IBM Korea Strategic Business

Basic Parallel Programming with MPI

ByungUn Ha (buha@kr.ibm.com)

Deep Computing Team, IBM Korea

Contents

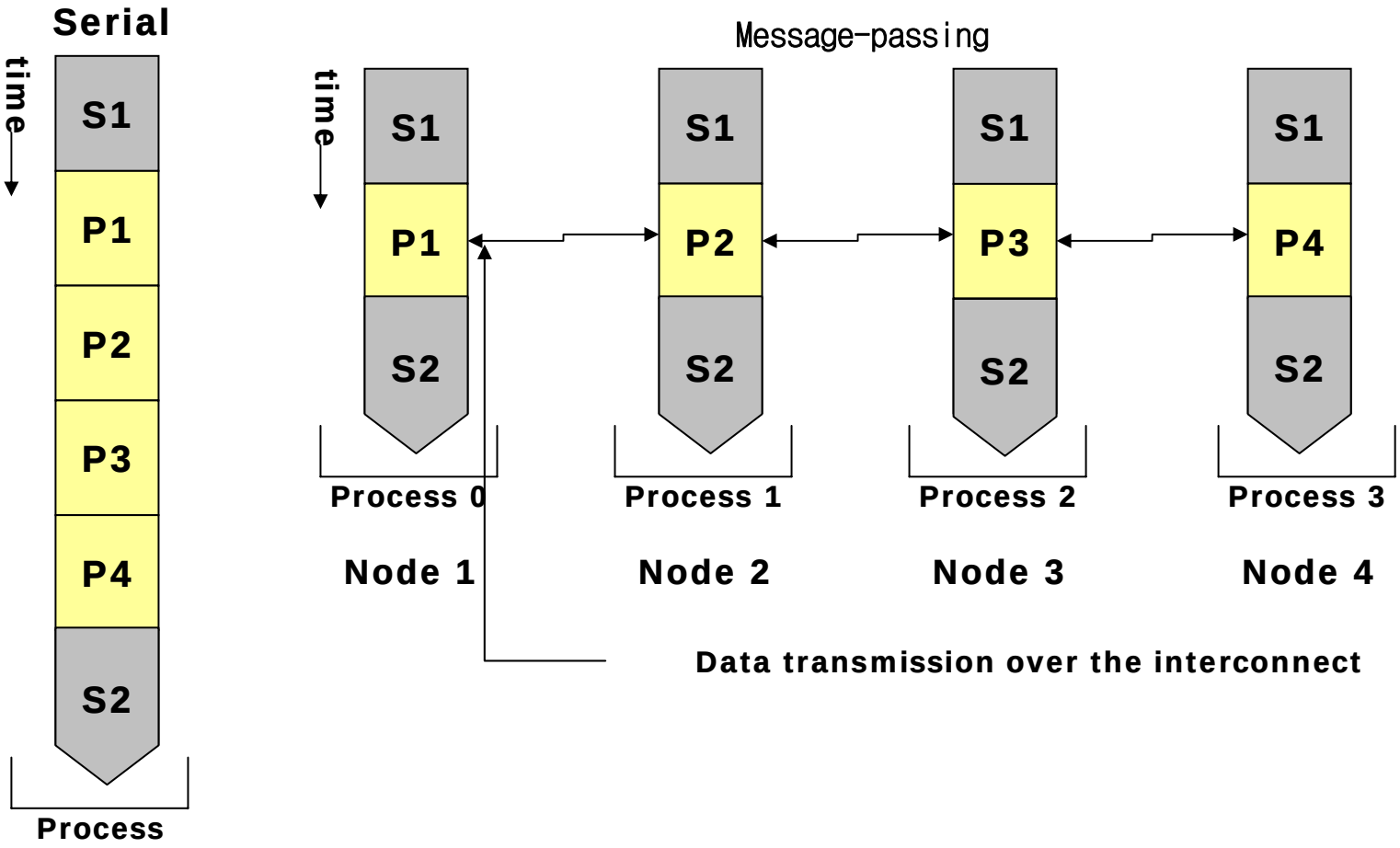
- Message Passing 및 MPI Programming 기본 개념
- MPI Data Type
- Point-to-Point Communication
- Collective Communication
- 병렬 프로그래밍 실제
- QnA

- **Message Passing 및 MPI Programming 기본 개념**
- MPI Data Type
- Point-to-Point Communication
- Collective Communication
- 병렬 프로그래밍 실제

Message Passing

- 지역적으로 메모리를 따로 가지는 Process들이 데이터를 공유하기 위해 Message(데이터)를 송신, 수신하여 통신하는 방식
  - 병렬화를 위한 작업할당, 데이터 분배, 통신의 운용 등 모든 것을 프로그래머가 담당 : 어렵지만 유용성이 좋음(Very Flexible)
  - 다양한 Hardware Platform에서 구현 가능
    - Distributed Memory Multi-Processor System
    - Shared Memory Multi-Processor System
    - Uni-Processor System
- Message Passing Library
  - MPI, PVM, Shmem

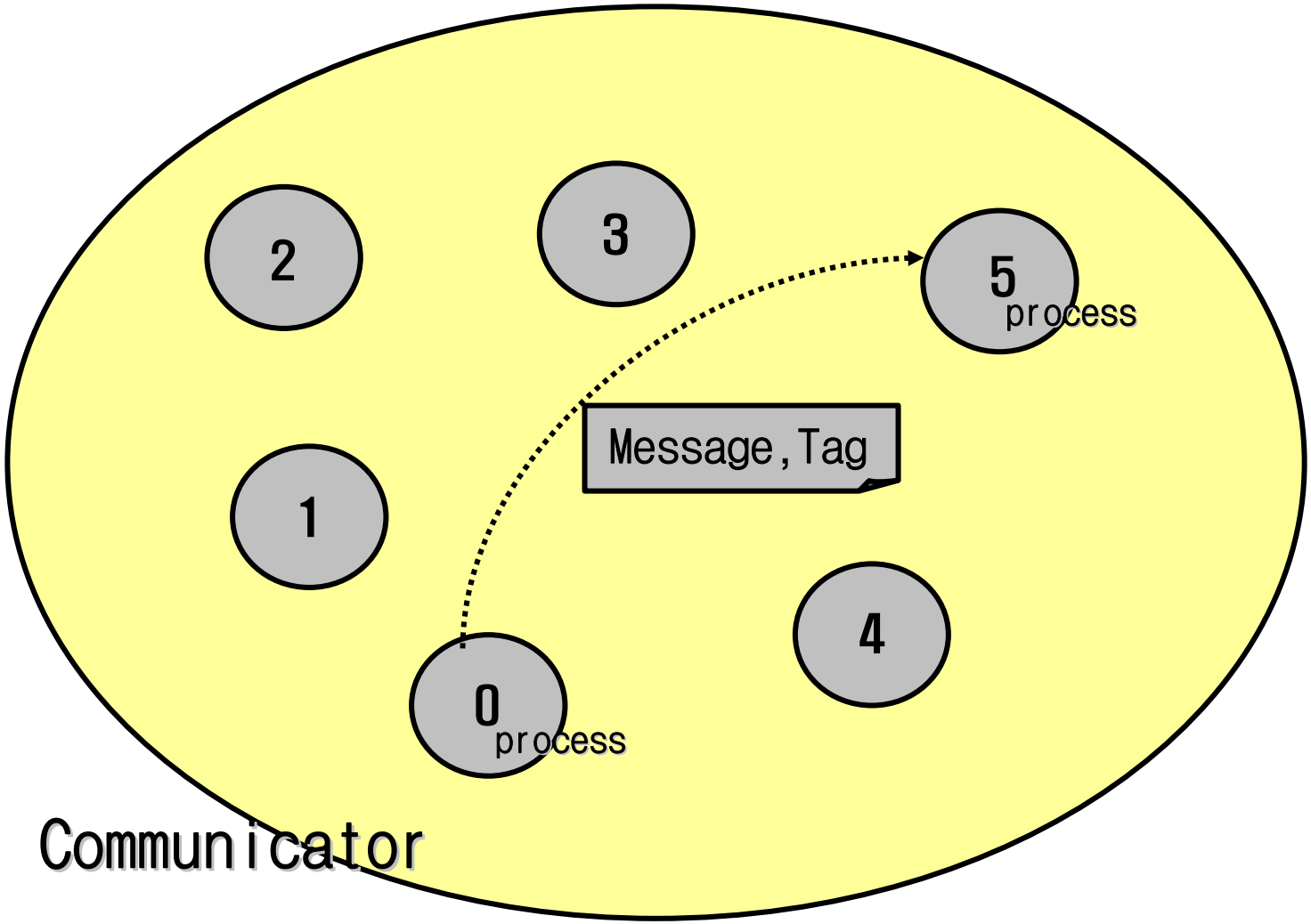
Message Passing



MPI란 무엇인가?

- Message Passing Interface
- Message Passing Parallel Programming을 위해 표준화된 데이터 통신 Library
  - **MPI-1 표준 마련(MPI Forum) : 1994년**
    - **http://www.mcs.anl.gov/mpi/index.html**
  - **MPI-2 발표 : 1997년**
    - **http://www.mpi-forum.org/docs/docs.html**

MPI의 기본 개념



MPI의 기본 개념 - Continued

- Point-to-Point Communication
  - 두 개 Process 사이의 통신
  - 하나의 송신 Process에 하나의 수신 Process가 대응

- Collective Communication
  - 동시에 여러 개의 Process가 통신에 참여
  - 일대다, 다대일, 다대다 대응 가능
  - 여러 번의 Point-to-Point Communication 사용을 하나의 Collective Communication으로 대체
    - 오류의 가능성이 적다.
    - 최적화 되어 일반적으로 빠르다.

