

# MPI를 이용한 병렬 프로그래밍



# 차 례

|  |          |
|--|----------|
| 차 례 .....                                    | i        |
| 그림 차례 .....                                  | iv       |
| 표 차례 .....                                   | x        |
| <b>제 1 장 MPI 소개.....</b>                     | <b>1</b> |
| 1.1 MPI 의 개요.....                            | 1        |
| 1.1.1 메시지 패싱 (message passing) 프로그래밍 모델..... | 1        |
| 1.1.2 MPI 란 무엇인가?.....                       | 3        |
| 1.1.3 MPI 의 목표.....                          | 4        |
| 1.1.4 MPI 의 기본 개념들.....                      | 5        |
| <b>제 2 장 MPI 를 이용한 병렬 프로그래밍 기초.....</b>      | <b>8</b> |
| 2.1 MPI 프로그램의 구조 .....                       | 8        |
| 2.1.1 MPI 헤더파일.....                          | 8        |
| 2.1.2 MPI 핸들 .....                           | 9        |
| 2.1.3 MPI 루틴의 호출과 리턴값.....                   | 9        |
| 2.1.4 MPI 초기화 .....                          | 11       |
| 2.1.5 커뮤니케이터.....                            | 12       |
| 2.1.6 MPI 종료.....                            | 13       |
| 2.1.7 MPI 프로그램 : Bones .....                 | 14       |
| 2.1.8 메시지.....                               | 15       |
| 2.2 점대점 통신 서브루틴.....                         | 17       |
| 2.2.1 통신모드.....                              | 19       |
| 2.2.2 블록킹 송신과 수신.....                        | 24       |

|                                    |                                      |            |
|------------------------------------|--------------------------------------|------------|
| 2.2.3                              | 논블록킹 송신과 수신                          | 34         |
| 2.2.4                              | 점대점 통신의 사용                           | 41         |
| 2.3                                | 집합 통신 서브루틴                           | 48         |
| 2.3.1                              | 방송 (Broadcast) : MPI_BCAST           | 49         |
| 2.3.2                              | 취합 (Gather)                          | 53         |
| 2.3.3                              | 환산 (Reduce)                          | 65         |
| 2.3.4                              | 확산 (scatter)                         | 72         |
| 2.3.5                              | 기타                                   | 76         |
| 2.4                                | 유도 데이터 타입 (Derived Data Type)        | 80         |
| 2.4.1                              | 기본적인 사용법                             | 81         |
| 2.4.2                              | 데이터 타입 구성                            | 82         |
| 2.4.3                              | 데이터 타입 등록                            | 98         |
| 2.4.4                              | 부분 배열의 전송 : MPI-2                    | 98         |
| 2.5                                | 커뮤니케이터                               | 103        |
| 2.5.1                              | MPI_COMM_SPLIT                       | 103        |
| 2.6                                | 가상 토폴로지 (Virtual Topology)           | 107        |
| 2.6.1                              | 직교좌표 가상 토폴로지 만들기 : MPI_CART_CREATE   | 108        |
| 2.6.2                              | 대응 함수                                | 112        |
| 2.6.3                              | 토폴로지 분해                              | 119        |
| <b>제 3 장 MPI 를 이용한 병렬 프로그래밍 실제</b> |                                      | <b>122</b> |
| 3.1                                | 병렬 프로그램에서의 입 / 출력                    | 122        |
| 3.1.1                              | 입력                                   | 122        |
| 3.1.2                              | 출력                                   | 125        |
| 3.2                                | DO(for) 루프의 병렬화                      | 127        |
| 3.2.1                              | 블록 분할 (Block Distribution)           | 127        |
| 3.2.2                              | 순환 분할 (Cyclic Distribution)          | 133        |
| 3.2.3                              | 블록 순환 분할 (Block-Cyclic Distribution) | 134        |

|              |  |            |
|--------------|--|------------|
| 3.2.4        | 배열 수축 (Shrinking Arrays).....                    | 135        |
| 3.2.5        | 내포된 (Nested) 루프의 병렬화 .....                       | 141        |
| 3.3          | 메시지 패싱과 병렬화.....                                 | 149        |
| 3.3.1        | 외부 데이터에 대한 참조 .....                              | 149        |
| 3.3.2        | 1 차원 유한 차분법 (1-D Finite Difference Method) ..... | 150        |
| 3.3.3        | 대량 데이터 전송.....                                   | 156        |
| 3.3.4        | 중첩 (Superposition).....                          | 177        |
| 3.3.5        | 파이프라인 방법.....                                    | 179        |
| 3.3.6        | 비틀림 분해 (The Twisted Decomposition).....          | 188        |
| 3.3.7        | 프리픽스 합.....                                      | 198        |
| <b>제 4 장</b> | <b>MPI 프로그램 예제 .....</b>                         | <b>203</b> |
| 4.1          | 2 차원 유한 차분법.....                                 | 203        |
| 4.1.1        | 열 방향 블록 분할.....                                  | 205        |
| 4.1.2        | 행 방향 블록 분할.....                                  | 209        |
| 4.1.3        | 양 방향 블록 분할.....                                  | 215        |
| 4.2          | 몬테카를로 방법 .....                                   | 226        |
| 4.3          | 분자 동역학 (Molecular Dynamics).....                 | 232        |
| 4.3.1        | MPMD Models .....                                | 239        |
| <b>제 5 장</b> | <b>부 록 .....</b>                                 | <b>242</b> |
| 5.1          | MPI-2.....                                       | 242        |
| 5.1.1        | 병렬 MPI I/O.....                                  | 242        |
| 5.1.2        | 일방통신 (One-Sided Communication).....              | 262        |
| 5.2          | 참고 자료 .....                                      | 278        |
|              | <b>찾아보기 .....</b>                                | <b>279</b> |

# 그림 차례

|         |                                  |    |
|---------|----------------------------------|----|
| 그림 1.1  | 분산 메모리 시스템                       | 2  |
| 그림 1.2  | 메시지 패싱 프로그래밍 모델                  | 3  |
| 그림 1.3  | MPI 의 메시지 패싱                     | 7  |
| 그림 2.1  | MPI 기본구조                         | 8  |
| 그림 2.2  | MPI 루틴의 호출과 리턴값                  | 11 |
| 그림 2.3  | 점대점 통신                           | 18 |
| 그림 2.4  | 동기 송신                            | 19 |
| 그림 2.5  | 준비 송신                            | 20 |
| 그림 2.6  | 버퍼 송신                            | 21 |
| 그림 2.7  | 블록킹 표준 통신 ( 메시지 크기 $\leq$ 임계값 )  | 22 |
| 그림 2.8  | 블록킹 표준 통신 ( 메시지 크기 $>$ 임계값 )     | 23 |
| 그림 2.9  | 논블록킹 표준 통신 ( 메시지 크기 $\leq$ 임계값 ) | 35 |
| 그림 2.10 | 논블록킹 표준 통신 ( 메시지 크기 $>$ 임계값 )    | 36 |
| 그림 2.11 | 단방향 통신                           | 42 |
| 그림 2.12 | 양방향 통신                           | 44 |
| 그림 2.13 | 집합 통신                            | 49 |
| 그림 2.14 | 방송                               | 50 |
| 그림 2.15 | MPI_GATHER 를 이용한 취합              | 54 |
| 그림 2.16 | MPI_GATHERV 를 이용한 취합             | 58 |
| 그림 2.17 | MPI_ALLGATHER 를 이용한 취합           | 62 |
| 그림 2.18 | MPI_ALLGATHERV 를 이용한 취합          | 64 |
| 그림 2.19 | MPI_REDUCE 를 이용한 환산              | 66 |
| 그림 2.20 | 배열에 적용된 환산 연산 MPI_REDUCE         | 70 |
| 그림 2.21 | MPI_ALLREDUCE                    | 72 |
| 그림 2.22 | MPI_SCATTER                      | 73 |
| 그림 2.23 | MPI_SCATTERV                     | 76 |

|         |                                   |     |
|---------|-----------------------------------|-----|
| 그림 2.24 | MPI_ALLTOALL                      | 77  |
| 그림 2.25 | MPI_ALLTOALLV                     | 78  |
| 그림 2.26 | MPI_REDUCE_SCATTER                | 79  |
| 그림 2.27 | MPI_SCAN                          | 80  |
| 그림 2.28 | 유도 데이터 타입                         | 81  |
| 그림 2.29 | MPI_TYPE_CONTIGUOUS               | 83  |
| 그림 2.30 | 루틴 MPI_TYPE_VECTOR 의 인수           | 86  |
| 그림 2.31 | MPI_TYPE_VECTOR                   | 86  |
| 그림 2.32 | MPI_TYPE_HVECTOR                  | 89  |
| 그림 2.33 | 루틴 MPI_TYPE_STRUCT 의 인수           | 90  |
| 그림 2.34 | MPI_TYPE_STRUCT                   | 91  |
| 그림 2.35 | MPI_TYPE_CREATE_SUBARRAY          | 100 |
| 그림 2.36 | MPI_COMM_SPLIT                    | 104 |
| 그림 2.37 | 차원 프로세스 그리드                       | 108 |
| 그림 2.38 | MPI_CART_CREATE                   | 110 |
| 그림 2.39 | MPI_CART_SHIFT                    | 117 |
| 그림 2.40 | MPI_CART_SUB                      | 120 |
|         |                                   |     |
| 그림 3.1  | 공유 파일 시스템에 입력파일을 둔 경우             | 123 |
| 그림 3.2  | 각 노드에 입력 파일을 복사해둔 경우              | 124 |
| 그림 3.3  | 한 프로세스가 입력 파일을 읽어 각 노드에 전달하는 경우 1 | 124 |
| 그림 3.4  | 한 프로세스가 입력 파일을 읽어 각 노드에 전달하는 경우 2 | 125 |
| 그림 3.5  | 한 프로세스가 데이터를 모아서 로컬 파일 시스템에 저장    | 126 |
| 그림 3.6  | 공유 파일 시스템에 순차적으로 저장하는 경우          | 127 |
| 그림 3.7  | 블록 분할                             | 128 |
| 그림 3.8  | 순환 분할                             | 134 |
| 그림 3.9  | 블록 순환 분할                          | 135 |
| 그림 3.10 | 순차 실행 : 데이터 원본                    | 136 |
| 그림 3.11 | 병렬 실행 : 배열 수축하지 않은 데이터            | 137 |
| 그림 3.12 | 병렬 실행 : 배열 수축에 의해 증가된 데이터         | 138 |
| 그림 3.13 | 2 차원 배열의 저장                       | 142 |
| 그림 3.14 | 내포된 루프의 병렬화                       | 144 |
| 그림 3.15 | 루프 C 의 의존성                        | 145 |

|         |   |     |
|---------|---|-----|
| 그림 3.16 | 열 방향으로 블록 분할된 루프 C 의 병렬화 의존성                          | 145 |
| 그림 3.17 | 루프 D 의 의존성  | 146 |
| 그림 3.18 | 열 방향 (1) 과 행 방향 (2) 으로 블록 분할된 루프 D 의 병렬화 의존성          | 147 |
| 그림 3.19 | 양 방향 블록 분할  | 148 |
| 그림 3.20 | 여러 가지 블록 분할과 통신량                                      | 148 |
| 그림 3.21 | 1 차원 FDM 의 데이터 의존성                                    | 152 |
| 그림 3.22 | 병렬 1 차원 FDM 의 데이터 이동                                  | 156 |
| 그림 3.23 | 한 프로세스로 데이터 취합 ( 연속 데이터 , 송 / 수신 버퍼 중복 없음 ) : Fortran | 157 |
| 그림 3.24 | 한 프로세스로 데이터 취합 ( 연속 데이터 , 송 / 수신 버퍼 중복 ) : Fortran    | 160 |
| 그림 3.25 | 한 프로세스로 데이터 취합 ( 불연속 데이터 , 송 / 수신 버퍼 중복 ) : Fortran   | 164 |
| 그림 3.26 | 데이터 동기화 ( 연속 데이터 , 송 / 수신 버퍼 중복 없음 ) : Fortran        | 168 |
| 그림 3.27 | 데이터 동기화 ( 연속 데이터 , 송 / 수신 버퍼 중복 ) : Fortran           | 170 |
| 그림 3.28 | 블록 분할의 전환   | 172 |
| 그림 3.29 | 블록 분할 전환을 위해 정의된 유도 데이터 타입                            | 173 |
| 그림 3.30 | 중첩  | 179 |
| 그림 3.31 | 프로그램 main 의 의존성                                       | 181 |
| 그림 3.32 | 프로그램 main2 의 의존성                                      | 183 |
| 그림 3.33 | 파이프 라인 방법에 의한 병렬화 : Fortran                           | 184 |
| 그림 3.34 | 파이프 라인 방법의 데이터 흐름                                     | 187 |
| 그림 3.35 | 파이프 라인 방법과 블록 크기                                      | 188 |
| 그림 3.36 | 비틀림 분해  | 190 |
| 그림 3.37 | 비틀림 분해의 데이터 흐름  | 197 |
| 그림 3.38 | 프리픽스 합  | 199 |
| 그림 4.1  | 2 차원 FDM : 열 방향 블록 분할                                 | 205 |
| 그림 4.2  | 2 차원 FDM : 행 방향 블록 분할                                 | 210 |
| 그림 4.3  | 양 방향 블록 분할과 프로세스 그리드                                  | 216 |
| 그림 4.4  | 2 차원 FDM : 양 방향 블록 분할                                 | 217 |
| 그림 4.5  | 2 차원 임의 행로  | 229 |

|         |                             |     |
|---------|-----------------------------|-----|
| 그림 4.6  | 두 입자의 상호작용 .....            | 233 |
| 그림 4.7  | 입자에 작용하는 힘 .....            | 234 |
| 그림 4.8  | 바깥쪽 루프 (i) 에 대한 순환 분할 ..... | 237 |
| 그림 4.9  | 안쪽 루프 (j) 에 대한 순환 분할 .....  | 239 |
| 그림 4.10 | MPMD 모델 .....               | 241 |
| 그림 5.1  | 병렬 프로그램과 순차 I/O .....       | 243 |
| 그림 5.2  | 다중 파일에 저장하는 순차 I/O .....    | 246 |
| 그림 5.3  | 단일 파일에 저장하는 병렬 I/O .....    | 257 |
| 그림 5.4  | 원격 메모리 접근 윈도우 .....         | 263 |



## 표 차례

|       |  |    |
|-------|--|----|
| 표 2.1 | MPI 데이터 타입 : Fortran .....                     | 16 |
| 표 2.2 | MPI 데이터 타입 : C .....                           | 16 |
| 표 2.3 | 여러 가지 통신 모드 .....                              | 19 |
| 표 2.4 | status 에 저장되는 정보 .....                         | 25 |
| 표 2.5 | MPI 집합통신 루틴 .....                              | 48 |
| 표 2.6 | MPI 환산 연산자와 데이터 타입 : Fortran .....             | 69 |
| 표 2.7 | MPI 환산 연산자와 데이터 타입 : C .....                   | 69 |
| 표 2.8 | 환산 연산자 MPI_MAXLOC, MPI_MINLOC 에 사용된 데이터 타입 : C | 70 |

# 제 1 장 MPI 소개

MPI 를 소개하고 MPI 를 이해하는데 필요한 기본 개념들에 대해 알아본다 .

## 1.1 MPI 의 개요

### 1.1.1 메시지 패싱 (message passing) 프로그래밍 모델

MPI(Message Passing Interface) 는 병렬 프로그래밍 모델 중 “메시지 패싱” 모델의 표준으로 알려져 있다 . MPI 에 대해 알아보기에 앞서 메시지 패싱 프로그래밍 모델에 대해 간단히 알아본다 .

하나의 작업을 다수의 프로세스들에게 나누어 실행시키는 병렬 계산에서는 필연적으로 프로세스들 사이의 통신이 필요하다 . 메시지 패싱 모델은 각자의 메모리를 지역적으로 따로 가지는 프로세스들로 구성된 분산 시스템 환경에서 ( 그림 1.1 참조 ), 프로세스들 사이의 통신을 오직 메시지들의 송신 (sending) 과 수신 (receiving) 으로만 구현하는 프로그래밍 모델을 말한다 . 그래서 메시지 패싱 모델은 프로세스들이 메모리 공간을 공유하지 않으며 한 프로세스가 다른 프로세스의 메모리에 직접 접근하는 것을 허용치 않는다 .

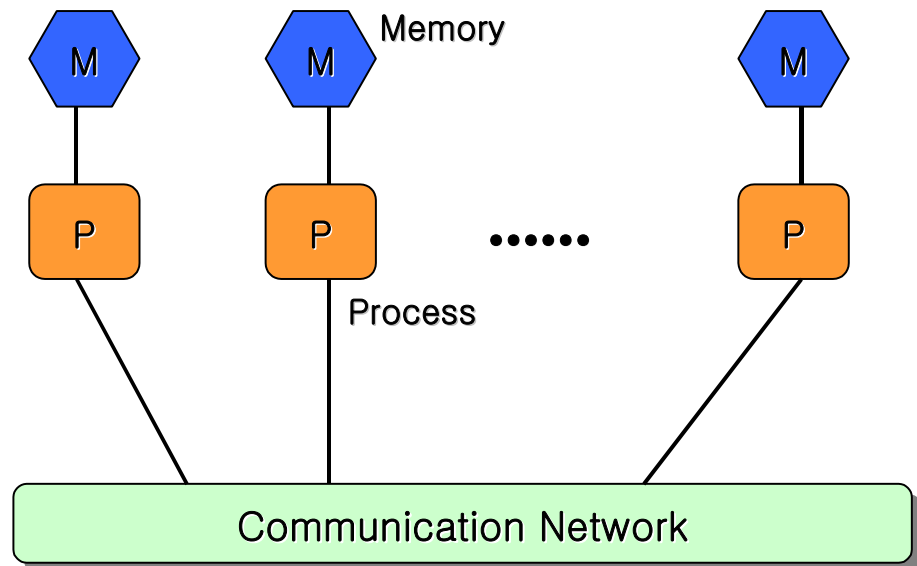


그림 1.1 분산 메모리 시스템

메시지 패싱 프로그래밍 모델은 최근 들어 많이 이용되고있는데, 주된 이유 중의 한가지는 메시지 패싱 모델은 여러 가지 다양한 플랫폼에서 구현될 수 있기 때문이다. 메시지 패싱 스타일로 작성된 병렬 프로그램들은 분산, 공유메모리 다중 프로세서 환경, 워크스테이션들의 네트워크, 그리고 단일 프로세서 환경에서도 실행할 수 있다. 지시어 기반의 OpenMP 를 이용한 병렬 프로그래밍이 손쉬울지라도 메시지 패싱 모델이 병렬 컴퓨팅에서 널리 이용되고있는 이유는 특별히 쉽기 때문이 아니라 보다 일반적이기 때문이다.

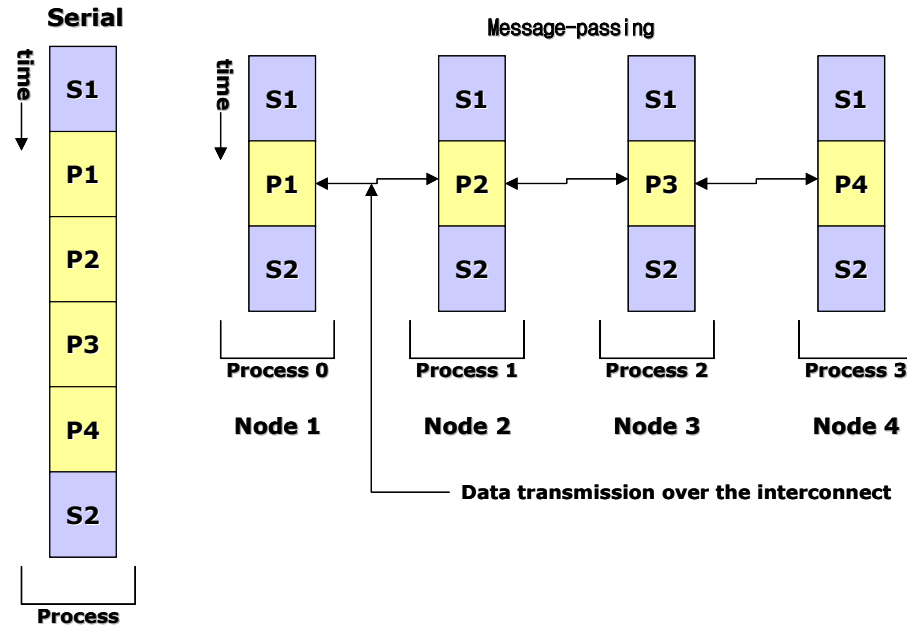


그림 1.2 메시지 패싱 프로그래밍 모델

### 1.1.2 MPI란 무엇인가 ?

MPI는 “Message Passing Interface”의 약어로서 프로세스들 사이의 통신을 위해 코드에서 호출해 사용하는 서브루틴(Fortran) 또는 함수(C)들의 라이브러리이다.

MPI는 Fortran 또는 C로 작성된 메시지 패싱 프로그램들에게 순차 프로그램들처럼 다양한 아키텍처들에 대한 풍부한 소스코드 이식성 (source-code portability)을 제공하고자 하는 표준화 작업의 결과이다. 지난 1994년 봄에 40개의 서로 다른 기구들을 대표하는 약 60명의 메시지 패싱 시스템 전문가들로 구성된 MPIF(MPI Forum)는 MPI-1 표준을 내놓았고, 1997년에 기존의 MPI-1에 병렬 I/O, C++와 Fortran90 지원, 동적 프로세스 관리 등의 도구를 추가한 MPI-2를 발표하였다.

참고로 MPI-2에서 추가된 내용은 다음과 같다.

- Dynamic process management
- One-sided operations
- Parallel I/O

- C++ and FORTRAN 90 bindings
- External interfaces
- Extended collective communications
- Real-time extensions
- Other areas

현재 MPI 를 구현하는 제품들은 다음과 같은 여러 가지가 있다 .

- MPI/Pro: MPI Software Technology implementation
- IBM MPI: IBM product implementation for the SP and RS/6000 workstation clusters
- MPICH: Argonne National Lab and Mississippi State University implementation
- UNIFY: Mississippi State University implementation
- CHIMP: Edinburgh Parallel Computing Centre implementation
- LAM: Ohio Supercomputer Center implementation

### 1.1.3 MPI 의 목표

MPI 의 주요 목표는 다음과 같다 .

- 어떤 플랫폼상에서도 컴파일과 실행이 가능한 소스코드 이식성
- 다양한 아키텍처에 걸쳐 효율적인 수행을 가능하게 하는 것
- 메시지 패싱 프로그래밍을 위한 풍부한 기능 제공

현재 MPI 는 집합 연산, 사용자 정의 데이터 타입과 토폴로지, 다양한 방식의 통신 등의 다양한 기능들과 이기종 병렬 아키텍처아키텍처 (heterogeneous parallel architecture) 에 대한 지원 등을 제공하며, MPI-2 에서는 코드가 실행되는 동안 프

로세스의 수를 변화시키는 동적 프로세스 관리, 원격 메모리 접근, 병렬 I/O 등을 지원하고 있다.

#### 1.1.4 MPI의 기본 개념들

**프로세스와 프로세서** : 스레드 기준으로 작업을 할당하는 OpenMP와 달리 MPI는 프로세스를 기준으로 작업을 할당한다. 일반적으로 한 개의 프로세서에서 여러 개의 프로세스가 실행될 수 있으나, 프로그램의 최적화된 성능을 위해 하나의 프로세서에서 하나의 프로세스가 실행되는 것이 바람직하다. 본 교재에서는 하나의 프로세서에 하나의 프로세스만을 가정할 것이며 따라서 프로세스와 프로세서를 특별한 언급이 없는 한 구분 없이 사용할 것이다.

**메시지** : 메시지는 데이터와 데이터의 송신지와 수신지 주소를 나타내는 봉투(envelope)로 구성된다. 메시지 전송은 한 서브프로그램의 변수에서 다른 서브프로그램의 변수로 데이터가 이동하는 것이며, 메시지 패싱 시스템은 이 데이터의 값 자체에는 관심이 없고 오로지 그것의 이동에만 관심이 있다. 일반적으로 메시지 전송 시스템이 메시지 전송을 확인하기 위해 다음과 같은 정보들이 필요하다.

- 어떤 프로세스가 메시지를 보내는가
- 메시지를 보내는 프로세스의 어디에 데이터가 있는가
- 보내는 데이터는 어떤 종류인가
- 데이터의 양은 얼마나 되는가
- 어떤 프로세스(들)가 메시지를 받는가
- 메시지를 받는 프로세스의 어느 위치에 데이터가 들어가는가
- 얼마나 많은 양의 데이터를 받을 준비를 해야 하는가

메시지를 받는 프로세스는 전달되는 데이터가 모두 도착해야 그 데이터를 사용할 수 있다. 보내는 프로세스도 보낸 메시지가 모두 도착했는지 알아야 할 경우가 있을 것이다. 따라서, 메시지 패싱 시스템은 데이터 전달뿐 아니라 통신에 대한 진행

상황에 대해서도 정보를 제공해야 한다 . 메시지 전송은 메시지에 동기화에 대한 정보를 제공한다 .

**꼬리표 (Tag):** 프로세스간에 서로 주고받는 메시지를 구분하고 확인할 목적으로 각 메시지에는 꼬리표를 붙인다 . 가령 , 1 번 프로세스 ( 송신 ) 에서 2 번 프로세스 ( 수신 ) 로 10 개의 메시지를 보낸다면 꼬리표를 이용해 각 메시지를 구분할 수 있으며 , 송신 프로세스측에서는 꼬리표를 이용해 받는 메시지를 순서대로 처리할 수 있다 . MPI 는 와일드카드 꼬리표 사용도 지원하고 있다 .

**커뮤니케이터 (communicator):** 커뮤니케이터는 서로 통신할 수 있는 프로세스들의 집합을 나타낸다 . MPI 에서의 통신은 같은 커뮤니케이터를 공유하는 즉 , 같은 프로세스 그룹에 속한 프로세스들끼리만 통신이 가능하다 .

커뮤니케이터 이름은 모든 점대점 통신과 집합통신에 인수로 필요하며 , 송신과 그에 대응하는 수신은 커뮤니케이터가 일치해야 한다 . 한 프로그램 내에서 여러 개의 커뮤니케이터가 있을 수 있고 , 한 프로세스는 서로 다른 여러 커뮤니케이터에 속할 수 있다 .

MPI\_COMM\_WORLD 는 MPI 에서 기본적으로 제공하는 커뮤니케이터로 MPI 헤더파일에 정의되어 있으며 , 병렬 작업에 참여하는 모든 프로세스들로 구성된다

**프로세스 랭크 (rank):** 커뮤니케이터 내에서 프로세스들은 0 부터 시작되는 연속적인 정수들을 할당 받게 되는데 , 이 정수들은 커뮤니케이터 내에서 프로세스를 구분 짓는 고유번호로서 프로세스 식별자 역할을 한다 .

**점대점 (Point to Point) 통신 :** 점대점 통신은 가장 간단한 메시지 전달 방식으로 MPI 통신의 기본이 된다 . 하나의 송신 프로세스로부터 다른 하나의 수신 프로세스 로 메시지가 전송되며 , 오직 두개의 프로세스만이 그 메시지에 대해 알 필요가 있을 때 사용된다 . 메시지 송신특성 , 그리고 송신과 프로그램 실행이 어떻게 상호작용 하는가에 따라 몇 가지 통신모드가 존재한다 .

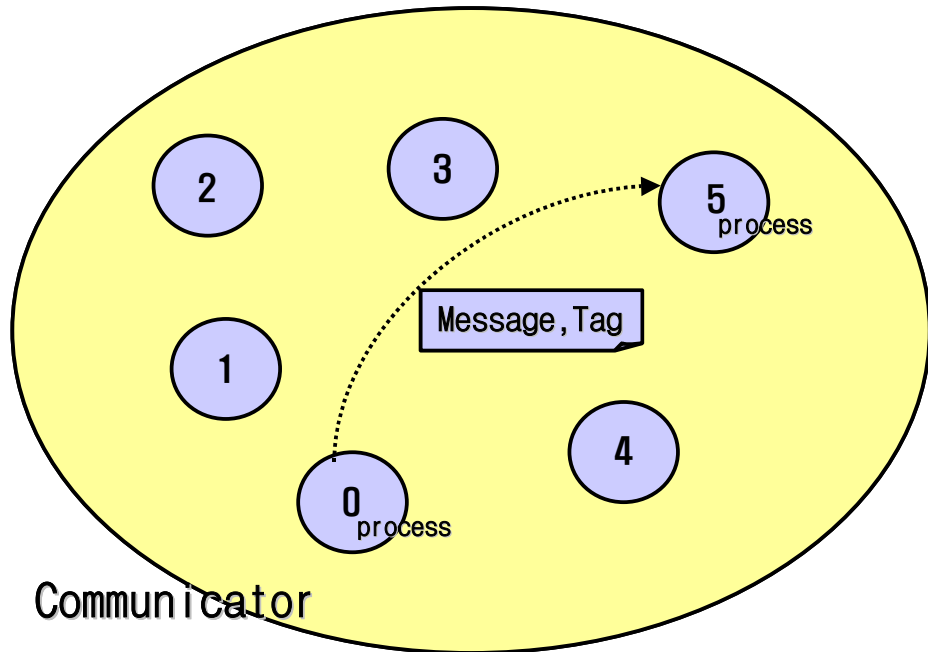


그림 1.3 MPI 의 메시지 패싱

**집합 통신 (Collective Communication):** 메시지 패싱 시스템들은 한 쌍의 프로세스만이 참여하는 점대점 통신뿐 아니라 한 무리의 프로세스들이 동시에 통신에 참여하는 다양한 집합통신을 제공한다.

일대다 (one-to-many) 또는 다대일 (many-to-one) 형식과 같은 여러 가지 통신방법이 있는데, 집합통신은 점대점 통신과 비교하여 다음과 같은 장점이 있다.

집합통신 루틴을 호출하는 한 줄의 코드가 여러 번의 점대점 호출을 대신하므로 오류의 가능성이 줄어든다.

코드를 좀 더 쉽게 읽을 수 있어 디버깅 작업 등이 간단해 진다.

최적화된 집합통신 루틴의 호출은 같은 동작을 하도록 하는 점대점 루틴들의 호출보다 일반적으로 빠르다.



## 제 2 장 MPI 를 이용한 병렬 프로그래밍 기초

이번 장에서는 MPI 를 이용하여 메시지 패싱 프로그램을 작성하는데 필요한 실질적인 방법들에 대해 알아본다. MPI 루틴들은 MPI-1, MPI-2 를 합하면 200 여 개가 넘는데, 병렬프로그램 작성을 위해 이러한 서브루틴들을 모두 알고 있을 필요는 없다. 실질적으로 필요한 루틴들은 30 개 안팎이며 본 교재에서는 자주 이용되는 루틴들을 중심으로 MPI 병렬 프로그램 작성 방법들을 소개 할 것이다.

### 2.1 MPI 프로그램의 구조

모든 MPI 프로그램들은 다음과 같은 일반적인 구조를 가진다.



그림 2.1 MPI 기본구조

#### 2.1.1 MPI 헤더파일

모든 MPI 프로그램은 다음과 같이 헤더 파일을 삽입시켜 주어야 한다.

**C : #include <mpi.h> 또는 #include "mpi.h"**

**Fortran : INCLUDE 'mpif.h'**

MPI 헤더파일은 MPI 서브루틴과 함수들의 프로토타입 (prototype) 을 포함하고 있으며, 매크로들의 정의, MPI\_COMM\_WORLD 나 MPI\_INTEGER 과 같은 MPI 와 관련된 인수들과, 데이터 타입들을 정의하고 있다.

참고로 KISTI 의 IBM 시스템에는 /usr/lpp/ppe.poe/include/ 안에 헤더파일이 저장되어 있다.

### 2.1.2 MPI 핸들

MPI 핸들은 MPI 고유의 내부 자료구조 참조에 이용되는 포인터 변수를 의미한다. C 에서의 MPI 핸들은 typedef 으로 정의된 특별한 데이터 타입 (MPI\_Comm, MPI\_Datatype, ...) 을 가지며, Fortran 의 핸들은 정수타입으로 선언된다.

MPI\_SUCCESS, MPI\_COMM\_WORLD 등은 모두 MPI 핸들이다.

### 2.1.3 MPI 루틴의 호출과 리턴값

MPI 루틴들은 C 에서는 함수로 Fortran 에서는 서브루틴으로 구현된다. Fortran 은 대소문자 구분없이 MPI\_XXXX(parameter) 의 형식으로 루틴을 호출하며 C 에서는 MPI\_Xxxx(parameter) 의 형식으로 MPI 와 “\_” 다음에 오는 첫 문자를 반드시 대문자로 하고 다음에 문자들은 소문자로 적어야한다.

MPI 루틴은 사용자가 루틴이 성공적으로 수행되었는가를 검사해 볼 수 있도록 에러코드를 리턴한다. Fortran 에서 MPI 서브루틴들은 다음과 같이 오류 상황을 나타내는 정수 인수를 항상 인수 리스트의 맨 마지막에 하나 추가해 가지고 있다.

**INTEGER IERR**

...

**CALL MPI\_INIT(IERR)**

...

C 에서 MPI 함수들은 함수 호출의 종료 상황을 나타내는 정수를 리턴한다 .

```
int err;
```

...

```
err = MPI_Init(&argc, &argv);
```

...

MPI 루틴이 성공적으로 실행되었을 때 리턴되는 에러코드는 헤더파일에 미리 정의되어 있는 정수 상수인 MPI\_SUCCESS 이다 . 그래서 사용자는 다음과 같이 MPI 루틴의 호출이 성공적이었는가를 검사해 볼 수 있다 .

**Fortran :**

```
IF (IERR .EQ. MPI_SUCCESS) THEN
```

```
    ... routine ran correctly ...
```

```
END IF
```

**C :**

```
if (err == MPI_SUCCESS) {
```

```
    ... routine ran correctly ...
```

```
}
```

MPI 루틴의 호출에서 만약 오류가 발생하면 특정오류를 나타내는 정수값이 리턴되며, 이때의 정수값은 시스템에 의존한다.

| Fortran           |   |
|-------------------|---|
| <b>Format</b>     | <code>CALL MPI_XXXXX(parameter,...,ierr)</code>         |
| <b>Example</b>    | <code>CALL MPI_INIT(ierr)</code>                        |
| <b>Error code</b> | Returned as "ierr" parameter, MPI_SUCCESS if successful |

| C                 |  |
|-------------------|--|
| <b>Format</b>     | <code>err = MPI_Xxxxx(parameter, ...);</code><br><code>MPI_Xxxxx(parameter, ...);</code> |
| <b>Example</b>    | <code>err = MPI_Init(&amp;argc, &amp;argv);</code>                                       |
| <b>Error code</b> | Returned as "err", MPI_SUCCESS if successful   |

그림 2.2 MPI 루틴의 호출과 리턴값

#### 2.1.4 MPI 초기화

MPI 프로그램에서 가장 먼저 호출되는 MPI 루틴은 초기화 루틴 MPI\_INIT 이다. MPI\_INIT 은 모든 MPI 프로그램에서 오직 한 번 반드시 호출되어야 하며, MPI 환경을 초기화하고 문제가 생기면 오류코드를 리턴한다.

**Fortran :**

**INTEGER IERR**

...

**MPI\_INIT(IERR)**

**C :**

```

Int err;

...

err = MPI_Init(&argc, &argv);

```

## 2.1.5 커뮤니케이터

커뮤니케이터는 서로 통신할 수 있는 프로세스들의 집합을 나타내는 MPI 핸들(handle)로서 MPI 통신은 같은 커뮤니케이터를 공유하는 프로세스들끼리만 가능하다. 커뮤니케이터 이름은 모든 점대점 통신과 집합통신에 인수로 필요하며, 송신과 그에 대응하는 수신은 커뮤니케이터가 반드시 일치해야 한다.

한 프로그램 내에는 여러 개의 커뮤니케이터가 있을 수 있고, 한 프로세스는 서로 다른 여러 커뮤니케이터에 속할 수 있다. 각 커뮤니케이터 내에서 프로세스들은 0 부터 시작되는 연속적인 정수들을 할당 받게 되는데, 이 정수들은 커뮤니케이터 내에서 프로세스를 구분 짓는 고유 번호로서 프로세스의 랭크라 부른다.

MPI\_COMM\_WORLD 는 MPI 에서 기본적으로 제공하는 커뮤니케이터로 헤더 파일에 정의되어 있으며, host.list 파일에 지정된 모든 프로세스들로 구성된다. 사용자들은 MPI\_COMM\_SPLIT 를 이용해 MPI\_COMM\_WORLD 의 부분집합으로 구성되는 추가적인 커뮤니케이터를 만들어 사용할 수 있다.

**프로세스 랭크 :** 프로세스 랭크는 동일한 커뮤니케이터에 속한 프로세스의 식별 번호로 메시지의 송신자와 수신자를 나타내기 위해 사용한다. 만약 n 개의 프로세스가 한 커뮤니케이터에 있다면 각각의 프로세스에는 0 부터 n-1 까지 번호가 할당된다.

여러 커뮤니케이터에 동시에 속한 프로세스는 각 커뮤니케이터마다 고유의 랭크를 가지게 되며, 프로세스는 루틴 MPI\_COMM\_RANK 를 호출해서 커뮤니케이터 내의 자신의 랭크를 가져올 수 있다.

**Fortran:**

**MPI\_COMM\_RANK(COMM, RANK, IERR)**

인수들은 모두 정수형 변수들이다.

**C:**

**int MPI\_Comm\_rank(MPI\_Comm comm, int \*rank)**

인수 comm 은 커뮤니케이터를 지정하는 MPI\_Comm 타입의 변수 이다 . 만약 모든 프로세스가 참여한다면 MPI\_COMM\_WORLD 를 쓰면된다 .

두 번째 인수는 정수 변수 랭크의 주소이다 .

**커뮤니케이터 사이즈 :** 프로세스는 MPI\_COMM\_SIZE 를 호출해서 자신이 속한 커뮤니케이터의 프로세스의 총 개수 , 즉 사이즈 (size) 를 결정한다 .

**Fortran:**

**MPI\_COMM\_SIZE(COMM, SIZE, IERR)**

여기서 인수들은 모두 정수형 변수들이다

**C:**

**int MPI\_Comm\_size (MPI\_Comm comm., int \*size)**

인수 comm 은 커뮤니케이터 이다 .

두 번째 인수는 정수 변수 size 의 주소이다 .

## **2.1.6 MPI 종료**

MPI 프로그램에서 마지막으로 호출되는 MPI 루틴은 MPI\_FINALIZE 이다 . MPI\_FINALIZE 는 모든 MPI 자료구조를 정리하며 , 이후에는 어떤 MPI 루틴도 호

출될 수 없다 . 모든 프로세스들에서 호출되어야 하며 , 그렇지 않으면 프로그램은 끝나지 않는다 .

**Fortran :**

**INTEGER ierr**

...

**CALL MPI\_FINALIZE(ierr)**

**C :**

**int err;**

...

**err = MPI\_Finalize();**

### **2.1.7 MPI 프로그램 : Bones**

Fortran 과 C, 각각에 대해 MPI 프로그램의 기본 골격을 소개한다 . 다음의 틀에 자신의 프로그램 코드를 삽입하여 MPI 프로그램을 작성해 가면 된다 .

예제 2.1 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
```

예제 2.2 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();
```

```
}
```

### 2.1.8 메시지

MPI 메시지는 근본적으로 데이터이며 이러한 데이터는 특정 MPI 데이터 타입을 가지는 원소들의 배열로 구성된다. 이 데이터와 데이터를 송 / 수신하는데 필요한 정보를 담은 봉투 (envelope) 가 합쳐져 하나의 메시지를 구성하게 된다.

**MPI 데이터 타입 :** MPI 는 C 와 Fortran 의 기본적인 데이터 타입에 대응하는 MPI 데이터 타입을 자체적으로 가진다. MPI 데이터 타입의 이름들은 MPI 루틴에서 인수로 사용되며, 통신에서는 기본적으로 송신 데이터 타입과 수신 데이터 타입이 동일해야 한다. C 와 Fortran 에서 사용되는 데이터 타입 이름이 조금 다르기 때문에 사용할 때 주의를 요한다.



| MPI Datatype                    | Description                               | Fortran Datatype |
|---------------------------------|---|------------------|
| MPI_INTEGER4, MPI_INTEGER       | 4-byte integer                            | INTEGER          |
| MPI_REAL4, MPI_REAL             | 4-byte floating point                     | REAL             |
| MPI_REAL8, MPI_DOUBLE_PRECISION | 8-byte floating point                     | DOUBLE PRECISION |
| MPI_COMPLEX 8, MPI_COMPLEX      | 4-byte float real, 4-byte float imaginary | COMPLEX          |
| MPI_LOGICAL4, MPI_LOGICAL       | 4-byte logical                            | LOGICAL          |
| MPI_CHARACTER                   | 1-byte character                          | CHARACTER (1)    |
| MPI_PACKED, MPI_BYTE            | N/A                                       |                  |

표 2.1 MPI 데이터 타입 : Fortran

| MPI Datatype                    | Description               | C Datatype                      |
|---------------------------------|---------------------------|---------------------------------|
| MPI_CHAR                        | 1-byte character          | signed char                     |
| MPI_UNSIGNED_CHAR               | 1-byte unsigned character | unsigned char                   |
| MPI_SHORT                       | 2-byte integer            | signed short int                |
| MPI_INT, MPI_LONG               | 4-byte integer            | signed int, signed long int     |
| MPI_UNSIGNED_SHORT              | 2-byte unsigned integer   | unsigned short int              |
| MPI_UNSIGNED, MPI_UNSIGNED_LONG | 4-byte unsigned integer   | unsigned int, unsigned long int |
| MPI_FLOAT                       | 4-byte floating point     | float                           |
| MPI_DOUBLE, MPI_LONG_DOUBLE     | 8-byte floating point     | double, long double             |
| MPI_BYTE, MPI_PACKED            | N/A                       |                                 |

표 2.2 MPI 데이터 타입 : C

Fortran에서는 모두 정수로 처리되지만, C의 경우 MPI는 다음과 같은 몇 가지 특별한 데이터 타입을 제공한다.

- MPI\_Comm : 커뮤니케이터
- MPI\_Status : MPI 호출에 필요한 정보들을 포함하는 자료구조
- MPI\_Datatype : 데이터 타입 핸들

- MPI\_Request : request 핸들
- MPI\_Op : reduction 연산자 핸들

이외에 위의 기본적인 데이터 타입들로부터 구축되는 유도 데이터 타입은 C 에서의 struct 구문과 유사하며 뒤에서 설명하게 될 것이다.

**데이터 :** MPI 메시지에서 데이터는 버퍼, 데이터 타입, 데이터 개수의 3 가지로 구성된다.

- **버퍼 (buffer) :** 송신 또는 수신되는 데이터가 저장되어 있는 메모리내의 시작 위치를 나타낸다.
- **데이터 타입 (data type) :** 대응되는 송신과 수신에서 반드시 같은 데이터 타입을 지정해야 한다. 송신과 수신에서 서로 일치하지 않는 데이터 타입을 이용해야 하는 경우 MPI\_PACKED 를 사용할 수 있다.
- **데이터 개수 (count) :** 데이터 원소의 개수

**봉투 (Envelope):** MPI 메시지의 봉투는 보내는 곳 (source), 받는 곳, 커뮤니케이터, 꼬리표의 4 가지로 구성된다.

- 보내는 곳 (source) : 송신 프로세스의 랭크
- 받는 곳 (destination) : 수신 프로세스의 랭크
- 커뮤니케이터 : 보내는 곳과 받는 곳이 포함된 프로세스 그룹을 나타낸다.
- 꼬리표 (tag) : 송 / 수신하는 메시지들을 구별하기 위해 이용한다.

