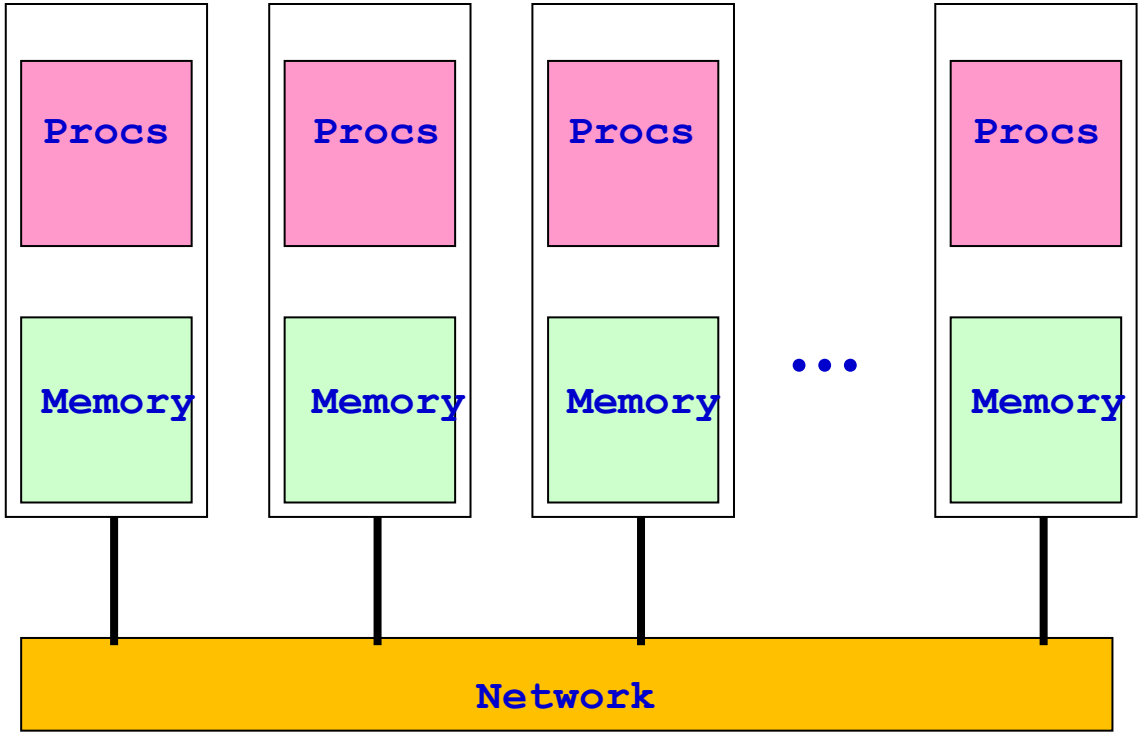
MPI를 이용한 병렬프로그래밍 기초:
Point-to-Point Communication

Hongsuk Yi

KISTI Supercomputing Center

MPI란 무엇인가?

- MPI
  - Message Passing Interface
- 병렬 프로그래밍을 위해 표준화된 데이터 통신 라이브러리
  - MPI-1 표준 마련(MPI Forum) : 1994년



MPI 포럼

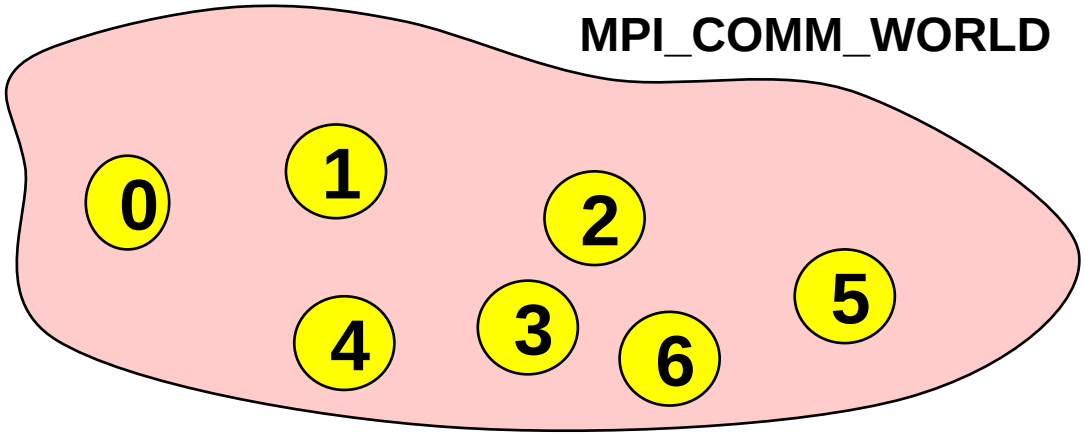
- **MPI Forum**
  - MPI 표준 제정
  - MPI 1.0 – June,1994.
  - MPI 1.1 – June 12, 1995.
  - MPI-2 - July 18, 1997

MPI의 목적과 범위

- MPI의 목적은…
  - MPI를 제공
  - 소스코드 이식성 보장
  - 효율적인 구현을 가능하도록
- MPI에는…
  - 많은 기능들이 포함
  - 이질적인 병렬 아키텍처를 위한 지원 (Grid 환경등)
- MPI-2에는…
  - 중요한 부가적인 몇 개의 기능 추가
  - MPI-1에는 변화가 없음

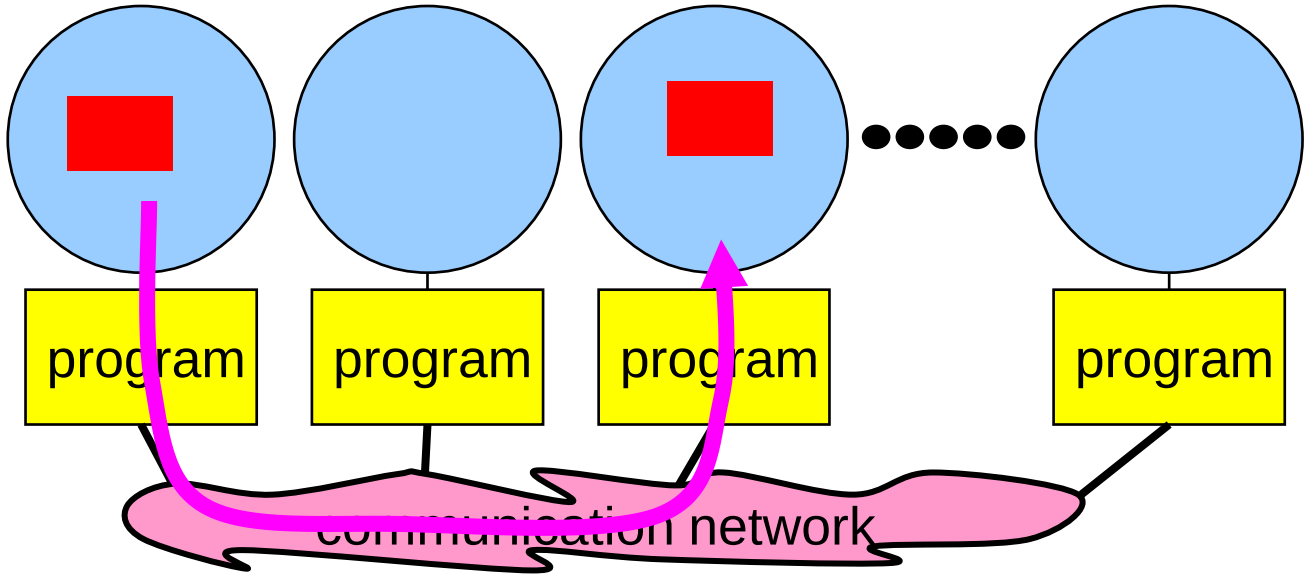
MPI_COMM_WORLD

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



Massage Passing

- 메시지 패싱은 …
  - 지역적으로 메모리를 따로 가지는 프로세스들이 데이터를 서로 공유하기 위해 메시지(데이터)를 주고 받음
    - 병렬화를 위한 작업할당, 데이터분배, 통신의 운용 등 모든 것을 프로그래머가 담당으로, 코딩은 어렵지만 유용성 좋음(Very Flexible)



MPI 메시지

◻ MPI 데이터

- 특정 MPI 데이터 타입을 가지는 원소들의 배열로 구성
- 송신과 수신 데이터 타입은 반드시 일치 해야 한다.

