Advanced MPI

Hongsuk Yi
(hsyi@kisti.re.kr)

KISTI Supercomputing Center

MPI-2 소개

- MPI-2에 추가된 영역
  1. 동적 프로세스 운영
  2. 원격 메모리 접근 (일방 통신)
  3. 병렬 I/O (MPI I/O)
  4. 언어 인터페이스
  5. 확장된 집합통신
  6. 기타

Dynamic Process Management Overview

프로세스 생성과 관리

- 목적
  - starting new MPI processes
  - connecting independently started MPI processes
- 이슈
  - maintaining simplicity, flexibility, and correctness
  - interaction with operating systems, resource manager, and process manager
- Spawn interfaces:
  - at initiators (parents)
  - at spawned processes (children)

프로세스 생성과 관리

- MPI-I
  - 프로세스를 생성하거나 생성된 프로세스 사이에 통신할 수 없다.
  - 어떤 프로세스도 더하거나 지울 수 없다.
- 프로세스 생성과 관리 (MPI-2)
  - MPI 어플리케이션이 시작된 후 새로운 프로세스의 생성과 제거
  - 이미 존재하는 MPI 어플리케이션 사이에 통신

프로세스 생성과 관리

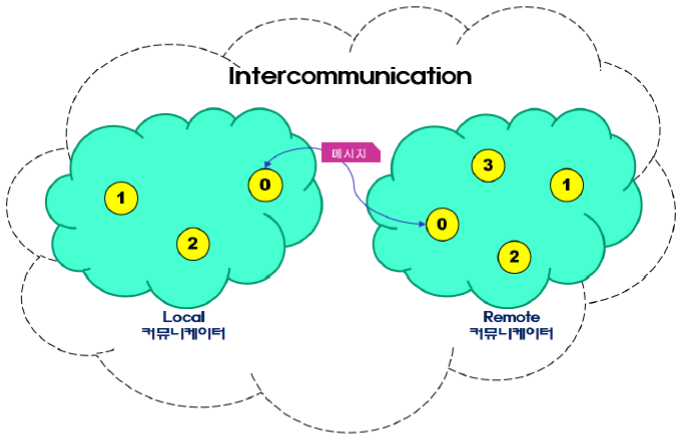


그림 7.4: InterCommunication : 임의의 다른 프로세스 그룹과의 통신

프로세스 관리: MPI_COMM_SPAWN (1/2)

C	`MPI_Comm_spawn(int char *command, char *argv[], int maxprocs, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *intercomm,int array_of_errcodes[])`
Fortran	`MPI_COMM_SPAWN(command, argv, maxprocs, info, root, comm, intercomm, array_of_errcodes, ierr)`

- CHARACTER(*) command : 프로그램 이름 (IN)
- CHARACTER(*) argv(*) : 프로그램에 대한 인수들 (IN)
  만약 인수가 없다면 MPI_ARGV_NULL 인수 사용
- INTEGER maxprocs : 생성할 프로세스 개수(IN)
- INTEGER info : 프로세스들을 어떻게, 어디서 생성할 지에 대한 key 정보(IN)
- INTEGER root : 프로세스를 생성할 프로세스 rank (IN)
- INTEGER comm : 생성할 프로세스들을 담고 있는 커뮤니케이터 (IN)
- INTEGER intercomm : 기존 커뮤니케이터와 생성된 프로세스 그룹들 사이의 통신할 수 있는 intercommunicator 핸들 (OUT)
- INTEGER array of errcodes(*) : 프로세스 별 에러 코드 (OUT)

프로세스 관리: MPI_COMM_SPAWN (2/2)

- command, argv 변수 사용 예
  - 프로세스 생성 : ./children.exe –arg1 –arg2 –arg3

C	`char *command;`
	`Char **argv;`
	`command = "./children.exe";`
	`argv = (char **)malloc(3 * sizeof(char *));`
	`argv[0] = "-arg1";`
	`argv[1] = "-arg2";`
	`argv[2] = "-arg3";`
	`MPI_Comm_spawn(command, argv, … );`

Fortran	`CHAREACTER*25 command, argv(3)`
	`command = './children.exe'`
	`argv(1) = '-arg1'`
	`argv(2) = '-arg2'`
	`argv(3) = '-arg3'`
	`CALL MPI_COMM_SPAWN(command, argv, … )`

