㈜컴퓨팅브릿지

일반 병렬 프로그래밍

김정한

COMPUTING bridge

목표

병렬 프로그래밍을 하기 위해 필요한 사전 지식을
전달하고, OpenMP와 MPI의 개념 및 사용법을 소개하여
간단한 병렬 프로그램을 직접 작성할 수 있게 한다.

INDEX

COMPUTING bridge

01. 병렬 프로그래밍 개요

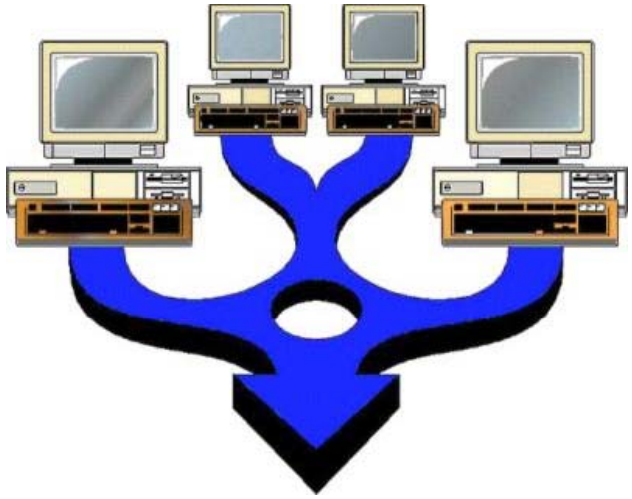
병렬 프로그래밍과 관련된 개념을 정리하고
병렬 프로그래밍 모델, 성능측정, 그리고
병렬 프로그램의 작성과정에 대해 알아본다.

병렬 처리 (1/3)

병렬 처리란, 순차적으로 진행되는 계산영역을

여러 개로 나누어 각각을 여러 프로세서에서

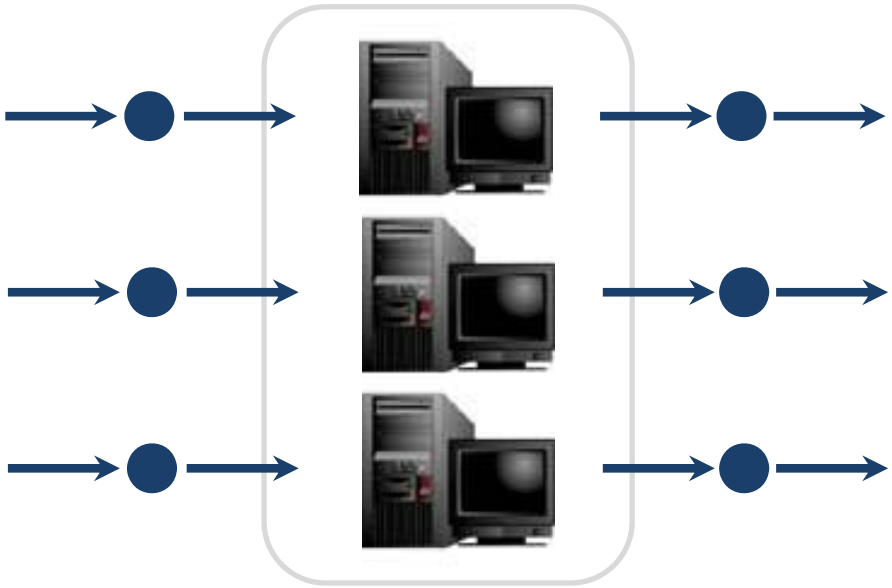
동시에 수행 되도록 하는 것



병렬 처리 (2/3)

- 순차실행

- 병렬실행



병렬 처리 (3/3)

- 주된 목적 : 더욱 큰 문제를 더욱 빨리 처리하는 것
  - 프로그램의 wall-clock time 감소
  - 해결할 수 있는 문제의 크기 증가

- 병렬 컴퓨팅 계산 자원
  - 여러 개의 프로세서(CPU)를 가지는 단일 컴퓨터
  - 네트워크로 연결된 다수의 컴퓨터

왜 병렬인가?

- 고성능 단일 프로세서 시스템 개발의 제한
  - 전송속도의 한계 (구리선 : 9 cm/nanosec)
  - 소형화의 한계
  - 경제적 제한

- 보다 빠른 네트워크, 분산 시스템, 다중 프로세서 시스템 아키텍처의 등장 ➔ 병렬 컴퓨팅 환경

- 상대적으로 값싼 프로세서를 여러 개 묶어 동시에 사용함으로써 원하는 성능이득 기대

프로그램과 프로세스

- 프로세스는 보조 기억 장치에 하나의 파일로서 저장되어 있던 실행 가능한 프로그램이 로딩되어 운영체제(커널)의 실행 제어 상태에 놓인 것

  - 프로그램 : 보조 기억 장치에 저장
  - 프로세스 : 컴퓨터 시스템에 의하여 실행 중인 프로그램
  - 태스크 = 프로세스

프로세스

- 프로그램 실행을 위한 자원 할당의 단위가 되고, 한 프로그램에서 여러 개 실행 가능

- 다중 프로세스를 지원하는 단일 프로세서 시스템
  - 자원 할당의 낭비, 문맥교환으로 인한 부하 발생
  - 문맥교환
    - 어떤 순간 한 프로세서에서 실행 중인 프로세스는 항상 하나
    - 현재 프로세스 상태 저장 → 다른 프로세스 상태 적재

- 분산메모리 병렬 프로그래밍 모델의 작업할당 기준

스레드

- 프로세스에서 실행의 개념만을 분리한 것
  - 프로세스 = 실행단위(스레드) + 실행환경(공유자원)
  - 하나의 프로세스에 여러 개 존재가능
  - 같은 프로세스에 속한 다른 스레드와 실행환경을 공유

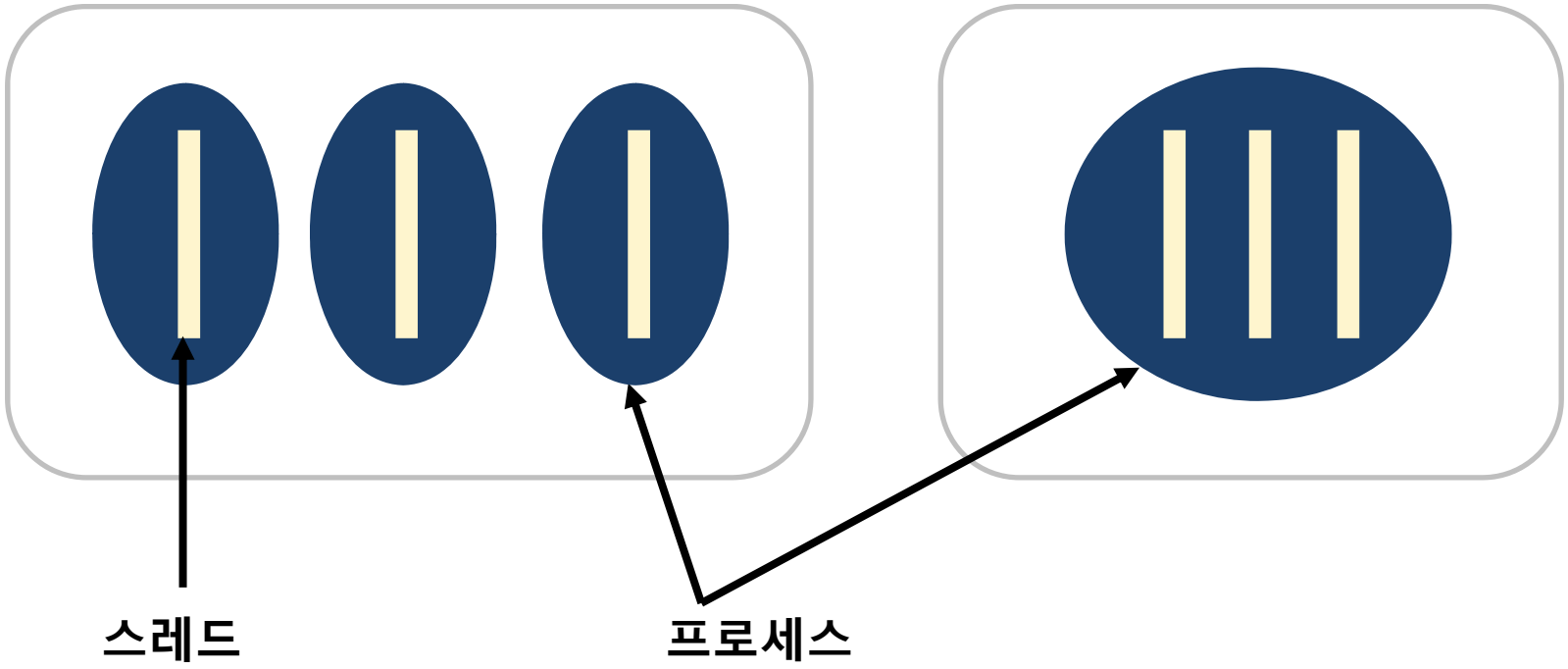
- 다중 스레드를 지원하는 단일 프로세서 시스템
  - 다중 프로세스보다 효율적인 자원 할당
  - 다중 프로세스보다 효율적인 문맥교환