MPI Data Type	C Data Type
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

MPI의 기본 개념

- 프로세스와 프로세서
  - MPI는 프로세스 기준으로 작업할당
  - 프로세서 대 프로세스 = 일대일 또는 일대다
- 메시지
  - 어떤 프로세스가 보내는가
  - 어디에 있는 데이터를 보내는가
  - 어떤 데이터를 보내는가
  - 얼마나 보내는가
  - 어떤 프로세스가 받는가
  - 어디에 저장할 것인가
  - 얼마나 받을 준비를 해야 하는가

MPI의 기본 개념

- 꼬리표(tag)
  - 메시지 매칭과 구분에 이용
  - 순서대로 메시지 도착을 처리할 수 있음
  - 와일드 카드 사용 가능
- 커뮤니케이터(Communicator)
  - 서로간에 통신이 허용되는 프로세스들의 집합
- 프로세스 랭크(Rank)
  - 동일한 커뮤니케이터 내의 프로세스들을 식별하기 위한 식별자

MPI 헤더파일

- 헤더파일 삽입

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

- MPI 서브루틴과 함수의 프로토타입 선언
- 매크로, MPI 관련 인수, 데이터 타입 정의

MPI의 기본 개념

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

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

MPI 참고도서

- MPI: A Message-Passing Interface Standard (1.1, June 12, 1995)
- MPI-2: Extensions to the Message-Passing Interface (July 18,1997)
- MPI: The Complete Reference
- Using MPI: Portable Parallel Programming With the Message-Passing Interface
- Using MPI-2: Advanced Features of the Message-Passing Interface.
- Parallel Programming with MPI

MPI를 이용한 병렬 프로그래밍 기초

필수 MPI Commands : 6개

- int MPI_Init(int *argc, char **argv)
- int MPI_Finalize(void)
- int MPI_Comm_size(MPI_Comm comm, int *size)
- int MPI_Comm_rank(MPI_Comm comm, int *rank)
- int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
- int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

시작과 끝

- **MPI_Init(int *argc, char **argv)**
  - MPI 루틴 중 가장 먼저 오직 한 번 반드시 호출되어야 함
  - 변수 선언 후 바로 다음에 위치 MPI 환경 초기화

- **MPI_Finalize(void)**
  - 코드의 마지막 끝에 위치
  - 모든 MPI 자료구조 정리
  - 모든 프로세스들에서 마지막으로 한 번 호출되어야 함

MPI 프로세스 설정

- **MPI_Comm_size(MPI_Comm comm, int *size)**
  - 커뮤니케이터에 포함된 프로세스들의 총 개수
  - 커뮤니케이터 사이즈 가져오기

- **MPI_Comm_rank(MPI_Comm comm, int *rank)**
  - 현재 프로세스의 ID
  - 같은 커뮤니케이터에 속한 프로세스의 식별 번호
    - 프로세스가 n개 있으면 0부터 n-1까지 번호 할당
    - $0 \le rank \le size-1$

Message Passing: Send

- MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
  - buf : 송신 버퍼의 시작 주소
  - count : 송신될 원소 개수
  - datatype : 각 원소의 MPI 데이터 타입(핸들)
  - dest : 수신 프로세스의 랭크
  - tag : 메시지 꼬리표
  - comm : MPI 커뮤니케이터(핸들)

  - MPI_Send(&x,1,MPI_DOUBLE,manager,me, MPI_COMM_WORLD)

Message Passing: Receive

- MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
  - buf : 수신 버퍼의 시작 주소
  - count : 수신될 원소 개수
  - datatype : 각 원소의 MPI 데이터 타입(핸들)
  - source : 송신 프로세스의 랭크
  - tag : 메시지 꼬리표
  - comm : MPI 커뮤니케이터(핸들)
  - status(MPI_STATUS_SIZE) : 수신된 메시지의 정보 저장

블록킹 수신

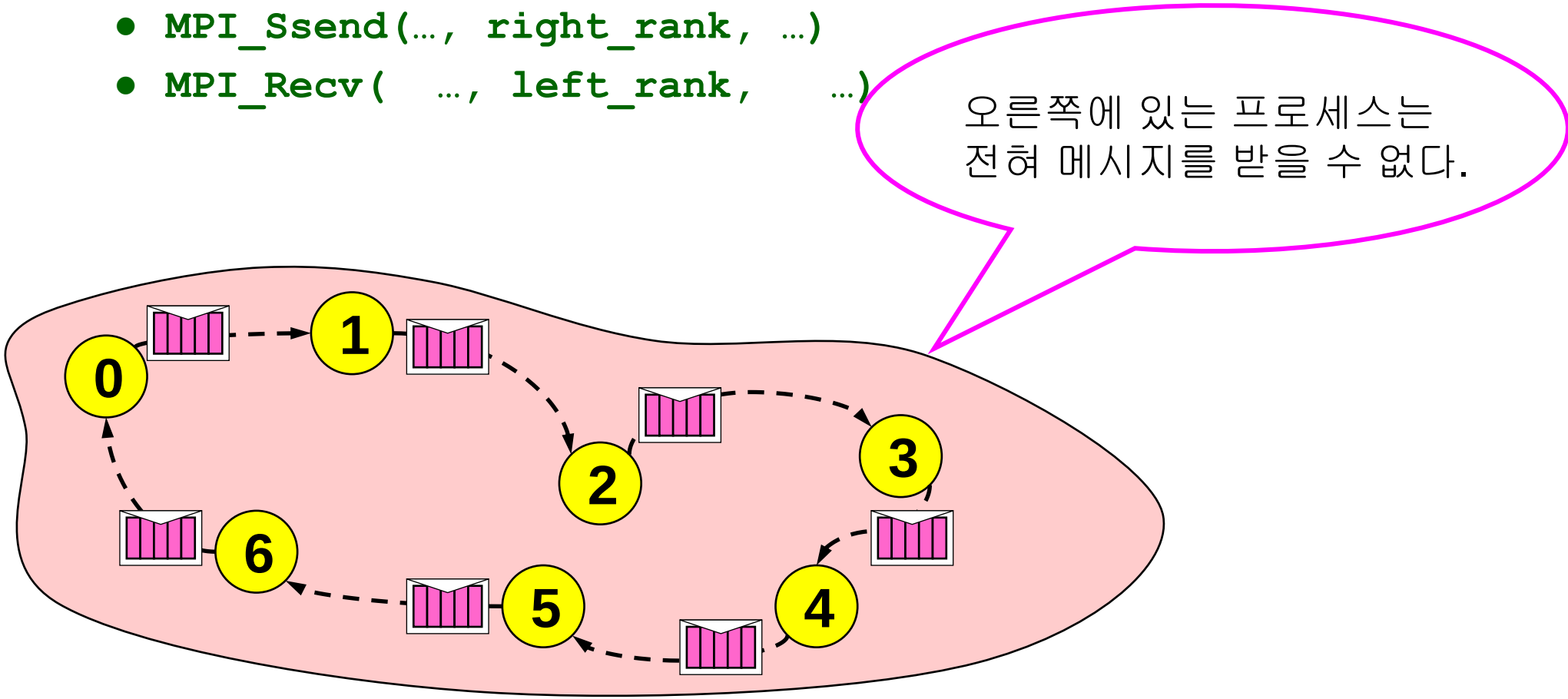
- 수신자는 와일드 카드를 사용할 수 있음
  - 모든 프로세스로부터 메시지 수신 : **MPI_ANY_SOURCE**
- 어떤 꼬리표를 단 메시지든 모두 수신
  - **MPI_ANY_TAG**
- 수신자의 **status** 인수에 저장되는 정보
  - 송신 프로세스, 꼬리표
- **MPI_GET_COUNT** : 수신된 메시지의 원소 개수를 리턴

Message Passing

- 블로킹 통신 : 주의 사항
  - 표준 송신과 수신은 블로킹 통신이다
  - MPI_Recv는 메시지 버퍼를 완전히 받은 후 통신 완료함
  - MPI_Send는 메시지 받을때까지 블로킹 이거나 아니거나 ..
  - 교착(deadlock)에 항상 주의 해야 함