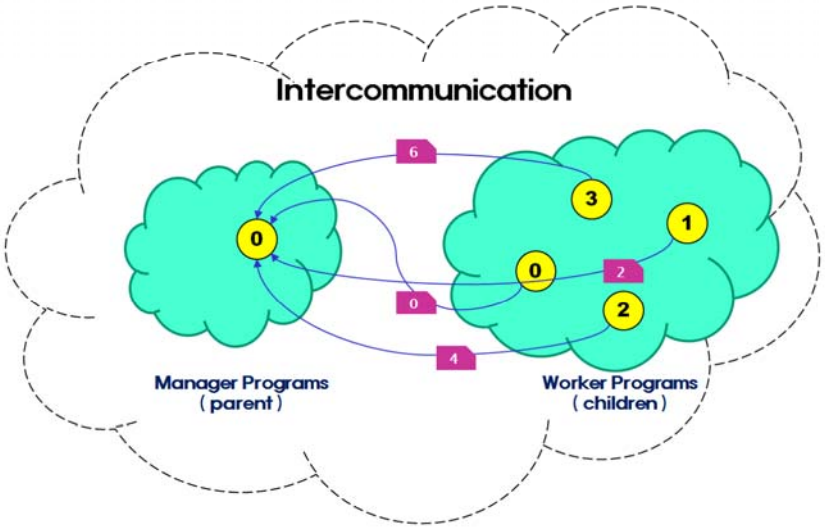
프로세스 관리: MPI_COMM_GET_PARENT

C	`int MPI_Comm_get_parent(MPI_Comm *parent)`
Fortran	`MPI_COMM_GET_PARENT(parent, ierr)`

- INTEGER parent : 부모 커뮤니케이터 핸들 (OUT)

MPI_COMM_SPAWN 루틴으로 생성된 프로세스에서MPI_COMM_PARENT 루틴을 사용해서 부모 intercommunicator 핸들을 리턴 받을 수 있다. 만약 생성된 프로세스가 아니라면 MPI_COMM_NULL 핸들을 리턴 한다.

MPI_COMM_SPAWN 예제 (1/6)



MPI_COMM_SPAWN 예제 (2/6)

```
! Manager Program
PROGRAM manager
IMPLICIT NONE
INCLUDE 'mpif.h'
INTEGER i, a, nprocs, myrank, nWorker, flag, ierr
INTEGER req, status, children
CHARACTER*100 worker_program
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
nWorker = 4
worker_program = './Worker.x'
CALL MPI_COMM_SPAWN(worker_program, MPI_ARGV_NULL, nWorker, &
MPI_INFO_NULL,0,MPI_COMM_WORLD, children, MPI_ERRCODES_IGNORE, ierr)
a = 0
DO i=1,nWorker
CALL MPI_RECV(a, 1, MPI_INTEGER, i-1, 1, children, status, ierr)
PRINT *, ' Receive a :', a, 'in intercommunicator rank:', i-1
ENDDO
CALL MPI_COMM_FREE(children, ierr)
CALL MPI_FINALIZE(ierr)
END
```

MPI_COMM_SPAWN 예제 (3/6)

```
! Worker Program
PROGRAM worker
IMPLICIT NONE
INCLUDE 'mpif.h'
INTEGER i, a, size, nprocs, myrank, ierr
INTEGER req, status, parent
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
CALL MPI_COMM_GET_PARENT(parent, ierr)
CALL MPI_COMM_REMOTE_SIZE(parent, size, ierr)
a = myrank*2
CALL MPI_SEND(a, 1, MPI_INTEGER, 0, 1, parent, ierr)
CALL MPI_FINALIZE(ierr)
END
```

MPI_COMM_SPAWN 예제 (4/6)

```
/* Manager Program */
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{
  int i, a, nprocs, nWorker, myrank, flag, ierr;
  MPI_Request req;
  MPI_Status status;
  MPI_Comm children; /* intercommunicator */
  char worker_program[100];
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  nWorker = 4;
  sprintf(worker_program, "./Worker.x");
  MPI_Comm_spawn(worker_program, MPI_ARGV_NULL, nWorker,
            MPI_INFO_NULL, 0, MPI_COMM_WORLD, &children,
                                    MPI_ERRCODES_IGNORE);
  if(children == MPI_COMM_NULL) printf("spawn fail.. \n");
  a = 0;
```

MPI_COMM_SPAWN 예제 (5/6)

```
  for(i=0;i<nWorker;i++){
    MPI_Recv(&a, 1, MPI_INT, i, 1, children, &status);
    printf(" Receive a : %d in intercommunicator rank : %d\n", a, i);
  }
  MPI_Comm_free(&children);
  MPI_Finalize();
  }
```

MPI_COMM_SPAWN 예제 (6/6)

```
/* Worker Program */
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[])
{
  int i, size, myrank, nprocs, a, ierr;
  MPI_Request req;
  MPI_Status status;
  MPI_Comm parent;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_get_parent(&parent);
  MPI_Comm_remote_size(parent, &size);
  a = myrank*2;
  MPI_Send(&a, 1, MPI_INT, 0, 1, parent);
  MPI_Finalize();
}
```