MPI 프로그램의 기본 구조

include MPI header file
variable declarations
initialize the MPI environment

... do computation and MPI communication calls ...

close MPI environment

MPI head file

- Head file 삽입

Fortran	C
`INCLUDE 'mpif.h'`	`#include "mpi.h"`

  - MPI 서브루틴과 함수의 Prototype 선언
  - Macro, MPI 관련 인수, Data Type 정의
  - Path in PE(Parallel Environment) of IBM System
    - /usr/lpp/ppe.poe/include/

MPI 초기화

Fortran	C
`CALL MPI_INIT(ierr)`	`int MPI_Init(&argc, &argv)`

- MPI 환경 초기화
- MPI 루틴 중 가장 먼저 오직 한 번 반드시 호출되어야 함

Communicator

- 서로 통신할 수 있는 Process들의 집합
- 모든 MPI 통신 루틴에는 Communicator 인수가 포함 됨
- Communicator를 공유하는 Process들끼리 통신 가능
- MPI_COMM_WORLD
  - 프로그램 실행시 정해진, 사용 가능한 모든 Process를 포함하는 기본적인 Communicator
  - MPI_Init이 호출될 때 정의 됨

Communicator

- Process Rank
  - 동일한 Communicator에 속한 Process의 식별 번호
    - Process가 n개 있으면 0부터 n-1까지 Rank 번호를 할당
  - Message의 송신자와 수신자를 나타내기 위해 사용
  - Process Rank 가져오기

Fortran	`CALL MPI_COMM_RANK(comm, rank, ierr)`
C	`int MPI_Comm_rank(MPI_Comm comm, int *rank)`

Communicator comm에서 이 루틴을 호출한
Process의 Rank를 인수 rank를 이용해 출력

Communicator

- Size of Communicator
  - Communicator에 포함된 Process들의 총 개수
  - Communicator Size 가져오기

Fortran	`CALL MPI_COMM_SIZE(comm, size, ierr)`
C	`int MPI_Comm_size(MPI_Comm comm, int *size)`

MPI 종료

Fortran	C
`CALL MPI_FINALIZE(ierr)`	`int MPI_Finalize();`

- 모든 MPI 자료구조 정리
- 모든 Process들에서 마지막으로 한 번 호출되어야 함
- Process를 종료 시키는 것은 아님

skeleton.f

```
PROGRAM skeleton
INCLUDE 'mpif.h'
INTEGER ierr, rank, size
CALL MPI_INIT(ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)

! … your code here …

CALL MPI_FINALIZE(ierr)
END
```

skeleton.c

```
/* program skeleton*/
#include “mpi.h”
void 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);

    /* … your code here … */

    MPI_Finalize();
}
```

- Message Passing 및 MPI Programming 기본 개념
- **MPI Data Type**
- Point-to-Point Communication
- Collective Communication
- 병렬 프로그래밍 실제

MPI 기본 데이터 타입 (1/2)

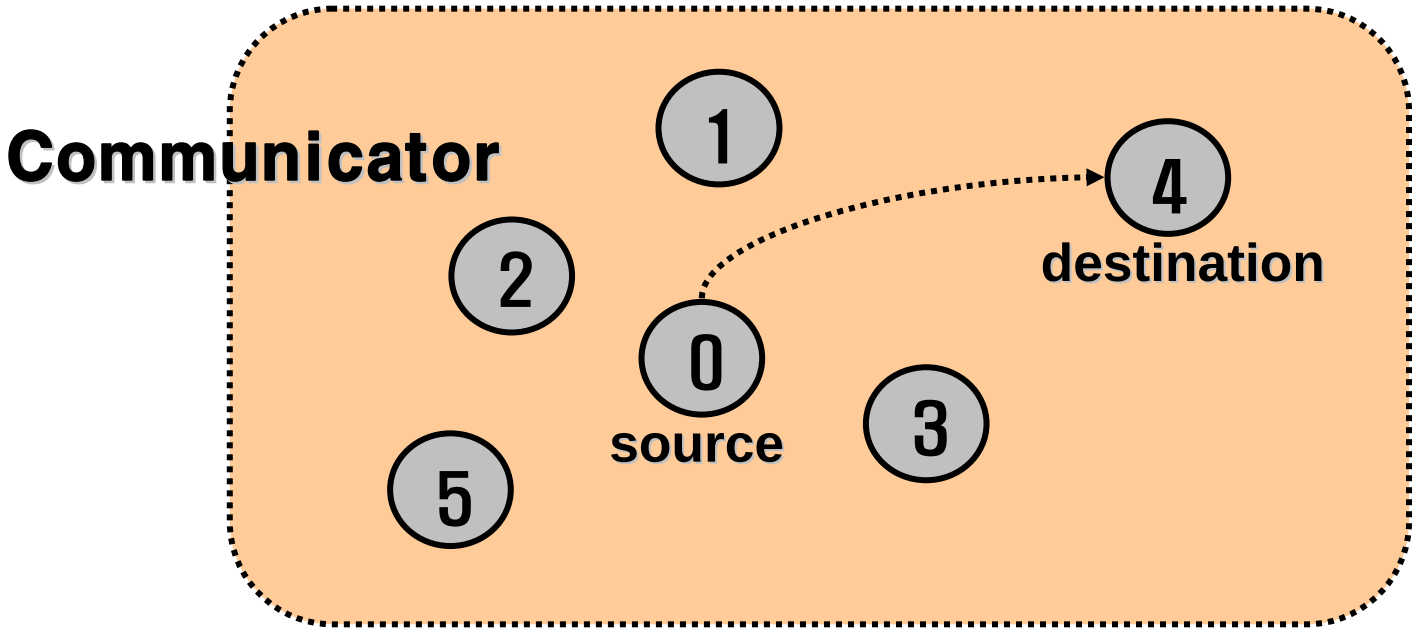
MPI Data Type	Fortran Data Type
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)

MPI 기본 데이터 타입 (2/2)

MPI Data Type	C Data Type
MPI_CHAR	signed char
MPI_SHORT	signed short int
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

- Message Passing 및 MPI Programming 기본 개념
- MPI Data Type
- **Point-to-Point Communication**
- Collective Communication
- 병렬 프로그래밍 실제

Point-to-Point Communication



- 반드시 두 개의 Process만 참여하는 통신
- 통신은 Communicator 내에서만 이루어 진다.
- 송신/수신 Process의 확인을 위해 Communicator와 Rank 사용
- Blocking Communication and Non-blocking Communication
  - Blocking - 통신이 완료된 후 루틴으로부터 리턴 됨
  - Non-Blocking - 통신이 시작되면 완료와 상관없이 리턴, 이후 완료 여부를 검사

Blocking Send

C	`int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
Fortran	`MPI_SEND(buf, count, datatype, dest, tag, comm, ierr)`

(CHOICE) buf : 송신 버퍼의 시작 주소 (IN)

INTEGER count : 송신될 원소 개수 (IN)

INTEGER datatype : 각 원소의 MPI Data Type (IN)

INTEGER dest : 수신 Process의 Rank (IN)

통신이 불필요하면 MPI_PROC_NULL

INTEGER tag : Message tag (IN)

INTEGER comm : MPI Communicator (IN)

```
MPI_SEND(a, 50, MPI_REAL, 5, 1, MPI_COMM_WORLD, ierr)
```

Blocking Receive

C	`int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
Fortran	`MPI_RECV(buf, count, datatype, source, tag, comm, status, ierr)`

(CHOICE) buf : 수신 버퍼의 시작 주소 (OUT)

INTEGER count : 수신될 원소 개수 (IN)

INTEGER datatype : 각 원소의 MPI Data Type (IN)

INTEGER source : 송신 Process의 Rank (IN)

통신이 불필요하면 MPI_PROC_NULL

INTEGER tag : Message Tag (IN)

INTEGER comm : MPI Communicator (IN)

INTEGER status(MPI_STATUS_SIZE) : 수신된 메시지의 정보 저장 (OUT)

```
MPI_RECV(a,50,MPI_REAL,0,1,MPI_COMM_WORLD,status,ierr)
```

성공적인 통신을 위해 주의할 점들

- 송신측에서 수신자 Rank를 명확히 할 것
- 수신측에서 송신자 Rank를 명확히 할 것
- Communicator가 동일 할 것
- Message Tag가 일치할 것
- 수신버퍼는 충분히 클 것

Non-Blocking Communication

- 통신을 세 가지 상태로 분류
  1. **Initialization : 송신 또는 수신의 포스팅**
  2. **전송 데이터를 사용하지 않는 다른 작업 수행**
     - 통신과 계산 작업을 동시 수행
  3. **통신 완료 : 대기 또는 검사**

- Deadlock 가능성 제거, 통신 부하 감소

Non-Blocking Communication

C	`int MPI_ISend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)`
Fortran	`MPI_ISEND(buf, count, datatype, dest, tag, comm, request, ierr)`

C	`int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)`
Fortran	`MPI_IRECV(buf, count, datatype, source, tag, comm, request, ierr)`

INTEGER request : 초기화된 통신의 식별에 이용 (OUT)

논블록킹 수신에는 status 인수가 없음

Non-Blocking Communication

- 대기(waiting)
  - 루틴이 호출되면 통신이 완료될 때까지 Process를 Blocking
  - Non-Blocking Communication + Wait = Blocking Communication

C	`int MPI_Wait(MPI_Request *request, MPI_Status *status)`
Fortran	`MPI_WAIT(request, status, ierr)`

INTEGER request : 포스팅된 통신의 식별에 이용 (IN)

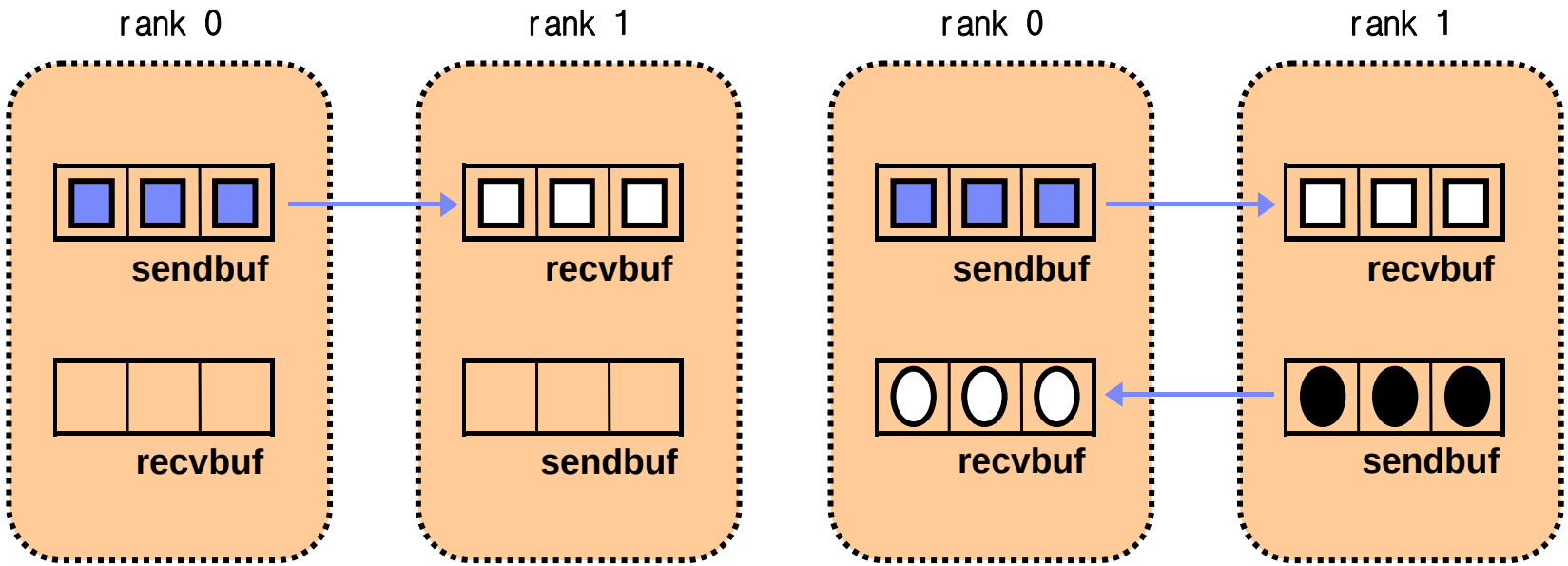
INTEGER status(MPI_STATUS_SIZE) : 수신 메시지에 대한 정보
또는 송신 루틴에 대한 에러코드 (OUT)