Deadlock

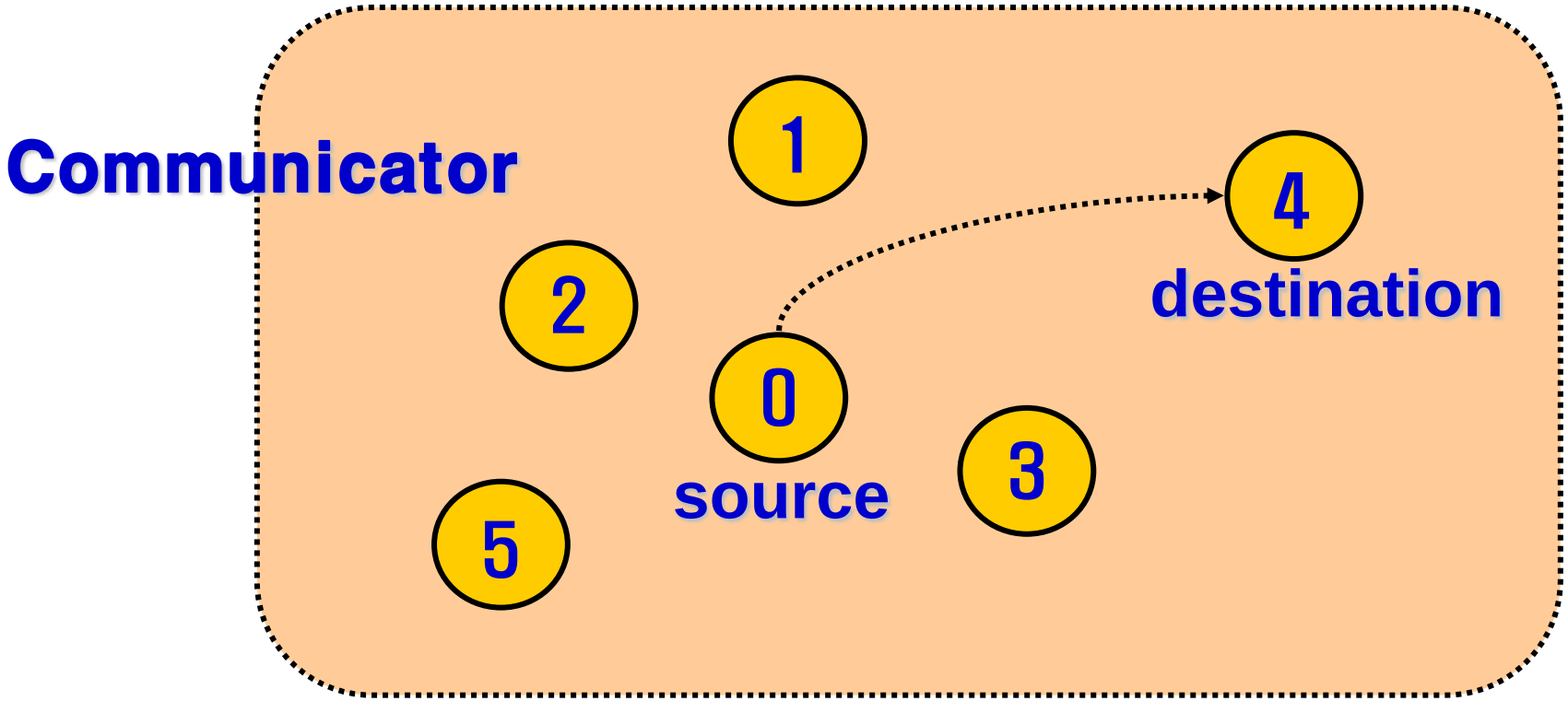
- **Code in each MPI process:**
  - `MPI_Ssend(…, right_rank, …)`
  - `MPI_Recv( …, left_rank, …)`



- 만일 MPI가 동기 프로토콜로 구현되어 있으면 표준송신 (MPI_Send) 모드에서도 교착이 발생함

점대점 통신

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



점대점 통신

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

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

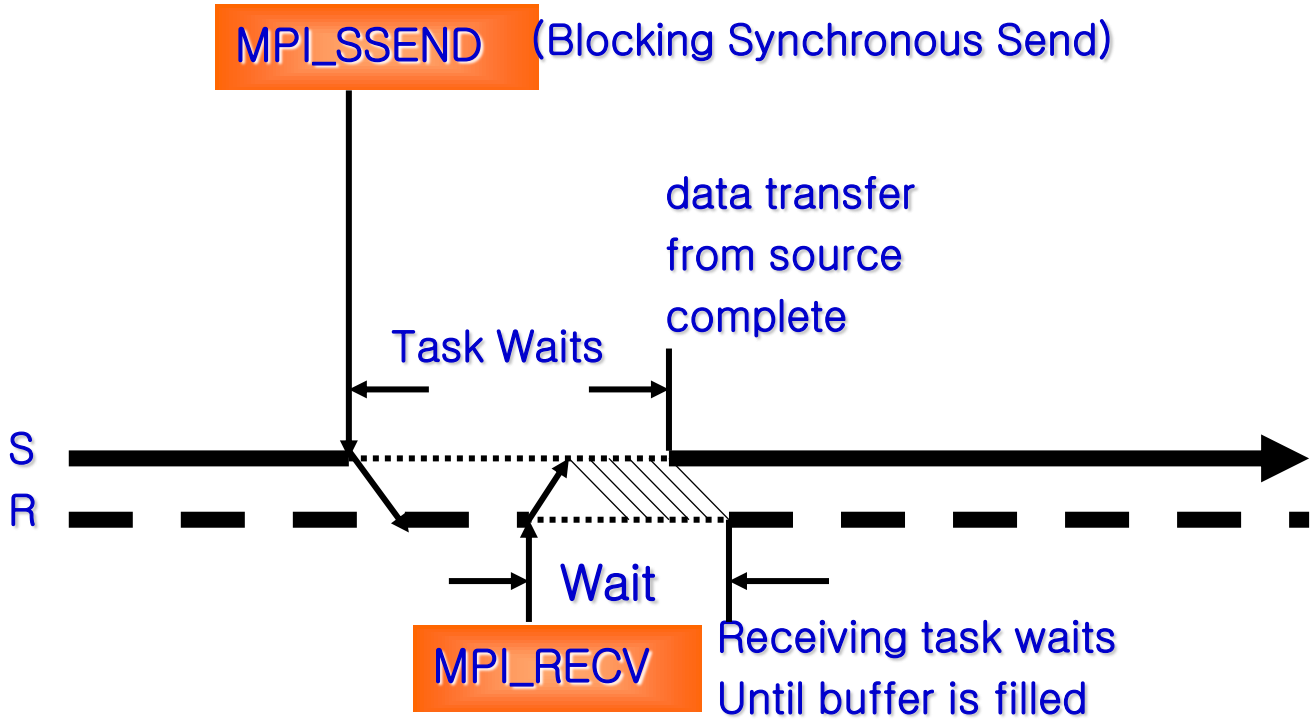
통신 모드

통신 모드	MPI 호출 루틴	
	블록킹	논블록킹
동기 송신	MPI_SSEND	MPI_ISSEND
준비 송신	MPI_RSEND	MPI_IRSEND
버퍼 송신	MPI_BSEND	MPI_IBSEND
표준 송신	MPI_SEND	MPI_ISEND
수 신	MPI_RECV	MPI_IRECV