One-Sided Operations

One-Sided Operations

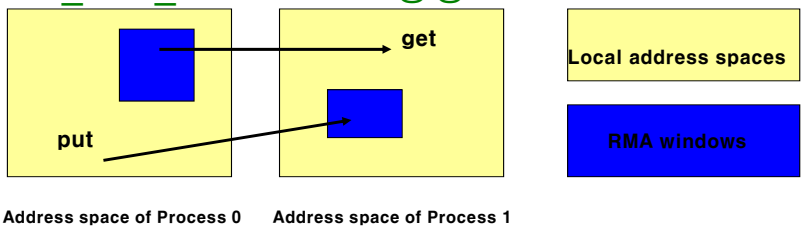
- 목적
  - PUT and GET data to/from memory of other processes
- 이슈
  - Synchronization is separate from data movement
  - Automatically dealing with subtle memory behavior: cache coherence, sequential consistency
  - Balancing efficiency and portability across a wide class of architectures, i.e., SMP, NUMA architecture
- Interface
  - PUTs and GETs are surrounded by special synchronization calls

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)
- MPI_WIN_CREATE으로 생성



MPI 프로그램 : PI 계산 (1/4)

```
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)

```
        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_PRECISION, &
                        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)

```
#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)

```
      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 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 변위의 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 : MPI_WIN_CREATE (2/2)

- `CALL MPI_WIN_CREATE(n,4,4,`
  `MPI_INFO_NULL,MPI_COMM_WORLD,nwin,err)`
  - 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 : MPI_WIN_FENCE

C	`int MPI_Win_fence(int assert, MPI_Win *win)`
Fortran	`MPI_WIN_FENCE(assert, win, ierr)`

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

- 원격 연산에서의 동기화 함수
- 원격 연산에서는 MPI_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	`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)`
Fortran	`MPI_GET(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win, ierr)`

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	`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)`
Fortran	`MPI_PUT(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win, ierr)`

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	`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)`
Fortran	`MPI_ACCUMULATE(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, op, win, ierr)`

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(mypi, 1, &
  MPI_DOUBLE_PRECISION, 0, 0, 1, &
  MPI_DOUBLE_PRECISION,MPI_SUM, piwin, ierr)
  - 갱신에 사용될 지역 변수 정보

    (mypi,1,MPI_DOUBLE_PRECISON)
    - mypi에서 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 1개를 연산 MPI_SUM을 이용해 갱신

일방 통신 루틴 : MPI_WIN_FREE

C	`int MPI_Win_free(MPI_Win *win)`
Fortran	`MPI_WIN_FREE(win, ierr)`

INTEGER win : 윈도우 객체 (핸들) (IN)

- 윈도우 객체를 풀어 널(null) 핸들을 리턴 함

Outline

- Posix I/O
  - Single reader/writer
  - All read/write
  - Subset reads/writes
- MPI I/O
  - Basic concepts
  - Non-contiguous data
  - Independent vs. collective I/O
    - Parallel I/O materials by Jussi
    - CSC Summer School,
    - Finland
    - http://www.csc.fi/english/csc/courses/archive/summerschool2011

33

Parallel I/O

- I/O (Input/output) is needed in all programs but is often overlooked
- Good I/O is non-trivial
  - Performance, scalability, reliability
  - Ease of use of output (number of files, format)
- Portability
- One cannot achieve all of the above – one needs to decide what is most

34

Parallel I/O

- New challenges for peta-/exa-scale computing
  - Number of tasks is rising rapidly
    - 10,000 ~ 500,000 cores
  - The size of the data is also rapidly increasing
  - The need for I/O tuning is algorithm & problem specific
- Without parallelization, I/O will become scalability bottleneck for practically every application!

35

I/O layers

- High-level
  - application: need to write or read data from disk
- Intermediate
  - libraries or system tools for I/O
  - high-level libraries (HDF5, netcdf)
  - MPI-I/O
  - POSIX system calls (fwrite / WRITE)
- Low-level:
  - parallel filesystem enables the actual parallel I/O
  - Lustre, GPFS, PVFS, dCache

36

POSIX I/O

- POSIX (a.k.a. IEEE standard 1003.1)
  - an international standard defining the basic system-level interfaces an operating system should provide.
  - Started in the late 1980s to try to simplify portability issues between UNIX variants
  - Also implemented by many non-UNIX-derived operating systems

37

POSIX I/O

- Built in language constructs for performing I/O
  - WRITE/READ/OPEN/CLOSE in Fortran
  - stdio.h routines in C (fopen, fread, fwrite, …)
- No parallel ability built in
  - all parallel I/O schemes have to be programmed manually
- Binary output not necessarily portable

38

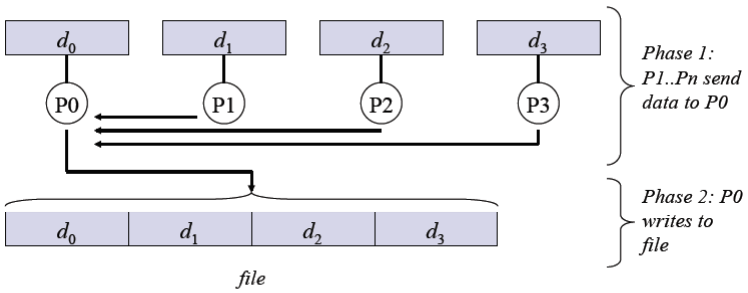
POSIX I/O

- C and Fortran binary output not necessarily compatible
- Non-contiguous access difficult to implement efficiently
- Contiguous access can be very fast