- 공유 메모리 병렬 프로그래밍 모델의 작업할당 기준

프로세스와 스레드

하나의 스레드를 갖는 3개의 프로세스

3개의 스레드를 갖는 하나의 프로세스

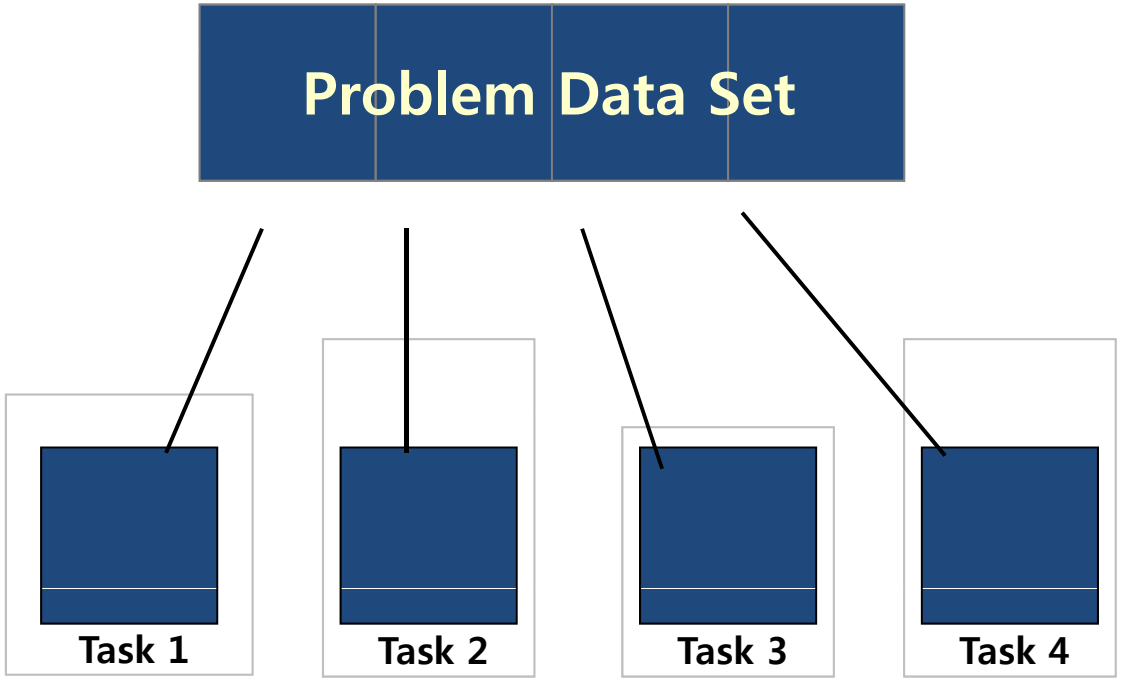


병렬성 유형

- 데이터 병렬성 (Data Parallelism)
  - 도메인 분해 (Domain Decomposition)
  - 각 태스크는 서로 다른 데이터를 가지고 동일한 일련의 계산을 수행
- 태스크 병렬성 (Task Parallelism)
  - 기능적 분해 (Functional Decomposition)
  - 각 태스크는 같거나 또는 다른 데이터를 가지고 서로 다른 계산을 수행

데이터 병렬성 (1/3)

- 데이터 병렬성 : 도메인 분해



데이터 병렬성 (2/3)

- 코드 예) : 행렬의 곱셈 (OpenMP)

Serial Code	Parallel Code
DO K=1,N DO J=1,N DO I=1,N C(I,J) = C(I,J) + (A(I,K)*B(K,J)) END DO END DO END DO	!$OMP PARALLEL DO DO K=1,N DO J=1,N DO I=1,N C(I,J) = C(I,J) + A(I,K)*B(K,J) END DO END DO END DO !$OMP END PARALLEL DO

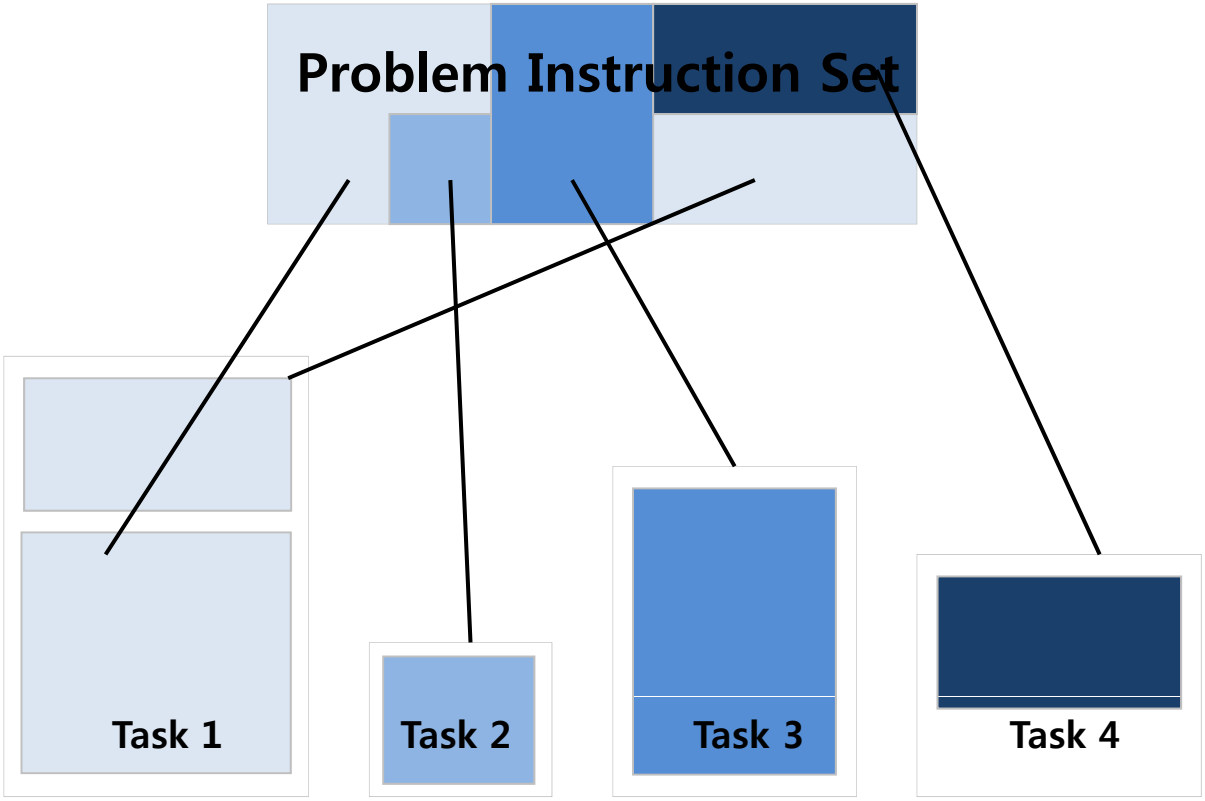
데이터 병렬성 (3/3)

- 데이터 분해 (프로세서 4개:K=1,20일 때)

Process	Iterations of K	Data Elements
Proc0	K = 1:5	A(I,1:5) B(1:5,J)
Proc1	K = 6:10	A(I,6:10) B(6:10,J)
Proc2	K = 11:15	A(I,11:15) B(11:15,J)
Proc3	K = 16:20	A(I,16:20) B(16:20,J)

태스크 병렬성 (1/3)

- 태스크 병렬성 : 기능적 분해



태스크 병렬성 (2/3)

- 코드 예) : (OpenMP)

Serial Code	Parallel Code

PROGRAM MAIN
  ...
  CALL interpolate()
  CALL compute_stats()
  CALL gen_random_params()
  ...
END
	PROGRAM MAIN
	  ...
	  !$OMP PARALLEL
	  !$OMP SECTIONS
	  CALL interpolate()
	  !$OMP SECTION
	  CALL compute_stats()
	  !$OMP SECTION
	  CALL gen_random_params()
	  !$OMP END SECTIONS
	  !$OMP END PARALLEL
	  ...
	END

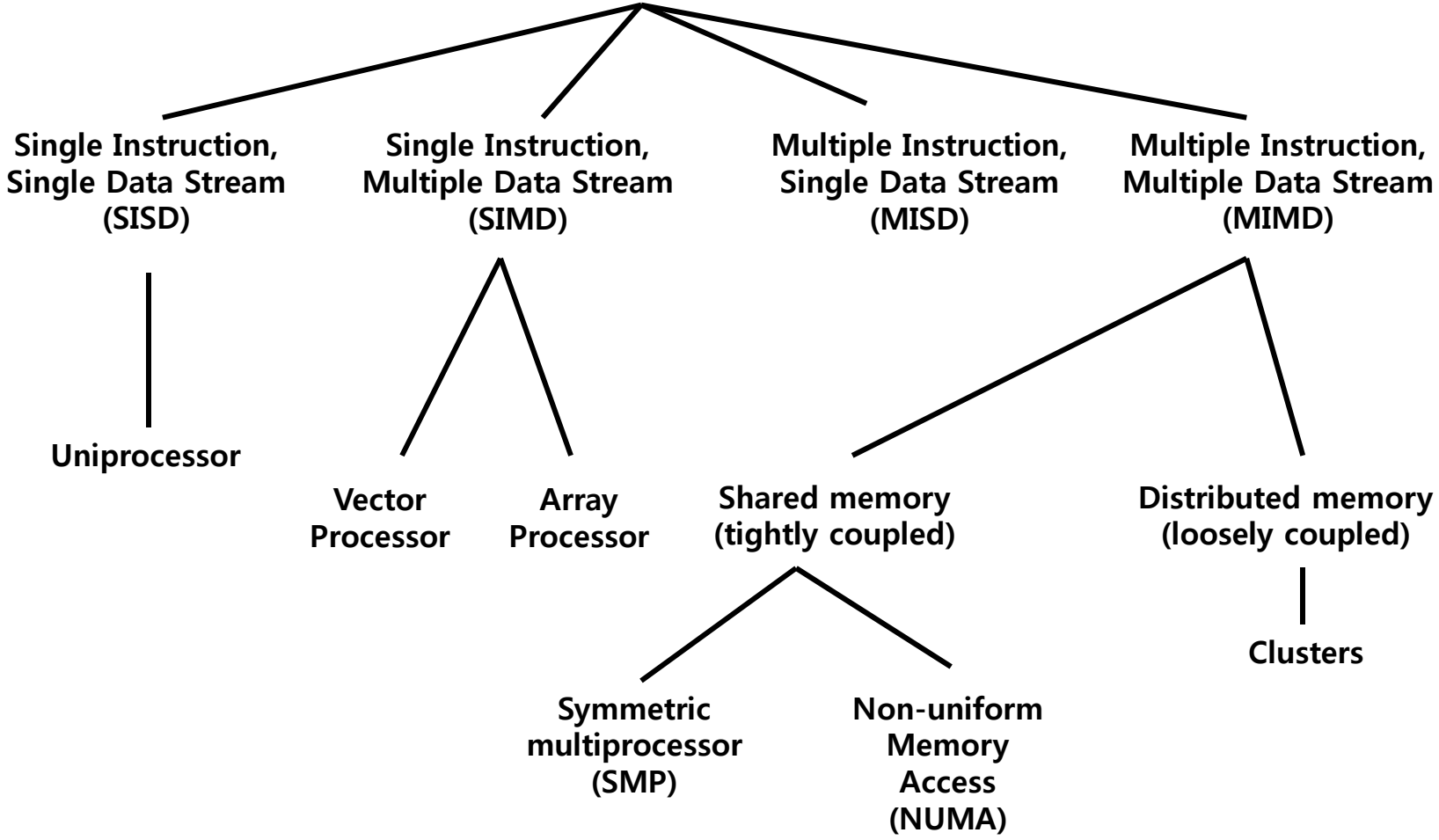
태스크 병렬성 (3/3)