통신 모드

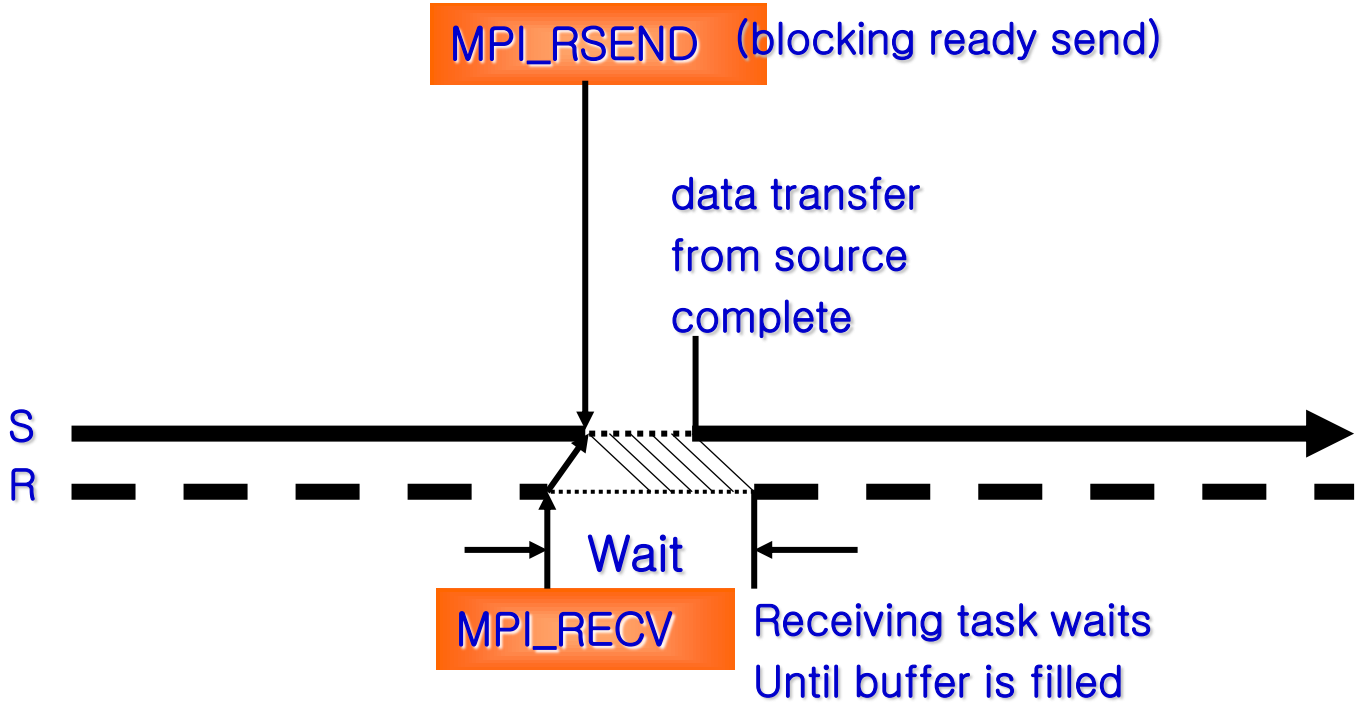
- 표준송신 : Standard send MPI_SEND)
  - minimal transfer time
  - may block due to synchronous mode
  - –> risks with synchronous send
- Synchronous send (MPI_SSEND)
  - risk of deadlock
  - risk of serialization
  - risk of waiting –> idle time
  - high latency / best bandwidth
- Buffered send (MPI_BSEND)
  - low latency / bad bandwidth
- Ready send (MPI_RSEND)
  - use never, except you have a 200% guarantee that Recv is already called in the current version and all futu re versions of your code

Synchronous Send: 동기 송신



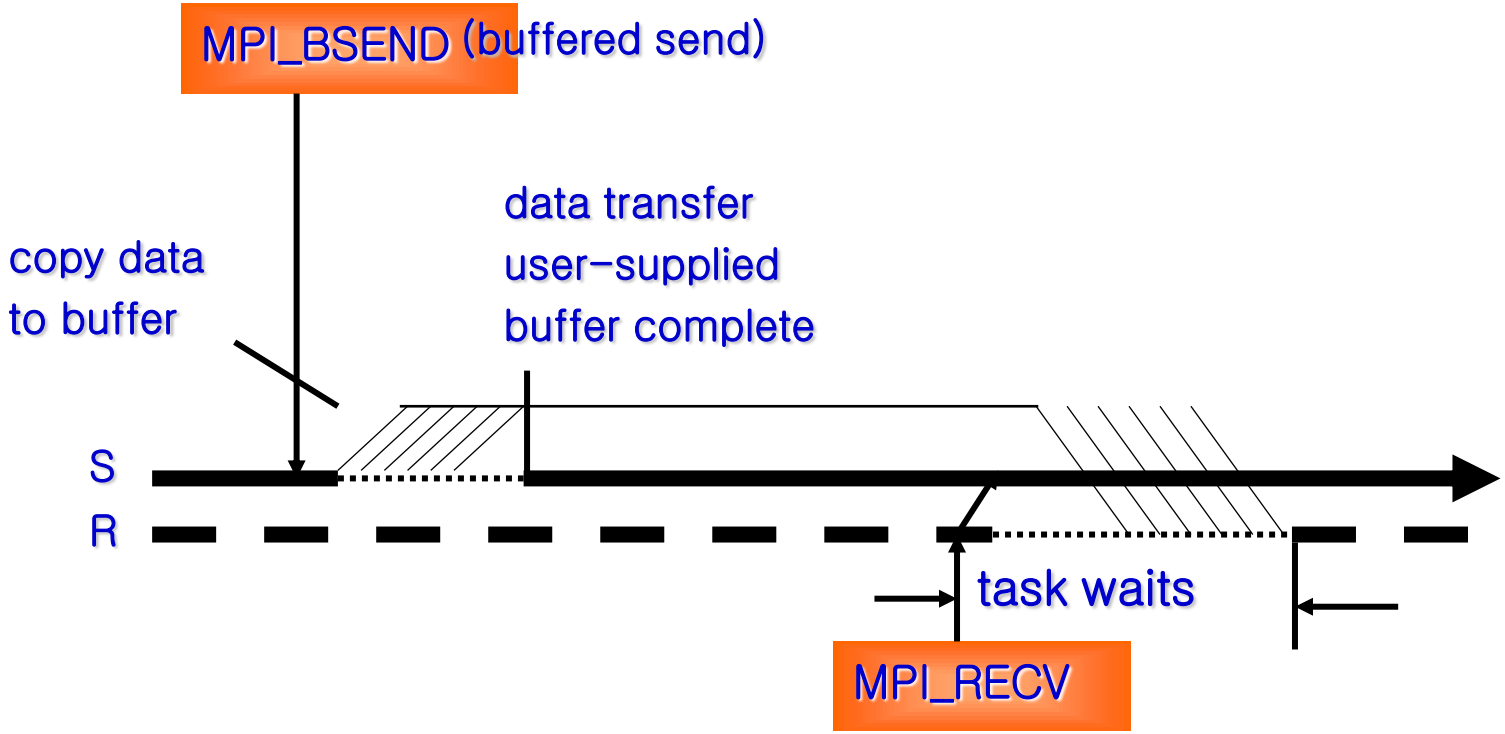
- 송신 시작 : 대응되는 수신 루틴의 실행에 무관하게 시작
- 송신 : 수신측이 받을 준비가 되면 전송시작
- 송신 완료 : 수신 루틴이 메시지를 받기 시작 + 전송 완료
- 가장 안전한 논-로컬 송신 모드

