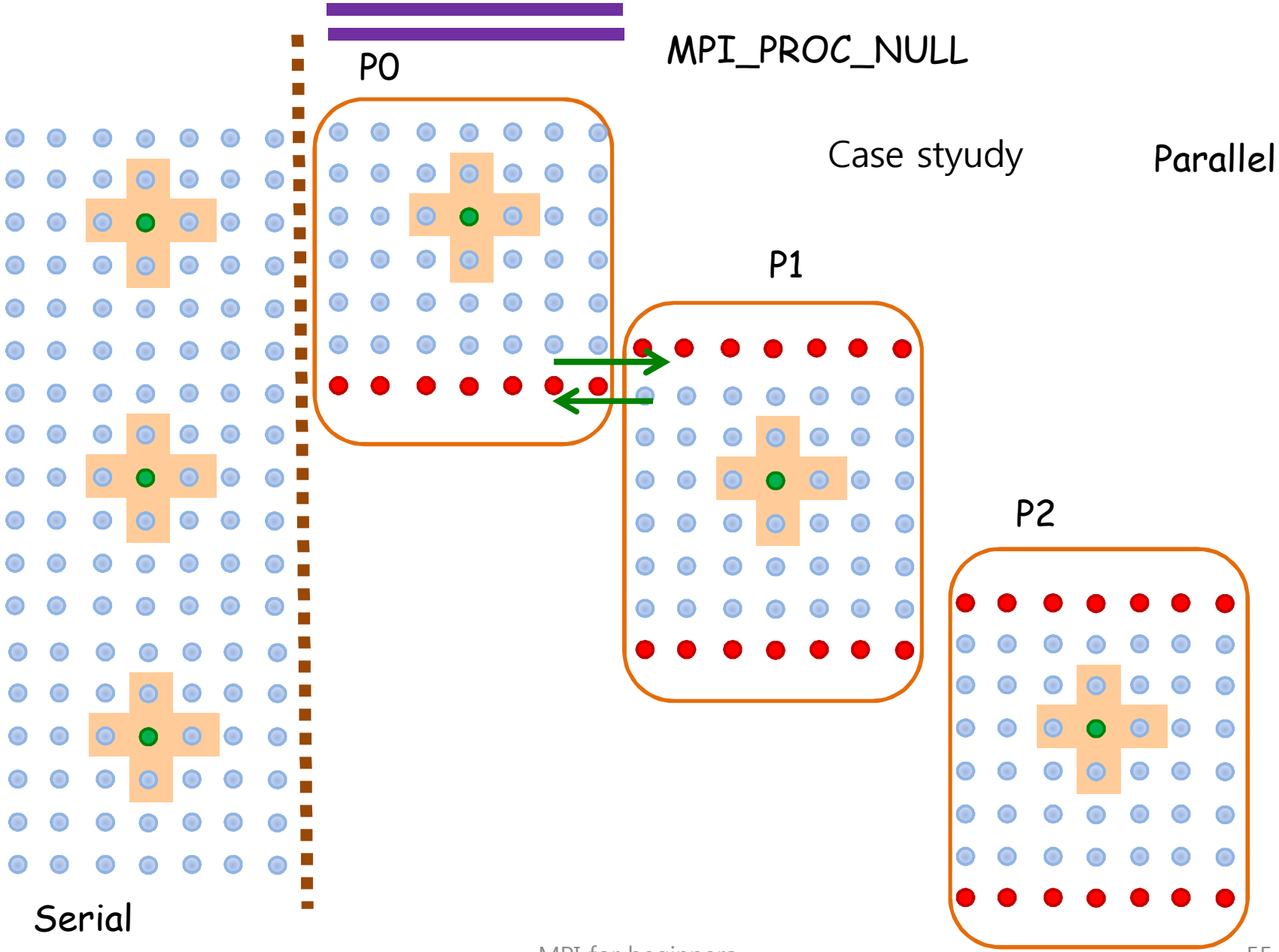


Case styudy #4 : Jacobi solver (1/4)

- Jacobi iterator is a way of solving Poisson's equation using an iterative scheme

$$Vnew(i,j) = [V(i-1,j)+V(i+1,j)+ V(i,j-1)+V(i,j+1)-beta(i,j)]/4$$




MPI_PROC_NULL
P0
Case styudy
Parallel
P1
P2
Serial

MPI_Send and MPI_Recv (3/4)

```
...
ngbr_up=rank-1;
ngbr_down=rank+1;
if(ngbr_up<0) ngbr_up=MPI_PROC_NULL;
if(ngbr_down>=nprocs) ngbr_down=MPI_PROC_NULL;
for(i=0;i<max_iterations;i++){
if(rank%2==0){
MPI_Send(V[1],M,MPI_DOUBLE,ngbr_up,10,MPI_COMM_WORLD);
MPI_Recv(V[0],M,MPI_DOUBLE,ngbr_up,11,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
MPI_Send(V[N],M,MPI_DOUBLE,ngbr_down,12,MPI_COMM_WORLD);
MPI_Recv(V[N+1],M,MPI_DOUBLE,ngbr_down,13,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
}
if(rank%2==1){
MPI_Recv(V[N+1],M,MPI_DOUBLE,ngbr_down,10,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
MPI_Send(V[N],M,MPI_DOUBLE,ngbr_down,11,MPI_COMM_WORLD);
MPI_Recv(V[0],M,MPI_DOUBLE,ngbr_up,12,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
MPI_Send(V[1],M,MPI_DOUBLE,ngbr_up,13,MPI_COMM_WORLD);
}
compute(V,Vn,beta,N,M);
SWAP(V,Vn,dpptemp);
}
...
```