- 태스크 분해 (3개의 프로세서에서 동시 수행)

Process	Code
Proc0	CALL interpolate()
Proc1	CALL compute_stats()
Proc2	CALL gen_random_params()

병렬 아키텍처 (1/2)

Processor Organizations



병렬 아키텍처 (2/2)

최근의 고성능 시스템 : 분산-공유 메모리 지원

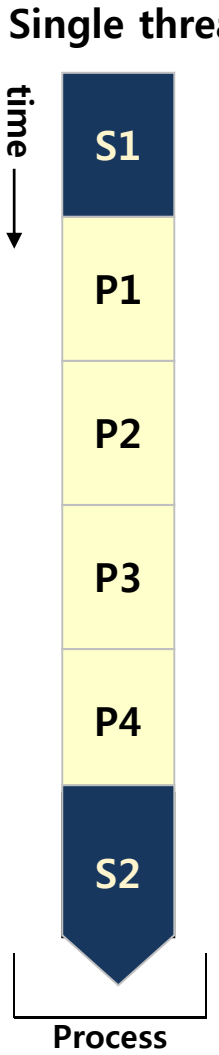
- 소프트 웨어적 DSM (Distributed Shared Memory) 구현
  - 공유 메모리 시스템에서 메시지 패싱 지원
  - 분산 메모리 시스템에서 변수 공유 지원

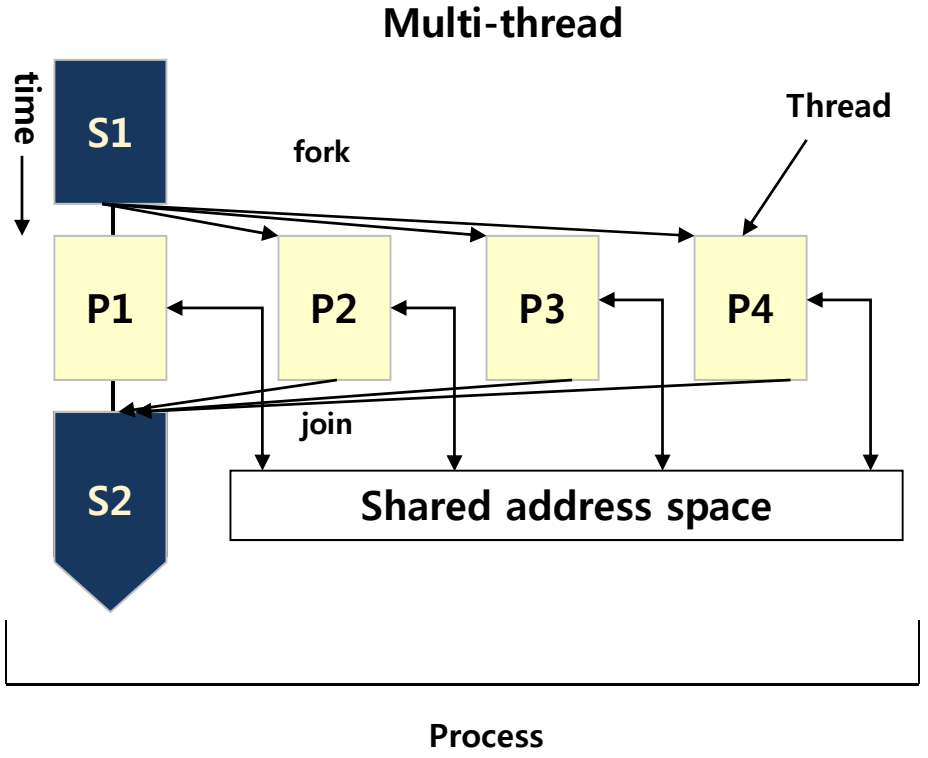
- 하드웨어적 DSM 구현 : 분산-공유 메모리 아키텍처
  - 분산 메모리 시스템의 각 노드를 공유 메모리 시스템으로 구성
  - NUMA : 사용자들에게 하나의 공유 메모리 아키텍처로 보여짐
    ex) Superdome(HP), Origin 3000(SGI)
  - SMP 클러스터 : SMP로 구성된 분산 시스템으로 보여짐
    ex) SP(IBM), Beowulf Clusters

병렬 프로그래밍 모델

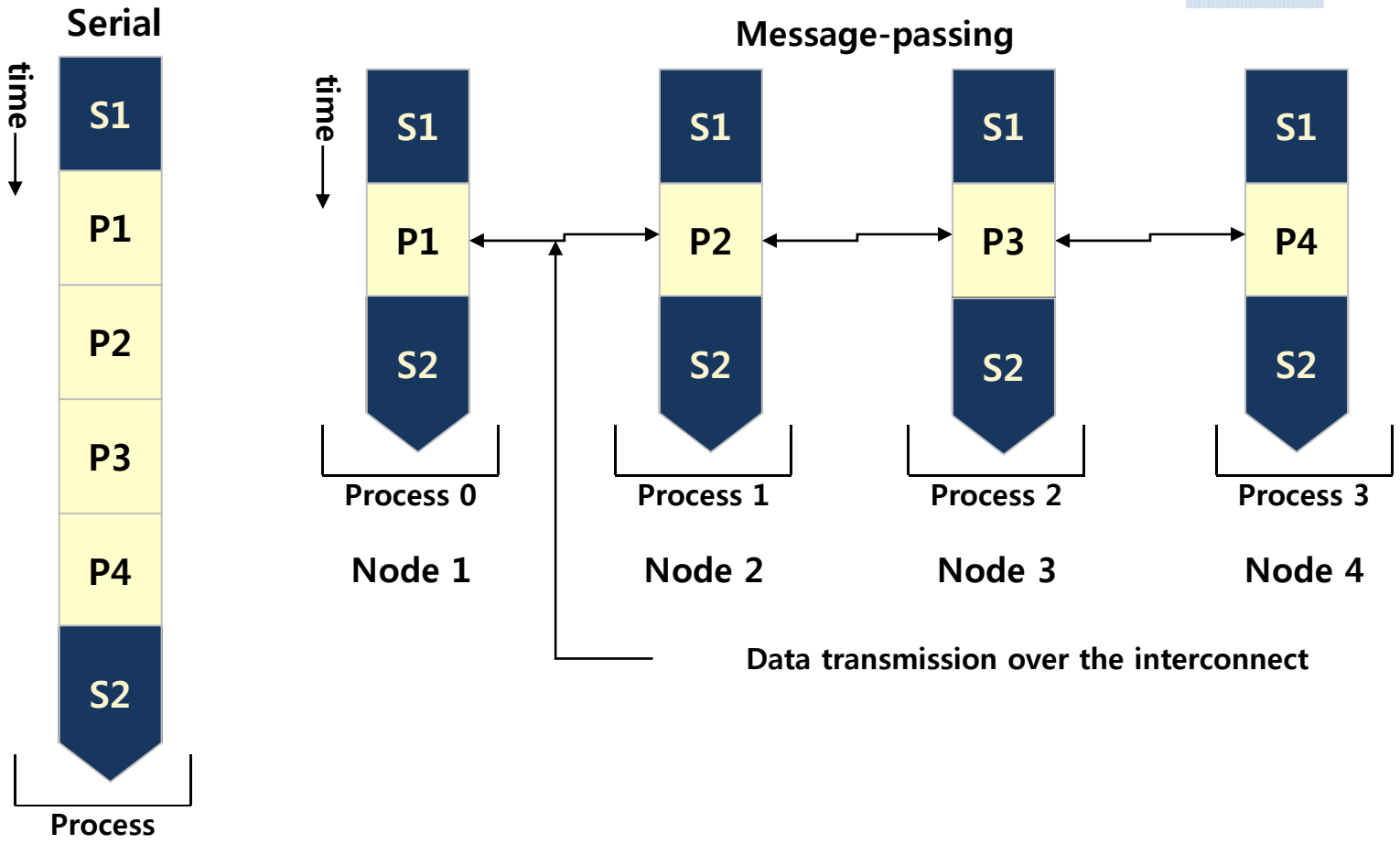
- 공유메모리 병렬 프로그래밍 모델
  - 공유 메모리 아키텍처에 적합
  - 다중 스레드 프로그램
  - OpenMP, Pthreads
- 메시지 패싱 병렬 프로그래밍 모델
  - 분산 메모리 아키텍처에 적합
  - MPI, PVM
- 하이브리드 병렬 프로그래밍 모델
  - 분산-공유 메모리 아키텍처
  - OpenMP + MPI

