

MPI를 이용한 병렬 프로그래밍



KISTI 슈퍼컴퓨팅 센터

목 표

**MPI를 활용해 메시지 패싱 기반의 병렬
프로그램 작성을 가능하도록 한다.**

차 례

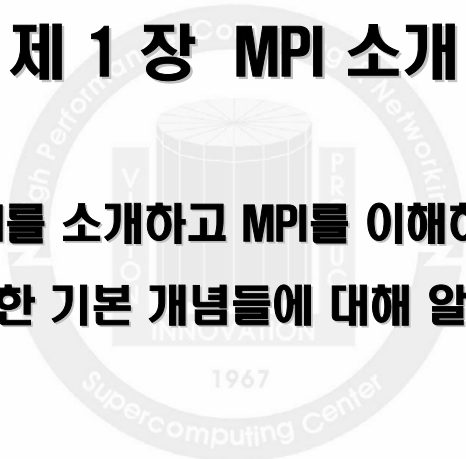
1. MPI 소개
2. MPI를 이용한 병렬 프로그래밍 기초
3. MPI를 이용한 병렬 프로그래밍 실제
4. MPI 병렬 프로그램 예제

부록 : MPI-2

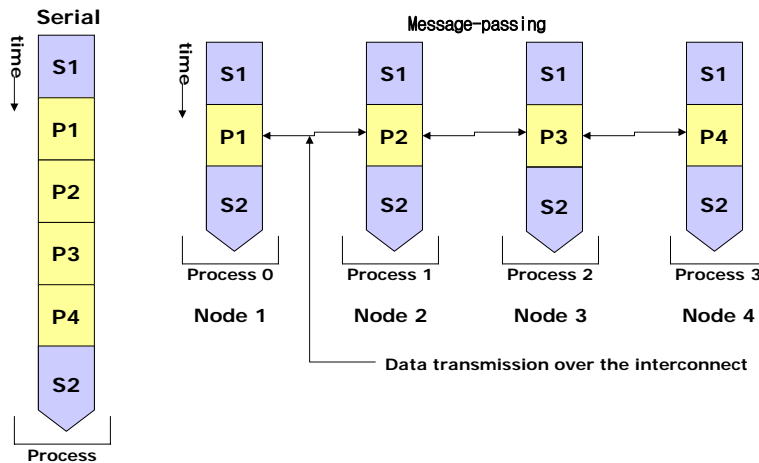
용어정리/참고자료

제 1 장 MPI 소개

MPI를 소개하고 MPI를 이해하는데
필요한 기본 개념들에 대해 알아본다.



메시지 패싱 (1/2)



메시지 패싱 (2/2)

- 지역적으로 메모리를 따로 가지는 프로세스들이 데이터를 공유하기 위해 메시지(데이터)를 송신, 수신하여 통신하는 방식
 - 병렬화를 위한 작업할당, 데이터분배, 통신의 운용 등 모든 것을 프로그래머가 담당 : 어렵지만 유용성 좋음(Very Flexible)
 - 다양한 하드웨어 플랫폼에서 구현 가능
 - 분산 메모리 다중 프로세서 시스템
 - 공유 메모리 다중 프로세서 시스템
 - 단일 프로세서 시스템
- 메시지 패싱 라이브러리
 - MPI, PVM, Shmem

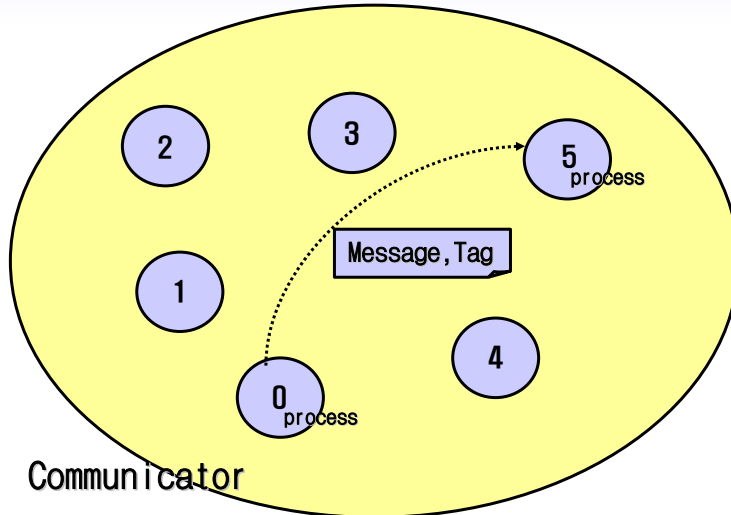
MPI란 무엇인가?

- **Message Passing Interface**
- **메시지 패싱 병렬 프로그래밍을 위해 표준화된 데이터 통신 라이브러리**
 - **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 목표

이식성 (Portability)
효율성 (Efficiency)
기능성 (Functionality)

MPI의 기본 개념 (1/4)



MPI의 기본 개념 (2/4)

- 프로세스와 프로세서
 - MPI는 프로세스 기준으로 작업할당
 - 프로세서 대 프로세스 = 일대일 또는 일대다

- 메시지 (= 데이터 + 봉투(Envelope))
 - 어떤 프로세스가 보내는가
 - 어디에 있는 데이터를 보내는가
 - 어떤 데이터를 보내는가
 - 얼마나 보내는가
 - 어떤 프로세스가 받는가
 - 어디에 저장할 것인가
 - 얼마나 받을 준비를 해야 하는가

MPI의 기본 개념 (3/4)

- **표지표(tag)**
 - 메시지 매칭과 구분에 이용
 - 순서대로 메시지 도착을 처리할 수 있음
 - 와일드 카드 사용 가능

- **커뮤니케이터(Communicator)**
 - 서로간에 통신이 허용되는 프로세스들의 집합

- **프로세스 랭크(Rank)**
 - 동일한 커뮤니케이터 내의 프로세스들을 식별하기 위한 식별자

MPI의 기본 개념 (4/4)

- **점대점 통신(Point to Point Communication)**
 - 두 개 프로세스 사이의 통신
 - 하나의 송신 프로세스에 하나의 수신 프로세스가 대응

- **집합통신(Collective Communication)**
 - 동시에 여러 개의 프로세스가 통신에 참여
 - 일대다, 다대일, 다대다 대응 가능
 - 여러 번의 점대점 통신 사용을 하나의 집합통신으로 대체
 - 오류의 가능성이 적다.
 - 최적화 되어 일반적으로 빠르다.

제 2 장 MPI를 이용한 병렬 프로그래밍 기초

MPI를 이용한 병렬 프로그램 작성의 기본
과 통신, 유도 데이터 타입의 이용, 그리고
가상 토폴로지에 대해 알아본다.

- MPI 프로그램의 기본구조
- 커뮤니케이터
- 메시지
- MPI 데이터 타입



MPI 프로그램의 기본 구조

include MPI header file
variable declarations
initialize the MPI environment

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

close MPI environment

MPI 헤더파일

□ 헤더파일 삽입

Fortran	C
<code>INCLUDE 'mpif.h'</code>	<code>#include "mpi.h"</code>

- MPI 서브루틴과 함수의 프로토타입 선언
- 매크로, MPI 관련 인수, 데이터 타입 정의
- 위치
 - `/usr/lpp/ppe.poe/include/`

MPI 핸들

- MPI 고유의 내부 자료구조 참조에 이용되는 포인터 변수
- C의 핸들은 typedef으로 정의된 특별한 데이터 타입을 가짐
 - MPI_Comm, MPI_Datatype, MPI_Request, ...
- Fortran의 핸들은 INTEGER 타입

MPI 루틴의 호출과 리턴 값 (1/2)

- MPI 루틴의 호출

Fortran	
Format	CALL MPI_XXXXX(parameter, ..., ierr)
Example	CALL MPI_INIT(ierr)
Error code	Returned as "ierr" parameter, MPI_SUCCESS if successful

C	
Format	err = MPI_XXXXX(parameter, ...); MPI_XXXXX(parameter, ...);
Example	err = MPI_Init(&argc, &argv);
Error code	Returned as "err", MPI_SUCCESS if successful

MPI 루틴의 호출과 리턴 값 (2/2)

□ MPI 루틴의 리턴 값

- 호출된 MPI 루틴의 실행 성공을 알려주는 에러코드 리턴
- 성공적으로 실행 되면 정수형 상수 'MPI_SUCCESS' 리턴
- Fortran 서브루틴은 마지막 정수인수가 에러코드를 나타냄
- MPI_SUCCESS는 헤더파일에 선언되어 있음

Fortran	C
<pre> INTEGER ierr CALL MPI_INIT(ierr) IF(ierr .EQ. MPI_SUCCESS) THEN ... ENDIF </pre>	<pre> int err; err = MPI_Init(&argc, &argv); if (err == MPI_SUCCESS){ ... } </pre>

MPI 초기화

Fortran	C
<pre> CALL MPI_INIT(ierr) </pre>	<pre> int MPI_Init(&argc, &argv) </pre>

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

커뮤니케이터 (1/3)

- 서로 통신할 수 있는 프로세스들의 집합을 나타내는 핸들
- 모든 MPI 통신 루틴에는 커뮤니케이터 인수가 포함 됨
- 커뮤니케이터를 공유하는 프로세스들끼리 통신 가능
- **MPI_COMM_WORLD**
 - 프로그램 실행시 정해진, 사용 가능한 모든 프로세스를 포함하는 커뮤니케이터
 - MPI_Init이 호출될 때 정의 됨

커뮤니케이터 (2/3)

- 프로세스 랭크
 - 같은 커뮤니케이터에 속한 프로세스의 식별 번호
 - 프로세스가 n개 있으면 0부터 n-1까지 번호 할당
 - 메시지의 송신자와 수신자를 나타내기 위해 사용
 - 프로세스 랭크 가져오기

Fortran	<code>CALL MPI_COMM_RANK(comm, rank, ierr)</code>
C	<code>int MPI_Comm_rank(MPI_Comm comm, int *rank)</code>

커뮤니케이터 comm에서 이 루틴을 호출한 프로세스의 랭크를 인수 rank를 이용해 출력

커뮤니케이터 (3/3)

□ 커뮤니케이터 사이즈

- 커뮤니케이터에 포함된 프로세스들의 총 개수
- 커뮤니케이터 사이즈 가져오기

Fortran	<code>CALL MPI_COMM_SIZE(comm, size, ierr)</code>
C	<code>int MPI_Comm_size(MPI_Comm comm, int *size)</code>

루틴을 호출되면 커뮤니케이터 comm의
사이즈를 인수 size를 통해 리턴

MPI 프로그램 종료

Fortran	C
<code>CALL MPI_FINALIZE(ierr)</code>	<code>int MPI_Finalize();</code>

- 모든 MPI 자료구조 정리
- 모든 프로세스들에서 마지막으로 한 번 호출되어야 함
- 프로세스를 종료 시키는 것은 아님

bones.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
```

bones.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();
}
```

MPI 메시지 (1/2)

□ 데이터 + 봉투

- **데이터** (버퍼, 개수, 데이터 타입)
 - 버퍼 : 수신(송신) 데이터의 변수 이름
 - 개수 : 수신(송신) 데이터의 개수
 - 데이터 타입 : 수신(송신) 데이터의 데이터 타입
- **봉투** (수신자(송신자), 교리표, 커뮤니케이터)
 - 수신자(송신자) : 수신(송신) 프로세스 랭크
 - 교리표 : 수신(송신) 데이터를 나타내는 고유한 정수
 - IBM/MPI, MPICH : 0 ~ 1073741823($2^{30}-1$)
 - 커뮤니케이터 : 송신, 수신 프로세스들이 포함된 프로세스 그룹

MPI 메시지 (2/2)

- **MPI 데이터**
 - 특정 MPI 데이터 타입을 가지는 원소들의 배열로 구성
- **MPI 데이터 타입**
 - 기본 타입
 - 유도 타입(derived type)
- 유도 타입은 기본 타입 또는 다른 유도 타입을 기반으로 만들 수 있다.
- C 데이터 타입과 Fortran 데이터 타입은 같지 않다.
- **송신과 수신 데이터 타입은 반드시 일치 해야 한다.**

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_BYTE	
MPI_PACKED	

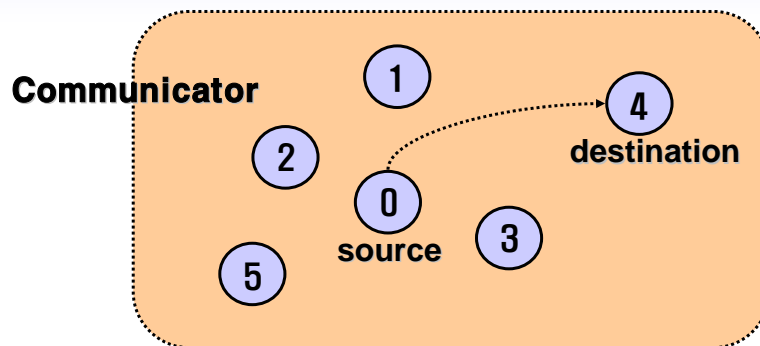
MPI 기본 데이터 타입 (2/2)

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
MPI_BYTE	
MPI_PACKED	

- 점대점통신과 통신 모드
- 블록킹 통신
- 논블록킹 통신
- 단방향 통신과 양방향 통신



점대점 통신 (1/2)



- 반드시 두 개의 프로세스만 참여하는 통신
- 통신은 커뮤니케이터 내에서만 이루어 진다.
- 송신/수신 프로세스의 확인을 위해 커뮤니케이터와 랭크 사용

점대점 통신 (2/2)

- **통신의 완료**
 - 메시지 전송에 이용된 메모리 위치에 안전하게 접근할 수 있음을 의미
 - 송신 : 송신 변수는 통신이 완료되면 다시 사용될 수 있음
 - 수신 : 수신 변수는 통신이 완료된 후부터 사용될 수 있음

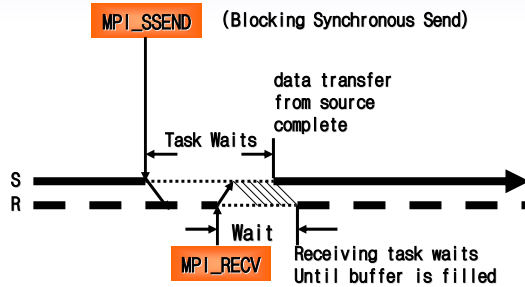
- **블록킹 통신과 논블록킹 통신**
 - 블록킹
 - 통신이 완료된 후 루틴으로부터 리턴 됨
 - 논블록킹
 - 통신이 시작되면 완료와 상관없이 리턴, 이후 완료 여부를 검사

- **통신 완료에 요구되는 조건에 따라 통신 모드 분류**

통신 모드

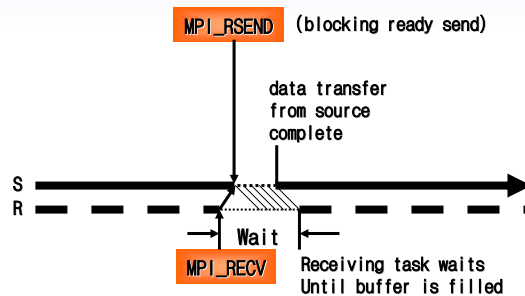
통신 모드	MPI 호출 루틴	
	블록킹	논블록킹
동기 송신	MPL_SSEND	MPL_ISSEND
준비 송신	MPL_RSEND	MPL_IRSEND
버퍼 송신	MPL_BSEND	MPL_IBSEND
표준 송신	MPL_SEND	MPL_ISEND
수신	MPL_RECV	MPL_Irecv

동기 송신



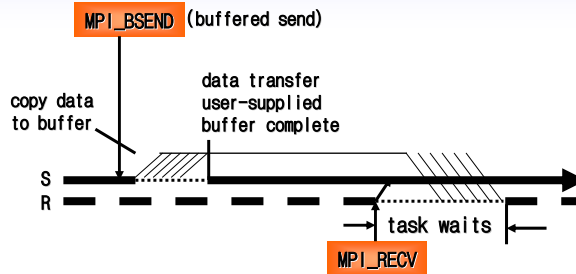
- 송신 시작 : 대응되는 수신 루틴의 실행에 무관하게 시작
- 송신 : 수신측이 받을 준비가 되면 전송시작
- 송신 완료 : 수신 루틴이 메시지를 받기 시작 + 전송 완료
- 가장 안전한 통신
- 논-로컬 송신 모드

준비 송신



- 수신측이 미리 받을 준비가 되어 있음을 가정하고 송신 시작
- 수신이 준비되지 않은 상황에서의 송신은 에러
- 성능면에서 유리
- 논-로컬 송신 모드

버퍼 송신



- 송신 시작 : 대응되는 수신 루틴의 실행에 무관하게 시작
- 송신 완료 : 버퍼로 복사가 끝나면 수신과 무관하게 완료
- 사용자가 직접 버퍼공간 관리
 - MPI_Buffer_attach
 - MPI_Buffer_detach
- 로컬 송신 모드

표준 송신

- 직접 복사
송신 버퍼 → 수신 버퍼
- 버퍼 사용
송신 버퍼 → 시스템 버퍼 → 수신 버퍼
- 상황에 따라 다르게 실행됨
- 버퍼관리 불필요
- 송신 완료가 반드시 메시지가 도착되었음을 의미하지 않음
- 논-로컬 송신 모드

블록킹 송신 : 표준

C	<code>int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)</code>
Fortran	<code>MPI_SEND(buf, count, datatype, dest, tag, comm, ierr)</code>

(CHOICE) buf : 송신 버퍼의 시작 주소 (IN)
 INTEGER count : 송신될 원소 개수 (IN)
 INTEGER datatype : 각 원소의 MPI 데이터 타입(핸들) (IN)
 INTEGER dest : 수신 프로세스의 랭크 (IN)
 통신이 불필요하면 MPI_PROC_NULL
 INTEGER tag : 메시지 꼬리표 (IN)
 INTEGER comm : MPI 커뮤니케이터(핸들) (IN)

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

블록킹 수신 (1/4)

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

(CHOICE) buf : 수신 버퍼의 시작 주소 (OUT)
 INTEGER count : 수신될 원소 개수 (IN)
 INTEGER datatype : 각 원소의 MPI 데이터 타입(핸들) (IN)
 INTEGER source : 송신 프로세스의 랭크 (IN)
 통신이 불필요하면 MPI_PROC_NULL
 INTEGER tag : 메시지 꼬리표 (IN)
 INTEGER comm : MPI 커뮤니케이터(핸들) (IN)
 INTEGER status(MPI_STATUS_SIZE) : 수신된 메시지의 정보 저장 (OUT)

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

블록킹 수신 (2/4)

- 수신자는 와일드 카드를 사용할 수 있음
- 모든 프로세스로부터 메시지 수신
MPL_ANY_SOURCE
- 어떤 교리표를 단 메시지는 모두 수신
MPL_ANY_TAG

블록킹 수신 (3/4)

- 수신자의 status 인수에 저장되는 정보
 - 송신 프로세스
 - 교리표
 - 데이터 크기 : MPL_GET_COUNT 사용

Information	Fortran	C
source	status(MPL_SOURCE)	status.MPL_SOURCE
tag	status(MPL_TAG)	status.MPL_TAG
count	MPL_GET_COUNT	MPL_Get_count

블록킹 수신 (4/4)

- **MPI_GET_COUNT** : 수신된 메시지의 원소 개수를 리턴

C	<code>int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)</code>
Fortran	<code>MPI_GET_COUNT(status, datatype, count, ierr)</code>

INTEGER status(MPI_STATUS_SIZE) : 수신된 메시지의 상태 (IN)
 INTEGER datatype : 각 원소의 데이터 타입 (IN)
 INTEGER count : 원소의 개수 (OUT)

블록킹 통신 예제 : Fortran

```
PROGRAM isend
  INCLUDE 'mpif.h'
  INTEGER err, rank, size, count
  REAL data(100), value(200)
  INTEGER status(MPI_STATUS_SIZE)
  CALL MPI_INIT(err)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD,rank,err)
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD,size,err)
  IF (rank.eq.1) THEN
    data=3.0
    CALL MPI_SEND(data,100,MPI_REAL,0,55,MPI_COMM_WORLD,err)
  ELSEIF (rank .eq. 0) THEN
    CALL MPI_RECV(value,200,MPI_REAL,MPI_ANY_SOURCE,55, &
      MPI_COMM_WORLD,status,err)
    PRINT *, "P:",rank," got data from processor ", &
      status(MPI_SOURCE)
    CALL MPI_GET_COUNT(status,MPI_REAL,count,err)
    PRINT *, "P:",rank," got ",count," elements"
    PRINT *, "P:",rank," value(5)=",value(5)
  ENDIF
  CALL MPI_FINALIZE(err)
END
```

블록킹 통신 예제 : C

```
#include <stdio.h>
#include <mpi.h>
void main(int argc, char *argv[]) {
    int rank, i, count;
    float data[100], value[200];
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if(rank==1) {
        for(i=0; i<100; ++i) data[i]=i;
        MPI_Send(data, 100, MPI_FLOAT, 0, 55, MPI_COMM_WORLD);
    }
    else if(rank==0) {
        MPI_Recv(value, 200, MPI_FLOAT, MPI_ANY_SOURCE, 55, MPI_COMM_WORLD,
                &status);
        printf("P:%d Got data from processor %d \n", rank,
                status.MPI_SOURCE);
        MPI_Get_count(&status, MPI_FLOAT, &count);
        printf("P:%d Got %d elements \n", rank, count);
        printf("P:%d value[5]=%f \n", rank, value[5]);
    }
    MPI_Finalize();
}
```

성공적인 통신을 위해 주의할 점들

- 송신측에서 수신자 랭크를 명확히 할 것
- 수신측에서 송신자 랭크를 명확히 할 것
- 커뮤니케이터가 동일 할 것
- 메시지 교리표가 일치할 것
- 수신버퍼는 충분히 클 것

논블록킹 통신

- 통신을 세 가지 상태로 분류
 1. 논블록킹 통신의 초기화 : 송신 또는 수신에 포스팅
 2. 전송 데이터를 사용하지 않는 다른 작업 수행
 - 통신과 계산 작업을 동시 수행
 3. 통신 완료 : 대기 또는 검사
- 교착 가능성 제거, 통신 부하 감소

논블록킹 통신의 초기화

C	<code>int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)</code>
Fortran	<code>MPI_ISEND(buf, count, datatype, dest, tag, comm, request, ierr)</code>
C	<code>int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)</code>
Fortran	<code>MPI_IRecv(buf, count, datatype, source, tag, comm, request, ierr)</code>

INTEGER request : 초기화된 통신의 식별에 이용 (핸들) (OUT)

논블록킹 수신에는 status 인수가 없음

논블록킹 통신의 완료

□ 대기(waiting) 또는 검사(testing)

- 대기
 - 루틴이 호출되면 통신이 완료될 때까지 프로세스를 블록킹
 - 논블록킹 통신 + 대기 = 블록킹 통신
- 검사
 - 루틴은 통신의 완료여부에 따라 참 또는 거짓을 리턴

대기

C	<code>int MPI_Wait(MPI_Request *request, MPI_Status *status)</code>
Fortran	<code>MPI_WAIT(request, status, ierr)</code>

INTEGER request : 포스팅된 통신의 식별에 이용 (핸들) (INOUT)

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

검사

C	<code>int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)</code>
Fortran	<code>MPI_TEST(request, flag, status, ierr)</code>

INTEGER request : 포스팅된 통신의 식별에 이용 (핸들) (INOUT)

INTEGER flag : 통신이 완료되면 참, 아니면 거짓을 리턴 (OUT)

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

논블록킹 통신 예제 : Fortran

```
PROGRAM isend
INCLUDE 'mpif.h'
INTEGER err, rank, count, req
REAL data(100), value(200)
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,200,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)
```

논블록킹 통신 예제 : C

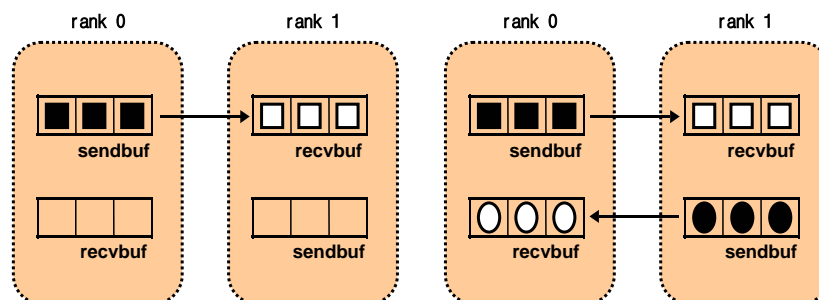
```

/* isend */
#include <stdio.h>
#include <mpi.h>
void main(int argc, char *argv[]) {
    int rank, i;
    float data[100],value[200];
    MPI_Request req; MPI_Status status;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    if(rank==1) {
        for(i=0;i<100;++i) data[i]=i;
        MPI_Isend(data,100,MPI_FLOAT,0,55,MPI_COMM_WORLD,&req);
        MPI_Wait(&req, &status);
    }
    else if(rank==0){
        MPI_Irecv(value,200,MPI_FLOAT,1,55,MPI_COMM_WORLD,&req);
        MPI_Wait(&req, &status);
        printf("P:%d value[5]=%f \n",rank,value[5]);
    }
    MPI_Finalize();
}

```

점대점 통신의 사용

- 단방향 통신과 양방향 통신
- 양방향 통신은 교착에 주의



단방향 통신 (1/2)

□ 블록킹 송신, 블록킹 수신

```
IF (myrank==0) THEN
  CALL MPI_SEND(sendbuf, icount, MPI_REAL8, 1, itag,
    MPI_COMM_WORLD, ierr)
ELSEIF (myrank==1) THEN
  CALL MPI_RECV(recvbuf, icount, MPI_REAL8, 0, itag,
    MPI_COMM_WORLD, istatus, ierr)
ENDIF
```

□ 논블록킹 송신, 블록킹 수신

```
IF (myrank==0) THEN
  CALL MPI_ISEND(sendbuf, icount, MPI_REAL8, 1, itag,
    MPI_COMM_WORLD, ireq, ierr)
  CALL MPI_WAIT(ireq, istatus, ierr)
ELSEIF (myrank==1) THEN
  CALL MPI_RECV(recvbuf, icount, MPI_REAL8, 0, itag,
    MPI_COMM_WORLD, istatus, ierr)
ENDIF
```

단방향 통신 (2/2)

□ 블록킹 송신, 논블록킹 수신

```
IF (myrank==0) THEN
  CALL MPI_SEND(sendbuf, icount, MPI_REAL8, 1, itag,
    MPI_COMM_WORLD, ierr)
ELSEIF (myrank==1) THEN
  CALL MPI_Irecv(recvbuf, icount, MPI_REAL8, 0, itag,
    MPI_COMM_WORLD, ireq, ierr)
  CALL MPI_WAIT(ireq, istatus, ierr)
ENDIF
```

□ 논블록킹 송신, 논블록킹 수신

```
IF (myrank==0) THEN
  CALL MPI_ISEND(sendbuf, icount, MPI_REAL8, 1, itag,
    MPI_COMM_WORLD, ireq, ierr)
ELSEIF (myrank==1) THEN
  CALL MPI_Irecv(recvbuf, icount, MPI_REAL8, 0, itag,
    MPI_COMM_WORLD, ireq, ierr)
ENDIF
CALL MPI_WAIT(ireq, istatus, ierr)
```

양방향 통신 (1/9)

□ 선 송신, 후 수신 1.: 메시지 크기에 따라 교착 가능

```
IF (myrank==0) THEN
    CALL MPI_SEND(sendbuf, ...)
    CALL MPI_RECV(recvbuf, ...)
ELSEIF (myrank==1) THEN
    CALL MPI_SEND(sendbuf, ...)
    CALL MPI_RECV(recvbuf, ...)
ENDIF
```

양방향 통신 (2/9)

□ 선 송신, 후 수신 2. (1.의 경우와 동일)

```
IF (myrank==0) THEN
    CALL MPI_ISEND(sendbuf, ..., ireq, ...)
    CALL MPI_WAIT(ireq, ...)
    CALL MPI_RECV(recvbuf, ...)
ELSEIF (myrank==1) THEN
    CALL MPI_ISEND(sendbuf, ..., ireq, ...)
    CALL MPI_WAIT(ireq, ...)
    CALL MPI_RECV(recvbuf, ...)
ENDIF
```

양방향 통신 (3/9)

- 선 송신, 후 수신 3. : 메시지 크기와 무관 하게 교착 없음

```
IF (myrank==0) THEN
  CALL MPI_ISEND(sendbuf, ..., ireq, ...)
  CALL MPI_RECV(recvbuf, ...)
  CALL MPI_WAIT(ireq, ...)
ELSEIF (myrank==1) THEN
  CALL MPI_ISEND(sendbuf, ..., ireq, ...)
  CALL MPI_RECV(recvbuf, ...)
  CALL MPI_WAIT(ireq, ...)
ENDIF
```

양방향 통신 (4/9)

- 전송 데이터 크기에 따른 교착여부 확인
- 교착을 피하기 위해 논블록킹 통신 사용

```
INTEGER N
PARAMETER (N=1024)
REAL a(N), b(N)
...
IF( myrank .eq. 0 ) THEN
  CALL MPI_SEND( a, N, ...)
  CALL MPI_RECV( b, N, ...)
ELSE IF( myrank .eq. 1 ) THEN
  CALL MPI_SEND( a, N, ...)
  CALL MPI_RECV( b, N, ...)
ENDIF
...
```

양방향 통신 (5/9)

- 선 수신, 후 송신 1.: 메시지 크기와 무관하게 교착

```
IF (myrank==0) THEN
    CALL MPI_RECV(recvbuf, ...)
    CALL MPI_SEND(sendbuf, ...)
ELSEIF (myrank==1) THEN
    CALL MPI_RECV(recvbuf, ...)
    CALL MPI_SEND(sendbuf, ...)
ENDIF
```

양방향 통신 (6/9)

- 선 수신, 후 송신 2.: 메시지 크기와 무관하게 교착 없음

```
IF (myrank==0) THEN
    CALL MPI_Irecv(recvbuf, ...,ireq, ...)
    CALL MPI_SEND(sendbuf, ...)
    CALL MPI_WAIT(ireq, ...)
ELSEIF (myrank==1) THEN
    CALL MPI_Irecv(recvbuf, ...,ireq, ...)
    CALL MPI_SEND(sendbuf, ...)
    CALL MPI_WAIT(ireq, ...)
ENDIF
```

양방향 통신 (7/9)

- 전송 데이터 크기와 무관한 교착발생 확인
- 교착을 피하기 위해 논블록킹 통신 사용

```
...  
REAL a(100), b(100)  
...  
IF (myrank==0) THEN  
    CALL MPI_RECV(b, 100,...)  
    CALL MPI_SEND(a, 100, ...)  
ELSE IF (myrank==1) THEN  
    CALL MPI_RECV(b, 100, ...)  
    CALL MPI_SEND(a, 100, ...)  
ENDIF  
...
```

양방향 통신 (8/9)

- 한쪽은 송신부터, 다른 한쪽은 수신부터
: 블록킹, 논블록킹 루틴의 사용과 무관하게 교착 없음

```
IF (myrank==0) THEN  
    CALL MPI_SEND(sendbuf, ...)  
    CALL MPI_RECV(recvbuf, ...)  
ELSEIF (myrank==1) THEN  
    CALL MPI_RECV(recvbuf, ...)  
    CALL MPI_SEND(sendbuf, ...)  
ENDIF
```


양방향 통신 (9/9)

□ 권장 코드

```
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, ...)
```

- 집합통신
- 방송(Broadcast)
- 취합(Gather)
- 환산(Reduce)
- 확산(Scatter)
- 장벽(Barrier)
- 기타



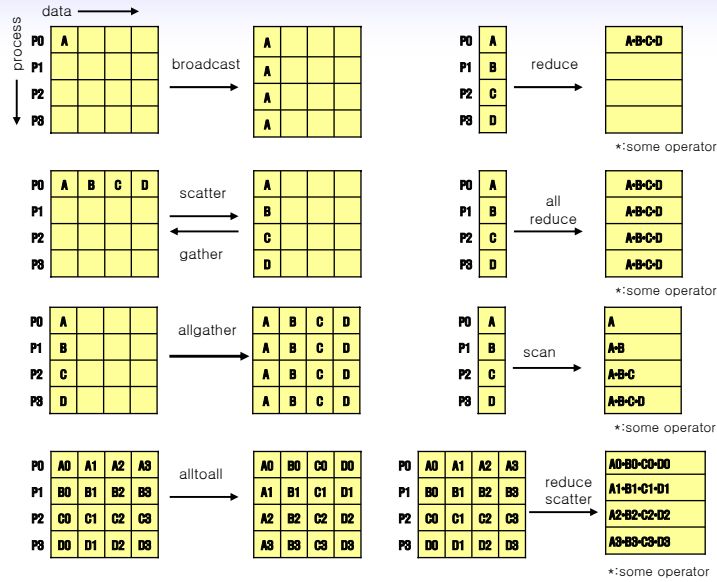
집합 통신 (1/3)

- 한 그룹의 프로세스가 참여하는 통신
- 점대점 통신 기반
- 점대점 통신을 이용한 구현보다 편리하고 성능면에서 유리
- 집합 통신 루틴
 - 커뮤네이터 내의 모든 프로세스에서 호출
 - 동기화가 보장되지 않음 (MPI_Barrier 제외)
 - 논블록킹 루틴 없음
 - 꼬리표 없음

집합 통신 (2/3)

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_ALLTOALL
Reduction	MPI_REDUCE , MPI_ALLREDUCE , MPI_SCAN , MPI_REDUCE_SCATTER
Others	MPI_BARRIER , MPI_OP_CREATE , MPI_OP_FREE

집합 통신 (3/3)