## 2.2 점대점 통신 서브루틴

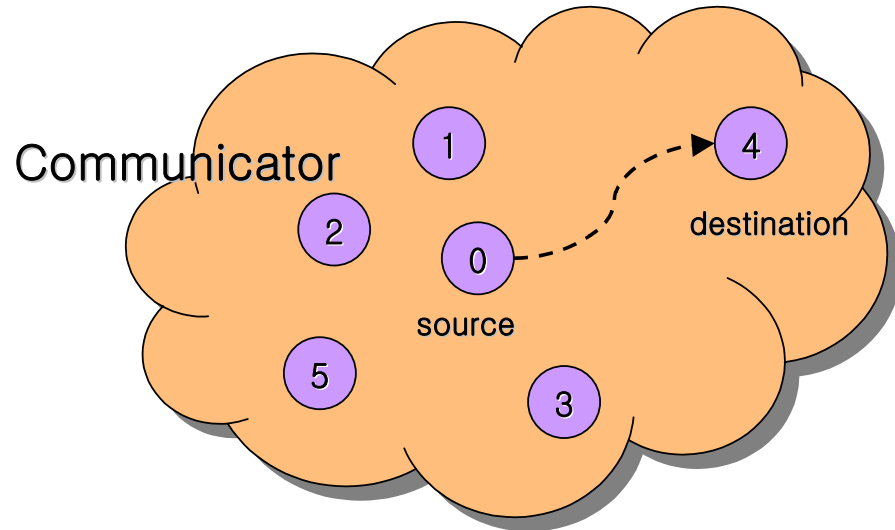


그림 2.3 점대점 통신

점대점 통신은 두 프로세스 사이의 통신이며 이 때 두 프로세스는 메시지를 보내는 송신 프로세스와 메시지를 받는 수신 프로세스가 된다. 통신은 커뮤니케이터 내의 프로세스 사이에서만 이루어지며 프로세스 랭크를 이용해 송신 프로세스와 수신 프로세스를 서로 확인하게 된다.

송신 프로세스와 수신 프로세스 간의 메시지 전송에 이용된 메모리 위치에 안전하게 접근할 수 있게 되었을 때 통신이 완료되었다고 말한다. 따라서 메시지를 보낼 때 사용되는 송신 변수는 통신이 완료되어야 다른 곳에서 다시 사용될 수 있고 메시지를 받는 수신 변수도 통신이 완료되어야 비로소 사용이 가능하게 된다.

점대점 통신은 크게 블로킹과 논블로킹의 두 가지 통신모드를 제공한다. 블로킹 통신 루틴을 사용하게 되면 프로그램은 통신이 완료되어야 통신루틴으로부터 리턴되며 논블로킹 통신 루틴을 사용하게 되면 프로그램은 통신의 완료여부와 상관 없이 송신 또는 수신 시작과 동시에 리턴되어 다른 작업을 수행할 수 있다. 이후 `MPI_WAIT` 또는 `MPI_TEST` 등의 루틴을 호출하여 시작된 통신의 완료여부를 검사하거나 아니면 통신이 완료될 때까지 대기하고 있게 된다.

비록 사용이 조금 복잡하기는 하지만 논블로킹 통신은 교착 (deadlock) 을 피할 수 있고, 또 대응하는 블로킹 통신보다 일반적으로 빠르기 때문에 많이 이용된다.

### 2.2.1 통신모드

통신 완료에 요구되는 조건에 따라 점대점 통신의 송신 루틴은 4 가지 통신모드로 구분할 수 있으며 각각 블록킹 루틴과 논블록킹 루틴으로 구분해 아래와 같이 정리할 수 있다. 수신 루틴의 경우는 통신모드 구분 없이 블록킹 루틴과 논블록킹 루틴으로 구분한다.

| Communication Mode | Blocking Routines      | Non-Blocking Routines   |
|--------------------|------------------------|-------------------------|
| Synchronous Send   | <code>MPI_SSEND</code> | <code>MPI_ISSEND</code> |
| Ready Send         | <code>MPI_RSEND</code> | <code>MPI_IRSEND</code> |
| Buffered Send      | <code>MPI_BSEND</code> | <code>MPI_IBSEND</code> |
| Standard Send      | <code>MPI_SEND</code>  | <code>MPI_ISEND</code>  |
| Receive            | <code>MPI_RECV</code>  | <code>MPI_ISEND</code>  |

표 2.3 여러 가지 통신 모드

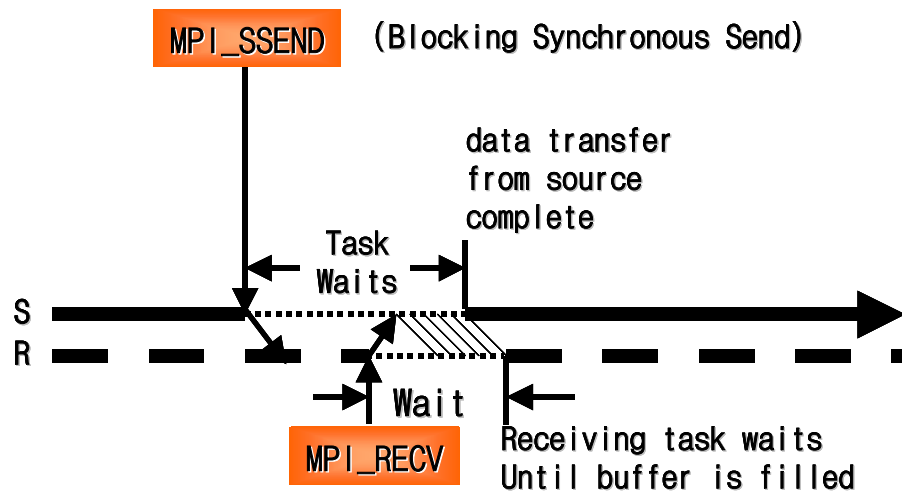


그림 2.4 동기 송신

**블록킹 동기 송신 (Synchronous Send):** 그림 2.4에서 시간은 왼쪽에서 오른쪽으로 흐르며 S 는 송신 프로세스가 송신 작업을 수행하는 시간을 나타내고 , R 은 수신 프로세스가 수신 작업을 수행하는 시간을 나타내고 있다 . 송 /수신 수행 시간을 나타내는 선이 끊어 지며 나타나는 점선은 메시지 패싱 이벤트 때문에 발생한 인터럽트로 인하여 각 프로세스가 대기상태에 있음을 나타낸다 .

동기 모드 송신 루틴 (MPI\_Ssend) 을 호출하면 송신 프로세스는 우선 송신 준비가 되었음을 수신측에 알린다 . 수신 프로세스는 수신 루틴을 호출하고 수신 준비가 되었음을 송신측에 알리고 송신 프로세스가 수신 준비 완료 메시지를 받으면 , 데이터 전송이 시작된다 . 데이터 전송에 소비되는 시스템 부하와 동기화 과정으로 인한 동기화 부하로 인해 성능은 느리지만 메시지가 전달되지 않거나 사라지는 상황을 방지할 수 있어 안전한 통신을 수행할 수 있다 .

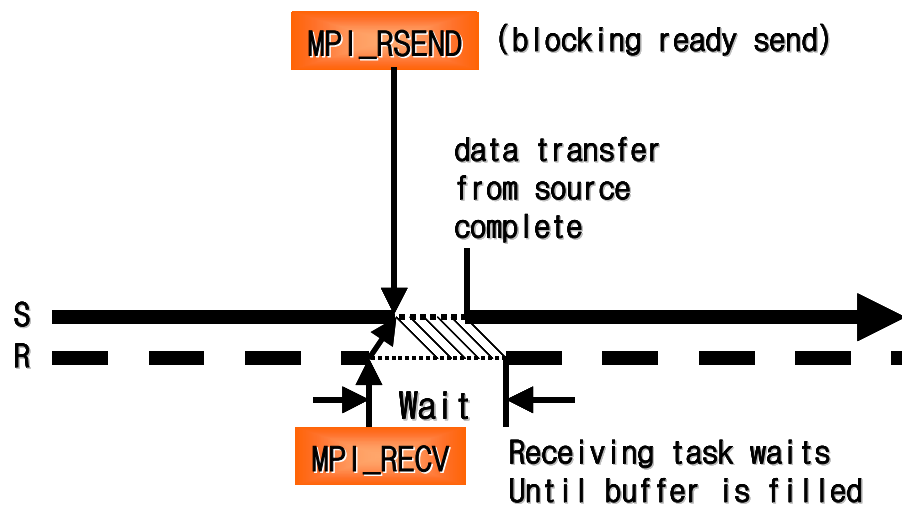


그림 2.5 준비 송신

**블록킹 준비 송신 (Ready Send):** 준비 송신은 목적지 프로세스에 이미 수신 루틴이 준비되어 있어야 한다 . 송신 프로세스는 메시지를 통신 네트워크로 보내고 즉시 완료되며 , 이미 수신 프로세스가 메시지를 받기 위해 대기 (ready) 하고 있음을 가정한다 . 만일 수신 프로세스가 메시지를 받을 준비를 하고 있으면 무사히 수신 될 것이고 그렇지 않으면 메시지는 사라지고 오류가 발생한다 .

준비 송신은 송신 작업에 의한 시스템 부하와 동기화 부하를 최소화 하기 때문에 성능향상에 유리하다. 그러나, 수신 측에서 대응되는 송신보다 얼마나 일찍 준비를 하고 있느냐에 의존하는 동기화 부하는 여전히 남아있으며, 디버깅에 어려움이 있다.

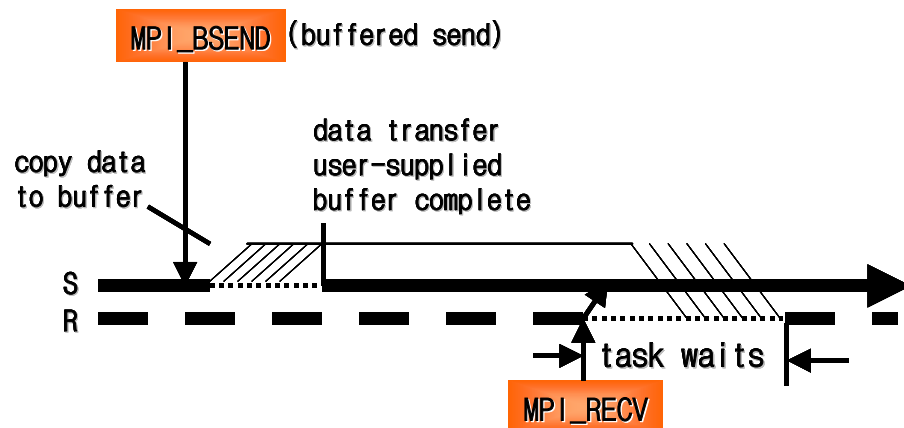


그림 2.6 버퍼 송신

**블록킹 버퍼 송신 (Bufferd Send):** MPI 는 송신될 때 까지 데이터를 보관할 수 있는 버퍼를 사용자가 지정해 쓸 수 있도록 하는 기능을 제공한다. 버퍼 송신은 버퍼 송신 루틴 (MPI\_Bsend) 이 호출되면 즉시, 유저가 정의한 버퍼로 데이터를 복사해 두고 수신 준비가 완료되면 이 버퍼로부터 수신 프로세스로 송신을 하게 된다. 송신 프로세스는 준비된 버퍼로 데이터 복사가 완료되면 송신 버퍼를 다른 목적으로 사용할 수 있다. 버퍼 송신은 송신 버퍼에서 사용자 정의 버퍼로의 작업으로 인해 추가적인 시스템 부하가 생기지만 송신측의 동기화 부하는 없어진다. 수신 작업에 의한 수신 측의 동기화 부하는 여전히 남아있으며 사용자는 버퍼 공간의 부가와 관리를 직접 담당해야 한다. 사용자는 MPI 루틴 MPI\_BUFFER\_ATTACH 와 MPI\_BUFFER\_DETACH 를 이용해 통신에 이용할 버퍼를 확보하고 해제 시킬 수 있다.

C :

**MPI\_Buffer\_attach (&buffer,size)**

**MPI\_Buffer\_detach (&buffer,size)**

**Fortran :**

**MPI\_BUFFER\_ATTACH (buffer,size,ierr)**

**MPI\_BUFFER\_DETACH (buffer,size,ierr)**

**블록킹 표준 송신 (Standard Send):** 블록킹 표준 모드 송신은 메시지 크기에 따라 두 가지 모드로 작동한다. 메시지 크기가 임계값보다 작으면 시스템에 준비된 시스템 버퍼를 이용하여 버퍼 송신과 같이 작동하며, 임계값보다 메시지 크기가 크면 동기 송신처럼 작동하게 된다. 임계값은 시스템 상황에 따라 다르게 설정할 수 있으며 참고로, IBM 의 MPI 라이브러리 구현 제품인 PE 에서는 환경변수 MP\_EAGER\_LIMIT 를 이용해 임계값 설정을 할 수 있다.

이렇듯 표준 모드 송신이 두 가지 모드로 작동되는 이유는 시스템 상황에 따라 통신의 안전성과 성능을 고려하여 최적화된 통신을 하기 위한 것이다. 표준 송신 모드는 이렇듯 메시지 크기에 따라 그 작동방식이 다르기 때문에 사용자는 송신의 완료가 메시지의 도착을 의미하지는 않는다는 사실을 항상 유념해야 한다.

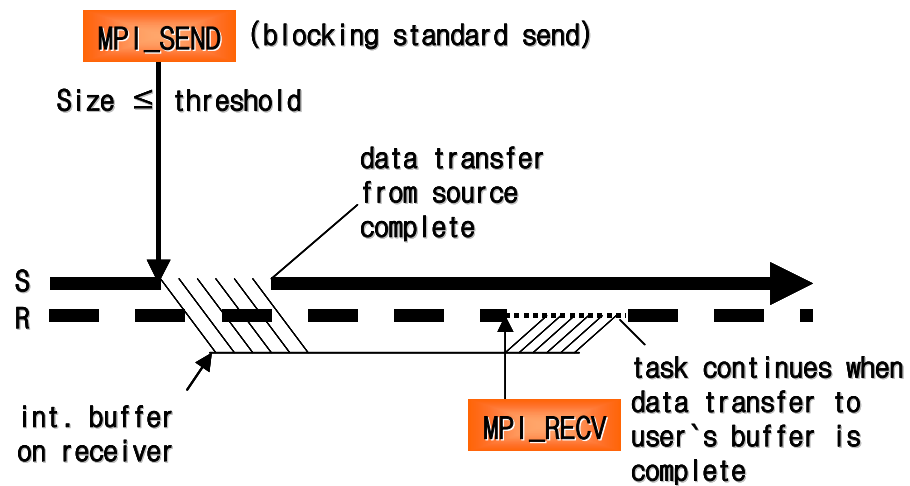


그림 2.7 블록킹 표준 통신 ( 메시지 크기 ≤ 임계값 )

보내고자 하는 메시지의 크기가 시스템 임계값보다 작을 때 루틴 MPI\_Send 를 호출한 수신 노드는 네트워크를 거쳐 수신 노드의 시스템 버퍼로 메시지를 우선 복사한다 . 그리고 , 곧바로 리턴되어 다른 작업을 수행할 수 있다 . 수신 루틴 MPI\_RECV 가 호출되면 시스템 버퍼의 메시지는 수신 프로세스로 복사되고 복사가 완료되면 수신 프로세스는 역시 다른 작업을 수행할 수 있다 .

시스템 버퍼를 사용하여 송신 측에서의 동기화 부하를 없애는 것은 버퍼 송신과 동일하나 , 표준 모드에서 사용되는 시스템 버퍼는 프로그램 시작시에 프로세스마다 하나씩 부가되어 사용자의 관리가 따로 필요없다 .

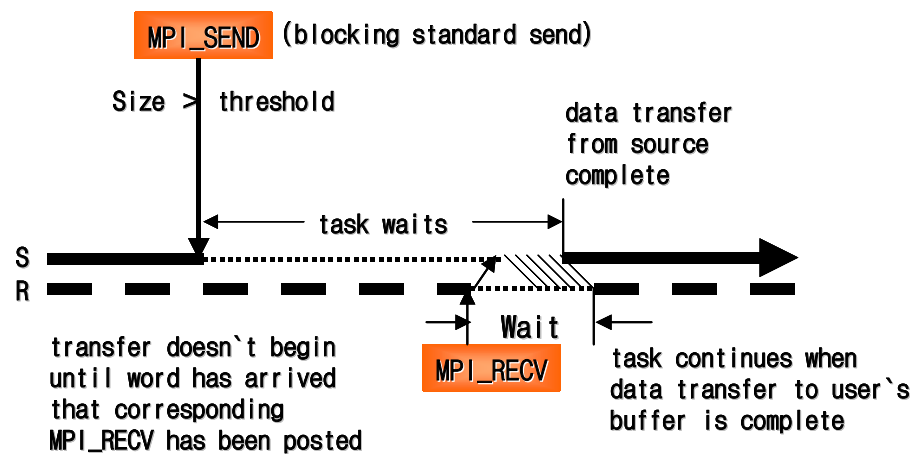


그림 2.8 블록킹 표준 통신 ( 메시지 크기 > 임계값 )

메시지 크기가 임계값보다 클 때의 표준 모드 송신은 동기 모드 송신과 동일하게 행동한다. 사용자는 임계값 설정을 0으로 함으로서 모든 표준 모드 송신을 가장 안전한 동기 송신과 똑같이 행동하도록 할 수 있다 .

이상에서 여러 가지 점대점 통신 모드에 대해서 알아보았다 . 이 중 일반적으로 가장 많이 사용되는 것은 표준 모드 송신과 수신이며 다음 절에서 표준 모드 송신과 수신을 블록킹 통신과 논블록킹 통신으로 나누어 좀더 자세히 알아볼 것이다 .



## 2.2.2 블록킹 송신과 수신

MPI\_SEND 와 MPI\_RECEIVE 는 MPI 에서 가장 기본적인 점대점 통신 루틴들이다 . 두 루틴은 통신이 완료될 때 까지 두 루틴을 호출한 프로세스가 다른 작업을 못하도록 막고 ( 블록킹 ) 있는 블록킹 연산이다 . 블록킹 연산은 교착 (deadlock) 이 발생하지 않도록 주의해서 사용하여야 한다 .

### 블록킹 송신 : MPI\_SEND

C:

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

Fortran:

```
MPI_SEND(buf, count, datatype, dest, tag, comm, ierr)
```

(CHOICE) buf : 송신 버퍼의 시작 주소 (out)

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 , 0~232-1 (IN)

INTEGER comm : MPI 커뮤니케이터 ( 핸들 ) (IN)

MPI\_SEND는 블록킹 표준 모드 송신 루틴으로, 위에서 MPI\_SEND 루틴은 datatype 타입의 count 개 원소들을 buf 에서 dest 프로세스로 송신한다 . dest 는 커뮤니케이터 comm 에 속한 총 n 개의 프로세스중에 하나의 프로세스 랭크를 나타낸다 . 송신된 메시지는 MPI\_RECV 나 MPI\_IRECV 로 수신할 수 있다 .

### 블록킹 수신 : MPI\_RECV

C:

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

**Fortran:**

**MPI\_RECV(buf, count, datatype, source, tag, comm, status, ierr)**

(CHOICE) buf : 수신 버퍼의 시작 주소 (OUT)

INTEGER count : 수신될 원소 개수 (IN)

INTEGER datatype : 각 원소의 MPI 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 는 블록킹 수신이다 . 수신버퍼는 datatype 타입의 count 개의 연속적인 원소들을 저장할 공간으로 , 주소 buf 로 시작된다 . 수신되는 메시지의 크기는 수신 버퍼의 크기보다 작거나 같아야 하며 만약 수신버퍼보다 큰 메시지가 수신되면 오버 플로우 오류가 발생한다 . 모든 프로세스로부터의 모든 메시지를 수신하기 위해 source 와 꼬리표에 와일드카드 (MPI\_ANY\_SOURCE, MPI\_ANY\_TAG) 를 사용할 수 있다 . 그러나 , 커뮤니케이터에는 와일드카드를 사용할 수 없다 .

인수 status 는 수신된 메시지에 대한 source 와 꼬리표 등의 정보를 출력하는 MPI\_STATUS\_SIZE 원소들을 가지는 정수형 배열이다 .

| Information | Fortran            | C                 |
|-------------|--------------------|-------------------|
| Source      | status(MPI_SOURCE) | status.MPI_SOURCE |
| Tag         | status(MPI_TAG)    | status.MPI_TAG    |
| count       | MPI_GET_COUNT      | MPI_Get_count     |

표 2.4 status 에 저장되는 정보

사용자는 다음과 같이 루틴 MPI\_Get\_count 를 호출하여 수신된 메시지의 원소 개수를 확인할 수 있다 .

**C :**

**int MPI\_Get\_count(MPI\_Status \*status, MPI\_Datatype datatype, int \*count)**

**Fortran :**

**MPI\_GET\_COUNT(status, datatype, count, ierr)**

INTEGER status(MPI\_STATUS\_SIZE) : 수신된 메시지의 상태 (IN)

INTEGER datatype : 각 원소의 데이터 타입 (IN)

INTEGER count : 원소의 개수 (OUT)

**블록킹 통신 사용 예제 :** 다음 예제는 1 번 프로세스의 배열 data 를 표준 블록킹 통신을 이용하여 0 번 프로세스로 보내는 MPI 프로그램이다 . 0 번 프로세스에서는 수신 완료된 후 , 루틴 MPI\_GET\_COUNT 를 호출하여 수신된 데이터의 원소 개수를 확인하고 있다 .

예제 2.3 블록킹 통신을 이용한 MPI 프로그램 : Fortran

PROGRAM std\_block

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)

ELSE

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

```

예제 2.4 블록킹 통신을 이용한 MPI 프로그램 : 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 {

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();

}

```

**교착 (Deadlock):** 블록킹 통신을 실행할 때 가장 주의해야 할 것이 교착에 빠지는 것이다 . 교착은 2 개의 프로세스들이 블록킹되어 각각 다른 프로세스의 진행을 기다릴 때 발생하는데 , MPI 프로그램에서 가장 조심해야 할 오류중의 하나인 교착에 대한 간단한 예제를 보자 .

예제 2.5 교착 1 : Fortran

```

PROGRAM deadlock

INCLUDE 'mpif.h'

INTEGER nprocs, myrank, ierr, status(MPI_STATUS_SIZE)

REAL a(100), b(100)

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_RECV(b, 100, MPI_REAL, 1, 19, MPI_COMM_WORLD, status, &
ierr)

    CALL MPI_SEND(a, 100, MPI_REAL, 1, 17, MPI_COMM_WORLD, ierr)

ELSE IF (myrank==1) THEN

    CALL MPI_RECV(b, 100, MPI_REAL, 0, 17, MPI_COMM_WORLD, status, &
ierr)

    CALL MPI_SEND(a, 100, MPI_REAL, 0, 19, MPI_COMM_WORLD, ierr)

ENDIF

CALL MPI_FINALIZE(ierr)

END

```

예제 2.6 교차 1 : C

```

/*deadlock*/

#include <mpi.h>

#include <stdio.h>

void main (int argc, char *argv[]){

    int nprocs, myrank ;

    MPI_Status status;

    double a[100], b[100];

    MPI_Init(&argc, &argv);

```

```

MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

if (myrank ==0) {

    MPI_Recv(b, 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD, &status);

    MPI_Send(a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD);

}

else if (myrank==1) {

    MPI_Recv(b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &status);

    MPI_Send(a, 100, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD);

}

MPI_Finalize();

}

```

위 예제에서 각 프로세스는 서로에 의해 송신되는 메시지를 수신하면서 시작되는데 프로세스 0 은 프로세스 1 이 메시지를 보내야 진행이 되며, 프로세스 1 은 역시 프로세스 0 이 메시지를 보내야 진행이 된다. 아무런 메시지도 송신되지않고 따라서 아무런 메시지도 수신되지 않는다.

다음의 프로그램은 앞의 예제와 유사하다. 앞서와 마찬가지로 프로세스 0 은 프로세스 1 과 데이터 교환을 시도하는데, 이번에는 두 프로세스가 송신을 먼저 시도한다. 이런 경우에는 통신의 성공여부는 MPI의 버퍼링 능력과 전송하는 메시지의 크기에 따라 달라진다. 작은 문제에서 코드는 아무런 문제없이 실행이 되지만, 큰 문제에 대해서는 동기화 모드 송신을 시도하며 교착에 빠지게 된다. 이렇게 MPI 내부버퍼의 크기에 의존하는 프로그램은 이식성과 확장성(scalability) 등의 측면에서 바람직하지 않다. 아래 예제에서 주고 받는 배열의 크기를 100000000 로 해서 실행시켜 보라. 아마도 대부분의 시스템에서 틀림없이 교착에 빠지게 될 것이다.

예제 2.7 교착 2 : Fortran

```

PROGRAM probable_deadlock

INCLUDE 'mpif.h'

INTEGER nprocs, myrank, ierr, status(MPI_STATUS_SIZE)

REAL a(100), b(100)

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_SEND(a, 100, MPI_REAL, 1, 17, MPI_COMM_WORLD, ierr)

    CALL MPI_RECV(b, 100, MPI_REAL, 1, 19, MPI_COMM_WORLD, status, &
        ierr)

ELSE IF (myrank==1) THEN

    CALL MPI_SEND(a, 100, MPI_REAL, 0, 19, MPI_COMM_WORLD, ierr)

    CALL MPI_RECV(b, 100, MPI_REAL, 0, 17, MPI_COMM_WORLD, status, &
        ierr)

ENDIF

CALL MPI_FINALIZE(ierr)

END

```

예제 2.8 교차 2 : C

```

/* probable_deadlock*/

#include <mpi.h>

#include <stdio.h>

```



```

void main (int argc, char *argv[]){

    int nprocs, myrank ;

    MPI_Status status;

    double a[100], b[100];

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    if (myrank ==0) {

        MPI_Send(a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD);

        MPI_Recv(b, 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD, &status);

    }

    else if (myrank==1) {

        MPI_Send(a, 100, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD);

        MPI_Recv(b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &status);

    }

    MPI_Finalize();

}

```

**교착 피하기 :** 다음은 앞의 예제와 유사하지만 통신 순서를 다르게 구성해 교착을 피하고 있다 . 앞서와 같이 프로세스 0 과 프로세스 1 은 서로 데이터 교환을 시도한다 . 프로세스 0 은 수신을 하고 그리고 송신하며 , 프로세스 1 은 송신을 하고 수신 을 한다 . 통신은 아무런 오류를 일으키지 않으며 안전하게 프로그램을 실행하게 된다 . 메시지의 크기나 배열의 차원을 증가시키는 것이 프로그램의 실행에 아무런 영향을 미치지 않는다 .

예제 2.9 블록킹 표준 통신에서 교착 피하기 : Fortran

```
PROGRAM safe_deadlock

INCLUDE 'mpif.h'

INTEGER nprocs, myrank, ierr, status(MPI_STATUS_SIZE)

REAL a(100), b(100)

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_RECV(b, 100, MPI_REAL, 1, 19, MPI_COMM_WORLD, status, &
        ierr)

    CALL MPI_SEND(a, 100, MPI_REAL, 1, 17, MPI_COMM_WORLD, ierr)

ELSE IF (myrank == 1) THEN

    CALL MPI_SEND(a, 100, MPI_REAL, 0, 19, MPI_COMM_WORLD, ierr)

    CALL MPI_RECV(b, 100, MPI_REAL, 0, 17, MPI_COMM_WORLD, status, &
        ierr)

END IF

CALL MPI_FINALIZE(ierr)

END
```

예제 2.10 블록킹 표준 통신에서 교착 피하기 : C

```
/*safe_deadlock*/

#include <mpi.h>
```

```

#include <stdio.h>

void main (int argc, char *argv[]){

    int nprocs, myrank ;

    MPI_Status status;

    double a[100], b[100];

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    if (myrank ==0) {

        MPI_Recv(b, 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD, &status);

        MPI_Send(a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD);

    }

    else if (myrank==1) {

        MPI_Send(a, 100, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD);

        MPI_Recv(b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &status);

    }

    MPI_Finalize();

}

```

### 2.2.3 논블록킹 송신과 수신

블록킹 통신 루틴인 MPI\_SEND 와 MPI\_RECV 는 통신이 완료될 때까지 호출한 프로세스를 블록킹해 두기 때문에 프로그램 실행이 느려지고 때로는 교착에 걸리게 된다 . 이런 문제에 대해 MPI 는 송신 또는 수신 연산의 초기화와 종료를 따로 분리해서 호출하게 함으로써 프로그램이 두 호출사이에서 다른 작업을 할 수 있도록 하는 논블록킹 통신 루틴을 제공한다 .

논블록킹 통신을 이용하면 프로세스는 먼저 통신 초기화를 위해 MPI 루틴을 호출 하지만 루틴은 통신이 완료되기 전에 리턴되어 프로세스는 다른 작업을 수행할 수 있다. 통신은 백그라운드로 계속되며 사용자는 통신의 성공여부가 필요할 시점에서 프로세스에 대한 블로킹없이 통신이 성공적으로 수행되었는가를 검사 (Test) 하거나 혹은 통신이 완료되기를 기다리게 할 수 있다 (Wait). 논블록킹 통신 루틴을 호출해 초기화 하는 것을 포스팅 (posting) 이라고 하며, 프로세스들은 포스팅된 연산들의 상태를 점검하거나 연산이 완료되기를 기다리게 하기 위해 request 핸들을 이용한다. 사용자는 논블록킹 연산을 이용함으로써 교착을 피할 수 있고, 동기화 부하를 줄일 수 있다.

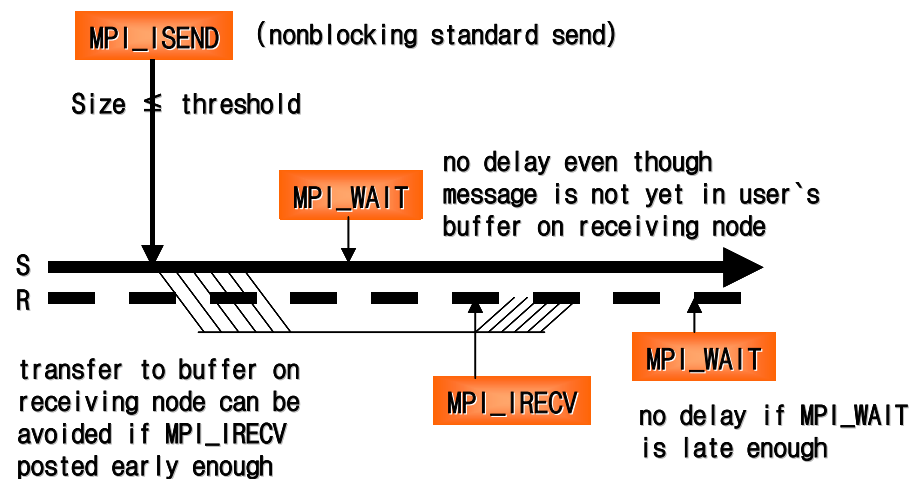


그림 2.9 논블록킹 표준 통신 (메시지 크기 ≤ 임계값)

그림 2.9 는 메시지 크기가 임계값보다 작은 경우 논블록킹 표준 통신의 행동을 보여준다. 송신 프로세스는 MPI 송신 루틴 (MPI\_Isend) 이 호출되고 전송할 준비가 되면 시스템 버퍼로 메시지 복사가 완료될 때 까지 기다리지 않고 바로 리턴되어 다른 작업을 수행하며, 송신 프로세스가 송신 버퍼를 덮어 쓸 필요가 있기 바로 전에 MPI\_Wait 루틴을 호출하여 송신 버퍼를 안전하게 사용할 수 있도록 하고 있다.

수신 프로세스는 수신 버퍼가 메시지를 받을 준비가 되면 바로 수신 루틴 (MPI\_Irecv) 를 호출하고 메시지가 수신 버퍼에 도착할 때까지 기다리지 않고 바로 리턴된다. 역시, 수신 버퍼를 사용할 필요가 있기 바로 전에 MPI\_Wait 루틴을 호출하여 수신된 데이터를 안전하게 사용할 수 있도록 하고 있다.

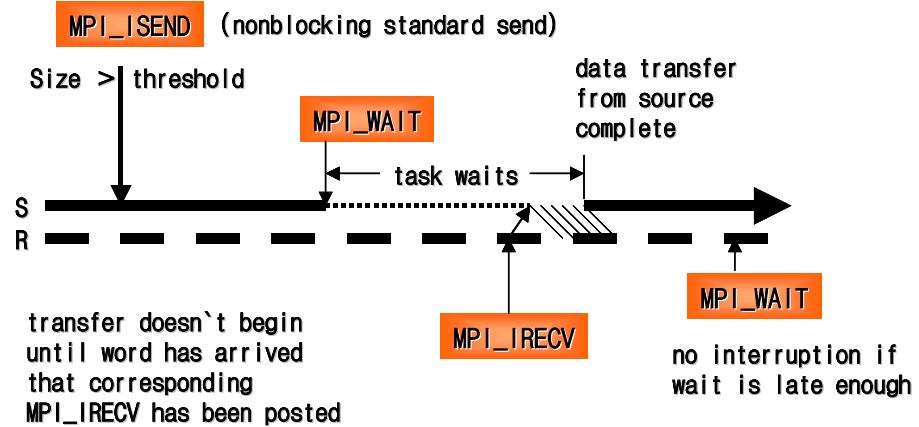


그림 2.10 논블록킹 표준 통신 ( 메시지 크기 > 임계값 )

그림 2.10 은 메시지 크기가 임계값보다 큰 경우 논블록킹 표준 통신의 행동을 보여 준다. 블록킹 통신과 비교해 논블록킹 송신 루틴(MPI\_Isend)의 호출부터 MPI\_Wait 호출까지의 동기화 부하가 줄어 들었다. 만약 수신 루틴 (MPI\_Irecv)의 호출이 먼저 되어 수신 대기가 일찍 된다면 동기화 부하를 더 줄일 수 있을 것이다.

#### 논블록킹 송신 : MPI\_ISEND

C:

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag,
MPI_Comm comm, MPI_Request *request)
```

Fortran:

```
MPI_ISEND(buf, count, datatype, dest, tag, comm, request, ierr)
```

(CHOICE) buf : 송신 버퍼의 시작 주소 (out)

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, 0~232-1 (IN)

INTEGER comm : MPI 커뮤니케이터 ( 핸들 ) (IN) INTEGER request : 통신 request, 초기화된 통신의 식별에 이용 ( 핸들 ) (OUT)

블록킹 루틴 MPI\_SEND 와 비교해 인수에 request 핸들이 추가되었다. 송신 버퍼는 MPI\_WAIT, MPI\_TEST 와 같은 대기 또는 검사 함수들에 의해 호출이 완료될 때까지 수정되어서는 안된다. MPI\_ISEND 로 송신된 메시지는 MPI\_RECV 또는 MPI\_Irecv 로 수신 할 수 있다.

**논블록킹 수신 : MPI\_Irecv**

**C:**

**int MPI\_Irecv(void \*buf, int count, MPI\_Datatype datatype, int source, int tag, MPI\_Comm comm, MPI\_Request \*request)**

**Fortran:**

**MPI\_Irecv(buf, count, datatype, source, tag, comm, request, ierr)**

(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 request : 통신 request, 초기화된 통신의 식별에 이용 ( 핸들 ) (OUT)

블록킹 루틴 MPI\_RECV 와 비교해 출력 인수 status 대신 request 핸들이 이용된다. MPI\_Irecv 루틴이 호출되었다는 것은 시스템이 수신 버퍼에 데이터를 쓸 준비가

되었다는 것을 의미한다. 한 번 논블록킹 루틴이 호출되면 수신이 끝날 때 까지 수신 버퍼에 접근할 수 없다. 수신된 메시지의 크기는 수신버퍼의 크기보다 작거나 같아야 하며 만약 수신버퍼보다 큰 메시지가 수신되면 오버플로우 오류가 발생한다. MPI\_IRecv 는 MPI\_SEND 나 MPI\_ISEND 로 송신된 메시지를 받을 수 있다.

#### **논블록킹 통신의 완료 :**

논블록킹 루틴에 의해 포스팅된 송신 또는 수신은 통신 버퍼를 다시 사용하거나, 통신결과를 이용하기 전에 완료되어야 한다. 송신 또는 수신은 완료 상태는 완료 루틴 들에 의해 검사되는데, MPI 는 블록킹 완료 루틴 (MPI\_WAIT, ...) 과 논블록킹 완료루틴 (MPI\_TEST, ...) 을 제공한다.

#### **대기 : MPI\_WAIT**

**C :**

**int MPI\_Wait(MPI\_Request \*request, MPI\_Status \*status)**

**Fortran :**

**MPI\_WAIT(request, status, ierr)**

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

INTEGER status(MPI\_STATUS\_SIZE) : 수신 메시지에 대한 정보 또는 송신 루틴에 대한 에러코드 (OUT)

MPI\_WAIT 는 request 에 의해 확인되는 연산이 완료될 때 까지 프로세스를 블록킹 하고 연산이 완료된 이후에 리턴된다. 따라서, 논블록킹연산과 이후에 바로 따라오는 MPI\_WAIT 는 블록킹 연산과 동일하다.

#### **검사 : MPI\_TEST**

**C :**

**int MPI\_Test(MPI\_Request \*request, int \*flag, MPI\_Status \*status)**

**Fortran :**

**MPI\_TEST(request, flag, status, ierr)**

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)

request에 의해 확인되는 연산의 완료 여부를 검사하여 그 결과(TRUE 또는 FALSE)를 즉시 인수 flag 를 통하여 리턴한다 .

**블록킹 통신 사용 예제 :** 두 개의 프로세스가 모두 수신 루틴을 먼저 호출하는 경우 , 앞서 블록킹 연산을 이용했을 때는 교착에 빠졌지만 , 논블록킹 연산을 이용하여 교착을 피해가는 것을 다음의 예가 보여준다 .

예제 2.11 논블록킹 표준 통신을 이용한 교착 피하기 : Fortran

```
PROGRAM avoid_deadlock
```

```
INCLUDE 'mpif.h'
```

```
INTEGER nprocs, myrank, ierr, status(MPI_STATUS_SIZE)
```

```
INTEGER request
```

```
REAL a(100), b(100)
```

```
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_IRECV(b, 100, MPI_REAL, 1, 19, MPI_COMM_WORLD, &
```



```

    request, ierr)

    CALL MPI_SEND(a, 100, MPI_REAL, 1, 17, MPI_COMM_WORLD, ierr)

    CALL MPI_WAIT(request, status, ierr)

ELSEIF (myrank==1) THEN

    CALL MPI_IRECV(b, 100, MPI_REAL, 0, 17, MPI_COMM_WORLD, &
    request, ierr)

    CALL MPI_SEND(a, 100, MPI_REAL, 0, 19, MPI_COMM_WORLD, ierr)

    CALL MPI_WAIT(request, status, ierr)

ENDIF

CALL MPI_FINALIZE(ierr)

END

```

예제 2.12 논블록킹 표준 통신을 이용한 교착 피하기 : C

```

/*avoid_deadlock*/

#include <mpi.h>

#include <stdio.h>

void main (int argc, char *argv[]){

    int nprocs, myrank ;

    MPI_Request request;

    MPI_Status status;

    double a[100], b[100];

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

```

```

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

if (myrank == 0) {

    MPI_Irecv(b, 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD, &request);

    MPI_Send(a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD);

    MPI_Wait(&request, &status);

}

else if (myrank == 1) {

    MPI_Irecv(b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &request);

    MPI_Send(a, 100, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD);

    MPI_Wait(&request, &status);

}

MPI_Finalize();

}

```

## 2.2.4 점대점 통신의 사용

두 개의 프로세스가 통신에 참여하는 점대점 통신은 한 프로세스가 데이터를 보내고 다른 프로세스는 데이터를 받는 단방향 통신과 두 프로세스가 데이터를 보내는 동시에 받아야 하는 양방향 통신으로 나누어 생각할 수 있다. 특히 양방향 통신에서는 교착에 주의하여야 한다.

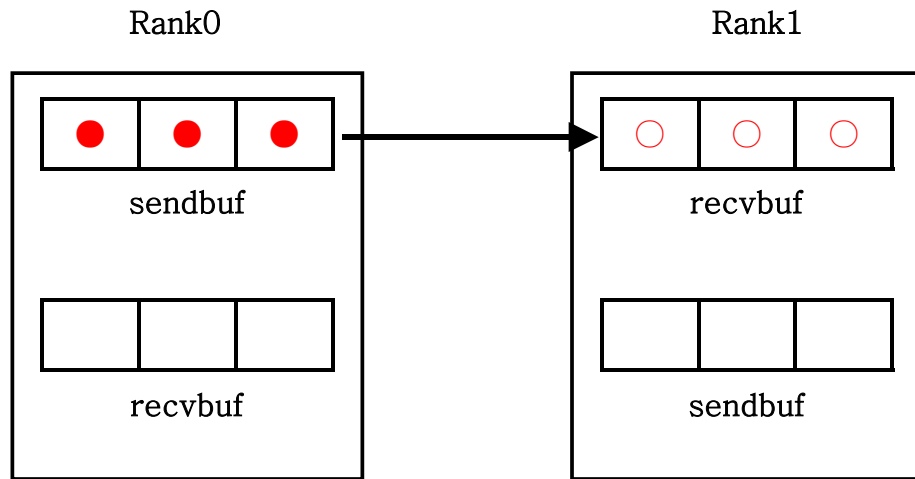


그림 2.11 단방향 통신

**단방향 통신** : 프로세스 0 으로부터 프로세스 1 로 메시지를 보낼 때 송 / 수신을 블록킹 루틴을 이용하느냐 non- 블록킹 루틴을 이용하느냐에 따라 다음과 같이 4 가지 조합을 생각할 수 있다 .

#### 경우 1. 블록킹 송신과 블록킹 수신

```
IF (myrank ==0) THEN

CALL MPI_SEND(sendbuf, icount, MPI_REAL, 1, itag, MPI_COMM_WORLD, ierr)

ELSEIF (myrank==1) THEN

CALL MPI_RECV(recvbuf, icount, MPI_REAL, 0, itag, MPI_COMM_WORLD,
istatus, ierr)

ENDIF
```

#### 경우 2. 논블록킹 송신과 블록킹 수신

```
IF (myrank ==0) THEN
```

```

CALL MPI_ISEND(sendbuf, icount, MPI_REAL, 1, itag, MPI_COMM_WORLD, ireq,
ierr)

CALL MPI_WAIT(ireq, istatus, ierr)

ELSEIF (myrank==1) THEN

CALL MPI_RECV(recvbuf, icount, MPI_REAL, 0, itag, MPI_COMM_WORLD,
istatus, ierr)

ENDIF

```

### 경우 3. 블록킹 송신과 논블록킹 수신

```

IF (myrank ==0) THEN

CALL MPI_SEND(sendbuf, icount, MPI_REAL, 1, itag, MPI_COMM_WORLD, ierr)

ELSEIF (myrank==1) THEN

CALL MPI_IRECV(recvbuf, icount, MPI_REAL, 0, itag, MPI_COMM_WORLD, ireq,
ierr)

CALL MPI_WAIT(ireq, istatus, ierr)

ENDIF

```

### 경우 4. 논블록킹 송신과 논블록킹 수신

```

If (myrank ==0) THEN

CALL MPI_ISEND(sendbuf, icount, MPI_REAL, 1, itag, MPI_COMM_WORLD, ireq,
ierr)

ELSEIF (myrank==1) THEN

CALL MPI_IRECV(recvbuf, icount, MPI_REAL, 0, itag, MPI_COMM_WORLD, ireq,
ierr)

ENDIF

CALL MPI_WAIT(ireq, istatus, ierr)

```

MPI\_WAIT 는 반드시 논블록킹 루틴 다음에 오며 송신 또는 수신 버퍼를 재사용하기 이전 어느 곳에 위치하든 상관없다 .

**양방향 통신 :** 두개의 프로세스가 서로 데이터를 교환하는 경우 사용자는 교착 (deadlock) 에 주의해야 한다 . 교착은 송신과 수신 순서를 잘못 사용하거나 또는 시스템버퍼의 제한된 크기로 인해 생기게 된다 .

양방향 통신은 두 프로세스에 의해 호출되는 송신과 수신 순서에 따라 다음과 같이 세가지 경우가 있다 .

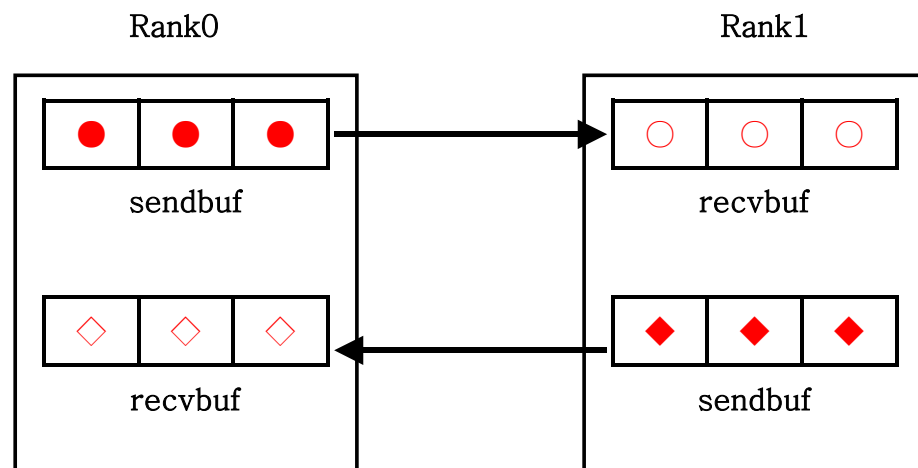


그림 2.12 양방향 통신

#### 경우 1. 선 송신 후 수신

다음 코드와 같은 경우 , 앞서 언급되었듯 주고받는 메시지의 크기가 작으면 아무 문제없이 실행되지만 메시지의 크기가 커지면 교착에 걸린다 .