Ready Send : 준비 송신



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

Buffered Send : 버퍼 송신



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

성공적인 통신을 위해 주의할 점들

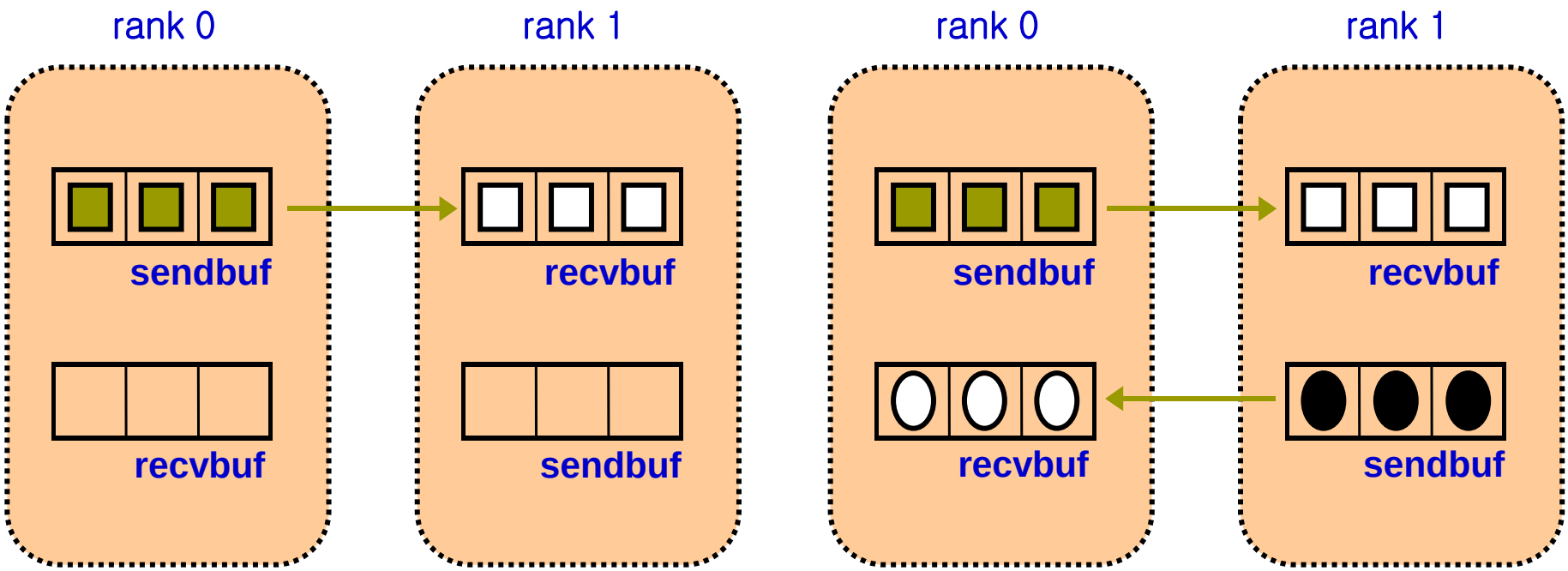
- 송신측에서 수신자 랭크를 명확히 할 것
- 수신측에서 송신자 랭크를 명확히 할 것
- 커뮤니케이터가 동일 할 것
- 메시지 꼬리표가 일치할 것
- 수신버퍼는 충분히 클 것

논블록킹 통신

- 통신을 세 가지 상태로 분류
  - 논블록킹 통신의 초기화 : 송신 또는 수신의 포스팅
  - 전송 데이터를 사용하지 않는 다른 작업 수행
    - 통신과 계산 작업을 동시 수행
  - 통신 완료 : 대기 또는 검사
- 교착 가능성 제거, 통신 부하 감소
- 대기(waiting)
  - 루틴이 호출되면 통신이 완료될 때까지 프로세스를 블록킹
  - 논블록킹 통신 + 대기 = 블록킹 통신
- 검사(testing)
  - 루틴은 통신의 완료여부에 따라 참 또는 거짓을 리턴

점대점 통신의 사용

- 단방향 통신과 양방향 통신
- 양방향 통신은 교착에 주의



단방향 통신 (1/2)

- 블록킹 송신, 블록킹 수신

```
IF (myrank==0) THEN
   CALL MPI_SEND(sendbuf, icount, MPI_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
```

- 논블록킹 송신, 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
```

단방향 통신 (2/2)

- 블록킹 송신, 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
```

- 논블록킹 송신, 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)
```

실습

1) MPI_Counting3s.c

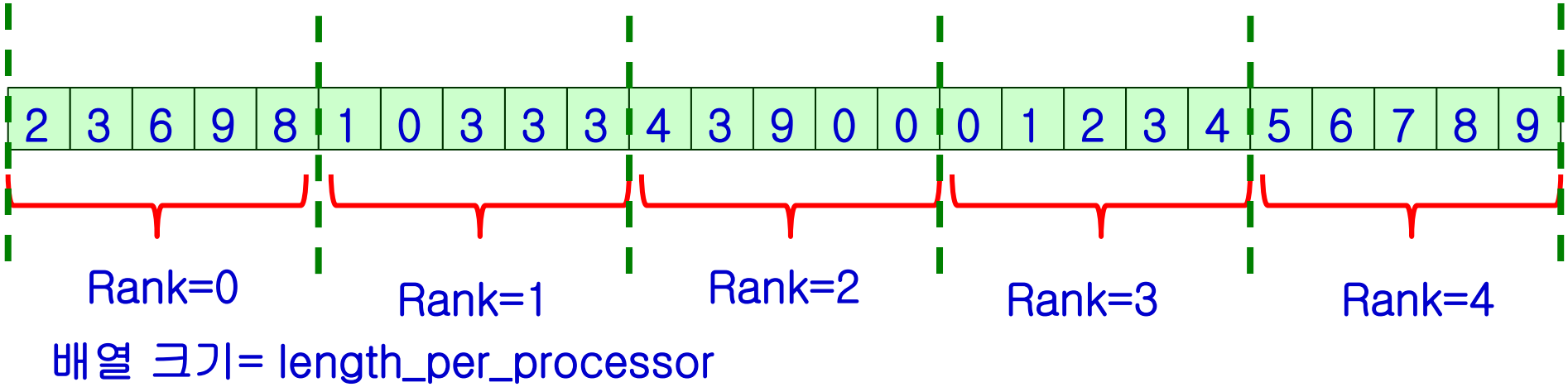
2) 점대점 통신 성능시험

Serial counting3s.c

```
#include <stdlib.h>; #include <stdio.h>   ; #include <time.h>
double dtime();
int main(int argc, char **argv)
{
    int i,j, *array, count=0;
    const int length = 100000000, iters = 10;
    double stime, etime;
    array = (int *)malloc(length * sizeof(int));
    for (i = 0; i < length; i++)
        array[i] = i % 10;
    dtime(&stime);
    for (j = 0; j < iters; j++) {
        for (i = 0; i < length; i++) {
            if (array[i] == 3) {  count++;  }
        }
    }
    dtime(&etime);
    printf("serial: Number of 3's: %d ₩t  Elapsed Time = %12.8lf (sec) ₩n ", count, etime-stime);
    return 0;
}
```

```
- tachyon190
$> ./c3s_serial.x
serial: Number of 3's: 100000000
Elapsed Time =   3.13399506 (sec)
```

mpi_counting 3s (1/4)



1) Rank=0 : 배열 초기화 globalArray[i
2) 작업 분할 : myArray[i]의 크기= length_per_processor
3) 각각의 rank로 myArray 배열 송신 (MPI_Send)
4) 각각의 rank는 myArray 수신 (MPI_Recv) 및 3s count 시작
5) 각각의 rank에서 계산된 count를 master 노드로 송신
6) Master 노드는 count를 합하여 최종 global_count를 출력
7) 시간은 MPI_Wtime()로 측정

mpi_counting3s.c (2/4)

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
int main(int argc, char **argv)  {
    const int length = 100000000,  iters = 10;
    int myid, nprocs, length_per_process, i, j, p;
    int *myArray, *gArray, root, tag, myCount, gCount;
    FILE *fp; double t1, t2;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    myCount = 0; root=0;   tag=0;
    length_per_process=length/nprocs;
    myArray=(int *)malloc(length_per_process*sizeof(int));
```

mpi_counting3s.c (3/4)

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

mpi_counting3s.c (4/4)

```
 for (j=0; j< iters; j++){
     for(i=0; i<length_per_process; i++) {
         if(myArray[i]==3)
             myCount++;
        }
    }
MPI_Reduce(&myCount,&gCount,1,MPI_INT,MPI_SUM root, MPI_COMM_WORLD);
  if(myid==root) {
     t2 = MPI_Wtime();
    printf("nprocs=%d Number of 3's: %d Elapsed Time =%12.8lf(sec)\n",
nprocs, gCount, t2-t1);
  }
    MPI_Finalize();
}
```

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

예제: mpi_multibandwidth.c

```
case 1:
MPI_Send(&msgbuf1, n, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
            MPI_Recv(&msgbuf1, n, MPI_CHAR, src, tag, MPI_COMM_WORLD, stats);
case 2:
MPI_Send(&msgbuf1, n, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
            MPI_Irecv(&msgbuf1, n, MPI_CHAR, src, tag,
MPI_COMM_WORLD,&reqs[0]);
            MPI_Wait(&reqs[0], stats);

case 3:
MPI_Isend(&msgbuf1, n, MPI_CHAR, dest, tag,MPI_COMM_WORLD,&reqs[0]);
            MPI_Irecv(&msgbuf1, n, MPI_CHAR, src,
tag,MPI_COMM_WORLD,&reqs[1]);
            MPI_Waitall(2, reqs, stats);
case 4:
MPI_Ssend(&msgbuf1, n, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
            MPI_Recv(&msgbuf1, n, MPI_CHAR, src, tag, MPI_COMM_WORLD, stats);
```