공유 메모리 병렬 프로그래밍 모델

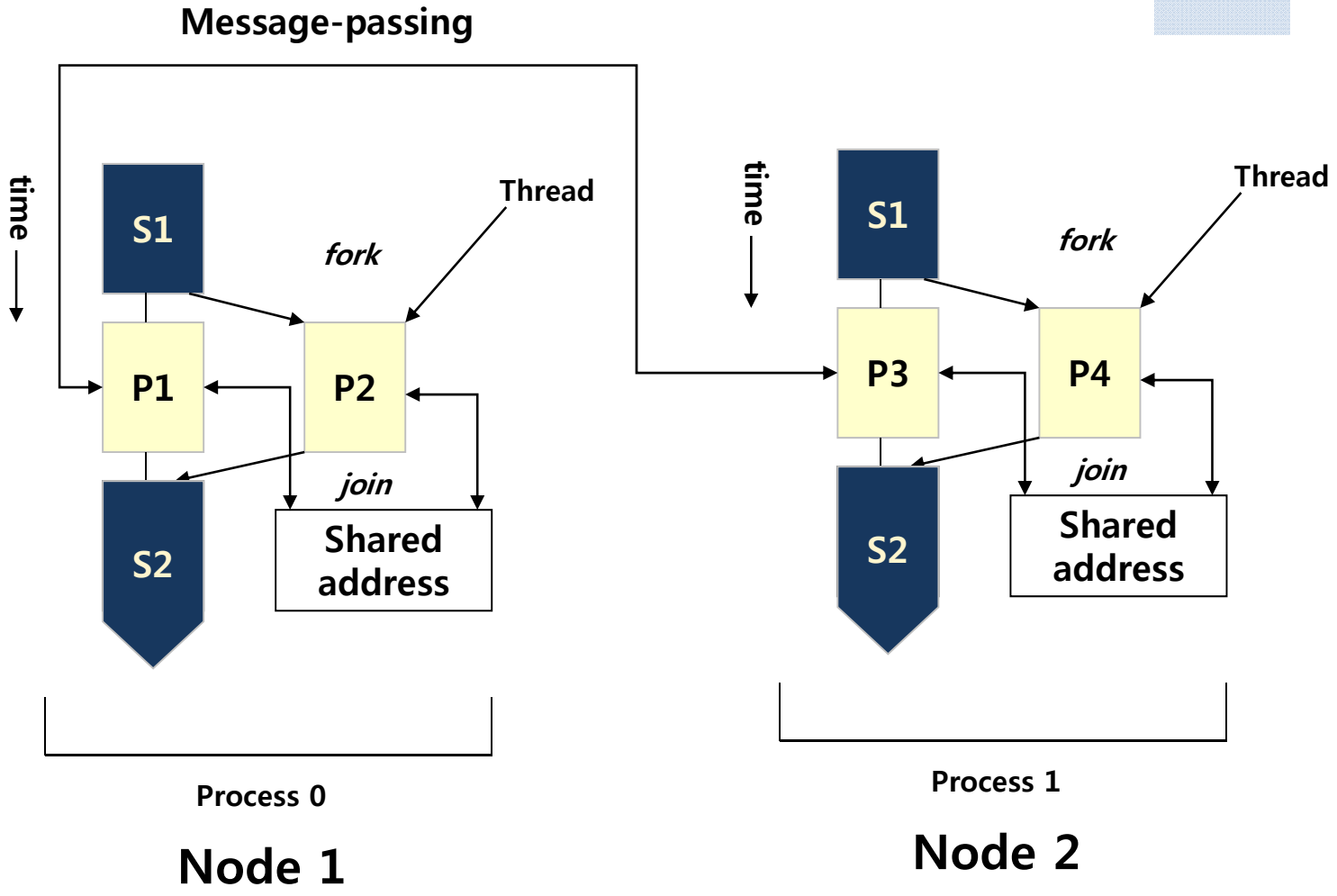




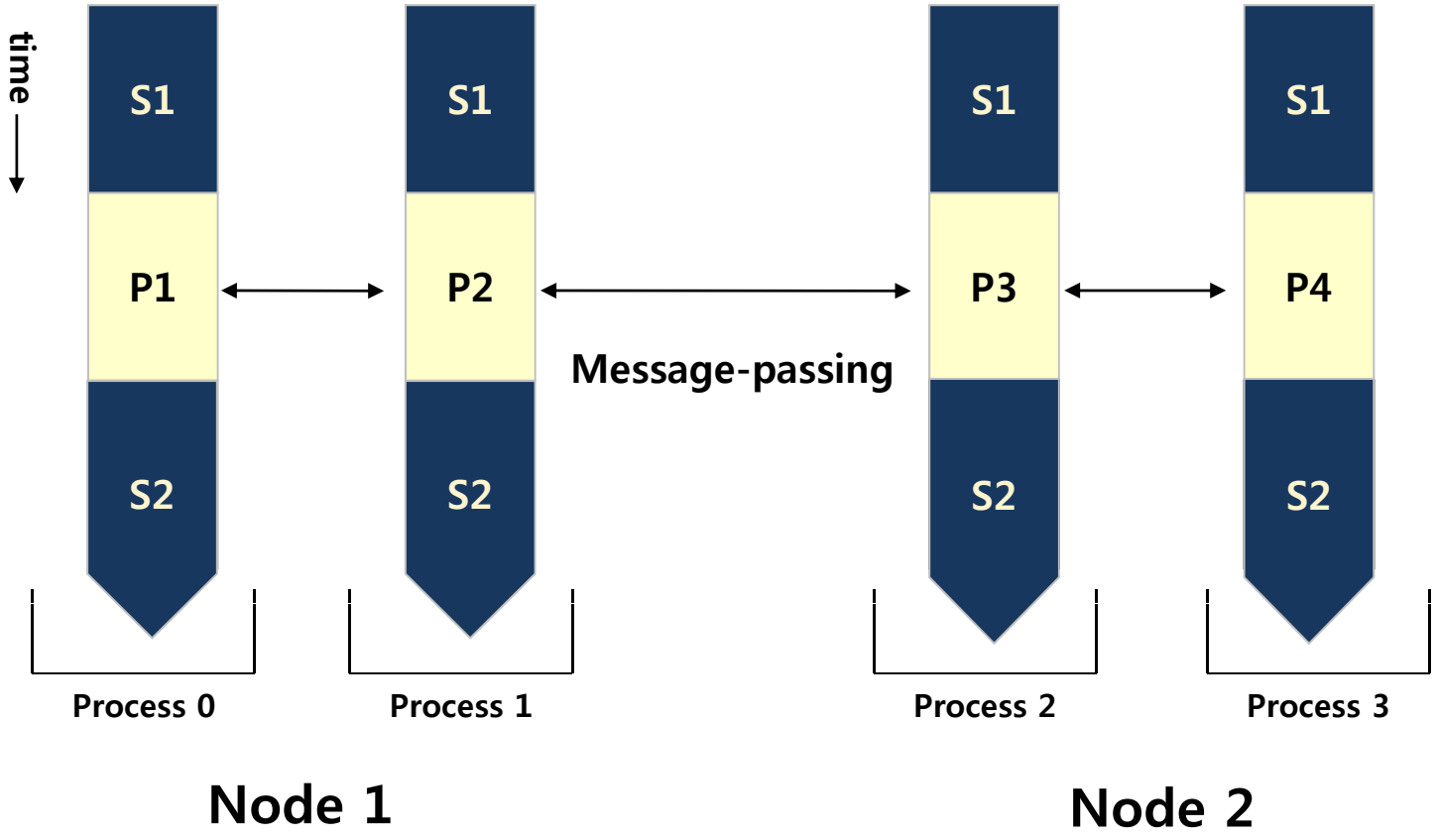
메시지 패싱 병렬 프로그래밍 모델



하이브리드 병렬 프로그래밍 모델



DSM 시스템의 메시지 패싱



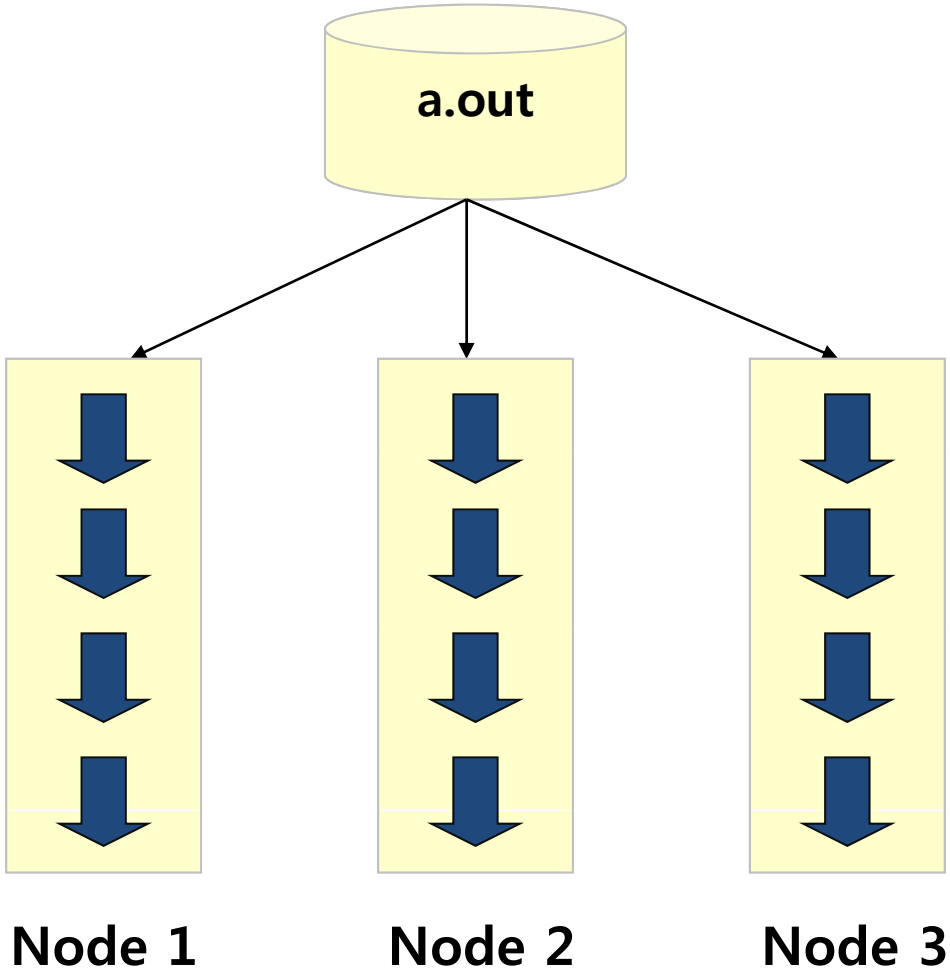
SPMD와 MPMD (1/4)

- **SPMD(Single Program Multiple Data)**
  - 하나의 프로그램이 여러 프로세스에서 동시에 수행됨
  - 어떤 순간 프로세스들은 같은 프로그램내의 명령어들을 수행하며 그 명령어들은 같을 수도 다를 수도 있음