Using MPI_Sendrecv (4/4)

- MPI_Sendrecv
  - allows each processor to simultaneously send to one processor and receive from another.
    - P_1 could send to P_0 while simultaneously receiving from P_2

```
MPI_Sendrecv(sendbuf, sendcount, sendtype, dest,
sendtag, recvbuf, recvcount, recvtype, source, recvtag,
comm, status )
```

Agenda

- What is MPI?
  - How to write a parallel program in MPI
  - Basic MPI
  - Point-to-point communication
- Slightly more intermediated topics:
  - Non-blocking communication in MPI
  - Collective communication in MPI
  - Lab2
- Summary and final Q&A

Collective Communication

- Communications involving a group of processes
  - Called by all processes in a communicator

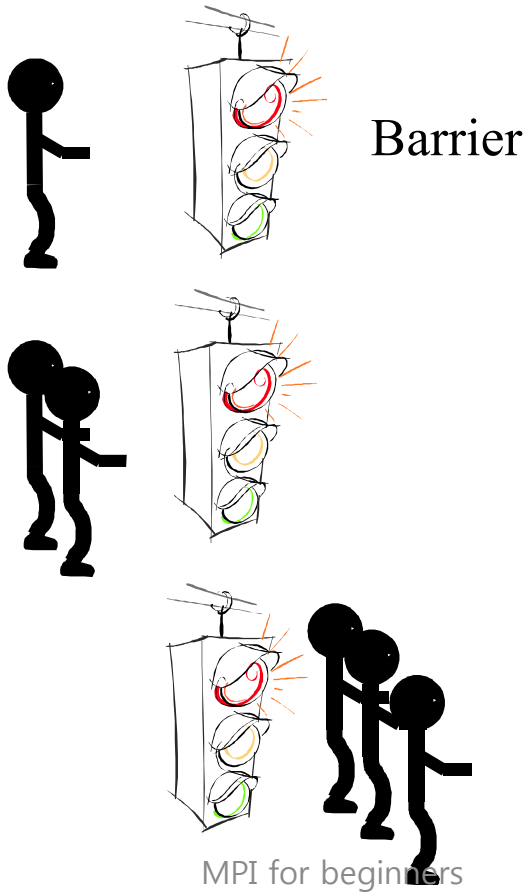
Category	Subroutines
One buffer	MPI_BCAST
One send buffer and one receive buffer	MPI_GATHER, MPI_SCATTER, MPI_ALLGATHER, MPI_ALLTOALL, MPI_GATHERV, MPI_SCATTERV, MPI_ALLGATHERV, MPI_ALLTOALLV
Reduction	MPI_REDUCE, MPI_ALLREDUCE, MPI_SCAN, MPI_REDUCE_SCATTER
Synchronization	MPI_BARRIER, MPI_OP_CREATE, MPI_OP_FREE

Characteristics of Collective Comms

- Collective action over a communicator.
- All processes must communicate.
- Synchronisation may or may not occur.
- All collective operations are blocking.
- No tags.
- Receive buffers must be exactly the right size.