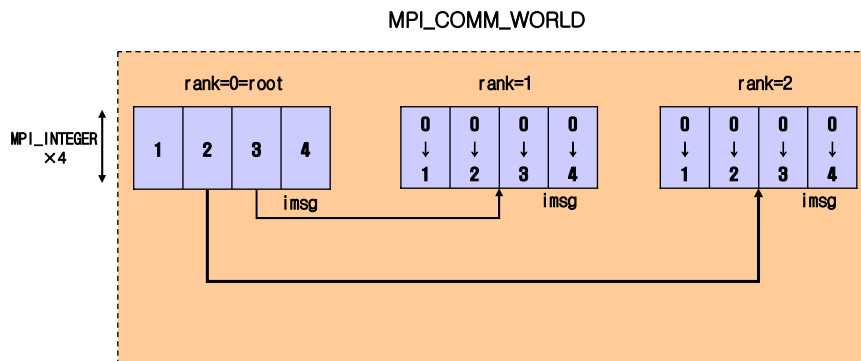
방송 : MPI_BCAST

C	<code>int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)</code>
Fortran	<code>MPI_BCAST(buffer, count, datatype, root, comm, ierr)</code>

(CHOICE) buffer : 버퍼의 시작 주소 (INOUT)
 INTEGER count : 버퍼 원소의 개수 (IN)
 INTEGER datatype : 버퍼 원소의 MPI 데이터 타입 (IN)
 INTEGER root : 루트 프로세스의 랭크 (IN)
 INTEGER comm : 커뮤니케이터 (IN)

- 루트 프로세스로부터 커뮤니케이터 내의 다른 프로세스로 동일한 데이터를 전송 : 일대다 통신

MPI_BCAST 예제



MPI_BCAST 예제 : Fortran

```

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

```

MPI_BCAST 예제 : C

```

/*broadcast*/
#include <mpi.h>
#include <stdio.h>
void main (int argc, char *argv[]){
    int i, myrank ;
    int imsg[4];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    if (myrank==0) for(i=0; i<4; i++) imsg[i] = i+1;
    else for (i=0; i<4; i++) imsg[i] = 0;
    printf("%d: BEFORE:", myrank);
    for(i=0; i<4; i++) printf(" %d", imsg[i]);
    printf("\n");
    MPI_Bcast(imsg, 4, MPI_INT, 0, MPI_COMM_WORLD);
    printf("%d: AFTER:", myrank);
    for(i=0; i<4; i++) printf(" %d", imsg[i]); printf("\n");
    MPI_Finalize();
}

```

취합 : 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)

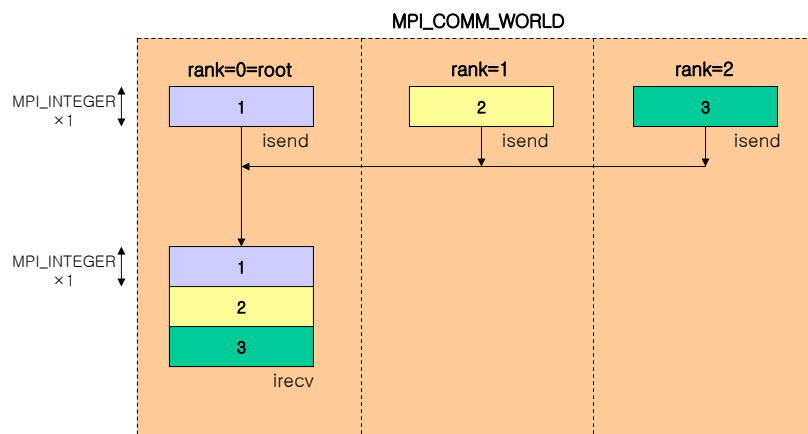
(CHOICE) sendbuf : 송신 버퍼의 시작 주소 (IN)
 INTEGER sendcount : 송신 버퍼의 원소 개수 (IN)
 INTEGER sendtype : 송신 버퍼 원소의 MPI 데이터 타입 (IN)
 (CHOICE) recvbuf : 수신 버퍼의 주소 (OUT)
 INTEGER recvcount : 수신할 원소의 개수 (IN)
 INTEGER recvtype : 수신 버퍼 원소의 MPI 데이터 타입 (IN)
 INTEGER root : 수신 프로세스(루트 프로세스)의 랭크 (IN)
 INTEGER comm : 커뮤니케이터 (IN)

- 모든 프로세스(루트 포함)가 송신한 데이터를 취합하여 랭크 순서대로 저장 : 다대일 통신

MPI_GATHER : 주의 사항

- 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 예제



MPI_GATHER 예제 : Fortran

```
PROGRAM gather
INCLUDE 'mpif.h'
INTEGER irecv(3)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
isend = myrank + 1
CALL MPI_GATHER(isend,1,MPI_INTEGER,irecv,1,MPI_INTEGER,&
    0, MPI_COMM_WORLD, ierr)
IF (myrank==0) THEN
    PRINT *, 'irecv =',irecv
ENDIF
CALL MPI_FINALIZE(ierr)
END
```

MPI_GATHER 예제 : C

```
/*gather*/
#include <mpi.h>
#include <stdio.h>
void main (int argc, char *argv[]){
    int i, nprocs, myrank ;
    int isend, irecv[3];
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    isend = myrank + 1;
    MPI_Gather(&isend,1,MPI_INT,irecv,1,MPI_INT,0,MPI_COMM_WORLD);
    if(myrank == 0) {
        printf(" irecv = ");
        for(i=0; i<3; i++)
            printf(" %d", irecv[i]); printf("\n");
    }
    MPI_Finalize();
}
```

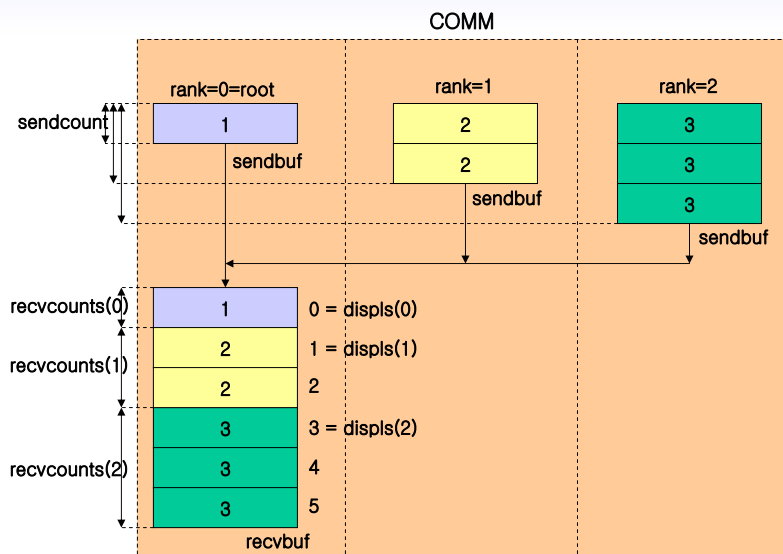
취합 : MPL_GATHERV

C	<pre>int MPI_Gatherv(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcunts, int displs, MPI_Datatype recvtype, int root, MPI_Comm comm)</pre>
Fortran	<pre>MPI_GATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcunts, displs, recvtype, root, comm, ierr)</pre>

...
 (CHOICE) recvbuf : 수신버퍼의 주소(OUT)
 INTEGER recvcunts(*) : 수신된 원소의 개수를 저장하는 정수 배열(IN)
 INTEGER displs(*) : 정수 배열, i번째 자리에는 프로세스 i에서 들어오는 데이
 터가 저장될 수신 버퍼 상의 위치를 나타냄(IN)
 ...

- ❑ 각 프로세스로부터 전송되는 데이터의 크기가 다를 경우 사용
- ❑ 서로 다른 메시지 크기는 배열 recvcunts에 지정, 배열 displs에 루트 프로세스의 어디에 데이터가 위치하게 되는가를 저장

MPI_GATHERV 예제



MPI_GATHERV 예제 : Fortran

```
PROGRAM gatherv
  INCLUDE 'mpif.h'
  INTEGER isend(3), irecv(6)
  INTEGER ircnt(0:2), idisp(0:2)
  DATA ircnt/1,2,3/ idisp/0,1,3/
  CALL MPI_INIT(ierr)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
  DO i=1,myrank+1
    isend(i) = myrank + 1
  ENDDO
  iscnt = myrank + 1
  CALL MPI_GATHERV(isend,iscnt,MPI_INTEGER,irecv,ircnt,idisp,&
    MPI_INTEGER,0,MPI_COMM_WORLD,ierr)
  IF (myrank==0) THEN
    PRINT *, 'irecv =',irecv
  ENDIF
  CALL MPI_FINALIZE(ierr)
END
```

MPI_GATHERV 예제 : C

```
/*gatherv*/
#include <mpi.h>
#include <stdio.h>
void main (int argc, char *argv[]){
  int i, myrank ;
  int isend[3], irecv[6];
  int iscnt, ircnt[3]={1,2,3}, idisp[3]={0,1,3};
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  for(i=0; i<myrank+1; i++) isend[i] = myrank + 1;
  iscnt = myrank +1;
  MPI_Gatherv(isend, iscnt, MPI_INT, irecv, ircnt, idisp,
    MPI_INT, 0, MPI_COMM_WORLD);
  if(myrank == 0) {
    printf(" irecv = "); for(i=0; i<6; i++) printf(" %d",
      irecv[i]);
    printf("\n");
  }
  MPI_Finalize();
}
```

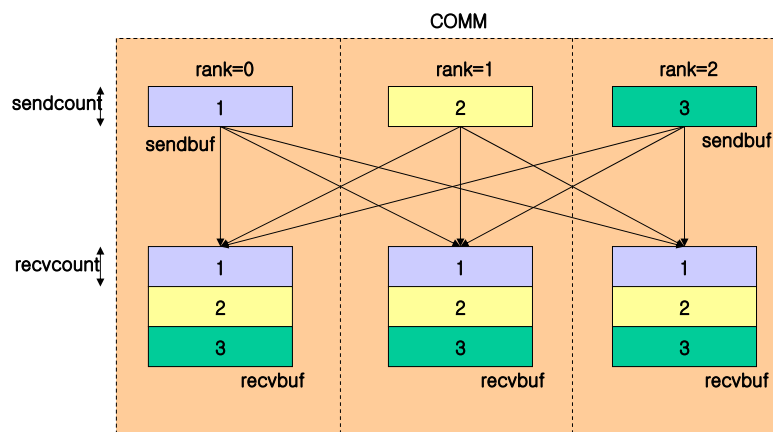
취합 : MPI_ALLGATHER

C	<code>int MPI_Allgather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)</code>
Fortran	<code>MPI_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, ierr)</code>

(CHOICE) sendbuf : 송신 버퍼의 시작 주소(IN)
 INTEGER sendcount : 송신 버퍼의 원소 개수(IN)
 INTEGER sendtype : 송신 버퍼 원소의 MPI 데이터 타입(IN)
 (CHOICE) recvbuf : 수신 버퍼의 주소(OUT)
 INTEGER recvcount : 각 프로세스로부터 수신된 데이터 개수(IN)
 INTEGER recvtype : 수신버퍼 데이터 타입(IN)
 INTEGER comm : 커뮤니케이터 (IN)

- ❑ **MPL_GATHER + MPL_BCAST**
- ❑ **프로세스 J 의 데이터 → 모든 수신 버퍼 J 번째 블록에 저장**

MPI_ALLGATHER 예제



MPI_ALLGATHER 예제 : Fortran

```
PROGRAM allgather
INCLUDE 'mpif.h'
INTEGER irecv(3)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
isend = myrank + 1
CALL MPI_ALLGATHER(isend, 1, MPI_INTEGER, &
    irecv, 1, MPI_INTEGER, MPI_COMM_WORLD, ierr)
PRINT *, 'irecv =', irecv
CALL MPI_FINALIZE(ierr)
END
```

MPI_ALLGATHER 예제 : C

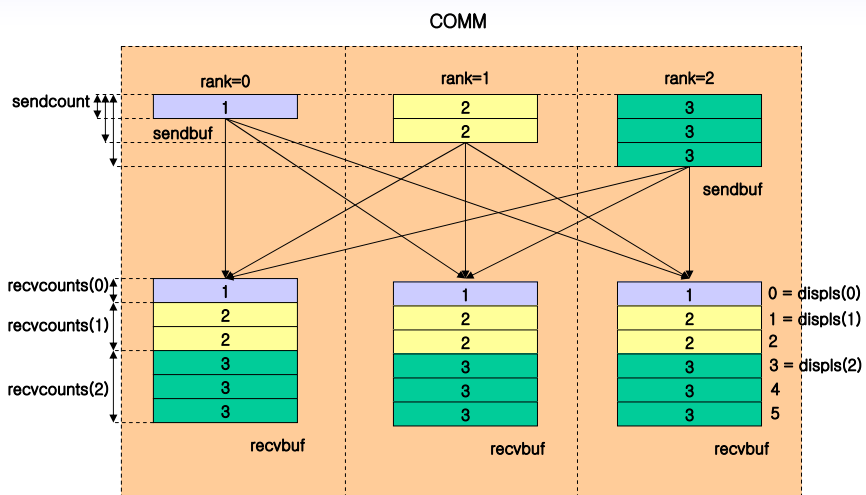
```
/*allgather*/
#include <mpi.h>
#include <stdio.h>
void main (int argc, char *argv[]){
    int i, myrank ;
    int isend, irecv[3];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    isend = myrank + 1;
    MPI_Allgather(&isend, 1, MPI_INT, irecv, 1,
        MPI_INT, MPI_COMM_WORLD);
    printf("%d irecv = ");
    for(i=0; i<3; i++) printf(" %d", irecv[i]);
    printf("\n");
    MPI_Finalize();
}
```

취합 : MPI_ALLGATHERV

C	<code>int MPI_Allgatherv(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcnts, int displs, MPI_Datatype recvtype, MPI_Comm comm)</code>
Fortran	<code>MPI_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcnts, displs, recvtype, comm, ierr)</code>

- MPI_ALLGATHER와 같은 기능을 하며 서로 다른 크기의 데이터를 취합할 때 사용

MPI_ALLGATHERV 예제



환산 : MPI_REDUCE

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

(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 데이터 타입 (1/3)

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 데이터 타입 (2/3)

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 : 연산과 데이터 타입 (3/3)

□ C의 MPI_MAXLOC, MPI_MINLOC에 사용된 데이터 타입

Data Type	Description (C)
MPI_FLOAT_INT	{ MPI_FLOAT, MPI_INT}
MPI_DOUBLE_INT	{ MPI_DOUBLE, MPI_INT}
MPI_LONG_INT	{ MPI_LONG, MPI_INT}
MPI_2INT	{ MPI_INT, MPI_INT}
MPI_SHORT_INT	{ MPI_SHORT, MPI_INT}
MPI_LONG_DOUBLE_INT	{ MPI_LONG_DOUBLE, MPI_INT}

MPL_REDUCE : 사용자 정의 연산 (1/2)

- 다음 형식으로 새로운 연산(my_operator)을 정의

C :

```
void my_operator (void *invec, void *inoutvec, int
                 *len, MPI_Datatype *datatype)
```

Fortran :

```
<type> INVEC(LEN), INOUTVEC(LEN)
INTEGER LEN, DATATYPE
FUNCTION MY_OPERATOR(INVEC(*), INOUTVEC(*), LEN,
                    DATATYPE)
```

MPL_REDUCE : 사용자 정의 연산 (2/2)

- 사용자 정의 연산의 등록 (: my_operator를 op로 등록)
 - 인수 commute가 참이면 환산이 좀 더 빠르게 수행됨

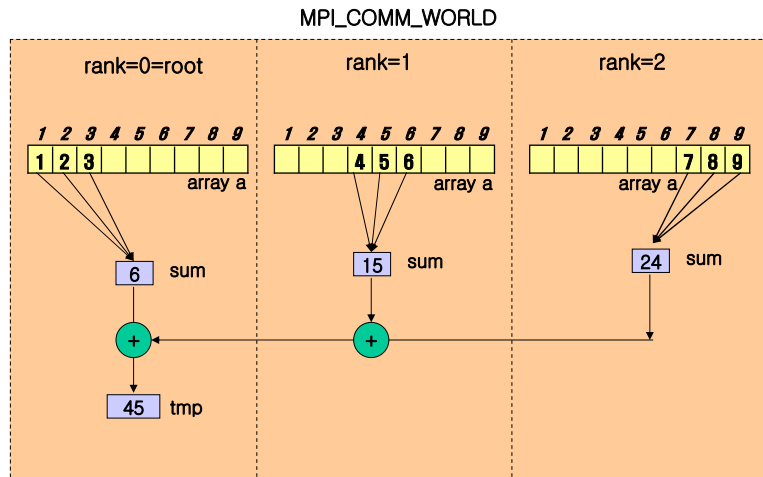
C :

```
int MPI_Op_create (MPI_User_function *my_operator,
                  int commute, MPI_Op *op)
```

Fortran :

```
EXTERNAL MY_OPERATOR
INTEGER OP, IERR
LOGICAL COMMUTE
MPI_OP_CREATE (MY_OPERATOR, COMMUTE, OP, IERR)
```

MPL_REDUCE 예제



MPI_REDUCE 예제 : 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
  
```

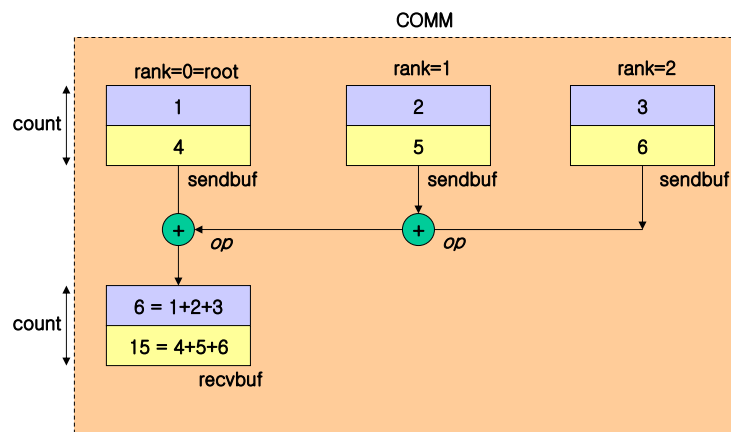

MPI_REDUCE 예제 : C

```

/*reduce*/
#include <mpi.h>
#include <stdio.h>
void main (int argc, char *argv[]){
    int i, myrank, ista, iend;
    double a[9], sum, tmp;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    ista = myrank*3 ;
    iend = ista + 2;
    for(i = ista; i<iend+1; i++) a[i] = i+1;
    sum = 0.0;
    for(i = ista; i<iend+1; i++) sum = sum + a[i];
    MPI_Reduce(&sum, &tmp, 1, MPI_DOUBLE, MPI_SUM, 0,
              MPI_COMM_WORLD);
    sum = tmp;
    if(myrank == 0) printf(" sum = %f \n", sum);
    MPI_Finalize();
}

```

MPI_REDUCE 예제 : 배열

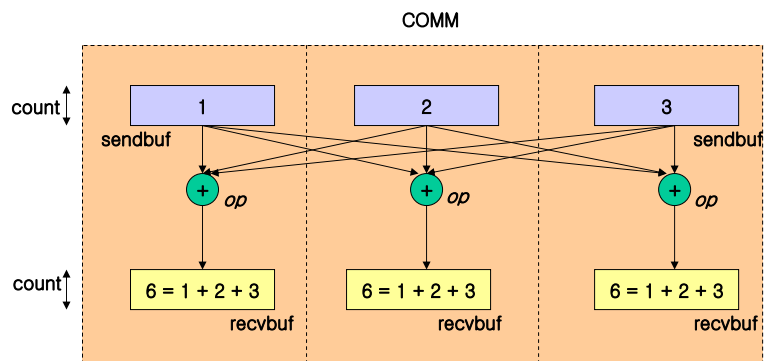


환산 : MPI_ALLREDUCE

C	<code>int MPI_Allreduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)</code>
Fortran	<code>MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm, ierr)</code>

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

MPI_ALLREDUCE 예제



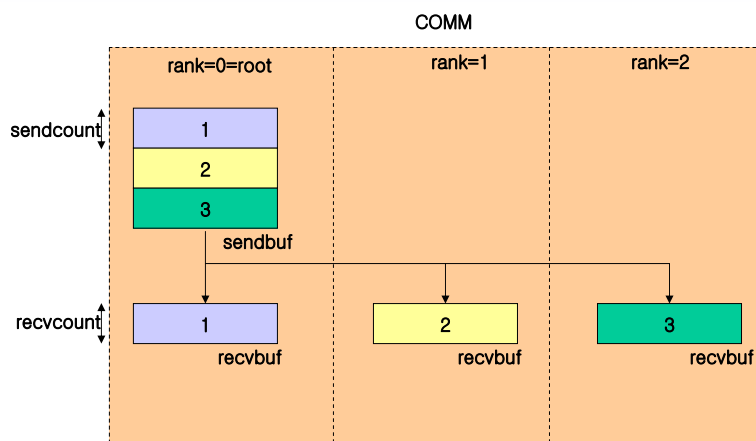
확산 : MPL_SCATTER

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

(CHOICE) sendbuf : 송신 버퍼의 주소 (IN)
 INTEGER sendcount : 각 프로세스로 보내지는 원소 개수 (IN)
 INTEGER sendtype : 송신 버퍼 원소의 MPI 데이터 타입 (IN)
 (CHOICE) recvbuf : 수신 버퍼의 주소 (OUT)
 INTEGER recvcount : 수신 버퍼의 원소 개수 (IN)
 INTEGER recvtype : 수신 버퍼의 MPI 데이터 타입 (IN)
 INTEGER root : 송신 프로세스의 랭크 (IN)
 INTEGER comm : 커뮤니케이터 (IN)

- 루트 프로세스는 데이터를 같은 크기로 나누어 각 프로세스에 랭크 순서대로 하나씩 전송

MPI_SCATTER 예제



MPI_SCATTER 예제 : Fortran

```
PROGRAM scatter
INCLUDE 'mpif.h'
INTEGER isend(3)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
IF (myrank==0) THEN
  DO i=1,nprocs
    isend(i)=i
  ENDDO
ENDIF
CALL MPI_SCATTER(isend, 1, MPI_INTEGER, irecv, 1, MPI_INTEGER, &
  0, MPI_COMM_WORLD, ierr)
PRINT *, 'irecv =',irecv
CALL MPI_FINALIZE(ierr)
END
```

MPI_SCATTER 예제 : C

```
/*scatter*/
#include <mpi.h>
#include <stdio.h>
void main (int argc, char *argv[]){
  int i, myrank, nprocs;
  int isend[3], irecv;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  for(i=0; i<nprocs; i++) isend[i]=i+1;
  MPI_Scatter(isend, 1, MPI_IN, &irecv, 1,
    MPI_INT, 0, MPI_COMM_WORLD);
  printf(" %d: irecv = %d\n", myrank, irecv);
  MPI_Finalize();
}
```

확산 : MPL_SCATTERV

C	<code>int MPI_Scatterv(void *sendbuf, int sendcounts, int displs, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)</code>
Fortran	<code>MPI_SCATTERV(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm, ierr)</code>

(CHOICE) sendbuf : 송신 버퍼의 주소 (IN)

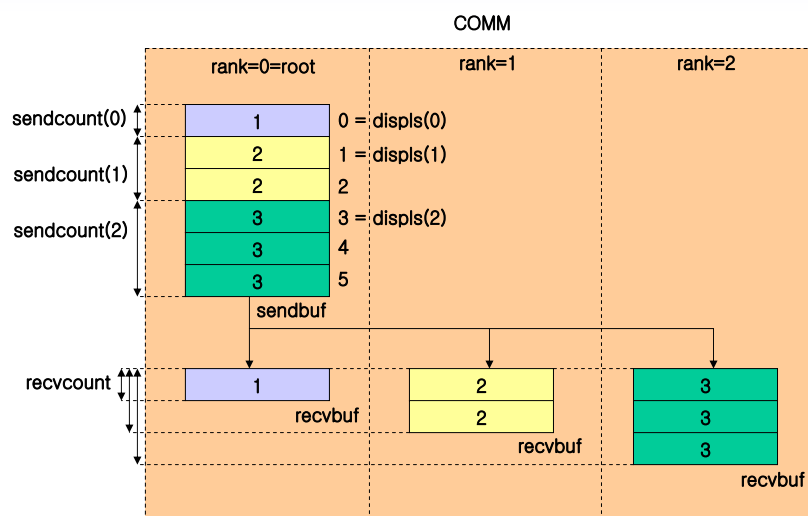
INTEGER sendcounts(*) : 정수 배열, i번째 자리에 프로세스 i 로 전송될 데이터 개수 저장(IN)

INTEGER displs(*) : 정수 배열, i번째 자리에 프로세스 i로 전송될 데이터의 송신 버퍼 상의 상대적 위치가 저장될 (IN)

...

- 루트 프로세스는 데이터를 서로 다른 크기로 나누어 각 프로세스에 랭크 순서대로 하나씩 전송

MPL_SCATTERV 예제



장벽 : MPI_BARRIER

C	<code>int MPI_Barrier(MPI_Comm comm)</code>
Fortran	<code>MPI_BARRIER(comm, ierr)</code>

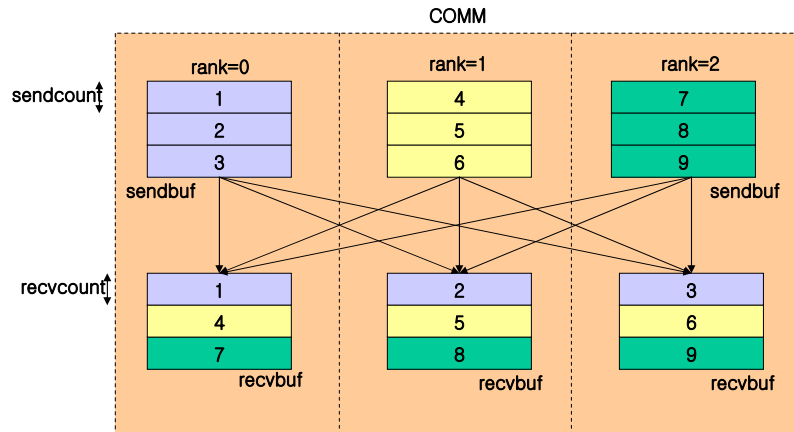
- ❑ 커뮤니케이터 내의 모든 프로세스가 MPI_BARRIER를 호출할 때까지 더 이상의 프로세스 진행을 막음

기타 : MPI_ALLTOALL

C	<code>int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)</code>
Fortran	<code>MPI_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, ierr)</code>

- ❑ 각 프로세스로부터 모든 프로세스에 동일한 크기의 개별적인 메시지를 전달
- ❑ 프로세스 *i* 로부터 송신되는 *j* 번째 데이터 블록은 프로세스 *j* 가 받아 수신버퍼의 *i* 번째 블록에 저장

MPI_ALLTOALL 예제

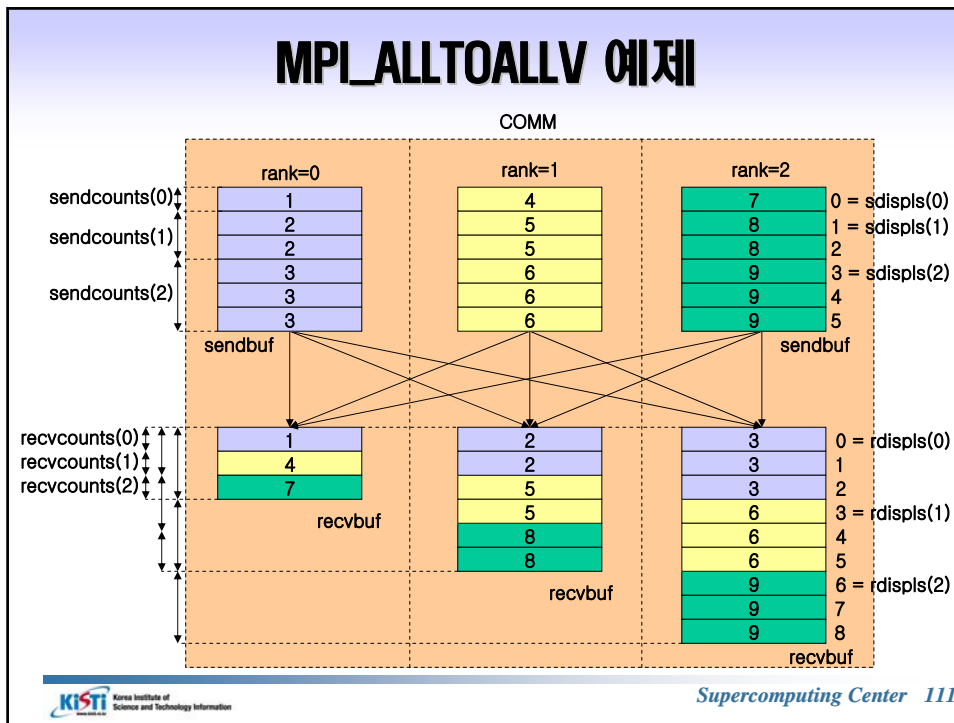


기타 : MPI_ALLTOALLV

C	<code>int MPI_Alltoallv(void *sendbuf, int sendcounts, int sdispls, MPI_Datatype sendtype, void *recvbuf, int rcvcounts, int rdispls, MPI_Datatype rcvtype, MPI_Comm comm)</code>
Fortran	<code>MPI_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, rcvcounts, rdispls, rcvtype, comm, ierr)</code>

- 각 프로세스로부터 모든 프로세스에 서로 다른 크기의 개별적인 메시지를 전달
- 프로세스 *i* 로부터 송신되는 `sendcounts[i]`개의 데이터는 프로세스 *j*가 받아 수신버퍼의 `rdispls[j]` 번째 위치부터 저장

MPI_ALLTOALLV 예제

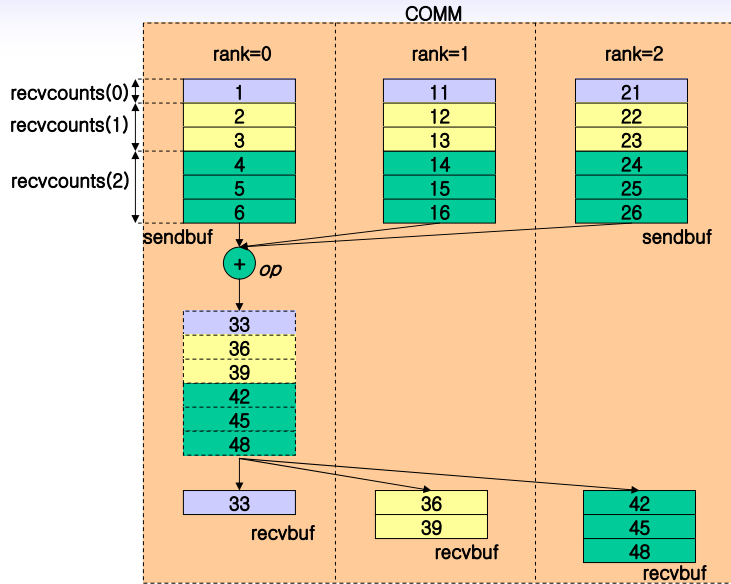


기타 : MPI_REDUCE_SCATTER

C	<pre>int MPI_Reduce_scatter(void *sendbuf, void *recvbuf, int rcvcounts, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)</pre>
Fortran	<pre>MPI_REDUCE_SCATTER(sendbuf, recvbuf, rcvcounts, datatype, op, comm, ierr)</pre>

- 각 프로세스의 송신버퍼에 저장된 데이터를 환산 연산 (reduction Operation)을 수행하고 그 결과를 차례대로 rcvcounts(i)개씩 모아서 프로세스 i로 송신
- MPI_REDUCE + MPI_SCATTERV

MPI_REDUCE_SCATTER 예제

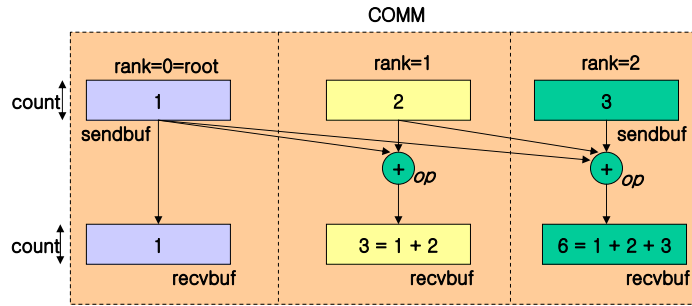


기타 : MPI_SCAN

C	<code>int MPI_Scan(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)</code>
Fortran	<code>MPI_SCAN(sendbuf, recvbuf, count, datatype, op, comm, ierr)</code>

- 프로세스 i의 수신버퍼에 프로세스 0에서 프로세스 i까지의 수신버퍼 데이터들에 대한 환산(reduction) 값을 저장한다

MPL_SCAN 예제



- 유도 데이터 타입
- 부분배열의 전송



유도 데이터 타입 (1/2)

□ 데이터 타입이 다르거나 불연속적인 데이터의 전송

- 동일한 데이터 타입을 가지는 불연속 데이터
- 다른 데이터 타입을 가지는 연속 데이터
- 다른 데이터 타입을 가지는 불연속 데이터

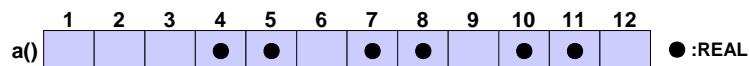
1. 각각을 따로 따로 전송

2. 새로운 버퍼로 묶어서 전송 후 묶음을 풀어 원위치로 저장 :
MPI_PACK/MPI_UNPACK, MPI_PACKED(데이터 타입)

→ 느린 속도, 불편, 오류의 위험

유도 데이터 타입 (2/2)

□ a(4), a(5), a(7), a(8), a(10), a(11)의 전송



● 유도 데이터 타입 itype1, 한 개 전송

```
CALL MPI_SEND(a(4), 1, itype1, idst, itag,
              MPI_COMM_WORLD, ierr)
```

● 유도 데이터 타입 itype2, 세 개 전송