39

Parallel I/O : Simple Solution

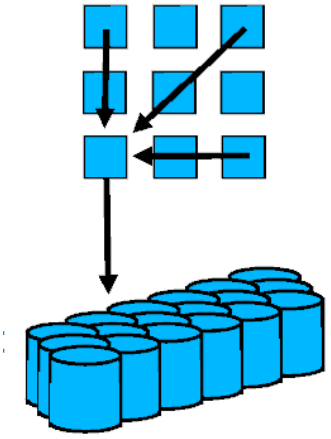
- Processes send data to P0
- P0 performs the file I/O



40

Parallel POSIX I/O

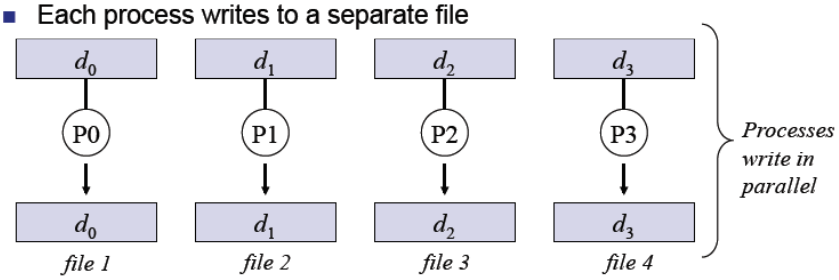
- Single writer/reader reduction
  - One process takes care of all I/O using normal (POSIX) routines
  - Requires a lot of communication
  - Writing/reading slow, single writer not able to fully utilize file-system
  - Does not scale, single writer is a bottleneck
  - Can be good option when the amount of data is small (e.g. input files)



41

Parallel I/O on Multiple files

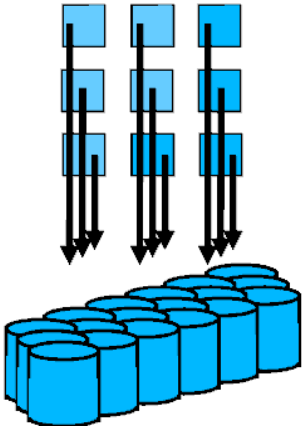
- Each process writes to a separate file



42

Parallel POSIX I/O

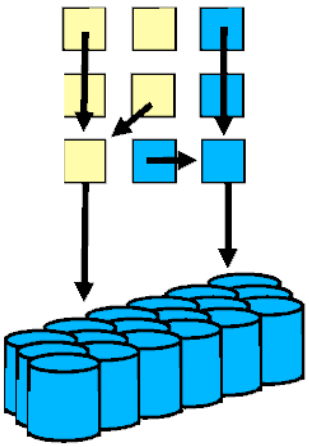
- N writers/readers to N files
  - Each process writes its local results to a separate file
  - Good bandwidth
  - Difficult to handle a huge number of files in later analysis
  - Can overwhelm filesystem



43

Parallel POSIX I/O

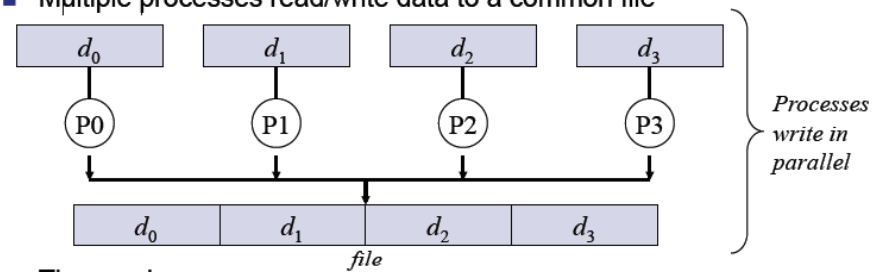
- Subset of writers/readers
  - Good compromise
  - Most difficult to implement
  - Number of readers/writers often chosen to be sqrt(N)
  - If readers/writers are dedicated then some computational resources are wasted



44

Parallel I/O Library support

- Multiple processes read/write data to a common file



45

Exercises 1 : Posix I/O

1. Write numbers 0-399 to a file in parallel.
   - You may assume that number of processes is a factor of 100. Implement a case where a single process performs the writing.
2. Verify the write in 1) by reading the file. Use different number of processes than in writing.
3. Implement the above write so that all the processes write to a different file.
4. Implement the write so that subset of processes performs the actual write.