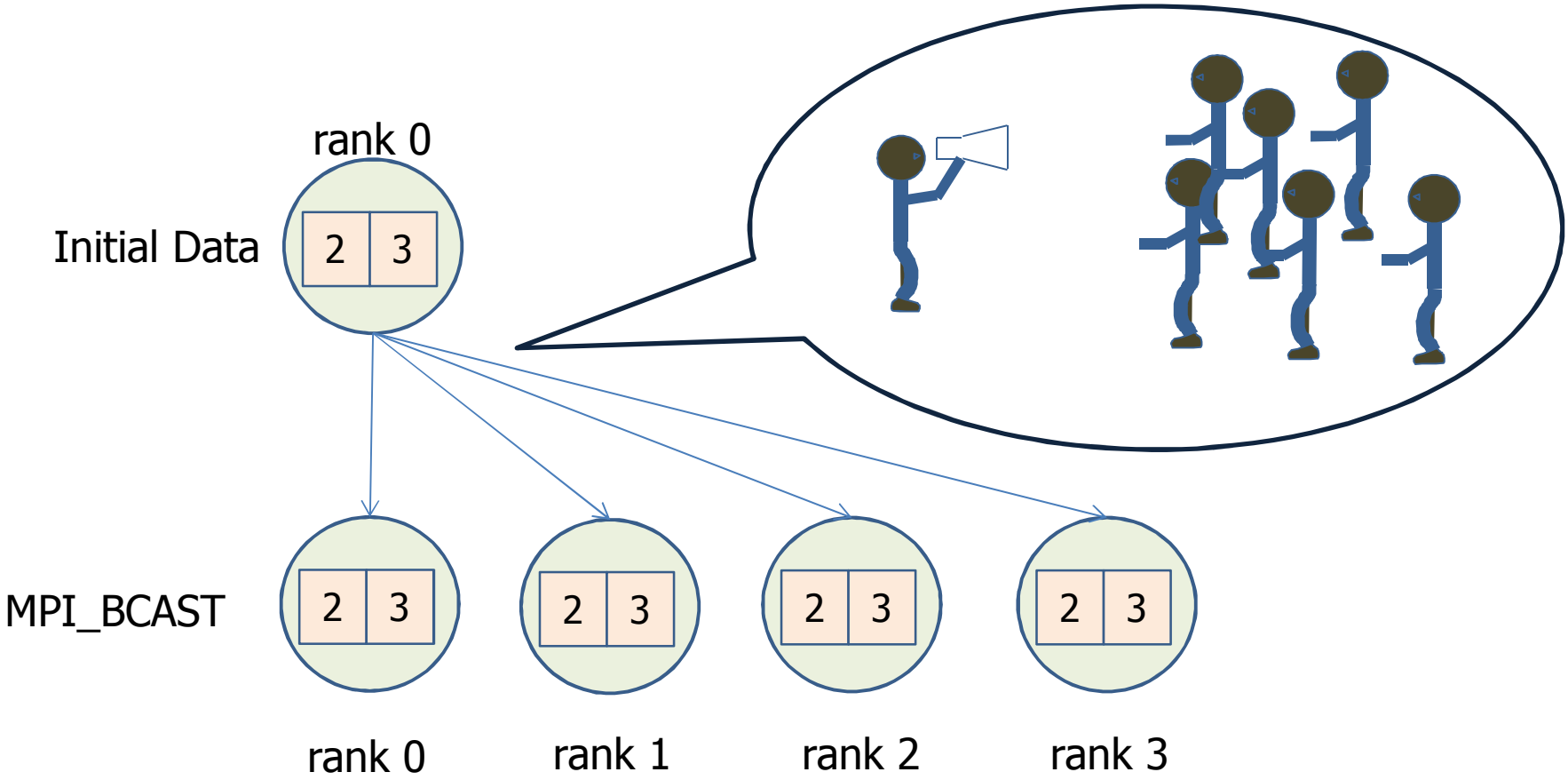
Synchronization

- int MPI_Barrier(MPI_Comm comm)
  - Blocks until all processes in the communicator have reached here



MPI BCAST

```
int MPI_Bcast (void *buffer, int count,
      MPI_Datatype datatype, int root, MPI_Comm comm)
```



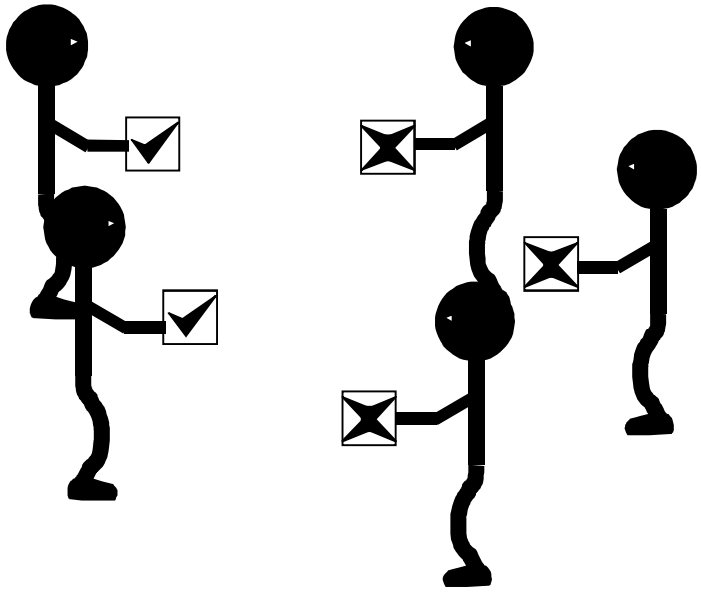
MPI_Bcast (examples)

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[]){
   int  i, rank, nprocs;
   int  buf[2] = { 0, 0 };
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   if (rank == 0)  {
      buf[0] = 2; buf[1] = 3;
   }
   printf("rank (%d) :            Before :  ", nrank);
   for (i=0; i<2; i++) printf(" %d", buf[i]);