```
CALL MPI_SEND(a(4), 3, itype2, idst, itag,
              MPI_COMM_WORLD, ierr)
```

유도 데이터 타입의 사용

- **CONSTRUCT**
 - MPI 루틴 이용해 새로운 데이터 타입 작성
 - `MPI_Type_contiguous`
 - `MPI_Type_(h)vector`
 - `MPI_Type_struct`

- **COMMIT**
 - 작성된 데이터 타입 등록
 - `MPI_Type_Commit`

- **USE**
 - 송신, 수신 등에 새로운 데이터 타입 사용

MPI_TYPE_COMMIT

C	<code>int MPI_Type_commit (MPI_Datatype *datatype)</code>
Fortran	<code>MPI_TYPE_COMMIT (datatype, ierr)</code>

INTEGER datatype : 등록 데이터 타입(핸들) (INOUT)

- 새롭게 정의된 데이터 타입을 통신상에서 사용 가능하게 함
- 등록된 데이터 타입은 `MPI_Type_free(datatype, ierr)`을 이용해 해제하기 전까지 계속 사용 가능

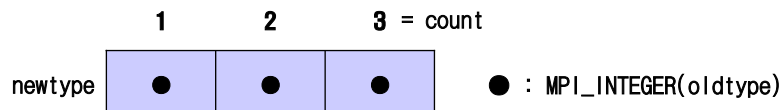
MPI_TYPE_CONTIGUOUS

C	<code>int MPI_Type_contiguous (int count, MPI_Datatype oldtype, MPI_Datatype *newtype)</code>
Fortran	<code>MPI_TYPE_CONTIGUOUS (count, oldtype, newtype, ierr)</code>

INTEGER count : 하나로 묶을 데이터 개수 (IN)
 INTEGER oldtype : 이전 데이터 타입 (핸들) (IN)
 INTEGER newtype : 새로운 데이터 타입 (핸들) (OUT)

- 같은 데이터 타입(oldtype)을 가지는 연속적인 데이터를 count 개 묶어 새로운 데이터 타입(newtype) 정의

MPI_TYPE_CONTIGUOUS 예제



MPI_TYPE_CONTIGUOUS 예제 : Fortran

```
PROGRAM type_contiguous
INCLUDE 'mpif.h'
INTEGER ibuf(20)
INTEGER inewtype
CALL MPI_INIT(ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
IF (myrank==0) THEN
  DO i=1,20
    ibuf(i) = i
  ENDDO
ENDIF
CALL MPI_TYPE_CONTIGUOUS(3, MPI_INTEGER, inewtype, ierr)
CALL MPI_TYPE_COMMIT(inewtype, ierr)
CALL MPI_BCAST(ibuf, 3, inewtype, 0, MPI_COMM_WORLD, ierr)
PRINT *, 'ibuf =',ibuf
CALL MPI_FINALIZE(ierr)
END
```

MPI_TYPE_CONTIGUOUS 예제 : C

```
/*type_contiguous*/
#include <mpi.h>
#include <stdio.h>
void main (int argc, char *argv[]){
  int i, myrank, ibuf[20];
  MPI_Datatype inewtype ;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  if(myrank==0) for(i=0; i<20; i++) ibuf[i]=i+1;
  else for(i=0; i<20; i++) ibuf[i]=0;
  MPI_Type_contiguous(3, MPI_INT, &inewtype);
  MPI_Type_commit(&inewtype);
  MPI_Bcast(ibuf, 3, inewtype, 0, MPI_COMM_WORLD);
  printf("%d : ibuf =", myrank);
  for(i=0; i<20; i++) printf(" %d", ibuf[i]);
  printf("\n");
  MPI_Finalize();
}
```

MPL_TYPE_VECTOR (1/2)

C	<code>int MPI_Type_vector (int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)</code>
Fortran	<code>MPI_Type_vector (count, blocklength, stride, oldtype, newtype, ierr)</code>

INTEGER count : 블록의 개수(IN)

INTEGER blocklength : 각 블록의 oldtype 데이터의 개수(IN)

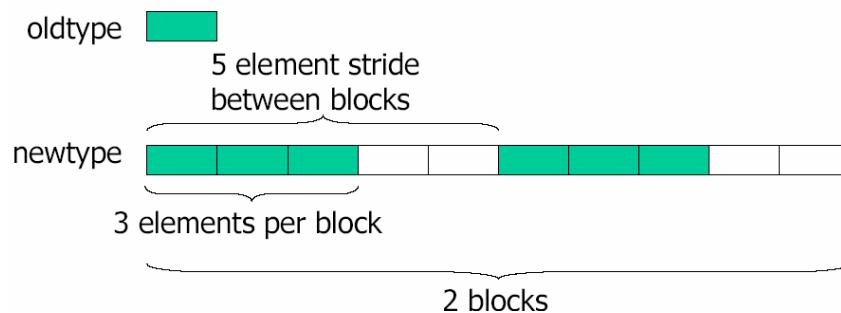
INTEGER stride : 인접한 두 블록의 시작점 사이의 폭(IN)

INTEGER oldtype : 이전 데이터 타입(핸들) (IN)

INTEGER newtype : 새로운 데이터 타입(핸들) (OUT)

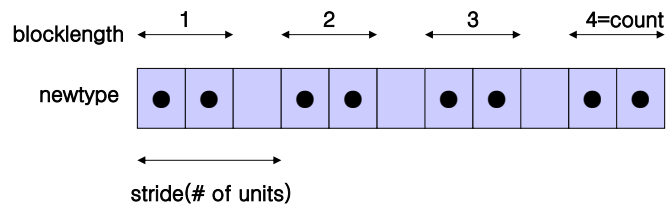
- 똑같은 간격만큼 떨어져 있는 count개 블록들로 구성되는 새로운 데이터 타입 정의
- 각 블록에는 blocklength개의 이전 타입 데이터 있음

MPL_TYPE_VECTOR (2/2)



- **count = 2**
- **stride = 5**
- **blocklength = 3**

MPI_TYPE_VECTOR 예제



● :MPIINTEGER(oldtype)

MPI_TYPE_VECTOR 예제 : Fortran

```
PROGRAM type_vector
INCLUDE 'mpif.h'
INTEGER ibuf(20), inewtype
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
IF (myrank==0) THEN
DO i=1,20
  ibuf(i) = i
ENDDO
ENDIF
CALL MPI_TYPE_VECTOR(4, 2, 3, MPI_INTEGER, inewtype, ierr)
CALL MPI_TYPE_COMMIT(inewtype, ierr)
CALL MPI_BCAST(ibuf, 1, inewtype, 0, MPI_COMM_WORLD, ierr)
PRINT *, 'ibuf =', ibuf
CALL MPI_FINALIZE(ierr)
END
```


MPL_TYPE_VECTOR 예제 : C

```

/*type_vector*/
#include <mpi.h>
#include <stdio.h>
void main (int argc, char *argv[]){
    int i, myrank, ibuf[20];
    MPI_Datatype newtype ;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    if(myrank==0) for(i=0; i<20; i++) ibuf[i]=i+1;
    else for(i=0; i<20; i++) ibuf[i]=0;
    MPI_Type_vector(4, 2, 3, MPI_INT, &newtype);
    MPI_Type_commit(&newtype);
    MPI_Bcast(ibuf, 1, newtype, 0, MPI_COMM_WORLD);
    printf("%d : ibuf =", myrank);
    for(i=0; i<20; i++) printf(" %d", ibuf[i]);
    printf("\n");
    MPI_Finalize();
}

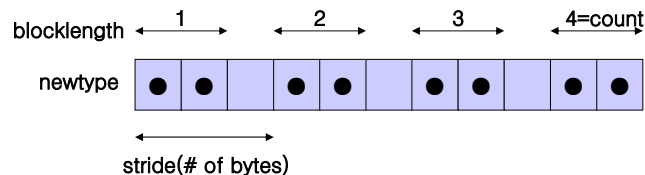
```

MPL_TYPE_HVECTOR

C	int MPI_Type_hvector (int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)
Fortran	MPI_TYPE_HVECTOR (count, blocklength, stride, oldtype, newtype, ierr)

□ stride를 바이트 단위로 표시

- bytes = stride : MPL_TYPE_HVECTOR
- bytes = stride*extent(oldtype) : MPL_TYPE_VECTOR



MPL_TYPE_STRUCT (1/3)

C	<pre>int MPI_Type_struct (int count, int *array_of_blocklengths, MPI_Aint *array_of_displacements, MPI_Datatype *array_of_type, MPI_Datatype *newtype)</pre>
Fortran	<pre>MPI_Type_struct (count, array_of_blocklengths, array_of_displacements, array_of_types, newtype, ierr)</pre>


- INTEGER count : 블록의 개수, 동시에 배열 array_of_blocklengths, array_of_displacements, array_of_types의 원소의 개수를 나타냄 (IN)
- INTEGER array_of_blocklengths(*) : 각 블록 당 데이터의 개수, array_of_blocklengths(i)는 데이터 타입이 array_of_types(i)인 i번째 블록의 데이터 개수 (IN)
- INTEGER array_of_displacements(*) : 바이트로 나타낸 각 블록의 위치 (IN)
- INTEGER array_of_types(*) : 각 블록을 구성하는 데이터 타입, i번째 블록은 데이터 타입이 array_of_types(i)인 데이터로 구성 (IN)
- INTEGER newtype : 새로운 데이터 타입 (OUT)

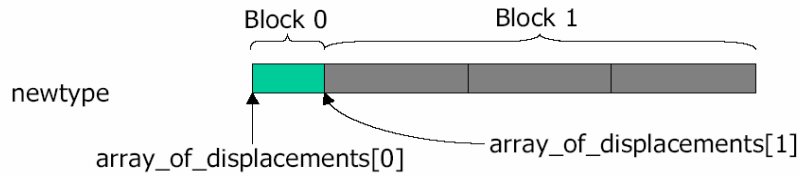
MPL_TYPE_STRUCT (2/3)

- 가장 일반적인 유도 데이터 타입
- 서로 다른 데이터 타입들로 구성된 변수 정의 가능
 - C 구조체
 - Fortran 커먼 블록
- count개 블록으로 구성된 새로운 데이터 타입 정의, i번째 블록은 데이터 타입이 array_of_types(i)인 array_of_blocklengths(i)개의 데이터로 구성되며 그 위치는 array_of_displacements(i)가 됨

MPL_TYPE_STRUCT (3/3)

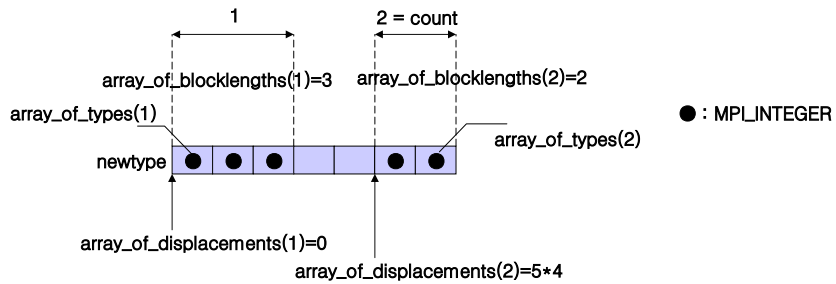
MPI_INT 

MPI_DOUBLE 

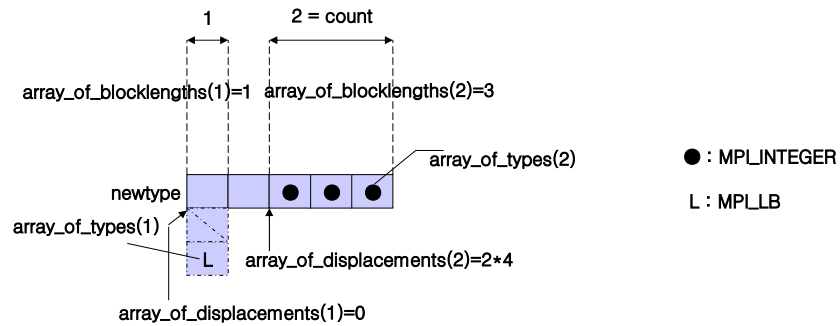


- **count = 2**
- **array_of_blocklengths = { 1, 3}**
- **array_of_types = {MPI_INT, MPI_DOUBLE}**
- **array_of_displacements = {0, extent(MPI_INT)}**

MPL_TYPE_STRUCT 예제 (1/2)



MPL_TYPE_STRUCT 예제 (2/2)



MPL_TYPE_STRUCT 예제 : Fortran (1/2)

```

PROGRAM type_struct
  INCLUDE 'mpif.h'
  INTEGER ibuf1(20), ibuf2(20)
  INTEGER iblock(2), idisp(2), itype(2)
  CALL MPI_INIT(ierr)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
  IF (myrank==0) THEN
    DO i=1,20
      ibuf1(i) = i
      ibuf2(i) = i
    ENDDO
  ENDIF
  iblock(1) = 3; iblock(2) = 2
  idisp(1) = 0; idisp(2) = 5 * 4
  itype(1) = MPI_INTEGER; itype(2) = MPI_INTEGER
  CALL MPI_TYPE_STRUCT(2, iblock, idisp, itype, newtype1, ierr)
  CALL MPI_TYPE_COMMIT(newtype1, ierr)

```

MPI_TYPE_STRUCT 예제 : Fortran (2/2)

```
CALL MPI_BCAST(ibuf1, 2, inewtype1, 0, MPI_COMM_WORLD, ierr)
PRINT *, 'Ex. 1:', ibuf1
iblock(1) = 1; iblock(2) = 3
idisp(1) = 0; idisp(2) = 2 * 4
itype(1) = MPI_LB
itype(2) = MPI_INTEGER
CALL MPI_TYPE_STRUCT(2, iblock, idisp, itype, inewtype2, ierr)
CALL MPI_TYPE_COMMIT(inewtype2, ierr)
CALL MPI_BCAST(ibuf2, 2, inewtype2, 0, MPI_COMM_WORLD, ierr)
PRINT *, 'Ex. 2:', ibuf2
CALL MPI_FINALIZE(ierr)
END
```

※ MPI_UB, MPI_LB : MPI 유사(pseudo) 타입

차지하는 공간 없이 데이터 타입의 시작, 끝에서 빈공간이 나타나도록 해야 할 때 사용

MPI_TYPE_STRUCT 예제 : C (1/3)

```
/*type_struct*/
#include <mpi.h>
#include <stdio.h>
void main (int argc, char *argv[]){
    int i, myrank ;
    int ibuf1[20], ibuf2[20], iblock[2];
    MPI_Datatype inewtype1, inewtype2;
    MPI_Datatype itype[2];
    MPI_Aint idisp[2];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    if(myrank==0)
        for(i=0; i<20; i++) {
            ibuf1[i]=i+1; ibuf2[i]=i+1;
        }
}
```

MPI_TYPE_STRUCT 예제 : C (2/3)

```
else
    for(i=0; i<20; i++){
        ibuf1[i]=0; ibuf2[i]=0;
    }
iblock[0] = 3; iblock[1] = 2;
idisp[0] = 0; idisp[1] = 5*4;
itype[0] = MPI_INT; itype[1] = MPI_INT;
MPI_Type_struct(2, iblock, idisp, itype, &inewtype1);
MPI_Type_commit(&inewtype1);
MPI_Bcast(ibuf1, 2, inewtype1, 0, MPI_COMM_WORLD);
printf("%d : Ex.1 :", myrank);
for(i=0; i<20; i++) printf(" %d", ibuf1[i]);
printf("\n");
```

MPI_TYPE_STRUCT 예제 : C (3/3)

```
iblock[0] = 1; iblock[1] = 3;
idisp[0] = 0; idisp[1] = 2*4;
itype[0] = MPI_LB; itype[1] = MPI_INT;

MPI_Type_struct(2, iblock, idisp, itype, &inewtype2);
MPI_Type_commit(&inewtype2);
MPI_Bcast(ibuf2, 2, inewtype2, 0, MPI_COMM_WORLD);

printf("%d : Ex.2 :", myrank);
for(i=0; i<20; i++) printf(" %d", ibuf2[i]);
printf("\n");

MPI_Finalize();
}
```

MPI_TYPE_EXTENT

C	<code>int MPI_Type_extent (MPI_Datatype *datatype, MPI_Aint *extent)</code>
Fortran	<code>MPI_TYPE_EXTENT (datatype, extent, ierr)</code>

INTEGER datatype : 데이터 타입(핸들) (INOUT)

INTEGER extent : 데이터 타입의 범위 (OUT)

□ 데이터 타입의 범위 = 메모리에서 차지하는 바이트 수

MPI_TYPE_EXTENT 예제 : Fortran (1/2)

```
PROGRAM structure
INCLUDE 'mpif.h'
INTEGER err, rank, num
INTEGER status(MPI_STATUS_SIZE)
REAL x
COMPLEX data(4)
COMMON /result/num,x,data
INTEGER blocklengths(3)
DATA blocklengths/1,1,4/
INTEGER displacements(3)
INTEGER types(3), restype
DATA types/MPI_INTEGER,MPI_REAL,MPI_COMPLEX/
INTEGER intex,realx
CALL MPI_INIT(err)
CALL MPI_COMM_RANK(MPI_COMM_WORLD,rank,err)
CALL MPI_TYPE_EXTENT(MPI_INTEGER,intex,err)
CALL MPI_TYPE_EXTENT(MPI_REAL,realx,err)
displacements(1)=0; displacements(2)=intex
```

MPI_TYPE_EXTENT 예제 : Fortran (2/2)

```
displacements(3)=intex+realex
CALL MPI_TYPE_STRUCT(3,blocklengths,displacements, &
    types,restype,err)
CALL MPI_TYPE_COMMIT(restype,err)
IF(rank.eq.3) THEN
    num=6; x=3.14
    DO i=1,4
        data(i)=cplx(i,i)
    ENDDO
    CALL MPI_SEND(num,1,restype,1,30,MPI_COMM_WORLD,err)
ELSE IF(rank.eq.1) THEN
    CALL MPI_RECV(num,1,restype,3,30,MPI_COMM_WORLD,status,err)
    PRINT *, 'P:',rank, ' I got'
    PRINT *, num
    PRINT *, x
    PRINT *, data
END IF
CALL MPI_FINALIZE(err)
END
```

MPI_TYPE_EXTENT 예제 : C (1/2)

```
#include <stdio.h>
#include <mpi.h>
void main(int argc, char *argv[]) {
    int rank,i;
    MPI_Status status;
    struct {
        int num; float x; double data[4];
    } a;
    int blocklengths[3]={1,1,4};
    MPI_Datatype types[3]={MPI_INT,MPI_FLOAT,MPI_DOUBLE};
    MPI_Aint displacements[3];
    MPI_Datatype restype;
    MPI_Aint intex,floatex;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Type_extent(MPI_INT,&intex);
    MPI_Type_extent(MPI_FLOAT,&floatex);
```


MPI_TYPE_EXTENT 예제 : C (2/2)

```

displacements[0]=0; displacements[1]=intex;
displacements[2]=intex+floatex;
MPI_Type_struct(3,blocklengths,displacements,types,&restype);
MPI_Type_commit(&restype);
if (rank==3){
    a.num=6; a.x=3.14;
    for(i=0;i<4;++i) a.data[i]=(double) i;
    MPI_Send(&a,1,restype,1,52,MPI_COMM_WORLD);
} else if(rank==1) {
    MPI_Recv(&a,1,restype,3,52,MPI_COMM_WORLD,&status);
    printf("P:%d my a is %d %f %lf %lf %lf %lf\n",
        rank,a.num,a.x,a.data[0],a.data[1],a.data[2],a.data[3]);
}
MPI_Finalize();
}
    
```

부분 배열의 전송 (1/2)

C	<pre> int MPI_Type_create_subarray (int ndims,int *array_of_sizes, int *array_of_subsizes, int *array_of_starts, int order, MPI_Datatype oldtype, MPI_Datatype *newtype); </pre>
Fortran	<pre> MPI_TYPE_CREATE_SUBARRAY (ndims, array_of_sizes, array_of_subsizes, array_of_starts, order, oldtype, newtype, ierr) </pre>

- INTEGER ndims : 배열의 차원(양의 정수) (IN)
- INTEGER array_of_sizes(*) : 전체 배열의 각 차원의 크기, i번째 원소는 i번째 차원의 크기(양의 정수) (IN)
- INTEGER array_of_subsizes(*) : 부분 배열의 각 차원의 크기, i번째 원소는 i번째 차원의 크기(양의 정수) (IN)
- INTEGER array_of_starts(*) : 부분 배열의 시작 좌표, i번째 원소는 i번째 차원의 시작좌표 (0부터 시작) (IN)
- INTEGER order : 배열 저장 방식(행우선 또는 열우선) 결정 (IN)
- INTEGER oldtype : 전체 배열 원소의 데이터 타입 (IN)
- INTEGER newtype : 부분배열로 구성된 새로운 데이터 타입(OUT)

부분 배열의 전송 (2/2)

- 부분 배열로 구성되는 유도 데이터 타입 생성 루틴
- **order** : 배열을 읽고 저장하는 방식 결정
 - order = MPL_ORDER_FORTRAN** : 행 우선
 - order = MPL_ORDER_C** : 열 우선

※ **MPI_TYPE_CREATE_SUBARRAY**는 MPI-2에서 지원하는 루틴으로 KISTI IBM 시스템에서 컴파일 하는 경우 “_r”을 붙일 것

```
% mpxlf90_r -o ...
```

```
% mpcc_r -o ...
```

부분 배열의 전송 예제

a(2:7,0:6)

	0	1	2	3	4	5	6
2							
3							
4							
5							
6							
7							

- **ndims = 2**
- **array_of_sizes(1) = 6; array_of_sizes(2) = 7**
- **array_of_subsizes(1) = 2; array_of_subsizes(2) = 5**
- **array_of_starts(1) = 1; array_of_starts(2) = 1**
- **order = MPI_ORDER_FORTRAN**

부분 배열의 전송 예제 : Fortran (1/2)

```
PROGRAM sub_array
INCLUDE 'mpif.h'
INTEGER ndims
PARAMETER(ndims=2)
INTEGER ibuf1(2:7,0:6)
INTEGER array_of_sizes(ndims), array_of_subsizes(ndims)
INTEGER array_of_starts(ndims)
CALL MPI_INIT(ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
DO j = 0, 6
  DO i = 2, 7
    IF (myrank==0) THEN
      ibuf1(i,j) = i
    ELSE
      ibuf1(i,j) = 0
    ENDIF
  ENDDO
ENDDO
```

부분 배열의 전송 예제 : Fortran (2/2)

```
array_of_sizes(1)=6; array_of_sizes(2)=7
array_of_subsizes(1)=2; array_of_subsizes(2)=5
array_of_starts(1)=1; array_of_starts(2)=1

CALL MPI_TYPE_CREATE_SUBARRAY(ndims, array_of_sizes, &
  array_of_subsizes, array_of_starts, MPI_ORDER_FORTRAN, &
  MPI_INTEGER, newtype, ierr)
CALL MPI_TYPE_COMMIT(newtype, ierr)
CALL MPI_BCAST(ibuf1, 1, newtype, 0, MPI_COMM_WORLD, ierr)
PRINT *, 'I am :', myrank
DO i=2,7
  PRINT *, (ibuf1(i,j), j=0,6)
ENDDO
CALL MPI_FINALIZE(ierr)
END
```

부분 배열의 전송 예제 : C (1/2)

```
#include <mpi.h>
#define ndims 2
void main(int argc, char *argv[]){
    int ibuf1[6][7];
    int array_of_sizes[ndims], array_of_subsizes[ndims],
        array_of_starts[ndims];
    int i, j, myrank;
    MPI_Datatype newtype;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    if(myrank==0) for(i=0; i<6; i++)
        for(j=0; j<7; j++) ibuf1[i][j] = i+2;
    else for(i=0; i<6; i++)
        for(j=0; j<7; j++) ibuf1[i][j] = 0 ;

    array_of_sizes[0]=6; array_of_sizes[1]=7;
    array_of_subsizes[0]=2; array_of_subsizes[1]=5;
    array_of_starts[0]=1; array_of_startst[1]=1;
```

부분 배열의 전송 예제 : C (2/2)

```
MPI_Type_create_subarray(ndims, array_of_sizes,
    array_of_subsizes, array_of_starts, MPI_ORDER_C,
    MPI_INT, &newtype);
MPI_Type_commit(&newtype);
MPI_Bcast(ibuf1, 1, newtype, 0, MPI_COMM_WORLD);
if(myrank != 0) {
    printf(" I am : %d \n ", myrank);
    for(i=0; i<6; i++) {
        for(j=0; j<7; j++) printf(" %d", ibuf1[i][j]);
        printf("\n");
    }
}
MPI_Finalize();
}
```

•프로세스 그룹 생성 : MPI_COMM_SPLIT

•가상 토폴로지



프로세스 그룹 생성 : MPI_COMM_SPLIT

C	<code>int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)</code>
Fortran	<code>MPI_COMM_SPLIT(comm, color, key, newcomm, ierr)</code>

INTEGER comm: 커뮤니케이터 (핸들) (IN)

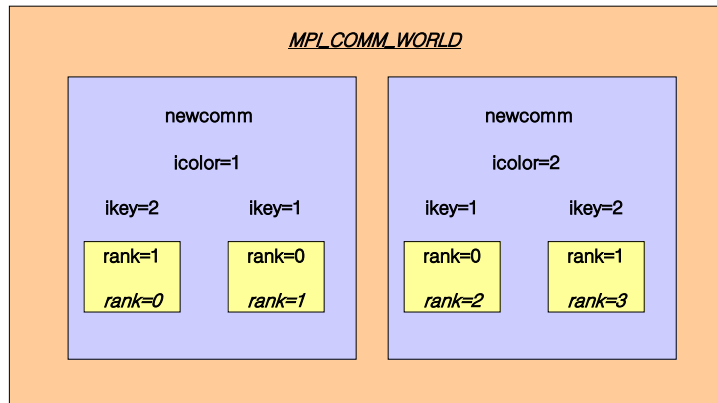
INTEGER color: 같은 color을 가지는 프로세스들을 같은 그룹에 포함 (IN)

INTEGER key: key 순서에 따라 그룹내의 프로세스에 새로운 랭크를 할당(IN)

INTEGER newcomm: 새로운 커뮤니케이터 (핸들) (OUT)

- comm내의 프로세스들을 여러 그룹으로 묶은 새로운 커뮤니케이터 newcomm 생성**
- color \geq 0**
- color = MPI_UNDEFINED \rightarrow newcomm = MPI_COMM_NULL**

MPI_COMM_SPLIT 예제



MPI_COMM_SPLIT 예제 : Fortran

```
PROGRAM comm_split
  INCLUDE 'mpif.h'
  CALL MPI_INIT(ierr)
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
  IF (myrank==0) THEN
    icolor = 1; ikey = 2
  ELSEIF (myrank==1) THEN
    icolor = 1; ikey = 1
  ELSEIF (myrank==2) THEN
    icolor = 2; ikey = 1
  ELSEIF (myrank==3) THEN
    icolor = 2; ikey = 2
  ENDIF
  CALL MPI_COMM_SPLIT(MPI_COMM_WORLD, icolor, ikey, newcomm, ierr)
  CALL MPI_COMM_SIZE(newcomm, newprocs, ierr)
  CALL MPI_COMM_RANK(newcomm, newrank, ierr)
  PRINT *, 'newcomm=', newcomm, 'newprocs=', newprocs,
    'newrank=', newrank
  CALL MPI_FINALIZE(ierr)
END
```

MPI_COMM_SPLIT 예제 : C (1/2)

```
/*comm_split*/
#include <mpi.h>
#include <stdio.h>
void main (int argc, char *argv[]){
    int i, nprocs, myrank ;
    int icolor, ikey;
    int newprocs, newrank;
    MPI_Comm newcomm;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    if(myrank == 0){
        icolor = 1; ikey = 2;
    }
    else if (myrank == 1){
        icolor = 1; ikey = 1;
    }
}
```

MPI_COMM_SPLIT 예제 : C (2/2)

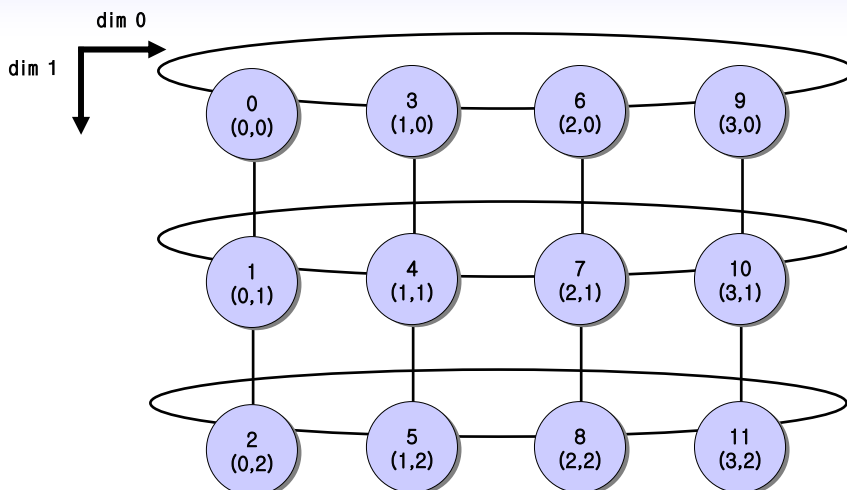
```
    else if (myrank == 2){
        icolor = 2; ikey = 1;
    }
    else if (myrank == 3){
        icolor = 2; ikey = 2;
    }
    MPI_Comm_split(MPI_COMM_WORLD, icolor, ikey, &newcomm);
    MPI_Comm_size(newcomm, &newprocs);
    MPI_Comm_rank(newcomm, &newrank);
    printf("%d", myrank);
    printf(" newcomm = %d", newcomm);
    printf(" newprocs = %d", newprocs);
    printf(" newrank = %d", newrank);
    printf("\n");

    MPI_Finalize();
}
```

가상 토폴로지 [1/2]

- 통신 패턴에 적합하도록 프로세스에 적절한 이름을 부여한 새로운 커뮤니케이터를 구성하는 것
- 코드 작성을 쉽게 하고 최적화된 통신을 가능케 함
- 직교 가상 토폴로지
 - 가상적인 그리드 상에서 각 프로세스가 인접한 이웃과 연결
 - 각 프로세스는 직교좌표 값으로 식별됨
 - 주기적 경계 조건 (periodic boundary)

가상 토폴로지 [2/2]



가상 토폴로지의 사용

- 토폴로지를 만들어 새로운 커뮤니케이터 생성
 - `MPI_CART_CREATE`

- MPI 대응 함수를 통해 토폴로지 상의 명명 방식에 근거한 프로세스 랭크 계산
 - `MPI_CART_RANK`
 - `MPI_CART_COORDS`
 - `MPI_CART_SHIFT`

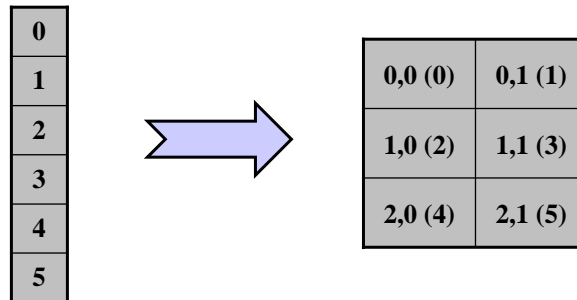
토폴로지 생성 : `MPI_CART_CREATE`

C	<code>int MPI_Cart_create(MPI_Comm oldcomm, int ndims, int *dimsizes, int *periods, int reorder, MPI_Comm *newcomm)</code>
Fortran	<code>MPI_CART_CREATE(oldcomm, ndims, dimsizes, periods, reorder, newcomm, ierr)</code>

INTEGER oldcomm: 기존 커뮤니케이터 (IN)
 INTEGER ndims: 직교좌표의 차원 (IN)
 INTEGER dimsizes(*): 각 좌표축의 길이. 크기 ndims의 배열 (OUT)
 LOGICAL periods(*): 각 좌표축의 주기성결정. 크기 ndims의 배열 (IN)
 LOGICAL reorder: MPI가 프로세스 랭크를 재 정렬할 것인가를 결정 (IN)
 INTEGER newcomm: 새로운 커뮤니케이터 (OUT)

- 가상 토폴로지의 구성을 가지는 커뮤니케이터 `newcomm` 리턴
- 인수 `reorder`가 거짓이면 기존 커뮤니케이터의 랭크를 그대로 가지고 랭크와 토폴로지 그리드 좌표사이의 대응만 설정함

MPI_CART_CREATE 예제



MPI_CART_CREATE 예제 : Fortran

```
PROGRAM cart_create
INCLUDE 'mpi.h'
INTEGER oldcomm, newcomm, ndims, ierr
INTEGER dimsize(0:1)
LOGICAL periods(0:1), reorder
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
oldcomm = MPI_COMM_WORLD
ndims = 2
dimsize(0) = 3; dimsize(1) = 2
periods(0) = .TRUE.; periods(1) = .FALSE.
reorder = .FALSE.
CALL MPI_CART_CREATE(oldcomm,ndims,dimsize,periods,reorder, newcomm,
ierr)
CALL MPI_COMM_SIZE(newcomm, newprocs, ierr)
CALL MPI_COMM_RANK(newcomm, newrank, ierr)
PRINT*,myrank, ':newcomm=',newcomm,'newprocs=',newprocs, &
'newrank=',newrank
CALL MPI_FINALIZE(ierr)
END
```

MPI_CART_CREATE 예제 : C