Non-Blocking Communication : Fortran

```
PROGRAM isend
INCLUDE 'mpif.h'
INTEGER err, rank, count, req
REAL data(100), value(100)
INTEGER status(MPI_STATUS_SIZE)
CALL MPI_INIT(err)
CALL MPI_COMM_RANK(MPI_COMM_WORLD,rank,err)
IF (rank.eq.1) THEN
    data=3.0
    CALL MPI_ISEND(data,100,MPI_REAL,0,55,MPI_COMM_WORLD,req,
   err)
    CALL MPI_WAIT(req, status, err)
ELSE IF(rank.eq.0) THEN
    CALL
   MPI_IRECV(value,100,MPI_REAL,1,55,MPI_COMM_WORLD,req,err)
    CALL MPI_WAIT(req, status, err)
    PRINT *, "P:",rank," value(5)=",value(5)
ENDIF
CALL MPI_FINALIZE(err)
END
```

Point-to-Point Communication

- Unidirectional and Bidirectional Communication
- Be careful about Deadlocks for Bidirectional Communication



Bidirectional Communication

- Recommended Coding for Performance and the Avoidance of Deadlocks

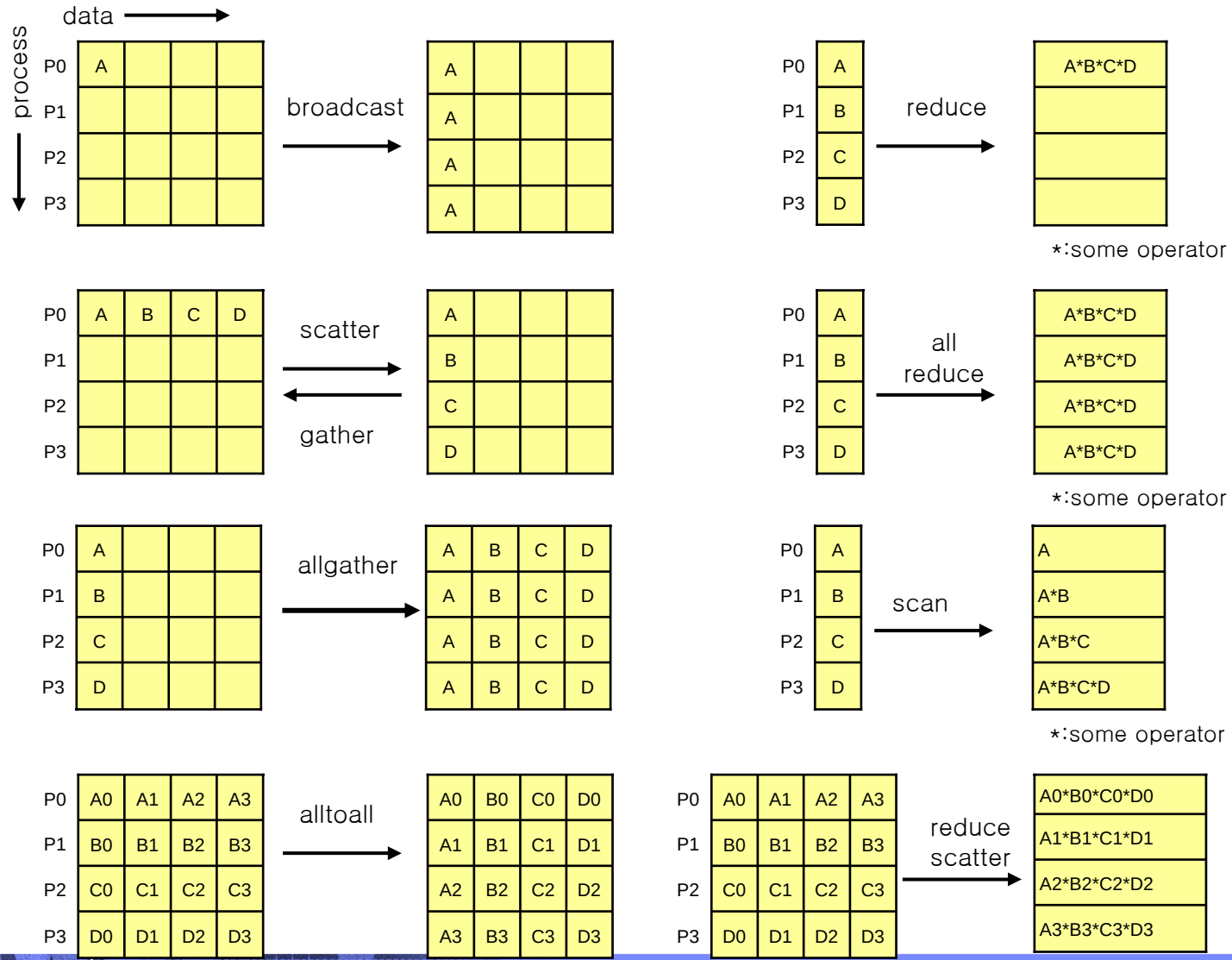
```
IF (myrank==0) THEN
   CALL MPI_ISEND(sendbuf, …, ireq1, …)
   CALL MPI_IRECV(recvbuf, …, ireq2, …)
ELSEIF (myrank==1) THEN
   CALL MPI_ISEND(sendbuf, …, ireq1, …)
   CALL MPI_IRECV(recvbuf, …, ireq2, …)
ENDIF
CALL MPI_WAIT(ireq1, …)
CALL MPI_WAIT(ireq2, …)
```

- Message Passing 및 MPI Programming 기본 개념
- MPI Data Type
- Point-to-Point Communication
- **Collective Communication**
- 병렬 프로그래밍 실제

Collective Communication

- 한 그룹내의 모든 Process가 참여하는 통신
- Based on Point-to-Point Communication
- Point-to-Point Communication을 이용한 구현보다 편리하고 성능면에서 유리
- Collective Communication 루틴
  - Communicator 내의 모든 Process에서 호출
  - Non-Blocking 루틴 없음
  - Tag 없음

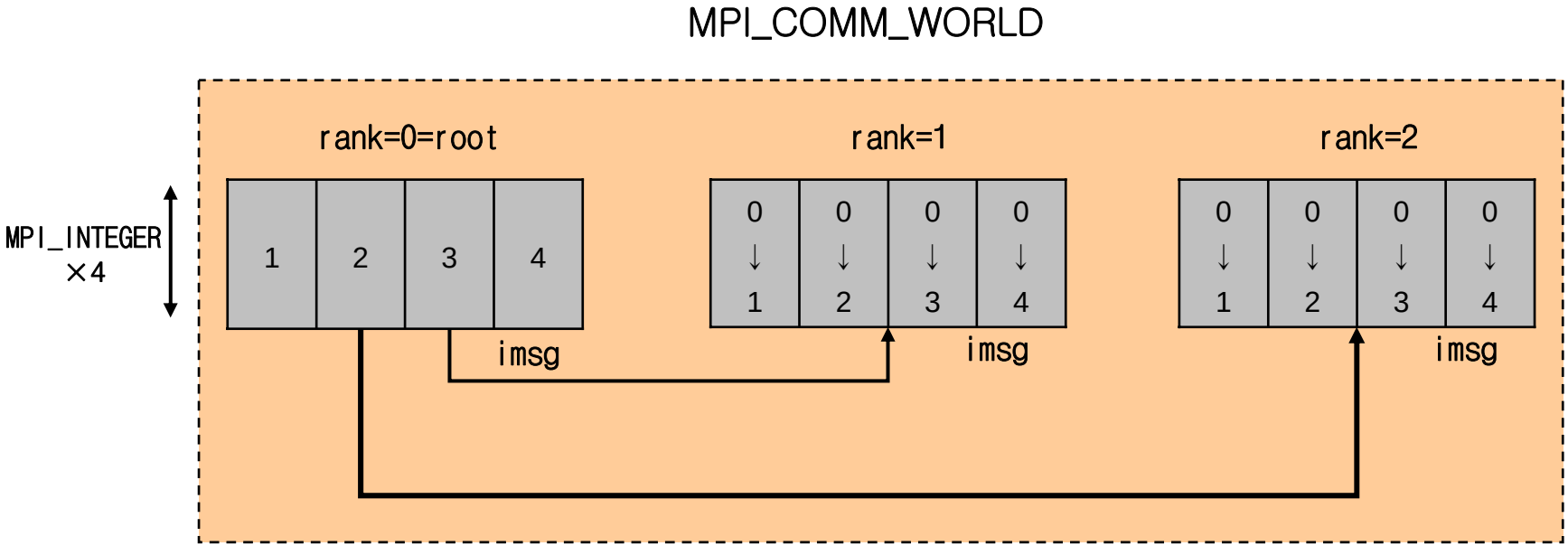
Collective Communication



MPI_BCAST

C	`int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)`
Fortran	`MPI_BCAST(buffer, count, datatype, root, comm, ierr)`

- Root Process로부터 Communicator 내의 다른 Process로 동일한 데이터를 전송 : 일대다 통신



MPI_BCAST 예제 : Fortran

```
PROGRAM bcast
INCLUDE ‘mpif.h’
INTEGER imsg(4)
CALL MPI_INIT(ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
IF (myrank==0) THEN
    DO i=1,4
        imsg(i) = i
    ENDDO
ELSE
    DO i=1,4
        imsg(i) = 0
    ENDDO
ENDIF
PRINT*,‘Before:’,imsg
CALL MPI_BCAST(imsg, 4, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
PRINT*,‘After :’,imsg
CALL MPI_FINALIZE(ierr)
END
```