```
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

```

다음의 코드는 논블록킹 송신 루틴을 사용하고 있지만 MPI\_Wait 루틴을 호출하기 때문에 위의 경우와 똑 같은 경우이다 .

```

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

```

위와 유사하지만 아래의 코드는 논블록킹 송신 루틴 MPI\_ISEND 로부터 바로 리턴 되어 , 데이터 수신을 시작하므로 교착에 걸리지 않는다 .

```

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

```

## **경우 2. 선 수신 후 송신**

가장 조심해야 하는 경우로 , 다음 코드는 데이터 크기에 무관하게 교착에 걸리게 된다 .

```

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

```

위의 코드에서 MPI\_SEND 대신 MPI\_ISEND 를 사용해도 역시 교착에 걸리게 되지 만 논블록킹 수신 루틴을 호출하는 다음의 코드는 안전하게 수행될것이다 .

```

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

```

### 경우 3. 한 쪽은 송신부터 다른 한 쪽은 수신부터

한 프로세스에서 송신 서브루틴을 먼저 호출하면 대응하는 프로세스에서는 수신 서브루틴을 먼저 호출하는 순서로 어떤 경우든 교착에 걸릴 위험은 없다. 이런 경우 사용자는 블록킹 또는 논블록킹 서브루틴중 어떤 것을 써도 무방하다.

```

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

```

**권장코드 :** 이상에서 살펴본 내용으로 양방향 점대점 통신을 사용하는 경우, 프로그램의 성능과 교착 여부를 고려해 다음과 같은 형태의 코드 사용을 권장한다.

```

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,...)

```

## 2.3 집합 통신 서브루틴

일반적으로 프로세스들 사이의 모든 데이터 이동은 점대점 통신을 이용해 처리할 수 있다. 그러나, 경우에 따라서 집합 통신의 사용으로 통신을 위한 코드 작업을 간결하게 할 수도 있고, 성능측면에서도 이득을 얻을 수 있다. 집합 통신 루틴도 근본적으로는 점대점 통신을 이용해 구성이 되어있어, 사용자는 직접 자신의 집합 통신 루틴을 만들어 사용할 수도 있지만, MPI가 제공하는 집합 통신 루틴들은 가장 효율적인 알고리즘을 이용해, 최적화된 루틴들이므로 편리성과 성능면에서 제공된 루틴을 사용하는 것이 유리하다.

프로그램에서 집합 통신 서브루틴을 호출할 때는 커뮤니케이터 인수를 이용해 어떤 프로세스들이 통신에 참여하는가를 결정하게 된다. 물론, 사용자는 직접 커뮤니케이터를 지정해 집합 통신에 참여하는 프로세스를 지정할 수 있다. MPI가 제공하는 집합통신 루틴은 다음과 같다.

| Category                               | Subroutines  |
|--|--|
| One buffer                             | MPI_BCAST  |
| One send buffer and one receive buffer | MPI_GATHER, MPI_SCATTER,<br>MPI_ALLGATHER, MPI_ALLTOALL,<br>MPI_GATHERV, MPI_SCATTERV,<br>MPI_ALLGATHERV, MPI_ALLTOALL |
| Reduction                              | MPI_REDUCE, MPI_ALLREDUCE, MPI_SCAN,<br>MPI_REDUCE_SCATTER   |
| Others                                 | MPI_BARRIER, MPI_OP_CREATE,<br>MPI_OP_FREE   |

표 2.5 MPI 집합통신 루틴

위의 표는 4 개 영역으로 구분한 16 개 집합통신 서브루틴 들을 나타낸다 . 이름이 굵게 표시된 것들은 자주 이용되는 루틴들을 나타내며 , MPI\_BCAST, MPI\_GATHER, MPI\_REDUCE 는 세 영역을 대표하는 루틴으로 특별히 자주 쓰인다 . 위의 16 개 MPI 집합통신 서브루틴들은 모두 블로킹이며 , 4 개 영역 중 1, 2, 3 세 개 영역에 속한 서브루틴들은 대응되는 논블로킹 서브루틴들이 존재한다 . 예를 들자면 MPI\_BCAST 의 논블로킹 루틴은 MPI\_IBCAST 이다 .

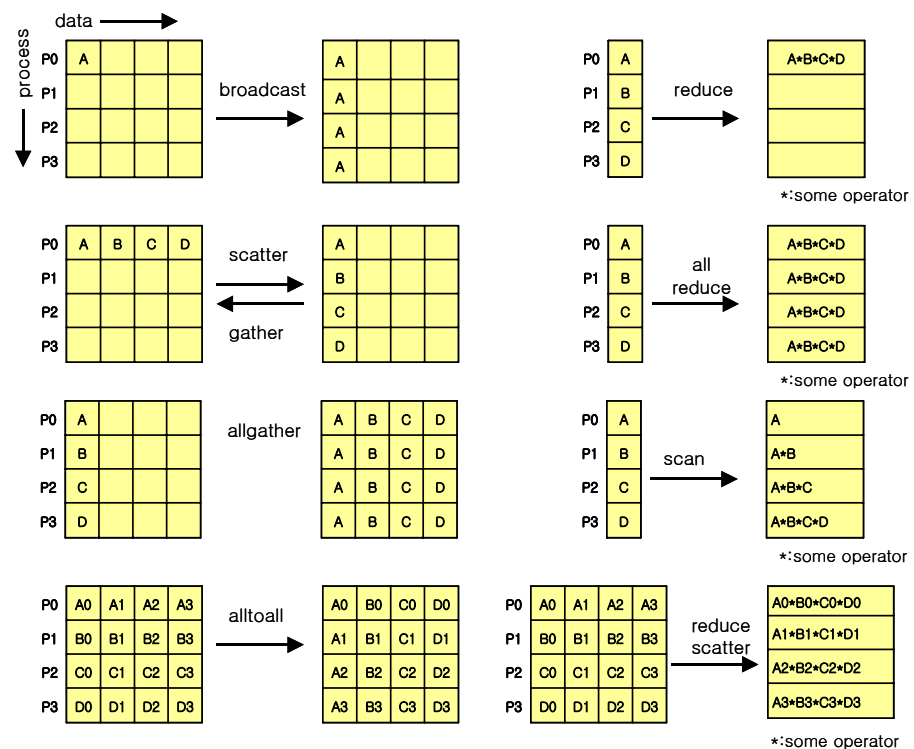


그림 2.13 집합 통신

### 2.3.1 방송 (Broadcast) : MPI\_BCAST

C:

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

Fortran:

```
MPI_BCAST(buffer, count, datatype, root, comm, ierr)
```

(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)

MPI\_BCAST 루틴은 루트 프로세스의 메모리에 있는 데이터를 동일 커뮤니케이터 내의 다른 모든 프로세스들의 메모리의 같은 위치로 복사하는 일대다 (one-to-all) 통신이다.

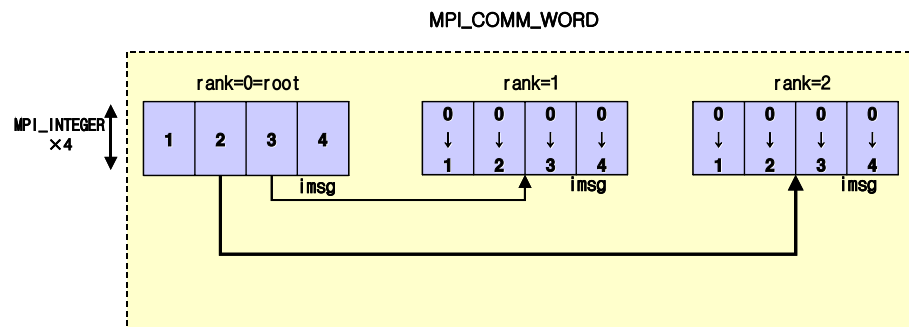


그림 2.14 방송

그림 2.14 는 rank=0 인 프로세스를 루트 프로세스로 하여 , 루트 프로세스에서 4 개의 정수를 MPI\_COMM\_WORLD내의 다른 프로세스에 방송하는 것을 나타낸 그림이다 . 그림 2.14 의 상황을 루틴 MPI\_Bcast 를 이용하여 프로그래밍한 것이 아래 예

제 2.13 과 2.14 이며 여기서 MPI\_Bcast 의 세 인수 (imsg, 4, MPI\_INTEGER) 는 버퍼의 주소 , 데이터의 개수 , 데이터의 데이터타입을 각각 나타낸다 .

예제 2.13 방송 : Fortran

```
PROGRAM bcast
```

```
INCLUDE 'mpif.h'
```

```
INTEGER imsg(4)
```

```
CALL MPI_INIT(ierr)
```

```
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
```

```
IF (myrank==0) THEN
```

```
    DO i=1,4
```

```
        imsg(i) = i
```

```
    ENDDO
```

```
ELSE
```

```
    DO i=1,4
```

```
        imsg(i) = 0
```

```
    ENDDO
```

```
ENDIF
```

```
PRINT*, 'Before:', imsg
```

```
CALL MPI_BCAST(imsg, 4, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
```

```
PRINT*, 'After :', imsg
```

```
CALL MPI_FINALIZE(ierr)
```

```
END
```

예제 2.14 방송 : 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_INTEGER, 0, MPI_COMM_WORLD);

    printf( "%d: AFTER:" , myrank);

    for(i=0; i<4; i++) printf( " %d" , imsg[i]); printf( "\n" );

    MPI_Finalize();

}
```

위의 예제에서 버퍼 `imsg` 는 루트 프로세스에서는 송신버퍼로 다른 프로세스에서는 수신버퍼로 사용되고 있음에 주의 하자 . 사용자가 수신버퍼를 다른 이름으로 하려면 위 예제를 다음과 같이 수정하면 된다 .

```
IF (myrank == 0) THEN
```

```
CALL MPI_BCAST(msg, 4, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
```

```
ELSE
```

```
CALL MPI_BCAST(jmsg, 4, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
```

```
ENDIF
```

이 경우에 프로세스 0 의 버퍼 msg 의 내용이 다른 프로세스들의 버퍼 jmsg 로 전송된다 . 이때 송신 프로세스와 수신 프로세스간에 주고 받는 데이터의 양은 반드시 일치해야 함에 주의하자 .

### 2.3.2 취합 (Gather)

**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)

MPI\_GATHER 루틴은 커뮤니케이터에 있는 모든 프로세스들로부터 하나의 수신 프로세스 ( 루트 프로세스 ) 로 데이터를 전송하는 다대일 (all-to-one) 통신이다 . MPI\_GATHER 루틴이 호출되면, 루트 프로세스를 포함한 모든 프로세스는 각자의 송신버퍼에 있는 데이터를 루트 프로세스로 보내고, 루트 프로세스는 데이터를 받아 랭크 순서대로 메모리에 저장한다 .

이 때 , 각 프로세스로부터 취합된 (gathered) 데이터의 크기는 모두 같아야 한다 .

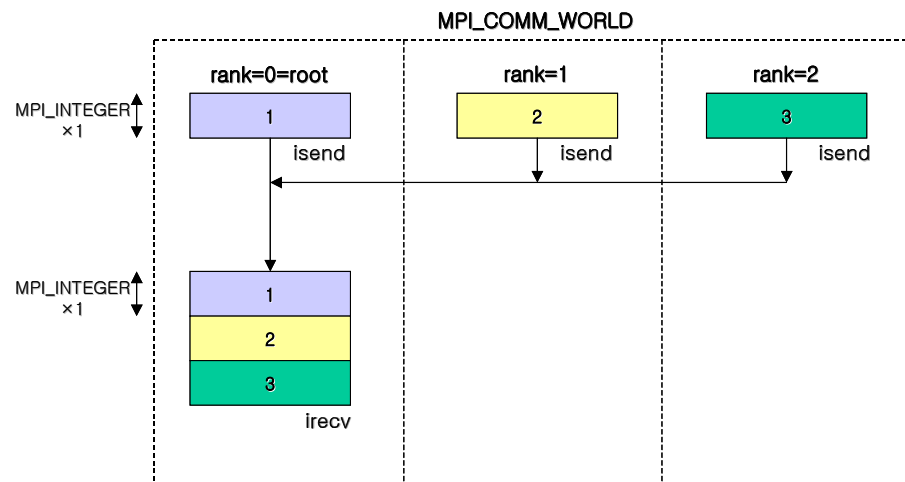


그림 2.15 MPI\_GATHER 를 이용한 취합

그림 2.15 는 프로세스 0, 1, 2 의 버퍼 isend 에 각각 저장된 정수 1, 2, 3 을 MPI\_GATHER 가 수신 프로세스 ( 프로세스 0 ) 로 취합하여 랭크가 증가하는 순서대로 정수 배열 irecv 에 저장하는 것을 보여주고 있다 . 이 상황을 프로그램한 것이 다음 예제 2.15 과 2.16 이다 .

MPI\_Gather 의 인수 (isend, 1, MPI\_INTEGER) 와 (irecv, 1, MPI\_INTEGER) 는 각각 송신과 수신버퍼의 주소, 데이터 개수, 데이터의 데이터 타입을 나타낸다 . 루트 프로세스가 각 프로세스들로부터 받게 되는 데이터 개수 ( 여기서는 1 ) 가 인수로 들

어 있는데, 이것은 루트 프로세스가 받게 되는 총 개수 ( 여기서는 3 개 ) 가 아니라  
는 점에 주의하여야 한다.

예제 2.15 MPI\_GATHER 를 이용한 취합 : 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
```

예제 2.16 MPI\_Gather 를 이용한 취합 : 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_INTEGER,irecv,1,MPI_INTEGER,0,
        MPI_COMM_WORLD);

if(myrank == 0) {

    printf( " irecv = " );

    for(i=0; i<3; i++) printf( " %d" , irecv[i]); printf( "\n" );

}

MPI_Finalize();

}

```

루트 프로세스가 자신에게 데이터를 보내기 때문에 송신버퍼 (isend) 와 수신버퍼 (irecv) 의 메모리 위치는 반드시 달라야 하는 점에 유의하자 .

참고로 , MPI-2 에서는 일부 집합 통신 루틴에서 송신 버퍼를 수신버퍼로 사용하는 것을 허용하고 있다 . 이런 경우 루트 프로세스에서 **MPI\_GATHER** 의 첫번째 인수를 **MPI\_IN\_PLACE** 로 지정하며 , 이때 sendcount 와 sendtype 은 루트 프로세스에서 무시되고 데이터를 취합하는 버퍼에 루트의 데이터는 미리 수신버퍼의 제자리를 차지하고 있게 된다 .

## **MPI\_GATHERV:**

**C:**

```
int MPI_Gatherv(void *sendbuf, int sendcount, MPI_Datatype sendtype,
void *recvbuf, int recvcunts, int displs, MPI_Datatype recvtype, int root,
MPI_Comm comm)
```

**Fortran:**

```
MPI_GATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcunts,
displs, 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 recvcunts(\*) : 수신된 원소의 개수를 저장하는 정수 배열 (IN)

INTEGER displs(\*) : 정수 배열, i 번째 자리에는 프로세스 i 에서 들어오는 데이터가 저장될 수신 버퍼 상의 위치를 나타냄 (IN) INTEGER recvtype : 수신 버퍼 원소의 MPI 데이터 타입 (IN)

INTEGER root : 수신 프로세스 ( 루트 프로세스 ) 의 랭크 (IN)

INTEGER comm : 커뮤니케이터 (IN)

MPI\_GATHER 는 크기가 같은 데이터를 각 프로세스로부터 취합한다 . 만약 커뮤니케이터내의 프로세스들로부터 크기가 서로 다른 데이터를 루트 프로세스로 모아야 한다면 루틴 MPI\_GATHERV 를 사용하면 된다 . MPI\_GATHERV 에 의해 취합되는 크기가 서로 다른 각 프로세스의 데이터는 그 크기가 배열 recvcunts 에 저장되고 , 루트 프로세스 수신버퍼에 저장되는 위치가 배열 displs 에 저장한다 . 즉 , 1 번 프로세스에서 취합되는 데이터의 크기는 recvcunts(1) 에 저장되며 , 그 데이터가 수신버퍼에서 차지하는 자리의 첫 주소가 displs(1) 에 저장되는 것이다 . 따라서 , 프로세스 i 의 sendcount, sendtype 은 루트 프로세스의 recvcunts(i), recvtype 과 반드시 일치해야 한다 .

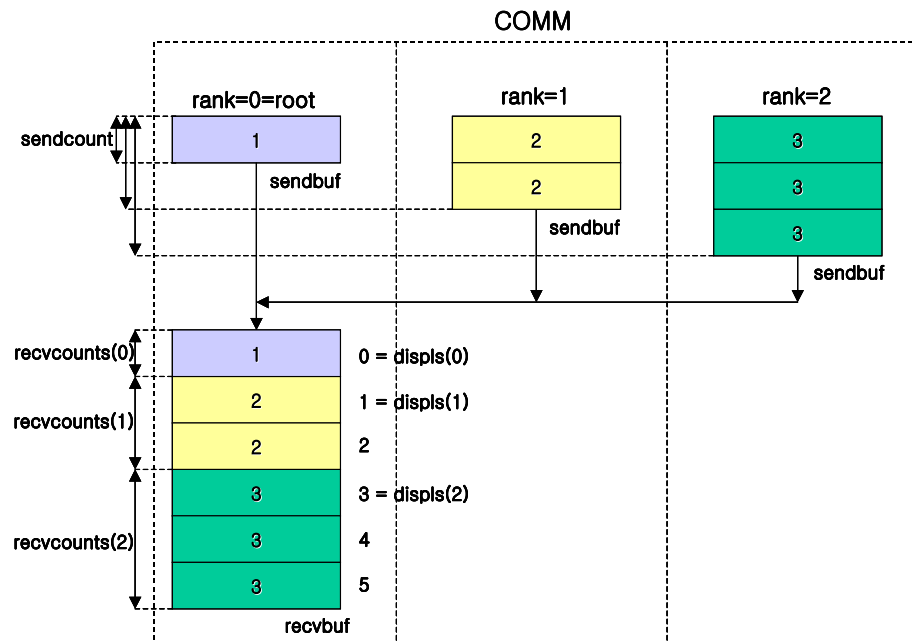


그림 2.16 MPI\_GATHERV 를 이용한 취합

그림 2.16 은 루틴 MPI\_GATHERV 를 이용해 0 번 프로세스에서 1 개 , 1 번 프로세스에서 2 개 , 그리고 2 번 프로세스에서 3 개의 데이터를 가져와 루트 (0 번) 프로세스의 수신 버퍼 (recvbuf) 에 저장하는 상황을 보여준다 . 각 프로세스에서 가져오는 데이터의 크기는 배열 recvcounts 에 저장되며 , 가져온 데이터가 수신버퍼에 저장되는 위치는 배열 displs 에 저장된다 .

예제 2.17 과 2.18 은 그림 2.16 의 상황을 프로그램한 것이다 .

예제 2.17 MPI\_GATHERV 를 이용한 취합 : 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

isnt = 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

```

예제 2.18 MPI\_GATHERV 를 이용한 취합 : C

```

/*gatherv*/

#include <mpi.h>

#include <stdio.h>

void main (int argc, char *argv[]){

    int i, myrank ;

    int isend[3], irecv[6];

```

```

int iscnt, irent[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_INTEGER, irecv, irent, idisp,
          MPI_INTEGER, 0, MPI_COMM_WORLD);

if(myrank == 0) {

    printf( " irecv = " );

    for(i=0; i<6; i++) printf( " %d" , irecv[i]);

    printf( "\n" );

}

MPI_Finalize();

}

```

MPI\_GATHER 와 마찬가지로 MPI\_SCATTER, MPI\_ALLGATHER, MPI\_ALLTOALL 도 크기가 다른 데이터 전송을 위한 MPI\_SCATTERV, MPI\_ALLGATHERV, MPI\_ALLTOALLV 루틴이 있다.

#### **MPI\_ALLGATHER:**

**C:**

```

int MPI_Allgather(void *sendbuf, int sendcount, MPI_Datatype sendtype,
void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)

```

**Fortran:**

**MPI\_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, 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 : 각 프로세스로부터 수신된 데이터 개수 (IN)

INTEGER recvtype : 수신버퍼 데이터 타입 (IN)

INTEGER comm : 커뮤니케이터 (IN)

MPI\_GATHER 를 이용해 프로세스 0 으로 데이터를 모으고 다음에 MPI\_BCAST 를 이용해 모은 데이터를 다른 모든 프로세스에 전달하는 경우를 생각할 수 있다. 이런 경우 MPI\_ALLGATHER 를 이용하면 편리하다. 각 프로세스로부터 모아지는 데이터의 크기는 같아야 하고 ( 만약 크기가 다르다면 MPI\_ALLGATHERV 를 사용할 수 있다.) 프로세스 j 에서 보내진 데이터 블록은 다른 모든 프로세스에서 수신 버퍼의 j 번째 블록에 수신하게 된다.

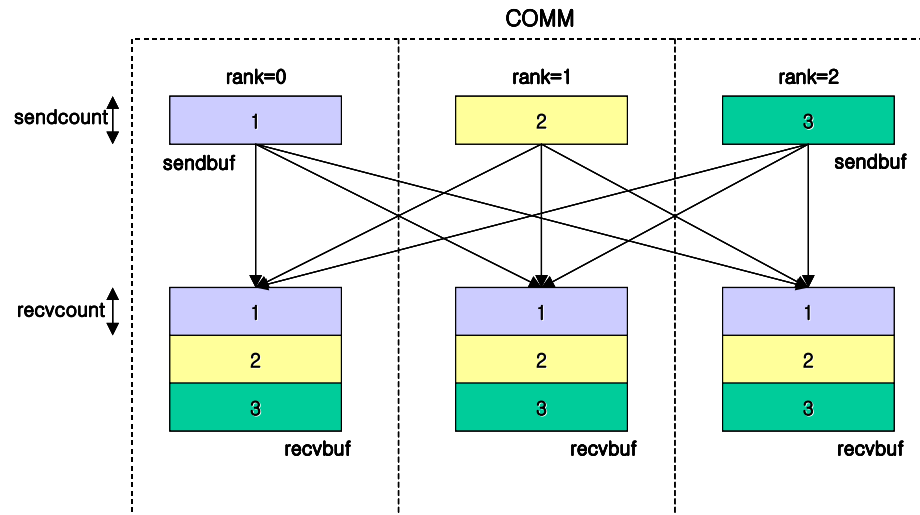


그림 2.17 MPI\_ALLGATHER 를 이용한 취합

그림 2.17 은 0, 1, 2 번 프로세스가 가진 하나씩의 데이터를 취합하여 0, 1, 2 번 프로세스의 수신 버퍼에 저장하는 MPI\_ALLGATHER 의 기능을 보여준다.  $i$  번째 프로세스의 데이터는 각 프로세스 수신 버퍼의  $i$  번째 블록에 저장되고 있음에 주의하자.

다음 예제 2.19 과 2.20 은 그림 2.17 에 대한 프로그램이다.

예제 2.19 MPI\_ALLGATHER 를 이용한 취합 : 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

```

예제 2.20 MPI\_Allgather 를 이용한 취합 : 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_INTEGER, irecv, 1,
                MPI_INTEGER, MPI_COMM_WORLD);

    printf( " %d irecv = " );

    for(i=0; i<3; i++) printf( " %d" , irecv[i]);

    printf( "\n" );

    MPI_Finalize();

```



```
}

```

## MPI\_ALLGATHERV :

C:

```
int MPI_Allgather(void *sendbuf, int sendcount, MPI_Datatype sendtype,
void *recvbuf, int recvcounts, int displs, MPI_Datatype recvtype,
MPI_Comm comm)
```

Fortran:

```
MPI_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
displs, recvtype, comm, ierr)
```

기본적으로 MPI\_ALLGATHER 와 동일하며 크기가 다른 데이터를 모은다는 면에서 차이가 있다. 커뮤니케이터내의 각 프로세스로부터 크기가 다른 데이터를 모아서 만든 메시지를 모든 프로세스에 전달한다. MPI\_ALLGATHERV 는 아래 그림과 같은 데이터 취합에 이용된다.

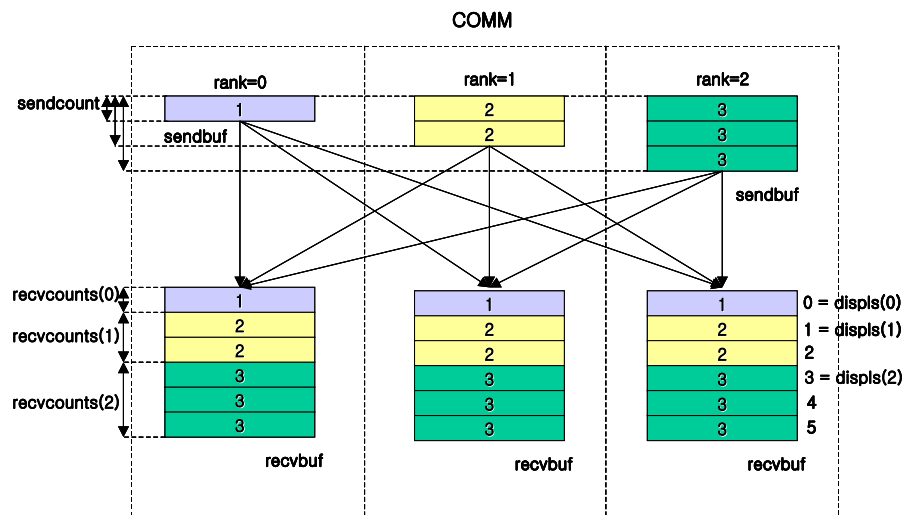


그림 2.18 MPI\_ALLGATHERV 를 이용한 취합

### 2.3.3 환산 (Reduce)

**MPI\_REDUCE :**

**C:**

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

**Fortran:**

```
MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm, ierr)
```

(CHOICE) sendbuf : 송신 버퍼의 시작 주소 (IN)

(CHOICE) recvbuf : 수신 버퍼의 주소 (OUT)

INTEGER count : 송신 버퍼의 원소 개수 (IN)

INTEGER datatype : 송신 버퍼 원소의 MPI 데이터 타입 (IN)

INTEGER op : 환산 연산자 (IN)

INTEGER root : 루트 프로세스의 랭크 (IN)

INTEGER comm : 커뮤니케이터 (IN)

MPI\_REDUCE 루틴은 각 프로세스로부터 데이터를 모아 하나의 값으로 환산하여 (reduce) 그 환산된 결과를 루트 프로세스에 저장한다 .

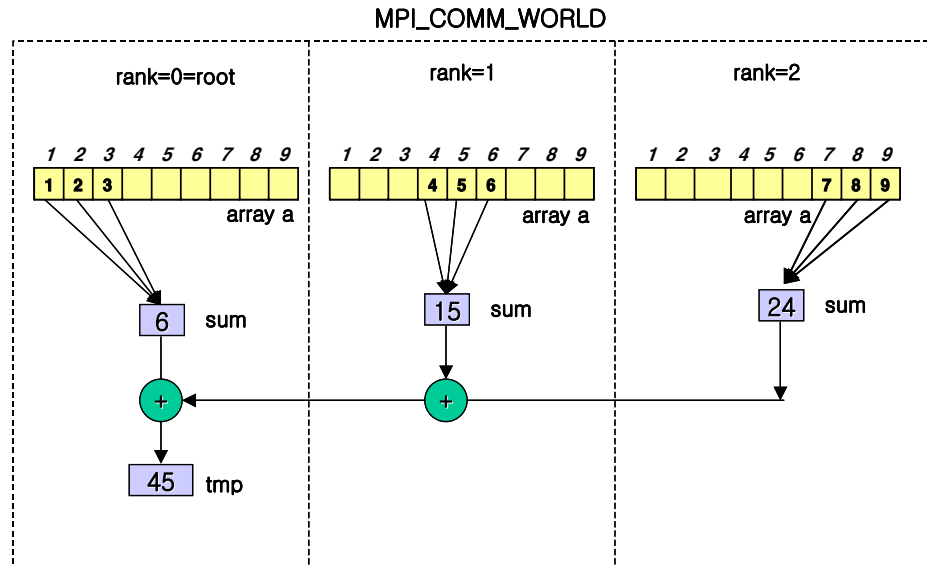


그림 2.19 MPI\_REDUCE 를 이용한 환산

그림 2.19 는 실수 배열  $a(i)$  ( $i=1\cdots 9$ ) 의 각 원소의 합을 세 개의 프로세스를 이용해 병렬로 구하는 과정이다. 루틴 MPI\_REDUCE 를 이용하면 세 개의 프로세스가 각각  $a(i)$  의 1/3 만큼을 담당해 부분합을 sum 에 저장하고 세 개의 부분합들이 다시 더해져 그 결과가 루트 프로세스 ( 프로세스 0) 에 보내진다.

다음 예제 2.21 과 2.22 는 그림 2.19 에 대한 프로그램으로, MPI\_GATHER 와 마찬가지로 송신버퍼와 수신버퍼가 같은 메모리 번지를 가질 수 없어 또 다른 변수 tmp 를 이용해 전체 합을 저장하고 있음을 볼 수 있다.

예제 2.21 MPI\_REDUCE 를 이용한 환산 : 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

```

예제 2.22 MPI\_REDUCE 를 이용한 환산 : 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();

}

```

루틴 `MPI_REDUCE` 의 다섯번째 인수 `MPI_SUM` 은 환산 연산자로서 `MPI` 에서 기본으로 제공하고 있는 몇 가지 연산자들 중 한가지 이다 . `MPI` 에서 기본으로 제공하는 연산자는 헤더파일 `mpif.h`(`mpi.h`) 에 정의 되어 있으며 사용자는 `MPI_OP_CREATE` 을 이용해 자신이 원하는 연산을 정의하여 환산 연산자로 사용할 수 도 있다 .

`MPI` 에서 기본으로 제공하는 연산자와 그 데이터 타입은 다음과 같다 .

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

표 2.6 MPI 환산 연산자와 데이터 타입 : Fortran

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

표 2.7 MPI 환산 연산자와 데이터 타입 : C

위의 표에서 MPI\_MAXLOC 과 MPI\_MINLOC 에 이용되는 데이터 타입들은 특별히 C 구조에만 있는 것들이다. 다음 표에 정리해 두었다.

| Data Type           | Description (C)              |
|---------------------|------------------------------|
| MPL_FLOAT_INT       | { MPL_FLOAT, MPL_INT }       |
| MPL_DOUBLE_INT      | { MPL_DOUBLE, MPL_INT }      |
| MPL_LONG_INT        | { MPL_LONG, MPL_INT }        |
| MPL_2INT            | { MPL_INT, MPL_INT }         |
| MPL_SHORT_INT       | { MPL_SHORT, MPL_INT }       |
| MPL_LONG_DOUBLE_INT | { MPL_LONG_DOUBLE, MPL_INT } |

표 2.8 환산 연산자  $MPI\_MAXLOC$ ,  $MPI\_MINLOC$  에 사용된 데이터 타입 : C

앞의 예제에서는 각 프로세스에서 계산된 스칼라 값 sum 을  $MPI\_REDUCE$  를 이용하여 환산한 결과를 리턴하는 것을 보여 주었다. 만약 스칼라 값이 아닌 배열에 대해  $MPI\_REDUCE$  연산을 행하면 다음 그림 2.20 과 같이 계산 되어진다.

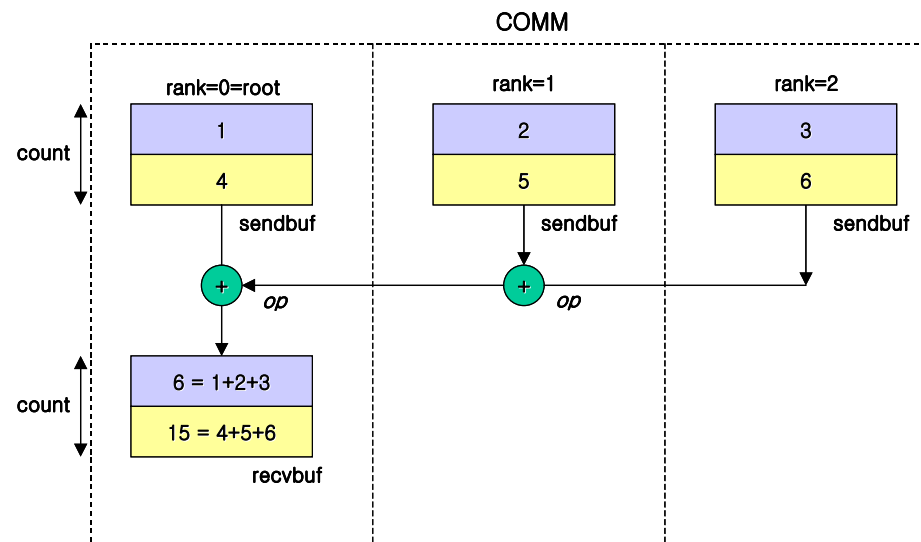


그림 2.20 배열에 적용된 환산 연산  $MPI\_REDUCE$

MPI\_REDUCE 를 사용할 때 사용자는 MPI\_REDUCE 에 의해 생성되는 라운딩 (rounding) 에러를 염두에 두어야 한다 . 정해진 정밀도를 가지는 실수 연산에서 일반적으로  $(a+b)+c \neq a+(b+c)$  이다 . 따라서 실수 배열  $a()$  의 합을 구하는 앞선 예제에서는 부분합을 먼저 구하기 때문에 순차프로그램을 이용해 얻어지는 결과와 조금 다를 수도 있다 . 게다가 , 부분합이 더해져 전체합이 구해지는 순서에 따라서도 결과가 조금 달라질 수 있다는 것을 밝혀둔다 .

순차계산 :

$$a(1) + a(2) + a(3) + a(4) + a(5) + a(6) + a(7) + a(8) + a(9)$$

병렬계산 :

$$[a(1) + a(2) + a(3)] + [a(4) + a(5) + a(6)] + [a(7) + a(8) + a(9)]$$

**MPI\_ALLREDUCE:**

**C:**

```
int MPI_Allreduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype
datatype, MPI_Op op, MPI_Comm comm)
```

**Fortran:**

```
MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm, ierr)
```

MPI\_REDUCE 는 환산 결과를 루트 프로세스에만 저장하지만 MPI\_ALLREDUCE 는 다음 그림과 같이 환산된 결과를 커뮤니케이터내 모든 프로세스의 수신버퍼로 보낸다 .



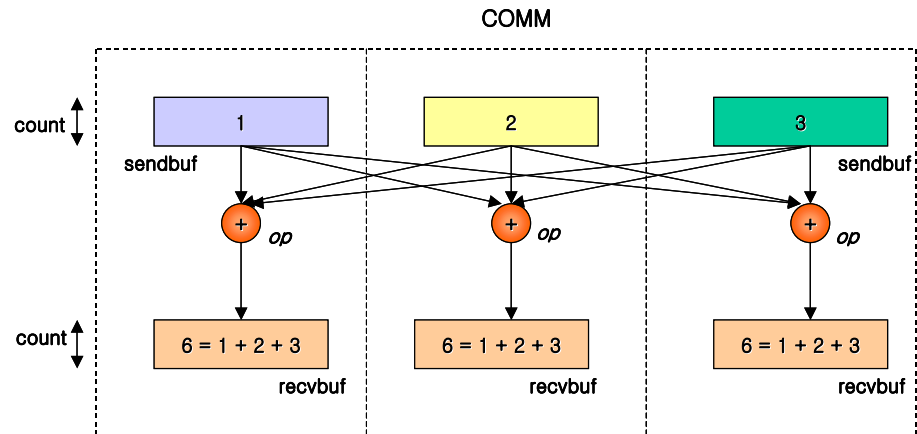


그림 2.21 MPI\_ALLREDUCE

### 2.3.4 확산 (scatter)

**MPI\_SCATTER :**

**C:**

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

**Fortran:**

```
MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount,
recvtype, root, comm, ierr)
```

(CHOICE) sendbuf : 송신 버퍼의 주소 (IN)

INTEGER sendcount : 각 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)

MPI\_SCATTER 루틴은 일대다 (one-to-all) 통신으로, MPI\_SCATTER 가 호출되면 루트 프로세스는 데이터를 같은 크기로 나누어 동일 커뮤니케이터내의 각 프로세스에 랭크 순서대로 하나씩 전달한다. 처음 sendcount 개의 데이터가 프로세스 0 으 로 다음 sendcount 개 데이터가 프로세스 1 로 ... 전달되는 식이다. 이 결과는 루트 프로세스가 N 개의 MPI\_SEND를 실행하고 나머지 프로세스가 각각 MPI\_RECV를 실행시킨 것과 동일하다.

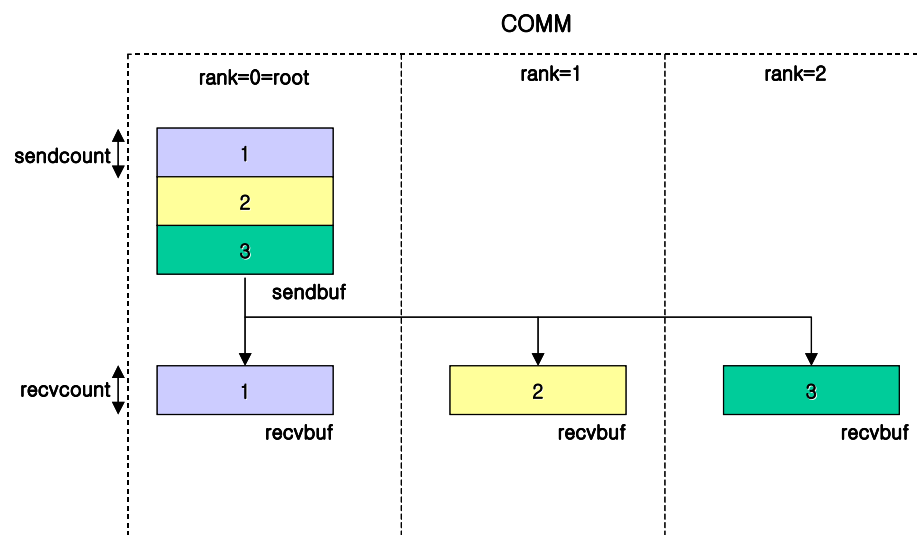


그림 2.22 MPI\_SCATTER

다음 예제 2.23 과 예제 2.24 는 MPI\_SCATTER 연산의 역할을 보여주는 그림 2.22 를 프로그램한 것이다.

예제 2.23 MPI\_SCATTER 연산 : Fortran

```

PROGRAM scatter

INCLUDE 'mpif.h'

INTEGER isend(3)

CALL MPI_INIT(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

```

예제 2.24 MPI\_SCATTER 연산 : C

```

/*scatter*/

#include <mpi.h>

#include <stdio.h>

void main (int argc, char *argv[]){

    int i, myrank ;

    int isend[3], irecv;

```

```

MPI_Init(&argc, &argv);

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

for(i=0; i<nprocs; i++) isend[i]=i+1;

MPI_Scatter(isend, 1, MPI_INTEGER, &irecv, 1,
             MPI_INTEGER, 0, MPI_COMM_WORLD);

printf( " %d: irecv = %d\n" , myrank, irecv);

MPI_Finalize();

}

```

#### **MPI\_SCATTERV :**

**C:**

```

int MPI_Scatter(void *sendbuf, int sendcounts, int displs, MPI_Datatype
sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
MPI_Comm comm)

```

**Fortran:**

```

MPI_SCATTER(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
recvtype, root, comm, ierr)

```

같은 크기의 데이터 만을 보내는 MPI\_SCATTER 와 달리 MPI\_SCATTERV 를 이용  
하면 서로 다른 크기로 나눈 데이터를 루트 프로세스로부터 각 프로세스에 보낼  
수 있다 .

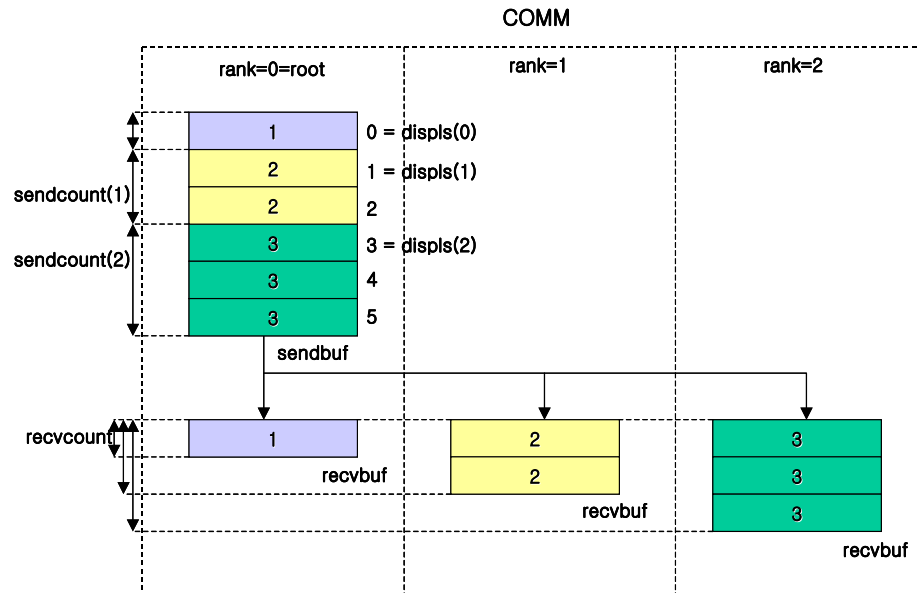


그림 2.23 MPI\_SCATTERV

### 2.3.5 기타

**MPI\_BARRIER:**

**C:**

`int MPI_Barrier(MPI_Comm comm)`

**Fortran:**

`MPI_BARRIER(comm, ierr)`

MPI\_BARRIER 는 커뮤니케이터내의 모든 프로세스들이 같이 다음 실행으로 넘어가고자 할 때 사용한다. 모든 프로세스가 MPI\_BARRIER 루틴을 호출할 때 까지 커뮤니케이터내 프로세스들의 진행을 막는다.

**MPI\_ALLTOALL:**

**C:**

```
int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype, void
*recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
```

**Fortran:**

```
MPI_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount,
recvtype, comm, ierr)
```

각 프로세스로부터 모든 프로세스에 동일한 크기의 개별적인 메시지를 전달한다. 프로세스  $i$ 로부터 송신되는  $j$  번째 데이터 블록은 프로세스  $j$ 가 받아 수신버퍼의  $i$  번째 블록에 저장하게 된다.

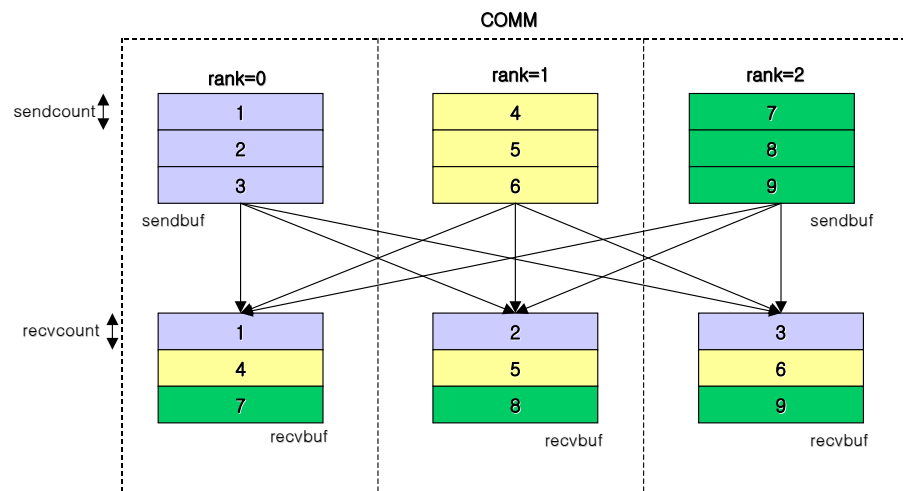


그림 2.24 MPI\_ALLTOALL

**MPI\_ALLTOALLV:**

**C:**

```
int MPI_Alltoallv(void *sendbuf, int sendcounts, int sdispls, MPI_Datatype
sendtype, void *recvbuf, int recvcounts, int rdispls, MPI_Datatype
recvtype, MPI_Comm comm)
```

Fortran:

```
MPI_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
recvcounts, rdispls, recvtype, comm, ierr)
```

MPI\_ALLTOALL 과 동일한 기능이며 크기가 다른 메시지를 처리할 수 있다. 프로세스  $i$  로부터 송신되는  $sendcounts(j)$  개의 데이터는 프로세스  $j$  가 받아 수신버퍼의  $rdispls(i)$  번째 위치부터 저장된다.