46

MPI I/O

- MPI I/O was introduced in MPI-2
- Defines parallel operations for reading and writing files
  - I/O to only one file and/or to many files
  - Contiguous and non-contiguous I/O
  - Individual and collective I/O
  - Asynchronous I/O
- Portable programming interface

47

MPI I/O

- Potentially good performance
  - Easy to use (compared with implementing the same algorithms on your own)
- Used as the backbone of many parallel I/O
- libraries such as parallel NetCDF and parallel HDF5
- By default, binary files are not necessarily portable

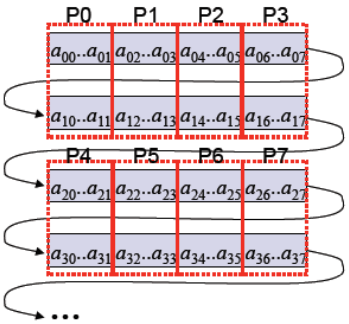
48

Parallel I/O Example

8×8 array (block,block)-distributed over 16 processors

P0	P1	P2	P3
P4	P5	P6	P7
P8	P9	P10	P11
P12	P13	P14	P15

Row-major array layout in file



Each process has a different "view" of the file content and starts reading at a different displacement and skipping "holes"

For example: P0 reads bytes 0..7, 64..71, P1 reads 8..15, 72..79, …

49

Basic concepts in MPI I/O

- MPI-IO defines the following concepts:
  - Etype (elementary data type)
    - A basic data type or derived data type
  - File
    - Is an ordered collection of etypes
    - Opened and manipulated collectively by a group of processes
  - File view
    - part of a file which is visible to process
    - enables efficient noncontiguous access to file
  - File type
    - The etype used to create a view
  - Displacement (offseet)
    - Defines the location where a view begins with respect to a process

50

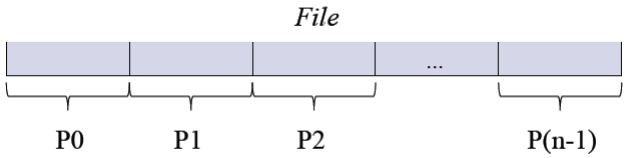
Basic concepts in MPI I/O

- File handle
  - data structure which is used for accessing the file
- File pointer
  - position in the file where to read or write
  - can be individual for all processes or shared between the processes
  - accessed through file handle
- Collective and independent I/O
  - collective: MPI coordinates the reads and writes of processes
  - independent: no coordination by MPI

51

MPI-I/O file views

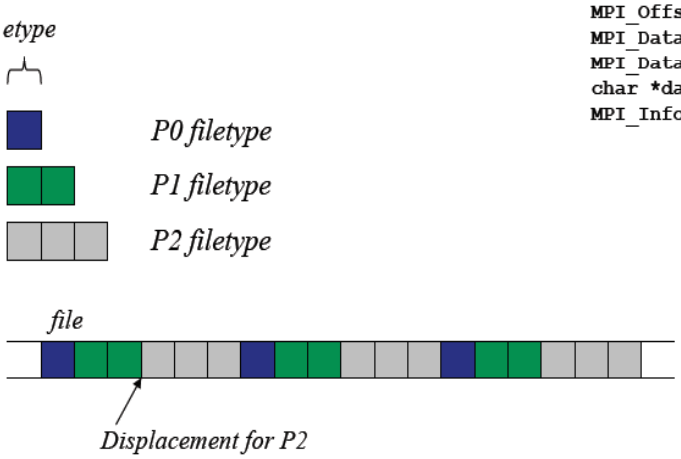
- The displacement, etype, and filetype creates a fileview by invoking MPI_File_set_view
  - A fileview allows simultaneous writing/reading of noncontiguous interleaved data by multiple processes
  - Each process has a different fileview of a single file



52

Interleaving with MPI-IO File Views

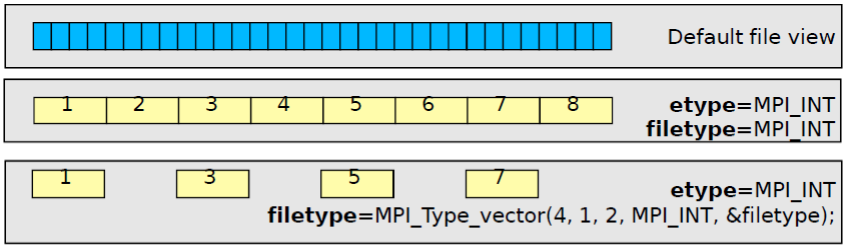
```
int MPI_File_set_view(MPI_File fh,
                      MPI_Offset disp,
                      MPI_Datatype etype,
                      MPI_Datatype filetype,
                      char *datarep,
                      MPI_Info info)
```



53

File view

- A file view defines which portion of a file is “visible” to a process
- File view defines also the type of the data in the file (byte, integer, float, …)



54

File view

- By default, file is treated as consisting of bytes, and process can access (read or write) any byte in the file
- A file view consists of three components
  - **displacement :** number of *bytes to skip from* the beginning of file
  - **etype :** type of data accessed, defines unit for offsets
  - **filetype :** portion of file visible to process
    - same as etype or MPI derived type consisting of etypes

55

File view

- MPI_File_set_view(fhandle, disp, etype, filetype, datarep, info)
  - disp
    - offset from beginning of file. Always in bytes
  - etype
    - Basic MPI type or user defined type, and basic unit of data access
    - Offsets in I/O commands in units of etype
  - filetype
    - Same type as etype or user defined type constructed of etype
    - Specifies which part of the file is visible
  - datarep
    - Data representation, sometimes useful for portability
    - "native": store in same format as in memory
  - info
    - Hints for implementation that can improve performance MPI_INFO_NULL: No hints