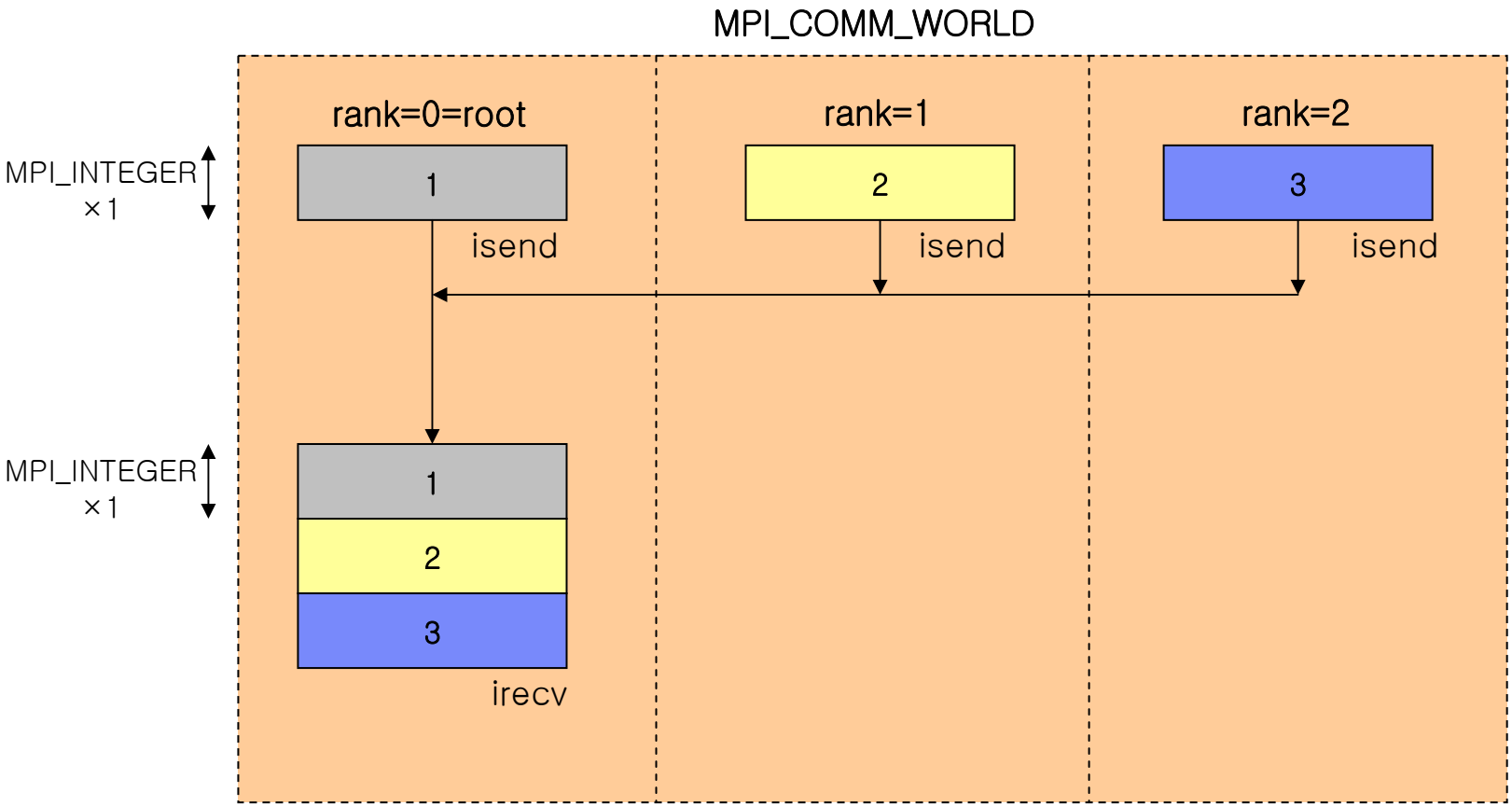
MPI_GATHER

C	`int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`
Fortran	`MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, ierr)`

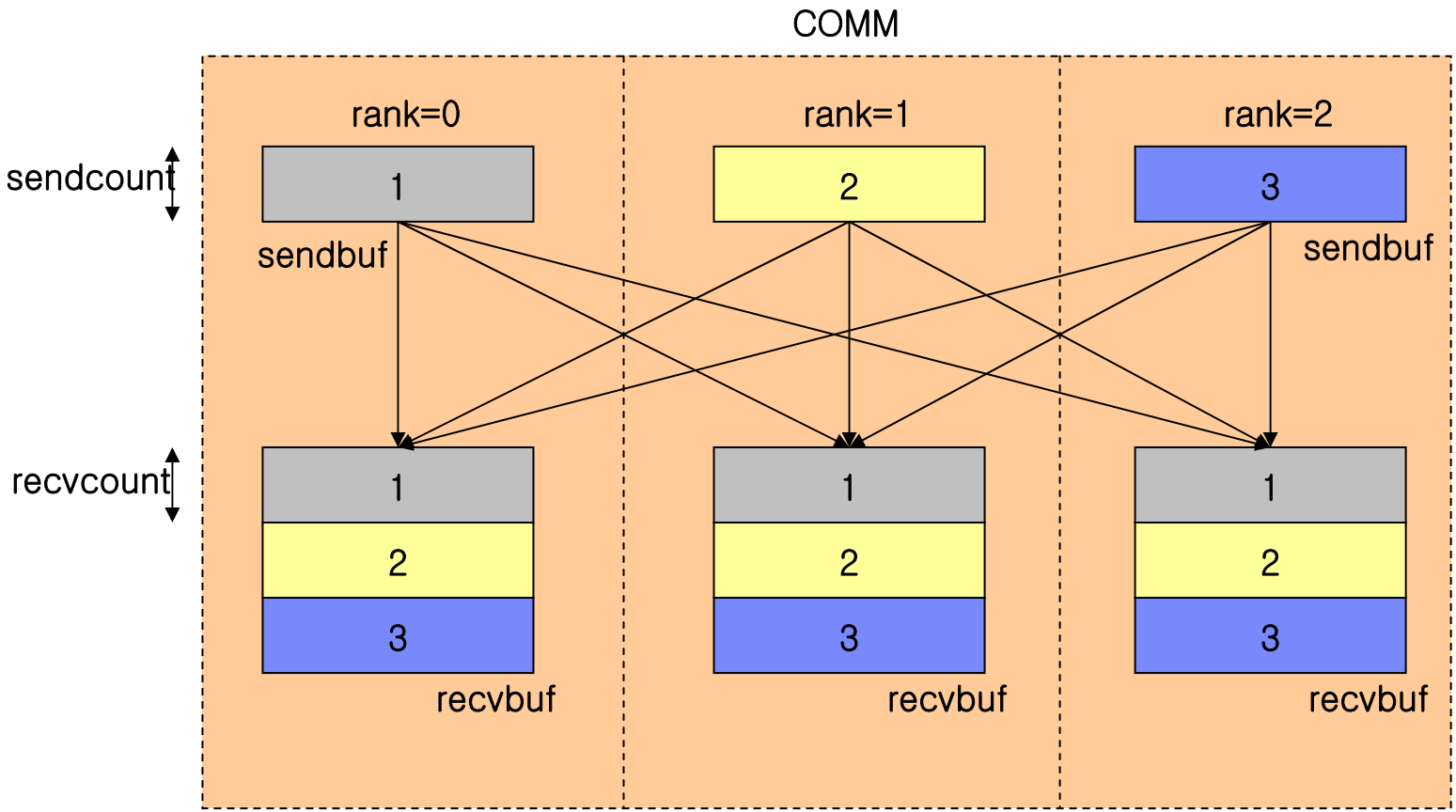
- 모든 프로세스(루트 포함)가 송신한 데이터를 취합하여 랭크 순서대로 저장 : 다대일 통신
- 송신 버퍼(sendbuf)와 수신 버퍼(recvbuf)의 메모리 위치가 겹쳐지지 않도록 주의할 것. 즉, 같은 이름을 쓰면 안됨

  ➔ 송신 버퍼와 수신 버퍼를 이용하는 모든 집합 통신에 해당
- 전송되는 데이터의 크기는 모두 동일할 것
- 크기가 서로 다른 데이터의 취합 ➔ MPI_GATHERV

MPI_GATHER



MPI_ALLGATHER : MPI_GATHER + MPI_BCAST



MPI_REDUCE

C	`int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)`
Fortran	`MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm, ierr)`

(CHOICE) sendbuf : 송신 버퍼의 시작 주소 (IN)
(CHOICE) recvbuf : 수신 버퍼의 주소 (OUT)
INTEGER count : 송신 버퍼의 원소 개수 (IN)
INTEGER datatype : 송신 버퍼 원소의 MPI 데이터 타입 (IN)
INTEGER op : 환산 연산자 (IN)
INTEGER root : 루트 프로세스의 랭크 (IN)
INTEGER comm : 커뮤니케이터 (IN)

- 각 프로세스로부터 데이터를 모아 하나의 값으로 환산,그 결과를 루트 프로세스에 저장

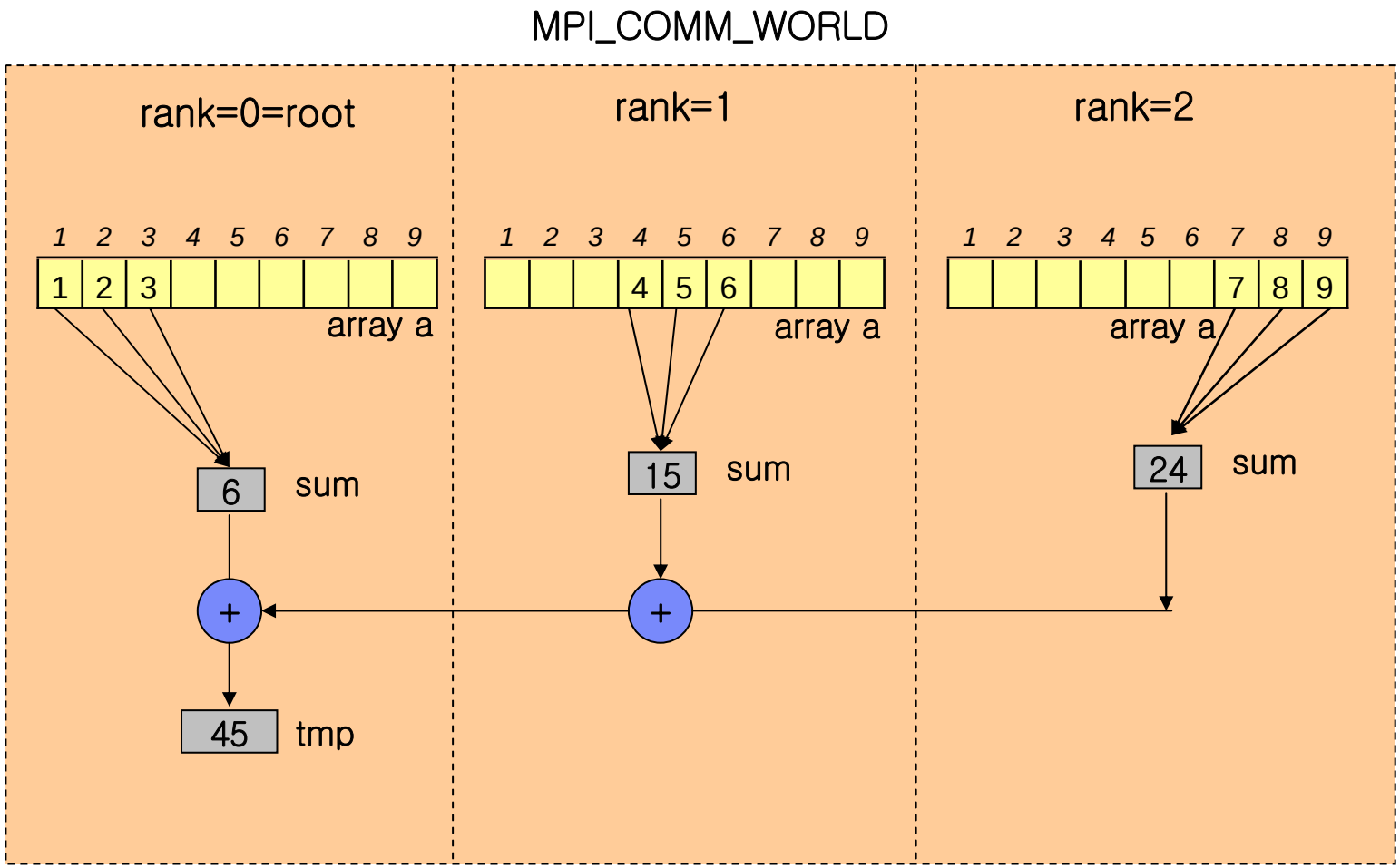
MPI_REDUCE : 연산과 Data Type(Fortran)