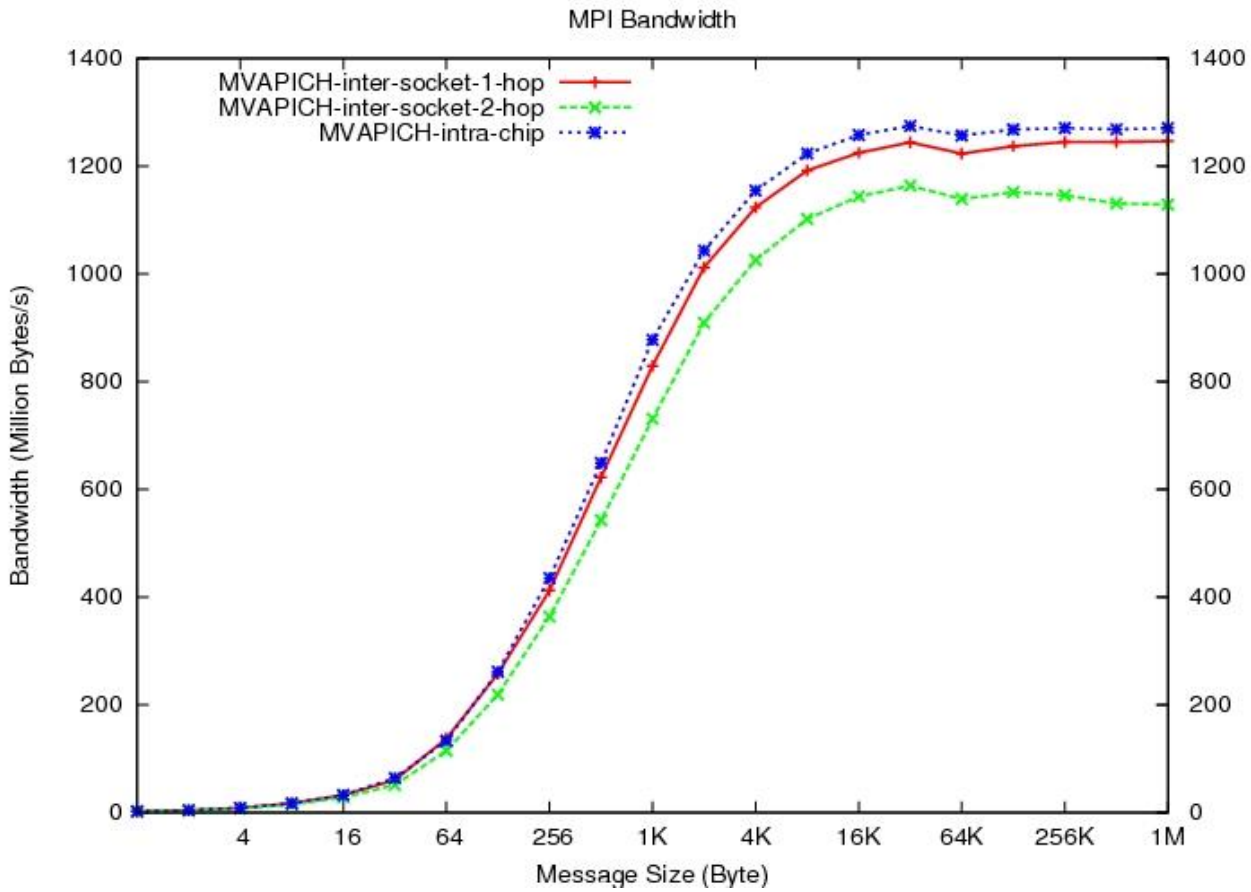
LAST REVISED: 12/27/2001 Blaise Barney

Point-to-Point 통신

```
case 5:
MPI_Ssend(&msgbuf1, n, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
MPI_Irecv(&msgbuf1, n, MPI_CHAR, src, tag, MPI_COMM_WORLD,&reqs[0]);
MPI_Wait(&reqs[0], stats);
case 6:
MPI_Sendrecv(&msgbuf1, n, MPI_CHAR, dest, tag, &msgbuf2, n,
                        MPI_CHAR, src, tag, MPI_COMM_WORLD, stats);
case 7:
MPI_Issend(&msgbuf1, n, MPI_CHAR, dest,tag,MPI_COMM_WORLD,&reqs[0]);
           MPI_Irecv(&msgbuf1, n, MPI_CHAR, src, tag,
MPI_COMM_WORLD,&reqs[1]);
MPI_Waitall(2, reqs, stats);
case 8:
MPI_Issend(&msgbuf1, n, MPI_CHAR, dest,tag,MPI_COMM_WORLD,&reqs[0]);
MPI_Recv(&msgbuf1, n, MPI_CHAR, src, tag, MPI_COMM_WORLD, stats);
MPI_Wait(&reqs[0], stats);
case 9:
MPI_Isend(&msgbuf1, n, MPI_CHAR, dest,tag,MPI_COMM_WORLD,&reqs[0]);
MPI_Recv(&msgbuf1, n, MPI_CHAR, src, tag, MPI_COMM_WORLD, stats);
MPI_Wait(&reqs[0], stats);
```

Time and Bandwidth

```
t1 = MPI_Wtime();
MPI_Isend(&msgbuf1, n,MPI_CHAR,dest,tag,MPI_COMM_WORLD,&reqs[0]);
MPI_Recv(&msgbuf1, n, MPI_CHAR, src, tag, MPI_COMM_WORLD, stats);
MPI_Wait(&reqs[0], stats);
t2 = MPI_Wtime(); thistime = t2 - t1;
bw = ((double)nbytes * 2) / thistime;
```



Bw~1200 MB/s

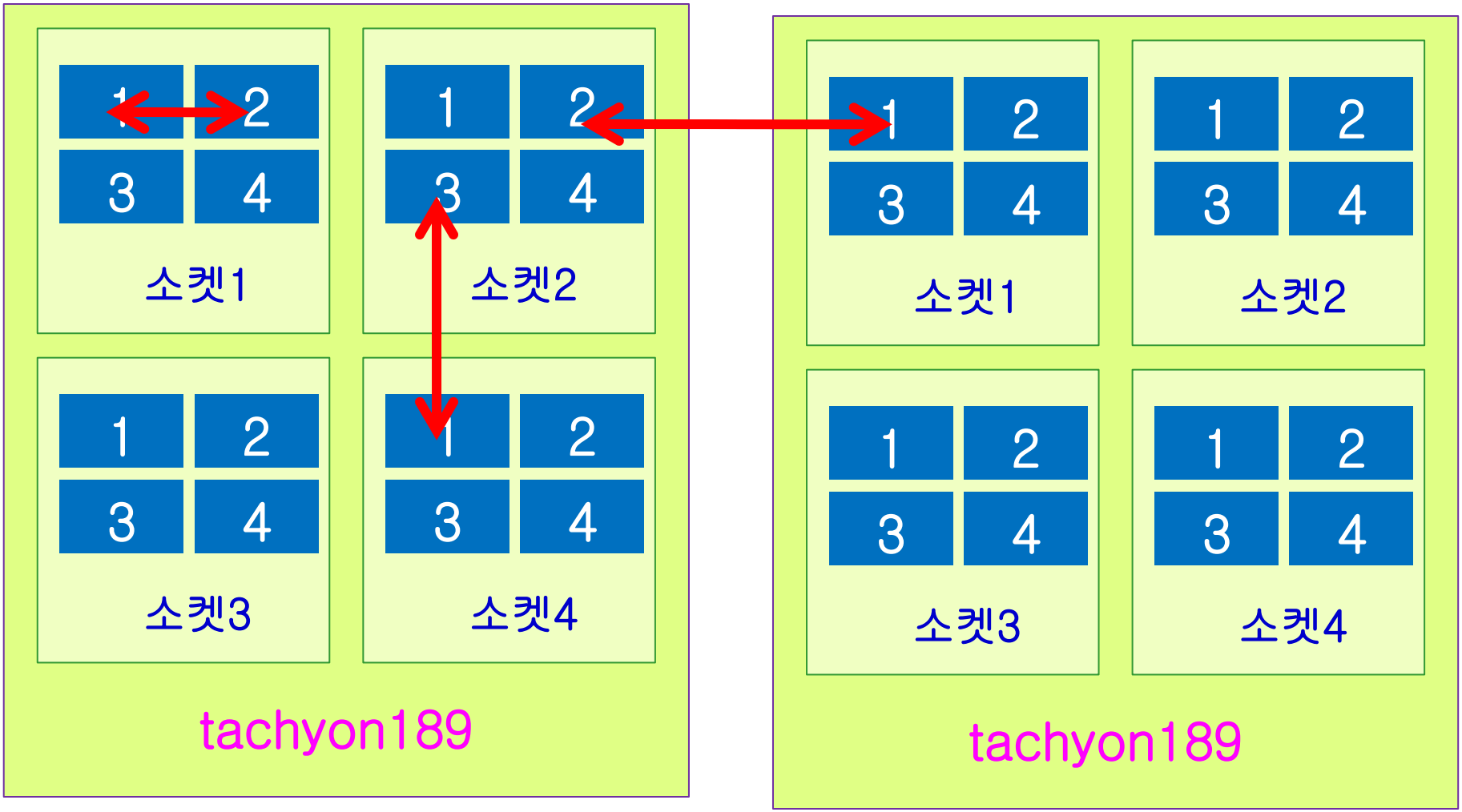
Output: Mpi_multibandwidth.c

```
**** MPI/POE Bandwidth Test ***
Message start size= 104857600 bytes
Message finish size= 104857600 bytes
Incremented by 100000 bytes per iteration
Roundtrips per iteration= 10
MPI_Wtick resolution = 1.000000e-05
*************************
task    0 is on tachyon192 partner=   1
task    1 is on tachyon192 partner=   0
************************************
```