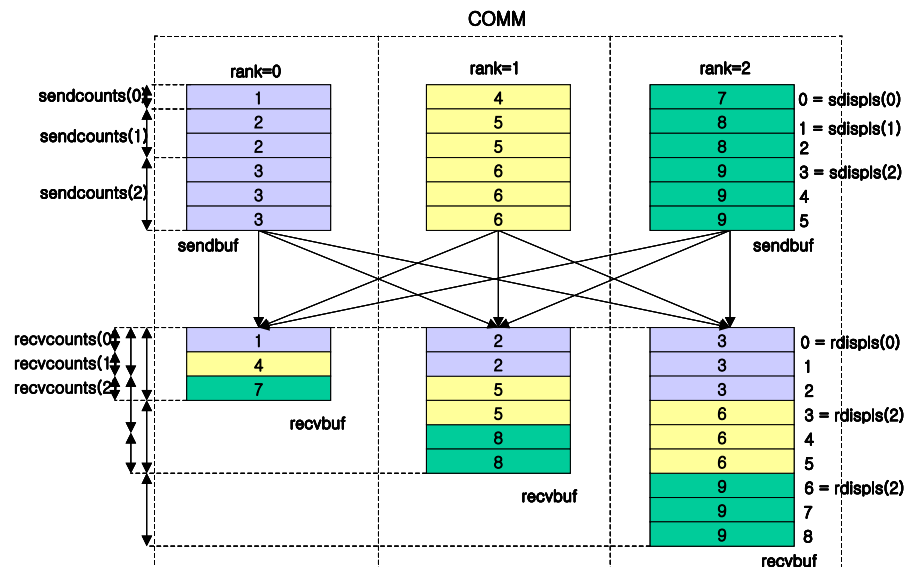


그림 2.25 MPI\_ALLTOALLV

MPI\_REDUCE\_SCATTER:

C:

```
int MPI_Reduce_scatter(void *sendbuf, void *recvbuf, int recvcounts,
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

Fortran:

**MPI\_REDUCE\_SCATTER(sendbuf, recvbuf, recvcounts, datatype, op, comm, ierr)**

커뮤니케이터내 프로세스들의 송신 버퍼 데이터들을 모아 환산 연산을 수행하고 그 결과를 recvcounts에 의해 지정된 값들에 따라 확산 연산을 수행한다. 루트 프로세스는 환산 연산이 수행된 결과를 차례대로 recvcounts(i) 개씩 모아서 프로세스 i로 송신 한다. MPI\_REDUCE\_SCATTER의 실행결과는 MPI\_REDUCE 루틴을 실행한 후 recvcounts(i)와 동일한 sendcounts(i)를 이용해 MPI\_SCATTERV 루틴을 실행한 것과 동일하다.

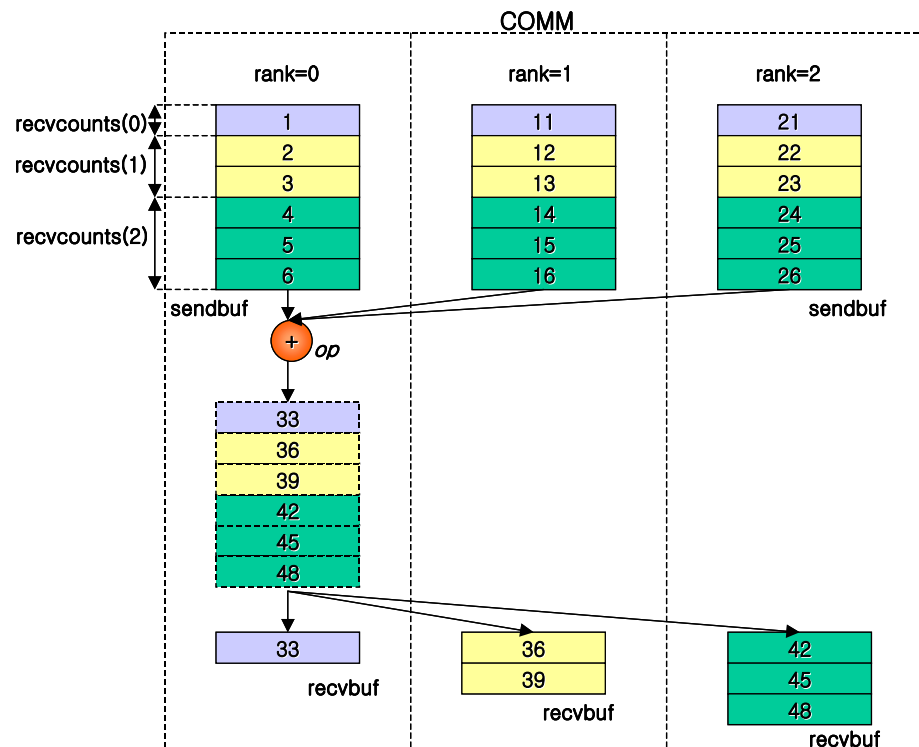


그림 2.26 MPI\_REDUCE\_SCATTER



**MPI\_SCAN:**

**C:**

```
int MPI_Scan(void *sendbuf, void *recvbuf, int count, MPI_Datatype
datatype, MPI_Op op, MPI_Comm comm)
```

**Fortran:**

```
MPI_SCAN(sendbuf, recvbuf, count, datatype, op, comm, ierr)
```

MPI\_SCAN 루틴은 프로세스 i의 수신버퍼에 프로세스 0에서 프로세스 i까지의 수신버퍼 데이터들에 대한 reduction 값을 저장한다.

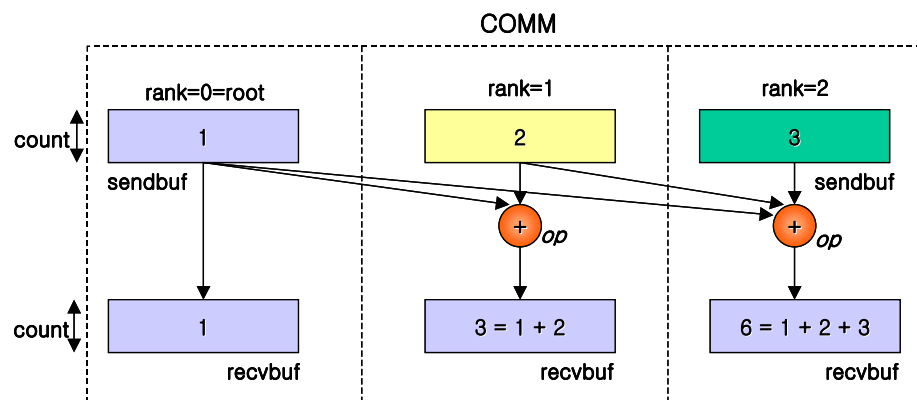


그림 2.27 MPI\_SCAN

## 2.4 유도 데이터 타입 (Derived Data Type)

다른 프로세스로 전송하고자 하는 데이터가 동일한 데이터 타입으로 연속적으로 구성되어 있다면 지금까지 언급된 통신 방법들을 이용해 메시지 송 / 수신을 성공적으로 수행할 수 있다. 그러나, 실질적으로 송 / 수신해야 할 많은 데이터들은 불연속적으로 구성되는 경우가 많으며, 또한 여러 가지 데이터 타입으로 구성된 데이터 묶음을 송 / 수신해야 하는 경우를 생각해 볼 수 있다. 이러한 경우 데이터 양이 그다지 많지 않다면 데이터 하나하나를 각각 송신하고 수신하는 방법을 생각해

볼 수 있지만 아마도, 어렵고 비효율적인 작업이 될 것이다. 또, 사용자는 불연속적인 데이터를 연속적인 버퍼에 우선 복사해 두고 한번에 그것을 전송해 버릴 수도 있지만 수신 프로세스는 그 데이터를 풀어 다시 적절한 위치 ( 원래의 불연속적인 위치 ) 에 데이터를 저장시켜야 하는 어려운 문제가 발생된다.

보다 일반적이고, 불연속적인 데이터를 다루기 위해 **MPI** 는 사용자가 직접 데이터 타입을 만들어 사용하는 유도 데이터 타입 기능을 제공한다. 유도 데이터 타입은 동일한 데이터 타입을 가지는 불연속 데이터, 서로 다른 데이터 타입을 가지는 연속 데이터, 서로 다른 데이터 타입을 가지는 불연속 데이터 등의 통신에 매우 편리하게 사용될 수 있다.

### 2.4.1 기본적인 사용법

배열  $a()$  의 원소 중 불연속적으로 분포된  $a(4)$ ,  $a(5)$ ,  $a(7)$ ,  $a(8)$ ,  $a(10)$  그리고  $a(11)$  을 다른 프로세스로 송신해야 하는 경우를 생각해 보자. 사용자가 아래 그림의  $itype1$  과 같이 유도 데이터 타입을 정의한다면, 사용자는  $a(4)$  에서 시작하는 하나의 데이터를 송신하면 된다. 만약,  $itype2$  와 같이 유도 데이터 타입을 정의한다면 사용자는  $a(4)$  에서 시작되는 데이터 세 개를 전송해야 할 것이다.

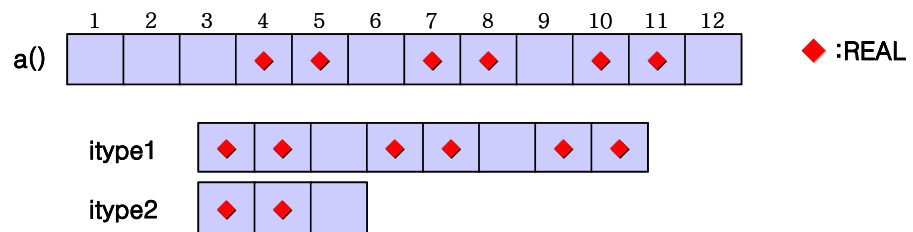


그림 2.28 유도 데이터 타입

위와 같이 데이터 타입을 정의 하고, 프로세스  $idist$  로 데이터를 송신한다면 실제 통신 코드는 다음처럼 구성하면 된다.

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

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

유도 데이터 타입은 프로그램 실행 중에 생성되는데 , MPI 통신에서 유도 데이터 타입을 사용하기 전에 프로그램은 다음과 같은 두 과정을 거치면서 새로운 데이터 타입을 정의하고 생성시키게 된다 .

**데이터 타입 구성 (Construct) :** MPI 기본 데이터 타입 또는 다른 유도 데이터 타입을 이용해 새로운 데이터 타입을 구성한다 . 이 때 다음과 같은 MPI 루틴을 이용한다 . MPI\_TYPE\_CONTIGUOUS, MPI\_TYPE\_VECTOR, MPI\_TYPE\_HVECTOR, MPI\_TYPE\_STRUCTOR

**데이터 타입 등록 (Commit) :** 루틴 MPI\_TYPE\_COMMIT 를 호출해 새로운 데이터 타입을 등록 한다 .

새로이 생성된 데이터 타입은 프로그램내의 모든 통신들에서 이용할 수 있고 , 더 이상 이용할 필요가 없어지면 루틴 MPI\_TYPE\_FREE 를 이용해 정의된 데이터 타입을 해지할 수 있다 .

## 2.4.2 데이터 타입 구성

**MPI\_TYPE\_CONTIGUOUS:**

**C:**

**int MPI\_Type\_contiguous (int count, MPI\_Datatype oldtype, MPI\_Datatype \*newtype)**

**Fortran:**

**MPI\_TYPE\_CONTIGUOUS (count, oldtype, newtype, ierr)**

INTEGER count : 하나로 묶을 데이터 개수 (IN)

INTEGER oldtype : 이전 데이터 타입 ( 핸들 ) (IN)

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

MPI\_TYPE\_CONTIGUOUS 는 같은 데이터 타입 (oldtype) 을 가지는 연속적인 데이터를 count 개 묶어서 하나의 새로운 데이터 타입 (newtype) 을 구성한다 .

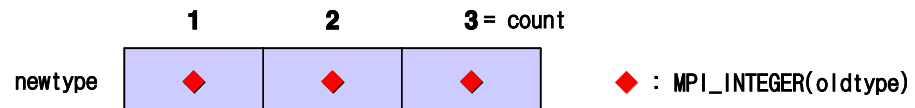


그림 2.29 MPI\_TYPE\_CONTIGUOUS

예제 2.25 MPI\_TYPE\_CONTIGUOUS : Fortran

```
PROGRAM type_contiguous

INCLUDE 'mpif.h'

INTEGER ibuf(20)

INTEGER newtype

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, newtype, ierr)

CALL MPI_TYPE_COMMIT(newtype, ierr)

CALL MPI_BCAST(ibuf, 3, newtype, 0, MPI_COMM_WORLD, ierr)

PRINT *, '  ibuf = ', ibuf
```

```
CALL MPI_FINALIZE(ierr)
```

```
END
```

예제 2.26 MPI\_TYPE\_CONTIGUOUS : C

```
/*type_contiguous*/  
  
#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_contiguous(3, MPI_INT, &newtype);  
  
    MPI_Type_commit(&newtype);  
  
    MPI_Bcast(ibuf, 3, newtype, 0, MPI_COMM_WORLD);  
  
    printf( " %d : ibuf=" , myrank);  
  
    for(i=0; i<20; i++) printf( " %d" , ibuf[i]);  
  
    printf( "\n" );  
  
    MPI_Finalize();  
  
}
```

## **MPI\_TYPE\_VECTOR:**

**C:**

**int MPI\_Type\_vector (int count, int blocklength, int stride, MPI\_Datatype oldtype, MPI\_Datatype \*newtype)**

**Fortran:**

**MPI\_TYPE\_VECTOR (count, blocklength, stride, oldtype, newtype, ierr)**

INTEGER count : 블록의 개수 (IN)

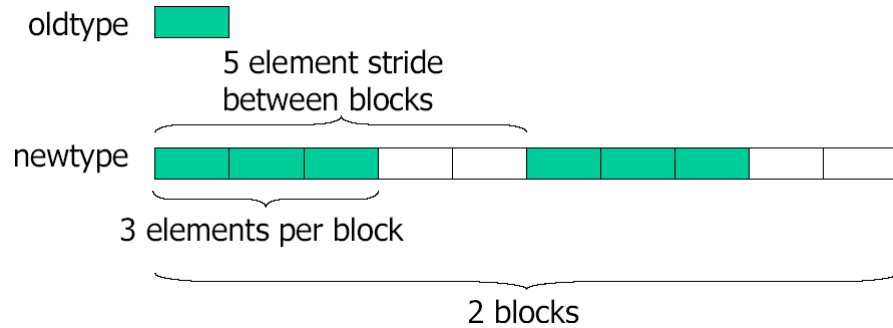
INTEGER blocklength : 각 블록의 oldtype 데이터의 개수 (IN)

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

INTEGER oldtype : 이전 데이터 타입 ( 핸들 ) (IN)

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

이 루틴은 똑 같은 간격만큼 떨어져 있는 count 개 블록들로 구성되는 새로운 데이터 타입을 구성하며 , 각 블록은 oldtype 의 blocklength 개 데이터의 연속으로 이루어진다. 한 블록의 시작점에서 다음 블록의 시작점 사이는 stride만큼 떨어져 있다.



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

그림 2.30 루틴 `MPI_Type_vector` 의 인수

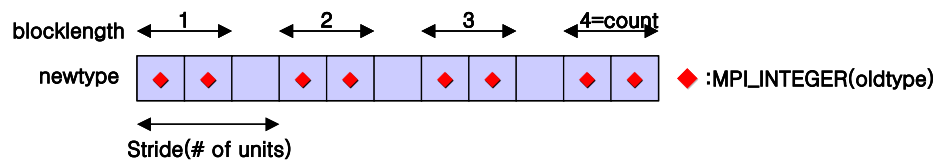


그림 2.31 `MPI_Type_vector`

예제 2.27 `MPI_Type_vector` : Fortran

```
PROGRAM type_vector
  INCLUDE 'mpif.h'
  INTEGER ibuf(20)
  INTEGER newtype
  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, newtype, ierr)

CALL MPI_TYPE_COMMIT(newtype, ierr)

CALL MPI_BCAST(ibuf, 1, newtype, 0, MPI_COMM_WORLD, ierr)

PRINT *, 'ibuf=' , ibuf

CALL MPI_FINALIZE(ierr)

END

```

예제 2.28 MPI\_TYPE\_VECTOR : 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_INTEGER, &inewtype);

MPI_Type_commit(&inewtype);

MPI_Bcast(ibuf, 1, inewtype, 0, MPI_COMM_WORLD);

printf( “%d : ibuf =” , myrank);

for(i=0; i<20; i++) printf( “ %d” , ibuf[i]);

printf( “\n” );

MPI_Finalize();

}

```

#### **MPI\_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)**

INTEGER count : 블록의 개수 (IN)

INTEGER blocklength : 각 블록의 oldtype 데이터의 개수 (IN)

INTEGER stride : 인접한 두 블록사이의 폭을 byte 로 나타낸 정수 (IN)

INTEGER oldtype : 이전 데이터 타입 ( 핸들 ) (IN)

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

MPI\_TYPE\_VECTOR 와 마찬가지로 똑 같은 간격만큼 떨어져 있는 count 개 블록들로 구성되는 새로운 데이터 타입을 정의하고, 각 블록은 oldtype 의 blocklength 개 데이터의 연속으로 이루어 진다. MPI\_TYPE\_VECTOR 와 다른 점은 한 블록의 시작점에서 다음 블록의 시작점 사이가 stride 만큼 떨어져 있는데, stride 의 단위가 bytes(stride = # of bytes) 로 주어져야 한다는 것이다.

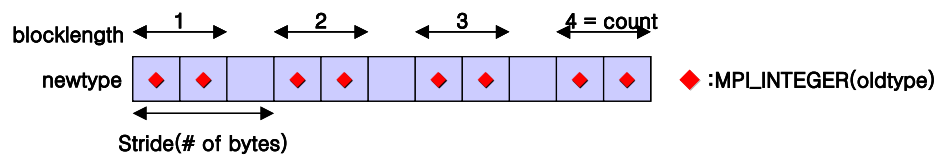


그림 2.32 MPI\_TYPE\_HVECTOR

#### MPI\_TYPE\_STRUCT:

C:

```
int MPI_Type_struct (int count, int *array_of_blocklengths, MPI_Aint
*array_of_displacements, MPI_Datatype *array_of_type, MPI_Datatype
*newtype)
```

Fortran:

```
MPI_TYPE_STRUCT (count, array_of_blocklengths,
array_of_displacements, array_of_types, newtype, ierr)
```

INTEGER count : 블록의 개수, 동시에 배열 array\_of\_blocklengths, array\_of\_displacements, array\_of\_types 의 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 i 번째 블록의 데이터 개수 (IN)

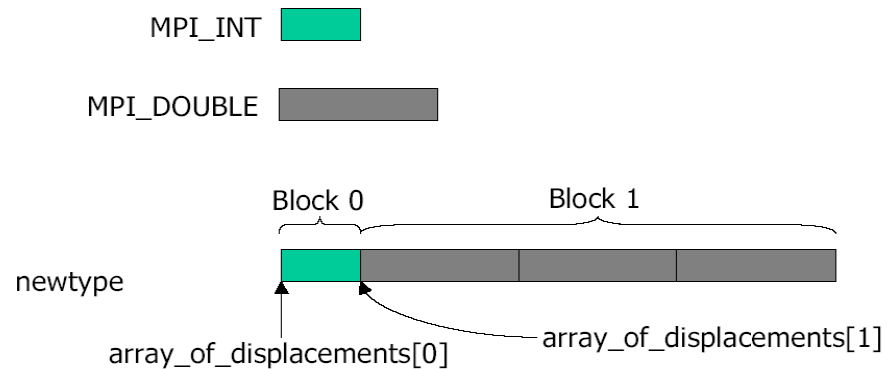
INTEGER array\_of\_displacements(\*) : 바이트로 나타낸 각 블록의 위치 (IN)

INTEGER array\_of\_types(\*) : 각 블록을 구성하는 데이터 타입, i 번째 블록은 데이터 타입이 array\_of\_types(i) 인 데이터로 구성 (IN)

INTEGER newtype : 새로운 데이터 타입 (OUT)

MPI\_TYPE\_STRUCT 는 가장 일반적인 데이터 타입 구성자 (constructor) 로 , 서로 다른 데이터 타입을 가지는 불연속적인 데이터로 구성된 새로운 데이터 타입을 구성할 수 있다 . MPI\_TYPE\_STRUCT 에 의해 생성된 count 개의 블록으로 이루어진 새로운 데이터 타입에서 i 번째 블록은 데이터 타입이 array\_of\_types(i) 이고 길이가 array\_of\_blocklengths(i) 이다 . 또 i 번째 블록의 위치는 bytes 단위로 array\_of\_displacements(i) 로 주어진다 .

MPI\_TYPE\_STRUCT 는 또한 MPI 수도 (pseudo) 타입인 MPI\_LB 와 MPI\_UB 를 사용할 수 있는 유일한 구성자이다 . 참고로 MPI\_LB 와 MPI\_UB 는 차지하는 공간이 없으며 , 데이터 타입의 시작이나 끝에서 빈공간이 나타나도록 해야 할 때 이용한다 .



- **count = 2**
- **array\_of\_blocklengths = { 1, 3}**
- **array\_of\_types = {MPI\_INT, MPI\_DOUBLE}**
- **array\_of\_displacements = {0, extent(MPI\_INT)}**

그림 2.33 루틴 MPI\_TYPE\_STRUCT 의 인수

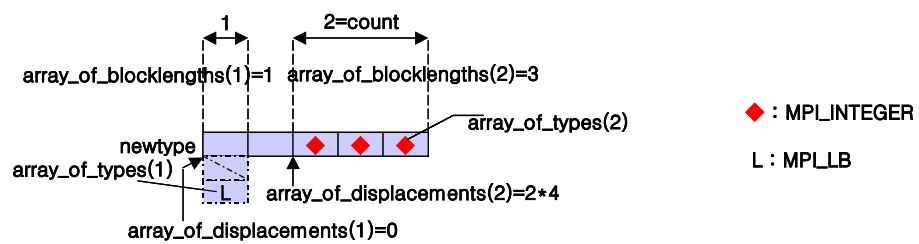
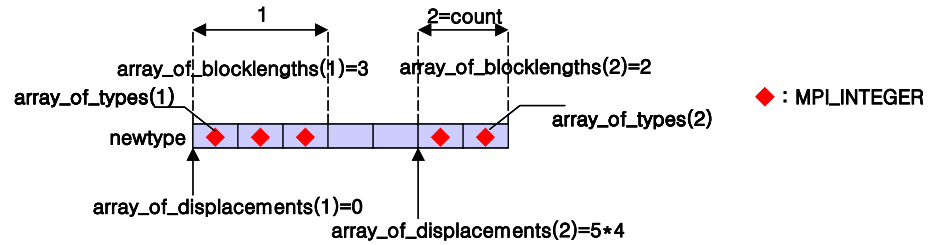


그림 2.34 MPI\_TYPE\_STRUCT

예제 2.29 MPI\_TYPE\_STRUCT : Fortran

```
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)

    CALL MPI_BCAST(ibuf1, 2, newtype1, 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, newtype2, ierr)

    CALL MPI_TYPE_COMMIT(newtype2, ierr)

    CALL MPI_BCAST(ibuf2, 2, newtype2, 0, MPI_COMM_WORLD, ierr)

    PRINT *, '  Ex. 2: ', ibuf2

    CALL MPI_FINALIZE(ierr)

    END

```

예제 2.30 MPI\_TYPE\_STRUCT : C

```

/*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 anewtype1, anewtype2;

    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;

        }

    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_INTEGER; itype[1] = MPI_INTEGER;

    MPI_Type_struct(2, iblock, idisp, itype, &anewtype1);

    MPI_Type_commit(&anewtype1);

    MPI_Bcast(ibuf1, 2, anewtype1, 0, MPI_COMM_WORLD);

```

```

printf( "%d : Ex.1 :", myrank);

for(i=0; i<20; i++) printf( " %d" , ibuf1[i]);

printf( "\n" );

iblock[0] = 1; iblock[1] = 3;

idisp[0] = 0; idisp[1] = 2*4;

itype[0] = MPI_LB;

itype[1] = MPI_INTEGER;

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:**

**int MPI\_Type\_extent(MPI\_Datatype \*datatype, MPI\_Aint \*extent)**

**Fortran:**

**MPI\_TYPE\_EXTENT (datatype, extent, ierr)**

INTEGER datatype : 데이터 타입 ( 핸들 ) (INOUT)

INTEGER extent : 데이터 타입의 범위 (OUT)

데이터 타입의 범위 (extent) 를 리턴한다 . 데이터 타입의 범위는 데이터 타입 내의 원소들이 차지하고 있는 byte 수를 의미한다 .

예 제 2.31 MPI\_TYPE\_EXTENT : Fortran

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,reallex**

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,reallex,err)**

displacements(1)=0; displacements(2)=intex

displacements(3)=intex+reallex

**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)=cmplx(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

```

예제 2.32 MPI\_TYPE\_EXTENT : C

```

#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);

displacements[0]= (MPI_Aint)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();
}

```

### 2.4.3 데이터 타입 등록

**MPI\_TYPE\_COMMIT:**

**C:**

```
int MPI_Type_commit(MPI_Datatype *datatype)
```

**Fortran:**

```
MPI_TYPE_COMMIT (datatype, ierr)
```

루틴 MPI\_TYPE\_COMMIT은 유도 데이터 타입을 통신상에서 사용할 수 있도록 등록한다. 한 번 데이터 타입이 등록되면 해제되기 전까지 통신상에서 반복적으로 재사용될 수 있다. 등록된 데이터 타입의 해제는 루틴 MPI\_TYPE\_FREE (datatype, ierr) 를 이용한다.

### 2.4.4 부분 배열의 전송 : MPI-2

**MPI\_TYPE\_CREATE\_SUBARRAY:**

**C :**

```
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);
```

**Fortran :**

**MPI\_TYPE\_CREATE\_SUBARRAY** (ndims, array\_of\_sizes,  
array\_of\_subsizes, array\_of\_starts, order, oldtype, newtype, ierr) **INTEGER**  
ndims : 배열의 차원 ( 양의 정수 ) (IN)

**INTEGER** array\_of\_sizes(\*) : 전체 배열의 각 차원의 크기 , i 번째 원소는 i 번째 차원의 크기 ( 양의 정수 ) (IN)

**INTEGER** array\_of\_subsizes(\*) : 부분 배열의 각 차원의 크기 , i 번째 원소는 i 번째 차원의 크기 ( 양의 정수 ) (IN)

**INTEGER** array\_starts(\*) : 부분 배열의 시작 좌표 , i 번째 원소는 i 번째 차원의 시작 좌표 ( 0 부터 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)

병렬화 과정에서 전송해야 하는 데이터가 메모리상에서 인접해있지 않은 부분행렬 (submatrix) 의 원소로 구성되는 경우가 자주있다 . 이런 경우 사용자는 부분행렬로 만들어지는 유도 데이터 타입을 정의하여 데이터를 전송하게 되는데 2 차원 이상의 행렬로 구성되는 데이터를 분석해 새로운 데이터 타입을 정의하는 것이 사용자에게 상당히 까다로운 문제가 된다 . MPI 는 새롭게 발표된 MPI-2 에서 사용자가 간단하게 부분행렬을 유도 데이터 타입으로 정의할 수 있도록 하는 루틴 **MPI\_TYPE\_CREATE\_SUBARRAY** 을 제공하고 있다 .

복잡한 부분행렬 데이터의 전송을 편리하게 할 수 있기 때문에 비록 MPI-2 에 포함된 루틴이지만 여기서 그 사용법을 간단히 알아본다 . 단 , KISTI IBM 1 차 시스템 환경에서는 MPI-2 에 포함된 루틴을 호출해 사용하기 위한 컴파일 스크립트를 “\_r” 을 붙여 사용하도록 설정해 두었기 때문에 컴파일 과정에서 사용자는 **mpxlf90\_r** 또는 **mpcc\_r** 등을 이용하기 바란다 .

**MPI\_TYPE\_CREATE\_SUBARRAY** 의 다섯번째 인수 **order** 는 사용언어에 따라 **MPI\_ORDER\_FORTRAN**, **MPI\_ORDER\_C** 중 하나를 선택하여 데이터의 행 우선순, 열 우선순 저장에 신경쓸 필요없이 편리하게 유도 데이터 타입을 작성할 수 있도록 한다 .

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**

그림 2.35 MPI\_TYPE\_CREATE\_SUBARRAY

예제 2.33 MPI\_TYPE\_CREATE\_SUBARRAY : Fortran

```

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

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, ibuf1
CALL MPI_FINALIZE(ierr)
END

```

예제 2.34 MPI\_TYPE\_CREATE\_SUBARRAY : C

```
#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;

    MPI_Type_create_subarray(ndims, array_of_sizes, array_of_subsizes,
array_of_starts, MPI_ORDER_C, MPI_INTEGER, &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();
}

```

## 2.5 커뮤니케이터

지금까지 커뮤니케이터라 하면 `MPI_COMM_WORLD` 를 떠올렸을 것이다. `MPI_COMM_WORLD` 는 프로그램 실행중에 서로 통신가능한 모든 프로세스를 포함하는 MPI 에서 미리 정의된 커뮤니케이터이다. 실제로 대부분의 MPI 프로그램들은 단일 문제를 해결하기 위해 모든 프로세스들이 같이 참여한다. 그러나, 동시에 처리될 수 있는 부분문제들로 구성된 문제를 푸는 경우 프로세스들의 부분집합에 각 문제를 할당시켜 좀더 효율적으로 그 문제를 해결할 수 있을 것이다. 예를 들어, 유체속을 비행하는 비행기를 모의실험하는 경우는 유체역학 부분과 구조해석 (structural analysis) 부분으로 문제를 나누어 각각을 다른 커뮤니케이터에 할당해 두 문제를 동시에 해결할 수 있다. 새로운 커뮤니케이터의 사용은 다음 장에서 다루어지는 가상 토폴로지의 개념과 더불어 프로그램의 가독성 (readability) 와 지속성 (maintainability) 을 증가 시킨다. 커뮤니케이터와 관련된 작업을 수행하는 여러 가지 MPI 루틴들이 있지만 여기서는 유용하게 사용할 수 있는 `MPI_COMM_SPLIT` 의 사용법에 대해서 알아본다.

### 2.5.1 MPI\_COMM\_SPLIT

**C:**

```
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm
*newcomm)
```

**Fortran:**

```
MPI_COMM_SPLIT(comm, color, key, newcomm, ierr)
```

INTEGER comm: 커뮤니케이터 ( 핸들 ) (IN)



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

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

INTEGER newcomm: 새로운 커뮤니케이터 ( 핸들 ) (OUT)

MPI\_COMM\_SPLIT 은 comm 에 포함된 프로세스들을 color 에 따라 여러 개로 나뉜 새로운 프로세스 집합 ( 커뮤니케이터 ) 을 만든다 . 각 부분집합들은 같은 color 를 가지는 모든 프로세스들을 포함하게 되며 , 인수 key 에 부여된 값의 순서에 따라 프로세스의 랭크를 결정한다 . color 는 0 이상의 정수이어야 하고 , 만약 어떤 프로세스의 color 가 MPI\_UNDEFINED 로 주어지면 , newcomm 은 MPI\_COMM\_NULL 을 리턴한다 .

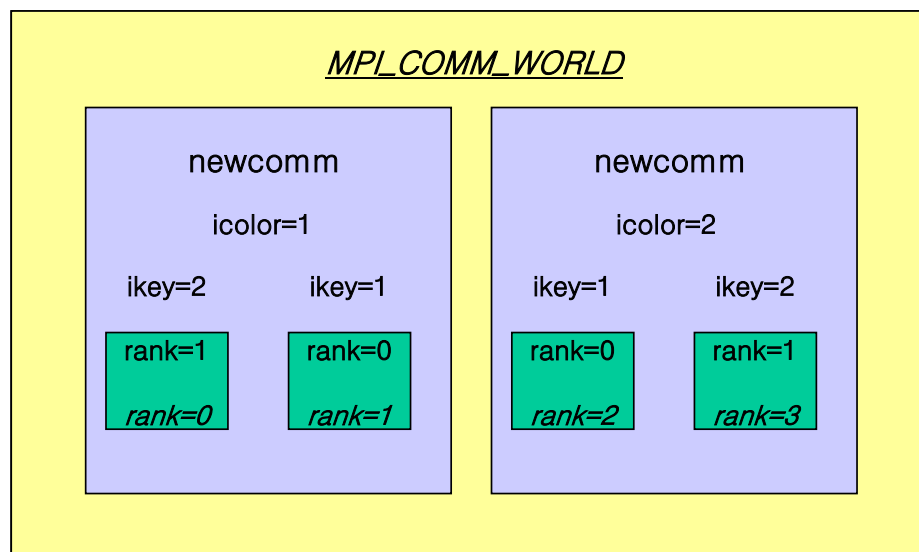


그림 2.36 MPI\_COMM\_SPLIT

예제 2.35 MPI\_COMM\_SPLIT : 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

```

예제 2.36 MPI\_COMM\_SPLIT : C

```
/*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;

    }

    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();

}

```

## 2.6 가상 토폴로지 (Virtual Topology)

가상 토폴로지는 보다 효율적인 통신을 위해 커뮤니케이터내의 프로세스들에게 적절한 이름을 부여하는 방법이며, 사용자는 가상 토폴로지를 통해 이후의 코드 작업을 좀더 단순화 시킬 수 있다. 예를 들어, 프로그램이 아래 그림에서 보는 2 차원 그리드의 프로세스 구성에서 주로 최인접 이웃 (nearest neighbor) 과 통신을 한다면, 사용자는 그 사실을 반영하는 가상 토폴로지를 우선 구성한 후, 그리드 좌표를 통해 인접 프로세스의 랭크를 계산하고 이를 송 / 수신 루틴의 인수로 사용하면 송 / 수신자의 식별이 보다 간단해질 것이다. 비록 프로세스의 랭크와 그리드상의 좌표를 대응시키는 문제는 단순한 정수 연산이어서 사용자에게 의해 쉽게 구현될 수 있다 하더라도 가상 토폴로지는 여전히 간단한 방법을 제공한다.

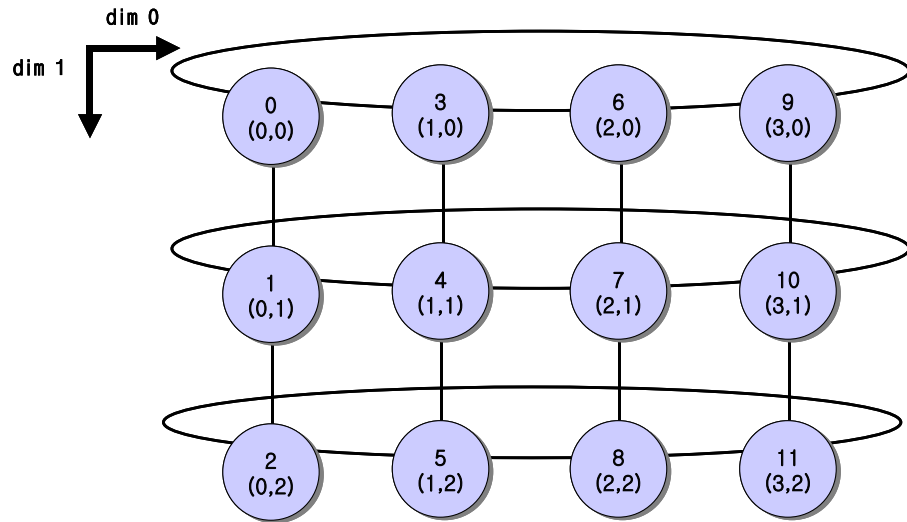


그림 2.37 차원 프로세스 그리드

가상 토폴로지가 현재의 커뮤니케이터에서 구성될 때, 자동적으로 새로운 커뮤니케이터가 하나 만들어지며, 사용자는 가상 토폴로지를 사용하기 위해 이전의 커뮤니케이터가 아닌 새로 만들어진 커뮤니케이터를 이용해야 한다.

MPI에서는 직교좌표와 그래프의 두 가지 토폴로지를 제공하고 있으나 여기서는 각 프로세스가 가상의 그리드상에서 최인접 이웃들과 연결되는, 그리드 형태의 토폴로지 구성에 적합한 직교좌표 가상 토폴로지를 다룬다. 그래프 가상 토폴로지는 보다 일반적인 토폴로지로서 한정된 프로세스와 연결되는 직교좌표와는 달리 커뮤니케이터 내의 어떤 프로세스와도 연결이 가능하다는 특징을 가진다.

### 2.6.1 직교좌표 가상 토폴로지 만들기 : MPI\_CART\_CREATE

C:

```
int MPI_Cart_create(MPI_Comm oldcomm, int ndims, int *dimsz, int
*periods, int reorder, MPI_Comm *newcomm)
```

Fortran:

**MPI\_CART\_CREATE(oldcomm, ndims, dimsize, periods, reorder, newcomm, ierr)**

INTEGER oldcomm: 기존 커뮤니케이터 (IN)

INTEGER ndims: 직교좌표의 차원 (IN)

INTEGER dimsize(\*): 각 좌표축의 길이 . 크기 ndims 의 배열 (OUT)

LOGICAL periods(\*): 각 좌표축의 주기성결정 . 크기 ndims 의 배열 (IN)

LOGICAL reorder: MPI 가 프로세스 랭크를 재 정렬할 것인가를 결정 (IN)

INTEGER newcomm: 새로운 커뮤니케이터 (OUT)

MPI\_CART\_CREATE는 현재 커뮤니케이터 oldcomm으로부터 가상 토폴로지의 구성을 가지게 되는 새로운 커뮤니케이터 newcomm 을 생성한다 . 인수 periods 는 정의된 좌표축 방향으로 주기적 경계 조건 (periodic boundary condition) 이 만족되면 TRUE 그렇지 않으면 FALSE 이다 . reorder 인수가 TRUE 이면 MPI 는 프로세스 랭크를 다시 할당하게 되는데 , 데이터를 아직 분배하지 않은 경우 사용하며 프로세스 랭크를 다시 할당한 후 새로운 커뮤니케이터에 데이터를 할당하게 된다 . 이미 데이터를 프로세스에 할당했다면 reorder 인수를 FALSE 로 설정하는데 이 때 프로세스 랭크는 oldcomm 커뮤니케이터 상태의 랭크로 남아있고 , 단지 랭크와 좌표 사이의 대응만이 설정된다 .

아래 예제에서는 랭크가 0, 1, 2, 3, 4, 5 로 순서가 정해진 6 개의 프로세스를 3 행 2 열의 2 차원 그리드 순으로 재배치 하고 있으며 , 다음 그림은 그 결과를 나타낸 것이다 .

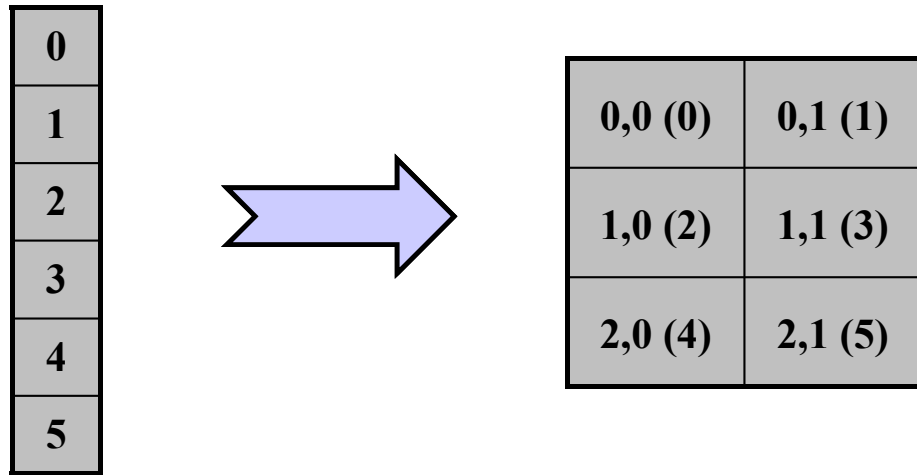


그림 2.38 MPI\_CART\_CREATE

예제 2.37 MPI\_CART\_CREATE : Fortran

```

PROGRAM cart_create

INCLUDE 'mpif.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 = .TRUE.
```

```
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
```

예제 2.38 MPI\_CART\_CREATE : 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 = 1;
```



```

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();
}

```

## 2.6.2 대응 함수

MPI\_CART\_CREATE 로 생성된 프로세스 토폴로지는 대응 함수를 통하여 토폴로지 상의 명명 방식에 근거한 프로세스 랭크가 계산된다.

### **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 ndims 의 배열 (IN)

INTEGER rank : coords 에 의해 표현되는 프로세스의 랭크 (OUT)

MPI\_CART\_RANK 루틴은 프로세스 그리드 좌표를 프로세스 랭크로 나타낸다. 이 루틴은 그리드 좌표를 알고 있는 경우 그 좌표에 해당하는 프로세스로 메시지를 보내거나 그 프로세스로부터 메시지를 받고자 할 때 프로세스의 랭크를 알기 위해 사용한다.

앞서 MPI\_CART\_CREATE 의 예제 2.37 과 2.38 에 아래의 코드를 삽입해 사용하면 된다.

예제 2.39 MPI\_CART\_RANK : 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

예제 2.40 MPI\_CART\_RANK : 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\_COORDS:**

**C:**

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)
```

**Fortran:**

```
MPI_CART_COORDS(comm, rank, maxdims, coords, ierr)
```

1

MPI\_CART\_COORDS 루틴은 MPI\_CART\_RANK 의 역함수 역할을 한다. 즉, 프로세스 랭크를 아는 경우 그 프로세스의 직교 좌표를 리턴한다.

아래 코드는 MPI\_CART\_CREATE 의 예제 2.37 과 2.38 에 삽입해 사용하면 된다.

예제 2.41 MPI\_CART\_COORDS : 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
```

예제 2.42 MPI\_CART\_COORDS : 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\_SHIFT:**

**C:**

**int MPI\_Cart\_shift(MPI\_Comm comm, int direction, int displ, int \*source, int \*dest)**

**Fortran:**

**MPI\_CART\_SHIFT(comm, direction, displ, source, dest, ierr)**

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)

MPI\_CART\_SHIFT는 실제 시프트를 실행하지는 않는다. 단지 직교 좌표 토폴로지 상에서 특정방향을 따라 호출프로세스의 이웃 프로세스를 발견하는데 이용된다. 발견되는 두 프로세스를 각각 근원 (source) 랭크와 수신 (destination) 랭크라 하며 루틴을 호출한 프로세스와의 인접정도는 인수 displ로 결정한다. 이렇게 발견된 두 프로세스의 랭크는 MPI\_SEND, MPI\_RECV, MPI\_SENDRECV 등의 인수로 이용하여 메시지 전송을 하게 된다.

다음 예제 역시 MPI\_CART\_CREATE의 예제 2.37 과 2.38에 삽입해 사용하면 된다.