Operation	Data Type (Fortran)
MPI_SUM(sum), MPI_PROD(product)	MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_COMPLEX
MPI_MAX(maximum), MPI_MIN(minimum)	MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION
MPI_MAXLOC(max value and location), MPI_MINLOC(min value and location)	MPI_2INTEGER, MPI_2REAL, MPI_2DOUBLE_PRECISION
MPI_LAND(logical AND), MPI_LOR(logical OR), MPI_LXOR(logical XOR)	MPI_LOGICAL
MPI_BAND(bitwise AND), MPI_BOR(bitwise OR), MPI_BXOR(bitwise XOR)	MPI_INTEGER, MPI_BYTE

MPI_REDUCE : 연산과 Data Type(C)

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

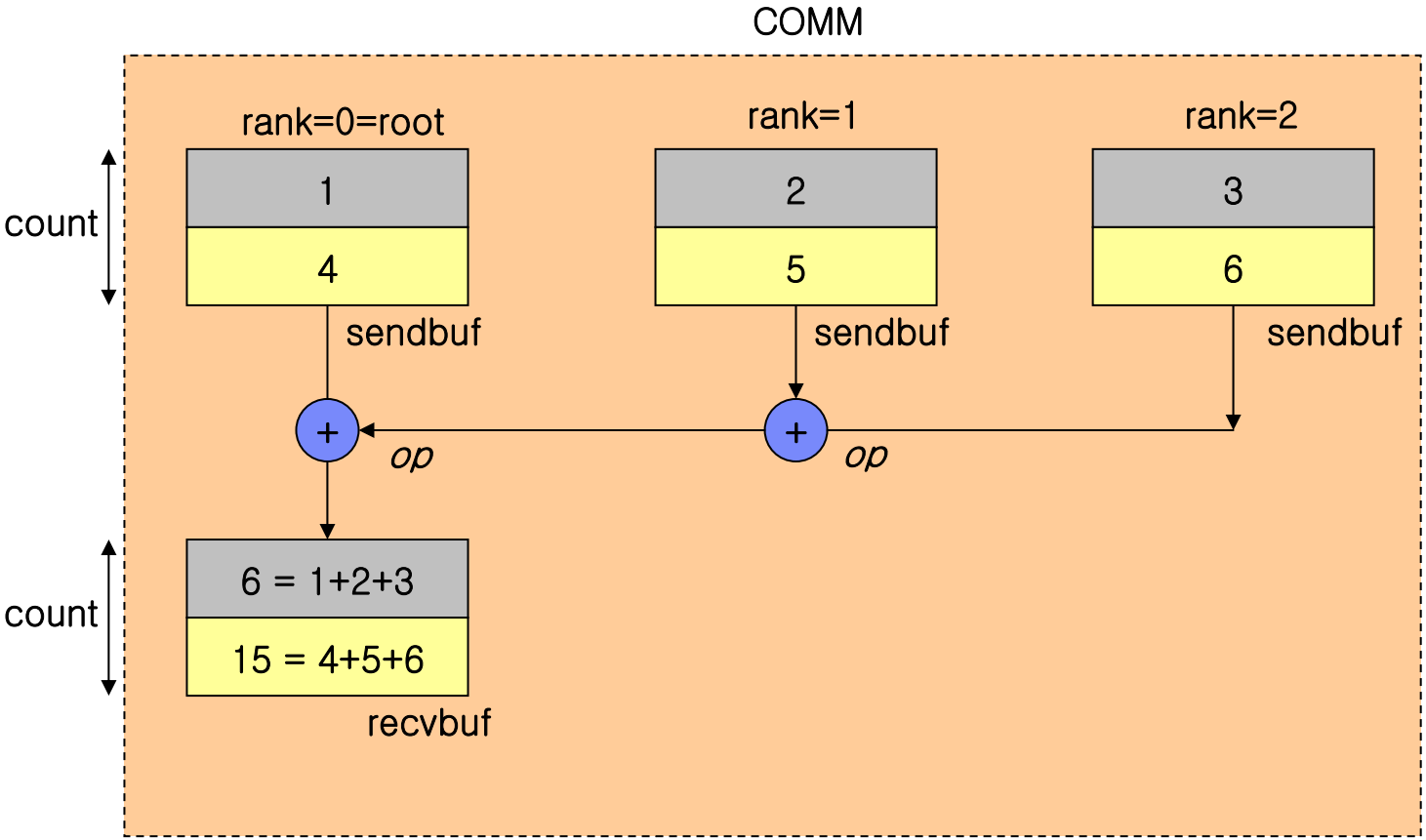
MPI_REDUCE



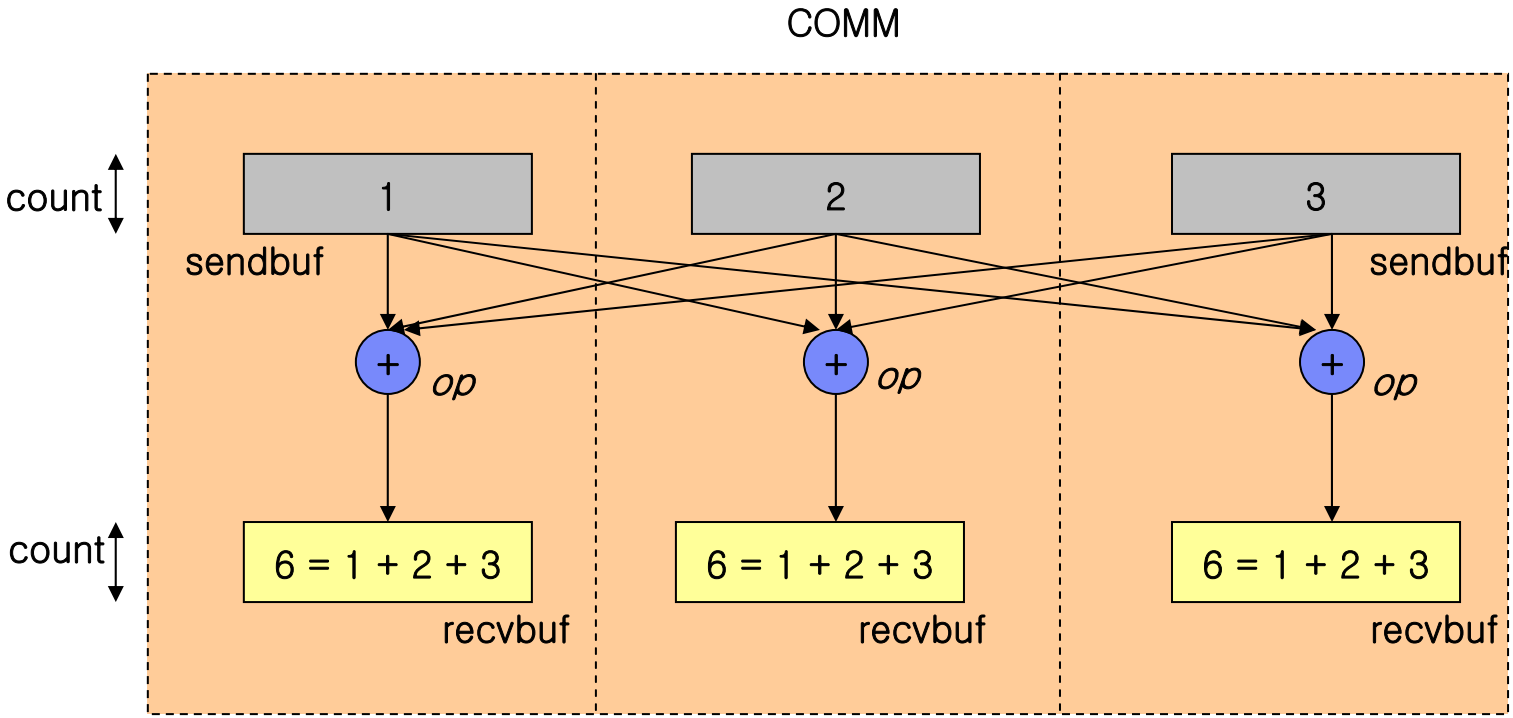
MPI_REDUCE : Fortran

```
PROGRAM reduce
INCLUDE ‘mpif.h’
REAL a(9)
CALL MPI_INIT(ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
ista = myrank * 3 + 1
iend = ista + 2
DO i=ista,iend
   a(i) = i
ENDDO
sum = 0.0
DO i=ista,iend
   sum = sum + a(i)
ENDDO
CALL MPI_REDUCE(sum,tmp,1,MPI_REAL,MPI_SUM,0,MPI_COMM_WORLD,ierr)
sum = tmp
IF (myrank==0) THEN
   PRINT *,’sum =’,sum
ENDIF
CALL MPI_FINALIZE(ierr)
END
```

MPI_REDUCE : Array



MPI_ALLREDUCE



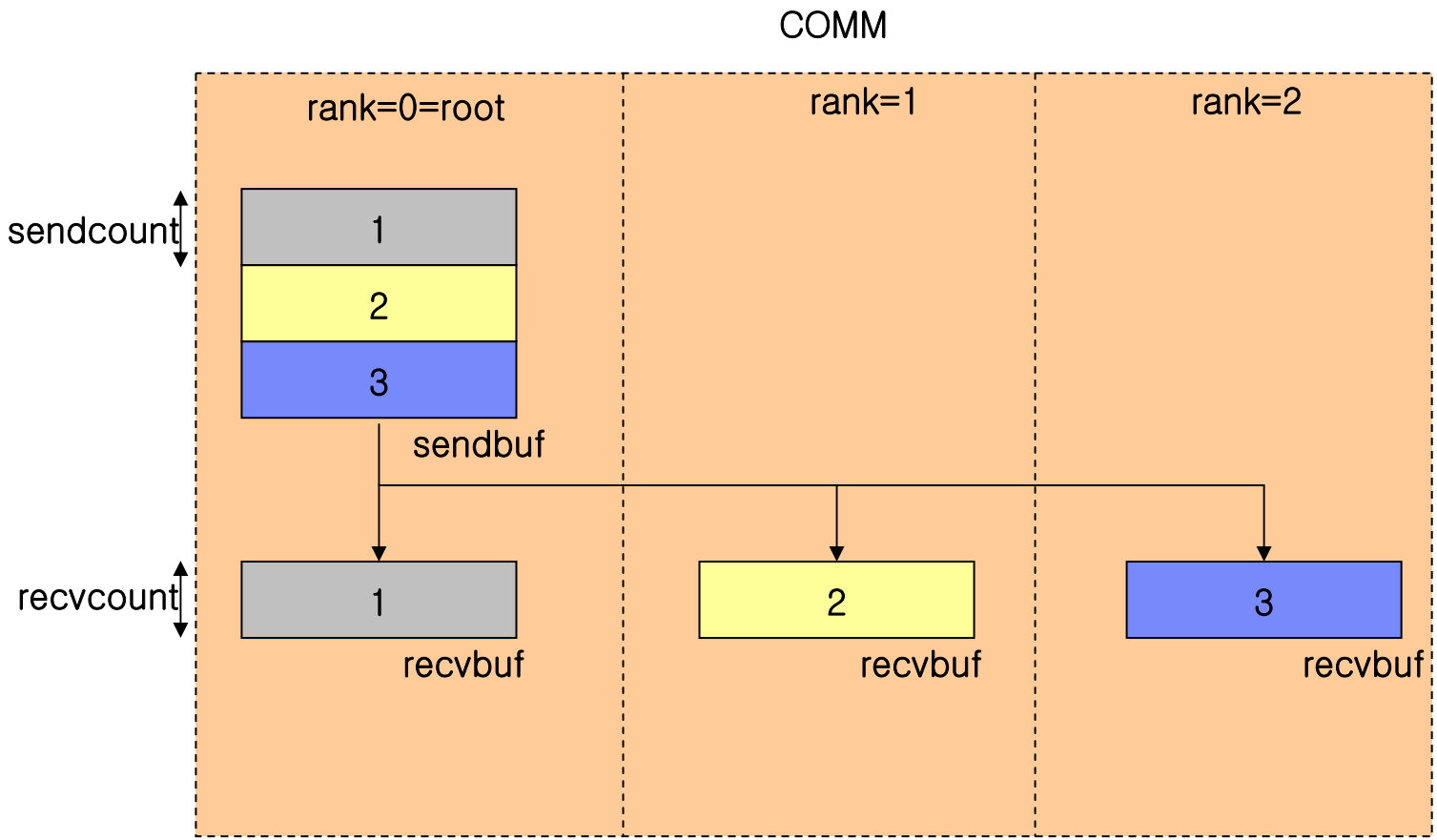
MPI_SCATTER

C	`int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`
Fortran	`MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, ierr)`