- **MPMD (Multiple Program Multiple Data)**
  - 한 MPMD 응용 프로그램은 여러 개의 실행 프로그램으로 구성
  - 응용프로그램이 병렬로 실행될 때 각 프로세스는 다른 프로세스와 같거나 다른 프로그램을 실행할 수 있음

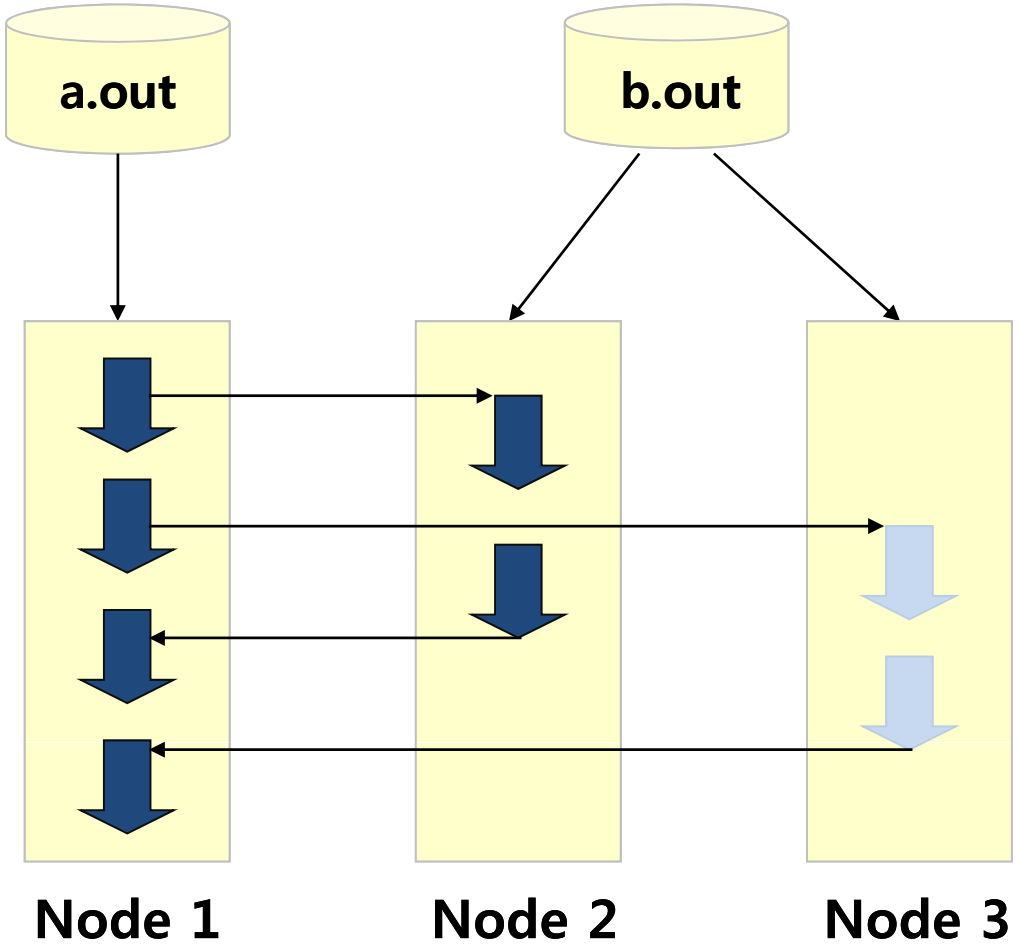
SPMD와 MPMD (2/4)

- SPMD



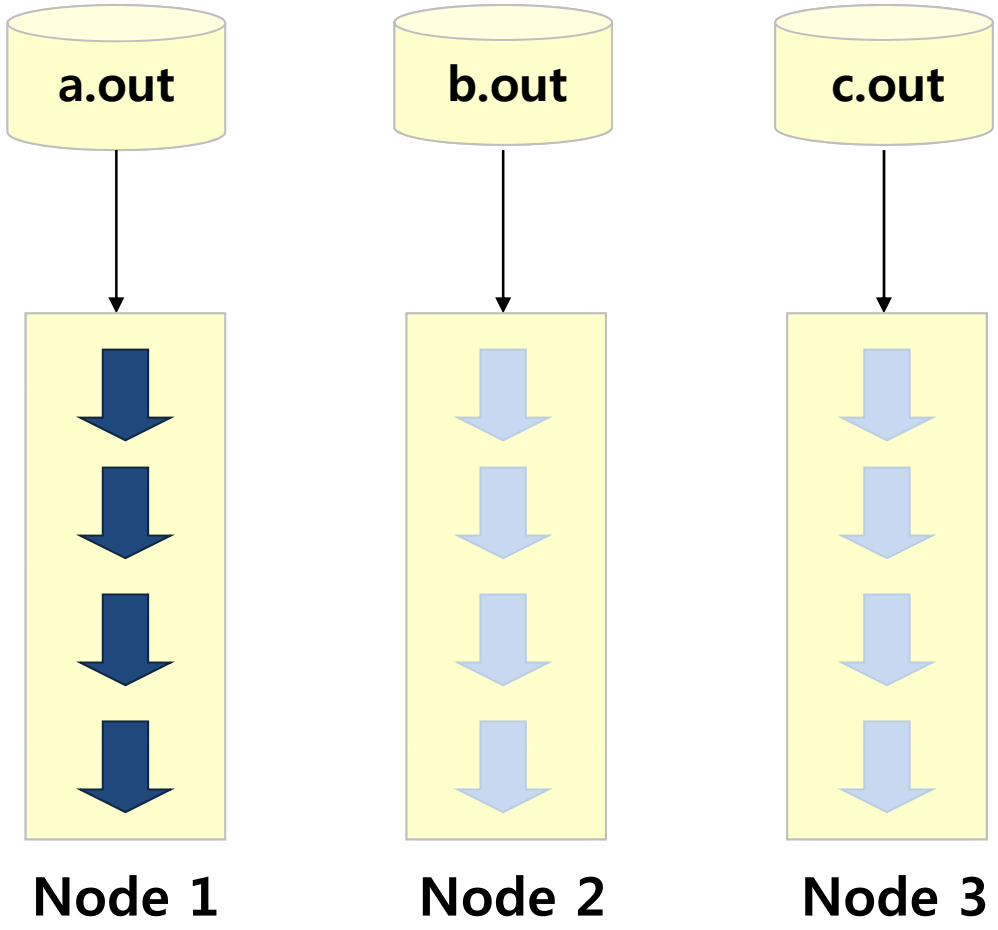
SPMD와 MPMD (3/4)

- MPMD : Master/Worker (Self-Scheduling)



SPMD와 MPMD (4/4)

- **MPMD : Coupled Analysis**



- 성능측정
- 성능에 영향을 주는 요인들
- 병렬 프로그램 작성순서

프로그램 실행시간 측정 (1/2)

- Time
- 사용방법(bash, ksh) : $time [executable]

```
$ time mpirun –np 4 –machinefile machines ./exmpi.x
real   0m3.59s
user   0m3.16s
sys    0m0.04s
```

  - real = wall-clock time
  - User = 프로그램 자신과 호출된 라이브러리 실행에 사용된 CPU 시간
  - Sys = 프로그램에 의해 시스템 호출에 사용된 CPU 시간
  - user + sys = CPU time

프로그램 실행시간 측정 (2/2)

- 사용방법(csh) : $time [executable]

```
$ time testprog
1.150u 0.020s 0:01.76 66.4% 15+3981k 24+10io 0pf+0w
   ①      ②      ③      ④       ⑤         ⑥      ⑦  ⑧
```

① user CPU time (1.15초)
② system CPU time (0.02초)
③ real time (0분 1.76초)
④ real time에서 CPU time이 차지하는 정도(66.4%)
⑤ 메모리 사용 : Shared (15Kbytes) + Unshared (3981Kbytes)
⑥ 입력(24 블록) + 출력(10 블록)
⑦ no page faults
⑧ no swaps

성능측정

- 병렬화를 통해 얻어진 성능이득의 정량적 분석
- 성능측정
  - 성능향상도
  - 효율
  - Cost

성능향상도 (1/7)

성능향상도 (Speed-up) : S(n)

$$S(n) = \frac{\text{순차 프로그램의 실행시간}}{\text{병렬 프로그램의 실행시간(n개 프로세서)}} = \frac{t_s}{t_p}$$

- 순차 프로그램에 대한 병렬 프로그램의 성능이득 정도
- 실행시간 = Wall-clock time
- 실행시간이 100초가 걸리는 순차 프로그램을 병렬화 하여 10개의 프로세서로 50초 만에 실행 되었다면,