    MPI_Bcast(buf,4, MPI_INT, 0, MPI_COMM_WORLD);

   printf("\n rank (%d) :        After  :  ", nrank);
   for (i=0; i<2; i++) printf(" %d", buf[i]);
   printf("\n");
   MPI_Finalize();
}
```

Reduction Operations

- Combine data from several processes to produce a single result

Strike



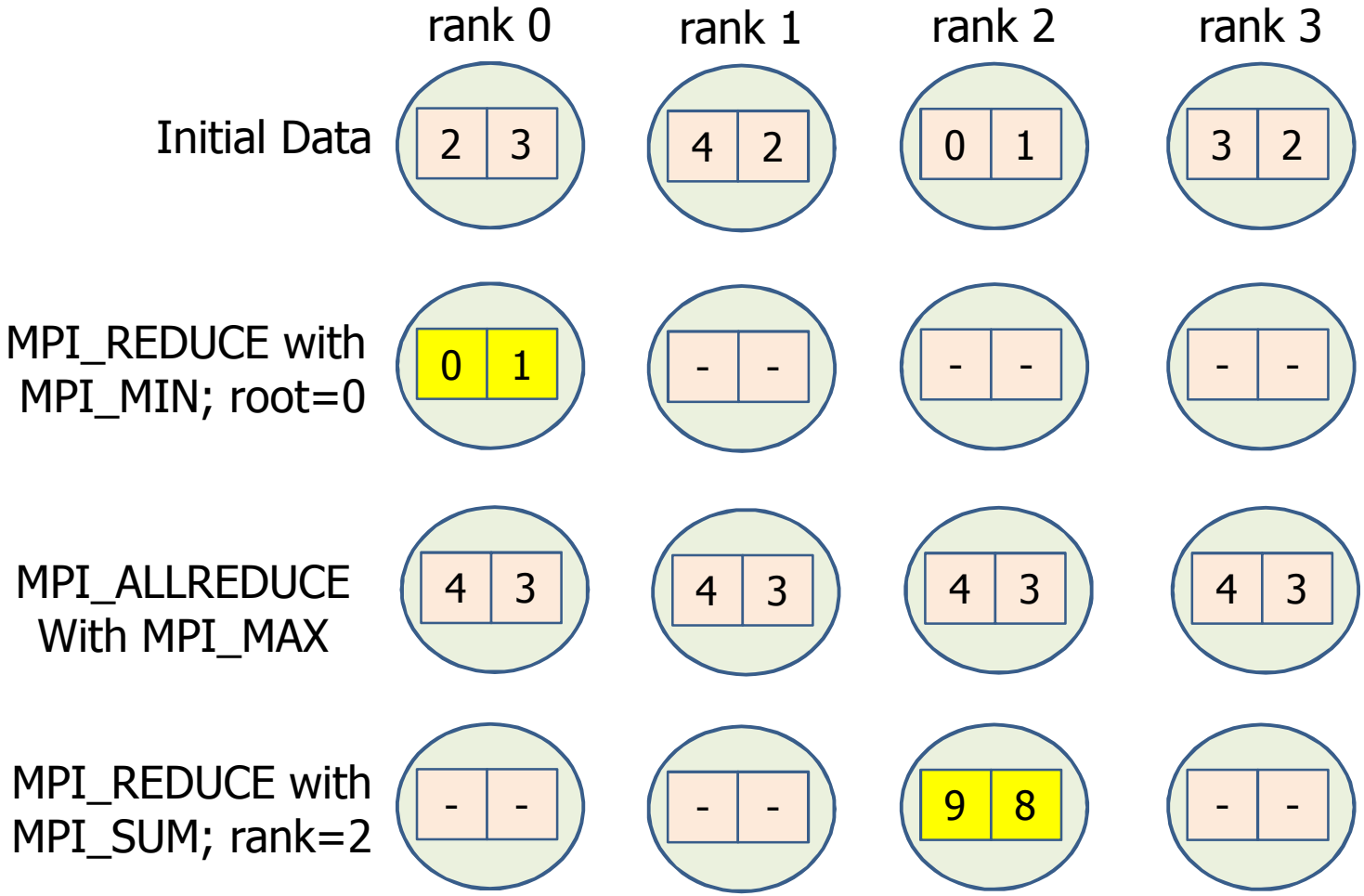
Global Reduction Operations

- Reduction
  - Used to compute a result involving data distributed over a group of processes
  - Examples:
    - global sum or product
    - global maximum or minimum
    - global user-defined operation

MPI_REDUCE

Operation	Data Type (C)