```

/*cart_create*/
#include <mpi.h>
#include <stdio.h>
void main (int argc, char *argv[]){
    int nprocs, myrank ;
    int ndims, newprocs, newrank;
    MPI_Comm newcomm;
    int dimsize[2], periods[2], reorder;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    ndims = 2; dimsize[0] = 3; dimsize[1] = 2;
    periods[0] = 1; periods[1] = 0; reorder = 0;
    MPI_Cart_create(MPI_COMM_WORLD, ndims, dimsize, periods, reorder,
        &newcomm);
    MPI_Comm_size(newcomm, &newprocs);
    MPI_Comm_rank(newcomm, &newrank);
    printf("%d", myrank); printf(" newcomm= %d", newcomm);
    printf(" newprocs= %d", newprocs); printf(" newrank= %d",
        newrank);
    printf("\n");
    MPI_Finalize();
}

```

대응 함수 : MPI_CART_RANK

C	int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)
Fortran	MPI_CART_RANK(comm, coords, rank, ierr)

INTEGER comm : 가상 토폴로지로 생성된 커뮤니케이터 (IN)

INTEGER coords(*) : 직교 좌표를 나타내는 크기 ndims의 배열 (IN)

INTEGER rank : coords에 의해 표현되는 프로세스의 랭크 (OUT)

- 프로세스 직교 좌표를 대응하는 프로세스 랭크로 나타냄
- 좌표를 알고있는 경우 그 좌표에 해당하는 프로세스와의 통신을 위해 사용

MPI_CART_RANK 예제 : Fortran

```
...
CALL MPI_CART_CREATE(oldcomm, ndims, dimsize, periods, reorder,
                    newcomm, ierr)
...
IF (myrank == 0) THEN
  DO i = 0, dimsize(0)- 1
    DO j = 0, dimsize(1)- 1
      coords(0) = i
      coords(1) = j
      CALL MPI_CART_RANK(newcomm, coords, rank, ierr)
      PRINT *, 'coords =', coords, 'rank =' rank
    ENDDO
  ENDDO
ENDIF
...
END
```

※MPI_CART_CREATE 예제에 첨부

MPI_CART_RANK 예제 : C

```
...
MPI_Cart_create(oldcomm,ndims,dimsize,periods,reorder, &newcomm);
if(myrank == 0) {
  for(i=0; i<dimsize[0]; i++){
    for(j=0; j<dimsize[1]; j++){
      coords[0] = i;
      coords[1] = j;
      MPI_Cart_rank(newcomm, coords, &rank);
      printf("coords = %d, %d, rank = %d\n", coords[0],
            coords[1], rank);
    }
  }
}
...

```

※MPI_CART_CREATE 예제에 첨부

대응 함수 : MPL_CART_COORDS

C	<code>int MPI_Cart_coords(MPI_Comm comm, int rank, int ndims, int *coords)</code>
Fortran	<code>MPI_CART_COORDS(comm, rank, ndims, coords, ierr)</code>

INTEGER comm : 가상 토폴로지로 생성된 커뮤니케이터 (IN)

INTEGER rank : 루틴을 호출한 프로세스의 랭크 (IN)

INTEGER ndims : 직교 좌표의 차원 (IN)

INTEGER coords(*) : 랭크에 대응하는 직교 좌표 (IN)

- 프로세스 랭크를 대응하는 직교좌표로 나타냄
- MPL_CART_RANK의 역함수

MPL_CART_COORDS 예제 : Fortran

```

...
CALL MPI_CART_CREATE(oldcomm, ndims, dimsize, periods,
  reorder, newcomm, ierr)
...
IF (myrank == 0) THEN
  DO rank = 0, nprocs - 1
    CALL MPI_CART_COORDS(newcomm,rank,ndims,coords,ierr)
    PRINT *, , 'rank =', rank, 'coords =', coords
  ENDDO
ENDIF
...
END

```

※MPI_CART_CREATE 예제에 첨부

MPI_CART_COORDS 예제 : C

```

...
MPI_Cart_create(oldcomm, ndims, dimsize, periods, reorder,
                &newcomm);
if(myrank == 0) {
    for(rank=0; rank<nprocs; rank++){
        MPI_Cart_coords(newcomm, rank, ndims, coords);
        printf("rank = %d, coords = %d, %d\n", rank,
               coords[0], coords[1]);
    }
}
...

```

※MPI_CART_CREATE 예제에 첨부

대응 함수 : MPI_CART_SHIFT

C	<code>int MPI_Cart_shift(MPI_Comm comm, int direction, int displ, int *source, int *dest)</code>
Fortran	<code>MPI_CART_SHIFT(comm, direction, displ, source, dest, ierr)</code>

INTEGER comm : 가상 토폴로지로 생성된 커뮤니케이터 (IN)

INTEGER direction : 시프트할 방향 (IN)

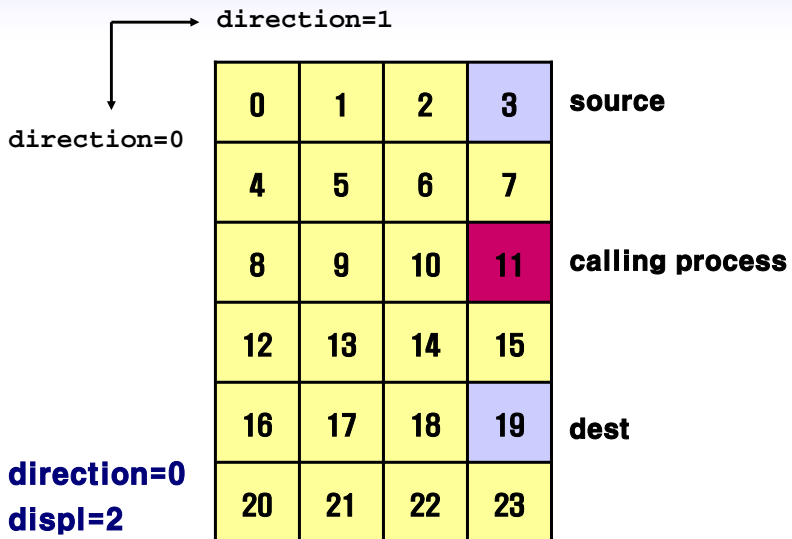
INTEGER displ : 프로세스 좌표상의 시프트할 거리 (+/-) (IN)

INTEGER source : direction 방향으로 displ 떨어진 거리에 있는 프로세스, displ > 0 일 때 직교 좌표가 작아지는 방향의 프로세스 랭크(OUT)

INTEGER dest : direction 방향으로 displ 떨어진 거리에 있는 프로세스, displ > 0 일 때 직교 좌표가 커지는 방향의 프로세스 랭크(OUT)

- 실제 시프트를 실행하는 것은 아님
- 특정 방향을 따라 루틴을 호출한 프로세스의 이웃 프로세스의 랭크를 발견하는데 사용

MPI_CART_SHIFT 예제



MPI_CART_SHIFT 예제 : Fortran

```

...
ndims = 2
dimsize(0) = 6; dimsize(1) = 4
periods(0) = .TRUE.; periods(1) = .TRUE.
reorder = .TRUE.
CALL MPI_CART_CREATE(oldcomm,ndims,dimsize,periods,reorder, &
    newcomm, ierr)
CALL MPI_COMM_RANK(newcomm, newrank, ierr)
CALL MPI_CART_COORDS(newcomm, newrank, ndims, coords, ierr)
direction=0
displ=2
CALL MPI_CART_SHIFT(newcomm, direction, displ, source, dest,
    ierr)
PRINT *, 'myrank =', newrank, 'coords=', coords
PRINT *, 'source =', source, 'dest =', dest
...

```

※MPI_CART_CREATE 예제에 첨부

MPL_CART_SHIFT 예제 : C

```

...
ndims = 2;
dimsize[0] = 6; dimsize[1] = 4;
periods[0] = 1; periods[1] = 1;
reorder = 1;
MPI_Cart_create(MPI_COMM_WORLD, ndims, dimsize, periods,
                reorder, &newcomm);
MPI_Comm_rank(newcomm, &newrank);
MPI_Cart_coords(newcomm, newrank, ndims, coords);
direction=0; displ=2;
MPI_Cart_shift(newcomm, direction, displ, &source, &dest);
printf(" myrank= %d, coords= %d, %d \n ", newrank, coords[0],
        coords[1]);
printf("source= %d, dest= %d \n", source, dest);
...

```

※MPI_CART_CREATE 예제에 첨부

토폴로지 분해 : MPI_CART_SUB

C	int MPI_Cart_sub(MPI_Comm oldcomm, int *belongs, MPI_Comm *newcomm)
Fortran	MPI_CART_SUB(oldcomm, belongs, newcomm, ierr)

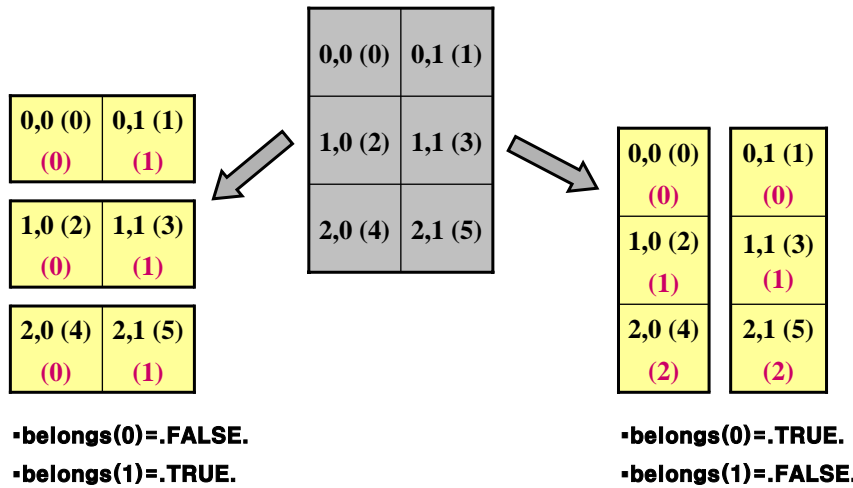
INTEGER oldcomm : 가상 토폴로지로 생성된 커뮤니케이터 (IN)

LOGICAL belongs(*) : 토폴로지 상에서 해당 좌표축 방향으로의 분해여부를 나타내는 ndims 크기의 배열(IN)

INTEGER newcomm : 토폴로지를 분해한 새로운 커뮤니케이터 (OUT)

- 직교 토폴로지를 축 방향으로 분해하여 여러 개의 서브토폴로지로 구성되는 새로운 커뮤니케이터 생성
- 토폴로지 상의 특정 행 또는 열 들에 대해서만 통신이 필요한 경우 사용
- MPL_COMM_SPLIT과 유사

MPI_CART_SUB 예제



MPI_CART_SUB 예제 : Fortran

```

...
ndims = 2
dimsz(0) = 3; dimsz(1) = 2
CALL MPI_CART_CREATE(oldcomm, ndims, dimsz, periods, &
  reorder, newcomm, ierr)
CALL MPI_COMM_RANK(newcomm, newrank, ierr)
CALL MPI_CART_COORDS(newcomm, newrank, ndims, coords,
  ierr)
belongs(0)=.FALSE.; belongs(1)=.TRUE.
CALL MPI_CART_SUB(newcomm, belongs, commrow, ierr)
CALL MPI_comm_rank(commrow, rank, ierr)
PRINT *, ' myrank = ', newrank, ' coords = ', coords
PRINT *, ' commrow = ', commrow
PRINT *, ' rank = ',
※ MPI_CART_CREATE 예제에 첨부
...

```

MPI_CART_SUB 예제 : C

```
...
ndims = 2;
dimsize[0] = 3; dimsize[1] = 2;
MPI_Cart_create(MPI_COMM_WORLD, ndims, dimsize, periods,
                reorder, &newcomm);
MPI_Comm_rank(newcomm, &newrank);
MPI_Cart_coords(newcomm, newrank, ndims, coords);
belongs[0]=0; belongs[1]=1;
MPI_Cart_sub(newcomm, belongs, &commrow);
MPI_Comm_rank(commrow, &rank);
printf("myrank= %d, coords= %d,%d \n", newrank,
        coords[0], coords[1]);
printf("commrow = %d \n", commrow);
printf(" rank= %d \n", rank);
...
```

※MPI_CART_CREATE 예제에 첨부

제 3 장 MPI를 이용한 병렬 프로그래밍 실제

2 장의 내용을 기본으로 MPI를 이용한 병렬
프로그램 작성시 염두에 두어야 할 데이터 처
리 방식과, 프로그래밍 테크닉, 주의할 점
등에 대해 알아본다.

• 병렬 프로그램의 입출력

• DO 루프의 병렬화

• 블록 분할

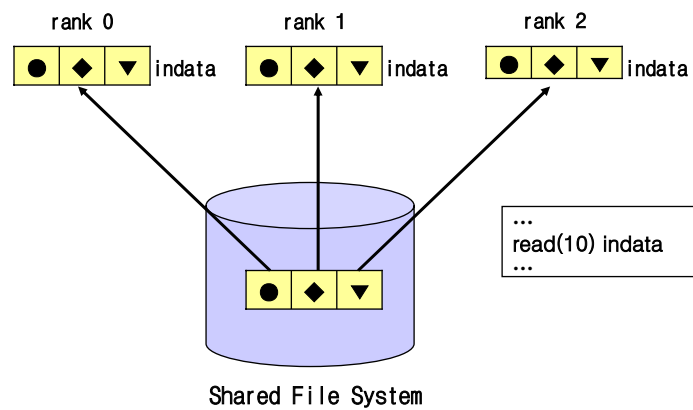
• 순환 분할

• 배열 수축



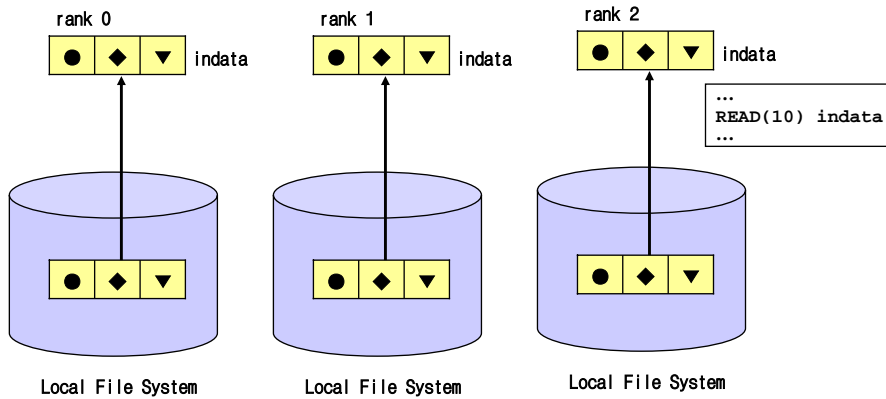
병렬 프로그램의 입력 (1/4)

□ 공유 파일 시스템으로부터 동시에 읽어오기



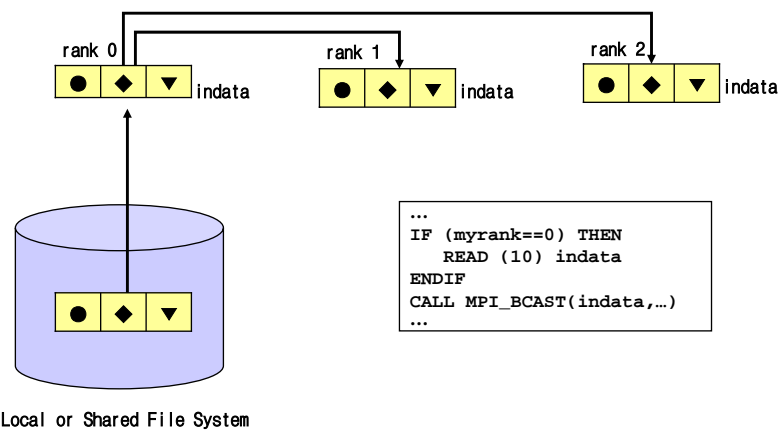
병렬 프로그램의 입력 (2/4)

- 입력파일의 복사본을 각각 따로 가지는 경우



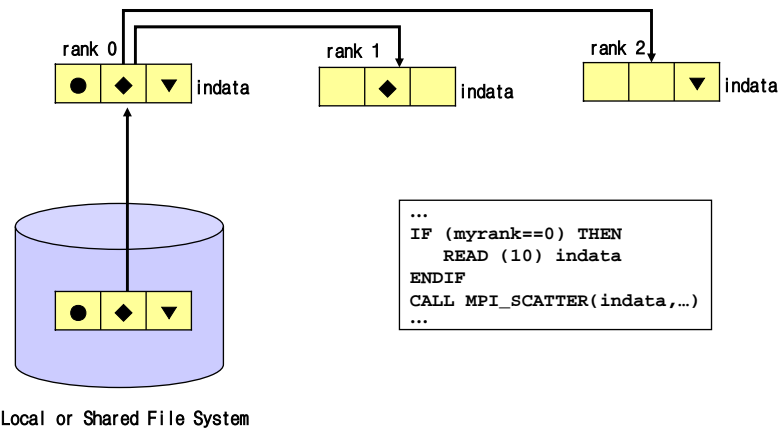
병렬 프로그램의 입력 (3/4)

- 한 프로세스가 입력파일을 읽어 다른 프로세스에 전달 1.



병렬 프로그램의 입력 (4/4)

- 한 프로세스가 입력파일을 읽어 다른 프로세스에 전달 2.



병렬 프로그램의 출력 (1/3)

- 표준 출력

```
print *, 'I am :', myrank, 'Hello world!' ...
```

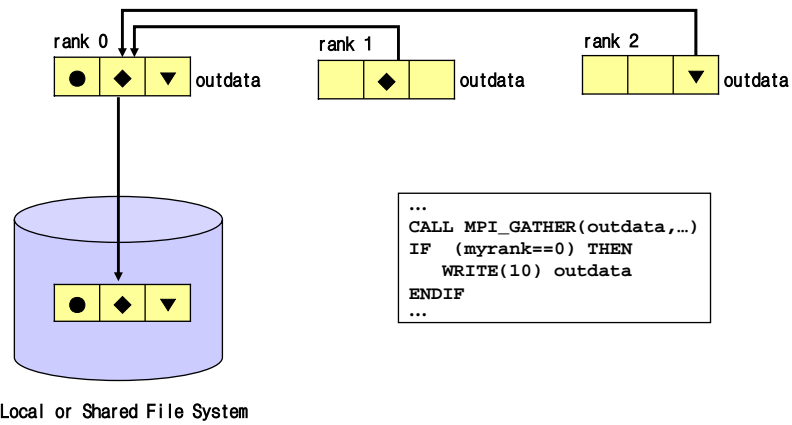
- 모든 프로세스들이 출력 (IBM 환경변수)
 - MP_STDOUTMODE = unordered (:디폴트)
 - MP_STDOUTMODE = ordered

```
if(myrank==0) then
  print *, 'I am :', myrank, 'Hello
world!'
endif
```

- 원하는 프로세스만 출력 (IBM 환경변수)
 - MP_STDOUTMODE = rank_id

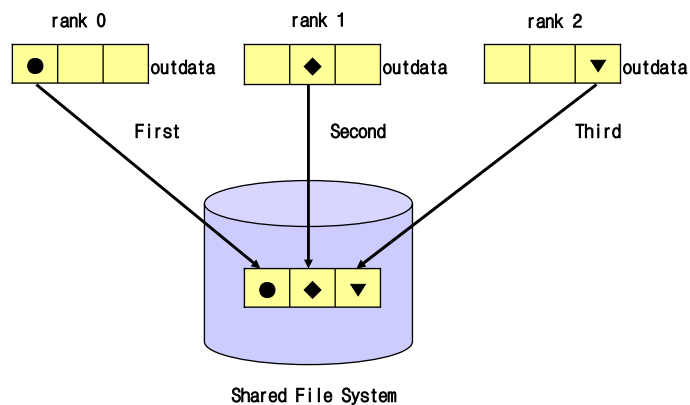
병렬 프로그램의 출력 (2/3)

- 한 프로세스가 데이터를 모아 로컬 파일시스템에 저장



병렬 프로그램의 출력 (3/3)

- 각 프로세스가 공유 파일 시스템에 순차적으로 저장



DO 루프의 병렬화

- 루프 내에서 반복되는 계산의 할당 문제는 루프 인덱스와 관련된 인덱스를 가지는 배열의 할당 문제가 됨
 - 배열을 어떻게 나눌 것인가?
 - 나눠진 배열과 관련 계산을 어떻게 효율적으로 할당할 것인가?

배열의 분할

- 블록 분할(Block Distribution)

iteration	1	2	3	4	5	6	7	8	9	10	11	12
rank	0	0	0	1	1	1	2	2	2	3	3	3

- 순환 분할(Cyclic Distribution)

iteration	1	2	3	4	5	6	7	8	9	10	11	...
rank	0	1	2	3	0	1	2	3	0	1	2	...

- 블록-순환 분할(Block-Cyclic Distribution)

iteration	1	2	3	4	5	6	7	8	9	10	11	...
rank	0	0	1	1	2	2	3	3	0	0	1	...

블록 분할 (1/3)

□ $n = p \times q + r$

- n : 반복 계산 회수
- p : 프로세스 개수
- q : n 을 p 로 나눈 몫
- r : n 을 p 로 나눈 나머지

□ r 개 프로세스에 $q+1$ 번, 나머지 프로세스에 q 번의 계산 할당

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

Iteration	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

블록 분할 (2/3)

□ 블록 분할 코드 예 : Fortran

```

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 (iwork2 > irank) iend = iend + 1
END
    
```

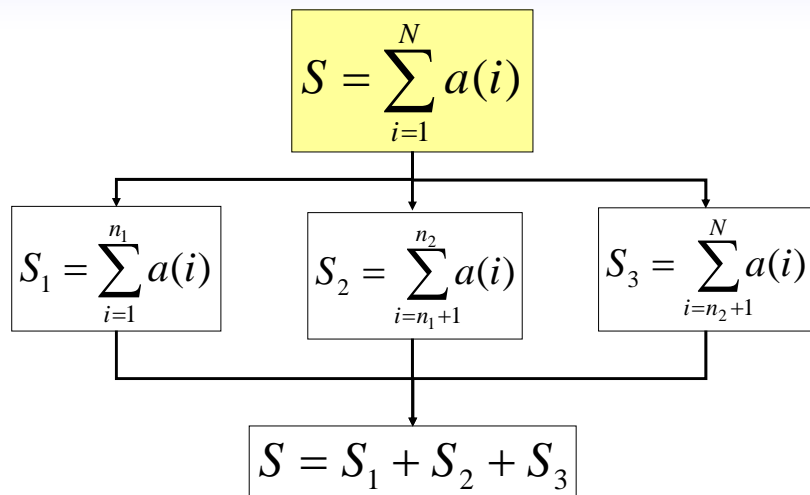
- $n1$ 부터 $n2$ 까지 반복되는 루프 계산을 $nprocs$ 개 프로세스에 블록 분할을 이용해 할당하는 서브루틴
- 프로세스 $irank$ 가 $ista$ 부터 $iend$ 까지 계산을 할당 받음

블록 분할 (3/3)

□ 블록 분할 코드 예 : C

```
void para_range(int n1,int n2, int nprocs, int myrank, int
*ista, int *iend){
    int iwork1, iwork2;
    iwork1 = (n2-n1+1)/nprocs;
    iwork2 = (n2-n1+1) % nprocs;
    *ista= myrank*iwork1 + n1 + min(myrank, iwork2);
    *iend = *ista + iwork1 - 1;
    if(iwork2 > myrank) *iend = *iend + 1;
}
```

블록 분할 예제



블록 분할 예제 : 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)
```

블록 분할 예제 : C (1/2)

```
/*parallel_main*/
#include <mpi.h>
#include <stdio.h>
#define n 100000
void para_range(int, int, int, int, int*, int*);
int min(int, int);
void main (int argc, char *argv[]){
  int i, nprocs, myrank ;
  int ista, iend;
  double a[n], sum, tmp;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  para_range(1, n, nprocs, myrank, &ista, &iend);
  for(i = ista-1; i<iend; i++) a[i] = i+1;
  sum = 0.0;
  for(i = ista-1; i<iend; i++) sum = sum + a[i];
```

블록 분할 예제 : C (2/2)

```

MPI_Reduce(&sum, &tmp, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);
sum = tmp;
if(myrank == 0) printf(" sum = %f \n", sum);
MPI_Finalize();
}

int min(int x, int y){
int v;
if (x>=y) v = y;
else v = x;
return v;
}

void para_range(int n1,int n2, int nprocs, int myrank, int
*ista, int *iend){
...
}

```

순환 분할

```

DO i = n1, n2      DO i = n1+myrank, n2, nprocs
  computation      computation
ENDDO              ENDDO

```

Iteration	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Rank	0	1	2	3	0	1	2	3	0	1	2	3	0	1

- 블록 분할보다 효과적인 로드밸런싱
- 블록 분할보다 캐시미스 발생이 많아짐

블록-순환 분할

```
DO ii = n1+myrank*iblock, n2, nprocs*iblock
  DO i = ii, MIN(ii+iblock-1, n2)
    computation
  ENDDO
ENDDO
```

Iteration	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Rank	0	0	1	1	2	2	3	3	0	0	1	1	2	2

←————→
iblock

순환 분할, 블록-순환 분할 예제

- 블록 분할 예제
 - 순환 분할
 - 블록-순환 분할

배열 수축 (1/4)

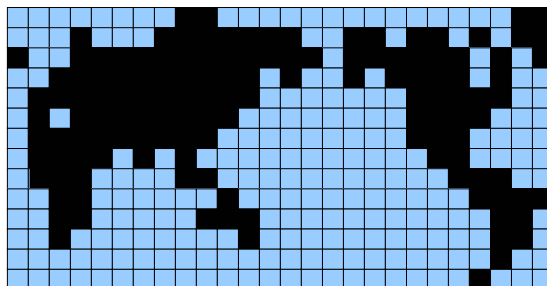
- 병렬화 작업에 참여하는 프로세스는 전체 배열을 가져올 필요 없이 자신이 계산을 담당할 부분의 데이터만 메모리에 가져와 계산을 수행하면 됨
 - 다른 프로세스의 데이터가 필요하면 통신을 통해 송/수신
- n개의 프로세서를 연결한 분산메모리 시스템은 1개의 프로세서를 가진 시스템보다 n배의 메모리를 사용할 수 있음
- 사용자는 분산 메모리 시스템에서의 병렬화 작업을 통하여 처리하는 데이터 크기를 증가 시킬 수 있게 됨

→ 배열 수축(Shrinking Arrays) 기술

배열 수축 (2/4)

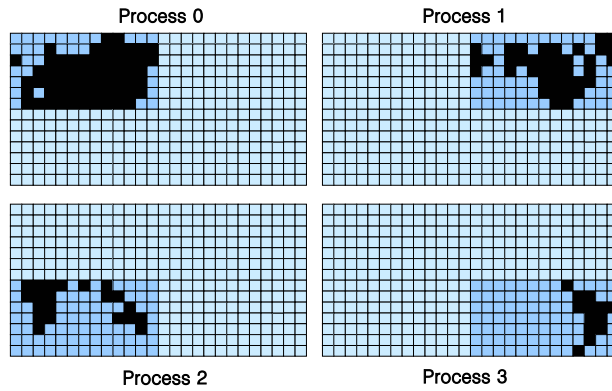
- 순차 실행

$a(i,j)$

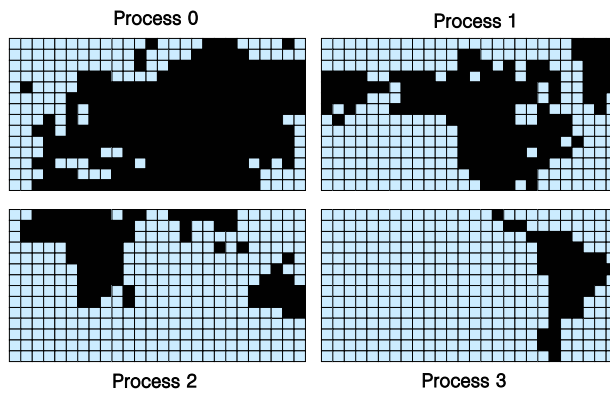


배열 수축 (3/4)

□ 병렬 실행



배열 수축 (4/4)



메모리 동적 할당 예제 : Fortran (1/2)

```
PROGRAM dynamic_alloc
INCLUDE 'mpif.h'
PARAMETER (n1 = 1, n2 = 1000)
REAL, ALLOCATABLE :: a(:)
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(n1, n2, nprocs, myrank, ista, iend)
ALLOCATE (a(ista:iend))
DO i = ista, iend
  a(i) = i
ENDDO
sum = 0.0
DO i = ista, iend
  sum = sum + a(i)
ENDDO
DEALLOCATE (a)
```

메모리 동적 할당 예제 : Fortran (2/2)

```
CALL MPI_REDUCE(sum, ssum, 1, MPI_REAL, MPI_SUM, 0, &
               MPI_COMM_WORLD, ierr)
sum = ssum
PRINT *, 'sum = ', sum
CALL MPI_FINALIZE(ierr)
END

SUBROUTINE para_range( ... )
...
```

메모리 동적 할당 예제 : C (1/2)

```
/*dynamic_alloc*/
#include <mpi.h>
#include <stdio.h>
#define n 1000
void para_range(int, int, int, int, int*, int*);
int min(int, int);
void main (int argc, char *argv[]){
    int i, nprocs, myrank ;
    int ista, iend, diff;
    double sum, tmp;
    double *a;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    para_range(0, n-1, nprocs, myrank, &ista, &iend);
    diff = iend-ista+1;
    a = (double *)malloc(diff*sizeof(double));
```

메모리 동적 할당 예제 : C (2/2)

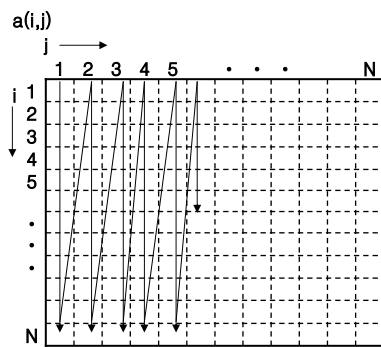
```
for(i = ista-1; i<iend; i++) a[i] = i+1;
sum = 0.0;
for(i = ista-1; i<iend; i++) sum = sum + a[i];
free(a);
MPI_Reduce(&sum, &tmp, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);
sum = tmp;
if(myrank == 0) {
    printf(" sum = %f \n", sum);
}
MPI_Finalize();
}
```


- 내포된 루프의 병렬화
- 캐시미스 줄이기
- 통신량 줄이기

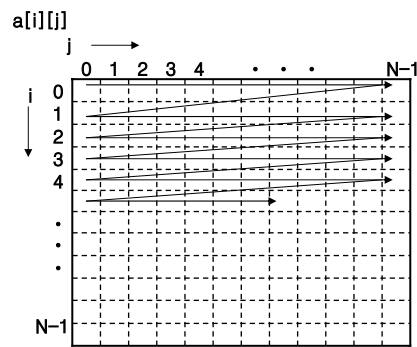


내포된 루프의 병렬화

- 캐시미스와 통신의 최소화에 유념할 것
- 2차원 배열의 메모리 저장 방식



(a) Fortran



(b) C

캐시미스 줄이기 (1/3)

- 메모리 저장 방식에 의한 캐시미스 차이

Loop A (column-major)	Loop B (row-major)
<pre>DO j = 1, n DO i = 1, n a(i,j) = b(i,j) + c(i,j) ENDDO ENDDO</pre>	<pre>DO i = 1, n DO j = 1, n a(i,j) = b(i,j) + c(i,j) ENDDO ENDDO</pre>

Fortran은 루프 B에서 더 많은 캐시미스가 발생하므로, 루프 A가 더 빠르게 실행 됨

→ 루프 A의 병렬화

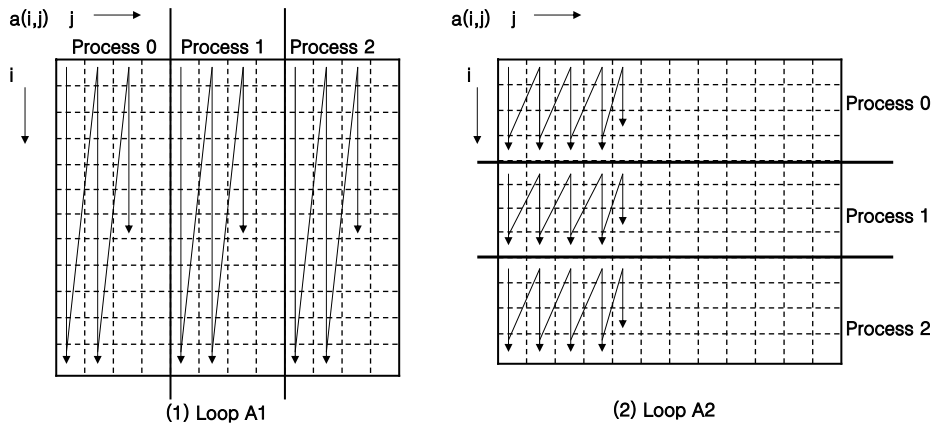
캐시미스 줄이기 (2/3)

- 바깥쪽 루프와 안쪽 루프의 병렬화에 따른 캐시미스 차이

Loop A1 (바깥쪽 병렬화)	Loop A2 (안쪽 병렬화)
<pre>DO j = jsta, jend DO i = 1, n a(i,j) = b(i,j) + c(i,j) ENDDO ENDDO</pre>	<pre>DO j = 1, n DO i = ista, iend a(i,j) = b(i,j) + c(i,j) ENDDO ENDDO</pre>

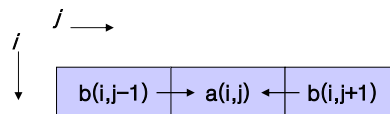
Fortran은 루프 A2에서 더 많은 캐시미스가 발생하므로, 루프 A1이 더 빠르게 실행 됨(다음 장 그림 참조)

캐시미스 줄이기 (3/3)



통신량 줄이기 (1/6)

- 한 방향으로 필요한 데이터를 통신해야 하는 경우



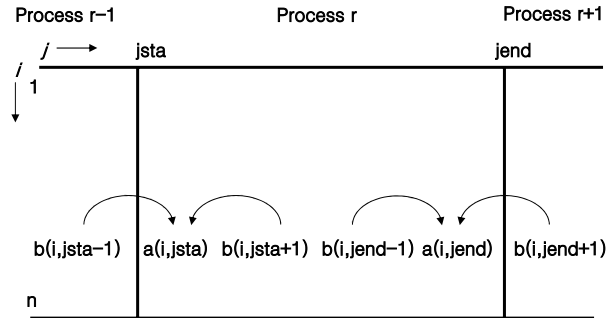
Loop C

```

DO j = 1, n
  DO i = 1, n
    a(i,j) = b(i, j-1) + b(i, j+1)
  ENDDO
ENDDO
    
```

통신량 줄이기 [2/6]

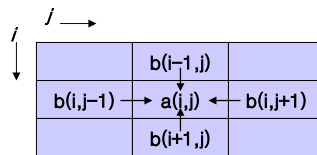
- 바깥쪽 루프(열 방향) 병렬화에 의해 요구되는 통신



- 이 경우 안쪽 루프(행 방향)의 병렬화는 통신 필요 없음

통신량 줄이기 [3/6]

- 양 방향으로 필요한 데이터를 통신해야 하는 경우



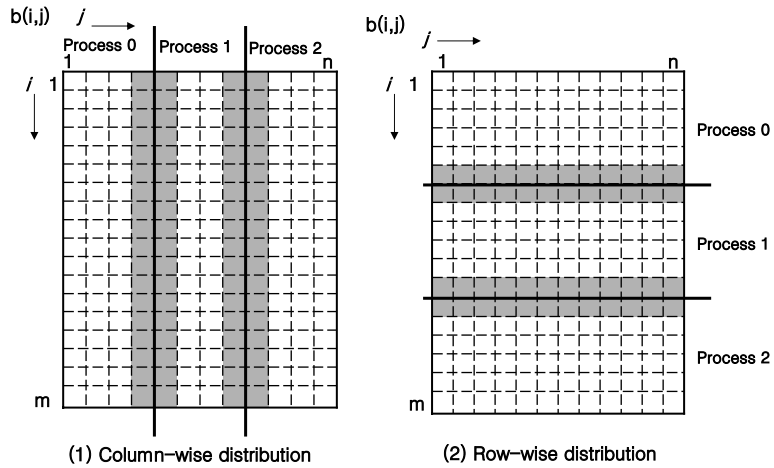
Loop D

```

DO j = 1, n
  DO i = 1, m
     $a(i, j) = b(i-1, j) + b(i, j-1) + b(i, j+1) + b(i+1, j)$ 
  ENDDO
ENDDO
  
```

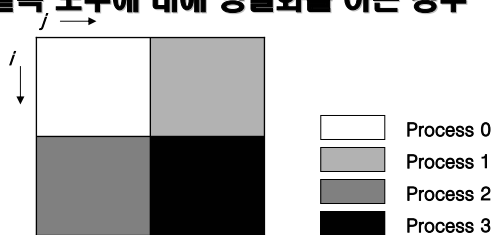
통신량 줄이기 (4/6)

□ m, n 의 크기에 의존



통신량 줄이기 (5/6)

□ 안쪽과 바깥쪽 모두에 대해 병렬화를 하는 경우

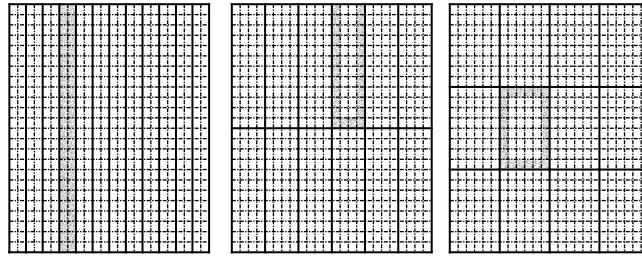


Loop E

```
DO j = jsta, jend
  DO i = ista, iend
    a(i,j) = b(i-1,j) + b(i,j-1) + b(i,j+1) + b(i+1,j)
  ENDDO
ENDDO
```

통신량 줄이기 [6/6]

- 프로세스의 개수는 합성수여야 함
- 블록의 형태가 정사각형에 가까울수록 통신량이 감소



- 다른 프로세스의 데이터 참조
- 1차원 유한 차분법의 병렬화
- 대량 데이터 전송
- 데이터 동기화



외부 데이터 참조의 간단한 예

- 병렬화 과정에서 다른 프로세스가 가진 데이터를 참조해야 하는 경우

```

...
REAL a(9), b(9)
...
DO i = 1, 9
  a(i) = i
ENDDO
DO i = 1, 9
  b(i) = b(i)*a(1)
ENDDO
...