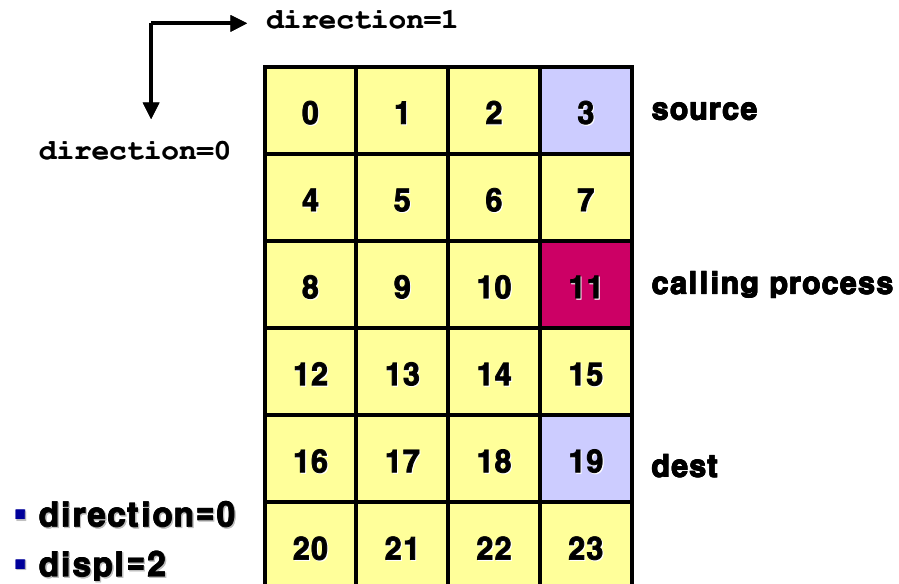


그림 2.39 MPI\_CART\_SHIFT

예제 2.43 MPI\_CART\_SHIFT : 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 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

...

예제 2.44 MPI\_CART\_SHIFT : 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);

...

### 2.6.3 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)

MPI\_CART\_SUB 는 N 차원 직교좌표 그리드를 행방향 또는 열방향으로 나누어 부분그리드 (subgrid) 로 새로운 커뮤니케이터를 만든다 . 프로그램에서 환산 (reduction) 등의 집합통신을 수행하고자 할 때 전체 그리드가 아닌 그리드의 행 (row) 들 또는 열 (coloumn) 들에 대해서만 통신이 필요한 경우가 있다 . 이런 경우 MPI\_CART\_SUB 를 이용해 그리드의 일부만을 포함하는 새로운 커뮤니케이터를 정의해 통신을 수행할 수 있다 . MPI\_COMM\_SPLIT 과 기능이 유사하다 .

아래의 예제는 MPI\_CART\_CREATE 의 예제 2.37 과 2.38 에 삽입해 사용하면 된다 .



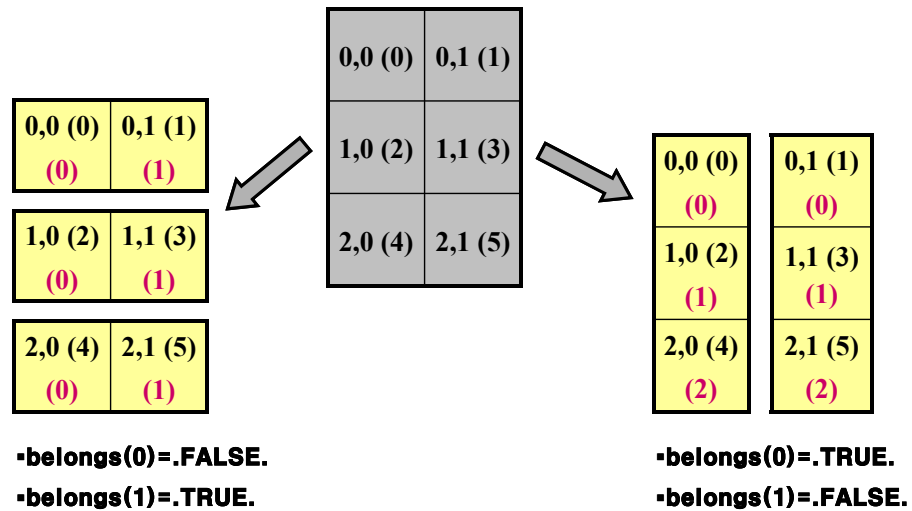


그림 2.40 MPI\_CART\_SUB

예제 2.45 MPI\_CART\_SUB : Fortran

...

ndims = 2

**dimsize(0) = 3; dimsize(1) = 2**

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)

**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=', rank
```

```
...
```

예제 2.46 MPI\_CART\_SUB : 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);
```

```
...
```

## 제 3 장 MPI 를 이용한 병렬 프로그래밍 실제

앞에서 MPI 의 기본개념과 기초적인 MPI 서브루틴들에 대해 알아보았다 . 이 단원에서는 이러한 내용을 바탕으로 실제 MPI 를 이용한 병렬 프로그램 작성시 염두에 두어야 할 데이터 처리 방식과 , 프로그래밍 테크닉 , 주의할 점 등에 대해 알아본다 .

### 3.1 병렬 프로그램에서의 입 / 출력

병렬 프로그램에서 입 / 출력 부분을 병렬화 하는 몇가지 전형적인 방법들에 대해 알아 본다 .

#### 3.1.1 입력

모든 프로세스들이 공유 파일 시스템으로부터 입력파일을 읽어오는 경우 :

입력파일은 공유 파일 시스템에 위치하고 각 프로세스들은 모두 동일한 파일로부터 데이터를 읽어들인다 . 예를 들어 파일 시스템이 NFS(Network File System) 라면 파일시스템은 고속네트워크를 통해 시스템에 마운트 되겠지만 파일을 읽기 위한 프로세스들 간의 경쟁은 피할 수 없다 . 이러한 경쟁을 없애기위해 GPFS(General Parallel File System) 과 같은 병렬 파일 시스템이 이용이 될 수 있다 .

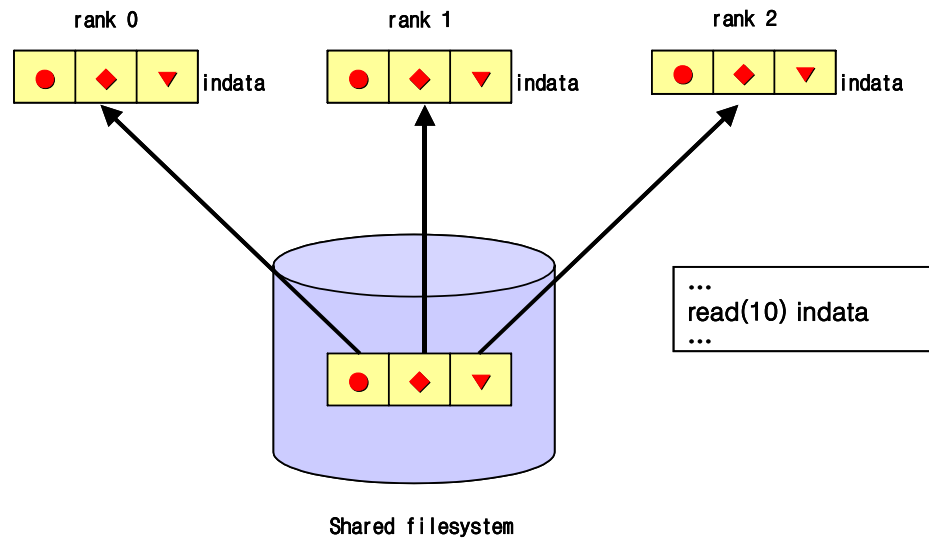


그림 3.1 공유 파일 시스템에 입력파일을 둔 경우

#### 각 프로세스가 입력파일의 복사본을 각각 따로 가지는 경우 :

프로그램을 실행하기전 각 프로세스들의 로컬 파일 시스템에 입력 파일을 복사해 둔다. 프로세스들은 각자 자신의 파일 시스템에서 데이터를 읽어들이게 되므로 공유 파일 시스템으로부터 읽게되는 경우보다 성능면에서 유리하다. 그러나, 디스크비용이 많이 들고 부가적으로 파일을 복사하는 작업이 필요하게된다.

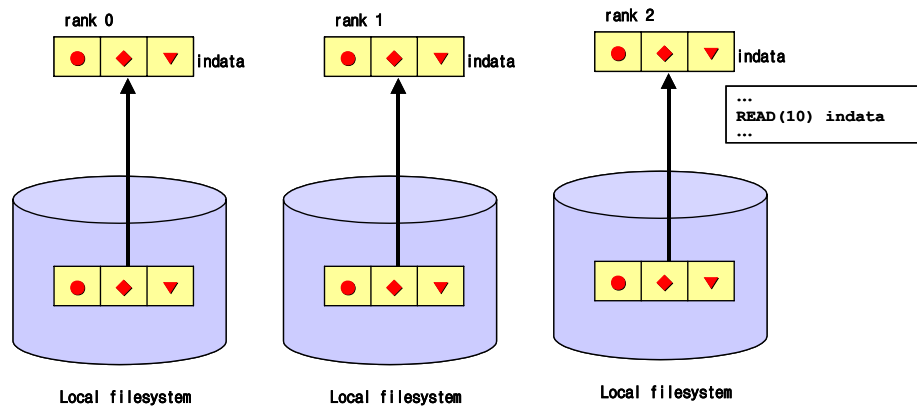


그림 3.2 각 노드에 입력 파일을 복사해둔 경우

한 프로세스가 입력파일을 읽어 다른 프로세스들에게 전달하는 경우 1.

한 프로세스가 입력파일을 읽어 들이고, MPI\_BCAST 와 같은 루틴들을 이용하여 다른 프로세스들에게 데이터를 전달한다

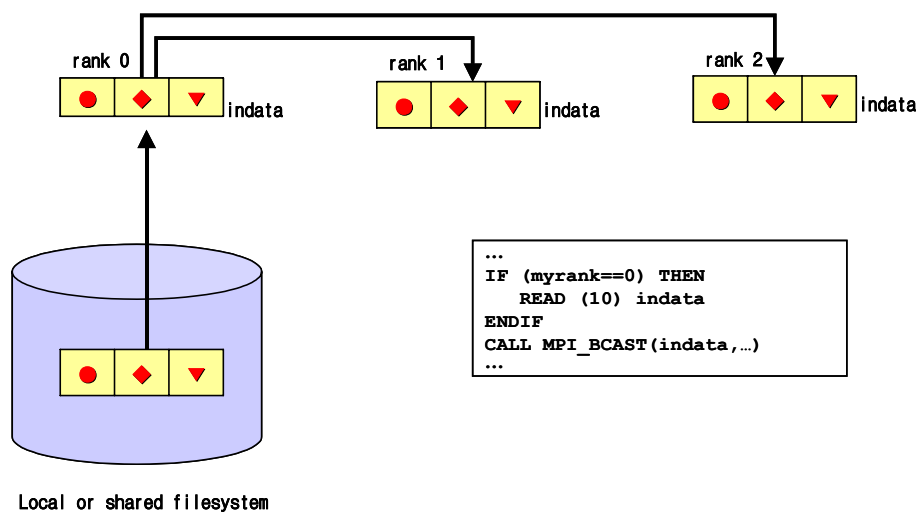


그림 3.3 한 프로세스가 입력 파일을 읽어 각 노드에 전달하는 경우 1

## 한 프로세스가 입력파일을 읽어 다른 프로세스들에게 전달하는 경우 2.

한 프로세스가 입력파일을 읽어 다른 프로세스들에게 전달하는 방법에서 사용자는 각 프로세스에서 필요한 최소한의 데이터만을 읽어 들이도록 코드를 수정할 수 있다. 예를 들어, 아래와 같이 MPI\_BCAST 대신 MPI\_SCATTER를 사용하면 각 프로세스는 필요한 데이터만 가져와 작업을 실행하므로 메모리를 절약할 수 있다.

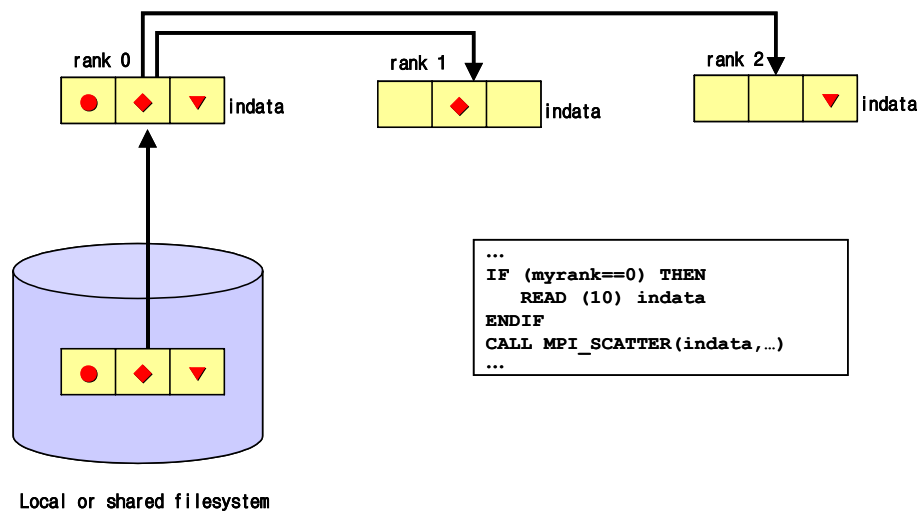


그림 3.4 한 프로세스가 입력 파일을 읽어 각 노드에 전달하는 경우 2

### 3.1.2 출력

#### 표준 출력 :

기본적으로 병렬 작업에 참여한 모든 프로세스들은 터미널에 표준 출력 메시지를 출력한다. 사용자는 코드를 다음과 같이 수정하여 원하는 프로세스만 출력을 하도록 할 수 있다.

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

IBM 의 AIX 환경에서 환경변수 MP\_STDOUTMODE 를  
 "MPI\_STDOUTMODE=rank\_id" 로 설정해도 원하는 한 프로세서만이 출력 하도록  
 할 수 있다. 참고로 ,MP\_STDOUTMODE 의 기본설정은 'unordered' 로 모든 프로세  
 스들이 랭크순서와 상관없이 출력 데이터를 출력하게 된다. 모든 프로세스들이 프  
 로세스 랭크순서와 상관없이 출력하길 원한다면  
 "MP\_STDOUTMODE=ordered" 와 같이 설정하면 된다.

한 프로세스가 데이터를 모두 모아서 로컬 파일시스템에 저장하는 경우.

먼저, 한 프로세스가 다른 프로세스들로부터 데이터를 모은다. 그리고, 그 프로세  
 스는 모은 데이터를 자신의 파일시스템에 파일로 저장한다.

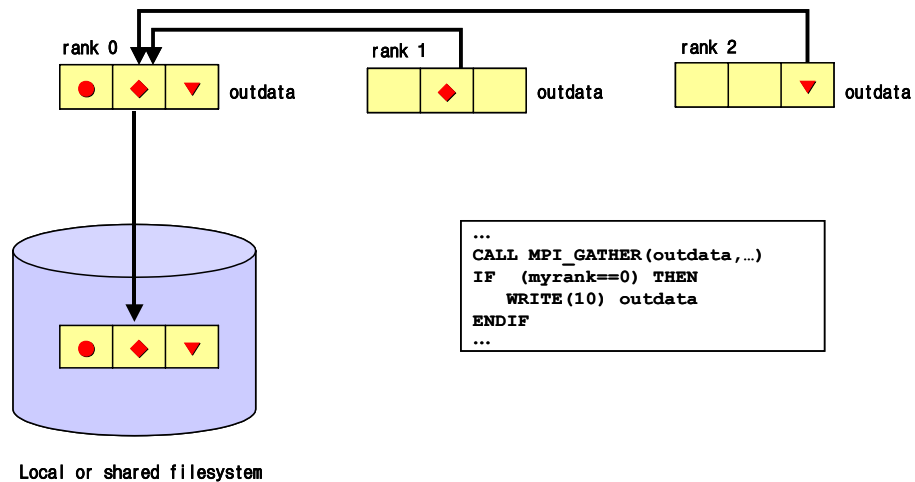


그림 3.5 한 프로세스가 데이터를 모아서 로컬 파일 시스템에 저장

각 프로세스가 공유 파일 시스템에 데이터를 순차적으로 저장하는 경우.

출력파일은 공유 파일시스템에 하나가 존재하므로, 각 프로세스들이 자신의 데이  
 터를 동시에 한 파일에 쓰려하면 경쟁이 발생하게 된다. 따라서, 각 프로세스들은  
 순차적으로 자신들의 데이터를 파일로 옮겨야 한다.

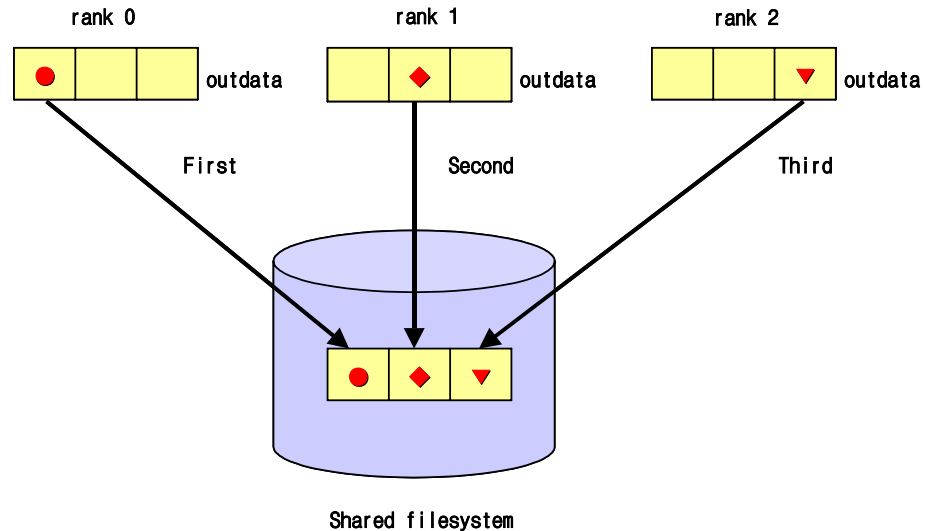


그림 3.6 공유 파일 시스템에 순차적으로 저장하는 경우

## 3.2 DO(for) 루프의 병렬화

대부분의 과학계산 프로그램에서 가장 시간을 많이 잡아먹는 부분 (hot spot) 들은 주로 DO 루프에서 발견된다. 따라서, 프로그램을 병렬화하여 성능이득을 원하는 사용자들에게 DO 루프의 병렬화는 무엇보다 중요하다. DO 루프 병렬화의 기본은 반복되는 루프를 프로세스들에게 할당하여 각자 할당된 부분을 병렬로 계산하게 하는 것이다. 그런데, 대부분 DO 루프는 루프 인덱스와 관련된 인덱스를 가지는 배열들을 포함하게 된다. 따라서 루프 반복의 분배는 배열을 어떻게 나누고, 나눠진 배열 덩어리 (chunk) 와 그것과 관련된 계산들을 어떻게 프로세스들에게 효율적으로 할당하는가의 문제이다.

### 3.2.1 블록 분할 (Block Distribution)

블록 분할에서 루프의 반복은 병렬 실행에 참여하는 프로세스 개수 만큼의 부분으로 나뉘게 된다. 이를테면 4 개의 프로세스에서 100 번의 반복계산을 수행하고자 하면, 각 프로세스는 고르게 25 개씩을 맡아 프로세스 0 는 1 에서 25 까지, 프로세스 1 은 26 에서 50 까지, 프로세스 2 는 51 에서 75 까지, 그리고 프로세스 3 은 76



에서 100까지의 계산을 담당하게 되는 식이다. 그러나, 대체적으로 반복계산의 회수  $n$ 이 프로세스의 개수  $p$ 로 나누어 떨어지지 않는 경우가 많을 것이며 그런 경우 작업부하의 균형을 고려하여 효율적인 작업 할당 방법을 고려해야 한다.

블록 분할을 위한 한 가지 방법을 소개하면  $n$ 을  $p$ 로 나누었을 때의 몫이  $q$ , 나머지가  $r$ 일 때 ( $n=p*q+r$ ), 프로세스  $0 \cdots r-1$ 은 각각  $q+1$ 번의 반복계산을 수행하고, 나머지 프로세스들은 각각  $q$ 번의 반복계산을 수행하도록 하는 것이다. 이것을 식으로  $n=r(q+1) + (p-r)q$ 와 같이 표현할 수 있다.

다음 그림은 위의 방법으로 14번의 반복계산을 4개의 프로세스에게 할당시킨 결과를 보여준다.

| 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  |

그림 3.7 블록 분할

다음 루틴은 물론, MPI에서 제공하는 것은 아니지만 위에서 설명한 방법으로 반복계산을 나눌 때 각 프로세스에 할당되는 인덱스 범위를 계산해주는 서브루틴으로서, 사용자들의 병렬 코드 작업을 좀더 간편히 해주는데 도움이 될 것이다.

예제 3.1 블록 분할 코드 : 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

```

INTEGER n1: 반복되는 루프 인덱스의 최소값 (IN)

INTEGER n2: 반복되는 루프 인덱스의 최대값 (IN)

INTEGER nprocs: 프로세스 개수 (IN)

INTEGER irank: 인덱스 범위만큼 작업을 받게되는 프로세스의 rank(IN)

INTEGER ista: 프로세스 irank 가 할당받는 인덱스 범위의 최소값 (OUT)

INTEGER iend: 프로세스 irank 가 할당받는 인덱스 범위의 최대값 (OUT)

예제 3.2 블록 분할 코드 : 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;  
}
```

다음에서는 배열 a()의 원소의 총합을 블록 분할을 이용해 병렬로 계산하는 예제를 이용해 위에서 언급한 서브루틴을 활용해 보자. 순차 프로그램은 다음과 같다.

예제 3.3 배열의 합을 구하는 순차 프로그램 : Fortran

```
PROGRAM main
```

```

PARAMETER (n = 1000)

DIMENSION a(n)

DO i = 1, n

    a(i) = i

ENDDO

sum = 0.0

DO i = 1, n

    sum = sum + a(i)

ENDDO

PRINT *, '    sum =', sum

END

```

예제 3.4 배열의 합을 구하는 순차 프로그램 : C

```

/* serial_main */

#include <stdio.h>

#define n 1000

void main(){

    double a[n], sum;

    int i;

    for(i=0; i<n; i++) a[i] = i+1;

    sum=0.0

    for(i=0; i<n; i++) sum = sum+a[i];

    printf( "sum = %f\n" , sum);
}

```

```
}
```

다음 프로그램은 서브루틴 `para_range` 를 이용해 반복계산을 블록 분할 방식으로 각 프로세스에 할당하고 프로세스에서 계산된 부분합을 `MPI_REDUCE` 를 이용해 총합을 구하는 병렬 프로그램이다. C 에는 최소값을 구하는 내부함수가 없어 함수 `min` 을 따로 만들어 사용하고 있다.

예제 3.5 블록 분할을 이용한 병렬 프로그램 : Fortran

```
PROGRAM para_sum

INCLUDE 'mpif.h'  PARAMETER (n = 100000)

DIMENSION a(n)

CALL MPI_INIT(ierr)

CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)

CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

CALL para_range(1, n, nprocs, myrank, ista, iend)

DO i = ista, iend

    a(i) = i

ENDDO

sum = 0.0

DO i = ista, iend

    sum = sum + a(i)

ENDDO

CALL MPI_REDUCE(sum, ssum, 1, MPI_REAL, MPI_SUM, 0, &
MPI_COMM_WORLD, ierr)

sum = ssum
```

```

IF (myrank == 0) PRINT *, 'sum =', sum

CALL MPI_FINALIZE(ierr)

END

```

예제 3.6 블록 분할을 이용한 병렬 프로그램 : C

```

/*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];

```

```

        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){

...}

```

### 3.2.2 순환 분할 (Cyclic Distribution)

순환 분할은 반복계산의 인덱스를 프로세스에게 라운드로빈 (round-robin) 방식으로 할당한다. 블록 분할을 사용하는 경우의 문제점은 프로세스들 간의 작업부하가 균등하게 분배되지 못하는 경우가 많다는 것인데, 순환 분할을 이용하면 어느 정도 이런 문제점에 대한 해결이 가능하다.

다음 예제 8.7의 계산은 예제 8.8과 같이 순환 분할할 수 있다.

예제 3.7 DO 루프 계산 코드

```
DO i = n1, n2
```

```
    computation
```

```
ENDDO
```

예제 3.8 순환 분할의 구현

```
DO i = n1+myrank, n2, nprocs
```

```
    computation
```

```
ENDDO
```

다음 그림은 순환 분할을 이용해 14 번의 반복계산을 4 개의 프로세스에 어떻게 할당하는가를 보여준다.

| 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  |

그림 3.8 순환 분할

일반적으로 순환 분할은 데이터들이 메모리내에 연속적으로 저장되지 않기 때문에 블록 분할보다 캐시미스가 많이 발생한다는 단점이 있다.

### 3.2.3 블록 순환 분할 (Block-Cyclic Distribution)

블록 분할보다 부하 균형을 좋게하고 순환 분할보다 캐시미스 발생을 줄이고자 한다면 다음과 같은 블록 순환 분할의 이용을 고려해볼 수 있다. 블록 순환 분할은 이 름에 나타난 그대로 반복계산의 인덱스를 같은 크기의 덩어리로 나누고 ( 마지막

덩어리는 크기가 다를 수 있다.) 이 덩어리들을 라운드-로빈 방식으로 프로세스에 분배하는 방식이다.

| 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

그림 3.9 블록 순환 분할

그림 3.9는 14 번의 반복계산을 4 개의 프로세스에 블록의 크기가 2 인 블록 순환 분할로 분배한 결과를 보여주고 있다.

예제 3.7 의 루프 계산을 블록의 크기를 iblock 으로 하여 블록 순환 분할한 코드는 다음과 같다.

예제 3.9 블록 순환 분할의 구현

```
DO ii = n1+myrank*iblock, n2, nprocs*iblock
```

```
  DO i = ii, MIN(ii + iblock-1, n2)
```

```
    computation
```

```
  ENDDO
```

```
ENDDO
```

### 3.2.4 배열 수축 (Shrinking Arrays)

위에서 살펴본 몇 가지 예와 같이 블록, 순환, 블록 순환 분할을 이용하여 배열을 프로세스에게 할당하는 경우, 프로세스들은 전체 배열중에서 각자가 담당하는 부분만을 가지고 그 부분에 대한 계산만 수행하면 되므로 전체 배열을 프로세스마다 메모리에 저장해 두는 것은 불필요한 일이 될것이다. 만일 다른 프로세스가 가지



고 있는 데이터가 필요하다면, 통신을 통하여 필요한 데이터를 주고 받으면 된다. 분산 메모리 아키텍처를 가지는 병렬 시스템에서는 각 노드마다 개별적인 메모리를 가지며, 다른 노드에서 실행되는 프로세스와 공유되지 않는 별도의 주소공간을 사용하게 된다. 결국  $n$  개의 프로세스를 연결한 분산 메모리 시스템에서 병렬 작업을 하게 되면 1 개의 프로세스를 이용하는 순차 작업과 비교해  $n$  배의 메모리를 더 이용할 수 있는 셈이 된다. 사용자는 병렬 작업을 통하여 순차 작업에서 사용하는 데이터의 크기를 증가 시킬 수 있으며, 이런 기술을 배열 수축이라고 한다. 다음 그림 3.10, 3.11, 3.12 가 이해를 도울 것이다.

$a(i,j)$

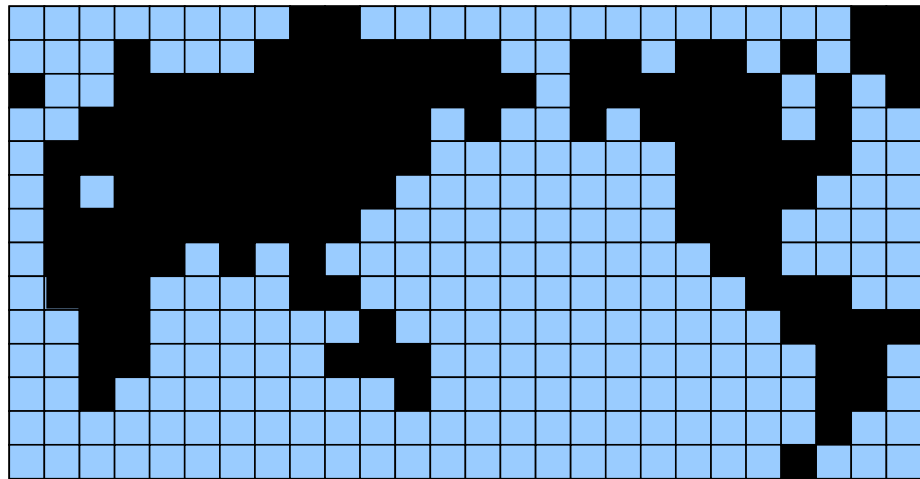


그림 3.10 순차 실행 : 데이터 원본

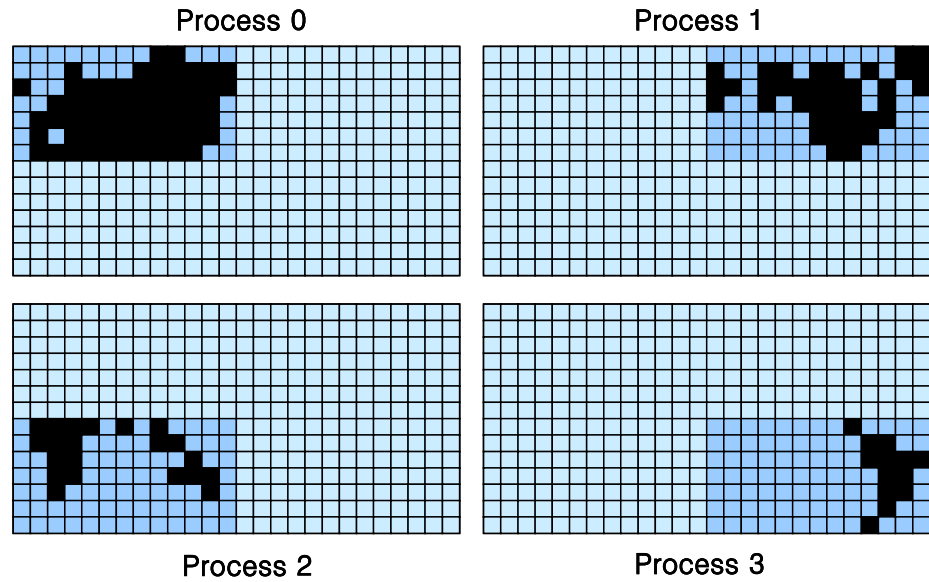


그림 3.11 병렬 실행 : 배열 수축하지 않은 데이터

위의 그림에서 배열 a() 는 지구 표면의 물리적 현상을 모의실험 하기위한 정보를 담고있다고 하자. 그림 3.10 에서 행렬 a() 의 크기와 노드 하나의 물리적 메모리가 꼭 맞게 되었다 가정하면 ,사용자는 그림 3.11 에서와 같이 4 개의 프로세스 즉 , 4 개의 노드에 데이터를 나누어 병렬로 모의실험을 수행할 수 있을 것이다 . 그림 3.11 은 아직 배열을 수축하지 않아 모의실험의 데이터 해상도가 그림 3.10 에서 메모리 제약을 받는 순차 실행의 경우와 같게된 모습을 나타내고 있다 .

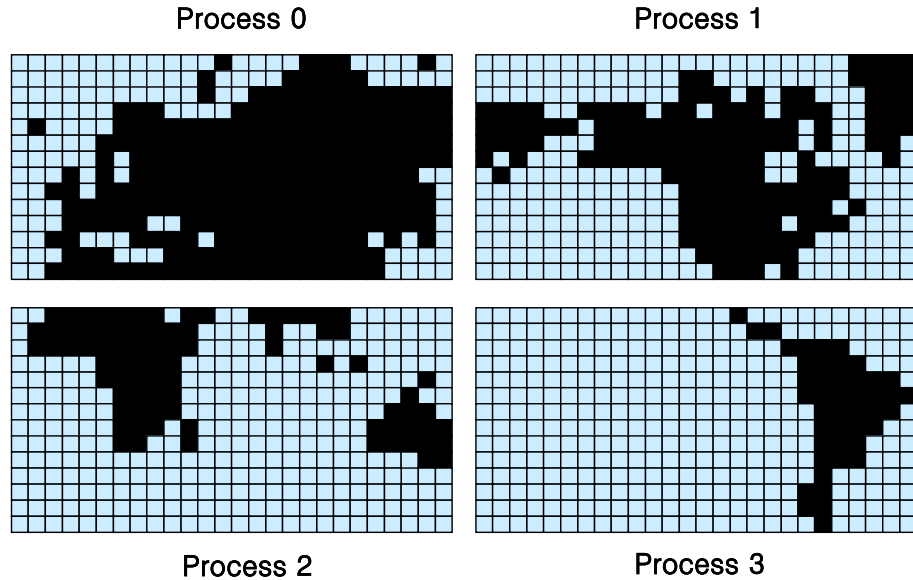


그림 3.12 병렬 실행 : 배열 수축에 의해 증가된 데이터

배열 수축의 핵심적인 생각은 프로세스들이 자신에게 할당된 데이터 이외의 데이터를 따로 가지고 있을 필요가 없다는 것이다. 이런 이유로, 각 프로세스는 자신의 영역에서 보다 많은 메모리를 사용할 수 있게 된다. 그림 3.12는 앞서의 모의실험 데이터의 해상도를 배열 수축을 통해 4 배로 높인 것이다. 위의 경우는 같은 영역의 데이터 해상도를 높인 것이지만 해상도를 그대로 하면서 다루고자 하는 데이터 영역을 4 배로 넓힐 수도 있을 것이다. 이와 같이 배열 수축 기술은 병렬처리를 통하여 보다 많은 메모리를 사용할 수 있도록 한다.

사용자가 노드를 추가 시키는 대로 메모리의 전체 양을 얼마든지 증가시킬 수 있는 MPP 시스템에서는 이 기술을 최고로 이용할 수 있다. 그러나, SMP 시스템은 상대적으로 적은 메모리 크기를 가지므로 사용자가 해결할 수 있는 문제의 크기에 제한이 있게 된다.

사용 가능한 프로세스 개수와 배열의 크기를 미리 알고 있다면 사용자가 직접 배열 수축을 하기에 큰 어려움이 없겠지만 만약 이러한 내용을 모른다면 다음 예제 3.10 과 3.11 같이 동적 메모리 할당 방법을 이용할 수 있다.

예제 3.10 메모리 동적 할당 : Fortran

```

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)

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( ... )
```

```
...
```

예제 3.11 메모리 동적 할당 : C

```
/*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));
```

```
    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();

}

```

### 3.2.5 내포된 (Nested) 루프의 병렬화

효율적인 프로그램 작성을 위해서 언제나 염두에 뒀어야 할 사실은 캐시미스와 통신을 최소화 하는 것이다 . 여기서는 내포된 루프의 병렬화 과정에서 캐시미스와 통신을 최소화하기 위한 방안들에 대해 알아 볼 것이다 .

**메모리 대응 :** 사용자는 우선 , 사용 언어에 따라 데이터가 메모리에 저장되는 방식이 다르다는 사실을 알고 있어야 한다 . Fortran은 다차원 배열을 열 우선순(column-major order) 으로 메모리에 저장하며 C는 이와 달리 행 우선순(row-major order) 으로 저장 한다 . 배열로 저장된 데이터에 접근을 할 때는 항상 메모리에 저장되어있는 연속적인 순서대로 접근을 하는게 편리하며 성능면에서도 효율적이다 .

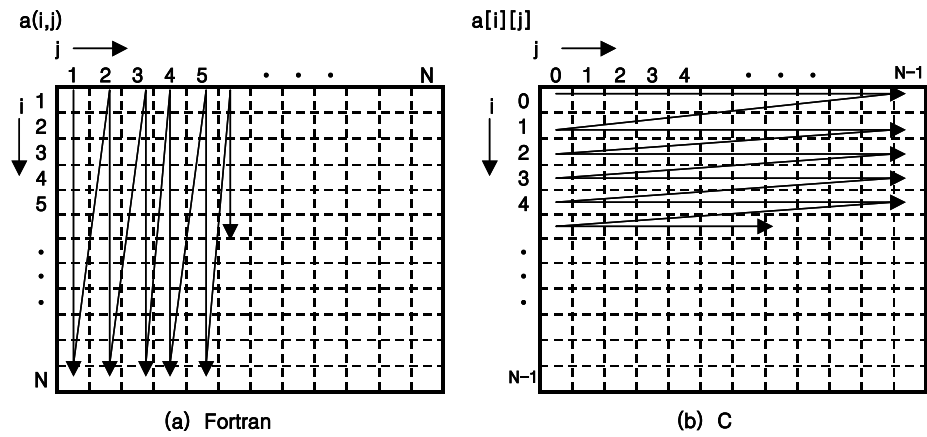


그림 3.13 2 차원 배열의 저장

캐시미스 줄이기 :

다음의 내포된 루프 A 와 B 를 보자 .

LOOP A

DO j = 1, n

DO i = 1, n

$a(i,j) = b(i,j) + c(i,j)$

ENDDO

ENDDO

LOOP B

DO i = 1, n

DO j = 1, n

$a(i,j) = b(i,j) + c(i,j)$

ENDDO

ENDDO

Fortran 에서 루프 B 는 행 우선순으로 데이터를 가져오고 저장하게 되므로 루프 A 와 비교해 많은 캐시미스가 발생하고 따라서 A 보다 느리게 실행된다. 물론 C 에서는 반대로 루프 B 가 루프 A 보다 빠르게 실행된다.

이제 루프 A를 선택해 병렬화 한다면 다음과 같이 바깥쪽 루프의 병렬화와 안쪽 루프의 병렬화, 두 가지 경우를 생각해 볼 수 있다.

LOOP A1

DO j = jsta, jend

DO i = 1, n

$a(i,j) = b(i,j) + c(i,j)$

ENDDO

ENDDO

LOOP A2

DO j = 1, n

DO i = ista, iend

$a(i,j) = b(i,j) + c(i,j)$

ENDDO

ENDDO



바깥쪽 루프를 병렬화한 A1 과 안쪽 루프를 병렬화한 A2 는 어느쪽이 더 빠르게 실행될까 ? 아래 그림을 참고로 하면 바깥쪽 루프를 병렬화한 루프 A1 이 루프 A2 보다 캐시미스가 적게 발생해 더 효율적이라는 사실을 알 수 있다 .

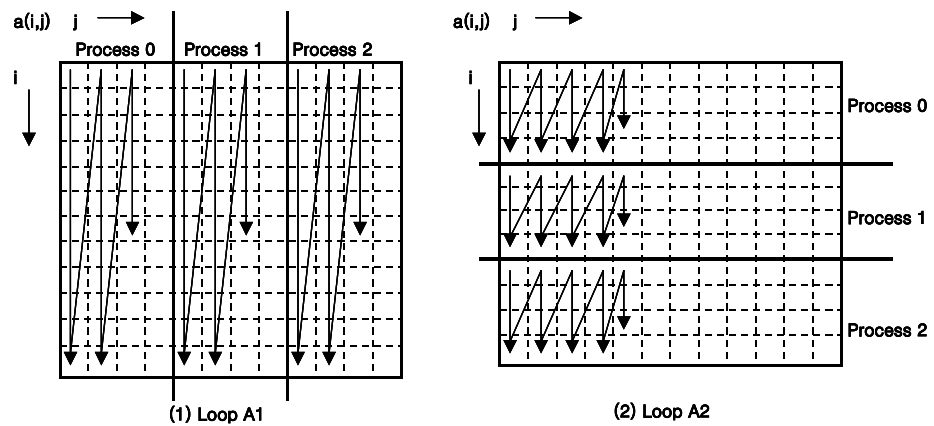


그림 3.14 내포된 루프의 병렬화

**통신량 줄이기 :** 병렬화 과정에서 다른 프로세스로부터 계산에 필요한 데이터를 전송받아야 하는 경우가 있다 . 앞서의 루프 A 에서 실행되는 계산에 필요한 데이터는 루프 인덱스를 각 프로세스에 할당하는 방식과 같이 각 프로세스에 분배된 것들이기 때문에 루프 A 의 계산은 지역적 (local) 이라고 할 수 있다 . 그러나 , 다음의 루프 C 는 다른 프로세스에 분배된 데이터를 필요로 한다 .

LOOP C

DO j = 1, n

DO i = 1, n

$$a(i,j) = b(i,j-1) + c(i,j+1)$$

ENDDO

ENDDO

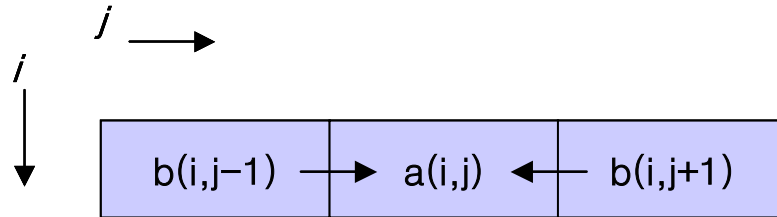


그림 3.15 루프 C 의 의존성

루프 C 는 같은 행에 있는 이웃한 데이터에 의존하는 계산을 실행한다 . 이런 경우 만약, 사용자가 바깥쪽 루프를 병렬화 한다면 사용자는 프로세스 사이의 경계에서 데이터를 교환해주어야 한다 . 아래 그림 8.14 에서 프로세스 r 은 데이터  $a(i,j)$  와  $b(i,j)$  의 값을  $1 \leq i \leq n, jsta \leq j \leq jend$  의 범위에서만 가지고 있다면 , 사용자는  $a(i,jsta)$  를 계산할 때 프로세스 r-1 로부터  $b(i,jsta-1)$  의 값을 전달 받아야 하며 ,  $a(i,jend)$  를 계산할 때는 프로세스 r+1 로부터  $b(i,jend+1)$  의 값을 전달 받아야 한다 . 전체적으로 프로세스 r 은 이웃 프로세스로부터  $2n$  개의 데이터를 전달 받아야 하고 다시 이웃 프로세스들에게  $2n$  개의 원소들을 전달해 주어야 한다 .

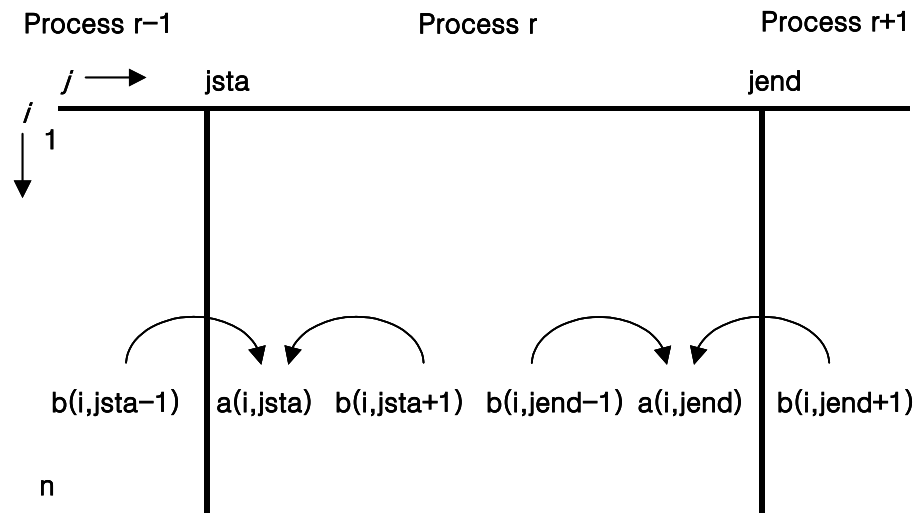


그림 3.16 열 방향으로 블록 분할된 루프 C 의 병렬화 의존성

여기서 만약 사용자가 안쪽 루프를 병렬화 한다면, 위와 같은 통신은 필요하지 않을 것이다. 일반적으로 통신에 의한 지연시간 (latency) 은 메모리 지연시간보다 크다. 따라서 이런 경우라면 사용자는 바깥쪽이 아닌 안쪽루프를 병렬화하는 것이 보다 효율적인 프로그램을 작성하는 길이 될 것이다.

루프 C 는 한쪽 방향으로만 인접 데이터에 의존하는 경우였다. 만일 루프가 양쪽 방향으로 다른 프로세스의 데이터를 필요로 한다면 어떻게 될것인가? 다음 루프 D 를 보자.

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

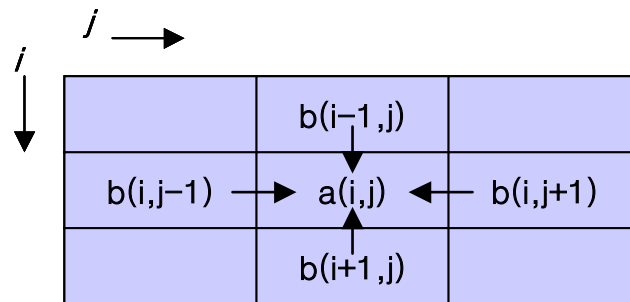


그림 3.17 루프 D 의 의존성.