MPI_SUM(sum), MPI_PROD(product) MPI_MAX(maximum), MPI_MIN(minimum)	MPI_INT, MPI_LONG, MPI_SHORT, MPI_UNSIGNED_SHORT, MPI_UNSIGNED MPI_UNSIGNED_LONG, MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE
MPI_MAXLOC(max value and location), MPI_MINLOC(min value and location)	MPI_FLOAT_INT, MPI_DOUBLE_INT, MPI_LONG_INT, MPI_2INT, MPI_SHORT_INT, MPI_LONG_DOUBLE_INT
MPI_LAND(logical AND), MPI_LOR(logical OR), MPI_LXOR(logical XOR)	MPI_INT, MPI_LONG, MPI_SHORT, MPI_UNSIGNED_SHORT, MPI_UNSIGNED MPI_UNSIGNED_LONG
MPI_BAND(bitwise AND), MPI_BOR(bitwise OR), MPI_BXOR(bitwise XOR)	MPI_INT, MPI_LONG, MPI_SHORT, MPI_UNSIGNED_SHORT, MPI_UNSIGNED MPI_UNSIGNED_LONG, MPI_BYTE

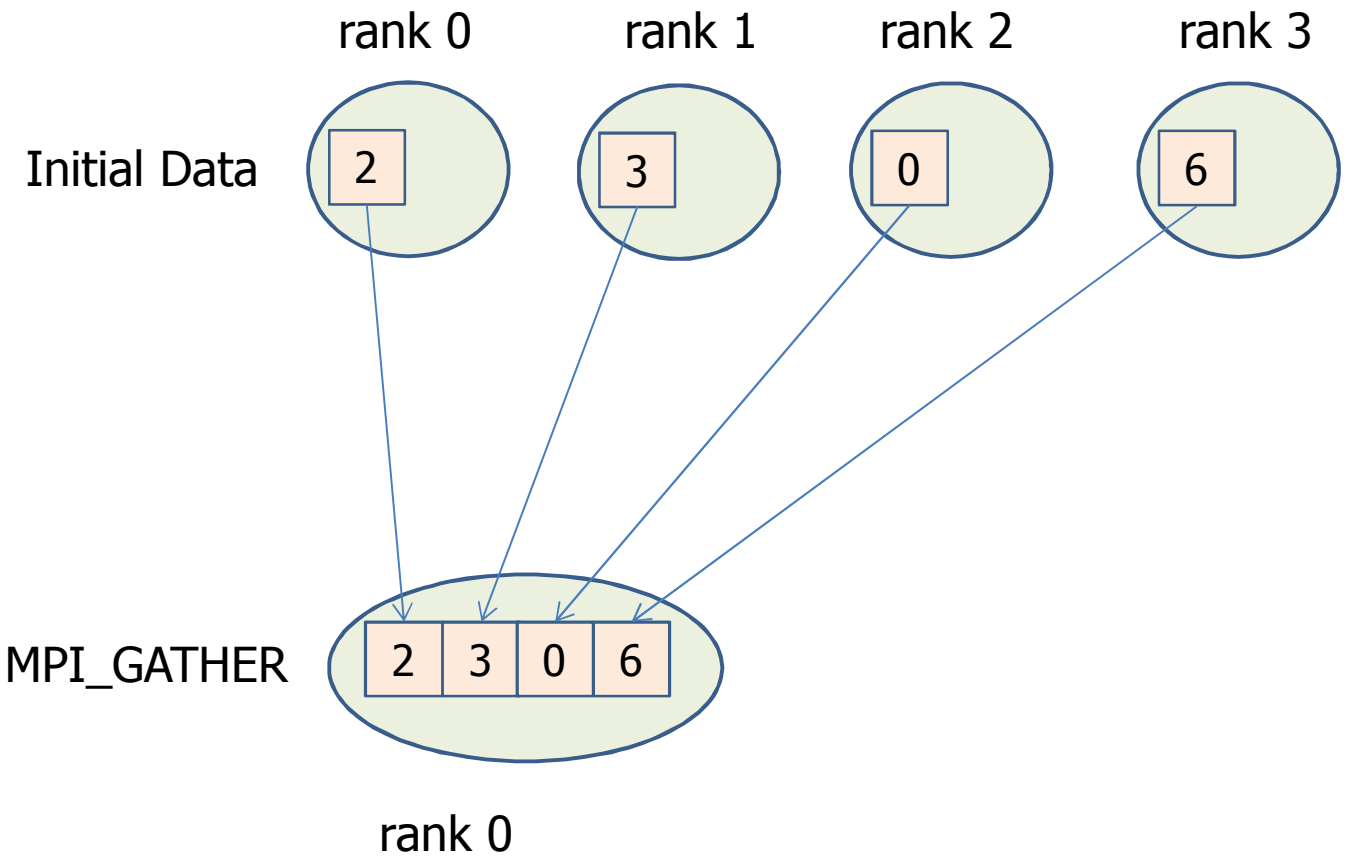
Reduction Operations (examples)



MPI_Reduce (Example)