```

```

...
REAL a(9), b(9)
...
DO i = ista, iend
  a(i) = i
ENDDO
CALL MPI_BCAST(a(1),1,MPI_REAL,0, &
  MPI_COMM_WORLD, ierr)
DO i = ista, iend
  b(i) = b(i)*a(1)
ENDDO
...

```

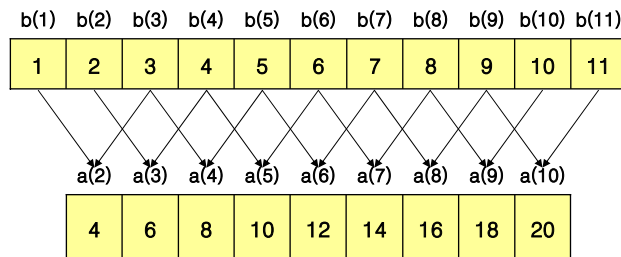
1차원 유한 차분법 (1/3)

- 1차원 유한 차분법(FDM)의 핵심부 : 순차 프로그램

Fortran	C
<pre> PROGRAM 1D_fdm_serial IMPLICIT REAL*8 (a-h, o-z) PARAMETER (n=11) DIMENSION a(n), b(n) DO i = 1, n b(i) = i ENDDO DO i = 2, n-1 a(i) = b(i-1) + b(i+1) ENDDO END </pre>	<pre> /*1D_fdm_serial*/ #define n 11 main(){ double a[n], b[n]; int i; for(i=0; i<n; i++) b[i] = i+1; for(i=1; i<n-1; i++) a[i] = b[i-1] + b[i+1]; } </pre>

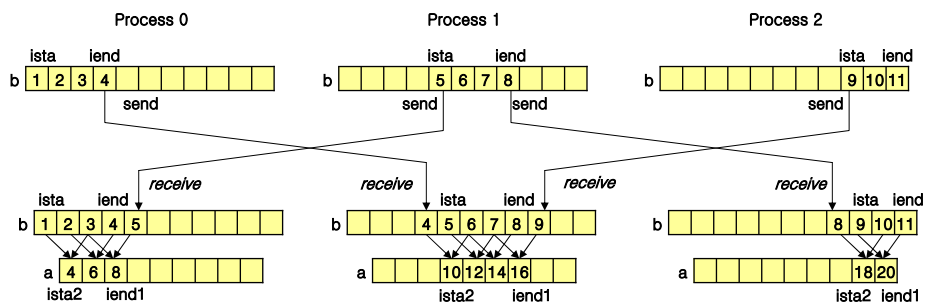
1차원 유한 차분법 (2/3)

□ 1차원 FDM의 데이터 의존성



1차원 유한 차분법 (3/3)

□ 병렬화된 1차원 FDM의 데이터 전송



※ 2차원 FDM은 2차원 배열의 경계선, 3차원 FDM은 3차원 배열의 경계면 전송

1차원 FDM 병렬화 코드 : Fortran (1/2)

```
PROGRAM parallel_1D_fdm
INCLUDE 'mpif.h'
PARAMETER (n=11)
DIMENSION a(n), b(n)
INTEGER istatus(MPI_STATUS_SIZE)
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)
ista2 = ista; iend1 = iend
IF (myrank == 0) ista2 = 2
IF (myrank == nprocs-1) iend1 = n-1
inext = myrank + 1; iprev = myrank - 1
IF (myrank == nprocs-1) inext = MPI_PROC_NULL
IF (myrank == 0) iprev = MPI_PROC_NULL

DO i = ista, iend
    b(i) = i
ENDDO
```

1차원 FDM 병렬화 코드 : Fortran (2/2)

```
CALL MPI_ISEND(b(iend), 1, MPI_REAL, inext, 1, &
    MPI_COMM_WORLD, isend1, ierr)
CALL MPI_ISEND(b(ista), 1, MPI_REAL, iprev, 1, &
    MPI_COMM_WORLD, isend2, ierr)
CALL MPI_IRECV(b(ista-1), 1, MPI_REAL, iprev, 1, &
    MPI_COMM_WORLD, irecv1, ierr)
CALL MPI_IRECV(b(iend+1), 1, MPI_REAL, inext, 1, &
    MPI_COMM_WORLD, irecv2, ierr)
CALL MPI_WAIT(isend1, istatus, ierr)
CALL MPI_WAIT(isend2, istatus, ierr)
CALL MPI_WAIT(irecv1, istatus, ierr)
CALL MPI_WAIT(irecv2, istatus, ierr)

DO i = ista2, iend1
    a(i) = b(i-1) + b(i+1)
ENDDO
CALL MPI_FINALIZE(ierr)
END
```

1차원 FDM 병렬화 코드 : C (1/2)

```
/*parallel_1D_fdm*/
#include <mpi.h>
#define n 11
void para_range(int, int, int, int, int*, int*);
int min(int, int);
main(int argc, char *argv[]){
    int i, nprocs, myrank ;
    double a[n], b[n];
    int ista, iend, ista2, iend1, inext, iprev;
    MPI_Request isend1, isend2, irecv1, irecv2;
    MPI_Status istatus;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    para_range(0, n-1, nprocs, myrank, &ista, &iend);
    ista2 = ista; iend1 = iend;
    if(myrank==0) ista2=1;
    if(myrank==nprocs-1)iend1=n-2;
```

1차원 FDM 병렬화 코드 : C (2/2)

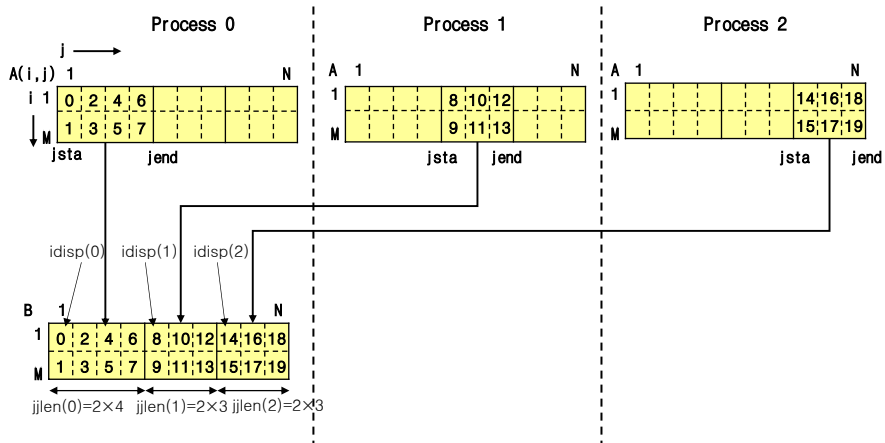
```
inext=myrank+1; iprev=myrank-1;
for(i=ista; i<=iend; i++) b[i]=i+1;
if(myrank==nprocs-1) inext=MPI_PROC_NULL;
if(myrank==0) iprev=MPI_PROC_NULL;
MPI_Isend(&b[iend], 1, MPI_DOUBLE, inext, 1, MPI_COMM_WORLD,
&isend1);
MPI_Isend(&b[ista], 1, MPI_DOUBLE, iprev, 1, MPI_COMM_WORLD,
&isend2);
MPI_Irecv(&b[ista-1], 1, MPI_DOUBLE, iprev, 1, MPI_COMM_WORLD,
&irecv1);
MPI_Irecv(&b[iend+1], 1, MPI_DOUBLE, inext, 1, MPI_COMM_WORLD,
&irecv2);
MPI_Wait(&isend1, &istatus);
MPI_Wait(&isend2, &istatus);
MPI_Wait(&irecv1, &istatus);
MPI_Wait(&irecv2, &istatus);
for(i=ista2; i<=iend1; i++) a[i] = b[i-1] + b[i+1];
MPI_Finalize();
}
```

대량 데이터 전송

- 각 프로세스의 모든 데이터를 한 프로세스로 취합
 1. 연속 데이터 : 송/수신 버퍼가 중복되지 않는 경우
 2. 연속 데이터 : 송/수신 버퍼가 중복되는 경우
 3. 불연속 데이터 : 송/수신 버퍼가 중복되는 경우
 4. 불연속 데이터 : 송/수신 버퍼가 중복되지 않는 경우
- 모든 프로세스가 가진 전체 배열을 동시에 갱신
 1. 송/수신 버퍼가 중복되지 않는 경우
 2. 송/수신 버퍼가 중복되는 경우
- 블록 분할된 데이터의 전환(행 방향 ↔ 열 방향)과 재할당

연속 데이터 취합 : 버퍼 중복 없음 (1/3)

- 송/수신 버퍼가 중복되지 않는 경우 : Fortran



연속 데이터 취합 : 버퍼 중복 없음 (2/3)

□ 병렬화 코드 : Fortran

```
REAL a(m,n), b(m,n)
INTEGER, ALLOCATABLE :: idisp(:), jjlen(:)
...
ALLOCATE (idisp(0:nprocs-1), jjlen(0:nprocs-1))
DO irank = 0, nprocs - 1
    CALL para_range(1, n, nprocs, irank, jsta, jend)
    jjlen(irank) = m * (jend - jsta + 1)
    idisp(irank) = m * (jsta - 1)
ENDDO
CALL para_range(1, n, nprocs, myrank, jsta, jend)
...
CALL MPI_GATHERV(a(1,jsta), jjlen(myrank), MPI_REAL, &
    b, jjlen, idisp, MPI_REAL, 0, MPI_COMM_WORLD, ierr)
DEALLOCATE (idisp, jjlen)
...
```

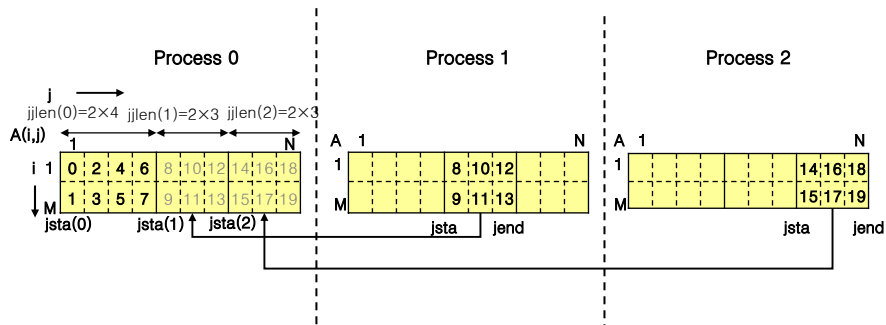
연속 데이터 취합 : 버퍼 중복 없음 (3/3)

□ 병렬화 코드 : C

```
double a[m][n], b[m][n];
int *idisp, *iilen;
...
idisp = (int *)malloc(nprocs*sizeof(int));
iilen = (int *)malloc(nprocs*sizeof(int));
for(irank = 0; irank<nprocs; irank++){
    para_range(0, m-1, nprocs, irank, &ista, &iend);
    iilen[irank] = n*(iend-ista+1);
    idisp[irank] = n*ista;
}
para_range(0, m-1, nprocs, myrank, &ista, &iend);
...
MPI_Gatherv(&a[ista][0], iilen[myrank], MPI_DOUBLE, b, iilen,
    idisp, MPI_DOUBLE, 0, MPI_COMM_WORLD);
free(idisp); free(iilen);
...
```

연속 데이터 취합 : 버퍼 중복 (1/5)

- 송/수신 버퍼가 중복되는 경우 : Fortran



연속 데이터 취합 : 버퍼 중복 (2/5)

- 병렬화 코드 : Fortran

```

REAL a(m,n)
INTEGER, ALLOCATABLE :: jjsta(:), jjlen(:), iireq(:)
INTEGER istatus(MPI_STATUS_SIZE)
...
ALLOCATE (jjsta(0:nprocs-1))
ALLOCATE (jjlen(0:nprocs-1))
ALLOCATE (iireq(0:nprocs-1))
DO irank = 0, nprocs - 1
    CALL para_range(1, n, nprocs, irank, jsta, jend)
    jjsta(irank) = jsta
    jjlen(irank) = m * (jend - jsta + 1)
ENDDO
CALL para_range(1, n, nprocs, myrank, jsta, jend)
...
    
```

연속 데이터 취합 : 버퍼 중복 (3/5)

□ 병렬화 코드 : Fortran (계속)

```
IF (myrank == 0) THEN
  DO irank = 1, nprocs - 1
    CALL MPI_Irecv(a(1,jjsta(irank)),jjlen(irank),MPI_REAL,&
                  irank, 1, MPI_COMM_WORLD, iireq(irank),ierr)
  ENDDO
  DO irank = 1, nprocs - 1
    CALL MPI_WAIT(iireq(irank), istatus, ierr)
  ENDDO
ELSE
  CALL MPI_ISEND(a(1,jsta), jjlen(myrank), MPI_REAL, &
                0, 1, MPI_COMM_WORLD, ireq, ierr)
  CALL MPI_WAIT(ireq, istatus, ierr)
ENDIF
DEALLOCATE (jjsta, jjlen, iireq)
...
```

연속 데이터 취합 : 버퍼 중복 (4/5)

□ 병렬화 코드 : C

```
...
double a[m][n];
int *iista, *iilen;
MPI_Request *iireq;
MPI_Status istatus;
...
iista = (int *)malloc(nprocs*sizeof(int));
iilen = (int *)malloc(nprocs*sizeof(int));
iireq = (int *)malloc(nprocs*sizeof(int));
for(irank = 0; irank<nprocs; irank++){
  para_range(0, m-1, nprocs, irank, &ista, &iend);
  iista[irank] = ista;
  iilen[irank] = n*(iend-ista+1);
}
para_range(0, m-1, nprocs, myrank, &ista, &iend);
```

연속 데이터 취합 : 버퍼 중복 (5/5)

□ 병렬화 코드 : C (계속)

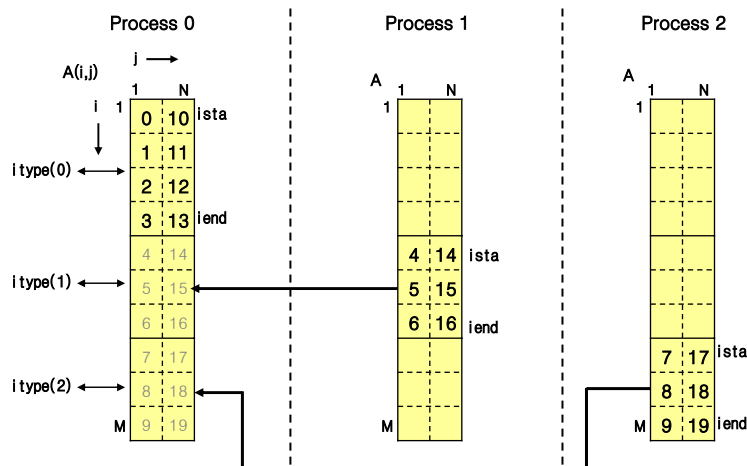
```

...
if (myrank == 0) {
    for(irank = 1; irank<nprocs; irank++)
        MPI_Irecv(&a[0][iista[irank]], iilen[irank],
                 MPI_DOUBLE, irank, 1, MPI_COMM_WORLD, &iireq[irank]);
    for(irank = 1; irank<nprocs; irank++)
        MPI_Wait(&iireq[irank], &istatus);
}
else {
    MPI_Isend(&a[0][iista], iilen[myrank], MPI_DOUBLE, 0, 1,
             MPI_COMM_WORLD, &iireq);
    MPI_Wait(&iireq, &istatus);
}
free(iista); free(iilen); free(iireq);
...

```

불연속 데이터 취합 : 버퍼 중복 (1/5)

□ 송/수신 버퍼가 중복되는 경우 : Fortran



불연속 데이터 취합 : 버퍼 중복 (2/5)

□ 병렬화 코드 : Fortran

```
REAL a(m,n)
PARAMETER(ndims=2)
INTEGER sizes(ndims), subsizes(ndims), starts(ndims)
INTEGER, ALLOCATABLE :: itype(:), iireq(:)
INTEGER istatus(MPI_STATUS_SIZE)
...
ALLOCATE (itype(0:nprocs-1), iireq(0:nprocs-1))
sizes(1)=m; sizes(2)=n
DO irank = 0, nprocs - 1
  CALL para_range(1, m, nprocs, irank, ista, iend)
  subsizes(1) = iend-ista+1; subsizes(2) = n
  starts(1) = ista-1; starts(2) = 0
  CALL MPI_TYPE_CREATE_SUBARRAY(ndims, sizes, subsizes, &
    starts, MPI_ORDER_FORTRAN, MPI_REAL, itype(irank), ierr)
  CALL MPI_TYPE_COMMIT(itype(irank), ierr)
ENDDO
```

불연속 데이터 취합 : 버퍼 중복 (3/5)

□ 병렬화 코드 : Fortran (계속)

```
CALL para_range(1, m, nprocs, myrank, ista, iend)
...
IF (myrank == 0) THEN
  DO irank = 1, nprocs - 1
    CALL MPI_Irecv(a, 1, itype(irank), irank, &
      1, MPI_COMM_WORLD, iireq(irank), ierr)
  ENDDO
  DO irank = 1, nprocs - 1
    CALL MPI_WAIT(iireq(irank), istatus, ierr)
  ENDDO
ELSE
  CALL MPI_Isend(a, 1, itype(myrank), 0, 1, MPI_COMM_WORLD, &
    ireq, ierr)
  CALL MPI_WAIT(ireq, istatus, ierr)
ENDIF
DEALLOCATE (itype, iireq)
...
```


불연속 데이터 취합 : 버퍼 중복 (4/5)

□ 병렬화 코드 : C

```
#define ndims 2
...
double a[m][n]; MPI_Datatype *itype; MPI_Request *iireq;
int sizes[ndims], subsizes[ndims], starts[ndims];
MPI_Status istatus;
...
itype = (int *)malloc(nprocs*sizeof(int));
iireq = (int *)malloc(nprocs*sizeof(int));
sizes[0]=m; sizes[1]=n;
for(irank = 0; irank<nprocs; irank++){
    para_range(0, n-1, nprocs, irank, &jsta, &jend);
    subsizes[0]= m; subsizes[1] = jend-jsta+1;
    starts[0] = 0; starts[1] = jsta;
    MPI_Type_create_subarray(ndims, sizes, subsizes, starts,
                             MPI_ORDER_C, MPI_DOUBLE,
    &itype[irank]);
    MPI_Type_commit(&itype[irank]);
}
```

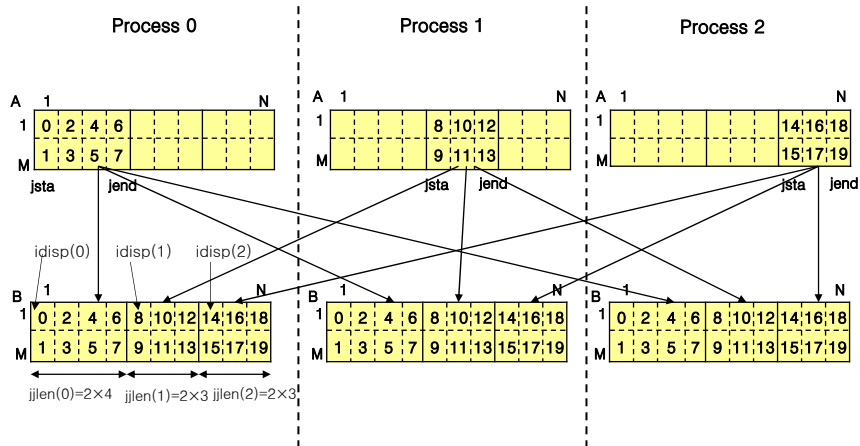
불연속 데이터 취합 : 버퍼 중복 (5/5)

□ 병렬화 코드 : C (계속)

```
para_range(1, n, nprocs, myrank, &jsta, &jend);
...
if (myrank == 0) {
    for(irank = 1; irank<nprocs; irank++)
        MPI_Irecv(a, 1, itype[irank], irank, 1, MPI_COMM_WORLD,
                 &iireq[irank]);
    for(irank = 1; irank<nprocs; irank++)
        MPI_Wait(&iireq[irank], &istatus);
}
else {
    MPI_Isend(a, 1, itype[myrank], 0, 1, MPI_COMM_WORLD,
             &iireq);
    MPI_Wait(&iireq, &istatus);
}
free(itype); free(iireq);
...
```

데이터 동기화 : 버퍼 중복 없음 (1/3)

□ 송/수신 버퍼가 중복되지 않는 경우 : Fortran



데이터 동기화 : 버퍼 중복 없음 (2/3)

□ 병렬화 코드 : Fortran

```

REAL a(m,n), b(m,n)
INTEGER, ALLOCATABLE :: idisp(:), jjlen(:)
...
ALLOCATE (idisp(0:nprocs-1), jjlen(0:nprocs-1))
DO irank = 0, nprocs - 1
  CALL para_range(1, n, nprocs, irank, jsta, jend)
  jjlen(irank) = m * (jend - jsta + 1)
  idisp(irank) = m * (jsta - 1)
ENDDO
CALL para_range(1, n, nprocs, myrank, jsta, jend)
...
CALL MPI_ALLGATHERV(a(1,jsta), jjlen(myrank), MPI_REAL, &
  b, jjlen, idisp, MPI_REAL, MPI_COMM_WORLD, ierr)
DEALLOCATE (idisp, jjlen)
...
    
```


데이터 동기화 : 버퍼 중복 (2/3)

□ 병렬화 코드 : Fortran

```
REAL a(m,n)
INTEGER, ALLOCATABLE :: jjsta(:), jjlen(:)
...
ALLOCATE (jjsta(0:nprocs-1), jjlen(0:nprocs-1))
DO irank = 0, nprocs - 1
    CALL para_range(1, n, nprocs, irank, jsta, jend)
    jjsta(irank) = jsta
    jjlen(irank) = m * (jend - jsta + 1)
ENDDO
CALL para_range(1, n, nprocs, myrank, jsta, jend)
...
DO irank = 0, nprocs - 1
    CALL MPI_BCAST(a(1,jjsta(irank)), jjlen(irank), MPI_REAL, &
        irank, MPI_COMM_WORLD, ierr)
ENDDO
DEALLOCATE (jjsta, jjlen)
...
```

데이터 동기화 : 버퍼 중복 (3/3)

□ 병렬화 코드 : C

```
double a[m][n];
int *iista, *iilen ;
...
iista = (int *)malloc(nprocs*sizeof(int));
iilen = (int *)malloc(nprocs*sizeof(int));
for(irank = 0; irank<nprocs; irank++){
    para_range_sta(0, m-1, nprocs, irank, &iista, &iend);
    iista[irank] = iista;
    iilen[irank] = n*(iend-iista+1);
}
para_range_sta(0, m-1, nprocs, myrank, &iista, &iend);
...
for(irank = 0; irank<nprocs; irank++){
    MPI_BCAST(&a[iista[irank]][0], iilen[irank], MPI_DOUBLE,
        irank, MPI_COMM_WORLD);
}
free(iista); free(iilen);
...
```


블록 분할의 전환 (2/8)

- 유도 데이터 타입 정의 :

MPL_TYPE_CREATE_SUBARRAY

A(i,j)	1	j →	N
1	itype(0,0)	itype(0,1)	itype(0,2)
i ↓	itype(1,0)	itype(1,1)	itype(1,2)
M	itype(2,0)	itype(2,1)	itype(2,2)

블록 분할의 전환 (3/8)

- 병렬화 코드 : Fortran

```
...  
PARAMETER (m=7, n=8)  
PARAMETER (ncpu=3)  
PARAMETER(ndims=2)  
REAL a(m,n)  
INTEGER sizes(ndims), subsizes(ndims), starts(ndims)  
INTEGER itype(0:ncpu-1, 0:ncpu-1)  
INTEGER ireq1(0:ncpu-1), ireq2(0:ncpu-1)  
INTEGER istatus(MPI_STATUS_SIZE)  
  
sizes(1)=m  
sizes(2)=n  
CALL MPI_INIT(ierr)  
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)  
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
```

블록 분할의 전환 (4/8)

□ 병렬화 코드 : Fortran (계속)

```
DO jrank = 0, nprocs-1
  CALL para_range(1, n, nprocs, jrank, jsta, jend)
  DO irank = 0, nprocs-1
    CALL para_range(1, m, nprocs, irank, ista, iend)
    subsizes(1) = iend-ista+1; subsizes(2) = jend-jsta+1
    starts(1) = ista-1; starts(2) = jsta-1;
    CALL MPI_TYPE_CREATE_SUBARRAY(ndims, sizes, subsizes, &
      starts, MPI_ORDER_FORTRAN, MPI_REAL, &
      itype(irank,jrank), ierr)
    CALL MPI_TYPE_COMMIT(itype(irank,jrank), ierr)
  ENDDO
ENDDO
CALL para_range(1, m, nprocs, myrank, ista, iend)
CALL para_range(1, n, nprocs, myrank, jsta, jend)
...
```

블록 분할의 전환 (5/8)

□ 병렬화 코드 : Fortran (계속)

```
DO irank = 0, nprocs-1
  IF (irank /= myrank) THEN
    CALL MPI_ISEND(a, 1, itype(irank, myrank), irank, 1, &
      MPI_COMM_WORLD, ireq1(irank), ierr)
    CALL MPI_IRECV(a, 1, itype(myrank, irank), irank, 1, &
      MPI_COMM_WORLD, ireq2(irank), ierr)
  ENDIF
ENDDO
DO irank = 0, nprocs-1
  IF (irank /= myrank) THEN
    CALL MPI_WAIT(ireq1(irank), istatus, ierr)
    CALL MPI_WAIT(ireq2(irank), istatus, ierr)
  ENDIF
ENDDO
...
```

블록 분할의 전환 (6/8)

□ 병렬화 코드 : C

```
#define ndims 2
#define ncpu 3
#define m 7
#define n 8
...
double a[m][n];
int *itype;
int sizes[ndims], subsizes[ndims], starts[ndims];
MPI_Request ireq1[ncpu], ireq2[ncpu];
MPI_Datatype itype[ncpu][ncpu];
MPI_Status istatus;
sizes[0]=m; sizes[1]=n;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
```

블록 분할의 전환 (7/8)

□ 병렬화 코드 : C (계속)

```
for(jrank = 0; jrank<nprocs; jrank++){
    para_range(0, n-1, nprocs, jrank, &jsta, &jend);
    for(irank = 0; irank<nprocs; irank++){
        para_range(0, m-1, nprocs, irank, &ista, &iend);
        subsizes[0] = iend-ista+1; subsizes[1] = jend-jsta+1;
        starts[0] = ista; starts[1] = jsta;
        MPI_Type_create_subarray(ndims, sizes, subsizes, starts,
                                MPI_ORDER_C, MPI_DOUBLE,
                                itype[irank][jrank]);
        MPI_Type_commit(&itype[irank][jrank]);
    }
}
para_range(0, m-1, nprocs, myrank, &ista, &iend);
para_range(0, n-1, nprocs, myrank, &jsta, &jend);
...
```


블록 분할의 전환 (8/8)

□ 병렬화 코드 : C (계속)

```
for(irank = 0; irank<nprocs; irank++){
    if (irank != myrank) {
        MPI_Isend(a, 1, itype[irank][myrank], irank, 1,
                 MPI_COMM_WORLD, &ireq1[irank]);
        MPI_Irecv(a, 1, itype[myrank][irank], irank, 1,
                 MPI_COMM_WORLD, &ireq2[irank]);
    }
}
for(irank = 0; irank<nprocs; irank++){
    if (irank != myrank) {
        MPI_Wait(&ireq1[irank], &istatus);
        MPI_Wait(&ireq2[irank], &istatus);
    }
}
...
```

중첩 (1/3)

□ 환산 연산을 이용한 데이터 취합

- 각 프로세스가 모든 데이터를 가지고 있으며(배열 수축 없음)
- 불연속적인 데이터에 대한 계산 수행 후
- 그 결과를 한 프로세스로 모으고자 할 때

→ 계산을 수행한 불연속 데이터를 제외한 모든 데이터를 0
으로 두고 환산 연산 수행

중첩 (2/3)

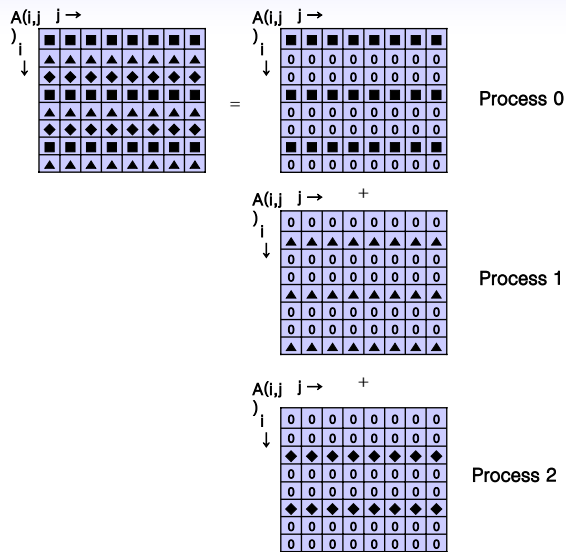
□ Fortran 코드

```

REAL a(n,n), aa(n,n)
...
DO j = 1, n
  DO i = 1, n
    a(i,j) = 0.0
  ENDDO
ENDDO
DO j = 1, n
  DO i = 1 + myrank, n, nprocs
    a(i,j) = computation
  ENDDO
ENDDO
CALL MPI_REDUCE(a, aa, n*n, MPI_REAL, MPI_SUM, 0, &
  MPI_COMM_WORLD, ierr)
...

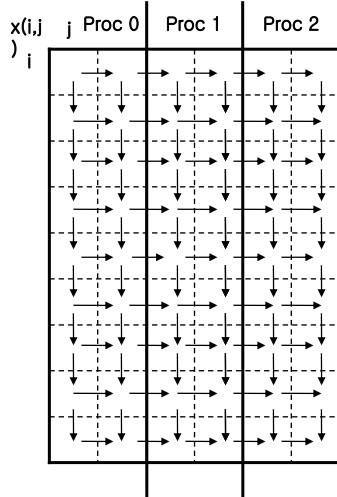
```

중첩 (3/3)



파이프라인 방법 (1/9)

□ 실행에 의존성을 가지는 루프 1.