(CHOICE) sendbuf : 송신 버퍼의 주소 (IN)
INTEGER sendcount : 각 Process로 보내지는 원소 개수 (IN)
INTEGER sendtype : 송신 버퍼 원소의 MPI Data Type (IN)
(CHOICE) recvbuf : 수신 버퍼의 주소 (OUT)
INTEGER recvcount : 수신 버퍼의 원소 개수 (IN)
INTEGER recvtype : 수신 버퍼의 MPI Data Type (IN)
INTEGER root : 송신 Process의 Rank (IN)
INTEGER comm : Communicator (IN)

- Root Process는 데이터를 같은 크기로 나누어 각 Process에 Rank 순서대로 하나씩 전송

MPI_SCATTER



MPI_BARRIER

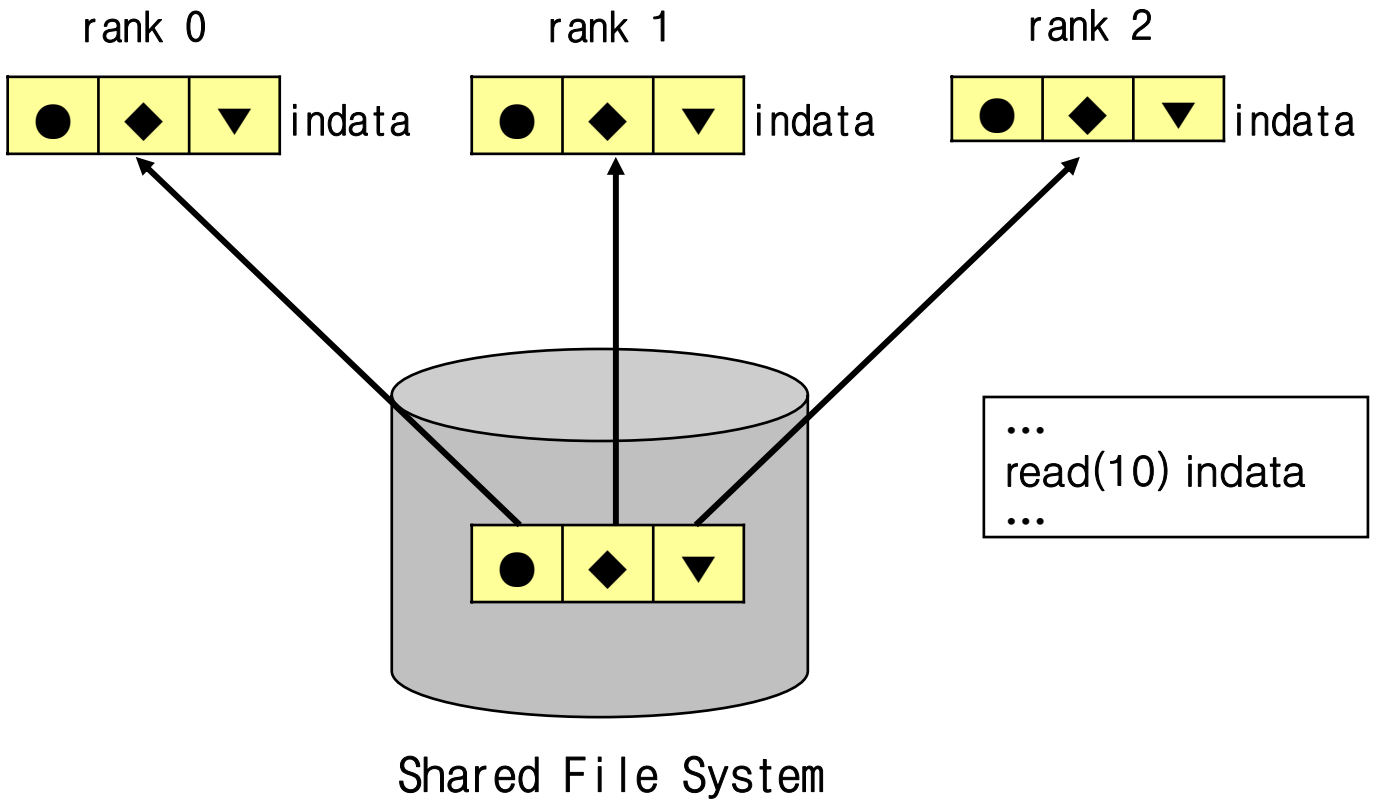
C	`int MPI_Barrier(MPI_Comm comm)`
Fortran	`MPI_BARRIER(comm, ierr)`

- Communicator 내의 모든 Process가 MPI_BARRIER를 호출할 때까지 더 이상의 Process 진행을 막음

- Message Passing 및 MPI Programming 기본 개념
- MPI Data Type
- Point-to-Point Communication
- Collective Communication
- **병렬 프로그래밍 실제**
  - 병렬 프로그램의 입출력
  - DO Loop의 병렬화
  - Block 분할

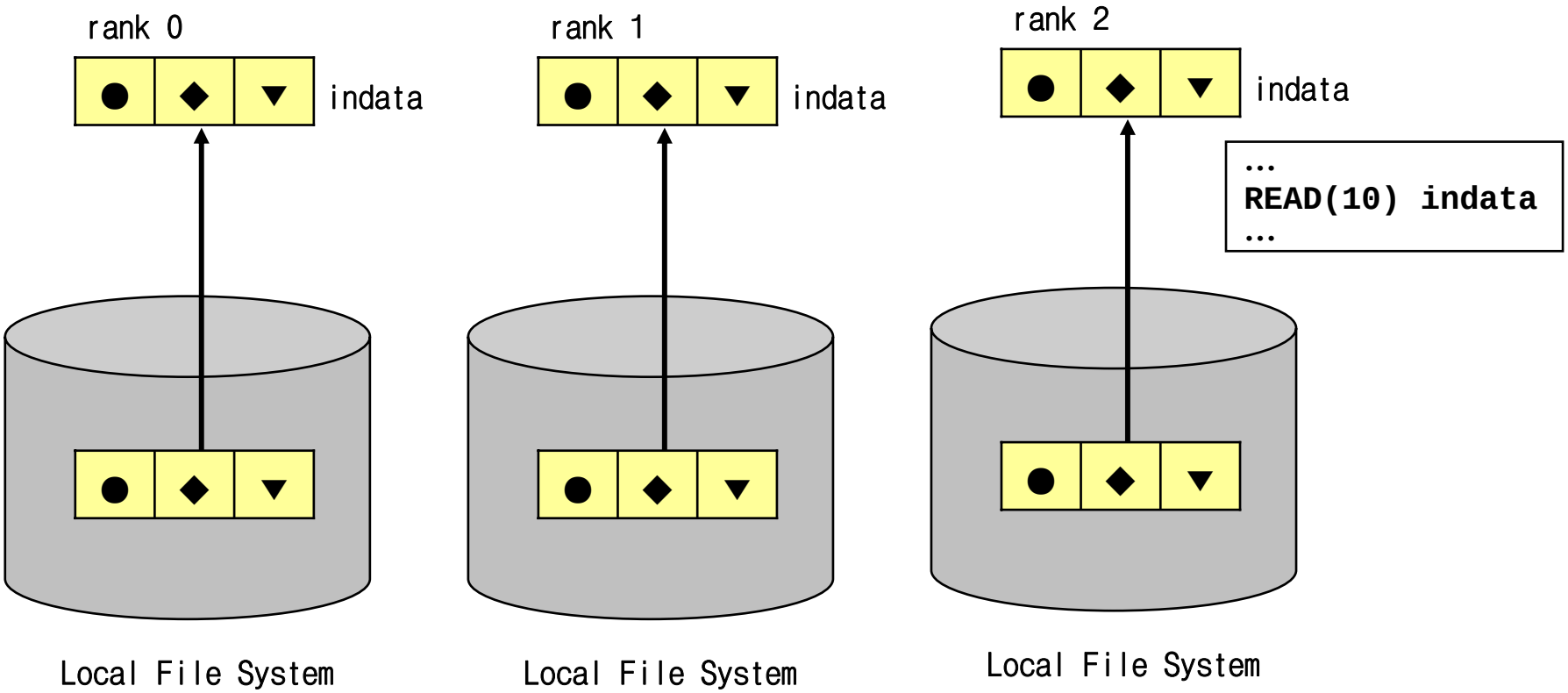
병렬 프로그램의 입력 (1/4)