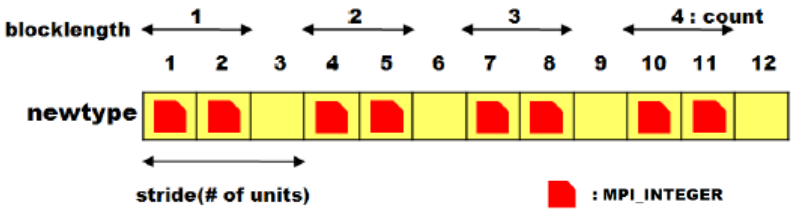
56

MPI_Type_Vector

– int MPI_Type_vector(int *count*, int *blocklength*, int *stride*, MPI_Datatype *old_type*, MPI_Datatype **newtype_p*);
  - *count* [in] number of blocks (nonnegative integer)
  - *blocklength* [in] number of elements in each block (nonnegative integer)
  - *stride* [in] number of elements between start of each block (integer)
  - *oldtype* [in] old datatype (handle)
  - *newtype_p* [out] new datatype (handle)

57

MPI_Type_Vector

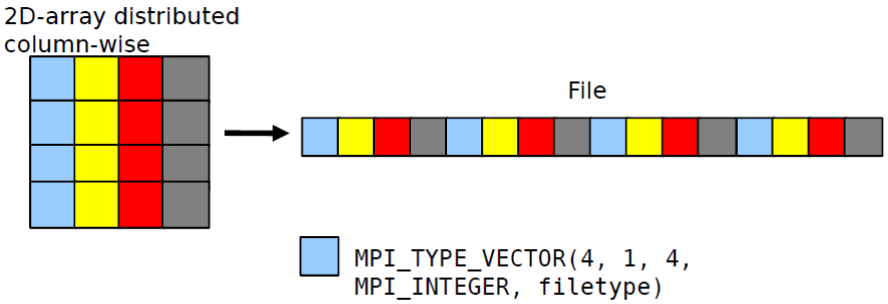



```
CALL MPI_TYPE_VECTOR(4, 2, 3, MPI_INTEGER, inewtype, ierr)
CALL MPI_TYPE_COMMIT(inewtype, ierr)
CALL MPI_BCAST(ibuf, 1, inewtype, 0, MPI_COMM_WORLD, ierr)
PRINT *, 'ibuf =' , ibuf
```

58

File view for noncontiguous data: filetype

- Use file type to implement the noncontiguous access



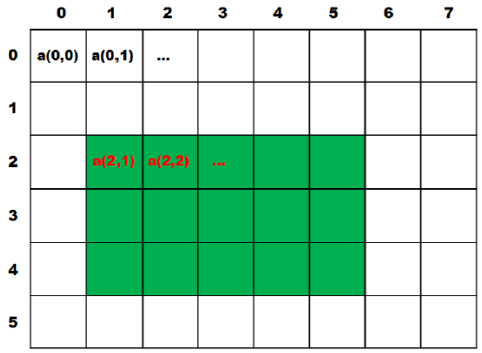
59

File view for noncontiguous data

```
...
 INTEGER :: count = 4
 INTEGER, DIMENSION(count) :: buf
 ...
 CALL MPI_TYPE_VECTOR(4, 1, 4, MPI_INTEGER, filetype, err)
 CALL MPI_TYPE_COMMIT(filetype, err)
 disp = myid * intsize
 CALL MPI_FILE_SET_VIEW(file, disp, MPI_INTEGER, filetype, “native”, &
      MPI_INFO_NULL, err)
 CALL MPI_FILE_WRITE(file, buf, count, MPI_INTEGER, status, err)
...
```

60

MPI TYPE CREATE SUBARRAY

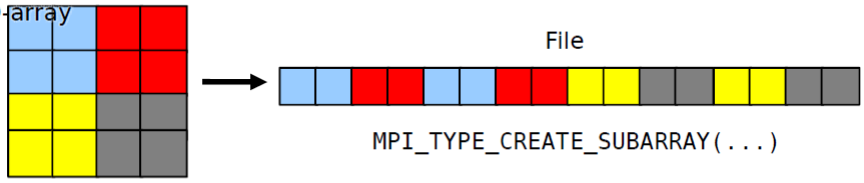


ndims = 2
array_of_sizes = {6, 8}
array_of_subsizes = {3, 5}
array_of_starts = {2, 1}
order = MPI_ORDER_FORTRAN