이런 경우는 반복되는 범위가 병렬화에 중요한 역할을 한다. 어떤 방향이든 데이터 전송은 필요하고 전송되어야 하는 데이터의 개수는 프로세스들사이의 경계의 길이에 비례한다. 다음 그림에서는 각각의 경우, 전송이 필요한 행렬  $b(i,j)$  의 원소들이 어두운 부분으로 표시되어 있다. 만약  $m$  이  $n$  보다 크다면 안쪽 루프를 병렬

화 해야하고 그렇지 않다면 바깥쪽 루프를 병렬화 해야 효율적인 프로그램이 될 것임을 그림에서 확인할 수 있다.

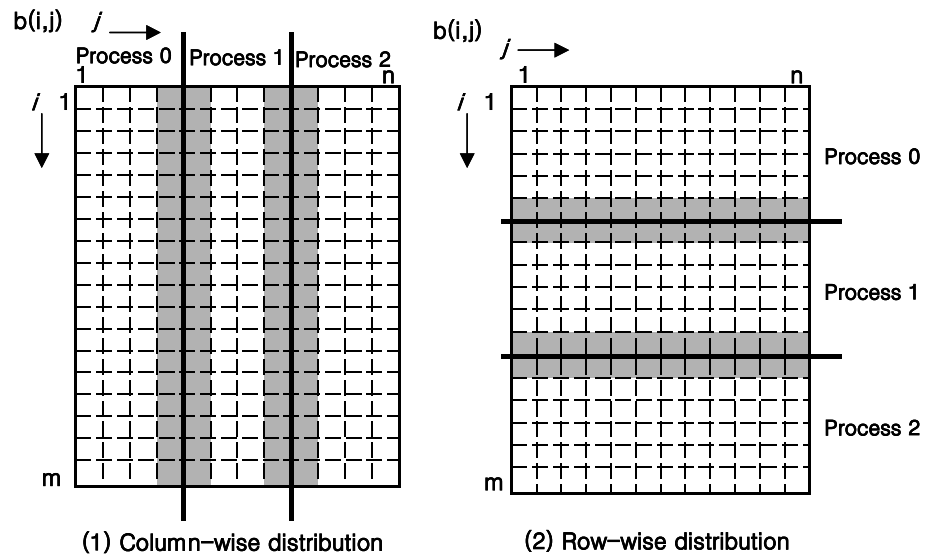


그림 3.18 열 방향(1) 과 행 방향(2) 으로 블록 분할된 루프 D 의 병렬화 의존성

이상에서는 안쪽이나 바깥쪽 중 하나의 루프를 병렬화 하는것에 대해 알아보았으나 다음의 루프 E 처럼 안쪽과 바깥쪽 루프를 모두 병렬화 할수도 있다.

```

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

```

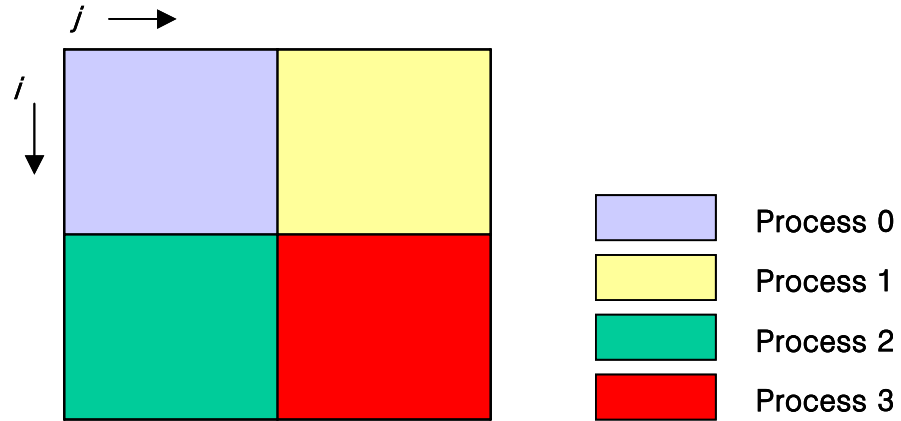


그림 3.19 양 방향 블록 분할

그림은 2 차원 행렬을 각 방향으로 두개씩으로 나누어 4 개의 프로세스에 분배 시키는 예를 보여준다 . 이렇듯 양 방향으로 블록 분할을 하여 프로세스에 할당하는 경우 프로세스의 개수는 합성수여야 하는 제한이 뒤따른다 . 프로세스의 개수가 합성수라면 이 수를 두 자연수의 곱으로 나타내는 방식은 여러 가지가 가능할 것이다 . 가령 12 개의 프로세스는  $2 \times 6$  또는  $3 \times 4$ 로 나타낼 수 있다 . 일반적으로 각 프로세스에 할당하는 사각형 모양의 데이터 영역은 가능하면 정사각형 형태가 되도록 자르면 보다 효율적인 계산이 가능하다 . 이 사실은 전송되는 데이터 양이 사각형 영역의 둘레의 길이에 비례하기 때문인데 다음의 그림이 간단한 예가 될 것이다 .

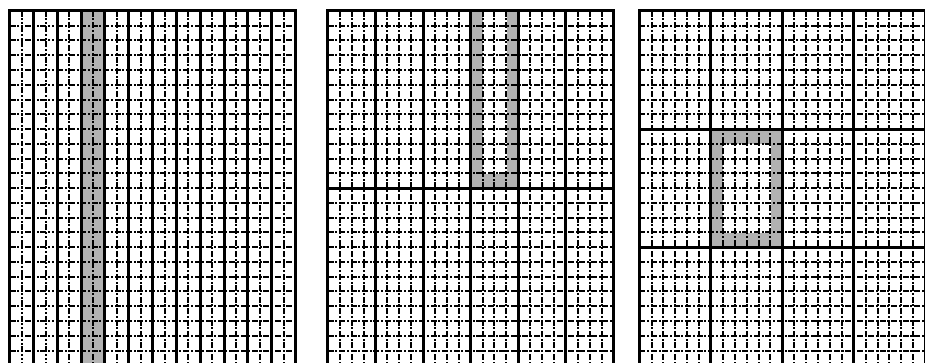


그림 3.20 여러 가지 블록 분할과 통신량

위의 그림은  $24 \times 24$  의 행렬을 12 개의 프로세스에 할당하는 3 가지 방식을 나타내고 있다. 좌측에서 우측으로 행렬은  $1 \times 12, 2 \times 6, 3 \times 4$  의 영역으로 구분이 되었으며 프로세스 마다 전송되는 최대 데이터 양은 어렵게 표시되고 있는데 48, 26, 24 개씩으로, 나뉘어진 영역이 정사각형 모양에 가까울수록 전송되는 양이 작아짐을 볼 수 있다.

### 3.3 메시지 패싱과 병렬화

앞장 “DO(for) 루프의 병렬화” 에서 보았듯 병렬화의 초점은 계산을 프로세스에 적절히 할당하여 계산속도를 빠르게 하는것이다. 그렇게 하기위하여 프로세스는 필요한 모든 데이터를 갖지 않고 계산을 하기도 한다. 이런 경우 사용자는 계산에 필요한 데이터를 메시지 패싱을 이용하여 프로세스에 전달해 주어야 하는데, 경우에 따라서 상당히 까다롭고 힘든 작업이 될 수 있다. 이 장에서는 MPI 를 이용한 병렬화 프로그램을 작성하고자하는 사용자들에게 도움이 될 수 있도록 다양한 분야에서 응용 가능한 전형적인 메시지 패싱 방식 몇 가지를 소개 한다.

#### 3.3.1 외부 데이터에 대한 참조

블록 분할을 이용한 다음 코드의 병렬화를 생각해보자.

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

두번째 루프는 특정 데이터  $a(1)$  을 참조하고 있어, 병렬화 과정에서 모든 프로세스들이 이 값을 필요로 하게 된다. 병렬 실행 코드는 다음과 같으며,  $a(1)$  을 가지고 있지 않은 프로세스는 방송을 통해 데이터를 전달 받고 있다.

```
...  
  
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  
  
...
```

### 3.3.2 1 차원 유한 차분법 (1-D Finite Difference Method)

FDM 은 편미분방정식을 풀기위해 가장 많이 사용하는 방법이다. FDM 의 가장 단순한 형태인 1 차원 FDM 을 MPI 를 이용해 병렬로 처리하는 것에 대해 알아본다.

우선 다음 프로그램은 1 차원 FDM 에서 가장 필수적인 부분을 요약해서 나타낸 것이다.

예제 3.12 1 차원 FDM : Fortran

```
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

```

예제 3.13 1 차원 FDM : C

```

/*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];

}

```

위의 프로그램을 순차적으로 실행하면 아래 그림에서 보듯 배열 **a()**가 왼쪽에서 오른쪽으로 계산이 된다 . 화살표로 데이터들의 의존성을 나타내었다 .



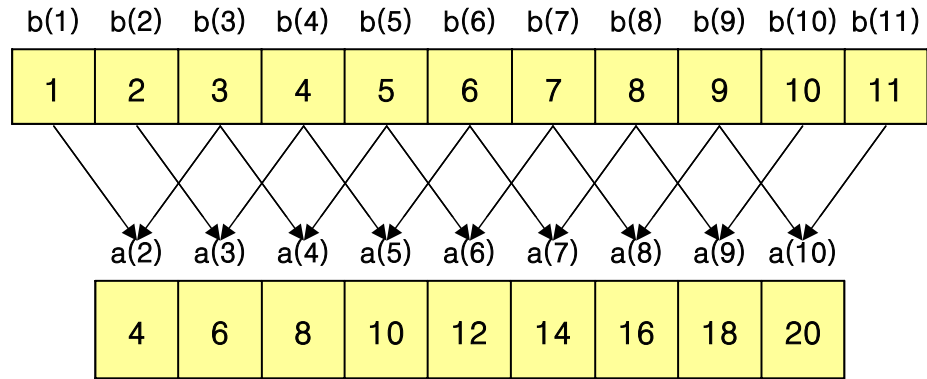


그림 3.21 1 차원 FDM 의 데이터 의존성

다음 병렬 프로그램은 순차 프로그램 3.12 와 3.13 을 병렬화 한 것이며 블록 분할로 데이터를 분배하여 1 차원 FDM 을 구현한 것이다 . 그림 3.21 의 의존성 관계에 주 의하며 살펴 보기 바란다 .

예제 3.14 병렬 1 차원 FDM : Fortran

```
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

CALL MPI_ISEND(b(iend), 1, MPI_REAL8, inext, 1, MPI_COMM_WORLD, &
               isend1, ierr)

CALL MPI_ISEND(b(ista), 1, MPI_REAL8, iprev, 1, MPI_COMM_WORLD, &
               isend2, ierr)

CALL MPI_IRECV(b(ista-1), 1, MPI_REAL8, iprev, 1, MPI_COMM_WORLD, &
               irecv1, ierr)

CALL MPI_IRECV(b(iend+1), 1, 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)

DO i = ista2, iend1

    a(i) = b(i-1) + b(i+1)

ENDDO

```

```
CALL MPI_FINALIZE(ierr)
```

```
END
```

예제 3.15 병렬 1 차원 FDM : C

```
/*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;  
  
    inext=myrank+1; iprev=myrank-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();
}

```

위의 프로그램은 논블록킹 통신 서브루틴을 이용하였으며, 서브루틴 `para_range`는 블록분할 방식으로 각 프로세스에 할당되는 반복계산의 인덱스를 계산한다 (3.2.1 참조). 다른 프로세스들은 이웃 프로세스가 두 개씩인데 반해 양쪽 끝의 프로세스 0 과 프로세스 `nprocs-1` 은 이웃하는 프로세스가 하나 밖에 없으므로 전송이 필요없는 경우가 있다는 점에 주의해야 한다. 이 두 프로세스에 대해 두번째 루프의 인덱스 `ista2`, `iend1` 은 IF 문을 이용해 따로 처리하였다. 프로세스 0 과 프로세스 `nprocs-1` 에서 데이터 전송이 필요없는 경우에 대해 송신 프로세스 또는 수신 프로세스로 `MPI_PROC_NULL` 을 지정해 모든 프로세스에 대해 동일한 형태로 메시지 송 / 수신을 처리하고 있다.

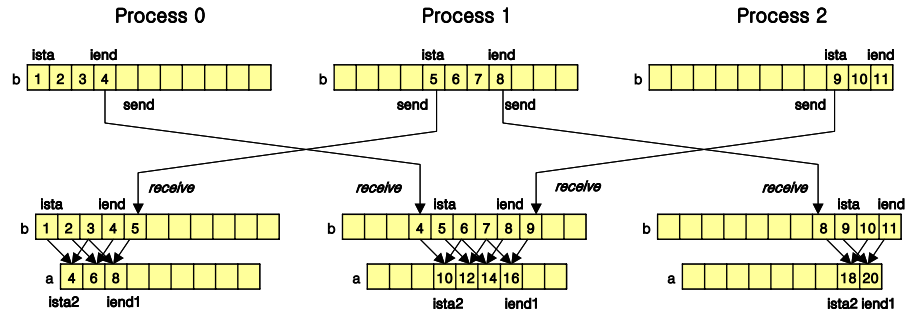


그림 3.22 병렬 1 차원 FDM 의 데이터 이동

1 차원 FDM 에서는 1 차원 배열의 경계점 데이터가 전송되고, 2 차원 FDM 에서는 2 차원 배열의 경계선의 데이터가 전송된다. 그리고, 3 차원 FDM 에서는 3 차원 배열의 경계면의 데이터들이 전송된다.

### 3.3.3 대량 데이터 전송

지금까지는 주로 각 프로세스가 가지고 있는 데이터의 경계에 위치한 일부 데이터들의 송/수신에 대해 다루었다. 여기서는 경계에 있는 데이터가 아니라 각 프로세스가 가지고 있는 전체 데이터를 특정 프로세스로 송신하는 경우(**gather**) 또는 프로세스들이 동시에 전체 배열을 갱신하는 경우(**allgather**) 또는 사용자가 블록 분할을 열 방향에서 행 방향으로 전환하여 프로세스에 재할당하는 경우 등의 대량 데이터 전송에 대해 알아 본다.

**한 프로세스로 데이터 취합 :** 때때로 ( 대개는 병렬프로그램의 마지막 부분에서 필요 ) 각 프로세스에서 병렬로 계산된 데이터를 한 프로세스로 모아야 하는 경우가 있는데, 데이터가 프로세스의 메모리에 연속적으로 저장되었는지 아닌지에 따라 그리고, 송/수신 버퍼가 중복되는지 아닌지에 따라 다음 4가지로 구분할 수 있다.

연속 데이터 ; 송 / 수신 버퍼가 중복되지 않는 경우

연속 데이터 ; 송 / 수신 버퍼가 중복되는 경우

불연속 데이터 ; 송 / 수신 버퍼가 중복되는 경우

불연속 데이터 ; 송 / 수신 버퍼가 중복되지 않는 경우

다음에서 2 차원 배열을 이용해 위의 4 가지 중 1, 2, 3 에 대해 설명할 것이다. 4 번의 경우는 앞선 1, 2, 3 의 경우를 먼저 다루게 되면 아주 간단한 문제가 되어 여기서는 다루지 않는다. 만약 전송되어야 하는 데이터가 메모리에 불연속적으로 저장되어 있다면, 앞서 5 장에서 다루었던 루틴

MPI\_TYPE\_CREATE\_SUBARRAY 를 이용해 새로운 데이터 타입을 정의해 사용하며, MPI\_GATHER ( 또는 MPI\_GATHERV, MPI\_ALLGATHER, MPI\_ALLGATHERV ) 사용시 송 / 수신 버퍼가 중복되지 않아야 하므로 중복되는 경우에 대해서는 점대점 통신루틴을 사용할 것이다

#### 경우 1. 연속 데이터 : 송 / 수신 버퍼가 중복되지 않는 경우

이 경우는 MPI\_GATHERV 를 사용해야 한다. 그림에서 배열의 데이터들은 각각 행렬의 시작점에서의 변위를 나타내고 있으므로 코드를 아래와 같이 병렬화 할 수 있다.

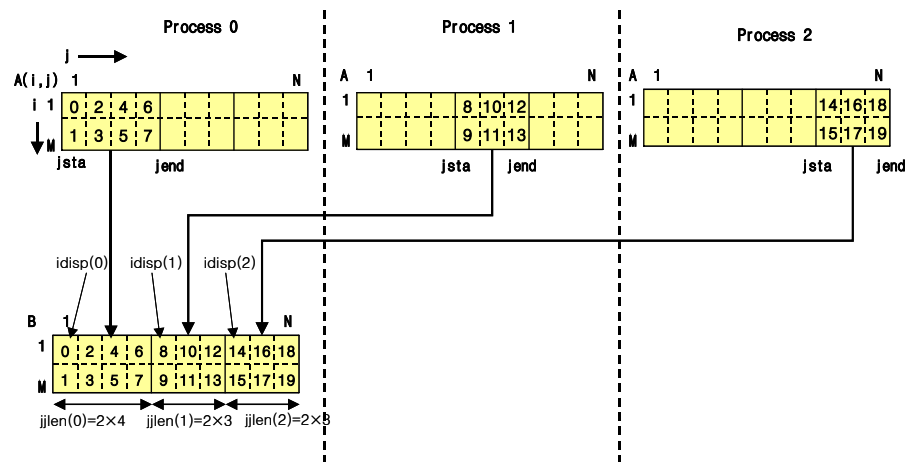


그림 3.23 한 프로세스로 데이터 취합 ( 연속 데이터, 송 / 수신 버퍼 중복 없음 ) : Fortran

연속 데이터를 송신 버퍼와 수신 버퍼가 중복되지 않도록 하며 한 프로세스로 취합하는 병렬화 코드 부분은 다음과 같다.

예제 3.16 한 프로세스로 데이터 취합 ( 연속 데이터 , 송 / 수신 버퍼 중복 없음 ) :  
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)

...
```

그림 3.23 은 Fortran 을 사용하는 경우에 대한 것이다 . 행 우선순으로 메모리에 저장하는 C 의 경우에 위의 그림은 불연속 데이터를 송 / 수신 버퍼가 중복되지 않도록해서 한 프로세스로 취합하는 예가 될것이다 . 아래 예제는 데이터가 연속적으로 저장된 경우 사용 가능한 C 코드의 일부이며 Fortran 코드와 비교해  $n \Rightarrow m$ ,  $jsta \Rightarrow ista$ ,  $jend \Rightarrow iend$ ,  $jjlen \Rightarrow iilen$  으로 각각 변경되어 사용되었음에 주의하기 바란다 .

예제 3.17 한 프로세스로 데이터 취합 ( 연속 데이터 , 송 / 수신 버퍼 중복 없음 ) : 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);

...
```

## 경우 2. 연속 데이터 : 송 / 수신 버퍼가 중복되는 경우

송 / 수신 버퍼가 메모리에서 중복되는 경우 사용자는 MPI\_GATHERV 를 사용할 수 없다 . 이때는 귀찮더라도 다음 예제와 같이 점대점 통신을 이용해야 한다 .



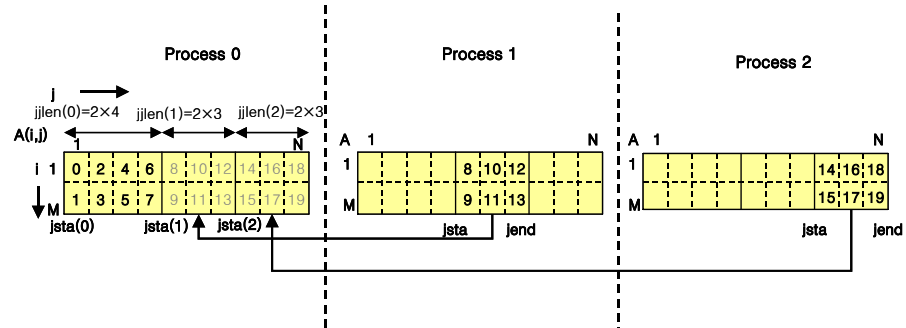


그림 3.24 한 프로세스로 데이터 취합 (연속 데이터, 송/수신 버퍼 중복) : Fortran

예제 3.18 한 프로세스로 데이터 취합 (연속 데이터, 송/수신 버퍼 중복) : 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)

```

```

...

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)

...

```

앞서와 마찬가지로 제시된 그림 3.24 는 Fortran 을 사용하는 경우에 대한 예이다 . 행 우선순으로 메모리에 저장을 하는 C 의 경우 위의 그림은 불연속 데이터를 송 / 수신 버퍼가 중복 되도록해서 한 프로세스로 취합하는 예가 될것이다 . 다음 C 코드는 Fortran 코드와 비교해  $n \Rightarrow m$ ,  $jsta \Rightarrow ista$ ,  $jend \Rightarrow iend$ ,  $jjlen \Rightarrow iilen$ ,  $jjsta \Rightarrow iista$  로 각각 변경되어 사용되었음에 주의하기 바란다 .

예제 3.19 한 프로세스로 데이터 취합 ( 연속 데이터 , 송 / 수신 버퍼 중복 ) : 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, &iista, &iend);

    iista[irank] = iista;

    iilen[irank] = n*(iend-ista+1);

}

para_range(0, m-1, nprocs, myrank, &iista, &iend);

...

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][ista], iilen[myrank], MPI_DOUBLE, 0, 1,
        MPI_COMM_WORLD, &iireq);

MPI_Wait(&iireq, &istatus);
}

free(iista);

free(iilen);

free(iireq);

...

```

### 경우 3. 불연속 데이터 : 송 / 수신 버퍼가 중복되는 경우

Fortran 을 사용하면 아래 그림과 같이 사용자가 2 차원 배열을 행 방향으로 나눌 때 각 프로세스는 메모리에 불연속적으로 데이터를 저장하게 된다 . 이렇게 불연속적으로 저장된 데이터 통신을 해야 하는 경우 새로운 유도 데이터 타입을 정의하는 것이 편리하다 .

예제 에서는 사용상의 편의를 위해 앞서 2 장의 4.4 절에서 다루었던 루틴 `MPI_TYPE_CREATE_SUBARRAY` 를 호출해 유도 데이터 타입을 만들고 , 송 / 수신 버퍼가 중복되는 경우 이므로 점대점 통신 루틴을 이용해 통신하고 있다 .

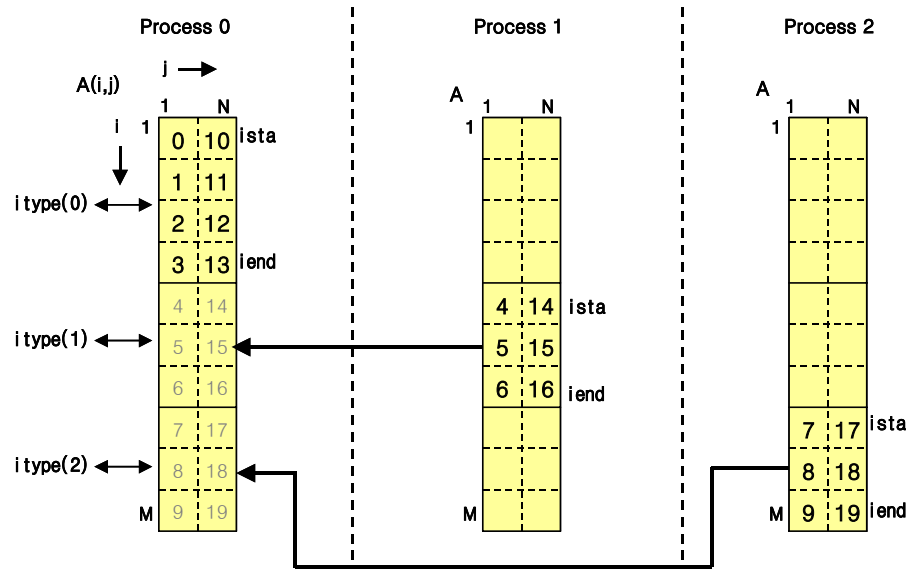


그림 3.25 한 프로세스로 데이터 취합 (불연속 데이터, 송/수신 버퍼 중복) : Fortran

예제 3.20 한 프로세스로 데이터 취합 (불연속 데이터, 송/수신 버퍼 중복) : 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; 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

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,iireq,ierr)

    CALL MPI_WAIT(iireq, istatus, ierr)

ENDIF

DEALLOCATE (itype, iireq)

...

```

예제 3.21 한 프로세스로 데이터 취합 ( 불연속 데이터 , 송 / 수신 버퍼 중복 ) : 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]);

}

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, &ireq);
    MPI_Wait(&ireq, &istatus);
}

free(itype);
free(iireq);
...

```

### 데이터 동기화 (Synchronizing Data):

프로그램의 한 지점에서 병렬 프로세스들의 모든 데이터를 동기화시켜야 하는 경우가 있다. 앞선 경우와 같이 데이터의 연속과 불연속, 송 / 수신 버퍼의 중복 여부에 따라 4 가지 경우가 있으며, 이에 따른 프로그램 작성 방법도 역시 동일하므로 여기서는 데이터가 메모리에 연속적으로 저장될 때 송 / 수신 버퍼가 중복되는 경우와 중복되지 않는 경우의 두 가지 예를 다룰 것이다.

#### 경우 1. 데이터 동기화 : 송 / 수신 버퍼가 중복되지 않는 경우

서브루틴 MPI\_ALLGATHERV 를 호출하여 처리할 수 있다.



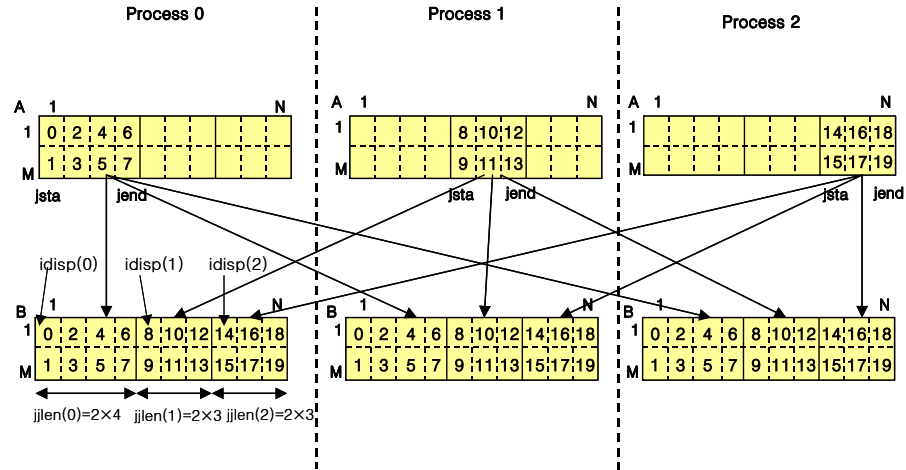


그림 3.26 데이터 동기화 (연속 데이터, 송/수신 버퍼 중복 없음) : Fortran

예제 3.22 데이터 동기화 (연속 데이터, 송/수신 버퍼 중복 없음) : 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)
...

```

예제 3.23 데이터 동기화 ( 연속 데이터 , 송 / 수신 버퍼 중복 없음 ) : 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_Allgathrev(&a[ista][0], iilen[myrank], MPI_DOUBLE, b,
               iilen, idisp, MPI_DOUBLE, 0, MPI_COMM_WORLD);

free(idisp);

free(iilen);

```

## 경우 2. 데이터 동기화 : 송 / 수신 버퍼가 중복되는 경우

MPI\_ALLGATHERV 를 사용할 수 없으므로 MPI\_BCAST 를 이용해 프로세스 개수 만큼 방송해야 한다 .

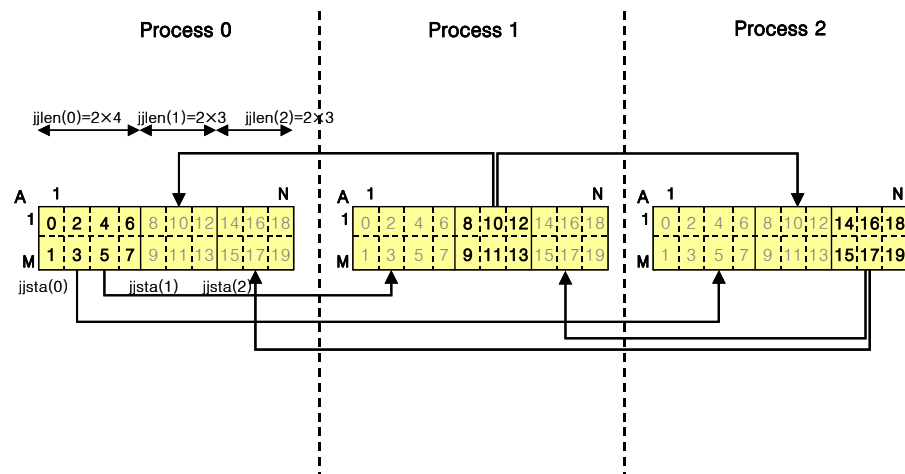


그림 3.27 데이터 동기화 ( 연속 데이터 , 송 / 수신 버퍼 중복 ) : Fortran

예제 3.24 데이터 동기화 ( 연속 데이터 , 송 / 수신 버퍼 중복 ) : 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)

...

```

예제 3.25 데이터 동기화 ( 연속 데이터 , 송 / 수신 버퍼 중복 ) : 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(0, m-1, nprocs, irank, &iista, &iend);

    iista[irank] = iista;

    iilen[irank] = n*(iend-ista+1);

}

para_range(0, m-1, nprocs, myrank, &iista, &iend);

...

```

```

for(irank = 0; irank<nprocs; irank++){

    MPI_BCAST(&a[iista[irank]][0], iilen[irank], MPI_DOUBLE,

              irank, 1, MPI_COMM_WORLD);

}

free(iista);

free(iilen);

...

```

**블록 분할의 전환 (Transposing Block Distribution):** 여기서는 계산 도중에 열 방향의 블록 분할을 행 방향의 블록 분할로 변환하는 방법에 대해 설명한다. 이러한 예는 2 차원 FFT 코드를 병렬화 하는 경우 볼 수 있는데 다음 그림은 처음에 열 방향의 블록 분할로 3 개의 프로세스에 나누어 배치된 2 차원 행렬 a() 를 행 방향의 블록 분할로 변환한 결과를 나타내고 있다.

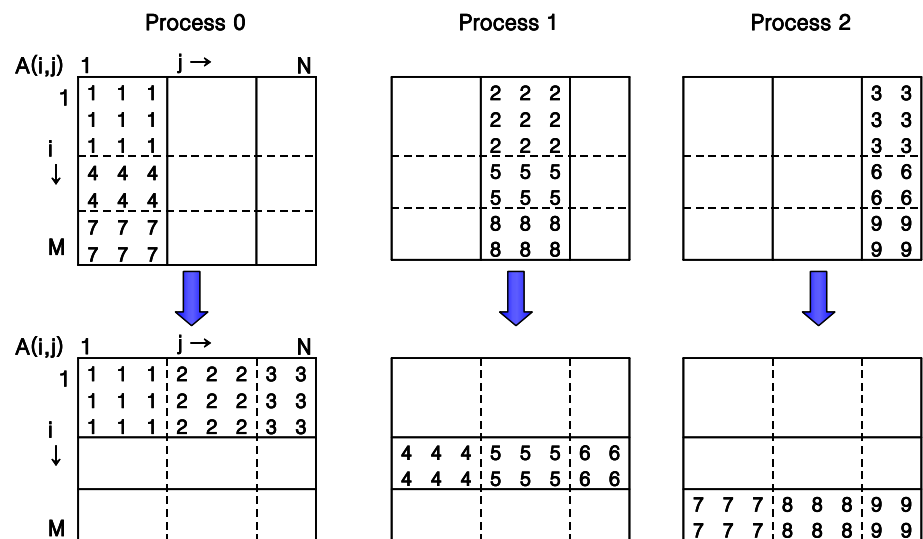


그림 3.28 블록 분할의 전환

블록 분할 전환을 위해서는 다음 그림 3.29 와 같이 우선 행렬을 9 개의 블록으로 나누어 이 블록을 단위로 데이터를 전송 하는 것이 편리하다 . 다음 예제에서는 각 블록에 대한 유도 데이터 타입을 다음과 같이 정의해 사용한다 .

|          |                |            |                 |            |
|----------|----------------|------------|-----------------|------------|
| $A(i,j)$ |                | 1          | $j \rightarrow$ | N          |
| 1        | $i \downarrow$ | itype(0,0) | itype(0,1)      | itype(0,2) |
|          |                | itype(1,0) | itype(1,1)      | itype(1,2) |
| M        |                | itype(2,0) | itype(2,1)      | itype(2,2) |

그림 3.29 블록 분할 전환을 위해 정의된 유도 데이터 타입

예제 3.26 블록 분할의 전환 : 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_STATUS_SIZE(MPI_COMM_WORLD, myrank, ierr)

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)

...

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

...

```

예제 3.27 블록 분할의 전환 : 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);

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);

...

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);

    }

}

```

### 3.3.4 중첩 (Superposition)

서브루틴 MPI\_REDUCE 와 MPI\_ALLREDUCE 는 각 프로세스에 흩어져있는 데이터들을 모아 연산을 수행하는데 이용 되지만 , 다음과 같이 단지 흩어진 데이터를 한 프로세스로 모으는데에도 이용될 수 있다 .

행 방향 순환 분할로 데이터가 프로세스에 분배되어 있는 행렬을 가정하자 . 각 프로세스는 수축되지 않고 배열 전체를 메모리에 저장해 두고 있다 . 각 프로세스는 할당된 데이터들에 대해 계산을 수행하고 사용자는 그 결과를 프로세스 0 에 모두 모으고 싶다 . 먼저 생각해 볼 수 있는 방법으로는 불연속적 (Fortran) 으로 분포된 데이터 처리를 위해 유도 데이터 타입을 정의하고 프로세스 0 으로 송신하거나, 송신 프로세스에서 불연속적인 데이터를 연속적인 위치로 모아 그 데이터를 프로세스 0 으로 송신하고 수신 프로세스는 그 데이터를 받아 다시 적절한 위치로 풀어 놓는 방법 등을 생각해 볼 수 있다 .

이러한 문제에 대해 각 프로세스에서 할당받은 데이터를 제외한 나머지 데이터를 0 으로 두고 루틴 MPI\_REDUCE 를 이용하면 의외로 문제를 간단하게 해결할 수 있다 . 다음 코드와 그림 3.30 을 보자 .

```

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)

...

```

아래 그림은 위의 방법을 이용하여  $n=8$ ,  $nprocs=3$  인 경우 어떻게 데이터를 취합하는지 보여주고 있다 . 이런 중첩 방식이 비록 구현이 쉽고 간단하기는 하지만 통신 부담이 큰 경우에는 사용하지 않는 것이 좋다 . 왜냐하면 각 프로세스가 송신해야 하는 데이터의 개수가  $n*n/nprocs$  개가 아니라  $n*n$  개로 많아져 통신 부하가 커지기 때문이다 .

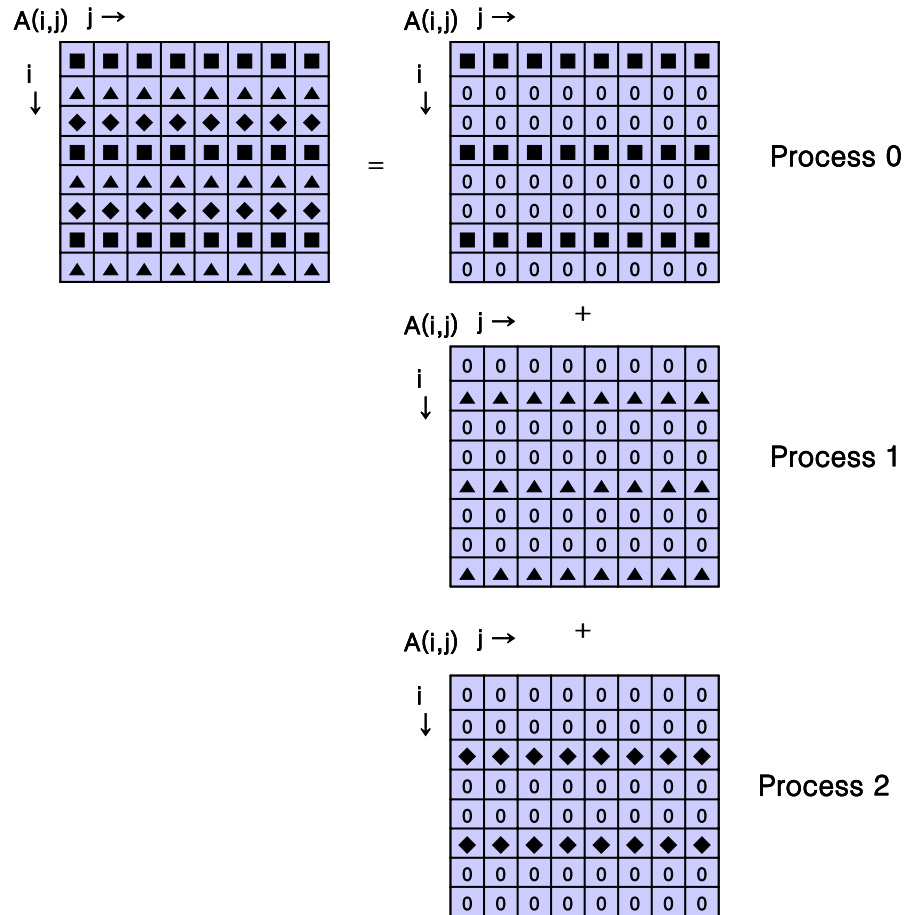


그림 3.30 중첩

### 3.3.5 파이프라인 방법

파이프 라인 방법은 루프 실행에 의존성을 가지는 루프를 병렬화 하는 기술이다 . 의존성은 루프 실행이 이전 계산 결과가 나와야 다음 계산이 가능하도록 되어있어 엄격히 순서대로 루프가 실행되어야 하는 것을 말한다 . 다음 루프는 각 실행이 순서대로 이루어져야 하는 의존성을 가지는 전형적인 모양이다 .

DO  $i = 1, n$

$x(i) = x(i-1) + 1.0$

ENDDO

이 루프는 가령  $x(2)$  의 계산은  $x(1)$  의 계산이 끝나야 하고 , 마찬가지로  $x(3)$  의 계산은  $x(2)$  의 계산이 끝나야 가능하다 .

이러한 의존성은 프로그램의 병렬화를 어렵게한다 . 다음 프로그램을 보자 .

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

...

이 프로그램에 포함된 의존성이 다음 그림 3.31 에 나타나 있다 . 그림에서 화살표의 방향으로 계산이 진행되어야 한다 . 프로그램을 그림에서 처럼 열 방향의 블록 분할을 이용해 병렬처리하고자 하면 각 프로세스는 좌측 경계의 데이터를 필요로 하게된다 . 게다가 그 데이터는 좌측의 프로세스가 갱신하고난 후에 사용해야 한다는 어려움이 있다 .

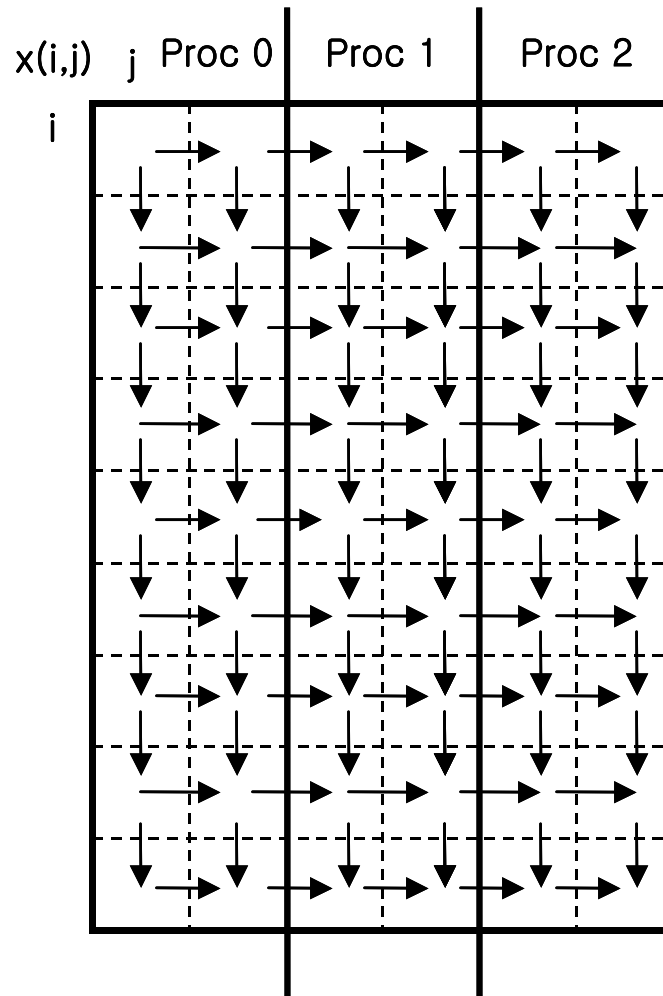


그림 3.31 프로그램 main 의 의존성

다음 프로그램 main2 의 의존성은 그림 3.32 에 나타나 있으면 앞서의 main 보다 덜 복잡한 의존성을 가지고 있다. 만약 프로그램의 다른 부분들을 무시해 버린다면, 행렬을 행 방향으로 잘라 의존성을 없앨수도 있으나, 프로그램의 다른 부분들과 관련하여 성능 등의 문제로 반드시 열 방향의 블록 분할을 사용해야 하는 경우를 가정한다면 main2 도 여전히 의존성을 가지는 프로그램이다.

```
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

...
```

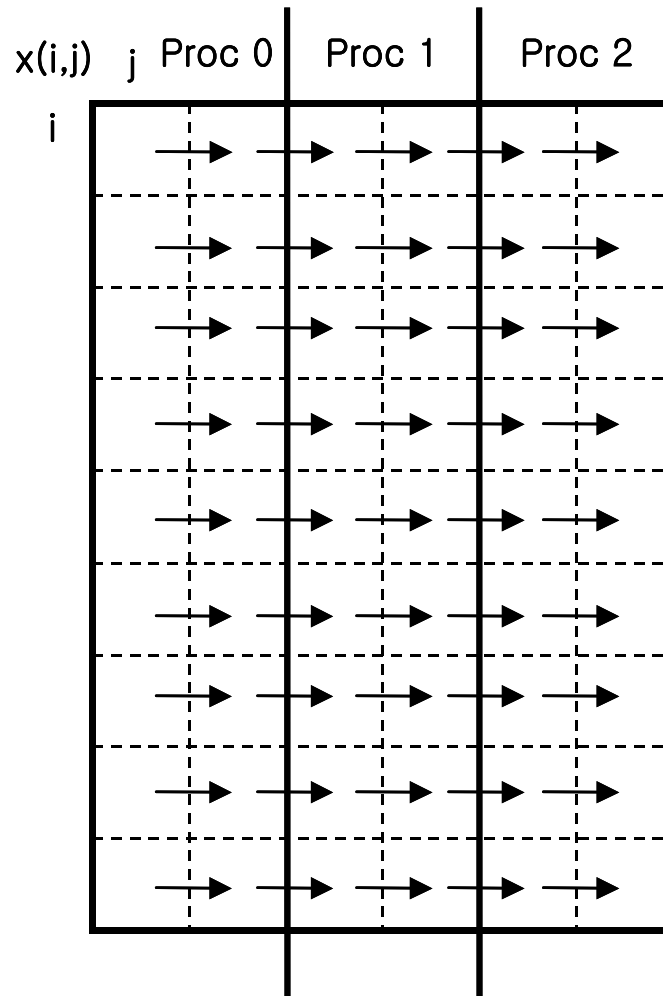


그림 3.32 프로그램 main2 의 의존성

파이프라인 방법은 이렇게 의존성을 가진 프로그램을 병렬화하기 위해 사용된다. 이 방법의 핵심적인 생각은, 그림 3.31 또는 3.32 처럼 세 개의 프로세스에 행렬  $x()$  가 열 방향의 블록 분할로 할당되어 있다면, 할당된 데이터를 다시 행 방향으로 블록화 하는 것이다 ( 그림 3.33 참조 ). 그림 3.33 에서는 행 방향 블록의 크기를 2 로 두었다 ( $iblock=2$ ). 의존성 때문에 각 프로세스는 행렬의 데이터를 동시에 갱신할 수 없으므로, 우선 프로세스 0 이 좌측의 블록 1 을 먼저 계산한다. 그리고, 경계의 데이터  $iblock$  개를 블록 2 를 계산하도록 프로세스 1 로 송신하고, 프로세스 1 과 프



프로세스 0 은 동시에 블록 2 를 계산한다 . 블록 2 의 계산이 끝나면 프로세스 1 은 경계 데이터를 프로세스 2 로 프로세스 0 은 경계 데이터를 프로세스 1 로 보내고 , 프로세스 0, 1, 2 는 동시에 블록 3 을 계산한다 . 같은 방식으로 블록 4 와 블록 5, 6 이 차례 차례 계산된다 . 각 블록에 크게 쓰여진 숫자는 그 블록이 계산되는 순서를 의미하며 , deg 는 프로그램 실행 중 동시에 가동되는 프로세스의 개수 (degree of parallelism) 를 나타내고 있다 .