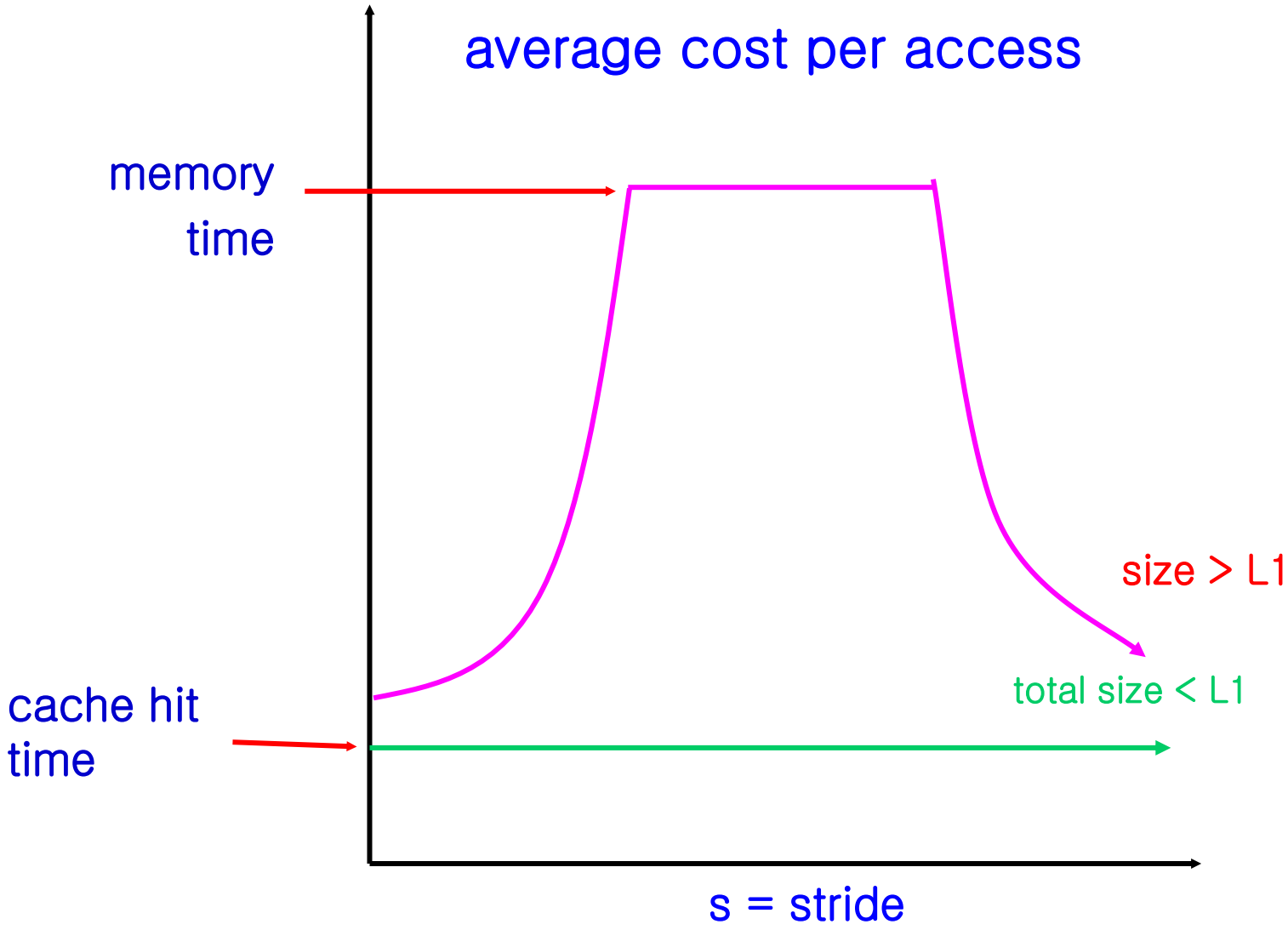
```
*** Case 1: Send with Recv
   Message size: 104857600     avg (MB/sec)
   task pair:    0 -    1:  1066.148
*** Case 2: Send with Irecv
   Message size: 104857600     avg (MB/sec)
   task pair:    0 -    1:  1063.166
*** Case 3: Isend with Irecv
   Message size: 104857600     avg (MB/sec)
   task pair:    0 -    1:  1062.683
*** Case 4: Ssend with Recv
   Message size: 104857600     avg (MB/sec)
   task pair:    0 -    1:  1069.341
*** Case 5: Ssend with Irecv
   Message size: 104857600     avg (MB/sec)
   task pair:    0 -    1:  1065.343
*** Case 6: Sendrecv
   Message size: 104857600     avg (MB/sec)
   task pair:    0 -    1:  1070.993
*** Case 7: Issend with Irecv
   Message size: 104857600     avg (MB/sec)
   task pair:    0 -    1:  1065.081
*** Case 8: Issend with Recv
```