61

Storing 2D arrays in files

Domain decomposition for 2D-array




```
...
INTEGER :: sizes = (/4, 4/)
INTEGER :: subsizes = (/2, 2/)
INTEGER, DIMENSION(2,2) :: buf
...
MPI_CART_COORDS(MPI_COMM_WORLD, myid, 2, starts, err)
CALL MPI_TYPE_CREATE_SUBARRAY(2, sizes, subsizes, starts,
     MPI_INTEGER, MPI_ORDER_C, filetype, err)
CALL MPI_TYPE_COMMIT(filetype)
CALL MPI_FILE_SET_VIEW(file, 0, MPI_INTEGER, filetype, "native", &
     MPI_INFO_NULL, err)
CALL MPI_FILE_WRITE(file, buf, count, MPI_INTEGER, status, err)
```

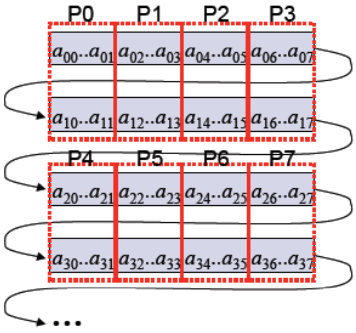
62

Example Revisited

8×8 array (block,block)-distributed over 16 processors

P0	P1	P2	P3
P4	P5	P6	P7
P8	P9	P10	P11
P12	P13	P14	P15

Row-major array layout in file




```
n_rows = 8
n_cols = 8
n_local_rows = 2
n_local_cols = 2
size = sizeof(int)
offset = (rank / 4) * n_cols * n_local_rows * size
       + (rank % 4) * n_local_cols * size
```

63

MPI I/O: Open/Close file

- All processes in a communicator open a file using
  - MPI_File_open(comm, filename, mode, info, fhandle)
  - *comm*
    - *communicator that performs parallel I/O*
  - *mode*
    - *MPI_MODE_RDONLY, MPI_MODE_WRONLY,* MPI_MODE_CREATE, MPI_MODE_RDWR,
  - *info*
    - *Hints to implementation for optimal* performance (No hints: MPI_INFO_NULL)
  - *fhandle*
    - *parallel file handle*
- File is closed using MPI_File_close(fhandle)

64

MPI I/O: Read file

- File opened with
  - MPI_MODE_RDONLY or MPI_MODE_RDWR
- Each process moves its local file pointer (individual file pointer) with
  - MPI_File_seek(fhandle, disp, whence)
  - *disp*
    - *Displacement in bytes (with default file view)*
  - *whence*
    - *MPI_SEEK_SET, MPI_SEEK_CUR, MPI_SEEK_END*

65

MPI I/O: Read file

- Read file at individual file pointer
  - MPI_File_read(fhandle, buf, count,datatype, status)
  - *buf*
    - *Buffer in memory where to read the data*
  - *count*
    - *number of elements to read*
  - *datatype*
    - *datatype of elements to read*
  - *status*
    - *similar to status in MPI_Recv, amount of data* read can be determined by **MPI_Get_count on** *status*
- Updates position of file pointer after reading

66

MPI I/O: Read file

- The location to read can be determined also within the read statement (explicit offset)
  - MPI_File_read_at(fhandle, disp, buf, count, datatype, status)
  - *disp*
    - *displacement in bytes (with the default file view)* from the beginning of file
  - Thread-safe
  - The file pointer is neither used or incremented

67

MPI I/O: Write file

- Similar to reading, file opened with
  - MPI_MODE_WRONLY or MPI_MODE_CREATE
- Write file at individual file pointer
  - MPI_File_write(fhandle, buf, count,datatype, status)
  - Updates position of file pointer after writing
  - Not thread safe

68

MPI I/O: Write file

- Determine location within the write statement (explicit offset)
  - MPI_File_write_at(fhandle, disp, buf,count, datatype, status)
  - Thread-safe
  - The file pointer is neither used or incremented

69

병렬 I/O 루틴 : MPI_FILE_OPEN

C	`int MPI_File_open(MPI_Comm comm, char *filename, int amode, MPI_Info info, MPI_File *fh)`
Fortran	`MPI_FILE_OPEN(comm, filename, amode, info, fh, ierr)`

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

- 파일 접근 모드 : OR(|:C), IOR(+:Fortran)로 연결 가능
  - info 객체 : 시스템 환경에 따른 프로그램 구현의 변화에 대한 정보 제공, 통상적으로 MPI_INFO_NULL 사용

MPI_MODE_APPEND	파일 포인터의 시작위치를 파일 마지막에 설정
MPI_MODE_CREATE	파일생성, 만약 파일이 있으면 덮어씀
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

병렬 I/O 루틴 : MPI_FILE_WRITE

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

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	`int MPI_File_close(MPI_File *fh)`
Fortran	`MPI_FILE_CLOSE(fh, ierr)`

INTEGER fh : 파일 핸들 (핸들) (INOUT)