➔ $S(10) = \frac{100}{50} = 2$

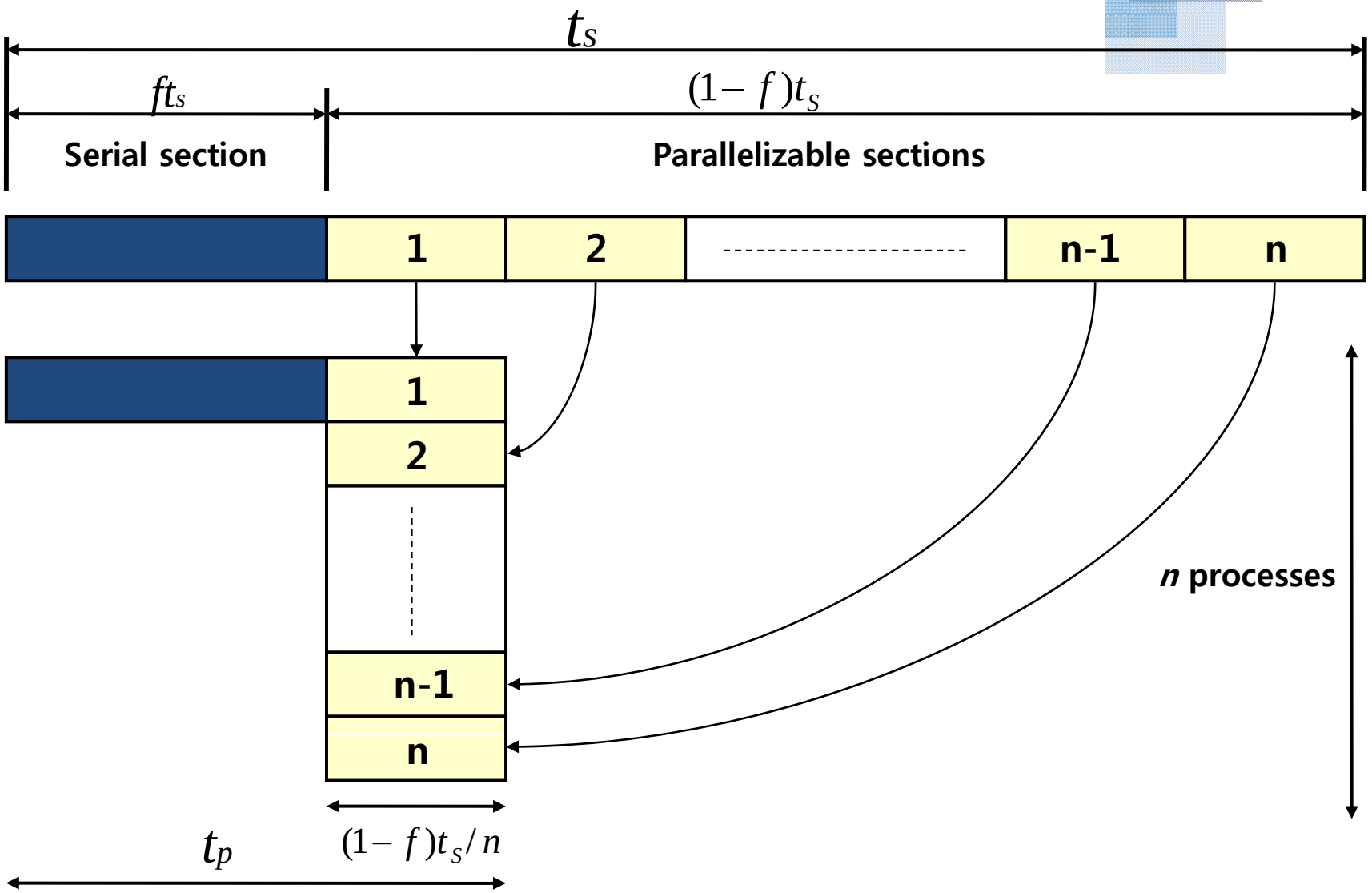
성능향상도 (2/7)

- 이상(Ideal) 성능향상도 : Amdahl's Law
  - f : 코드의 순차부분 ($0 \le f \le 1$)
  - $t_p = ft_s + (1-f)t_s/n$

순차부분 실행시간

병렬부분 실행시간

성능향상도 (3/7)


t_s
ft_s
$(1-f)t_s$
Serial section
Parallelizable sections
1
2
n-1
n
1
2
n-1
n
n processes
t_p
$(1-f)t_s/n$

성능향상도 (4/7)

- $S(n) = \dfrac{t_s}{t_p} = \dfrac{t_s}{ft_s + (1-f)t_s/n}$

$$S(n) = \frac{1}{f + (1-f)/n}$$

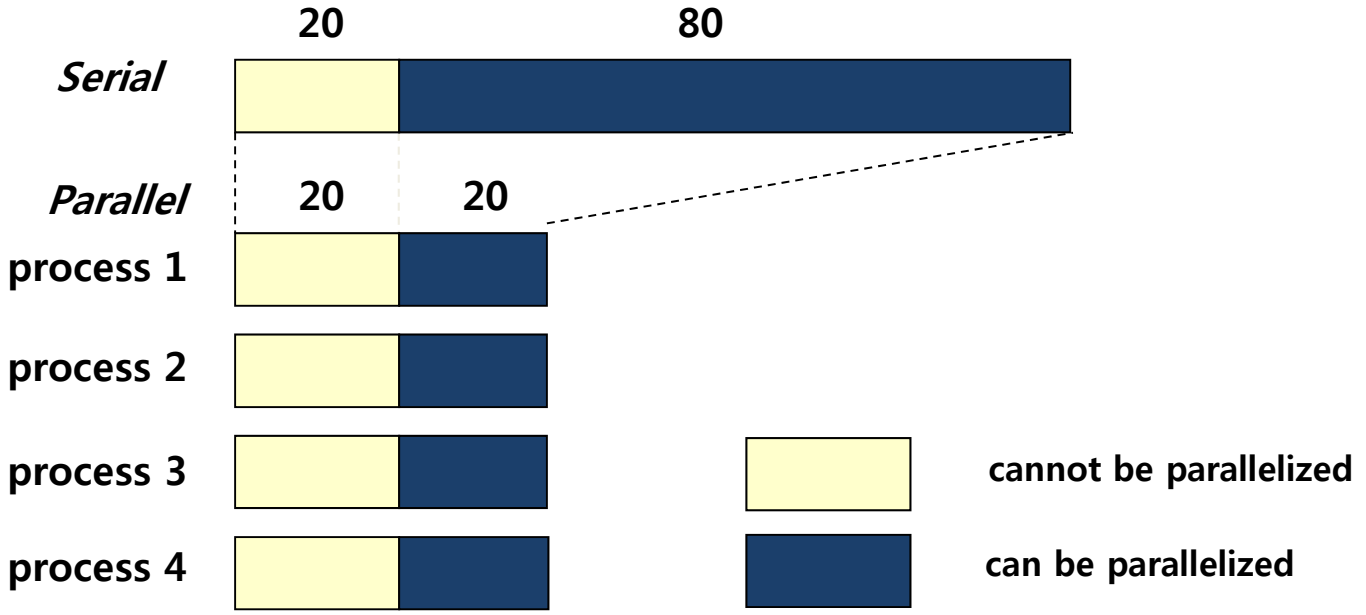
- 최대 성능향상도 ($n \rightarrow \infty$)

$$S(n) = \frac{1}{f}$$

- 프로세서의 개수를 증가하면, 순차부분 크기의 역수에 수렴

성능향상도 (5/7)