Project 2 : Bandwidth & Latency

- 멀티코어 Barcelona Chip의 Bandwidth를 측정하기

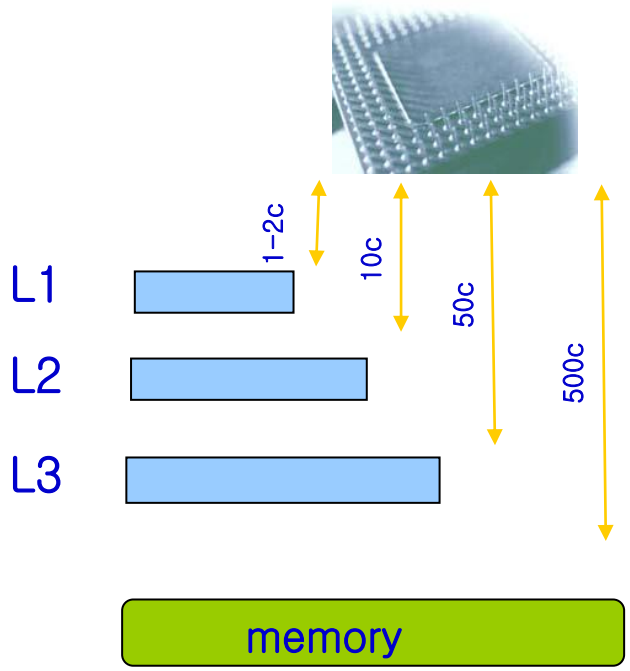


Membench: What to Expect

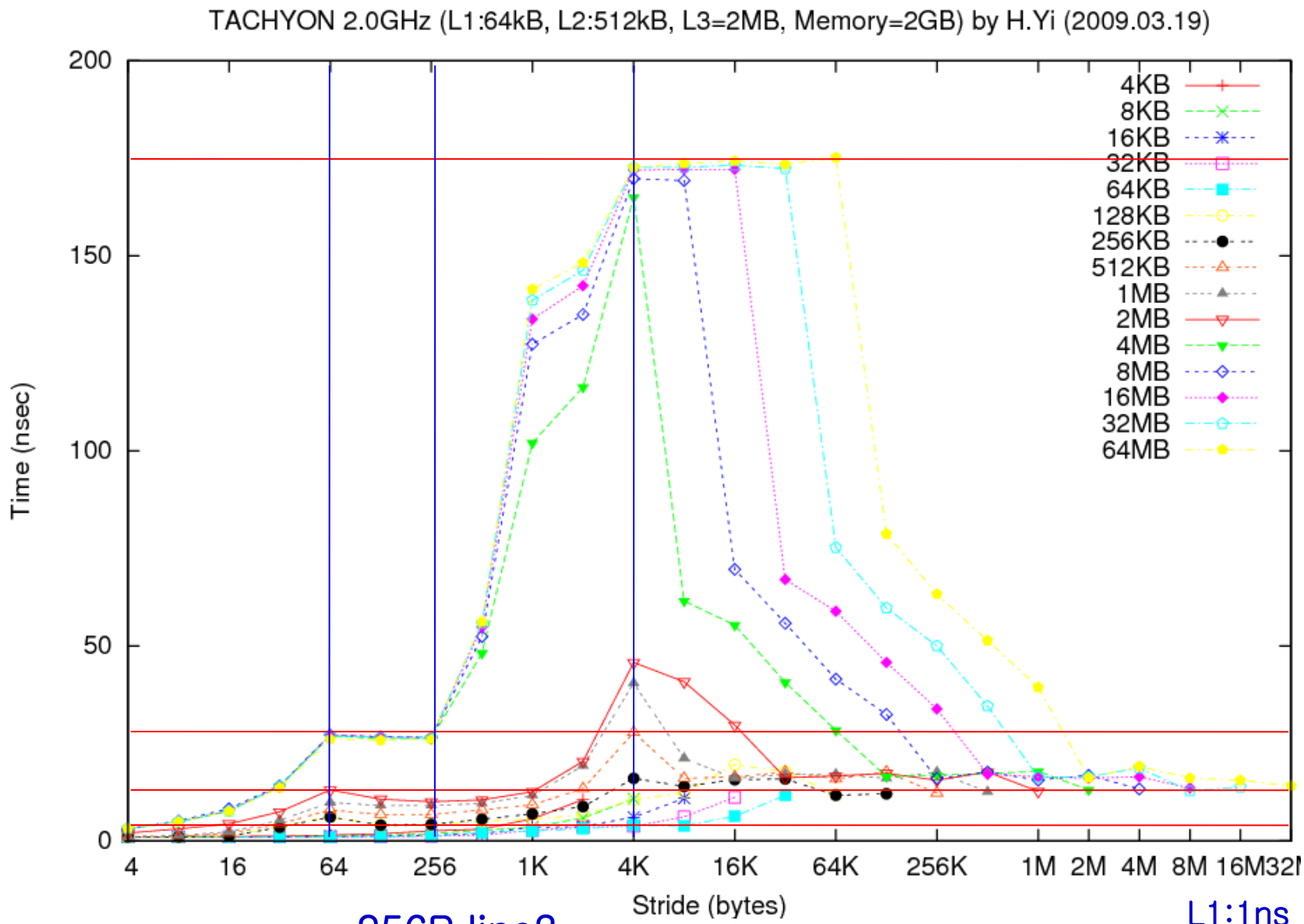


Core-Memory 속도차이: AMD 2350

- L1 (64kB)
  - 1.5ns : 3 cycle
- L2 (512kB)
  - 15 cycle
    - 3cycle (L1)
    - 3(L1-L2)
    - 9cycle(L2 only)
- L3 cache (2MB)
  - 47 cycle
    - 15cyle(L2)
    - 9cycle(L2-L3)
    - 23 cycle (L3 only)



AMD 2350 Membenchmark



Memory
:176 ns
(0.17us)

L3:24ns
48cycle

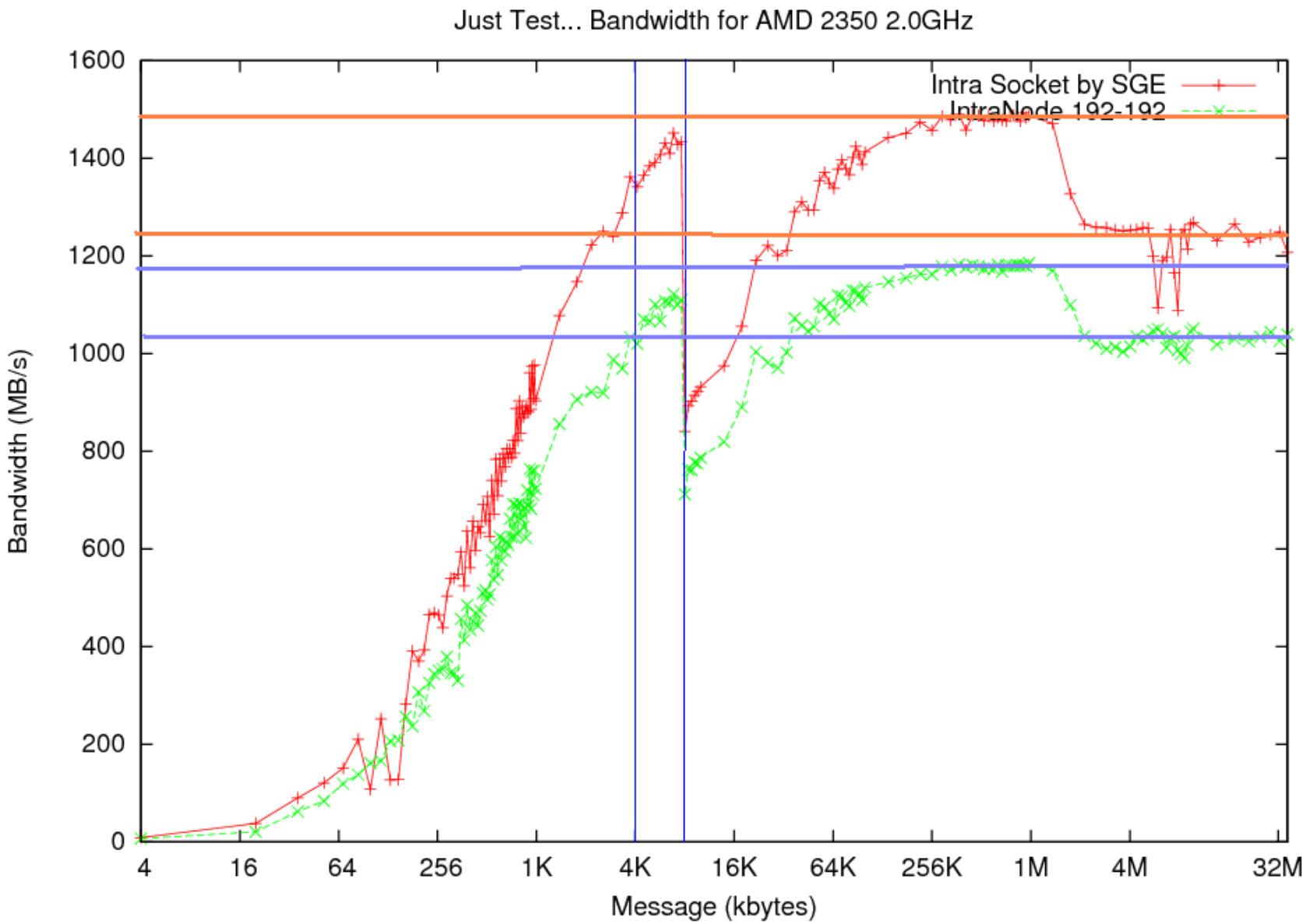
L2:8ns
(16cycle)

L1:1ns
(2cycle)

256B line?

4k page TLB?

Bandwidth


Just Test... Bandwidth for AMD 2350 2.0GHz
Intra Socket by SGE
IntraNode 192-192
Bandwidth (MB/s)
1600
1400
1200
1000
800
600
400
200
0
4
16
64
256
1K
4K
16K
64K
256K
1M
4M
32M
Message (kbytes)

Q & A