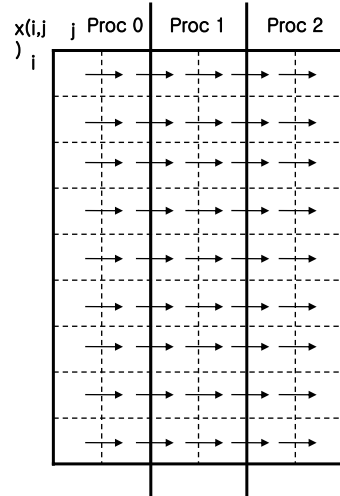


```

PROGRAM main
PARAMETER (mx = ..., my = ...)
DIMENSION x(0:mx, 0:my)
...
DO j = 1, my
  DO i = 1, mx
    x(i,j) = x(i,j)+x(i-1,j)+x(i,j-1)
  ENDDO
ENDDO
...
    
```

파이프라인 방법 (2/9)

□ 실행에 의존성을 가지는 루프 2.

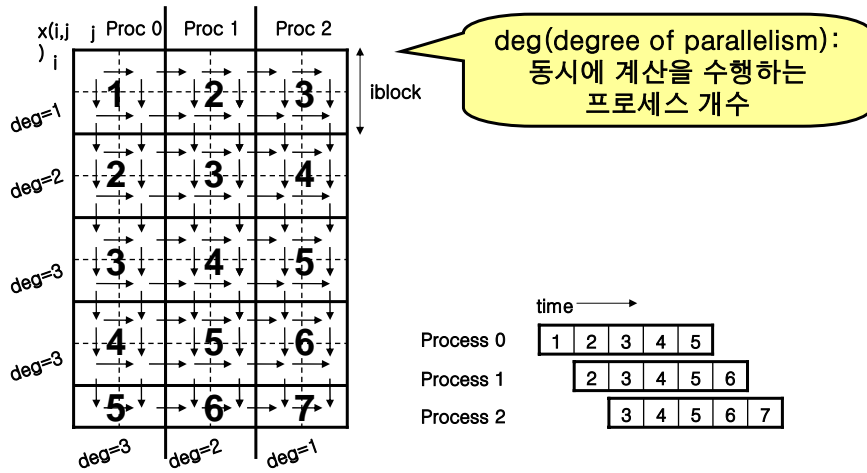


```

PROGRAM main2
PARAMETER (mx = ..., my = ...)
DIMENSION x(0:mx, 0:my)
...
DO j = 1, my
  DO i = 1, mx
    x(i,j) = x(i,j) + x(i,j-1)
  ENDDO
ENDDO
...
    
```

파이프라인 방법 (3/9)

□ 의존성을 가지는 루프의 병렬화 : Fortran



파이프라인 방법 (4/9)

□ 의존성을 가지는 루프의 병렬화 코드 : Fortran

```
PROGRAM main_pipe
INCLUDE 'mpif.h'
PARAMETER (mx=..., my=...)
DIMENSION x(0:mx, 0:my)
INTEGER istatus(MPI_STATUS_SIZE)
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, my, nprocs, myrank, jsta, jend)
inext = myrank + 1
IF (inext == nprocs) inext = MPI_PROC_NULL
iprev = myrank - 1
IF (iprev == -1) iprev = MPI_PROC_NULL
iblock = 2
```

파이프라인 방법 (5/9)

□ 의존성을 가지는 루프의 병렬화 코드 : Fortran (계속)

```
DO ii = 1, mx, iblock
  iblklen = MIN(iblock, mx-ii+1)
  CALL MPI_Irecv (x(ii, jsta-1), iblklen, MPI_REAL, iprev, 1, &
    MPI_COMM_WORLD, ireqr, ierr)
  CALL MPI_WAIT(ireqr, istatus, ierr)
  DO j = jsta, jend
    DO i = ii, ii+iblklen-1
      x(i,j) = x(i,j) + x(i-1,j) + x(i,j-1)
    ENDDO
  ENDDO
  CALL MPI_ISEND (x(ii, jend), iblklen, MPI_REAL, inext, 1, &
    MPI_COMM_WORLD, ireqs, ierr)
  CALL MPI_WAIT(ireqs, istatus, ierr)
ENDDO
...
```

파이프라인 방법 (6/9)

□ 의존성을 가지는 루프의 병렬화 코드 : C

```
/* main_pipe */
...
main(int argc, char *argv){
  ...
  double x[mx+1][my+1];
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  para_range(1, mx, nprocs, myrank, &ista, &iend);
  inext = myrank + 1;
  if (inext == nprocs) inext = MPI_PROC_NULL;
  iprev = myrank - 1;
  if (iprev == -1) iprev = MPI_PROC_NULL;
  jblock = 2;
}
```

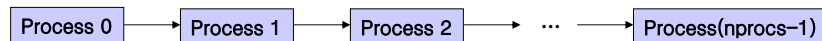
파이프라인 방법 (7/9)

□ 의존성을 가지는 루프의 병렬화 코드 : C (계속)

```
for(jj=1; jj<=my; jj+=jblock){
    jblklen = min(jblock, my-jj+1);
    MPI_Irecv(&x[ista-1][jj], jblklen, MPI_DOUBLE, iprev, 1,
             MPI_COMM_WORLD, &ireqr);
    MPI_Wait(&ireqr, &istatus);
    for(i=ista; i<=iend; i++){
        for(j=jj; j<=jj+jblklen-1; j++){
            if((i-1)==0) x[i-1][j]=0.0;
            if((j-1)==0) x[i][j-1]=0.0;
            x[i][j] = x[i][j] + x[i-1][j] + x[i][j-1];
        }
        MPI_Isend(&x[iend][jj], jblklen, MPI_DOUBLE, inext, 1,
                MPI_COMM_WORLD, &ireqs);
        MPI_Wait(&ireqs, &istatus);
    }
    ...
}
```

파이프라인 방법 (8/9)

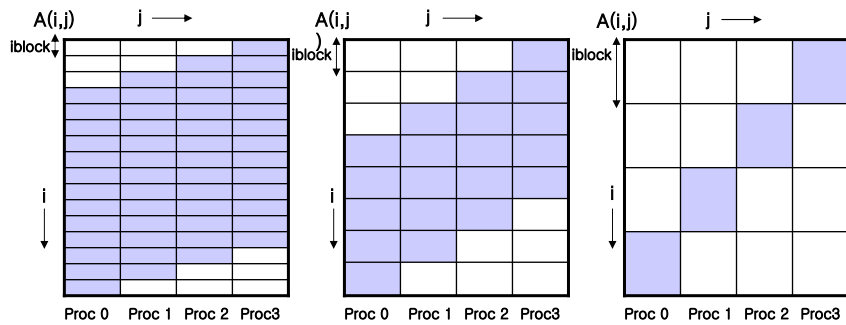
□ 데이터의 흐름 : 교착에 빠지지 않음



파이프라인 방법 (9/9)

□ 블록(block/jblock) 크기 결정

- 동시 수행되는 부분이 많을수록 통신 부담이 커지게 됨



비틀림 분해 (1/14)

□ 항상 모든 프로세스가 참여하는 병렬화 가능

- 파이프라인 방법보다 성능면에서 유리
- 효과적인 로드밸런싱
- 데이터 분배가 복잡해 코드 작성이 어려움

비틀림 분해 (2/14)

□ 비틀림 분해 구현의 예 : Fortran

```

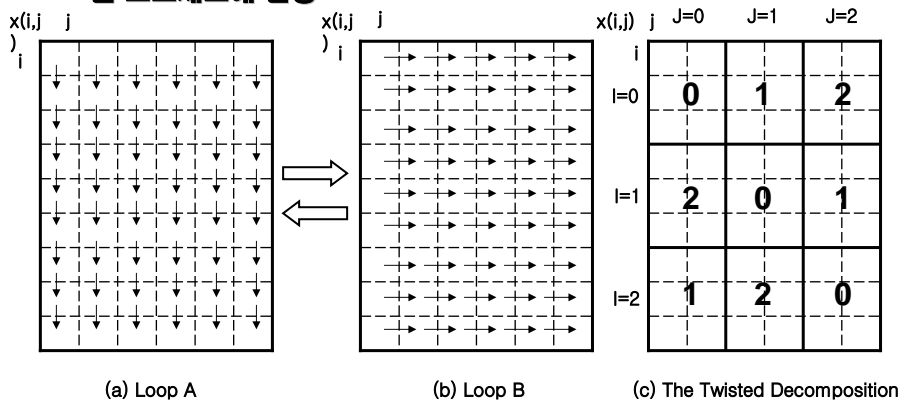
...
! Loop A
DO j = 1, my
  DO i = 1, mx
    x(i,j) = x(i,j) + x(i-1,j)
  ENDDO
ENDDO

! Loop B
DO j = 1, my
  DO i = 1, mx
    x(i,j) = x(i,j) + x(i,j-1)
  ENDDO
ENDDO
...

```

비틀림 분해 (3/14)

1. 행과 열을 각각 프로세스 개수(nprocs)의 블록으로 분해
2. 각 블록의 위치를 좌표 (i, j)로 표현
3. (i, j)블록의 계산을 (j-i+nprocs)를 nprocs로 나눈 나머지에 해당되는 프로세스에 할당



비틀림 분해(4/14)

□ 비틀림 분해를 이용한 병렬화 코드 : Fortran

```
PROGRAM main_twist
INCLUDE 'mpif.h'
INTEGER istatus(MPI_STATUS_SIZE)
INTEGER, ALLOCATABLE :: is(:), ie(:), js(:), je(:)
PARAMETER (mx=..., my=..., m=...)
DIMENSION x(0:mx, 0:my)
DIMENSION bufs(m), bufr(m)

CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
ALLOCATE (is(0:nprocs-1), ie(0:nprocs-1))
ALLOCATE (js(0:nprocs-1), je(0:nprocs-1))
```

비틀림 분해 (5/14)

□ 비틀림 분해를 이용한 병렬화 코드 : Fortran (계속)

```
DO ix = 0, nprocs-1
CALL para_range(1, mx, nprocs, ix, is(ix), ie(ix))
CALL para_range(1, my, nprocs, ix, js(ix), je(ix))
ENDDO
inext = myrank + 1
IF (inext == nprocs) inext = 0
iprev = myrank - 1
IF (iprev == -1) iprev = nprocs-1
...
! Loop A
DO ix = 0, nprocs-1
iy = MOD(ix+myrank, nprocs)
ista = is(ix); iend = ie(ix); jsta = js(iy); jend = je(iy)
jlen = jend - jsta + 1
```

비틀림 분해 (6/14)

□ 비틀림 분해를 이용한 병렬화 코드 : Fortran (계속)

```
IF (ix /= 0) THEN
  CALL MPI_IRECV (bufr(jsta), jlen, MPI_REAL, inext, 1, &
    MPI_COMM_WORLD, ireqr, ierr)
  CALL MPI_WAIT(ireqr, istatus, ierr)
  CALL MPI_WAIT(ireqs, istatus, ierr)
  DO j = jsta, jend
    x(ista-1,j) = bufr(j)
  ENDDO
ENDIF
DO j = jsta, jend
  DO i = ista, iend
    x(i,j) = x(i,j) + x(i-1,j)
  ENDDO
ENDDO
```

비틀림 분해 (7/14)

□ 비틀림 분해를 이용한 병렬화 코드 : Fortran (계속)

```
IF (ix /= nprocs-1) THEN
  DO j = jsta, jend
    bufs(j) = x(iend,j)
  ENDDO
  CALL MPI_ISEND (bufs(jsta), jlen, MPI_REAL, iprev, 1, &
    MPI_COMM_WORLD, ireqs, ierr)
ENDIF
ENDDO

! Loop B
DO iy = 0, nprocs-1
  ix = MOD(iy+nprocs-myrank, nprocs)
  ista = is(ix); iend = ie(ix); jsta = js(iy); jend = je(iy)
  jlen = jend - jsta + 1
```

비틀림 분해 (8/14)

□ 비틀림 분해를 이용한 병렬화 코드 : Fortran (계속)

```
IF (iy /= 0) THEN
  CALL MPI_Irecv (x(ista,jsta-1), ilen, MPI_REAL, iprev, 1, &
    MPI_COMM_WORLD, ireqr, ierr)
  CALL MPI_WAIT(ireqr, istatus, ierr)
  CALL MPI_WAIT(ireqs, istatus, ierr)
ENDIF
DO j = jsta, jend
  DO i = ista, iend
    x(i, j) = x(i,j) + x(i,j-1)
  ENDDO
ENDDO
IF (iy /= nprocs-1) THEN
  CALL MPI_Isend (x(ista, jend), ilen, MPI_REAL, inext, 1, &
    MPI_COMM_WORLD, ireqs, ierr)
ENDIF
ENDDO
...
```

비틀림 분해 (9/14)

□ 비틀림 분해를 이용한 병렬화 코드 : C

```
/* main_twist*/
...
main(int argc, char *argv){
  ...
  double x[mx+1][my+1];
  double bufs[m], bufr[m];
  int *is, *ie, *js, *je;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  is = (int *)malloc(nprocs*sizeof(int));
  ie = (int *)malloc(nprocs*sizeof(int));
  js = (int *)malloc(nprocs*sizeof(int));
  je = (int *)malloc(nprocs*sizeof(int));
```

비틀림 분해 (10/14)

□ 비틀림 분해를 이용한 병렬화 코드 : C (계속)

```
for(ix=0; ix<nprocs; ix++){
    para_range(1, mx, nprocs, myrank, &is[ix], &ie[ix]);
    para_range(1, my, nprocs, myrank, &js[ix], &je[ix]);
}
inext = myrank + 1;
if (inext == nprocs) inext = 0;
iprev = myrank - 1;
if (iprev == -1) iprev = nprocs-1;
...
c Loop A
for(ix=0; ix<nprocs; ix++){
    iy = (ix+myrank)%nprocs;
    ista=is[ix]; iend=ie[ix]; jsta=js[iy]; jend=je[iy];
    jlen = jend-jsta+1;
```

비틀림 분해 (11/14)

□ 비틀림 분해를 이용한 병렬화 코드 : C (계속)

```
if(ix != 0){
    MPI_Irecv(&x[ista-1][jsta], jlen, MPI_DOUBLE, inext, 1,
             MPI_COMM_WORLD, &ireqr);
    MPI_Wait(&ireqr, &istatus);
    MPI_Wait(&ireqs, &istatus);
}
for(i=ista; i<=iend; i++){
    for(j=jsta; j<=jend; j++){
        if((i-1)==0) x[i-1][j]=0.0;
        x[i][j] = x[i][j] + x[i-1][j];
    }
}
if(ix != nprocs-1){
    MPI_Isend(&x[iend][jsta], jlen, MPI_DOUBLE, iprev, 1,
             MPI_COMM_WORLD, &ireqs);
}
}
```

비틀림 분해 (12/14)

□ 비틀림 분해를 이용한 병렬화 코드 : C (계속)

```
c Loop B
for(iy=0; iy<nprocs; iy++){
    ix = (iy+nprocs-myrank)%nprocs
    ista=is[ix]; iend=ie[ix]; jsta=js[iy]; jend=je[iy];
    ilen = iend-ista+1;
    if(iy != 0){
        MPI_Irecv(&bufr[ista], ilen, MPI_DOUBLE, inext, 1,
                 MPI_COMM_WORLD, &ireqr);
        MPI_Wait(&ireqr, &istatus);
        MPI_Wait(&ireqs, &istatus);
        for(i=ista; i<=iend; i++) x[i][jsta-1]=bufr[i];
    }

    for(i=ista; i<=iend; i++)
        for(j=jsta; j<=jend; j++){
```

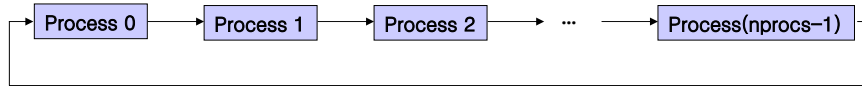
비틀림 분해 (13/14)

□ 비틀림 분해를 이용한 병렬화 코드 : C (계속)

```
        if((j-1)==0) x[i][j-1]=0.0;
        x[i][j] = x[i][j] + x[i][j-1];
    }
    if(iy != nprocs-1){
        for(i=ista; i<=iend; i++) bufc[i]=x[i][jend];
        MPI_Isend(&bufc[ista], ilen, MPI_DOUBLE, iprev, 1,
                 MPI_COMM_WORLD, &ireqs);
    }
}
free(is);
free(ie);
free(js);
free(je);
...
```

비틀림 분해 (14/14)

- 데이터의 흐름 : 교착에 주의

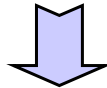


```
! Loop B
DO iy = 0, nprocs-1
  ...
  IF (iy /= 0) THEN
    Receive
    Wait for receive to complete
    Wait for send to complete
  ENDIF
  ...
  IF (iy /= nprocs-1) THEN
    send
  ENDIF
ENDDO
```

프리픽스 합 (1/6)

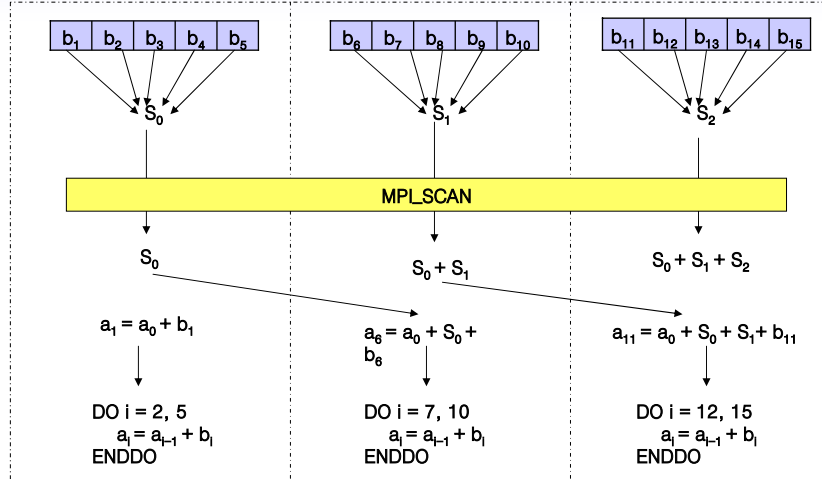
- 내포되지 않은 루프(1차원 배열)가 의존성을 가지는 경우

```
DO i = 1, n
  a(i) = a(i-1) op b(i)
ENDDO
```



```
PROGRAM main
PARAMETER(n=...)
REAL a(0:n), b(n)
...
DO i=1,n
  a(i) = a(i-1) + b(i)
ENDDO
...
```

프리픽스 합 (2/6)



프리픽스 합 (3/6)

□ 프리픽스 합을 이용한 병렬화 코드 : Fortran

```
PROGRAM main_prefix_sum
INCLUDE 'mpif.h'
PARAMETER (n = ...)
REAL a(0:n), b(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)
...
sum = 0.0
DO i = ista, iend
    sum = sum + b(i)
ENDDO
```

프리픽스 합 (4/6)

□ 프리픽스 합을 이용한 병렬화 코드 : Fortran (계속)

```
IF (myrank == 0) THEN
    sum = sum + a(0)
ENDIF
CALL MPI_SCAN (sum, ssum, 1, MPI_REAL, MPI_SUM, &
               MPI_COMM_WORLD, ierr)
a(ista) = b(ista) + ssum - sum
IF (myrank == 0) THEN
    a(ista) = a(ista) + a(0)
ENDIF
DO i = ista+1, iend
    a(i) = a(i-1) + b(i)
ENDDO
...
```

프리픽스 합 (5/6)

□ 프리픽스 합을 이용한 병렬화 코드 : C

```
/* prefix_sum */
#include <mpi.h>
#define n ...
void main(int argc, char *argv[]){
    double a[n+1], b[n+1];
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    para_range(1, n, nprocs, myrank, &ista, &iend);
    ...
    sum = 0.0;
    for(i=ista; i<=iend; i++)
        sum = sum + b[i];
}
```


프리픽스 합 (6/6)

□ 프리픽스 합을 이용한 병렬화 코드 : C (계속)

```
if(myrank==0) sum = sum + a[0];
MPI_Scan (&sum, &ssum, 1, MPI_DOUBLE, MPI_SUM,
          MPI_COMM_WORLD);
a[ista] = b[ista] + ssum - sum;
if (myrank == 0)
    a[ista] = a[ista] + a[0];
for(i=ista+1; i<=iend; i++)
    a[i] = a[i-1] + b[i];
...
}
```

제 4 장 MPI 병렬 프로그램 예제

2차원 유한 차분법, 분자 동역학 등의 실제
계산 문제들에서 어떻게 MPI가 사용되는지
살펴본다.

•2차원 유한 차분법의 병렬화

•몬테카를로 방법의 병렬화

•분자 동역학

•MPMD 모델



2차원 유한 차분법의 병렬화 (1/2)

□ 2차원 유한 차분법(FDM)의 핵심부 : 순차 프로그램

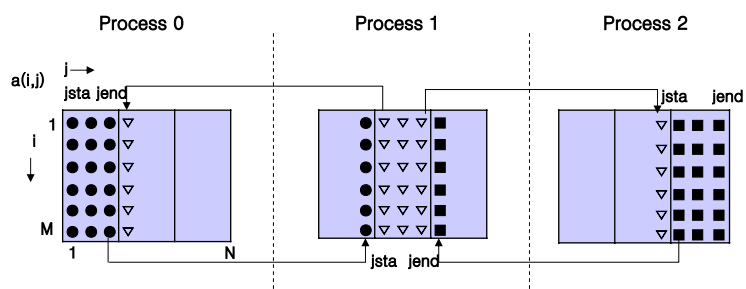
Fortran	C
<pre>... PARAMETER (m=6,n=9) DIMENSION a(m,n), b(m,n) DO j = 1, n DO i = 1, m a(i,j) = i+10.0*j ENDDO ENDDO DO j = 2, n-1 DO i = 2, m-1 b(i,j) = a(i-1,j)+a(i,j-1) & + a(i,j+1) + a(i+1,j) ENDDO ENDDO ...</pre>	<pre>... #define m 6 #define n 9 main(){ double a[m][n], b[m][n]; for(i=0; i<m; i++) for(j=0; j<n; j++) a[i][j] = (i+1)+10.*(j+1); for(i=1; i<m-1; i++) for(j=1; j<n-1; j++) b[i][j] = a[i-1][j] + a[i][j-1] + a[i][j+1] + a[i+1][j] ... }</pre>

2차원 유한 차분법의 병렬화 (2/2)

- 양 방향 의존성을 모두 가지고 있음
- 통신량을 최소화하는 데이터 분배 방식 결정
 - 열 방향 블록 분할
 - 행 방향 블록 분할
 - 양 방향 블록 분할

열 방향 블록 분할

- 경계 데이터 : Fortran(연속), C(불연속)



열 방향 블록 분할 코드 : Fortran (1/3)

```
PROGRAM parallel_2D_FDM_column
INCLUDE 'mpif.h'
PARAMETER (m = 6, n = 9)
DIMENSION a(m,n), b(m,n)
INTEGER istatus(MPI_STATUS_SIZE)
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, jsta, jend)
jsta2 = jsta; jend1 = jend
IF (myrank == 0) jsta2 = 2
IF (myrank == nprocs - 1) jend1 = n - 1
inext = myrank + 1
iprev = myrank - 1
```

열 방향 블록 분할 코드 : Fortran (2/3)

```
IF (myrank == nprocs - 1) inext = MPI_PROC_NULL
IF (myrank == 0) iprev = MPI_PROC_NULL
DO j = jsta, jend
  DO i = 1, m
    a(i,j) = i + 10.0 * j
  ENDDO
ENDDO
CALL MPI_ISEND(a(1,jend), m, MPI_REAL, inext, 1, &
  MPI_COMM_WORLD, isend1, ierr)
CALL MPI_ISEND(a(1,jsta), m, MPI_REAL, iprev, 1, &
  MPI_COMM_WORLD, isend2, ierr)
CALL MPI_IRECV(a(1,jsta-1), m, MPI_REAL, iprev, 1, &
  MPI_COMM_WORLD, irecv1, ierr)
CALL MPI_IRECV(a(1,jend+1), m, MPI_REAL, inext, 1, &
  MPI_COMM_WORLD, irecv2, ierr)
```

열 방향 블록 분할 코드 : Fortran (3/3)

```
CALL MPI_WAIT(isend1, istatus, ierr)
CALL MPI_WAIT(isend2, istatus, ierr)
CALL MPI_WAIT(irecv1, istatus, ierr)
CALL MPI_WAIT(irecv2, istatus, ierr)
DO j = jsta2, jend1
  DO i = 2, m - 1
    b(i,j) = a(i-1,j) + a(i,j-1) + a(i,j+1) + a(i+1,j)
  ENDDO
ENDDO
CALL MPI_FINALIZE(ierr)
END
```

열 방향 블록 분할 코드 : C (1/4)

```
/*parallel_2D_FDM_column*/
#include <mpi.h>
#define m 6
#define n 9
void para_range(int, int, int, int, int*, int*);
int min(int, int);
main(int argc, char *argv[]){
  int i, j, nprocs, myrank ;
  double a[m][n],b[m][n];
  double works1[m],workr1[m],works2[m],workr2[m];
  int jsta, jend, jsta2, jend1, inext, iprev;
  MPI_Request isend1, isend2, irecv1, irecv2;
  MPI_Status istatus;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
```

열 방향 블록 분할 코드 : C (2/4)

```
para_range(0, n-1, nprocs, myrank, &jsta, &jend);
jsta2 = jsta; jend1 = jend;
if(myrank==0) jsta2=1;
if(myrank==nprocs-1) jend1=n-2;
inext = myrank + 1;
iprev = myrank - 1;
if (myrank == nprocs-1) inext = MPI_PROC_NULL
if (myrank == 0) iprev = MPI_PROC_NULL
for(i=0; i<m; i++)
    for(j=jsta; j<=jend; j++) a[i][j] = i + 10.0 * j
if(myrank != nprocs-1)
    for(i=0; i<m; i++) works1[i]=a[i][jend];
if(myrank != 0)
    for(i=0; i<m; i++) works2[i]=a[i][jsta];
```

열 방향 블록 분할 코드 : C (3/4)

```
MPI_Isend(works1, m, MPI_DOUBLE, inext, 1,
          MPI_COMM_WORLD, &isend1);
MPI_Isend(works2, m, MPI_DOUBLE, iprev, 1,
          MPI_COMM_WORLD, &isend2);
MPI_Irecv(workr1, m, MPI_DOUBLE, iprev, 1,
          MPI_COMM_WORLD, &irecv1);
MPI_Irecv(workr2, m, MPI_DOUBLE, inext, 1,
          MPI_COMM_WORLD, &irecv2);
MPI_Wait(&isend1, &istatus);
MPI_Wait(&isend2, &istatus);
MPI_Wait(&irecv1, &istatus);
MPI_Wait(&irecv2, &istatus);
if (myrank != 0)
    for(i=0; i<m; i++) a[i][jsta-1] = workr1[i];
if (myrank != nprocs-1)
    for(i=0; i<m; i++) a[i][jend+1] = workr2[i];
```

열 방향 블록 분할 코드 : C (4/4)

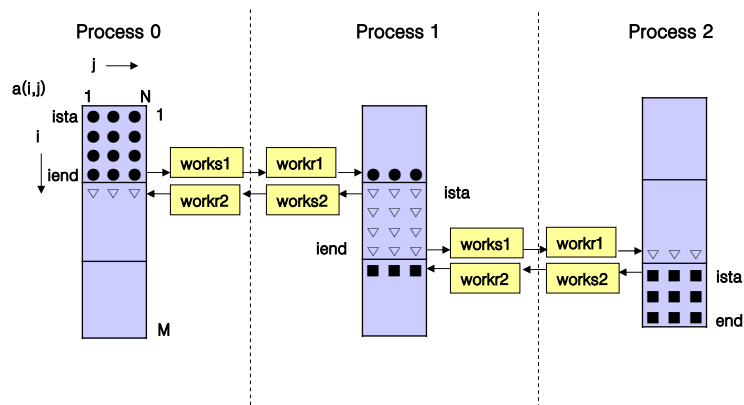
```

for (i=1; i<=m-2; i++)
  for(j=jsta2; j<=jend1; j++)
    b[i][j] = a[i-1][j] + a[i][j-1]
              + a[i][j+1] + a[i+1][j];
MPI_Finalize();
}

```

행 방향 블록 분할

□ **경계 데이터 : Fortran(불연속), C(연속)**



행 방향 블록 분할 코드 : Fortran (1/4)

```
PROGRAM parallel_2D_FDM_row
INCLUDE 'mpif.h'
PARAMETER (m = 12, n = 3)
DIMENSION a(m,n), b(m,n)
DIMENSION works1(n), workr1(n), works2(n), workr2(n)
INTEGER istatus(MPI_STATUS_SIZE)
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, m, nprocs, myrank, ista, iend)
ista2 = ista; iend1 = iend
IF (myrank == 0) ista2 = 2
IF (myrank == nprocs - 1) iend1 = m-1
inext = myrank + 1; iprev = myrank - 1
```

행 방향 블록 분할 코드 : Fortran (2/4)

```
IF (myrank == nprocs - 1) inext = MPI_PROC_NULL
IF (myrank == 0) iprev = MPI_PROC_NULL

DO j = 1, n
  DO i = ista, iend
    a(i,j) = i + 10.0 * j
  ENDDO
ENDDO

IF (myrank /= nprocs - 1) THEN
  DO j = 1, n
    works1(j) = a(iend,j)
  ENDDO
ENDIF
IF (myrank /= 0) THEN
  DO j = 1, n
    works2(j) = a(ista,j)
  ENDDO
ENDIF
```


행 방향 블록 분할 코드 : Fortran (3/4)

```
CALL MPI_ISEND(works1,n,MPI_REAL8,inext,1, &
               MPI_COMM_WORLD, isend1,ierr)
CALL MPI_ISEND(works2,n,MPI_REAL8,iprev,1, &
               MPI_COMM_WORLD, isend2,ierr)
CALL MPI_IRecv(workr1,n,MPI_REAL8,iprev,1, &
               MPI_COMM_WORLD, irecv1,ierr)
CALL MPI_IRecv(workr2,n,MPI_REAL8,inext,1, &
               MPI_COMM_WORLD, irecv2,ierr)
CALL MPI_WAIT(isend1, istatus, ierr)
CALL MPI_WAIT(isend2, istatus, ierr)
CALL MPI_WAIT(irecv1, istatus, ierr)
CALL MPI_WAIT(irecv2, istatus, ierr)
```

행 방향 블록 분할 코드 : Fortran (4/4)

```
IF (myrank /= 0) THEN
  DO j = 1, n
    a(ista-1,j) = workr1(j)
  ENDDO
ENDIF

IF (myrank /= nprocs - 1) THEN
  DO j = 1, n
    a(iend+1,j) = workr2(j)
  ENDDO
ENDIF

DO j = 2, n - 1
  DO i = ista2, iend1
    b(i,j) = a(i-1,j) + a(i,j-1) + a(i,j+1) + a(i+1,j)
  ENDDO
ENDDO
CALL MPI_FINALIZE(ierr)
END
```

행 방향 블록 분할 코드 : C (1/3)

```
/*parallel_2D_FDM_row*/
#include <mpi.h>
#define m 12
#define n 3
void para_range(int, int, int, int, int*, int*);
int min(int, int);
main(int argc, char *argv[]){
    int i, j, nprocs, myrank ;
    double a[m][n],b[m][n];
    int ista, iend, ista2, iend1, inext, iprev;
    MPI_Request isend1, isend2, irecv1, irecv2;
    MPI_Status istatus;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
```

행 방향 블록 분할 코드 : C (2/3)

```
para_range(0, m-1, nprocs, myrank, &ista, &iend);
ista2 = ista; iend1 = iend;
if(myrank==0) ista2=1;
if(myrank==nprocs-1) iend1=m-2;
inext = myrank + 1;
iprev = myrank - 1;
if (myrank == nprocs-1) inext = MPI_PROC_NULL
if (myrank == 0) iprev = MPI_PROC_NULL
for(i=ista; i<=iend; i++)
    for(j=0; j<n; j++) a[i][j] = i + 10.0 * j
MPI_Isend(&a[iend][0], n, MPI_DOUBLE, inext, 1,
          MPI_COMM_WORLD, &isend1);
MPI_Isend(&a[ista][0], n, MPI_DOUBLE, iprev, 1,
          MPI_COMM_WORLD, &isend2);
```