```
int main(int argc, char *argv[]){
      int i, rank, start, end;
    double a[9], sum=0.0, total_sum = 0.0;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &nrank);
    start = rank * 3;            end = start + 2;
    for (i=start; i<end+1; i++) {
          a[i] = i + 1;
          if (i == start)   printf("rank (%d) ", rank);
          printf("a[%d] = %.2f   ", i, a[i]);
    }
    for (i=start; i<end+1; i++)   sum = sum + a[i];
    MPI_Reduce(&sum,  &total_sum,  1,  MPI_DOUBLE,  MPI_SUM,  0,
    MPI_COMM_WORLD);
    if (nrank == 0)
        printf("\nrank(%d):sum= %.2f.\n",rank, total_sum);
    MPI_Finalize();
}
```

MPI_GATHER

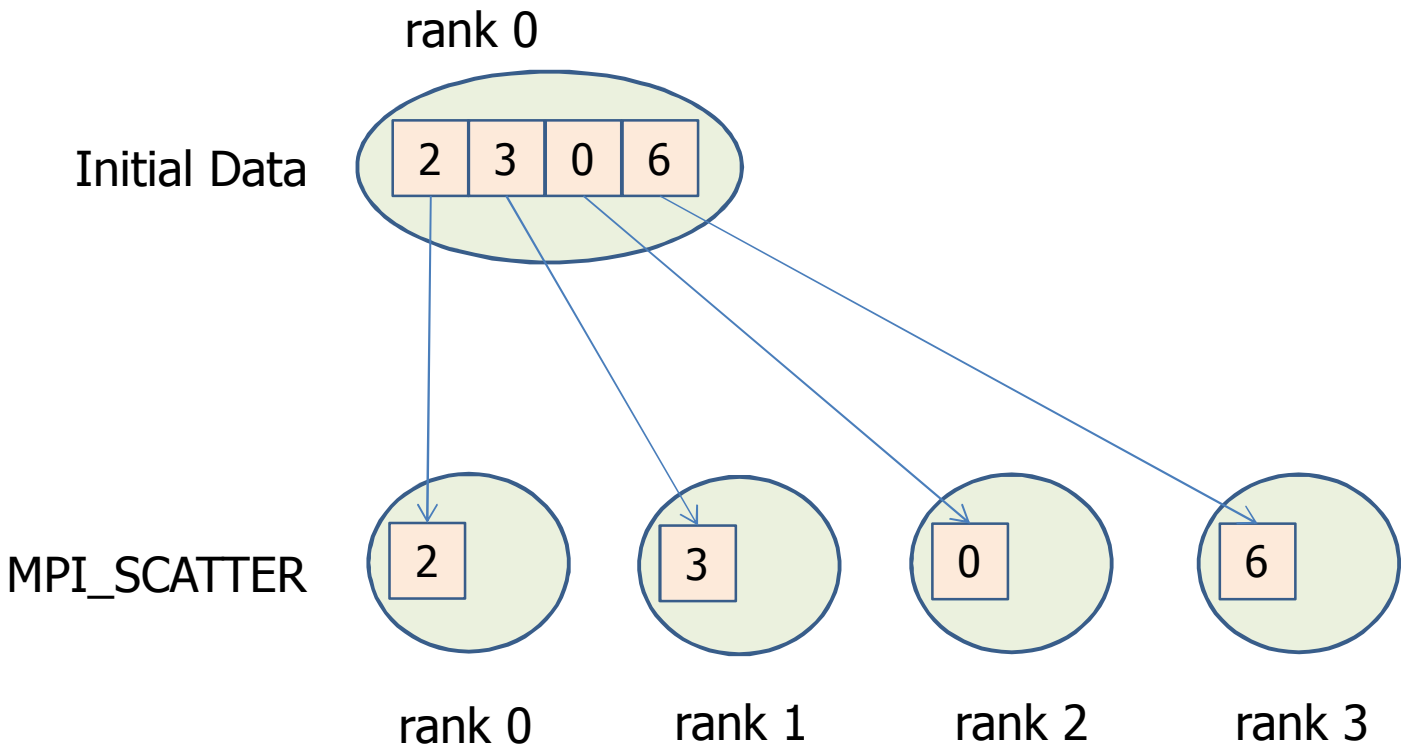

rank 0
rank 1
rank 2
rank 3
Initial Data
2
3
0
6
MPI_GATHER
2
3
0
6
rank 0

MPI_GATHER (Example)

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[]){
    int i, nprocs, rank,  isend, irecv[4];
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    isend = rank + 1;
    printf("rank (%d) : isend = %d\n", rank, isend);

    MPI_Gather(&isend, 1, MPI_INT, irecv, 1, MPI_INT, 0,
     MPI_COMM_WORLD);
    if (rank == 0)
        for (i=0; i<3; i++)
            printf("rank (%d) : irecv[%d] = %d\n",
                    nrank, i, irecv[i]);
    MPI_Finalize();
}
```