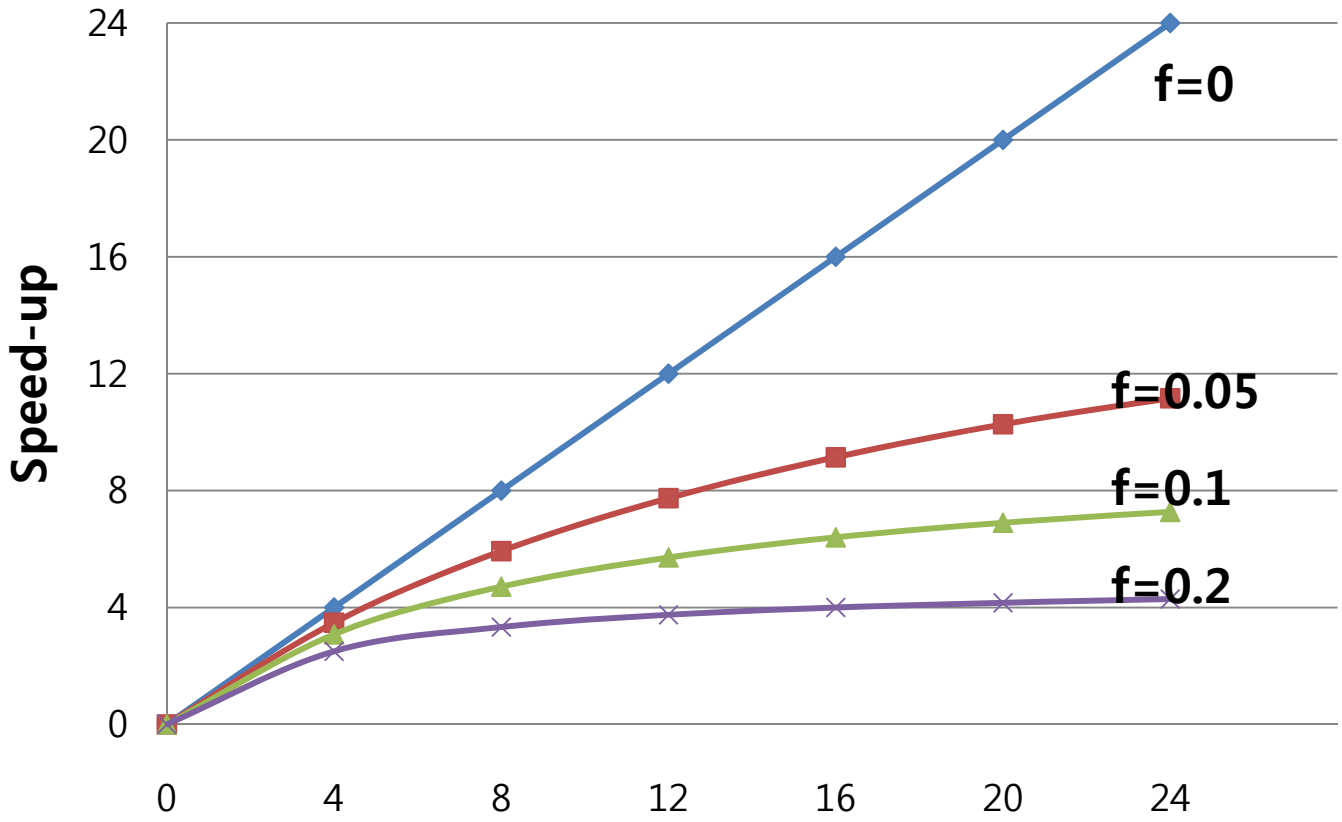
- f = 0.2, n = 4



$$S(4) = \frac{1}{0.2 + (1-0.2)/4} = 2.5$$

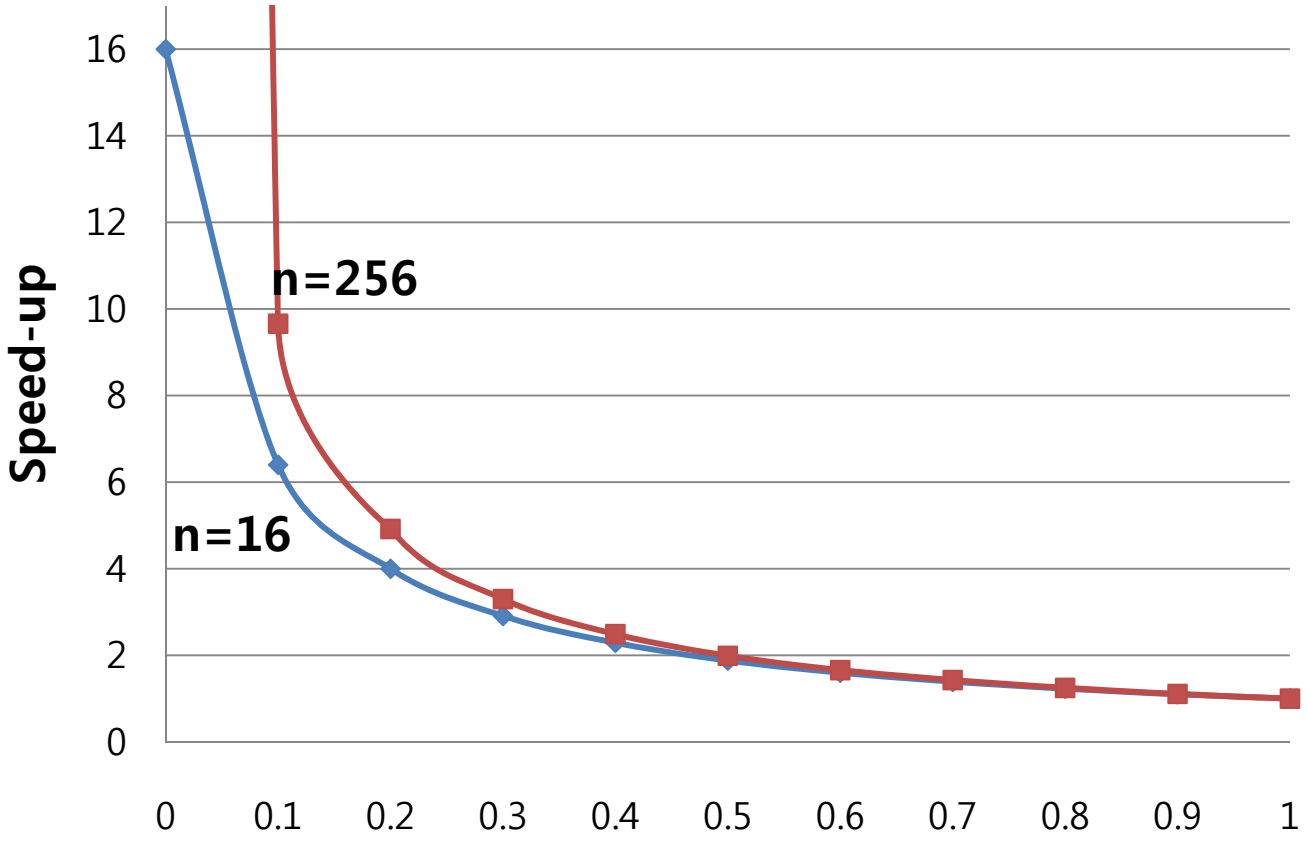
성능향상도 (6/7)

프로세서 개수 대 성능향상도



성능향상도 (7/7)

- 순차부분 대 성능향상도



효율

효율 (Efficiency) : E(n)

$$E(n) = \frac{t_s}{t_p \times n} = \frac{S(n)}{n} \quad [\times 100(\%)]$$

- 프로세서 개수에 따른 병렬 프로그램의 성능효율을 나타냄
  - 10개의 프로세서로 2배의 성능향상 :
    - S(10) = 2 → E(10) = 20 %
  - 100개의 프로세서로 10배의 성능향상 :
    - S(100) = 10 → E(100) = 10 %

Cost

Cost = 실행시간 x 프로세서 개수

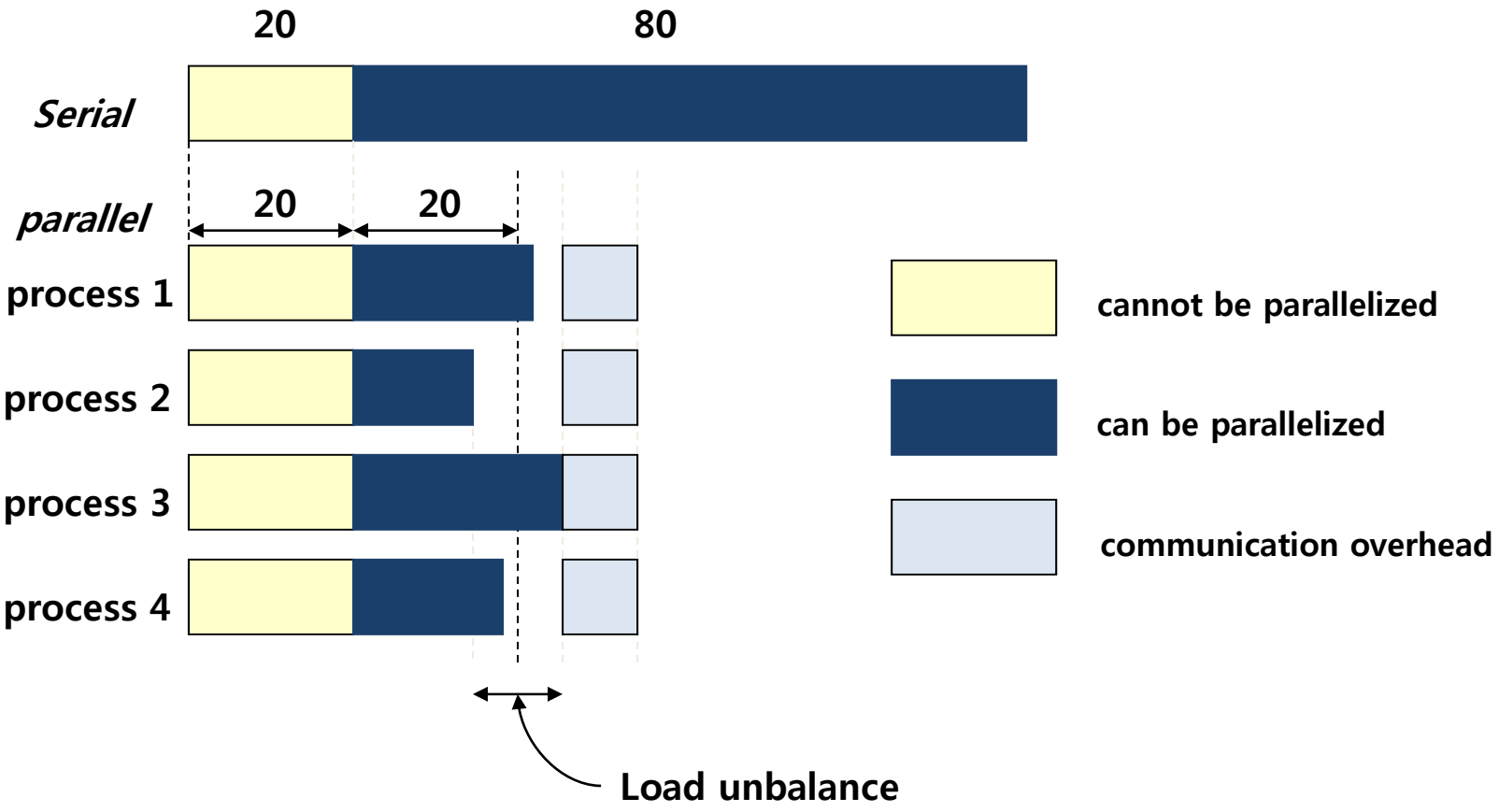
- 순차 프로그램 : Cost = t_s
- 병렬 프로그램 : Cost = t_p x n = =

예) 10개의 프로세서로 2배, 100개의 프로세서로 10배의 성능향상

t_s	t_p	n	S(n)	E(n)	Cost
100	50	10	2	0.2	500
100	10	100	10	0.1	1000

실질적 성능향상에 고려할 사항

- 실제 성능향상도 : 통신부하, 로드 밸런싱 문제



성능증가를 위한 방안들

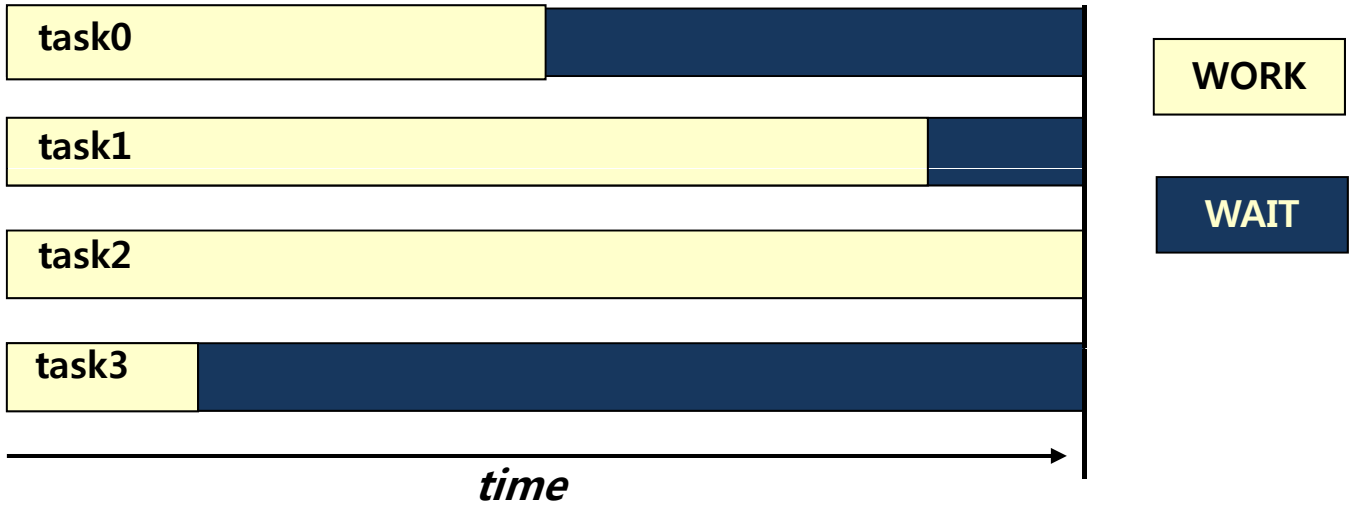
1. 프로그램에서 병렬화 가능한 부분(Coverage) 증가
   - 알고리즘 개선
2. 작업부하의 균등 분배 : 로드 밸런싱
3. 통신에 소비하는 시간(통신부하) 감소