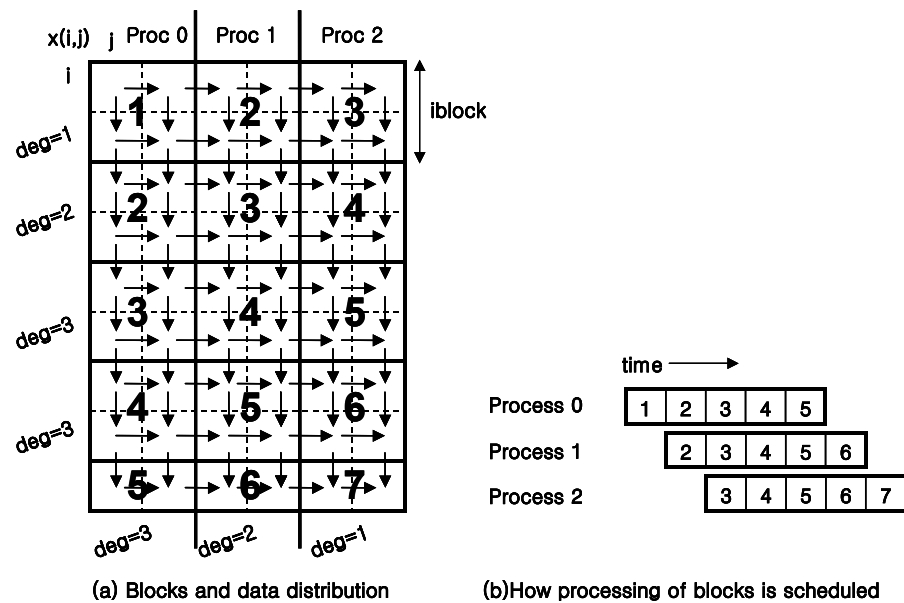


그림 3.33 파이프 라인 방법에 의한 병렬화 : Fortran

다음은 앞서의 프로그램 main 을 파이프라인 방식으로 병렬화 시킨 것이다 . 각 프로세스는 왼쪽 이웃으로부터의 데이터 전송을 기다려서 계산을 하고 다시 오른쪽 이웃으로 경계값을 보낸다 .

예제 3.28 파이프 라인 방법에 의한 병렬화 : 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_STATUS_SIZE(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

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

...

```

예제 3.29 파이프 라인 방법에 의한 병렬화 : 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;

    for(jj=1; jj<=my; jj+=jblock){

        jblklen = min(jblock, my-jj+1);

```

```

MPI_Irecv(&x[jj][ista-1], 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[jj][iend], jblklen, MPI_DOUBLE, inext, 1,
          MPI_COMM_WORLD, &ireqs);

MPI_Wait(&ireqs, &istatus);

}

...

```

예제에서는 서브루틴 MPI\_WAIT가 MPI\_Irecv, MPI\_Isend와 붙어서 사용되고 있는데 이러한 사용은 교착의 위험이 있는 블록킹 통신과 같은 기능을 수행한다. 그러나, 위와 같은 파이프 라인 방법에서는 교착 (deadlocks)에 빠질 위험성은 없다. 프로세스 0이 수신없이 송신을 수행하므로 데이터 흐름이 다음과 같이 이루어지기 때문이다.

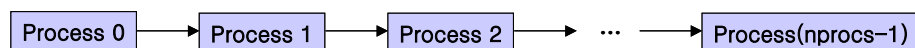


그림 3.34 파이프 라인 방법의 데이터 흐름

만약 프로세스 0 이 마지막 프로세스 ( 프로세스  $nprocs-1$ ) 로부터 데이터를 받아야 하는 닫힌 토폴로지를 가지는 경우는 교착이 발생하기 때문에 파이프 라인 방법 대신 다음 절에 다루게 되는 비틀림 분해 (twisted decomposition) 방식을 써야한다.

파이프라인 방법에서 유의할 점중 한가지는 블록의 크기 (iblock) 를 결정하는 것이다. 다음 그림 3.35 에서는 블록의 크기에 따라 동시에 실행되는 프로세스의 개수 (degree of parallelism 이라고 함) 를 최대로 하여 계산할 수 있는 영역을 보여주고 있다. 그림에서 볼 수 있듯 블록의 크기를 작게 잡으면 많은 부분을 최대한의 프로세스를 사용하여 병렬로 처리할 수 있지만 통신을 해야 하는 부분이 많아져 통신 부하가 증가하게 된다. 따라서 사용자는 최대 성능이 나올 수 있도록 적절한 선에서 블록의 크기를 결정할 필요가 있다.

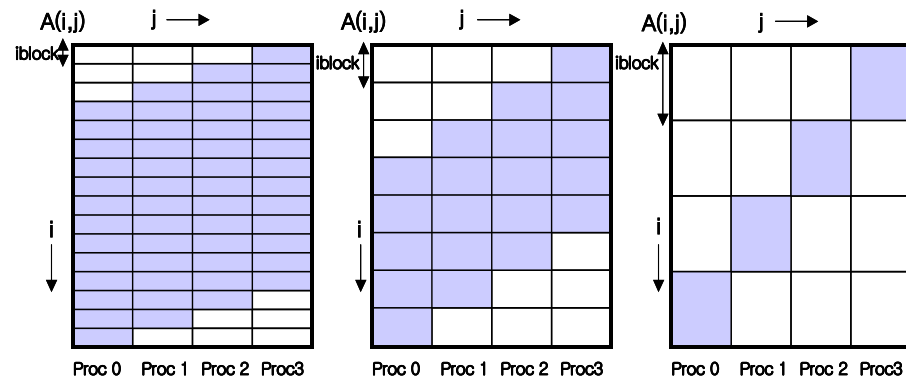


그림 3.35 파이프 라인 방법과 블록 크기

### 3.3.6 비틀림 분해 (The Twisted Decomposition)

다음 프로그램에서 의존성을 가진 루프 A 와 루프 B 의 병렬화를 생각해 보자.

...

! 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
...

```

앞 절에서 설명한 파이프라인 방법을 이용해도 좋지만 성능면에서 비틀림 분해 방법이 유용하게 쓰일수 있다 . 비틀림 분해 방법은 파이프라인 방법과 달리 프로세스들이 대기 시간없이 동시에 계산을 시작하므로 성능면에서 유리하고 보다 나은 작업 부하 균형을 제공한다 . 단 , 데이터가 프로세스에 복잡하게 분배되므로 사용자가 프로그램을 작성하기 어렵다는 단점이 있다 .

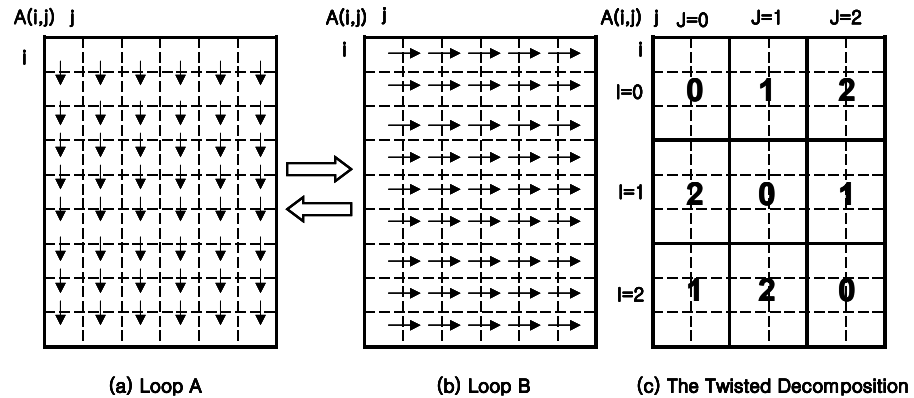


그림 3.36 비틀림 분해

비틀림 분해를 위해 우선 2 차원 배열  $x()$  의 행과 열을 각각  $nprocs$  ( 프로세스 개수 ) 개의 블록으로 나누어 전체를  $nprocs^2$  개의 블록으로 구분하고 , 블록의 위치를 좌표  $(I,J)$  로 나타낸다 . 좌표  $(I,J)$  에 해당하는 블록은  $I \cdot J + nprocs$  를  $nprocs$  로 나눈 나머지에 해당하는 프로세스에 할당 시켜준다 ( 그림 9.17 참조 ) . 다음 블록의 계산으로 넘어가기전에 각 프로세스는 경계 부분의 데이터를 이웃 프로세스들과 주고 받아야 하는데 위와 같은 방식으로 데이터를 할당하게 되면 의존성에 무관한 블록끼리 병렬로 처리할 수 있다 . 실제 계산에서 루프 A 는  $I=0$  인 블록 세 개를 먼저 병렬로 계산하고 다음  $I=1, I=2$  의 블록을 계산하는 순서로 진행되고 , 루프 B 는 블록 좌표  $J$  순으로 계산이 진행된다 . 블록 좌표계의 모든 행과 열은 각각 모든 프로세스를 포함하게 되므로 비틀림 분해에서는 파이프 라인 방법과 달리 항상 모든 프로세스가 참여하는 병렬화가 가능 하게 된다 .

예제 3.30 비틀림 분해에 의한 병렬화 : 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_STATUS_SIZE(MPI_COMM_WORLD, myrank, ierr)

ALLOCATE (is(0:nprocs-1), ie(0:nprocs-1))

ALLOCATE (js(0:nprocs-1), je(0:nprocs-1))

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

    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

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

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

...

```

예제 3.31 비틀림 분해에 의한 병렬화 : 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));

    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;

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);

}

}

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++){

        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++) bufs[i]=x[i][jend];

        MPI_Isend(&bufs[ista], ilen, MPI_DOUBLE, iprev, 1,
            MPI_COMM_WORLD, &ireqs);

    }

}

free(is); free(ie);

free(js); free(je);

...

}

```

예제에서 MPI\_ISEND에 대응하는 MPI\_WAIT의 위치에 주의하자. 이는 교착을 피하기 위한 것이다. 예를 들어 루프 B의 경우를 보면, 다음과 같은 구조로 되어있다.

```
! Loop B

DO iy = 0, nprocs-1

  ...

  IF (ix /= 0) THEN

    Receive

    Wait for receive to complete

    Wait for send to complete

  ENDIF

  ...

  IF (iy /= nprocs-1) THEN

    send

  ENDIF

ENDDO
```

파이프라인 방법과 달리 교착에 주의해야 하는 것이 다음 그림을 보면 명확해진다. 비틀림 분해에서 어떤 프로세스는 항상 특정한 다른 프로세스로부터 데이터를 받는 형태의 토폴로지로 데이터 흐름이 이루어진다.

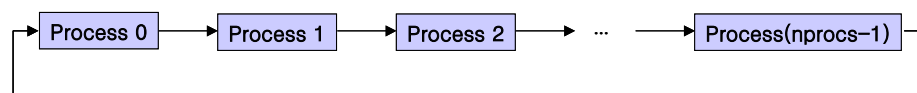


그림 3.37 비틀림 분해의 데이터 흐름

### 3.3.7 프리픽스 합

앞서 살펴본 파이프라인 방법과 비틀림 분해 방법에서 고려되었던 루프들은 모두 내포된 루프들이었기 때문에 병렬화가 가능하였다. 만일 내포되지 않은 루프 실행이 의존성을 가지고 있다면 어떻게 병렬화 할 수 있는지 알아보자.

다음과 같이 의존성을 가지는 1 차원 배열  $a()$  는 데이터 분배에 의해서 병렬화 성능 향상을 기대할 수 없으며, 파이프라인 방법을 쓰게 되면 단순한 순차처리가 될 뿐이다.

```
PROGRAM main
```

```
PARAMETER (n = ...)
```

```
REAL a(0:n), b(n)
```

```
...
```

```
DO j = 1, n
```

```
    a(i) = a(i-1) + b(i)
```

```
ENDDO
```

```
...
```

위 프로그램은 프리픽스 합 (prefix sum) 의 방법을 이용해 병렬로 처리할 수 있다.

배열  $a()$  의 각 원소는 다음과 같은 값을 가진다.

$$a(1) = a(0) + b(1)$$

$$a(2) = a(0) + b(1) + b(2)$$

$$a(3) = a(0) + b(1) + b(2) + b(3)$$

...

$$a(n) = a(0) + b(1) + b(2) + b(3) + \cdots + b(n)$$

여기서 연산이 반드시 덧셈일 필요는 없으며, 일반적으로 다음과 같은 형태를 가지는 루프에 대해 프리픽스 합의 방법을 이용할 수 있다. 여기서 행렬  $a()$ 의 데이터 타입과 연산  $op$ 의 조합은 루틴  $MPI\_REDUCE$ (또는  $MPI\_SCAN$ )의 경우와 동일하게 이용가능하다.

DO j = 1, n

$a(i) = a(i-1) \text{ op } b(i)$

ENDDO

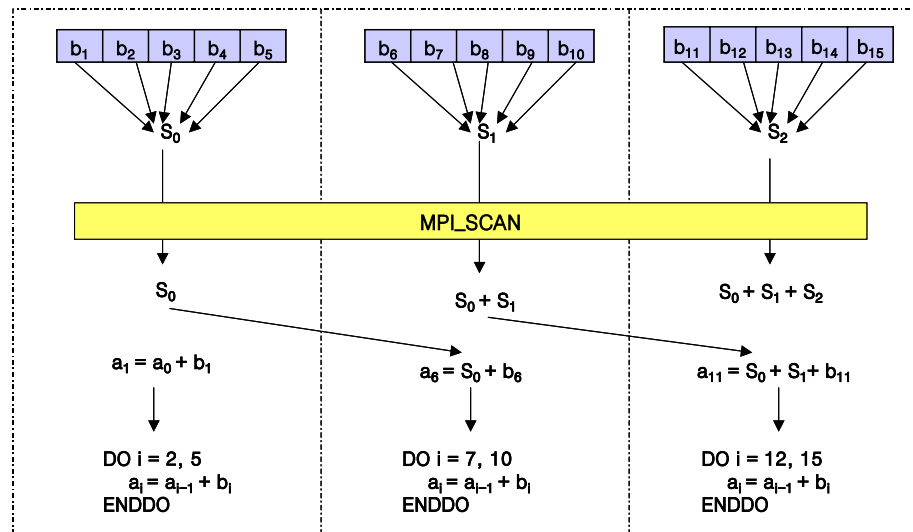


그림 3.38 프리픽스 합

배열  $a()$ 와  $b()$ 는 프로세스들에 블록 분할로 분배되며 각 프로세스는 배열  $b()$ 의 부분합을 계산한다. 이 때 부분합의 프리픽스 합이 루틴  $MPI\_SCAN$ 에 의해 계산되고, 그것이 각 프로세스에서 배열  $a()$ 의 첫 데이터를 계산하는데 이용된다. 그림 3.38에서처럼 이 부분합은 필요한 프로세스로 전송되어도 좋고 또는 다음 프로그램에서처럼 간단하게 계산을 통하여 자체적으로 구할 수도 있다.



아래의 병렬화 프로그램을 보면 DO 루프가 두 번 들어가 전체 덧셈의 횟수는 대략 순차프로그램의 두배 정도 된다. 따라서 프로세스의 개수가 작다면 이 방법 ( 프리픽스 합 ) 은 성능면에서 효과적이지 못할 수도 있음을 알아두자.

예제 3.32 프리픽스 합을 이용한 병렬화 : 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_STATUS_SIZE(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

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

...

```

예제 3.33 프리픽스 합을 이용한 병렬화 : 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];

    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 프로그램 예제

이 단원에서는 2 차원 유한 차분법 , 분자 동역학 등의 실제 계산 문제들에서 어떻게 MPI 가 사용되는지 살펴본다 .

### 4.1 2 차원 유한 차분법

앞서 다루었던 1 차원 유한 차분법 (FDM) 의 병렬화 과정을 좀더 일반화 시켜 2 차원 FDM 을 병렬화 시키는 방법을 알아보자 . 우선 고려할 점은 전송할 데이터의 양과 연속성을 고려하여 어떻게 행렬을 프로세스에 분배시킬것인가 하는 것이다 .

2 차원 FDM 에서 병렬화 시켜야 할 핵심이 되는 부분은 다음과 같다 .

예제 4.1 2 차원 FDM 순차 프로그램 : Fortran

...

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

...

예제 4.2 2 차원 FDM 순차 프로그램 : C...

```
#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]
```

```
...
```

```
}
```

2차원 FDM 프로그램은 양쪽 방향의 의존성을 모두 가지고 있다. 앞서 1차원 FDM에서 행렬의 모양에 따라 데이터 분배를 행 방향으로 할 것인가 또는 열 방향으로 할 것인가를 결정 해야 했던 것처럼 2차원 FDM에서는 행 방향 블록 분할, 열 방향 블록 분할 그리고 양 방향 블록 분할 중 행렬의 모양에 따라 통신량을 최소화하도록 하는 데이터 분배방식을 선택해야 한다.

### 4.1.1 열 방향 블록 분할

Fortran 에서 열 방향 블록 분할을 하면 프로세스들 사이의 경계에 있는 데이터들이 메모리에서 연속적으로 위치하게 된다 . 그러나 , C 를 사용한다면 열 방향 블록 분할에서 경계 데이터는 불연속적으로 위치하게 될 것이다 . 경계 데이터가 불연속적인 상황은 다음 4.1.2 절의 행 방향 블록 분할에서 다루게 되는데 , C 예제 4.4 는 4.1.2 절의 내용과 그림 4.2 를 참고로 한 것이다 .

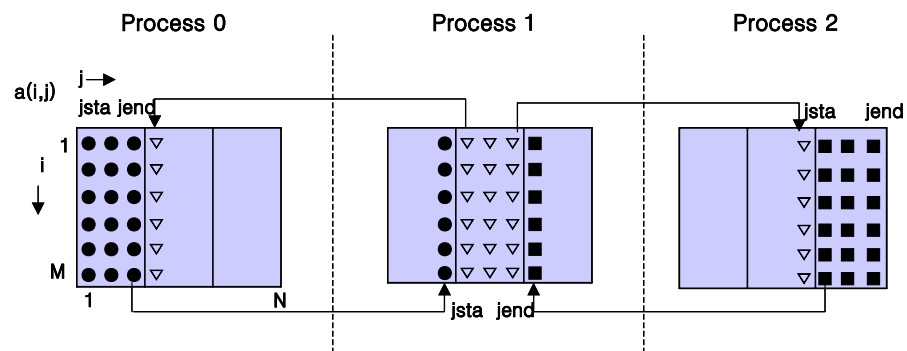


그림 4.1 2 차원 FDM : 열 방향 블록 분할

다음의 2 차원 FDM 병렬 프로그램은 각 프로세스마다 서브루틴 `para_range` 를 이용해 범위를 계산하고있다 .

예제 4.3 열 방향 블록 분할을 이용한 2 차원 FDM 병렬 프로그램 : Fortran

```
PROGRAM parallel_2D_FDM_column
```

```
INCLUDE 'mpif.h'
```

```
IMPLICIT REAL*8 (a-h,o-z)
```

```
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

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_REAL8, inext, 1, &
    MPI_COMM_WORLD, isend1, ierr)

CALL MPI_ISEND(a(1,jsta), m, MPI_REAL8, iprev, 1, &
    MPI_COMM_WORLD, isend2, ierr)

CALL MPI_IRECV(a(1,jsta-1), m, MPI_REAL8, iprev, 1, &
    MPI_COMM_WORLD, irecv1, ierr)

```

```

CALL MPI_Irecv(a(1,jend+1), m, 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)

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

```

예제 4.4 열 방향 블록 분할을 이용한 2 차원 FDM 병렬 프로그램 : C

```

/*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);

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];

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];

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();

}

```

#### 4.1.2 행 방향 블록 분할

Fortran 에서 행 방향 분할을 하면 프로세스들 사이의 경계에 있는 데이터들이 메모리상에서 연속적이지 않다 . 따라서 사용자는 불연속적인 데이터를 송 / 수신하기

위하여 유도 데이터 타입을 이용하거나 데이터를 묶고 송 / 수신하고 다시 풀어주는 코드를 직접 작성해야 한다 . 코드작성과 읽기가 쉽다는 측면에서 유도 데이터 타입의 사용이 권장되지만 때때로 성능측면에서 사용자가 직접 코드를 작성하는 것이 유리할 수도 있다 . 다음의 예제는 유도 데이터 타입 을 사용하지 않고 일차원 배열 works1(), works2, workr1(), workr2()를 이용해 사용자가 직접 불연속적인 데이터를 처리하는 예를 보여준다 .

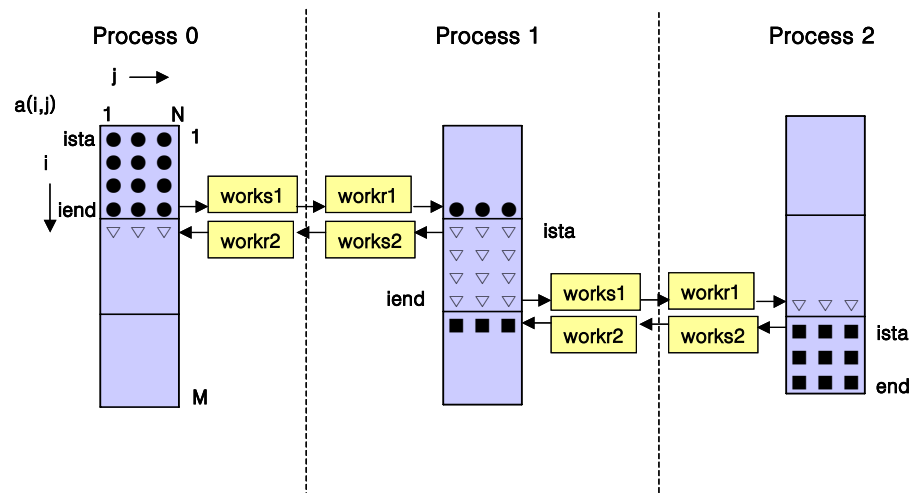


그림 4.2 2 차원 FDM : 행 방향 블록 분할

예제 4.5 행 방향 블록 분할을 이용한 2 차원 FDM 병렬 프로그램 : Fortran

```
PROGRAM parallel_2D_FDM_row

INCLUDE 'mpif.h'

IMPLICIT REAL*8 (a-h,o-z)

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

IF (myrank == nprocs - 1) inext = MPI_PROC_NULL

IF (myrank == 0) iprev = MPI_PROC_NULLDO 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

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)

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

```

C 를 사용한다면 행 방향 분할은 프로세스들 사이의 경계에 있는 데이터들이 4.1.1 절의 그림 4.1 에서 볼 수 있듯 메모리상에서 연속적으로 위치한다 .

예제 4.6 행 방향 블록 분할을 이용한 2 차원 FDM 병렬 프로그램 : C

```

/*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);

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);

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();
}

```

#### 4.1.3 양 방향 블록 분할

양 방향으로 블록 분할을 하는 경우는 통신부하를 줄이기 위해 전송되는 데이터 양이 가능한 작아지도록 해야 하므로 사용자는 미리 행렬의 크기와 프로세스의 개수를 고려하여야 한다. 따라서, 양 방향 블록 분할을 이용할 때는 사용하는 프로세스의 개수를 사용자가 미리 알고 있는 것이 좋다. 다음 예제는 데이터를 9 개의 프로세스에 양 방향 블록 분할한 2 차원 FDM 병렬화 프로그램이다.



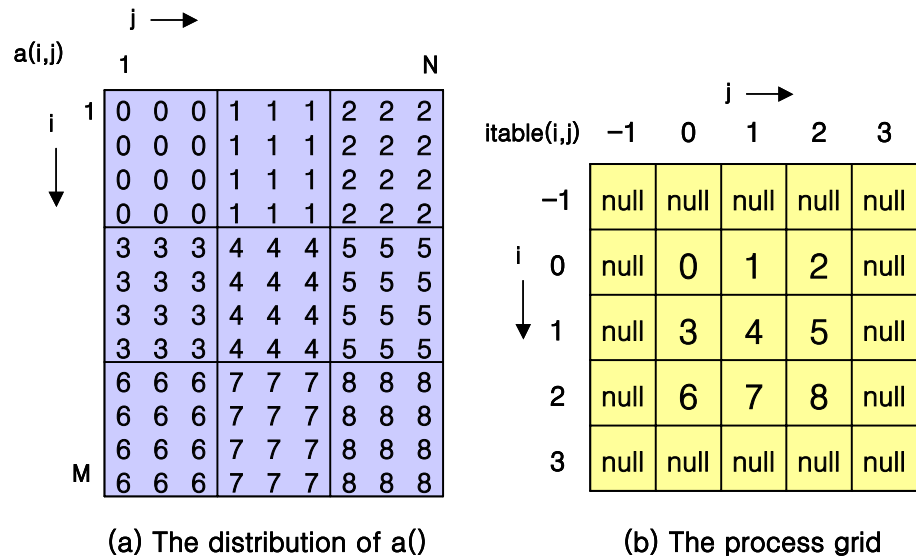


그림 4.3 양 방향 블록 분할과 프로세스 그리드

행렬 a() 는 그림 10.3 의 (a) 와 같이 양 방향으로 나누어 9 개의 프로세스에 할당한다. 그림에서 각 영역에 적힌 수는 그 영역이 할당된 프로세스의 랭크를 나타낸다. 그리고, 인접 프로세스에 보다 손쉽게 접근하기 위해 그림 (b) 와 같은 프로세스 그리드 itable() 을 준비한다. 그림에서 null 은 프로그램상에서 MPI\_PROC\_NULL 이 할당됨을 의미하고 따라서 그 영역과는 실질적인 데이터 전송이 이루어 지지 않게 된다. 각 프로세스는 프로세스 그리드 상에서 좌표 (myranki, myrankj) 를 가진다. 즉, 프로세스 7 번의 좌표는 (2,1) 이 된다. 이렇게 프로세스 그리드를 사용함으로써 이웃 프로세스로의 접근은 itable(myrank ± 1, myrank ± 1) 을 이용해 쉽게 이루어 지게 된다.

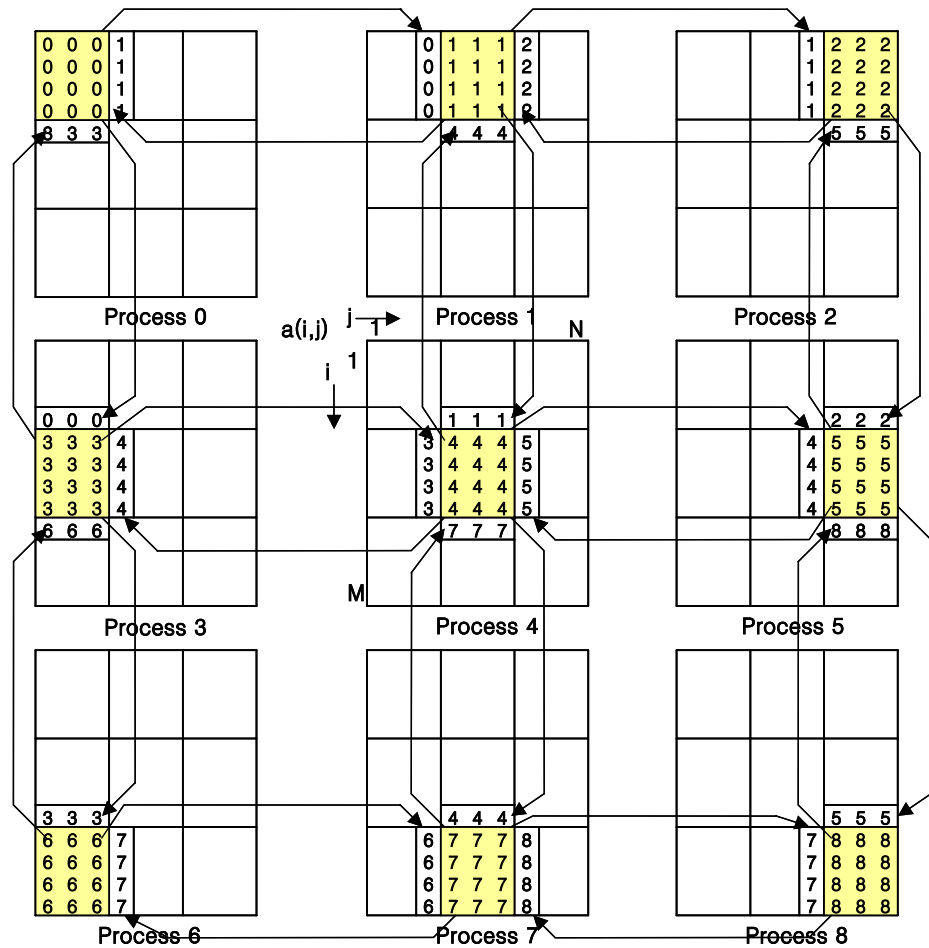


그림 4.4 2 차원 FDM : 양 방향 블록 분할

Fortran 과 C 는 불연속 경계 데이터와 연속 경계 데이터 방향이 서로 다르므로 그 점에 유의하여 예제를 참고하기 바란다.

예제 4.7 양 방향 블록 분할을 이용한 2 차원 FDM 병렬 프로그램 : Fortran

```
PROGRAM parallel_2D_FDM_both
```

```
INCLUDE 'mpif.h'
```

```

IMPLICIT REAL*8 (a-h, o-z)

PARAMETER (m = 12, n = 9)

DIMENSION a(m,n), b(m,n)

DIMENSION works1(n), workr1(n), works2(n), workr2(n)

INTEGER istatus(MPI_STATUS_SIZE)

PARAMETER (iprocs = 3, jprocs = 3)

INTEGER itable(-1:iprocs, -1:jprocs)

CALL MPI_INIT(ierr)

CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)

CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

IF(nprocs /= iprocs*jprocs) THEN

    PRINT *, '== ERROR ==' STOP

ENDIF

DO j = -1, jprocs

    DO i = -1, iprocs

        itable(i,j) = MPI_PROC_NULL

    ENDDO

ENDDO

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

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

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)

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

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

```

예제 4.8 양 방향 블록 분할을 이용한 2 차원 FDM 병렬 프로그램 : C

```

/*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);

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;

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];

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); 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];

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();
}

```

## 4.2 몬테카를로 방법

이 절에서는 몬테카를로 방법의 예로 2 차원 임의 행로에 대한 병렬화 방법에 대해 알아 보는데, 그 주된 논의는 난수 (random number) 발생에 관한 것이 될 것이다. 다음 프로그램은 100000 개 입자들이 열 걸음의 임의 행로를 전산실험하여 입자들이 이동한 거리에 대한 분포를 구하는 것이다.

예제 4.9 2 차원 임의 행로 : Fortran

```
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

```

예제 4.10 2 차원 임의 행로 : C

```

/*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]);

}

}

```

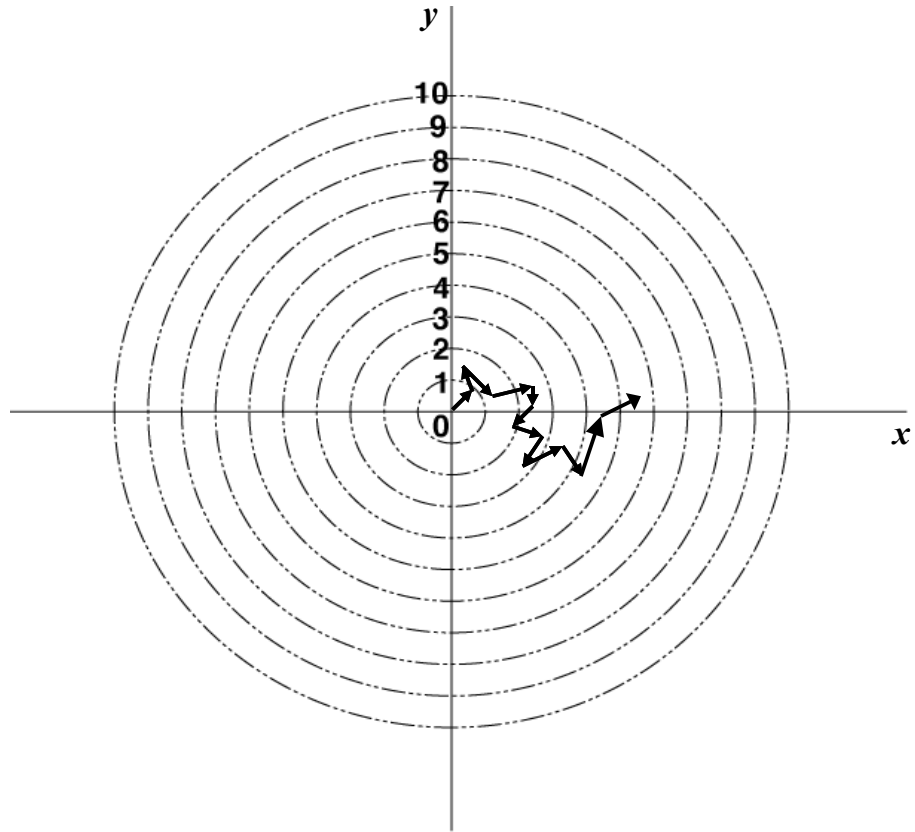


그림 4.5 2 차원 임의 행로

이와 같은 임의 행로 전산실험은 입자들을 프로세스에 분배함으로서 다음과 같이 손쉽게 병렬화할 수 있다. 프로세스마다 난수의 시드 (seed) 를 다르게 주고 있음에 주의하자. 따라서, 병렬실행에서 사용하고 있는 난수의 순서가 순차실행과는 다르므로 병렬실행의 결과와 순차실행의 결과가 일치하지 않을 수 있다.

예제 4.11 2 차원 임의 행로 병렬 프로그램 : Fortran

```
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)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

```

예제 4.12 2 차원 임의 행로 병렬 프로그램 : C

```

/*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;

    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_INTEGER, MPI_SUM, 0,

           MPI_COMM_WORLD);

for(i=0; i<10; i++){

    printf( " %d :", i);

    printf( " total = %d\n" ,iitotal[i]);

}

MPI_Finalize();

}

```

## 4.3 분자 동역학 (Molecular Dynamics)

분자 동역학 프로그램은 시간 진행을 나타내는 가장 바깥쪽의 루프가 있고 그안의 일부 루프가 대부분의 CPU 시간을 소비하는 형태로 구성된다.

1 차원에서 상호작용하는  $n$  개의 입자를 전산실험 하는 분자 동역학 모델을 병렬화 하는 방법을 소개한다 .

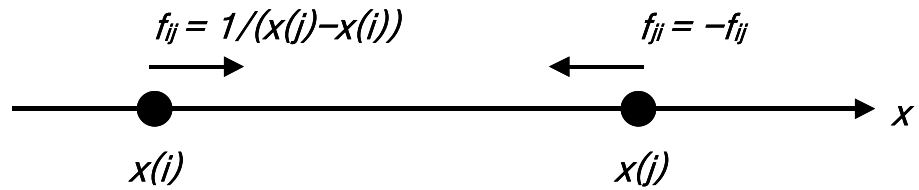


그림 4.6 두 입자의 상호작용

입자  $i$  가 입자  $j$  로부터 받는 힘  $f_{ij}$  는  $f_{ij} = 1/(x_j - x_i)$  로 주었고  $x_i$  는 입자  $i$  의 위치를 나타낸다 . 작용 반작용의 법칙에 의해  $f_{ij} = -f_{ji}$  이고 따라서 입자  $i$  가 받게되는 힘의 총합은 다음과 같다 .

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

아래 그림은 위의 식을 7 개의 입자에 대해 그림으로 나타낸 것이다

|          |           |           |           |           |           |           |           |
|----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
| $f(1) =$ |           | $+f_{12}$ | $+f_{13}$ | $+f_{14}$ | $+f_{15}$ | $+f_{16}$ | $+f_{17}$ |
| $f(2) =$ | $-f_{12}$ |           | $+f_{23}$ | $+f_{24}$ | $+f_{25}$ | $+f_{26}$ | $+f_{27}$ |
| $f(3) =$ | $-f_{13}$ | $-f_{23}$ |           | $+f_{34}$ | $+f_{35}$ | $+f_{36}$ | $+f_{37}$ |
| $f(4) =$ | $-f_{14}$ | $-f_{24}$ | $-f_{34}$ |           | $+f_{45}$ | $+f_{46}$ | $+f_{47}$ |
| $f(5) =$ | $-f_{15}$ | $-f_{25}$ | $-f_{35}$ | $-f_{45}$ |           | $+f_{56}$ | $+f_{57}$ |
| $f(6) =$ | $-f_{16}$ | $-f_{26}$ | $-f_{36}$ | $-f_{46}$ | $-f_{56}$ |           | $+f_{67}$ |
| $f(7) =$ | $-f_{17}$ | $-f_{27}$ | $-f_{37}$ | $-f_{47}$ | $-f_{57}$ | $-f_{67}$ |           |

그림 4.7 입자에 작용하는 힘

아래 순차 프로그램은 각 입자들이 상호작용에 의해 받는 힘을 계산하고 시간진행에 따라 그 힘에 의한 입자들의 위치를 갱신하는 코드이다. 두 입자가 상호작용하는 힘은 서로 대칭적이므로  $i < j$  인 경우만  $f_{ij}$  를 계산하면 된다.

...

PARAMETER (n = ...)

REAL f(n), x(n)

...

DO itime = 1, 100

DO i = 1, n

f(i) = 0.0

ENDDO

DO i = 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

...

```

시간 진행을 나타내는 가장 바깥쪽의 루프내에 힘을 계산하는 두번째 루프가 핫스팟 (hot spot) 임에 주의하자 . 이중으로 내포 되어있는 두번째 루프는 반복 계산 변수인 (i,j) 가 삼각형 모양으로 제한되므로 ( 그림 12.2 참조 ) 블록 분할을 쓰기에 적합치 않다 . 따라서 변수 i 또는 j 에 대한 순환 분할을 사용하는데 i 또는 j 에 대한 순환 분할 중 어떤쪽이 보다 효율적인가 하는 것은 부하 균형 , 캐시미스 등과 같은 요인들에 의해 결정 된다 . 다음 프로그램은 바깥쪽 변수 i 에 대해 순환 분할을 적용해 병렬화 한 것이다 .

```

...

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

        ffj = 1.0 / (x(j) - x(i))

        f(i) = f(i) + ffj

        f(j) = f(j) - ffj

    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

...

```

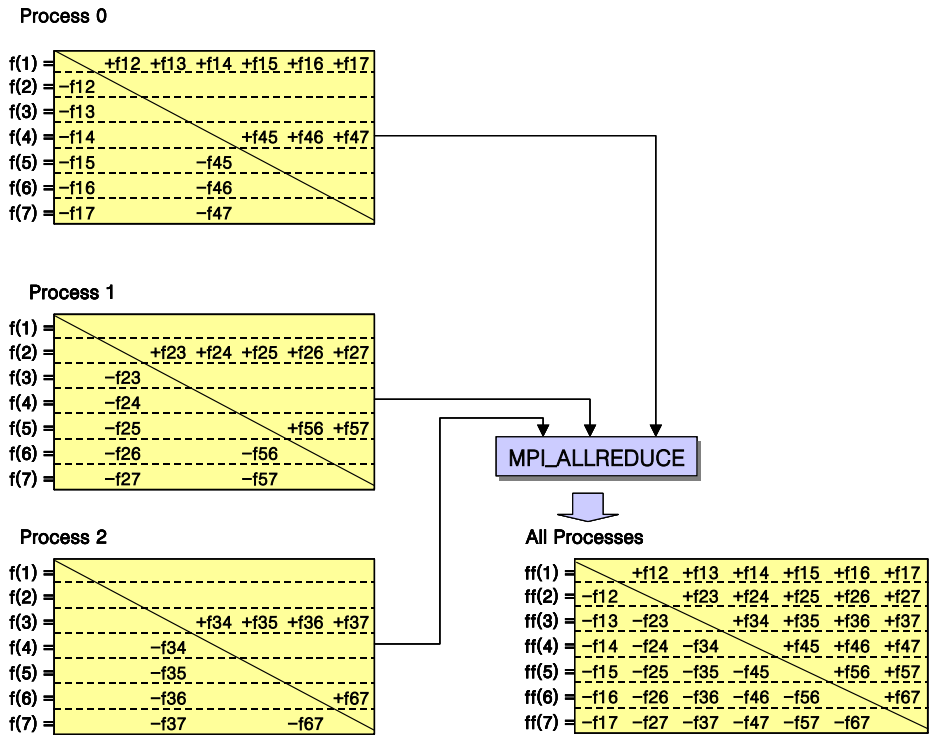


그림 4.8 바깥쪽 루프 (i) 에 대한 순환 분할

다음 프로그램은 안쪽 변수 j 에 대해 순환 분할을 적용해 병렬화 한 것이다.

```

...
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

             $f_{ij} = 1.0 / (x(j) - x(i))$ 

             $f(i) = f(i) + f_{ij}$ 

             $f(j) = f(j) - f_{ij}$ 

        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

...

```

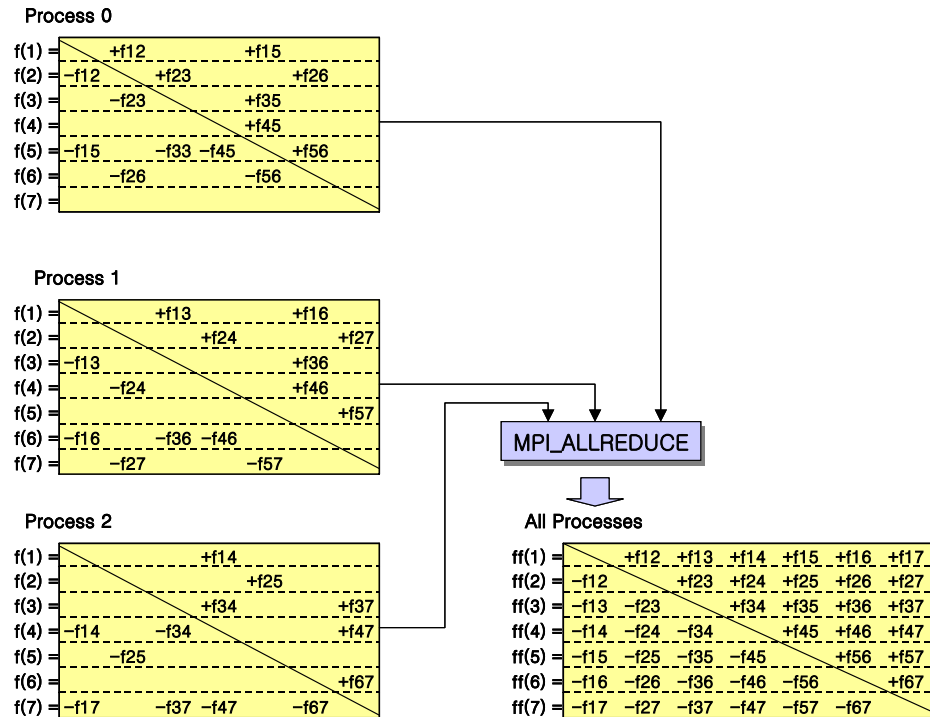


그림 4.9 안쪽 루프 (j) 에 대한 순환 분할

### 4.3.1 MPMD Models

같은 프로그램을 여러 프로세스에서 병렬로 처리하는 SPMD(Single Program Multiple Data) 모델과 달리 MPMD(Multiple Programs Multiple Data) 모델은 다른 프로그램들을 병렬로 실행하고 서로 통신하면서 필요한 데이터를 주고받게 된다. PE for AIX 에서는 환경변수 MP\_PGMMODEL 을 이용하여 “spmd” 와 “mpmd” 를 선택할 수 있으며, 기본적으로는 “spmd” 로 설정되어있다. 유체역학적인 문제와 구조해석 문제로 이루어진 항공기의 전산실험과 같이 서로 다른 해석이 필요한 문제들로 이루어진 프로그램을 실행하고자 하는 경우를 생각해 보자. 가령 fluid.f 와 struct.f 의 두 개의 프로그램이 준비 되어 있다면 AIX 에서 다음과 같이 실행할 수 있다.