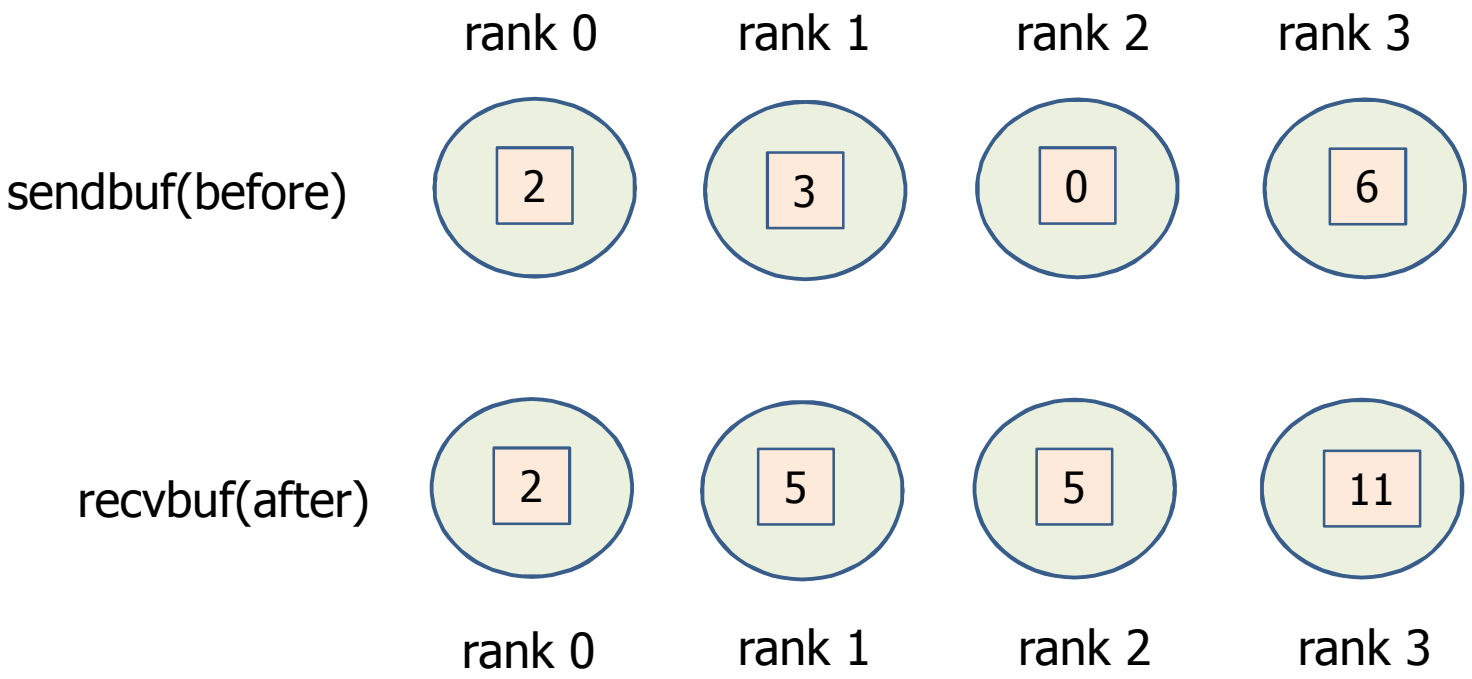
MPI_SCATTER

MPI_Scatter(sendbuf, sendcount, MPI_INT, recvbuf, recvcount, MPI_INT, root, MPI_COMM_WORLD)