- Shared Filesystem으로부터 동시에 읽어오기



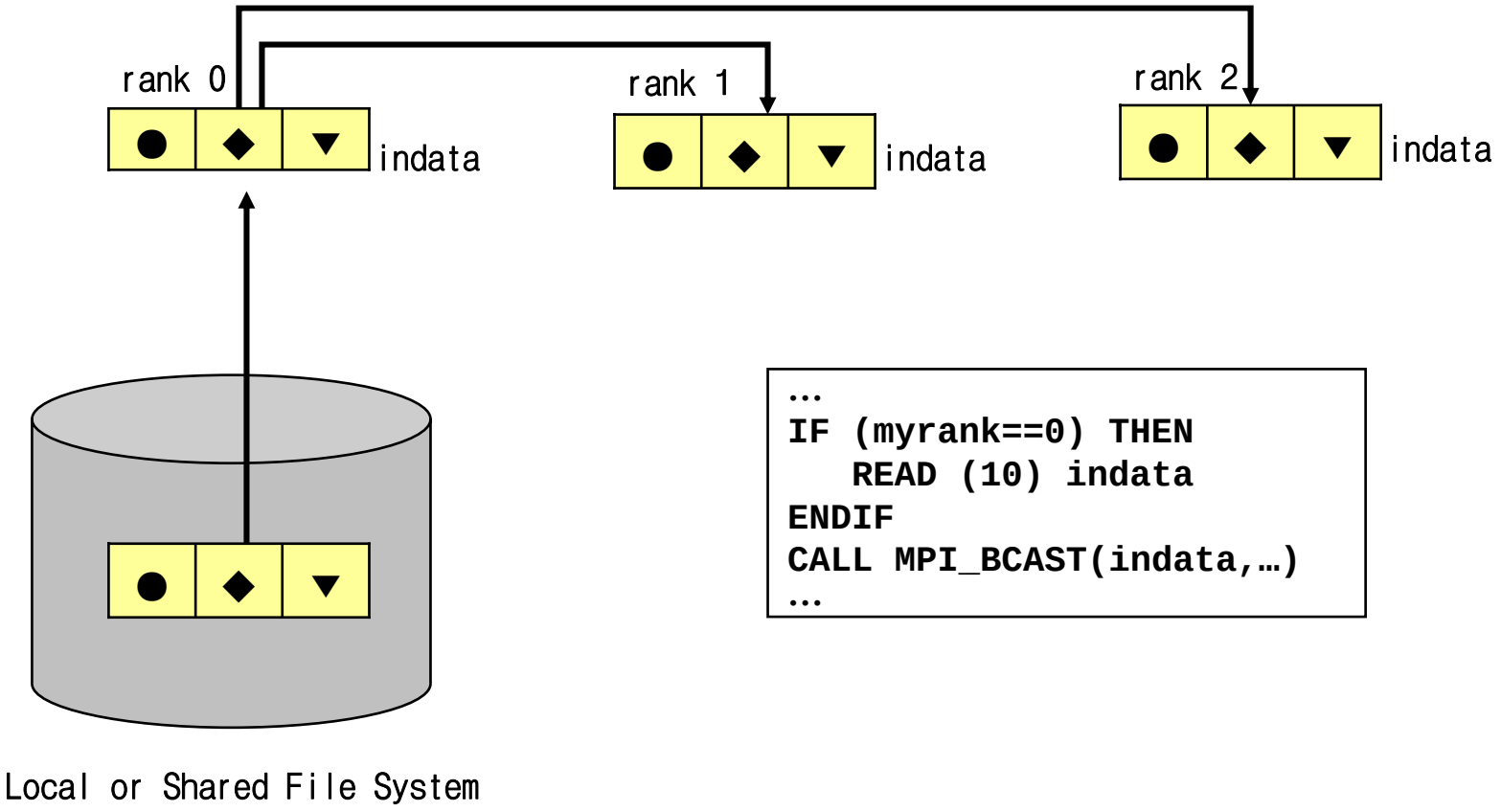
병렬 프로그램의 입력 (2/4)

- Local Filesystem에 입력파일의 복사본을 각각 따로 가지는 경우



병렬 프로그램의 입력 (3/4)

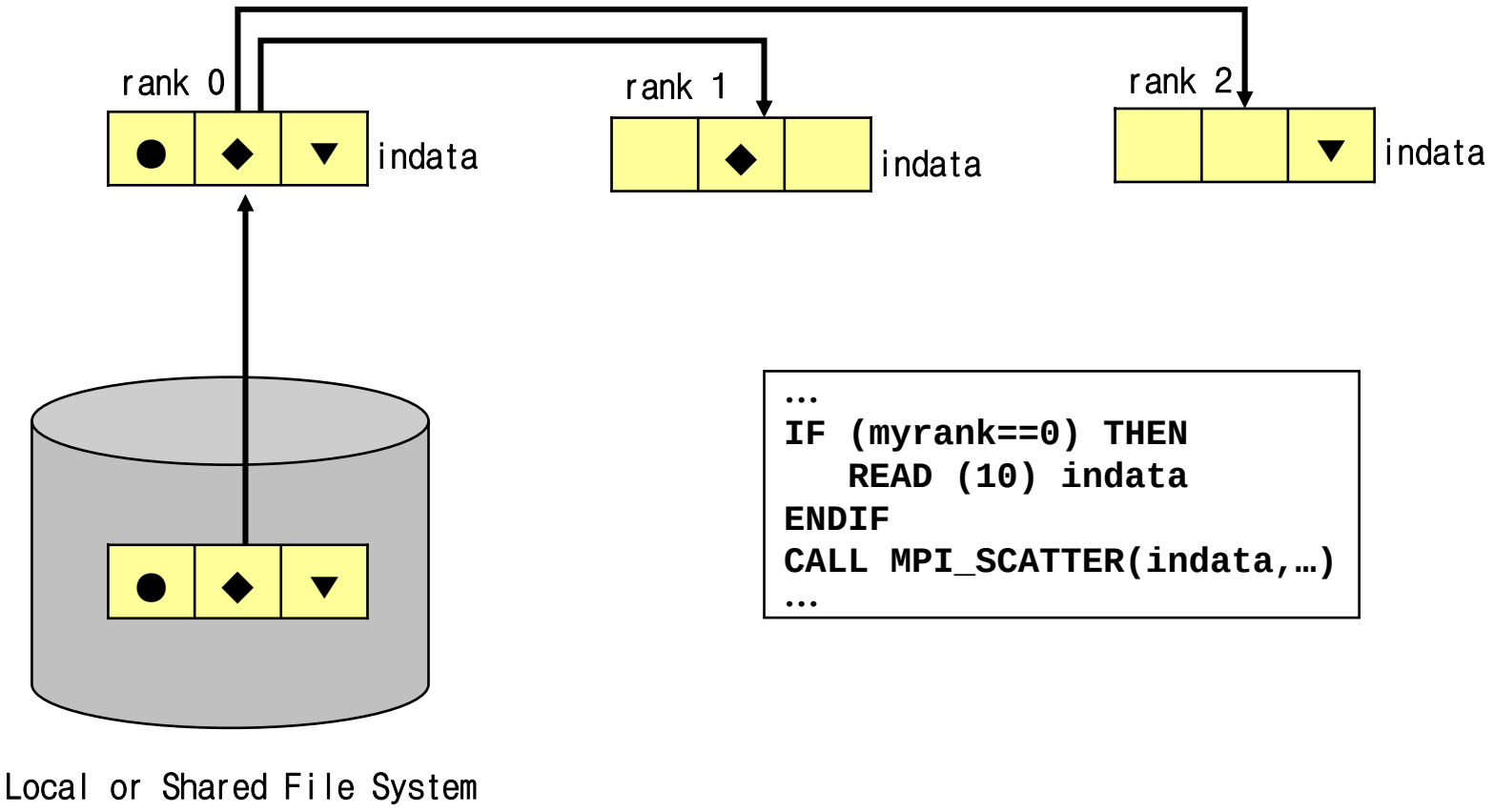
- 한 Process가 입력파일을 읽어 다른 Process에 전달
- MPI_BCAST




```
...
IF (myrank==0) THEN
   READ (10) indata
ENDIF
CALL MPI_BCAST(indata,…)
...
```

병렬 프로그램의 입력 (4/4)

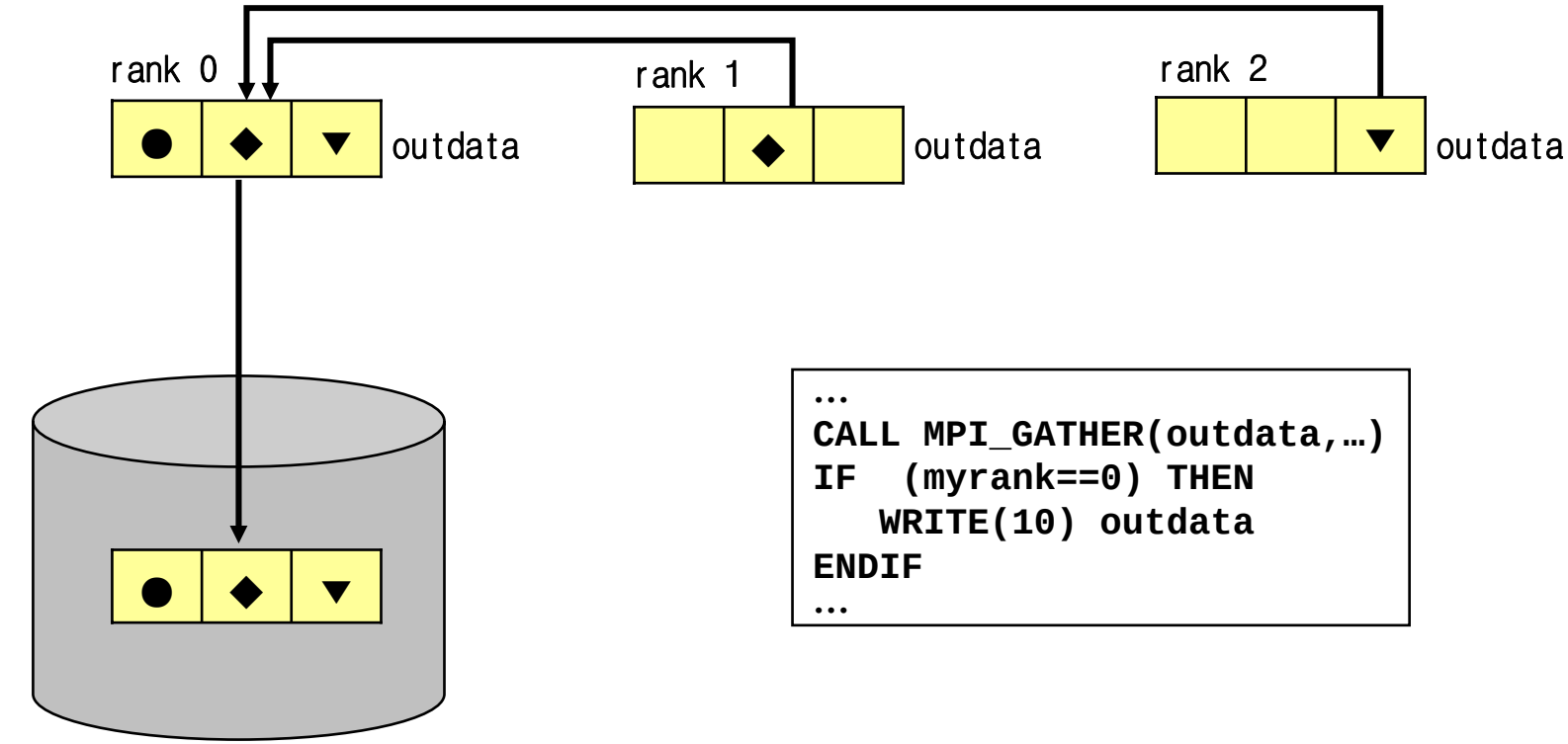
- 한 Process가 입력파일을 읽어 다른 Process에 전달
- MPI_SCATTER




```
...
IF (myrank==0) THEN
   READ (10) indata
ENDIF
CALL MPI_SCATTER(indata,…)
...
```

병렬 프로그램의 출력 (1/2)