```
$ mpxlf fluid.f -o fluid
```



```
$ mpxlf struct.f -o struct
```

( 만일 필요하다면 실행파일을 원격노드에 복사해 두어야 한다 .)

```
$ export MP_PGMMODEL=mpmd
```

```
$ export MP_CMDFILE=cmdfile
```

```
$ poe -procs 2
```

실행 명령으로 구성되는 cmdfile 의 내용은 다음처럼 작성한다 .

```
fluid
```

```
struct
```

각 실행 파일들은 적힌 순서대로 프로세스 랭크에 할당된다 . 즉 , 위의 경우 프로세스 0 에서 fluid 를 프로세스 1 에서 struct 를 실행하게 된다 .

사용자는 유체역학 문제를 위한 프로세스 그룹과 구조해석 문제를 위한 프로세스 그룹을 만들어 각 그룹별로 따로따로 집합통신 루틴을 호출하는 것이 가능하다 . 다음 그림은 fluid 와 struct 의 MPMD 실행을 나타낸 것이다 .

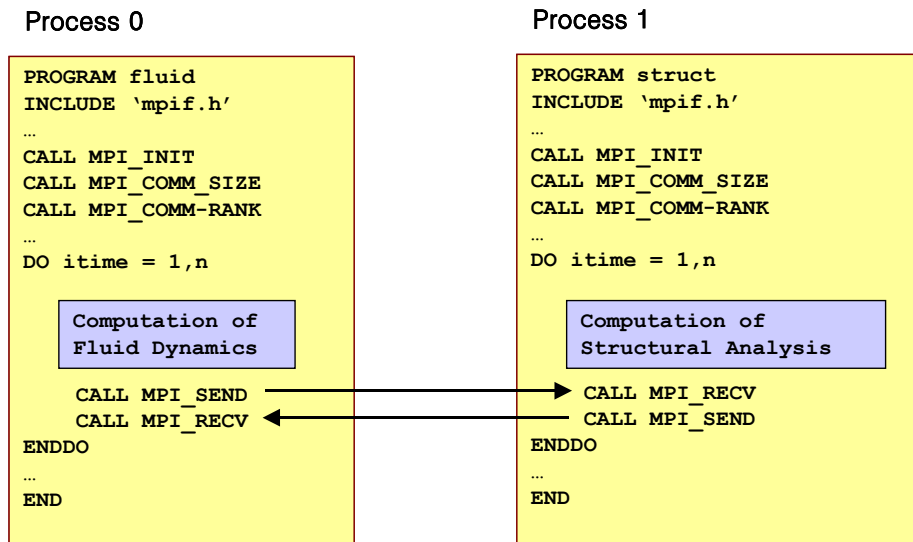


그림 4.10 MPMD 모델

## 제 5 장 부 록

MPI-2 에 새롭게 포함된 병렬 MPI I/O 와 일방통신에 대해 알아 본다 .

### 5.1 MPI-2

1995 년 3 월에 시작된 MPI-2 포럼은 94 년 7 월에 발표된 MPI-1.0 에 대한 오류수정 , 오해가 야기되는 부분들에 대한 명확한 설명 등의 작업을 거쳐 원본을 소폭 수정한 MPI-1.1 을 1995 년 5 월에 발표하였다 . 그 후 2 년여에 걸친 수정과 보완 작업을 거쳐 현재 표준이 된 MPI-1.2 를 포함하는 MPI-2 를 1997 년에 발표하였다 .

MPI-2 는 MPI-1 에서 제시한 MPI 프로그래밍 모델을 확장시키는 완전히 새로운 다음 3 가지 영역을 포함하고 있다 .

- 병렬 I/O
- 원격메모리접근
- 동적 프로세스 운영

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

#### 5.1.1 병렬 MPI I/O

MPI-2 에서 지원하는 병렬 I/O(MPI-I/O) 는 일반적인 UNIX I/O 에 성능과 안정성을 위한 특징들을 추가한 것이다 . 여기서는 기본적인 I/O 기능들인 파일의 열기 , 닫기와 데이터의 쓰기와 읽기에 대해 병렬 I/O 가 추가적으로 제공하는 기능들에 대해 알아볼 것이다 .

#### 5.1.1.1 MPI 프로그램과 순차 I/O (1)

MPI-1 은 병렬 I/O 에 대한 지원이 없었고 따라서 그동안 개발된 MPI 응용 프로그램들의 I/O 는 운영시스템 ( 주로 UNIX ) 에 의존하는 수준이었다 . 그것은 하나의 프로세스가 모든 I/O 를 담당하는 것이었는데 다음의 그림과 예제를 참조하자 .

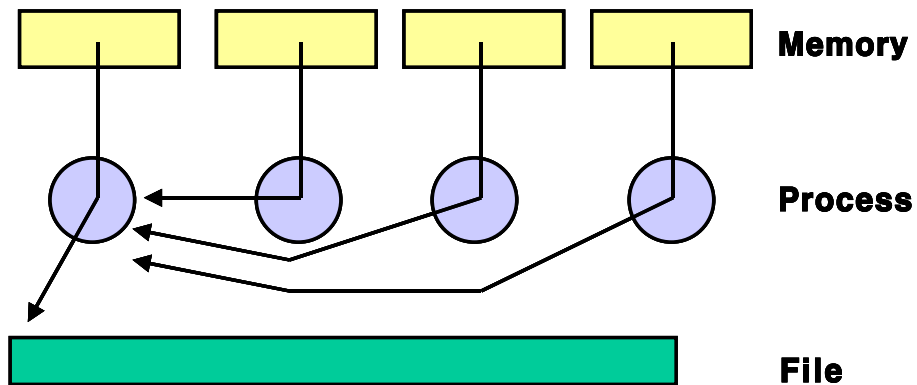


그림 5.1 병렬 프로그램과 순차 I/O

문제를 간단히 하기 위해 각 프로세스가 100 개의 정수를 가지는 배열을 가지고 있다 가정하자 . 그림 5.1 에서 원은 프로세스를 , 위쪽 사각형은 100 개의 정수로 만들어진 배열을 저장한 메모리를 , 그리고 아래쪽 사각형은 쓰기 작업이 이루어지는 파일을 나타낸다 .

예제 5.1 과 위의 5.2 는 다음과 같은 과정을 거치며 I/O 작업을 수행하는 병렬 프로그램이다 .

- 각 프로세스들은 배열 buf() 를 초기화 시킨다 .
- 프로세스 0 을 제외한 모든 프로세스들은 프로세스 0 에게 자신들의 메모리에 가지고 있는 배열 buf() 를 송신한다 .
- 프로세스 0 는 먼저 자신이 가지고 있는 배열을 파일에 기록하고 , 다른 프로세스로부터 차례로 데이터를 수신하여 파일에 기록한다 .

이처럼 프로세스 하나가 모든 I/O 작업을 수행하는 것은 어떤 면에서는 편리할 수도 있지만 병렬성이 없어 성능과 확장성 (scalability) 측면에서 한계가 있게 된다 .

예제 5.1 병렬 프로그램과 순차 I/O : Fortran

```
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

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

```

예제 5.2 병렬 프로그램과 순차 I/O : C

```

/*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;

    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();

}

```

#### 5.1.1.2 MPI 프로그램과 순차 I/O (2)

순차 I/O의 병렬성 부족을 해결하기 위한 제안된 다음 단계는 다음 그림과 같이 프로세스들이 독립적으로 각자의 파일에 쓰기를 하는 것이다.

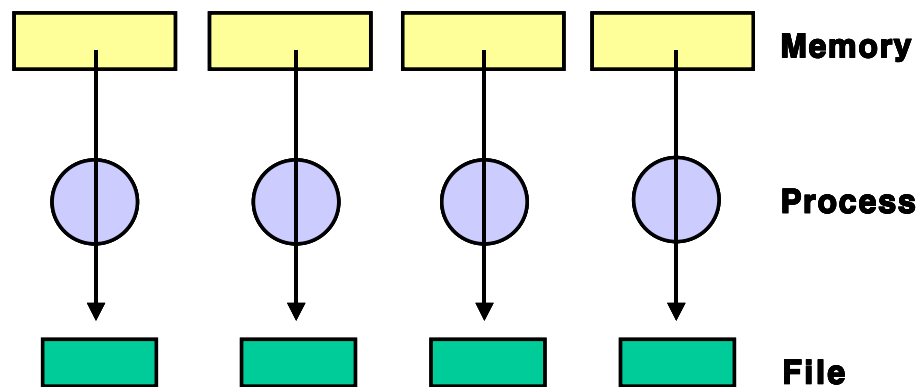


그림 5.2 다중 파일에 저장하는 순차 I/O

각 프로세스는 I/O에 관하여 완전히 독립적으로 기능하며 각자 운영시스템에서 제공하는 I/O를 사용해 순차적으로 파일에 접근한다. 각 프로세스는 자신의 파일을 열고, 쓰고, 닫게 된다. 이 방식은 I/O 기능들이 순차적인 라이브러리를 여전히 이용하면서도, 병렬적으로 실행된다는 유리한 점이 있기는 하지만 프로그램의 실행 결과 하나의 파일이 아닌 여러 개의 파일이 생성되어 차후에 이 결과들을 이용하기가 어렵다는 심각한 단점을 가진다.

### 예제 5.3 다중 파일에 저장하는 순차 I/O : Fortran

```
PROGRAM serial_IO2

INCLUDE 'mpif.h'

INTEGER BUFSIZE

PARAMETER (BUFSIZE = 100)

INTEGER nprocs, myrank, ierr, buf(BUFSIZE)

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

WRITE(number, *) myrank

filename(myrank) = "testfile." //number

OPEN(UNIT=myrank+10, FILE=filename(myrank), STATUS= "NEW" , &

    ACTION= "WRITE" )
```



```

WRITE(myrank+10,*) buf

CLOSE(myrank+10)

CALL MPI_FINALIZE(ierr)

END

```

예제 5.4 다중 파일에 저장하는 순차 I/O : C

```

/*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();

}

```

### 5.1.1.3 MPI 프로그램과 병렬 I/O (1)

우선 예제 5.3 과 5.4 를 기본적인 병렬 I/O 루틴을 이용해 고쳐보자 . 기존의 I/O 기능과 병렬 I/O 의 유사성에 대해 알 수 있을 것이다 . 다음 예제는 병렬 I/O 루틴 MPI\_FILE\_OPEN, MPI\_FILE\_WRITE, MPI\_FILE\_CLOSE 를 이용했으며 그 결과는 예제 5.3 과 5.4 와 동일하다 .

예제 5.5 다중 파일에 저장하는 병렬 I/O : Fortran

```

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

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

```

예제 5.6 다중 파일에 저장하는 병렬 I/O : C

```

/*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;

    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();
}

```

예제 5.3, 5.4 와 똑 같은 실행 결과를 보이지만 병렬 I/O 루틴을 이용한 예제 5.5, 5.6 의 특징을 살펴보자 . 우선 변수 myfile 의 타입이 FILE 에서 MPI\_File 로 바뀌었으며 , 루틴 fopen 에 대응하는 MPI 루틴으로 MPI\_File\_open 이 쓰이고 있다 .

**CALL MPI\_FILE\_OPEN(MPI\_COMM\_SELF, filename, MPI\_MODE\_WRONLY + MPI\_MODE\_CREATE, MPI\_INFO\_NULL, myfile, ierr)**

**MPI\_File\_open(MPI\_COMM\_SELF, filename, MPI\_MODE\_CREATE | MPI\_MODE\_WRONLY, MPI\_INFO\_FULL, &myfile);**

첫 인수 MPI\_COMM\_SELF 는 커뮤니케이터이다 . MPI\_File\_open 은 집합 통신으로 병렬 I/O 에서 파일들은 이 MPI 커뮤니케이터로 확인되는 프로세스들의 집합에서 열린다 . 그렇게 함으로써 전체 프로세스중 일부만이 참여하는 I/O 가 가능해진다 . 여기서는 각 프로세스가 서로 공유하지 않는 자신만의 파일을 열게 되므로 커뮤니케이터 MPI\_COMM\_SELF 를 썼다 . 두번째 인수 filename 은 여는 파일의 이름을 나타내는 캐릭터 변수이다 . 세번째 인수는 파일 접근모드를 나타내는 핸들로 오직 쓰기 작업을 (MPI\_MODE\_WRONLY) 위해 파일을 새로 생성하는 것을 나타

낸다 (MPI\_MODE\_CREATE). 참고로 MPI 에서 사용가능한 접근모드들은 다음과 같다. 그리고, 이러한 상수들은 C/C++ 에서는 OR() 로 Fortran 에서는 IOR(+) 로 연결해 사용할 수 있다.

- MPI\_MODE\_APPEND : 파일 포인터의 초기위치를 파일 마지막에 설정
- MPI\_MODE\_CREATE : 파일이 없으면 파일을 만들고 있으면 덮어씀
- MPI\_MODE\_DELETE\_ON\_CLOSE : 파일을 닫으면 삭제시킴
- MPI\_MODE\_EXCL : MPI\_MODE\_CREATE을 설정했는데 파일이 이미 존재하면 에러 리턴
- MPI\_MODE\_RDONLY : 읽기만 가능
- MPI\_MODE\_RDWR : 읽기와 쓰기 모두 가능
- MPI\_MODE\_SEQUENTIAL : 파일은 순차적으로만 접근
- MPI\_MODE\_UNIQUE\_OPEN : 다른 곳에서 동시에 열수 없음
- MPI\_MODE\_WRONLY : 쓰기만 가능

네번째 인수 info 는 시스템 환경에 따른 프로그램 구현의 변화에 대한 정보를 준다. 여기서와 마찬가지로 대부분 MPI\_INFO\_NULL 의 기본값을 사용한다. MPI\_File\_open 은 마지막 인수 fh 로 MPI\_File 타입 변수의 주소를 리턴한다. 인수 fh 는 루틴 MPI\_File\_close 가 호출되어 파일이 닫기기 전까지 유효하다.

MPI\_File\_open 의 일반적인 사용법은 다음과 같다.

**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)

예제에서 쓰여지고 있는 또다른 병렬 I/O 루틴은 MPI\_File\_write 와 MPI\_File\_close 이다 .

**CALL MPI\_FILE\_WRITE(myfile, buf, BUFSIZE, MPI\_INTEGER,  
MPI\_STATUS\_IGNORE, ierr)**

**CALL MPI\_FILE\_CLOSE(myfile, ierr)**

**MPI\_File\_write(myfile, buf, BUFSIZE, MPI\_INT, MPI\_STATUS\_IGNORE);**

**MPI\_File\_close(&myfile);**

두 루틴에서 사용되는 인수들은 이미 익숙한 것들이므로 아래의 일반적인 사용법에 대한 설명으로 대신한다 .

**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)

**C:**

**int MPI\_File\_close(MPI\_File \*fh)**

**Fortran:**

**MPI\_FILE\_WRITE(fh, ierr)**

INTEGER fh : 닫아야 하는 파일의 파일 핸들 (INOUT)

#### **5.1.1.4 MPI 프로그램과 병렬 I/O (2)**

이제 각 프로세스가 따로 파일을 작성하는 대신 하나의 파일을 공유하도록 예제를 수정해 보자 . 앞선 예제의 병렬성을 그대로 유지하면서 여러 개의 파일을 작성하는 문제점을 해결함으로서 병렬 I/O 가 가지는 효율성을 살펴볼 수 있을것이다 . 다음 수정된 프로그램을 보자 .

예제 5.7 단일 파일에 저장하는 병렬 I/O : Fortran

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

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

```

예제 5.8 단일 파일에 저장하는 병렬 I/O : C

```

/*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);

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();

}

```

앞절의 예제와 다른점은 루틴 MPI\_File\_open 의 첫번째 인수를 MPI\_COMM\_SELF 대신 MPI\_COMM\_WORLD 를 쓴것이다 . 이것은 모든 프로세스들이 하나의 공유 파일을 여는 것을 나타낸다 .

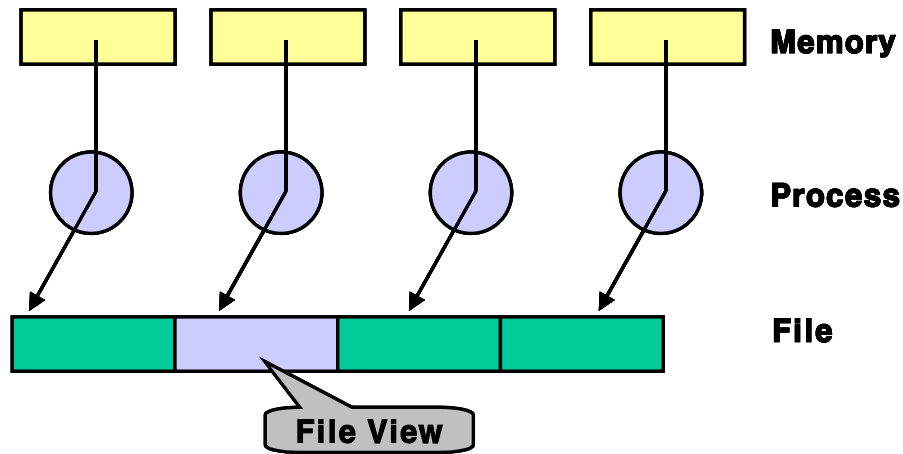


그림 5.3 단일 파일에 저장하는 병렬 I/O

위의 그림은 하나의 파일을 열어 여러 프로세스들이 파일의 각 부분에 따로 접근하는 병렬 I/O의 기능을 보여준다. 하나의 프로세스가 접근하는 파일의 부분은 파일 뷰 (file view)로 불리며, 다음처럼 루틴 MPI\_File\_set\_view를 호출해 각 프로세스에 설정된다.

**CALL MPI\_FILE\_SET\_VIEW(thefile, disp, MPI\_INTEGER, MPI\_INTEGER, &'native', MPI\_INFO\_NULL, ierr)**

**MPI\_File\_set view(thefile, myrank\*BUFSIZE\*sizeof(int), MPI\_INT, MPI\_INT, "native", MPI\_INFO\_NULL)**

첫 번째 인수는 열어둔 파일을 나타내고, 두 번째 인수는 MPI\_Offset 데이터 타입을 가지며 프로세스의 파일 뷰가 시작되는 파일내의 위치를 바이트 단위로 나타낸다. 다음 인수는 뷰의 etype으로 불리며, 파일내 데이터의 단위를 정의한다. 여기서는 MPI\_INT이고 이것은 파일내에 MPI\_INT 타입의 데이터들을 쓰게 됨을 나타낸다. 네 번째 인수는 filetype으로 불리며, 미리 정의된 MPI 타입과 더불어 유도 데이터 타입을 사용해 파일내의 불연속적인 뷰들을 나타내는데 유용하게 쓰인다.

다음 인수 “native” 는 파일에서 사용되는 데이터 표현 (data representation) 을 기술하는 문자열로 데이터가 메모리에 있는 것과 똑같이 파일에서 표현됨을 나타낸다. native 는 동종 시스템 환경에서는 타입 변환에 따른 데이터 정밀도와 I/O 성능의 손실이 없는 장점이 있어서 대부분의 경우에 많이 사용된다. 이외에 internal 과 external32 가 있다. 마지막 인수는 앞선 MPI\_File\_open 에서 쓰였던 MPI\_Info 타입 변수 MPI\_INFO\_NULL 이다.

루틴 MPI\_File\_write 의 호출은 앞선 예제에서와 동일하게 쓰였지만, 앞서와 달리 동일한 파일의 적절한 위치들에 커뮤니케이터내의 모든 프로세스들이 쓰기 작업을 병렬로 실행하게 된다. MPI\_File\_set\_view 는 집합통신으로 커뮤니케이터내의 모든 프로세스들에 의해 호출된다. 일반적인 사용법은 다음과 같다.

**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)

#### **5.1.1.5 MPI 프로그램과 병렬 I/O (3)**

앞절에서와 같이 병렬 I/O 를 이용해 하나의 파일을 작성하게 되면 그 파일 하나를 다시 병렬 I/O 를 통하여 다른 여러 프로세스에서 병렬로 읽을 수 있어서 유리하다. 파일에 병렬로 쓰기를 하면서 프로세스의 개수 등과 관련된 내부 구조없이 하나의 파일을 작성했다면 그 파일을 읽기 위해 이전과 같은 개수의 프로세스를 이용할

필요는 없다 . 다음은 예제 5.7 과 5.8 에서 작성한 파일을 병렬로 읽어들이는 프로그램이다 . 이 프로그램은 프로세스의 개수와 무관하며 , 프로그램 내에서 파일 전체의 크기가 계산되고 설정된 프로세스의 뷰들은 각각 근사적으로 동일한 양의 데이터를 읽어들이게 된다 .

예제 5.9 다중 프로세스에서 단일 파일 읽기 : Fortran

```
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)

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

```

예제 5.10 다중 프로세스에서 단일 파일 읽기 : C

```

/* 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);

    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();

}

```

예제에서 사용되고 있는 새로운 MPI 루틴 `MPI_File_get_size` 는 파일 `thefile` 을 열어  
서 그 파일의 크기를 바이트 단위로 저장한다 . 또 `MPI_File_read` 는 파일에서 지정  
된 개수의 데이터를 읽어들이 버퍼에 저장하는 역할을 한다 .

**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)

**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)

### 5.1.2 일방통신 (One-Sided Communication)

메시지 패싱 모델의 특징은 조합된 한 쌍의 송 / 수신 연산을 통하여 데이터를 한쪽 프로세스의 주소 공간에서 다른쪽 프로세스의 주소공간으로 옮기는 것이다 . 이러한 사실은 여러 프로세스들이 하나의 메모리 풀 (pool) 을 공유하여 읽기 , 쓰기 등의 메모리 연산을 간단히 수행하는 공유 메모리 모델과 메시지 패싱 모델을 확연하게 구분짓는다 .

MPI-2 는 MPI 환경에서 공유 메모리 모델에서의 메모리 연산과 같은 기능을 제공하는 새로운 API 를 정의하고 있다 . MPI 의 ‘일방’ 또는 ‘원격 메모리’ 연산으로 불리우는 이 같은 기능들은 프로세스간의 데이터 전송을 송 / 수신 조합이 아닌 한쪽 프로세스만의 연산으로 완전히 시작될 수 있게 한다 . get, put, 그리고 accumulate 등이 대표적인 일방통신 연산들이며 , 원격 메모리 연산이라는 의미로 RMA(Remote Memory Access) 연산이라 부르기도 한다 .

MPI-2 가 실질적인 공유 메모리 프로그래밍 모델을 제공하는 것은 아니지만 일방통신 기능을 통하여 공유 메모리 모델과 같은 유연성 (flexibility) 을 사용자에게 제공하고 있다 . 한가지 주의할 점은 일방통신의 사용이 송 / 수신 사용보다 탁월한 성능 향상을 보장하지는 않는다는 것이다 . 일방통신은 공유 메모리 시스템과 분산 메모리 시스템 모두에서 작업이 가능하도록 설계되었으며 , 사용자에게 성능측면에서의 이득보다는 알고리즘 설계를 위한 유연성을 제공하고 있다 .

### 5.1.2.1 메모리 윈도우

메시지 패싱 모델에서 송 / 수신 버퍼들은 다른 프로세스들로 보내어질 ( 송신 ) 또는 다른 프로세스들에게 쓰기를 허용하는 ( 수신 ) 프로세스의 주소공간 일부를 의미한다 . MPI-2 에서 이러한 “통신 메모리” 의 개념은 “원격 메모리 접근 윈도우” 의 개념으로 일반화 된다 . 메모리 윈도우는 단일 프로세스의 메모리 일부로서 다른 프로세스들의 읽기 (get), 쓰기 (put), 그리고 갱신 (accumulate) 등의 메모리 연산을 허용하는 주소공간이다 .

아래 그림은 두 개의 프로세스에 만들어진 윈도우로 구성된 윈도우 객체를 보여준다 . 다른 프로세스의 원격 메모리로 (put) 또는 원격 메모리로부터 (get) 데이터를 옮기는 put 과 get 연산은 논블록킹이어서 연산의 완료를 확인하기 위해서는 따로 동기화 과정이 필요하다 .

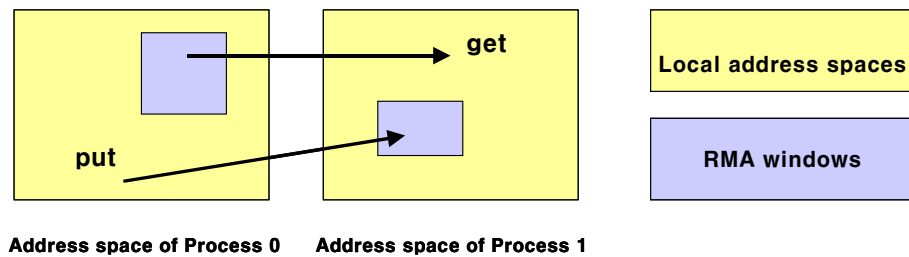


그림 5.4 원격 메모리 접근 윈도우

### 5.1.2.2 일방통신의 사용 : pi 프로그램

기본적인 일방통신 루틴을 소개하고 그 사용법을 알아보기 위해 다음 예제 프로그램을 이용한다 . 주어진 예제는 수치적분을 통하여  $\pi$  값을 병렬로 계산하는 MPI 프로그램이다 . 프로그램은 우선 프로세스 0 에서 사용자로부터 적분간격의 개수를 받아 MPI\_BCAST 를 통하여 다른 프로세스들에게 전달한다 . 각 프로세스는 루프 인덱스를 할당 받아 부분합 sum 을 계산하고 이 부분합을 MPI\_REDUCE 를 이용해 모두 취합해서 원하는  $\pi$  값을 계산하고 있다 .

예제 5.11  $\pi$  계산 MPI 병렬 프로그램 : Fortran



```

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

        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

    ENDIF

END DO

```

```

    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

```

예제 5.12  $\pi$  계산 MPI 병렬 프로그램 : C

```

#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);

    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();

}

```

위의 프로그램을 기본적인 일방통신 연산을 사용해 작성한 다음의 예제 예제 5.13 과 예제 5.14 를 보자 . 대략적인 흐름을 보면 , 우선 프로세스 0 가 사용자로부터 읽어들이는 값을 방송으로 전달하지 않고 , RMA 윈도우 객체 부분에 저장해서 다른 프로세스들이 간단히 그 값을 `get` 할 수 있도록 한다 . 부분합을 계산한 이후에는 모든 프로세스들의 윈도우 객체로부터 부분합을 취합하여 총합을 계산하도록 연산

accumulate 를 이용한다 . 그리고 동기화를 위해 윈도우 동기화 연산으로 가장 간단한 fence 를 이용한다 .

예제 5.13 일방통신을 이용한  $\pi$  계산 MPI 병렬 프로그램 : Fortran

```
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

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

        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

```

예제 5.14 일방통신을 이용한  $\pi$  계산 MPI 병렬 프로그램 :

```

C#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); }

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);

}

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();

}

```

일방통신을 이용하는 프로그램은 윈도우 객체의 설정으로 시작된다. 윈도우 객체들은 단일 데이터 타입의 변수들로 구성되기 때문에 위 예제에서 필요한 윈도우 객체는 정수타입 (n) 과 실수타입 (pi) 두 개가 된다. 각 윈도우 객체는 프로세스 0의 메모리에 있는 하나의 수로 구성되며 C에서는 MPI\_Win 타입, Fortran에서는 정수 타입의 변수로 표현된다. 프로세스 0에서 호출되는 윈도우 생성 루틴을 살펴보자. 각각 Fortran 과 C에서 호출한 것이다.



**CALL MPI\_WIN\_CREATE(n, 4, 1, MPI\_INFO\_NULL, MPI\_COMM\_WORLD,  
nwin, ierr)**

**MPI\_Win\_create(&n, sizeof(int), 1, MPI\_INFO\_NULL, MPI\_COMM\_WORLD,  
&nwin);**

윈도우 생성 루틴은 집합통신으로 윈도우 객체에 대해 어떠한 메모리 기여부분이 없더라도 커뮤니케이터내의 다른 모든 프로세스에서 호출되어야 한다. 프로세스 0 에서 호출된 위의 생성루틴과 대응하여 다른 프로세스들에서는 다음 루틴을 호출하고 있다.

**CALL MPI\_WIN\_CREATE(MPI\_BOTTOM, 0, 1, MPI\_INFO\_NULL, &  
MPI\_COMM\_WORLD, nwin, ierr)**

**MPI\_Win\_create(MPI\_BOTTOM, 0, 1, MPI\_INFO\_NULL,  
MPI\_COMM\_WORLD, &nwin);**

각 인수들에 대해 살펴보면, 처음 두 인수는 다른 프로세스들에게 put/get 연산을 허용하도록 이 생성루틴을 호출하는 프로세스의 지역 메모리에 있는 윈도우의 주소와 바이트 단위의 길이를 나타낸다. 프로세스 0에서는 정수 n에 접근을 허용하고 있으며, 다른 프로세스에서는 접근을 허용하는 윈도우 객체 생성이 없다는 것을 나타내기 위해 길이를 0으로 주소를 MPI\_BOTTOM으로 두고있다.

다음 인수는 변위를 나타내며 여기서는 오직 하나의 변수만 가지는 윈도우 객체이므로 중요한 값이 아니며 1(바이트)로 두었다. 네번째 인수는 상황별로 연산의 성능을 최적화 하는데 사용될 수 있는 MPI\_Info 변수로 MPI\_INFO\_NULL로 두었다. 다섯번째 인수는 윈도우 객체에 접근을 허용하는 프로세스들의 집합을 나타내는 커뮤니케이터이고, MPI는 마지막 인수를 통해 MPI\_Win 객체를 리턴하게 된다.

프로그램에서 MPI\_Win\_create 루틴이 호출되면 커뮤니케이터내의 모든 프로세스들은 put(쓰기), get(읽기), accumulate(갱신) 연산을 통해 nwin에 있는 데이터에 접근할 수 있다. 두번째 MPI\_Win\_create 루틴 호출은 윈도우 객체 piwin을 생성하고 각 프로세스들은 변수 pi에 접근하여 각자 계산한  $\pi$  값을 취합하게 된다.

이제 프로세스 랭크가 0이 아닌 프로세스들은 생성된 윈도우 객체로부터 직접 n 값을 get 할 수 있다 . 그러나 , MPI\_Get 을 호출하여 , get 을 하기 전에 동기화 함수 MPI\_Win\_fence 를 호출해야 한다 . 기존의 동기화 함수 MPI\_Barrier 는 일방통신 연산에서는 사용할 수 없다 .

**CALL MPI\_WIN\_FENCE(0, nwin, ierr)**

**MPI\_Win\_fence(0, nwin);**

fence 연산을 수행하는 MPI\_Win\_fence 루틴은 두개의 인수를 가진다 . 처음 인수는 최적화와 관련된 “assertion” 인수이며 , 여기서는 항상 허용되는 assertion 값인 0 을 사용하였다 . 두 번째 인수는 fence 연산이 수행되는 윈도우를 나타낸다 . MPI\_Win\_fence 는 윈도우에 대한 원격연산과 지역연산을 또는 두 개의 원격연산을 분리시키는 장벽 역할을 한다고 할 수 있다 . 가령 , 예제에서 처음 나오는 MPI\_Win\_fence 는 터미널에서 n 값을 읽어들이는 과정과 원격연산 MPI\_Get 을 분리시켜 주고 있다 .

**CALL MPI\_GET(n, 1, MPI\_INTEGER, 0, 0, 1, MPI\_INTEGER, nwin, ierr)**

**MPI\_Get(&n, 1, MPI\_INT, 0, 0, 1, MPI\_INT, nwin);**

get 연산을 수행하는 루틴 MPI\_Get 의 인수들은 MPI 송 / 수신 루틴의 인수들을 하나의 호출에서 모두 다루고 있다고 할 수 있다 . get 은 수신과 유사하다 . 따라서 우선 수신버퍼에 대해 처음 세 인수 (&n, 1, MPI\_INT) 를 이용해 시작주소 , 개수 , 데이터 타입을 기술한다 . 다음 인수는 메모리 접근을 허용하는 도착지 프로세스의 랭크이다 . 여기서는 모든 프로세스들이 0 번 프로세스의 메모리에 접근하기 때문에 0 으로 설정되었다 . 다음 세 개의 인수들은 윈도우에서 송신버퍼의 주소 , 개수 , 데이터 타입을 정의한다 . 주소는 도착지 프로세스의 원격 메모리상에서 윈도우의 시작 주소로부터의 변위로 주어지는데 여기서는 윈도우에 하나의 값만 있으므로 변위는 0 이 된다 . 맨 마지막 인수는 윈도우 객체를 나타내고 있다 .

주의해야 할 사실은 원격 메모리 연산들은 오직 데이터 이동을 시작해줄 뿐이라는 것이다. 즉, 여기서 사용된 `MPI_Get`의 경우도 논블록킹 연산이어서 연산의 완료를 확정지으려면 동기화 함수 `MPI_Win_fence`를 반드시 호출해야 한다.

다음의 코드들은  $\pi$  값을 계산하기 위해 각 프로세스에서 부분합 `mypi`를 계산하는 과정이다. 각 프로세스들은 `mypi`의 값을 윈도우 객체 `piwin`에 더해줌으로서  $\pi$  값을 갱신시켜 원하는  $\pi$ 의 근사치를 구하게 된다. 코드를 보면 먼저 `MPI_Win_fence`를 호출해 지역연산과 원격연산을 분리 시키고 원격연산의 실행을 준비한다. 다음으로 아래의 루틴을 호출해 `accumulate` 연산을 수행한다.

```
CALL MPI_ACCUMULATE(mympi, 1, MPI_DOUBLE_PRECISION, 0, 0, &  
1, MPI_DOUBLE_PRECISION, MPI_SUM, piwin, ierr)
```

```
MPI_Accumulate(&mympi, 1, MPI_DOUBLE, 0, 0, 1, MPI_DOUBLE, MPI_SUM,  
piwin);
```

처음 세 개의 인수들은 갱신에 사용되는 지역 변수로서 각각 주소, 개수, 데이터 타입을 나타낸다. 네번째 인수는 목적 프로세스의 랭크이고 이어지는 세 개의 인수는 갱신되어지는 값을 범위, 개수, 데이터 타입의 형태로 나타내고 있다. 다음 인수는 갱신에 사용되는 연산을 나타내며, 이용가능한 연산들은 `MPI_Reduce`에서 사용가능한 것들과 동일하지만 단, 사용자가 정의한 연산은 `MPI_Accumulate`에서 사용할 수 없다. `MPI_Accumulate` 역시 논블록킹 연산이므로 `MPI_Win_fence`를 호출해 연산을 완료시킨다. 프로그램의 마지막에서는 `MPI_Win_free`를 호출해 윈도우 객체들을 해제하고 있다.

기본적인 일방통신 루틴들의 일반적인 사용법을 소개한다.

**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)

**C:**

**int MPI\_Win\_fence(int assert, MPI\_Win \*win)**

**Fortran:**

**MPI\_WIN\_FENCE(assert, win, ierr)**

INTEGER assert : 성능 향상 관련 인수 , 0 은 항상 허용됨 (IN)

INTEGER win : 펜스연산이 수행되는 윈도우 객체 (IN)

**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)

**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)

**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_Win win)
```

**Fortran:**

```
MPI_ACCUMULATE(origin_addr, origin_count, origin_datatype,
target_rank, target_disp, target_count, target_datatype, 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)

**C :**

```
int MPI_Win_free(MPI_Win *win)
```

**Fortran :**

```
MPI_WIN_FREE(win, ierr)
```

INTEGER win : 윈도우 객체 ( 핸들 ) (IN)

## 5.2 참고 자료

1. Gropp, Lusk, and Skjellum. Using MPI, second edition. MIT Press. 1999
2. Gropp, Lusk, and Thakur. Using MPI-2, second edition. MIT Press. 1999
3. Snir, Otto, Huss-Lederman, Walker, and Dongarra. MPI-The Complete Reference Volume 1. Second Edition. MIT Press. 1998.
4. Andrews. Foundations of Multithreaded, Parallel, and Distributed Programming. Addison-Wesley. 2000.
5. SP Parallel Programming Workshop  
<http://www.mhpcc.edu/training/workshop/>
6. Parallel Programming Concepts  
<http://www.tc.cornell.edu/Services/Edu/Topics/ParProgCons/more.asp>
7. Introduction to Parallel Computing  
[http://foxtrot.ncsa.uiuc.edu:8900/SCRIPT/HPC/scripts/serve\\_home](http://foxtrot.ncsa.uiuc.edu:8900/SCRIPT/HPC/scripts/serve_home)
8. Practical MPI Programming  
<http://www.redbooks.ibm.com/pubs/pdfs/redbooks/sg245380.pdf>
9. Lawrence Libermore National Laboratory  
<http://www.llnl.gov/computing/tutorials/workshops/workshop/>
10. Parallel Programming with MPI  
<http://oscinfo.osc.edu/training/mpi/>

# 찾아보기

## A

accumulate 262, 267

## B

Block Distribution 127  
Block-Cyclic Distribution 134  
Broadcast 49  
Bufferd Send 21

## C

chunk 127  
column-major order 141  
communicator 6  
Cyclic Distribution 133

## D

data representation 258  
deadlock 18, 28, 44  
degree of parallelism 184  
Derived Data Type 80

## E

envelope 5, 17  
external32 258

## F

FDM 150, 203  
fence 267  
file view 257  
Finite Difference Method 150  
Fortran 3

## G

Gather 53

get 262  
GPFS 122

## H

host.list 12  
hot spot 127, 235

## I

internal 258

## M

Molecular Dynamics 232  
MP\_EAGER\_LIMIT 22  
MPI Data Type 15  
MPI Handle 9  
MPI Message 15  
MPI\_TYPE\_FREE 98  
MPI-1 3  
MPI-2 3, 56, 98, 242  
MPI\_ACCUMULATE 274, 277  
MPI\_ALLGATHER 60  
MPI\_ALLGATHERV 64  
MPI\_ALLREDUCE 71  
MPI\_ALLTOALL 77  
MPI\_ALLTOALLV 77  
MPI\_ANY\_SOURCE 25  
MPI\_ANY\_TAG 25  
MPI\_BARRIER 76  
MPI\_BCAST 49  
MPI\_BOTTOM 272  
MPI\_BUFFER\_ATTACH 21  
MPI\_BUFFER\_DETACH 21  
MPI\_CART\_COORDS 114  
MPI\_CART\_CREATE 108  
MPI\_CART\_RANK 112



|                              |                             |
|------------------------------|-----------------------------|
| MPI_CART_SHIFT 116           | MPI_MODE_SEQUENTIAL 252     |
| MPI_CART_SUB 119             | MPI_MODE_UNIQUE_OPEN 252    |
| MPI_Comm 13                  | MPI_MODE_WDONLY 252         |
| MPI_COMM_NULL 104            | MPI_MODE_WRONLY 251         |
| MPI_COMM_RANK 12             | MPI_Offset 257              |
| MPI_COMM_SELF 251, 256       | MPI_OP_CREATE 68            |
| MPI_COMM_SIZE 13             | MPI_ORDER_C 99              |
| MPI_COMM_SPLIT 103, 119      | MPI_ORDER_FORTRAN 99        |
| MPI_COMM_WORLD 6, 12         | MPI_PROC_NULL 155           |
| MPIF                         | MPI_PUT 276                 |
| MPI Forum 3                  | MPI_RECV 24                 |
| MPI_FILE 251                 | MPI_REDUCE 65               |
| MPI_FILE_CLOSE 249, 252, 253 | MPI_REDUCE_SCATTER 78       |
| MPI_FILE_GET_SIZE 261        | MPI_SCAN 80                 |
| MPI_FILE_OPEN 249, 251, 252  | MPI_SCATTER 72              |
| MPI_FILE_READ 261            | MPI_SCATTERV 75             |
| MPI_FILE_SET_VIEW 257, 258   | MPI_SEND 24                 |
| MPI_FILE_WRITE 249, 253      | MPI_STATUS_SIZE 25          |
| MPI_FINALIZE 13              | MPI_SUCCESS 10              |
| MPI_GATHER 53                | MPI_SUM 68                  |
| MPI_GATHERV 56               | MPI_TEST 38                 |
| MPI_GET 273, 275             | MPI_TYPE_COMMIT 98          |
| MPI_GET_COUNT 26             | MPI_TYPE_CONTIGUOUS 82      |
| MPI-I/O 242                  | MPI_TYPE_CREATE_SUBARRAY    |
| MPI_INFO_NULL 252, 258, 272  | 98, 157, 163                |
| MPI_IN_PLACE 56              | MPI_TYPE_EXTENT 94          |
| MPI_Irecv 37                 | MPI_TYPE_HVECTOR 88         |
| MPI_ISEND 36                 | MPI_TYPE_STRUCT 89          |
| MPI_LB 90                    | MPI_TYPE_VECTOR 85          |
| MPI_MAXLOC 69                | MPI_UB 90                   |
| MPI_MODE_APPEND 252          | MPI_UNDEFINED 104           |
| MPI_MODE_CREATE 252          | MPI_WAIT 38, 44             |
| MPI_MODE_DELETE_ON_CLOSE     | MPI_Win 271                 |
| 252                          | MPI_WIN_CREATE 272, 274     |
| MPI_MODE_EXCL 252            | MPI_WIN_FENCE 273, 274, 275 |
| MPI_MODE_RDONLY 252          | MPI_WIN_FREE 274, 277       |
| MPI_MODE_RDWR 252            | MPMD                        |

Multiple Programs Multiple  
Data 239  
MP\_PGMMODEL 239  
MP\_STDUTMODE 126

## **N**

native 258  
NFS  
Network File System 122

## **O**

One-Sided Communication 262

## **P**

periodic boundary condition 109  
posting 35  
prefix sum 198  
put 262

## **R**

random number 226  
rank 6  
Ready Send 20  
Reduce 65  
request 35, 37, 38, 39  
RMA  
Remote Memory Access 262  
round-robin 133  
row-major order 141

## **S**

scatter 72  
Shrinking Arrays 135  
SPMD

Single Program Multiple Data  
239

Standard Send 22  
submatrix 99  
Superposition 177  
Synchronizing Data 167  
Synchronous Send 20

## **T**

Tag 6  
Twisted Decomposition 188

## **U**

UNIX I/O 242

## **V**

Virtual Topology 107

## **ㄱ**

가상 토폴로지 107  
검사 38  
교착 18, 28, 44  
꼬리표 6, 25

## **ㄴ**

난수 226  
내포된 (Nested) 루프 141  
논블록킹 송신 36  
논블록킹 수신 37  
논블록킹 통신 18

## **ㄷ**

단방향 통신 42  
대기 38

대응 함수 112  
데이터 동기화 167  
데이터 표현 258  
동적 프로세스 운영 242

## ㄹ

라운드로빈 133

## ㄱ

메모리 대응 141  
메모리 윈도우 263  
메모리 풀 262  
메시지 5  
몬테카를로 방법 226

## ㅂ

방송 49  
배열 덩어리 127  
배열 수축 135  
병렬 I/O 242  
봉투 5, 17  
부분행렬 99  
부하 균형 134, 235  
분자 동역학 모델 233  
블록 분할 127  
블록 순환 분할 134  
블록킹 동기 송신 20  
블록킹 버퍼 송신 21  
블록킹 송신 24  
블록킹 수신 24  
블록킹 준비 송신 20  
블록킹 통신 18  
블록킹 표준 송신 22  
비틀림 분해 188

## ㅅ

수신버퍼 25  
순환 분할 133

## ㅇ

양방향 통신 44  
열 우선순 141  
와일드카드 25  
원격 메모리 연산 262  
원격메모리접근 242  
유도 데이터 타입 80  
유한 차분법 150, 203  
일방통신 262  
임의 행로 226

## ㅈ

점대점 통신 6, 17  
주기적 경계 조건 109  
중첩 177  
직교좌표 가상 토폴로지 108  
집합 통신 7, 48

## ㅊ

취합 53

## ㅋ

캐시미스 134, 141, 142, 235  
커뮤니케이터 6, 12

## ㅌ

토폴로지 분해 119  
통신모드 19

## 표

파이프라인 179  
파일 뷰 257  
포스팅 35  
프로세서 5  
프로세스 5  
프로세스 랭크 6, 12  
프리픽스 합 198

## ㅎ

행 우선순 141  
확산 72  
환산 65