행 방향 블록 분할 코드 : C (3/3)

```

MPI_Irecv(&a[ista-1][0], n, MPI_DOUBLE, iprev, 1,
          MPI_COMM_WORLD, &irecv1);
MPI_Irecv(&a[iend+1][0], n, MPI_DOUBLE, inext, 1,
          MPI_COMM_WORLD, &irecv2);

MPI_Wait(&isend1, &istatus);
MPI_Wait(&isend2, &istatus);
MPI_Wait(&irecv1, &istatus);
MPI_Wait(&irecv2, &istatus);

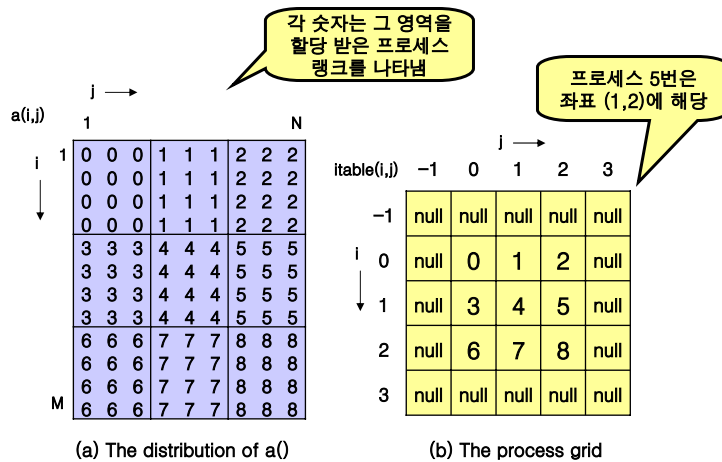
for (i=ista2; i<=iend1; i++)
  for(j=1; j<=n-2; j++)
    b[i][j] = a[i-1][j] + a[i][j-1] +
              a[i][j+1] + a[i+1][j];

MPI_Finalize();
}

```

양 방향 블록 분할 (1/2)

□ 프로세스 그리드 이용



양 방향 블록 분할 코드 : Fortran (2/6)

```
irank = 0
DO i = 0, iprocs-1
  DO j = 0, jprocs-1
    itable(i,j) = irank
    IF (myrank == irank) THEN
      myranki = i; myrankj = j
    ENDIF
    irank = irank + 1
  ENDDO
ENDDO
CALL para_range(1, n, jprocs, myrankj, jsta, jend)
jsta2 = jsta; jend1 = jend
IF (myrankj == 0) jsta2 = 2
IF (myrankj == jprocs-1) jend1 = n-1
CALL para_range(1, m, iprocs, myranki, ista, iend)
ista2 = ista; iend1 = iend
IF (myranki == 0) ista2 = 2
IF (myranki == iprocs-1) iend1 = m-1
ilen = iend - ista + 1; jlen = jend - jsta
```

양 방향 블록 분할 코드 : Fortran (3/6)

```
jnext = itable(myranki, myrankj + 1)
jprev = itable(myranki, myrankj - 1)
inext = itable(myranki+1, myrankj)
iprev = itable(myranki-1, myrankj)
DO j = jsta, jend
  DO i = ista, iend
    a(i,j) = i + 10.0*j
  ENDDO
ENDDO
IF (myranki /= iprocs-1) THEN
  DO j = jsta, jend
    works1(j) = a(iend,j)
  ENDDO
ENDIF
IF (myranki /= 0) THEN
  DO j = jsta, jend
    works2(j) = a(ista,j)
  ENDDO
ENDIF
```

양 방향 블록 분할 코드 : Fortran (4/6)

```
CALL MPI_ISEND(a(ista,jend), ilen, MPI_REAL8, jnext, 1,&
               MPI_COMM_WORLD, isend1, ierr)
CALL MPI_ISEND(a(ista,jsta), ilen, MPI_REAL8, jprev, 1,&
               MPI_COMM_WORLD, isend2, ierr)
CALL MPI_ISEND(works1(jsta), jlen, MPI_REAL8, inext, 1,&
               MPI_COMM_WORLD, jsend1, ierr)
CALL MPI_ISEND(works2(jsta), jlen, MPI_REAL8, iprev, 1,&
               MPI_COMM_WORLD, jsend2, ierr)
CALL MPI_IRECV(a(ista,jsta-1), ilen, MPI_REAL8, jprev, 1,&
               MPI_COMM_WORLD, irecv1, ierr)
CALL MPI_IRECV(a(ista,jend+1), ilen, MPI_REAL8, jnext, 1,&
               MPI_COMM_WORLD, irecv2, ierr)
CALL MPI_IRECV(workr1(jsta), jlen, MPI_REAL8, iprev, 1,&
               MPI_COMM_WORLD, jrecv1, ierr)
CALL MPI_IRECV(workr2(jsta), jlen, MPI_REAL8, inext, 1,&
               MPI_COMM_WORLD, jrecv2, ierr)
```

양 방향 블록 분할 코드 : Fortran (5/6)

```
CALL MPI_WAIT(isend1, istatus, ierr)
CALL MPI_WAIT(isend2, istatus, ierr)
CALL MPI_WAIT(jsend1, istatus, ierr)
CALL MPI_WAIT(jsend2, istatus, ierr)
CALL MPI_WAIT(irecv1, istatus, ierr)
CALL MPI_WAIT(irecv2, istatus, ierr)
CALL MPI_WAIT(jrecv1, istatus, ierr)
CALL MPI_WAIT(jrecv2, istatus, ierr)
IF (myranki /= 0) THEN
  DO j = jsta, jend
    a(ista-1,j) = workr1(j)
  ENDDO
ENDIF
IF (myranki /= iprocs-1) THEN
  DO j = jsta, jend
    a(iend+1,j) = workr2(j)
  ENDDO
ENDIF
```

양 방향 블록 분할 코드 : Fortran (6/6)

```
DO j = jsta2, jend1
  DO i = ista2, iend1
    b(i,j) = a(i-1,j) + a(i,j-1) + a(i,j+1) + a(i+1,j)
  ENDDO
ENDDO
CALL MPI_FINALIZE(ierr)
END
```

양 방향 블록 분할 코드 : C (1/6)

```
/*parallel_2D_FDM_both*/
#include <mpi.h>
#define m 12
#define n 9
#define iprocs 3
#define jprocs 3
void para_range(int, int, int, int, int*, int*);
int min(int, int);
main(int argc, char *argv[]){
  int i, j, irank, nprocs, myrank ;
  double a[m][n],b[m][n];
  double works1[m],workr1[m],works2[m],workr2[m];
  int jsta, jend, jsta2, jend1, jnext, jprev, jlen;
  int ista, iend, ista2, iend1, inext, iprev, ilen;
  int itable[iprocs+2][jprocs+2];
  int myranki, myrankj;
  MPI_Request
  isend1, isend2, irecv1, irecv2, jsend1, jsend2, jrecv1, jrecv2;
  MPI_Status istatus;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
```

양 방향 블록 분할 코드 : C (2/6)

```
for(i=0; i<=iprocs+1; i++)
    for(j=0; j<=jprocs+1; j++)  itable[i][j]=MPI_PROC_NULL;
irank = 0;
for(i=1; i<=iprocs; i++)
    for(j=1; j<=jprocs; j++){
        itable[i][j]=irank;
        if(myrank==irank){
            myranki = i-1; myrankj = j-1;
        }
        irank = irank + 1;
    }
para_range(0, n-1, jprocs, myrankj, &jsta, &jend);
jsta2 = jsta; jend1 = jend;
if(myrankj==0) jsta2=1;
if(myrankj==jprocs-1) jend1=n-2;
para_range(0, m-1, iprocs, myranki, &ista, &iend);
ista2 = ista; iend1 = iend;
if(myranki==0) ista2=1;
if(myranki==iprocs-1) iend1=m-2;
```

양 방향 블록 분할 코드 : C (3/6)

```
ilen = iend-ista+1;  jlen = jend-jsta+1;
jnext = itable[myranki][myrankj+1];
jprev = itable[myranki][myrankj-1];
inext = itable[myranki+1][myrankj];
iprev = itable[myranki-1][myrankj];

for(i=ista; i<=iend; i++)
    for(j=jsta; j<=jend; j++)  a[i][j] = i + 10.0 * j

if(myrankj != jprocs-1)
    for(i=ista; i<=iend; i++) works1[i]=a[i][jend];
if(myrankj != 0)
    for(i=ista; i<=iend; i++) works2[i]=a[i][jsta];
```


양 방향 블록 분할 코드 : C (4/6)

```
MPI_Isend(&works1[ista], ilen, MPI_DOUBLE, jnext, 1,
          MPI_COMM_WORLD, &isend1);
MPI_Isend(&works2[ista], ilen, MPI_DOUBLE, jprev, 1,
          MPI_COMM_WORLD, &isend2);
MPI_Isend(&a[iend][jsta], jlen, MPI_DOUBLE, inext, 1,
          MPI_COMM_WORLD, &jsend1);
MPI_Isend(&a[ista][jsta], jlen, MPI_DOUBLE, iprev, 1,
          MPI_COMM_WORLD, &jsend2);
MPI_Irecv(&workr1[ista], ilen, MPI_DOUBLE, jprev, 1,
          MPI_COMM_WORLD, &irecv1);
MPI_Irecv(&workr2[ista], ilen, MPI_DOUBLE, jnext, 1,
          MPI_COMM_WORLD, &irecv2);
MPI_Irecv(&a[ista-1][jsta], jlen, MPI_DOUBLE, iprev, 1,
          MPI_COMM_WORLD, &jrecv1);
MPI_Irecv(&a[iend+1][jsta], jlen, MPI_DOUBLE, inext, 1,
          MPI_COMM_WORLD, &jrecv2);
```

양 방향 블록 분할 코드 : C (5/6)

```
MPI_Wait(&isend1, &istatus);
MPI_Wait(&isend2, &istatus);
MPI_Wait(&jsend1, &istatus);
MPI_Wait(&jsend2, &istatus);
MPI_Wait(&irecv1, &istatus);
MPI_Wait(&irecv2, &istatus);
MPI_Wait(&jrecv1, &istatus);
MPI_Wait(&jrecv2, &istatus);

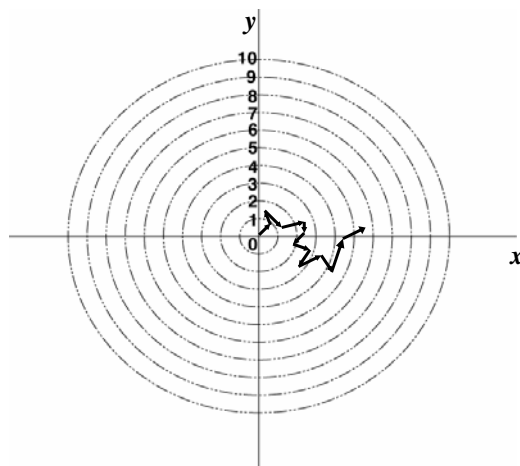
if (myrankj != 0)
    for(i=ista; i<=iend; i++) a[i][jsta-1] = workr1[i];
if (myrankj != jprocs-1)
    for(i=ista; i<=iend; i++) a[i][jend+1] = workr2[i];
```

양 방향 블록 분할 코드 : C (6/6)

```
for (i=ista2; i<=iend1; i++)
  for(j=jsta2; j<=jend1; j++)
    b[i][j] = a[i-1][j] + a[i][j-1]
              + a[i][j+1] + a[i+1][j];
MPI_Finalize();
}
```

몬테카를로 방법의 병렬화 (1/2)

□ 2차원 임의 행로



몬테카를로 방법의 병렬화 (2/2)

```
PROGRAM random_serial
PARAMETER (n = 100000)
INTEGER itotal(0:9)
REAL seed
pi = 3.1415926
DO i = 0, 9
    itotal(i) = 0
ENDDO
seed = 0.5
CALL srand(seed)
DO i = 1, n
    x = 0.0; y = 0.0
    DO istep = 1, 10
        angle = 2.0*pi*rand()
        x = x + cos(angle)
        y = y + sin(angle)
    ENDDO
    itemp = sqrt(x**2 + y**2)
    itotal(itemp) = &
        itotal(itemp) + 1
ENDDO
PRINT *, 'total =', itotal
END
```

```
/*random serial*/
#include <math.h>
#define n 100000
main(){
    int i,istep,itotal[10],itemp;
    double r, seed, pi, x, y, angle;
    pi = 3.1415926;
    for(i=0;i<10;i++) itotal[i]=0;
    seed = 0.5; srand(seed);
    for(i=0; i<n; i++){
        x = 0.0; y = 0.0;
        for(istep=0;istep<10;istep++){
            r = (double)rand();
            angle = 2.0*pi*r/32768.0;
            x = x + cos(angle);
            y = y + sin(angle);
        }
        itemp = sqrt(x*x + y*y);
        itotal[itemp]=itotal[itemp]+1;
    }
    for(i=0; i<10; i++){
        printf("%d :", i);
        printf("total=%d\n",itotal[i]);
    }
}
```

2차원 임의 행로 코드 : Fortran (1/2)

```
PROGRAM random_parallel
INCLUDE 'mpif.h'
PARAMETER (n = 100000)
INTEGER itotal(0:9), iitotal(0:9)
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)
pi = 3.1415926
DO i = 0, 9
    itotal(i) = 0
ENDDO
seed = 0.5 + myrank
CALL srand(seed)
```

2차원 임의 행로 코드 : Fortran (2/2)

```
DO i = ista, iend
  x = 0.0; y = 0.0
  DO istep = 1, 10
    angle = 2.0*pi*rand()
    x = x + cos(angle)
    y = y + sin(angle)
  ENDDO
  itemp = sqrt(x**2 + y**2)
  itotal(itemp) = itotal(itemp) + 1
ENDDO
CALL MPI_REDUCE(itotal, iitotal, 10, MPI_INTEGER, &
               MPI_SUM, 0, MPI_COMM_WORLD, ierr)
PRINT *, 'total =', iitotal
CALL MPI_FINALIZE(ierr)
END
```

2차원 임의 행로 코드 : C (1/2)

```
/*para_random*/
#include <mpi.h>
#include <stdio.h>
#include <math.h>
#define n 100000
void para_range(int, int, int, int, int*, int*);
int min(int, int);
main (int argc, char *argv[]){
  int i, istep, itotal[10], iitotal[10], itemp;
  int ista, iend, nprocs, myrank;
  double r, seed, pi, x, y, angle;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  para_range(0, n-1, nprocs, myrank, &ista, &iend);
  pi = 3.1415926;
  for(i=0; i<10; i++) itotal[i] = 0;
```

2차원 임의 행로 코드 : C (2/2)

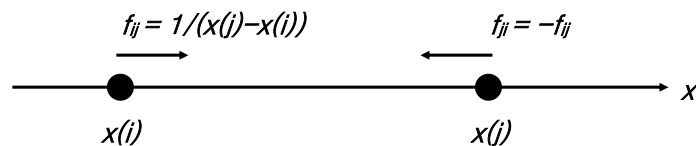
```

seed = 0.5 + myrank; srand(seed);
for(i=ista; i<=iend; i++){
  x = 0.0; y = 0.0;
  for(istep=0; istep<10; istep++){
    r = (double)rand();
    angle = 2.0*pi*r/32768.0;
    x = x + cos(angle); y = y + sin(angle);
  }
  itemp = sqrt(x*x + y*y);
  itotal[itemp] = itotal[itemp] + 1;
}
MPI_Reduce(itotal, iitotal, 10, MPI_INT, MPI_SUM, 0,
           MPI_COMM_WORLD);
for(i=0; i<10; i++){
  printf(" %d :", i);
  printf(" total = %d\n", iitotal[i]);
}
MPI_Finalize();
}

```

분자 동역학 (1/7)

- 1차원 상에서 상호작용 하는 두 개 입자



- 입자 i가 받게 되는 힘의 총합

$$f_i = \sum_{j \neq i} f_{ij} = -\sum_{j < i} f_{ji} + \sum_{j > i} f_{ij}$$

분자 동역학 (2/7)

□ 7개의 입자

f(1) =		+f12	+f13	+f14	+f15	+f16	+f17
f(2) =	-f12		+f23	+f24	+f25	+f26	+f27
f(3) =	-f13	-f23		+f34	+f35	+f36	+f37
f(4) =	-f14	-f24	-f34		+f45	+f46	+f47
f(5) =	-f15	-f25	-f35	-f45		+f56	+f57
f(6) =	-f16	-f26	-f36	-f46	-f56		+f67
f(7) =	-f17	-f27	-f37	-f47	-f57	-f67	

분자 동역학 (3/7)

□ Fortran 순차코드

$$f_{ij} = -f_{ji}$$

- 삼각형 모양의 루프 실행
- 순환 분할(i 또는 j에 대해)

```

...
PARAMETER (n = ...)
REAL f(n), x(n)
...
DO itime = 1, 100
  DO i = 1, n
    f(i) = 0.0
  ENDDO
  DO j = 1, n-1
    DO j = i+1, n
      fij = 1.0 / (x(j)-x(i))
      f(i) = f(i) + fij
      f(j) = f(j) - fij
    ENDDO
  ENDDO
  DO i = 1, n
    x(i) = x(i) + f(i)
  ENDDO
ENDDO
...
    
```

hot spot

분자 동역학 (4/7)

Process 0

f(1) =		+f12	+f13	+f14	+f15	+f16	+f17
f(2) =	-f12						
f(3) =	-f13						
f(4) =	-f14		+f45	+f46	+f47		
f(5) =	-f15		-f45				
f(6) =	-f16		-f46				
f(7) =	-f17		-f47				

Process 1

f(1) =		+f23	+f24	+f25	+f26	+f27	
f(2) =							
f(3) =		-f23					
f(4) =		-f24					
f(5) =		-f25		+f56	+f57		
f(6) =		-f26		-f56			
f(7) =		-f27		-f57			

Process 2

f(1) =			+f34	+f35	+f36	+f37	
f(2) =							
f(3) =							
f(4) =			-f34				
f(5) =			-f35				
f(6) =			-f36		+f67		
f(7) =			-f37		-f67		

변수 j에 대한
순환 분할

MPI_ALLREDUCE

All Processes

ff(1) =		+f12	+f13	+f14	+f15	+f16	+f17
ff(2) =	-f12		+f23	+f24	+f25	+f26	+f27
ff(3) =	-f13	-f23		+f34	+f35	+f36	+f37
ff(4) =	-f14	-f24	-f34		+f45	+f46	+f47
ff(5) =	-f15	-f25	-f35	-f45		+f56	+f57
ff(6) =	-f16	-f26	-f36	-f46	-f56		+f67
ff(7) =	-f17	-f27	-f37	-f47	-f57	-f67	

분자 동역학 (5/7)

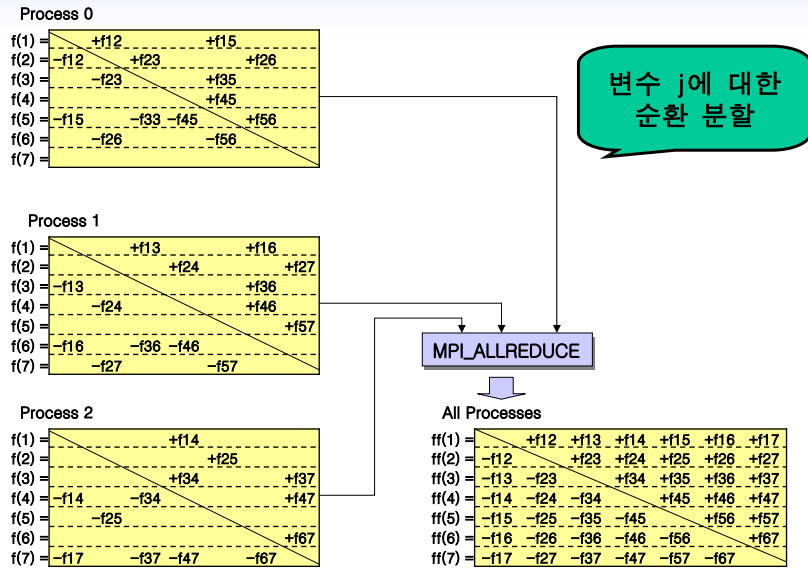
```

...
PARAMETER (n = ...)
REAL f(n), x(n), ff(n)
...
DO itime = 1, 100
  DO i = 1, n
    f(i) = 0.0
  ENDDO
  DO i = 1+myrank, n-1, nprocs
    DO j = i+1, n
      fij = 1.0 / (x(j) - x(i))
      f(i) = f(i) + fij
      f(j) = f(j) - fij
    ENDDO
  ENDDO
  CALL MPI_ALLREDUCE(f, ff, n, MPI_REAL, MPI_SUM, &
    MPI_COMM_WORLD, ierr)
  DO i = 1, n
    x(i) = x(i) + ff(i)
  ENDDO
ENDDO
...

```

변수 j에 대한
순환 분할

분자 동역학 (6/7)



분자 동역학 (7/7)

```

...
PARAMETER (n = ...)
REAL f(n), x(n), ff(n)
...
DO itime = 1, 100
  DO i = 1, n
    f(i) = 0.0
  ENDDO
  irank = -1
  DO i = 1, n-1
    DO j = i+1, n
      irank = irank + 1
      IF (irank == nprocs) irank = 0
      IF (myrank == irank) THEN
        fij = 1.0 / (x(j) - x(i))
        f(i) = f(i) + fij
        f(j) = f(j) - fij
      ENDIF
    ENDDO
  ENDDO
  CALL MPI_ALLREDUCE (f, ff, n, MPI_REAL, MPI_SUM, MPI_COMM_WORLD, ierr)
  DO i = 1, n
    x(i) = x(i) + ff(i)
  ENDDO
ENDDO
...

```

변수 j에 대한 순환 분할

MPMD 모델 (1/4)

Process 0

```
PROGRAM fluid
INCLUDE 'mpif.h'
...
CALL MPI_INIT
CALL MPI_COMM_SIZE
CALL MPI_COMM_RANK
...
DO itime = 1,n
  Computation of Fluid Dynamics
  CALL MPI_SEND
  CALL MPI_RECV
ENDDO
...
END
```

Process 1

```
PROGRAM struct
INCLUDE 'mpif.h'
...
CALL MPI_INIT
CALL MPI_COMM_SIZE
CALL MPI_COMM_RANK
...
DO itime = 1,n
  Computation of Structural Analysis
  CALL MPI_RECV
  CALL MPI_SEND
ENDDO
...
END
```

MPMD 모델 (2/4)

□ MPMD 병렬 실행 : IBM AIX(ksh)

```
$mpxlf90 fluid.f -o fluid
$mpxlf90 struct.f -o struct
$export MP_PGMMODEL=mpmd
$export MP_CMDFILE=cmdfile
$poe -procs 2
```

□ cmdfile : 실행 명령 구성 파일

```
fluid
struct
```

MPMD 모델 (3/4)

□ 마스터/워커 MPMD 프로그램

```

PROGRAM main
PARAMETER (njobmax = 100)
DO njob = 1, njobmax
    CALL work(njob)
ENDDO
...
END
    
```

서로 독립적으로 수행되는
100개의 작업

MPMD 모델 (4/4)

Master

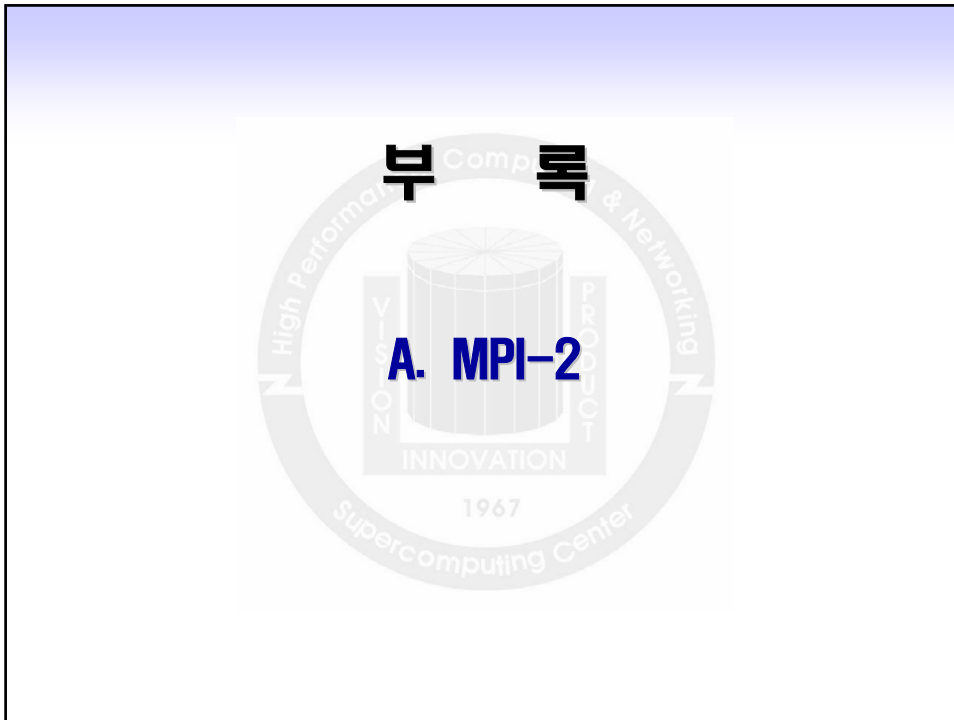
```

PROGRAM master
INCLUDE 'mpif.h'
PARAMETER (njobmax=100)
INTEGER istatus(MPL_STATUS_SIZE)
CALL MPL_INIT(ierr)
CALL MPL_COMM_SIZE(MPL_COMM_WORLD, nprocs, ierr)
CALL MPL_COMM_RANK(MPL_COMM_WORLD, myrank, ierr)
itag = 1
DO njob = 1, njobmax
    CALL MPL_RECV(iwk, 1, MPL_INTEGER, MPL_ANY_SOURCE,
                 itag, MPL_COMM_WORLD, istatus, ierr)
    idest = istatus(MPL_SOURCE)
    CALL MPL_SEND(njob, 1, MPL_INTEGER, idest, itag,
                 MPL_COMM_WORLD, ierr)
ENDDO
DO i = 1, nprocs-1
    CALL MPL_RECV(iwk, 1, MPL_INTEGER, MPL_ANY_SOURCE,
                 itag, MPL_COMM_WORLD, istatus, ierr)
    idest = istatus(MPL_SOURCE)
    CALL MPL_SEND(-1, 1, MPL_INTEGER, idest,
                 itag, MPL_COMM_WORLD, ierr)
ENDDO
CALL MPL_FINALIZE(ierr)
...
END
    
```

Worker

```

PROGRAM worker
INCLUDE 'mpif.h'
INTEGER istatus(MPL_STATUS_SIZE)
CALL MPL_INIT(ierr)
CALL MPL_COMM_SIZE(MPL_COMM_WORLD, nprocs, ierr)
CALL MPL_COMM_RANK(MPL_COMM_WORLD, myrank, ierr)
itag = 1
iwk = 0
DO
    CALL MPL_SEND(iwk, 1, MPL_INTEGER, 0, itag, MPL_COMM_
                 WORLD, ierr)
    CALL MPL_RECV(njob, 1, MPL_INTEGER, 0, itag,
                 MPL_COMM_WORLD, istatus, ierr)
    IF(njob == -1) EXIT
    CALL work(njob)
ENDDO
CALL MPL_FINALIZE(ierr)
END
    
```



MPI-2

- MPI-2에 추가된 영역
 1. 병렬 I/O (MPI I/O)
 2. 원격 메모리 접근 (일방 통신)
 3. 동적 프로세스 운영

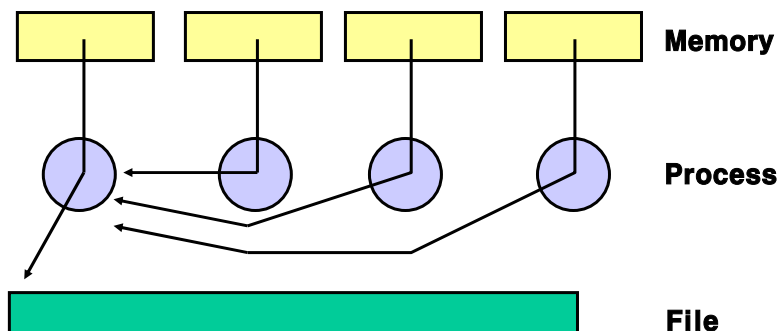
- IBM 시스템의 병렬 환경 지원 소프트웨어, PE(v3.2)에서는 동적 프로세스 운영을 제외한 대부분의 MPI-2 규약을 지원하고 있다.

병렬 I/O

- MPI-2에서 지원
- 운영체제가 지원하는 일반적인 순차 I/O기능 기반에 추가적인 성능과 안정성 지원

병렬 프로그램과 순차 I/O (1) (1/6)

- 하나의 프로세스가 모든 I/O 담당
- 편리할 수 있지만 성능과 범위성에 한계가 있음



병렬 프로그램과 순차 I/O (1) (2/6)

□ 예 : 각 프로세스가 가진 100개의 정수 출력

1. 각 프로세스가 배열 buf() 초기화
2. 프로세스 0를 제외한 모든 프로세스는 배열 buf()를 프로세스 0으로 송신
3. 프로세스 0은 가지고 있는 배열을 파일에 먼저 기록하고, 다른 프로세스로부터 차례로 데이터를 받아 파일에 기록

병렬 프로그램과 순차 I/O (1) (3/6)

□ Fortran 코드

```
PROGRAM serial_IO1
INCLUDE 'mpif.h'
INTEGER BUFSIZE
PARAMETER (BUFSIZE = 100)
INTEGER nprocs, myrank, ierr, buf(BUFSIZE)
INTEGER status(MPI_STATUS_SIZE)

Call MPI_INIT(ierr)
Call MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
Call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
DO i = 1, BUFSIZE
    buf(i) = myrank * BUFSIZE + i
ENDDO
```

병렬 프로그램과 순차 I/O (1) (4/6)

□ Fortran 코드 (계속)

```
IF (myrank /= 0) THEN
    CALL MPI_SEND(buf, BUFSIZE, MPI_INTEGER, 0, 99, &
                 MPI_COMM_WORLD, ierr)
ELSE
    OPEN (UNIT=10, FILE="testfile", STATUS="NEW", ACTION="WRITE")
    WRITE(10,*) buf
    DO i = 1, nprocs-1
        CALL MPI_RECV(buf, BUFSIZE, MPI_INTEGER, i, 99, &
                     MPI_COMM_WORLD, status, ierr)
        WRITE (10,*) buf
    ENDDO
ENDIF
CALL MPI_FINALIZE(ierr)
END
```

병렬 프로그램과 순차 I/O (1) (5/6)

□ C 코드

```
/*example of serial I/O*/
#include <mpi.h>
#include <stdio.h>
#define BUFSIZE 100
void main (int argc, char *argv[]){
int i, nprocs, myrank, buf[BUFSIZE] ;
MPI_Status status;
FILE *myfile;

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
for(i=0; i<BUFSIZE; i++)
    buf[i] = myrank * BUFSIZE + i;
```

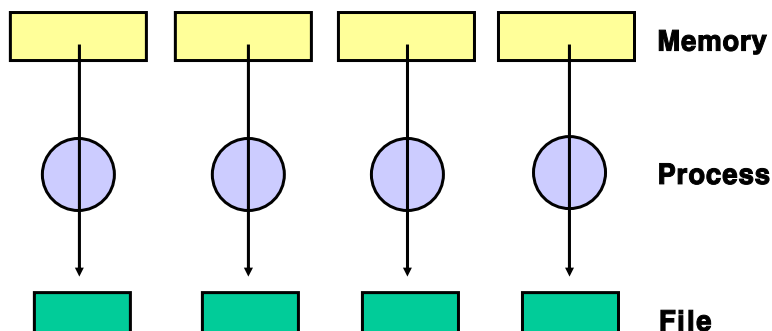
병렬 프로그램과 순차 I/O (1) (6/6)

□ C 코드 (계속)

```
if(myrank != 0)
    MPI_Send(buf, BUFSIZE, MPI_INT, 0, 99, MPI_COMM_WORLD);
else{
    myfile = fopen("testfile", "wb");
    fwrite(buf, sizeof(int), BUFSIZE, myfile);
    for(i=1; i<nprocs; i++){
        MPI_Recv(buf, BUFSIZE, MPI_INT, i, 99,
                MPI_COMM_WORLD, &status);
        fwrite(buf, sizeof(int), BUFSIZE, myfile);
    }
    fclose(myfile);
}
MPI_Finalize();
}
```

병렬 프로그램과 순차 I/O (2) (1/3)

- 모든 프로세스가 독립적으로 각자의 I/O 기능 담당
- 여러 파일 생성으로 결과 처리가 불편



병렬 프로그램과 순차 I/O (2) (2/3)

□ Fortran 코드 : 개별적인 파일 생성

```
PROGRAM serial_IO2
INCLUDE 'mpif.h'
INTEGER BUFSIZE
PARAMETER (BUFSIZE = 100)
INTEGER nprocs, myrank, ierr, buf(BUFSIZE)
CHARACTER*2 number
CHARACTER*20 fname(0:128)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
DO i = 1, BUFSIZE
    buf(i) = myrank * BUFSIZE + i
ENDDO
WRITE(number, *) myrank
fname(myrank) = "testfile."//number
OPEN(UNIT=myrank+10, FILE=fname(myrank), STATUS="NEW", ACTION="WRITE")
WRITE(myrank+10, *) buf
CLOSE(myrank+10)
CALL MPI_FINALIZE(ierr)
END
```

병렬 프로그램과 순차 I/O (2) (3/3)

□ C 코드 : 개별적인 파일 생성

```
/*example of parallel UNIX write into separate files */
#include <mpi.h>
#include <stdio.h>
#define BUFSIZE 100
void main (int argc, char *argv[]){
    int i, nprocs, myrank, buf[BUFSIZE] ;
    char filename[128];
    FILE *myfile;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    for(i=0; i<BUFSIZE; i++)
        buf[i] = myrank * BUFSIZE + i;
    sprintf(filename, "testfile.%d", myrank);
    myfile = fopen(filename, "wb");
    fwrite(buf, sizeof(int), BUFSIZE, myfile);
    fclose(myfile);
    MPI_Finalize();
}
```


병렬 I/O의 사용 (1/5)

□ 기본적인 병렬 I/O 루틴의 사용

- MPI_FILE_OPEN
- MPI_FILE_WRITE
- MPI_FILE_CLOSE

병렬 I/O의 사용 (2/5)