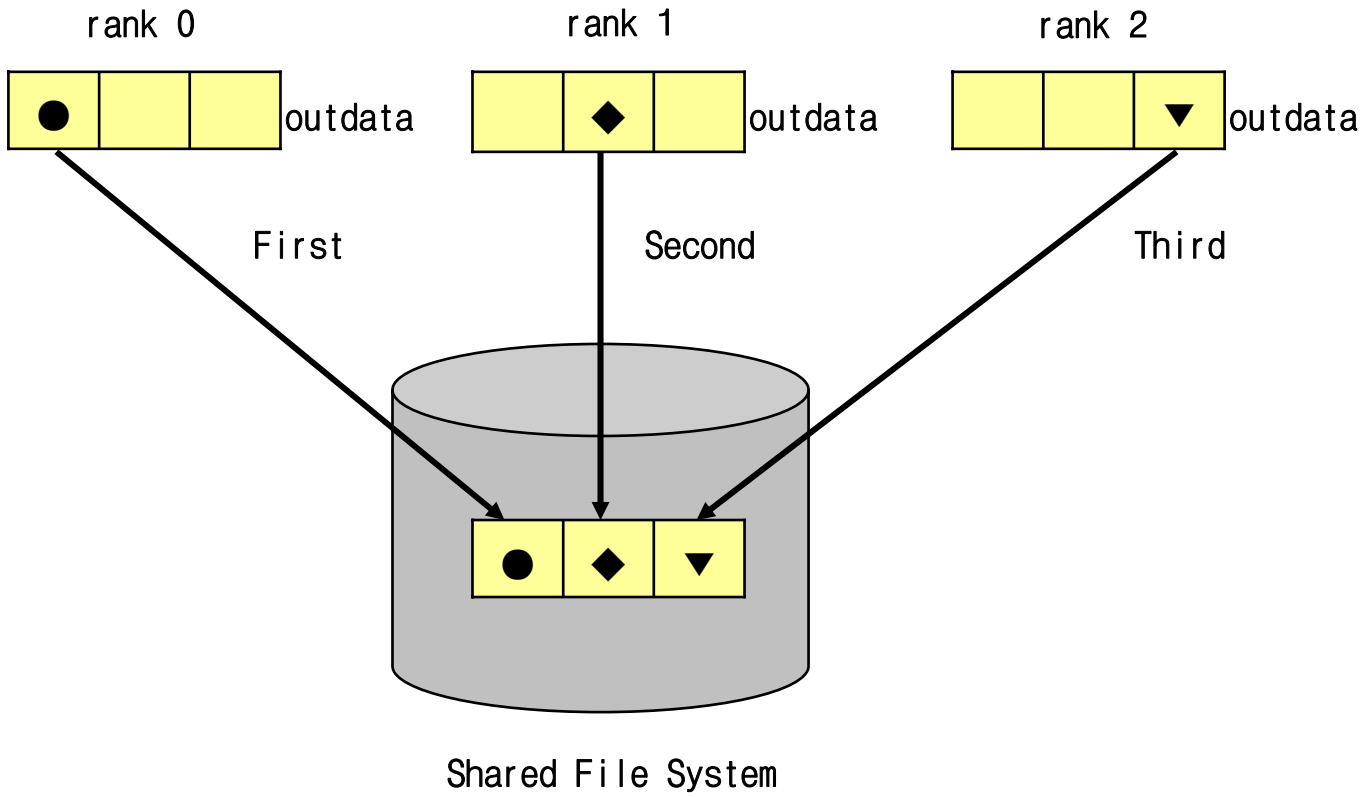
- 한 Process가 데이터를 모아 Local Filesystem에 저장




```
…
CALL MPI_GATHER(outdata,…)
IF  (myrank==0) THEN
   WRITE(10) outdata
ENDIF
…
```

병렬 프로그램의 출력 (2/2)

- 각 Process가 Shared Filesystem에 순차적으로 저장

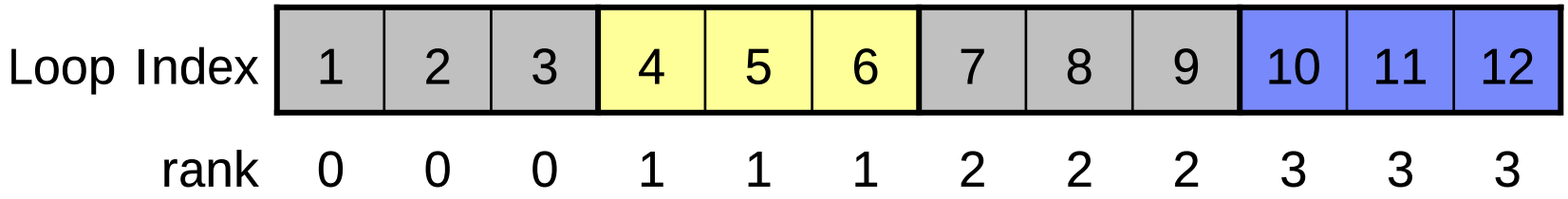


DO Loop의 병렬화

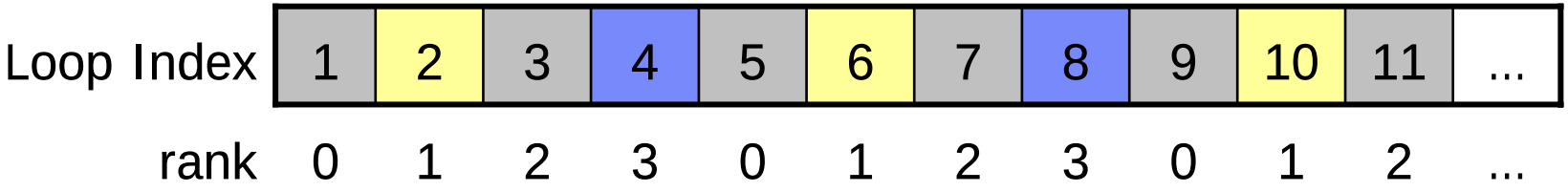
- Loop 내에서 반복되는 계산의 할당 문제는 Loop Index와 관련된 Index를 가지는 Array의 할당 문제가 됨
  - Array를 어떻게 나눌 것인가?
  - 나눠진 Array와 관련 계산을 어떻게 효율적으로 할당할 것인가?

Array의 분할

- Block Distribution



- Cyclic Distribution



Block Distribution

- n = p X q + r
  - n : 반복 계산 회수
  - p : Process 개수
  - q : n을 p로 나눈 몫
  - r : n을 p로 나눈 나머지

- r개 Process에 q+1번, 나머지 Process에 q번의 계산 할당

$$n=r(q+1) + (p-r)q$$

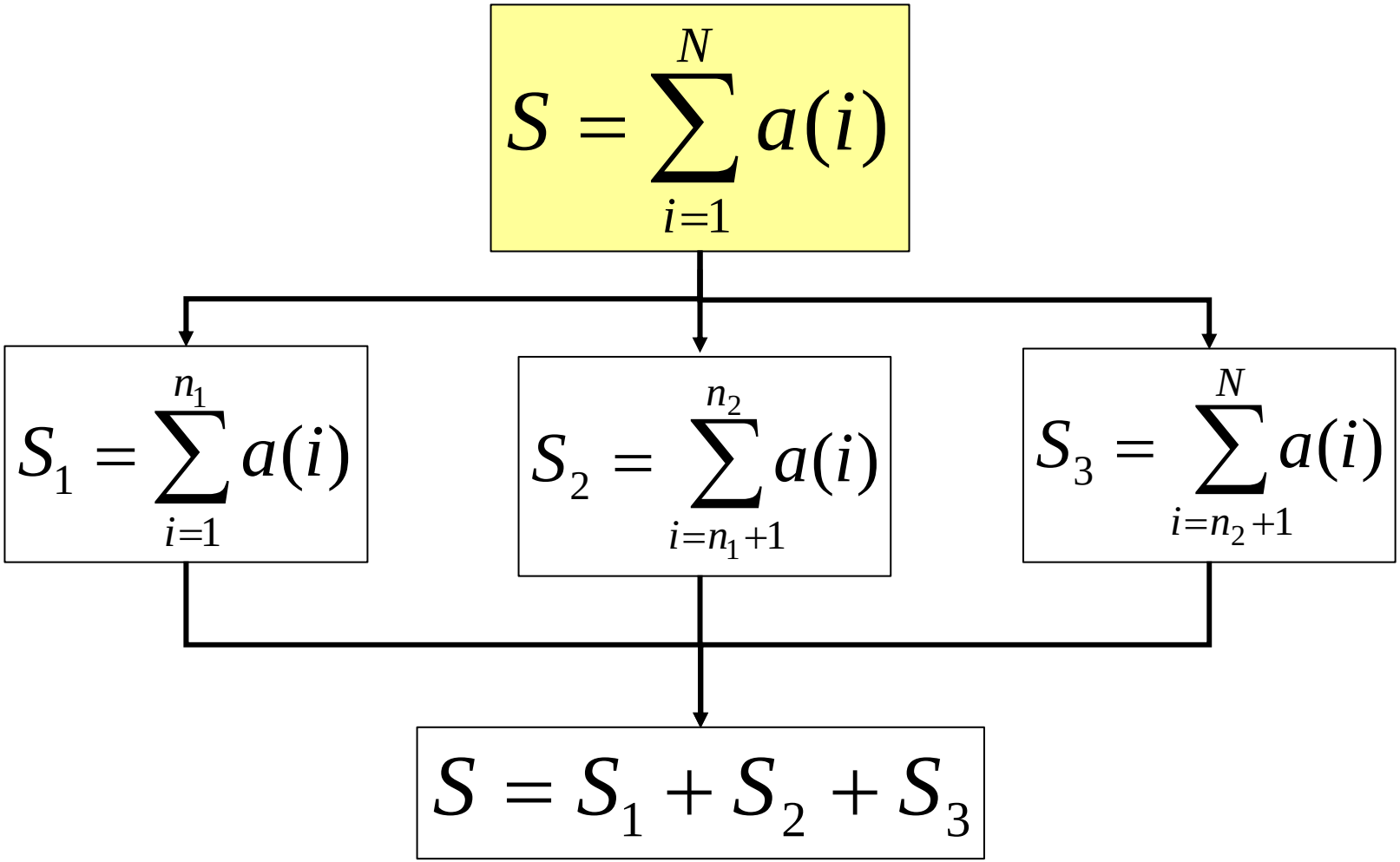
Loop Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14
rank	0	0	0	0	1	1	1	1	2	2	2	3	3	3

Block Distribution

```
SUBROUTINE para_range(n1, n2, nprocs, irank, ista, iend)
  iwork1 = (n2 - n1 + 1) / nprocs
  iwork2 = MOD(n2 - n1 + 1, nprocs)
  ista = irank * iwork1 + n1 + MIN(irank, iwork2)
  iend = ista + iwork1 - 1
  IF (irank < iwork2) iend = iend + 1
END
```

- n1부터 n2까지 반복되는 루프 계산을 nprocs개 Process에 Block 분할을 이용해 할당하는 서브루틴
- Process irank가 ista부터 iend까지 계산을 할당 받음

Block Distribution



Block Distribution : Fortran

```
PROGRAM para_sum
INCLUDE ‘mpif.h’
PARAMETER (n = 100000)
DIMENSION a(n)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
CALL para_range(1, n, nprocs, myrank, ista, iend)
DO i = ista, iend
   a(i) = i
ENDDO
sum = 0.0
DO i = ista, iend
   sum = sum + a(i)
ENDDO
CALL
    MPI_REDUCE(sum,ssum,1,MPI_REAL,MPI_SUM,0,MPI_COMM_WORLD,ierr)
sum = ssum
IF (myrank == 0) PRINT *, ‘sum =’,sum
CALL MPI_FINALIZE(ierr)
END
```

Q & A