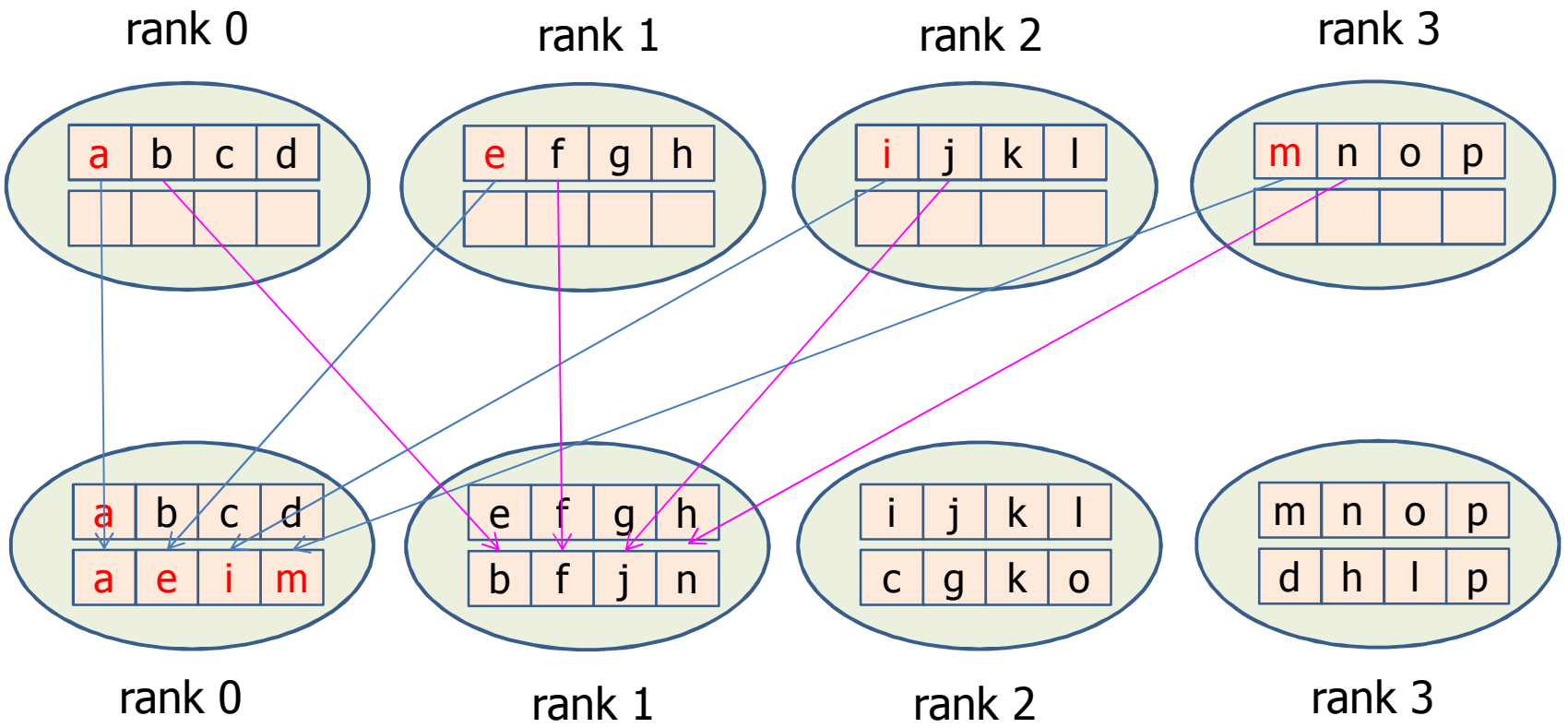


MPI_SCAN

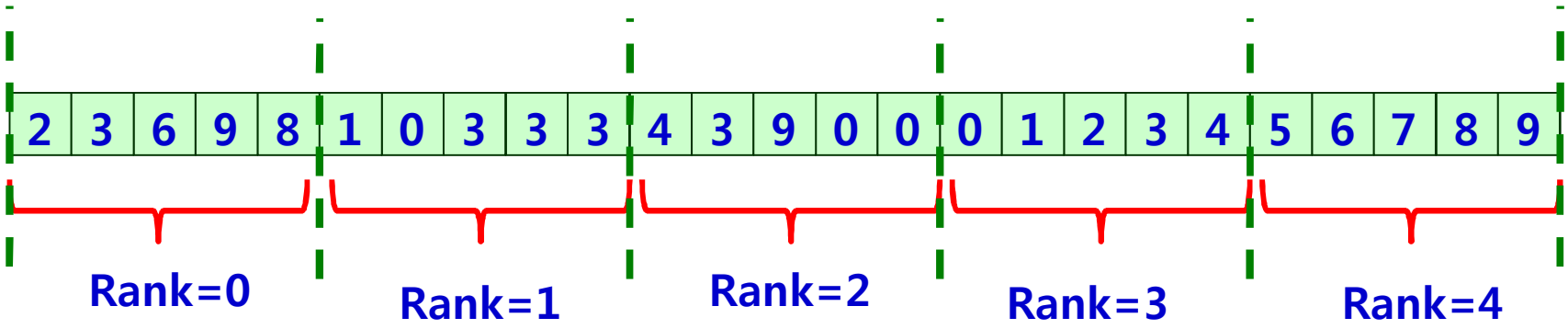
MPI_Scan(sendbuf, recvbuf, count, MPI_INT, MPI_SUM, MPI_COMM_WORLD)



MPI_ALLTOALL



Case styudy #3: counting 3's (1/3)




```
int main(int argc, char **argv)  {
    const int length = 100000000,  iters = 10;
    int myid, nprocs, length_per_process, i, j, p;
    int *myArray, *gArray, root, tag, myCount, gCount;
    FILE *fp; double t1, t2;

    myCount = 0; root=0;   tag=0;
  length_per_process=length/nprocs;
  myArray=(int *)malloc(length_per_process*sizeof(int));
```

mpi_counting3s.c (2/3)

```
if (myid==root {
   gArray=(int *)malloc(length*sizeof(int));
   for (i=0; i< length;i++) gArray[i]=i%10;
}
if(myid==root) {
   t1 = MPI_Wtime();
   for(p=0; p<nprocs-1; p++) {
      for(i=0;i< length_per_process;i++){
         j = i + p*length_per_process;
         myArray[i]=gArray[j];
      }
      MPI_Send(myArray, length_per_process, MPI_INT, p+1, tag,
            MPI_COMM_WORLD);
   }
}else{
   MPI_Recv(myArray, length_per_process, MPI_INT, root, tag,
         MPI_COMM_WORLD, &status);
}
```

mpi_counting3s.c (3/3)

```
 for (j=0; j< iters; j++){
     for(i=0; i<length_per_process; i++) {
         if(myArray[i]==3)
             myCount++;
        }
    }
MPI_Reduce(&myCount,&gCount,1,MPI_INT,MPI_SUM root, MPI_COMM_WORLD);
  if(myid==root) {
     t2 = MPI_Wtime();
    }
    MPI_Finalize();
}
```

```
$> ./serial.x
    serial=1       Number of 3's: 100000000   Elapsed Time =   3.05412793 (sec)
$> mpirun -np 10 -machinefile hostname ./parallel.x -O3
    nprocs=10    Number of 3's: 100000000   Elapsed Time =   0.22715100 (sec)
```

Summary

- Parallelism is critical today
- MPI is an industry standard model for parallel programming
  - A large number of implementations of MPI exist
  - Gives user explicit control on data management
  - Widely used by many scientific applications with great success
- Your application can be next!