□ Fortran 코드 : 개별적인 파일 생성

```
PROGRAM parallel_IO_1
INCLUDE 'mpif.h'
INTEGER BUFSIZE
PARAMETER (BUFSIZE = 100)
INTEGER nprocs, myrank, ierr, buf(BUFSIZE), myfile
CHARACTER*2 number
CHARACTER*20 filename(0:128)

CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
DO i = 1, BUFSIZE
    buf(i) = myrank * BUFSIZE + i
ENDDO
```

병렬 I/O의 사용 (3/5)

□ Fortran 코드 (계속) : 개별적인 파일 생성

```
WRITE(number, *) myrank
filename(myrank) = "testfile."//number

CALL MPI_FILE_OPEN(MPI_COMM_SELF, filename, &
    MPI_MODE_WRONLY+MPI_MODE_CREATE, MPI_INFO_NULL, &
    myfile, ierr)
CALL MPI_FILE_WRITE(myfile, buf, BUFSIZE, MPI_INTEGER, &
    MPI_STATUS_IGNORE, ierr)
CALL MPI_FILE_CLOSE(myfile, ierr)

CALL MPI_FINALIZE(ierr)
END
```

병렬 I/O의 사용 (4/5)

□ C 코드 : 개별적인 파일 생성

```
/*example of parallel MPI write into separate files */
#include <mpi.h>
#include <stdio.h>
#define BUFSIZE 100

void main (int argc, char *argv[]){
    int i, nprocs, myrank, buf[BUFSIZE] ;
    char filename[128];
    MPI_File myfile;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    for(i=0; i<BUFSIZE; i++)
        buf[i] = myrank * BUFSIZE + i;
```

병렬 I/O의 사용 (5/5)

□ C 코드 (계속): 개별적인 파일 생성

```
sprintf(filename, "testfile.%d", myrank);

MPI_File_open(MPI_COMM_SELF, filename,
              MPI_MODE_WRONLY | MPI_MODE_CREATE,
              MPI_INFO_NULL, &myfile);
MPI_File_write(myfile, buf, BUFSIZE, MPI_INT,
              MPI_STATUS_IGNORE);
MPI_File_close(&myfile);

MPI_Finalize();
}
```

병렬 I/O 루틴 : MPI_FILE_OPEN (1/2)

C	<code>int MPI_File_open(MPI_Comm comm, char *filename, int amode, MPI_Info info, MPI_File *fh)</code>
Fortran	<code>MPI_FILE_OPEN(comm, filename, amode, info, fh, ierr)</code>

INTEGER comm : 커뮤니케이터 (핸들) (IN)
CHARACTER filename : 오픈하는 파일 이름 (IN)
INTEGER amode : 파일 접근 모드 (IN)
INTEGER info : info 객체 (핸들) (IN)
INTEGER fh : 새 파일 핸들 (핸들) (OUT)

- 집합 통신 : 동일 커뮤니케이터의 프로세스는 같은 파일 오픈
 - **MPI_COMM_SELF** : 프로세스 하나로 구성되는 커뮤니케이터

병렬 I/O 루틴 : MPI_FILE_OPEN (2/2)

□ 파일 접근 모드 : OR(:C), IOR(+:Fortran)로 연결 가능

MPI_MODE_APPEND	파일 포인터의 시작위치를 파일 마지막에 설정
MPI_MODE_CREATE	파일생성, 만약 파일이 있으면 덮어쓰
MPI_MODE_DELETE_ON_CLOSE	파일을 닫으면 삭제
MPI_MODE_EXCL	파일생성시 파일이 있으면 에러 리턴
MPI_MODE_RDONLY	읽기만 가능
MPI_MODE_RDWR	읽기와 쓰기 가능
MPI_MODE_SEQUENTIAL	파일을 순차적으로만 접근가능
MPI_MODE_UNIQUE_OPEN	다른 곳에서 동시에 열 수 없음
MPI_MODE_WRONLY	쓰기만 가능

□ Info 객체 : 시스템 환경에 따른 프로그램 구현의 변화에 대한 정보 제공, 통상적으로 MPI_INFO_NULL 사용

병렬 I/O 루틴 : MPI_FILE_WRITE

C	<code>int MPI_File_write(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)</code>
Fortran	<code>MPI_FILE_WRITE(fh, buf, count, datatype, status(MPI_STATUS_SIZE), ierr)</code>

INTEGER fh : 파일 핸들 (핸들) (INOUT)
 CHOICE buf : 버퍼의 시작 주소 (IN)
 INTEGER count : 버퍼의 원소 개수 (IN)
 INTEGER datatype : 버퍼 원소의 데이터 타입 (핸들) (IN)
 INTEGER status(MPI_STATUS_SIZE) : 상태 객체 (OUT)

■ MPI_STATUS_IGNORE (MPI-2) : 상태 저장 없음

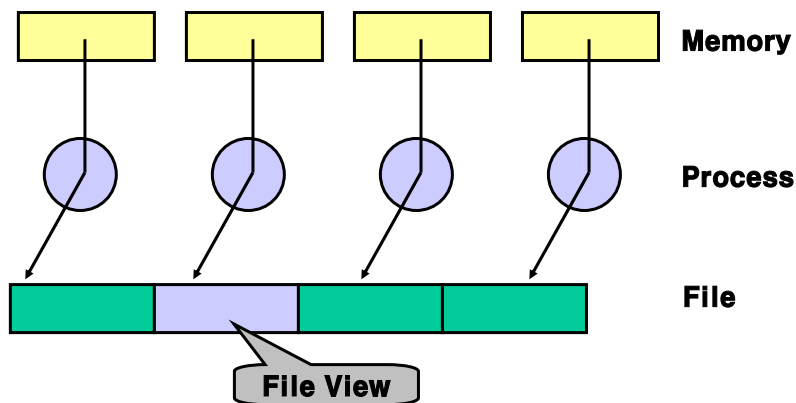
병렬 I/O 루틴 : MPI_FILE_CLOSE

C	<code>int MPI_File_close(MPI_File *fh)</code>
Fortran	<code>MPI_FILE_CLOSE(fh, ierr)</code>

INTEGER fh : 파일 핸들 (핸들) (INOUT)

병렬 프로그램과 병렬 I/O (1/6)

- 병렬 프로세스가 하나의 공유 파일 작성



병렬 프로그램과 병렬 I/O (2/6)

- 여러 프로세스 들이 하나의 파일 공유
 - `MPI_COMM_SELF` → `MPI_COMM_WORLD`
- 파일 뷰 : 공유된 파일에 각 프로세스가 접근하는 부분
 - `MPI_FILE_SET_VIEW`로 설정
- 각 프로세스의 파일 뷰 시작 위치 계산 4바이트 정수
 - `disp = myrank * BUFSIZE * 4`
 - `disp = myrank*BUFSIZE*sizeof(int);`

병렬 프로그램과 병렬 I/O (3/6)

- Fortran 코드 : 하나의 공유 파일 생성

```
PROGRAM parallel_IO_2
INCLUDE 'mpif.h'
INTEGER BUFSIZE
PARAMETER (BUFSIZE = 100)
INTEGER nprocs, myrank, ierr, buf(BUFSIZE), thefile
INTEGER(kind=MPI_OFFSET_KIND) disp
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
DO i = 1, BUFSIZE
    buf(i) = myrank * BUFSIZE + i
ENDDO
```

병렬 프로그램과 병렬 I/O (4/6)

□ Fortran 코드 (계속) : 하나의 공유 파일 생성

```
CALL MPI_FILE_OPEN(MPI_COMM_WORLD, 'testfile', &
  MPI_MODE_WRONLY + MPI_MODE_CREATE, MPI_INFO_NULL, &
  thefile, ierr)
disp = myrank * BUFSIZE * 4
CALL MPI_FILE_SET_VIEW(thefile, disp, MPI_INTEGER, &
  MPI_INTEGER, 'native', MPI_INFO_NULL, ierr)
CALL MPI_FILE_WRITE(thefile, buf, BUFSIZE, MPI_INTEGER, &
  MPI_STATUS_IGNORE, ierr)
CALL MPI_FILE_CLOSE(thefile, ierr)
CALL MPI_FINALIZE(ierr)
END
```

병렬 프로그램과 병렬 I/O (5/6)

□ C 코드 : 하나의 공유 파일 생성

```
/*example of parallel MPI write into single files */
#include <mpi.h>
#include <stdio.h>
#define BUFSIZE 100

void main (int argc, char *argv[]){
  int i, nprocs, myrank, buf[BUFSIZE] ;
  MPI_File thefile;
  MPI_Offset disp;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
```

병렬 프로그램과 병렬 I/O (6/6)

□ C 코드 (계속): 하나의 공유 파일 생성

```

for(i=0; i<BUFSIZE; i++)
    buf[i] = myrank * BUFSIZE + i;
MPI_File_open(MPI_COMM_WORLD, "testfile",
              MPI_MODE_WRONLY | MPI_MODE_CREATE,
              MPI_INFO_NULL, &thefile);
disp = myrank*BUFSIZE*sizeof(int);
MPI_File_set_view(thefile, disp, MPI_INT, MPI_INT, "native",
                  MPI_INFO_NULL);
MPI_File_write(thefile, buf, BUFSIZE, MPI_INT,
               MPI_STATUS_IGNORE);
MPI_File_close(&thefile);
MPI_Finalize();
}
    
```

병렬 I/O 루틴 : MPI_FILE_SET_VIEW (1/2)

C	int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype, MPI_Datatype filetype, char *datarep, MPI_Info info)
Fortran	MPI_FILE_SET_VIEW(fh, disp, etype, filetype, datarep, info, ierr)

INTEGER fh : 파일 핸들(IN)

INTEGER(kind=MPI_OFFSET_KIND) disp : 파일 뷰의 시작 위치(IN)

INTEGER etype : 기본 데이터 타입, 파일 안의 데이터 타입(IN)

INTEGER filetype : 파일 뷰의 데이터 타입, 유도 데이터 타입을 이용하여 뷰 접근을 불연속적으로 할 수 있도록 함(IN)

CHARACTER datarep(*) : 데이터 표현 (IN)

INTEGER info : info 객체 (IN)

병렬 I/O 루틴 : MPI_FILE_SET_VIEW (2/2)

- 커뮤니케이터의 모든 프로세스가 호출하는 집합통신 루틴
- 데이터 표현
 - 시스템에 따른 데이터 표현 방식의 기술
 - 파일의 이식성을 높여 줌
 - native :
 - 데이터가 메모리에 있는 것과 똑같이 파일에 저장 됨
 - 동종 환경 시스템에 적합하며, 대부분 사용
 - internal
 - 시스템에서 정의된 내부 포맷으로 데이터 전환
 - 동종 환경 또는 이기종 환경에서 사용 가능
 - external32
 - MPI-2 에서 정의한 포맷으로 데이터 전환

병렬 I/O : 파일 읽기 (1/5)

- 여러 프로세스가 하나의 파일을 공유하여 병렬로 읽기 가능
- 프로그램 내에서 파일 크기 계산
MPI_FILE_GET_SIZE
- 근사적으로 동일한 크기로 설정된 파일 뷰로부터 각 프로세스
는 동시에 데이터를 읽어 들임
MPI_FILE_READ

병렬 I/O : 파일 읽기 (2/5)

□ Fortran 코드

```
PROGRAM parallel_IO_3
INCLUDE 'mpif.h'
INTEGER nprocs, myrank, ierr
INTEGER count, bufsize, thefile
INTEGER (kind=MPI_OFFSET_KIND) filesize, disp
INTEGER, ALLOCATABLE :: buf(:)
INTEGER status(MPI_STATUS_SIZE)

CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
CALL MPI_FILE_OPEN(MPI_COMM_WORLD, 'testfile', &
    MPI_MODE_RDONLY, MPI_INFO_NULL, thefile, ierr)
```

병렬 I/O : 파일 읽기 (3/5)

□ Fortran 코드 (계속)

```
CALL MPI_FILE_GET_SIZE(thefile, filesize, ierr)
filesize = filesize/4
bufsize = filesize/nprocs + 1
ALLOCATE(buf(bufsize))
disp = myrank * bufsize * 4
CALL MPI_FILE_SET_VIEW(thefile, disp, MPI_INTEGER, &
    MPI_INTEGER, 'native', MPI_INFO_NULL, ierr)
CALL MPI_FILE_READ(thefile, buf, bufsize, MPI_INTEGER, &
    status, ierr)
CALL MPI_GET_COUNT(status, MPI_INTEGER, count, ierr)
print *, 'process ', myrank, 'read ', count, 'ints'
CALL MPI_FILE_CLOSE(thefile, ierr)
CALL MPI_FINALIZE(ierr)
END
```

병렬 I/O : 파일 읽기 (4/5)

□ C 코드

```
/* parallel MPI read with arbitrary number of processes */
#include <mpi.h>
#include <stdio.h>

void main (int argc, char *argv[]){
    int nprocs, myrank, bufsize, *buf, count;
    MPI_File thefile;
    MPI_Status status;
    MPI_Offset filesize;

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

병렬 I/O : 파일 읽기 (5/5)

□ C 코드 (계속)

```
MPI_File_open(MPI_COMM_WORLD, "testfile", MPI_MODE_RDONLY,
              MPI_INFO_NULL, &thefile);
MPI_File_get_size(thefile, &filesize);
filesize = filesize / sizeof(int);
bufsize = filesize / nprocs + 1;
buf = (int *) malloc(bufsize * sizeof(int));
MPI_File_set_view(thefile, myrank*bufsize*sizeof(int),
                  MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
MPI_File_read(thefile, buf, bufsize, MPI_INT, &status);
MPI_Get_count(&status, MPI_INT, &count);
printf("process %d read%c ints \n ", myrank, count);
MPI_File_close(&thefile);
MPI_Finalize();
}
```

병렬 I/O 루틴 : MPI_FILE_GET_SIZE

C	<code>int MPI_File_get_size(MPI_File fh, MPI_Offset *size)</code>
Fortran	<code>MPI_FILE_GET_SIZE(fh, size, ierr)</code>

INTEGER fh : 파일 핸들 (핸들) (IN)

INTEGER (kind=MPI_OFFSET_KIND) size : 파일의 크기 (OUT)

- 파일의 크기를 바이트 단위로 저장

병렬 I/O 루틴 : MPI_FILE_READ

C	<code>int MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)</code>
Fortran	<code>MPI_FILE_READ(fh, buf, count, datatype, status(MPI_STATUS_SIZE), ierr)</code>

INTEGER fh : 파일 핸들 (핸들) (INOUT)

CHOICE buf : 버퍼의 시작 주소 (OUT)

INTEGER count : 버퍼의 원소 개수 (IN)

INTEGER datatype : 버퍼 원소의 데이터 타입 (핸들) (IN)

INTEGER status : 상태 객체 (OUT)

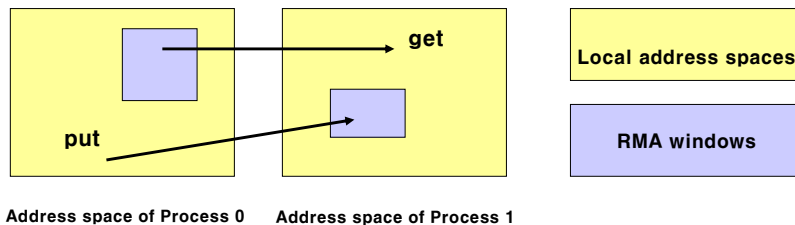
- 파일에서 지정된 개수의 데이터를 읽어 들여 버퍼에 저장

일방 통신

- 메시지 패싱 모델의 통신
 - 한 쌍의 통신 연산(송신과 수신)을 통한 데이터 전송
- 일방 통신 (one-sided communication)
 - 송/수신 조합이 아닌 한 쪽 프로세스 만으로 데이터 전송 가능
 - 원격 메모리 접근(RMA)
 - 알고리즘 설계를 위한 유연성 제공
 - get, put, accumulate 등

메모리 윈도우

- 단일 프로세스의 메모리 일부
- 다른 프로세스들에게 메모리 연산을 허용하는 공간
 - 메모리 연산 : 읽기(get), 쓰기(put), 갱신(accumulate)
- MPL_WIN_CREATE으로 생성



MPI 프로그램 : PI 계산 (1/4)

□ Fortran 코드

```
PROGRAM parallel_pi
INCLUDE 'mpif.h'
DOUBLE PRECISION mypi, pi, h, sum, x
LOGICAL continue
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
continue = .TRUE.
DO WHILE(continue)
  IF(myrank==0) THEN
    PRINT*, 'Enter the Number of intervals: (0 quits)'
    READ*, n
  ENDIF
  CALL MPI_BCAST(n, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
  IF(n==0) THEN
    continue = .FALSE.
    GOTO 10
  ELSE
```

MPI 프로그램 : PI 계산 (2/4)

□ Fortran 코드 (계속)

```
  h = 1.0d0/DBLE(n)
  sum=0.0d0
  DO i=myrank+1, n, nprocs
    x = h*(DBLE(i)-0.5d0)
    sum = sum + 4.0d0/(1.0d0+x*x)
  ENDDO
  mypi = h*sum
  CALL MPI_REDUCE(mypi, pi, 1, MPI_DOUBLE_PRECISIN, &
    MPI_SUM, 0, MPI_COMM_WORLD, ierr)
  IF(myrank==0) THEN
    PRINT*, 'pi is approximately ', pi
  ENDIF
  ENDIF
ENDDO
10 CALL MPI_FINALIZE(ierr)
END
```

MPI 프로그램 : pi 계산 (3/4)

□ C 코드

```
#include <mpi.h>
void main (int argc, char *argv[]){
    int n, i, myrank, nprocs;
    double mypi, x, pi, h, sum;
    MPI_Init(&argc, &argv) ;
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank) ;
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs) ;
    while(1){
        if(myrank==0) {
            printf("Enter the Number of Intervals: (0 quits)\n");
            scanf("%d", &n);
        }
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

MPI 프로그램 : pi 계산 (3/4)

□ C 코드 (계속)

```
if(n==0) break;
else{
    h = 1.0/(double) n;
    sum=0.0;
    for (i=myrank; i<n ; i+=nprocs) {
        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(myrank==0)
        printf("pi is approximately %f \n", pi );
}
MPI_Finalize();
}
```

일방 통신을 이용한 병렬화 코드 (1/7)

- 메모리 윈도우 생성
MPL_WIN_CREATE
- 일방 통신 루틴을 이용한 데이터 전송
MPL_BCAST → MPL_GET
MPL_REDUCE → MPL_ACCUMULATE
- 동기화 루틴
MPL_WIN_FENCE

일방 통신을 이용한 병렬화 코드 (2/7)

□ Fortran 코드

```
PROGRAM PI_RMA
INCLUDE 'mpif.h'
INTEGER nwin
DOUBLE PRECISION piwin
DOUBLE PRECISION mypi, pi, h, sum, x
LOGICAL continue
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
IF(myrank==0) THEN
    CALL MPI_WIN_CREATE(n, 4, 1, MPI_INFO_NULL, MPI_COMM_WORLD, &
        nwin, ierr)
    CALL MPI_WIN_CREATE(pi, 8, 1, MPI_INFO_NULL, MPI_COMM_WORLD, &
        piwin, ierr)
ELSE
    CALL MPI_WIN_CREATE(MPI_BOTTOM, 0, 1, MPI_INFO_NULL, &
        MPI_COMM_WORLD, nwin, ierr)
    CALL MPI_WIN_CREATE(MPI_BOTTOM, 0, 1, MPI_INFO_NULL, &
        MPI_COMM_WORLD, piwin, ierr)
ENDIF
ENDIF
```


일방 통신을 이용한 병렬화 코드 (3/7)

□ Fortran 코드 (계속)

```
continue=.TRUE.
DO WHILE(continue)
  IF(myid == 0) THEN
    PRINT*, 'Enter the Number of intervals: (0 quits)'
    READ*, n
    pi=0.0d0
  ENDIF
  CALL MPI_WIN_FENCE(0, nwin, ierr)
  IF(myrank /= 0) THEN
    CALL MPI_GET(n, 1, MPI_INTEGER, 0, 0, 1, MPI_INTEGER, &
      nwin, ierr)
  ENDIF
  CALL MPI_WIN_FENCE(0, nwin, ierr)
  IF(n==0) THEN
    continue = .FALSE.
    GOTO 10
  ELSE
    h = 1.0d0/DBLE(n)
    sum = 0.0d0
```

일방 통신을 이용한 병렬화 코드 (4/7)

□ Fortran 코드 (계속)

```
DO i=myrank+1, n, nprocs
  x = h*(DBLE(i)-0.5d0)
  sum = sum + 4.0d0/(1.0d0+x*x)
ENDDO
mypi = h*sum
CALL MPI_WIN_FENCE(0, piwin, ierr)
CALL MPI_ACCUMULATE(mypi, 1, MPI_DOUBLE_PRECISION, 0, 0,&
  1, MPI_DOUBLE_PRECISION, MPI_SUM, piwin, ierr)
CALL MPI_WIN_FENCE(0, piwin, ierr)
IF(myrank==0) THEN
  PRINT*, 'pi is approximately ', pi
ENDIF
ENDIF
ENDDO
CALL MPI_WIN_FREE(nwin, ierr)
CALL MPI_WIN_FREE(piwin, ierr)
10 CALL MPI_FINALIZE(ierr)
END
```

일방 통신을 이용한 병렬화 코드 (5/7)

□ C 코드

```
#include <mpi.h>
void main (int argc, char *argv[]){
    int n, i, myrank, nprocs;
    double pi, mypi, x, h, sum;
    MPI_Win nwin, piwin;
    MPI_Init(&argc, &argv) ;
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank) ;
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs) ;
    if (myrank==0) {
        MPI_Win_create(&n, sizeof(int), 1, MPI_INFO_NULL,
                      MPI_COMM_WORLD, &nwin);
        MPI_Win_create(&pi, sizeof(double), 1, MPI_INFO_NULL,
                      MPI_COMM_WORLD, &piwin);
    }
    else{
        MPI_Win_create(MPI_BOTTOM, 0, 1, MPI_INFO_NULL,
                      MPI_COMM_WORLD, &nwin);
        MPI_Win_create(MPI_BOTTOM, 0, 1, MPI_INFO_NULL,
                      MPI_COMM_WORLD, &piwin);
    }
}
```

일방 통신을 이용한 병렬화 코드 (6/7)

□ C 코드 (계속)

```
}
while(1){
    if(myrank==0) {
        printf("Enter the Number of Intervals: (0 quits)\n");
        scanf("%d", &n);
        pi=0.0;
    }
    MPI_Win_fence(0, nwin);
    if(myrank != 0)
        MPI_Get(&n, 1, MPI_INT, 0, 0, 1, MPI_INT, nwin);
    MPI_Win_fence(0, nwin);
    if(n==0) break;
    else{
        h = 1.0/(double) n;
        sum=0.0;
        for (i=myrank+1; i<=n ; i+=nprocs) {
            x = h*((double)i-0.5);
            sum += 4.0/(1.0+x*x);
        }
    }
}
```

일방 통신을 이용한 병렬화 코드 (7/7)

□ C 코드 (계속)

```

    mypi = h*sum;
    MPI_Win_fence(0, piwin);
    MPI_Accumulate(&mypi, 1, MPI_DOUBLE, 0, 0, 1,
                  MPI_DOUBLE, MPI_SUM, piwin);
    MPI_Win_fence(0, piwin);
    if(myrank==0) printf("pi is approximately %f \n", pi);
}
}
MPI_Win_free(&nwin);
MPI_Win_free(&piwin);
MPI_Finalize();
}

```

일방 통신 루틴 : MPI_WIN_CREATE (1/2)

C	int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, MPI_Win *win)
Fortran	MPI_WIN_CREATE(base, size, disp_unit, info, comm, win, ierr)

CHOICE base : 윈도우의 시작 주소 (IN)

INTEGER size : 바이트로 나타낸 윈도우의 크기(음 아닌 정수) (IN)

INTEGER disp_unit : 바이트로 나타낸 범위의 크기 (양의 정수) (IN)

INTEGER info : info 객체 (핸들) (IN)

INTEGER comm : 커뮤니케이터 (핸들) (IN)

INTEGER win : 리턴 되는 윈도우 객체 (핸들) (OUT)

□ 메모리 윈도우 생성 루틴

□ 커뮤니케이터 내부의 모든 프로세스들이 참여하는 집합통신

일방 통신 루틴 : MPL_WIN_CREATE (2/2)

- `CALL MPI_WIN_CREATE(n, 4, 1, MPI_INFO_NULL, MPI_COMM_WORLD, nwin, ierr)`
 - 프로세스 0의 정수 n에 접근 허용하는 윈도우 객체 nwin 생성, 윈도우 시작 주소는 n, 길이는 4 바이트 임을 나타냄
 - 변수가 하나인 윈도우 객체이므로 변수간의 변위는 의미 없음
 - 커뮤니케이터내의 모든 프로세스는 직접 n값을 get할 수 있다.

- `CALL MPI_WIN_CREATE(MPI_BOTTOM, 0, 1, MPI_INFO_NULL, MPI_COMM_WORLD, nwin, ierr)`
 - 다른 프로세스에서는 접근 허용하는 윈도우 생성이 없음을 나타내기 위해 주소는 MPI_BOTTOM, 길이를 0으로 두었음

일방 통신 루틴 : MPL_WIN_FENCE

C	<code>int MPI_Win_fence(int assert, MPI_Win *win)</code>
Fortran	<code>MPI_WIN_FENCE(assert, win, ierr)</code>

INTEGER assert : 성능 향상 관련 인수, 0은 항상 허용됨 (IN)
 INTEGER win : 펜스연산이 수행되는 윈도우 객체 (IN)

- 원격 연산에서의 동기화 함수
- 원격 연산에서는 MPL_BARRIER를 쓸 수 없음
- 원격 연산과 지역 연산 또는 두 개의 원격 연산 사이를 분리 시켜 줌
- 원격 연산은 논블록킹이기 때문에 연산의 완료를 확인하기 위해서는 반드시 동기화 함수를 호출해야 함

일방 통신 루틴 : MPI_GET (1/2)

C	<pre>int MPI_Get(void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win win)</pre>
Fortran	<pre>MPI_GET(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win, ierr)</pre>

CHOICE origin_addr : 데이터를 가져오는(get) 버퍼(원 버퍼)의 시작 주소 (IN)

INTEGER origin_count : 원 버퍼의 데이터 개수 (IN)

INTEGER origin_datatype : 원 버퍼의 데이터 타입 (핸들) (IN)

INTEGER target_rank : 메모리 접근을 허용하는 목적 프로세스의 랭크 (IN)

INTEGER target_disp : 윈도우 시작 위치에서 목적 버퍼까지의 변위 (IN)

INTEGER target_count : 목적 버퍼의 데이터 원소 개수 (IN)

INTEGER target_datatype : 목적 버퍼 원소의 데이터 타입 (핸들) (IN)

INTEGER win : 윈도우 객체 (핸들) (IN)

일방 통신 루틴 : MPI_GET (2/2)

❑ `CALL MPI_GET(n, 1, MPI_INTEGER, 0, 0, 1, &MPI_INTEGER, nwin, ierr)`

● 수신지 정보 (n, 1, MPI_INTEGER)

- MPI_INTEGER 타입의 1개 데이터를 n에 저장

● 송신지 정보 (0, 0, 1, MPI_INTEGER)

- 0번 프로세스의 윈도우 시작위치에서 0만큼 떨어져 있는 MPI_INTEGER 타입 데이터를 1개 가져옴

일방 통신 루틴 : MPI_PUT

C	<pre>int MPI_Put(void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win win)</pre>
Fortran	<pre>MPI_PUT(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win, ierr)</pre>

CHOICE origin_addr : 데이터를 보내는(put) 버퍼(원 버퍼)의 시작 주소 (IN)
 INTEGER origin_count : 원 버퍼의 데이터 개수 (IN)
 INTEGER origin_datatype : 원 버퍼의 데이터 타입 (핸들)(IN)
 INTEGER target_rank : 메모리 접근을 허용하는 프로세스의 랭크 (IN)
 INTEGER target_disp : 윈도우 시작점에서 목적 버퍼까지의 변위 (IN)
 INTEGER target_count : 목적 버퍼의 데이터 원소 개수 (IN)
 INTEGER target_datatype : 목적 버퍼 원소의 데이터 타입 (핸들) (IN)
 INTEGER win : 윈도우 객체 (핸들) (IN)

일방 통신 루틴 : MPI_ACCUMULATE (1/2)

C	<pre>int MPI_Accumulate(void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)</pre>
Fortran	<pre>MPI_ACCUMULATE(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, op, win, ierr)</pre>

CHOICE origin_addr : 데이터를 갱신(accumulate) 하는 버퍼(원 버퍼)의
시작 주소 (IN)
 INTEGER origin_count : 원 버퍼의 데이터 개수 (IN)
 INTEGER origin_datatype : 원 버퍼의 데이터 타입 (핸들) (IN)
 INTEGER target_rank : 메모리 접근을 허용하는 프로세스의 랭크 (IN)
 INTEGER target_disp : 윈도우 시작점에서 목적 버퍼까지의 변위 (IN)
 INTEGER target_count : 목적 버퍼의 데이터 원소 개수 (IN)
 INTEGER target_datatype : 목적 버퍼 원소의 데이터 타입 (핸들) (IN)
 INTEGER op : 환산(reduction) 연산 (IN)
 INTEGER win : 윈도우 객체 (핸들) (IN)

일방 통신 루틴 : MPI_ACCUMULATE (2/2)

- `CALL MPI_ACCUMULATE(mympi, 1, &MPI_DOUBLE_PRECISION, 0, 0, 1, &MPI_DOUBLE_PRECISION, MPI_SUM, piwin, ierr)`
 - 갱신에 사용될 지역 변수 정보 (mympi, 1, MPI_DOUBLE_PRECISION)
 - mympi에서 시작되는 MPI_DOUBLE_PRECISION 타입의 1개 데이터를 목적지의 윈도우 정보 갱신에 이용
 - 갱신할 목적지 정보 (0, 0, 1, MPI_DOUBLE_PRECISION, MPI_SUM)
 - 0번 프로세스의 윈도우 시작위치에서 0만큼 떨어져 있는 데이터



일방 통신 루틴 : MPI_WIN_FREE

C	<code>int MPI_Win_free(MPI_Win *win)</code>
Fortran	<code>MPI_WIN_FREE(win, ierr)</code>

INTEGER win : 윈도우 객체 (핸들) (IN)

- 윈도우 객체를 풀어 널(null) 핸들을 리턴 함



용어정리/참고자료



용어정리 (1/3)

- MPI : Message Passing Interface
- 봉투 : Envelope
- 꼬리표 : Tag
- 식별자 : Identifier
- 유도 데이터 타입 : Derived Data Type
- 가상 토폴로지 : Virtual Topology
- 핸들 : Handle
- 송신 : Send
- 수신 : Receive
- 동기 송신 : Synchronous Send
- 준비 송신 : Ready Send
- 버퍼 송신 : Buffered Send
- 표준 송신 : Standard Send
- 포스팅 : Posting
- 교착 : Deadlock
- 통신 부하 : Communication Overhead
- 점대점 통신 : Point-To-Point Communication
- 집합 통신 : Collective Communication
- 단 방향 통신 : Unidirectional Communication

용어정리 (2/3)

- 양 방향 통신 : Bidirectional Communication
- 취합 : Gather
- 환산 : Reduction
- 연산자 : Operator
- 확산 : Scatter
- 장벽 : Barrier
- 커먼 블록 : Common Block
- 유사 타입 : Pseudo Type
- 부분 행렬 : Subarray
- 행 우선 순 : Row major Order
- 열 우선 순 : Column Major Order
- 메모리 대응 : Memory Mapping
- 직교 가상 토폴로지 : Cartesian Virtual Topology
- 그래프 가상 토폴로지 : Graph Virtual Topology
- 대응 함수 : Mapping Function
- 블록 분할 : Block Distribution
- 순환 분할 : Cyclic Distribution
- 블록-순환 분할 : Block-Cyclic Distribution
- 로드 밸런싱 : Load Balancing

용어정리 (3/3)

- 캐시미스 : Cache Miss
- 배열 수축 : Shrinking Array
- 내포된 루프 : Nested Loop
- 유한 차분법 : Finite Difference Method(FDM)
- 의존성 : dependence
- 대량 데이터 : Bulk Data
- 중첩 : Superposition
- 비틀림 분해 : Twisted Decomposition
- 프리픽스 합 : Prefix Sum
- 임의 행로 : Random Walk
- 분자 동역학 : Molecular Dynamics
- 원격 메모리 접근 : Remote Memory Access(RMA)
- 일방 통신 : One sided Communication
- 동적 메모리 운영 : Dynamic Memory Management
- 객체 : Object

참고자료(1/2)

- ❑ Gropp, Lusk, and Skjellum. *Using MPI*, second edition. MIT Press. 1999
- ❑ Gropp, Lusk, and Thakur. *Using MPI-2*, second edition. MIT Press. 1999
- ❑ Snir, Otto, Huss-Lederman, Walker, and Dongarra. *MPI-The Complete Reference Volume 1*. Second Edition. MIT Press. 1998.
- ❑ Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley. 2000.
- ❑ SP Parallel Programming Workshop
<http://www.mhpc.edu/training/workshop/>
- ❑ Parallel Programming Concepts
<http://www.tc.cornell.edu/Services/Edu/Topics/ParProgCons/more.asp>

참고자료(2/2)

- ❑ Introduction to Parallel Computing
http://foxtrot.ncsa.uiuc.edu:8900/SCRIPT/HPC/scripts/serve_home
- ❑ Practical MPI Programming
<http://www.redbooks.ibm.com/pubs/pdfs/redbooks/sg245380.pdf>
- ❑ Lawrence Livermore National Laboratory
<http://www.llnl.gov/computing/tutorials/workshops/workshop/>
- ❑ Parallel Programming with MPI
<http://oscinfo.osc.edu/training/mpi/>

기술지원

- Helpdesk
 - www.supercomputing.re.kr
- 교육센터 게시판
 - webedu.supercomputing.re.kr
- E-Mail
 - consult@supercomputing.re.kr