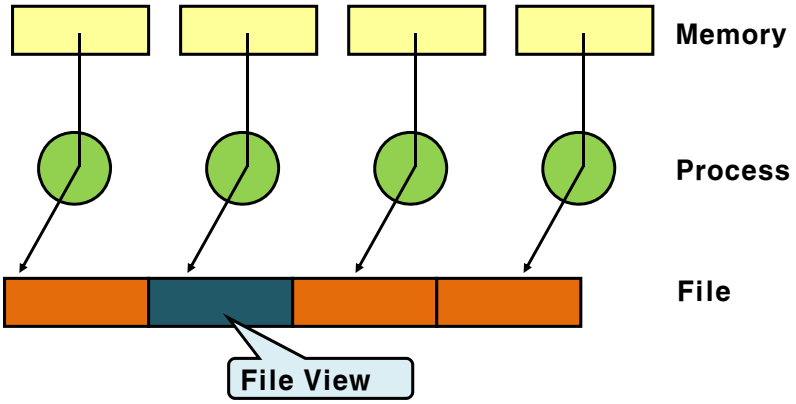
Exercises 2 : MPI I/O

1. Write numbers 0-399 to a file in parallel using MPI I/O. Verify the write by reading the file with different number of processes than in writing.
2. Try to do multiple writes in the same process and investigate how the file pointer is updated when using MPI_File_write.
   – Hint: MPI provides a function MPI_File_get_position

74

병렬 프로그램과 병렬 I/O

- 병렬 프로세스가 하나의 공유 파일 작성



병렬 프로그램과 병렬 I/O

- 공유된 파일에 각 프로세스가 접근하는 부분
  - **MPI_FILE_SET_VIEW로 설정**
- 각 프로세스의 파일 뷰 시작 위치 계산
  - `disp = myrank*BUFSIZE*sizeof(int);`

4바이트 정수

병렬 프로그램과 병렬 I/O (1)

```
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 (2)

```
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 (1)

```
/*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 (2)

```
  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();
}
```

MPI_FILE_SET_VIEW (1)

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) (유도데이터 타입의 기본 데이터 타입)
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)

MPI_FILE_SET_VIEW (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

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

MPI_FILE_GET_SIZE

C	`int MPI_File_get_size(MPI_File fh, MPI_Offset *size)`
Fortran	`MPI_FILE_GET_SIZE(fh, size, ierr)`

INTEGER fh : 파일 핸들 (핸들) (IN)
INTEGER (kind=MPI_OFFSET_KIND) size : 파일의 크기 (OUT)

- 파일의 크기를 바이트 단위로 저장

MPI_FILE_READ

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

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

Exercises 3 : Using File views

1. Write numbers 0-99 to a file.
   - Distribute the numbers to the processes in strides, (e.g. 0, 4, 8, ... for rank 0 with 4 processes) but have the numbers in the correct order in the file. Verify the write by reading.
2. Distribute a 2 D array to processes in blocks, and write the array to a file in parallel

86

Collective operations

- I/O can be performed collectively by all processes in a communicator
  - **MPI_File_read_all**
  - **MPI_File_write_all**
  - **MPI_File_read_at_all**
  - **MPI_File_write_at_all**
- Same parameters as in independent I/O functions (MPI_File_read, MPI_File_write,MPI_File_read_at, MPI_File_write_at)

87

Collective operations

- All processes in communicator that opened file must call function
- Performance potentially better than for individual functions
  - Even if each processor reads a non-contiguous segment, in total the read is contiguous

88

Non-blocking MPI I/O - Independent I/O

- Non-blocking independent I/O is similar to non-blocking send/recv routines
  - MPI_File_iread
  - MPI_File_iwrite
  - MPI_File_iread_at
  - MPI_File_iwrite_at
- Wait for completion using MPI_Test,MPI_Wait, etc.
- Can be used to overlap I/O with computation

89

Non-blocking MPI I/O - Collective I/O

- Split-collective I/O supported
  - Restrictive form of non-blocking I/O
  - Only one collective operation per file handle is supported at a time
  - Can be used to overlap I/O with computation

90

Non-blocking MPI I/O - Collective I/O

- Split-collectives tarted with _begin functions
  - MPI_File_write_all_begin
  - MPI_File_read_all_begin
  - MPI_File_write_at_all_begin
  - MPI_File_read_at_all_begin
- Closed with _end functions
  - MPI_File_write_all_end
  - MPI_File_read_all_end
  - MPI_File_write_at_all_end
  - MPI_File_read_at_all_end

91

Giving hints to MPI I/O

- Hints may enable the implementation to optimize performance
- MPI 2 standard defines several hints via MPI_Info object
  - MPI_INFO_NULL : no info
  - Functions **MPI_Info_create, MPI_Info_set** allow one to create and set hints

92

Giving hints to MPI I/O

- Some implementations allow setting of hints via environment variables
  - e.g. MPICH_MPIIO_HINTS
  - Example: for file “test.dat”, in collective I/O aggregate data to 32 nodes
    - Export MPICH_MPIIO_HINTS=”test.dat:cb_nodes=32”
- Effect of hints on performance is implementation and application dependent

93

Parallel I/O - summary

- POSIX
  - single reader/writer, all read/write, subset read/write
  - user is responsible for communication
- MPI I/O
  - MPI library is responsible for communication
  - file views enable non-contiguous access patterns
  - collective I/O can enable the actual disk access to remain contiguous

94