성능에 영향을 주는 요인들

- Coverage : Amdahl's Law
- 로드 밸런싱
- 동기화
- 통신부하
- 세분성
- 입출력

로드 밸런싱

- 모든 프로세스들의 작업시간이 가능한 균등하도록 작업을 분배하여 작업대기시간을 최소화 하는 것
  - 데이터 분배방식(Block, Cyclic, Block-Cyclic) 선택에 주의
  - 이기종 시스템을 연결시킨 경우, 매우 중요함
  - 동적 작업할당을 통해 얻을 수도 있음



동기화

- 병렬 태스크의 상태나 정보 등을 동일하게 설정하기 위한 조정작업
  - 대표적 병렬부하 : 성능에 악영향
  - 장벽, 잠금, 세마포어(semaphore), 동기통신 연산 등 이용

- 병렬부하 (Parallel Overhead)
  - 병렬 태스크의 시작, 종료, 조정으로 인한 부하
    - 시작 : 태스크 식별, 프로세서 지정, 태스크 로드, 데이터 로드 등
    - 종료 : 결과의 취합과 전송, 운영체제 자원의 반납 등
    - 조정 : 동기화, 통신 등

통신부하 (1/4)

- 데이터 통신에 의해 발생하는 부하
  - 네트워크 고유의 지연시간과 대역폭 존재

- 메시지 패싱에서 중요
- 통신부하에 영향을 주는 요인들
  - 동기통신? 비동기 통신?
  - 블록킹? 논블록킹?
  - 점대점 통신? 집합통신?
  - 데이터전송 횟수, 전송하는 데이터의 크기

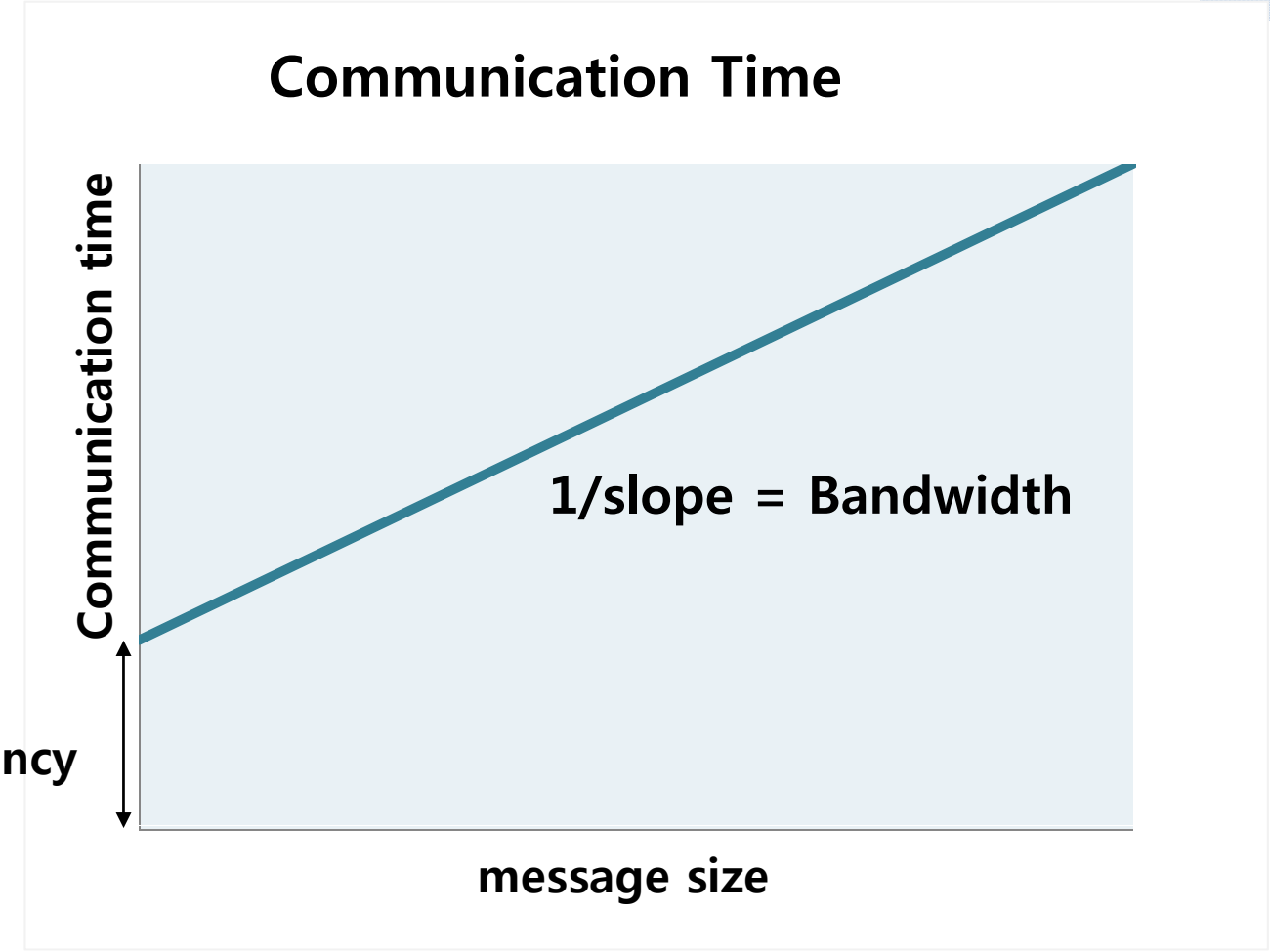
통신부하 (2/4)

$$\text{통신시간} = \text{지연시간} + \frac{\text{메시지 크기}}{\text{대역폭}}$$

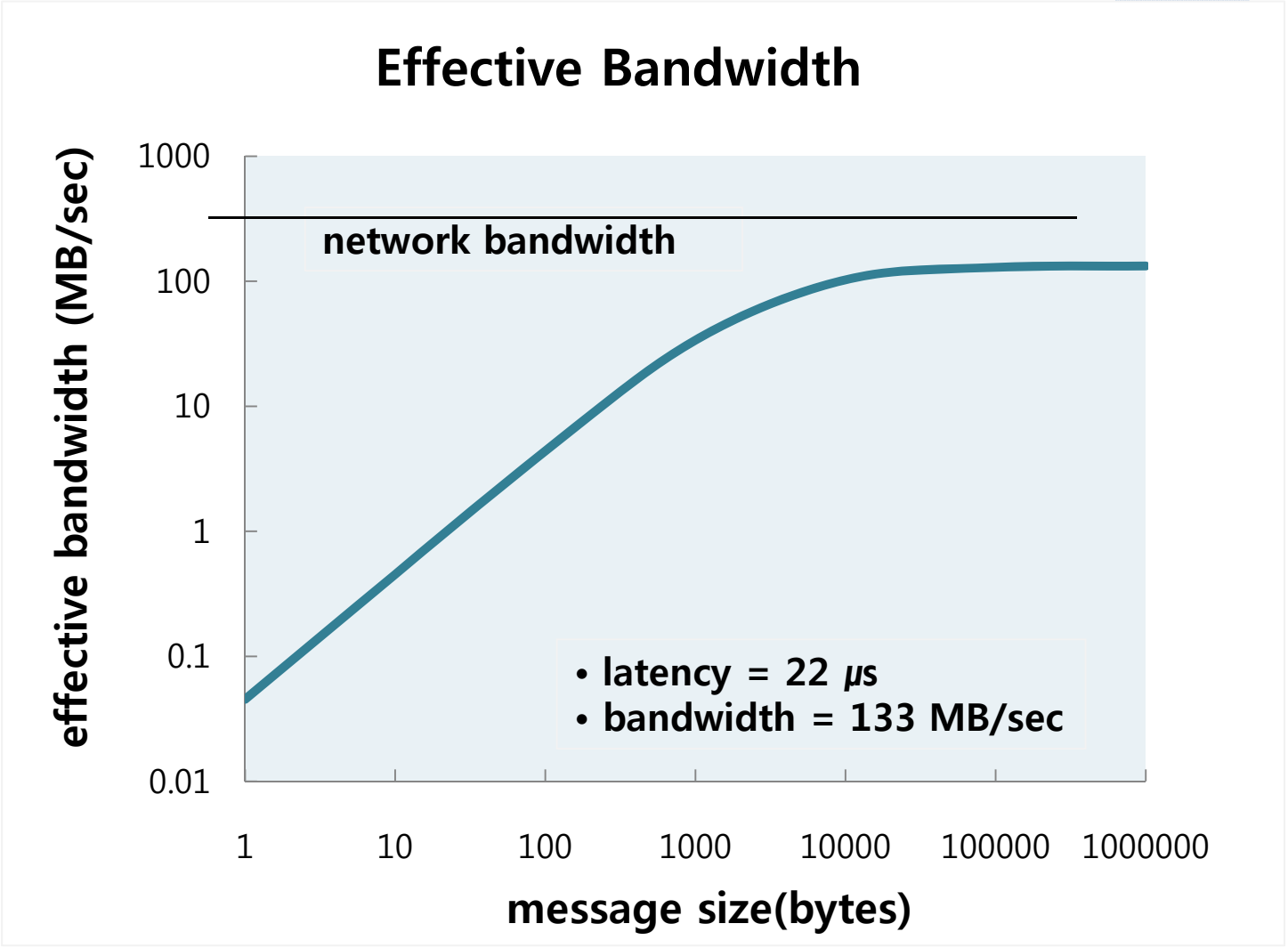
- 지연시간 : 메시지의 첫 비트가 전송되는데 걸리는 시간
  - 송신지연 + 수신지연 + 전달지연
- 대역폭 : 단위시간당 통신 가능한 데이터의 양(MB/sec)

$$\text{유효 대역폭} = \frac{\text{메시지 크기}}{\text{통신시간}} = \frac{\text{대역폭}}{1 + \text{지연시간} \times \text{대역폭}/\text{메시지크기}}$$

통신부하 (3/4)


Communication Time
Communication time
1/slope = Bandwidth
Latency
message size

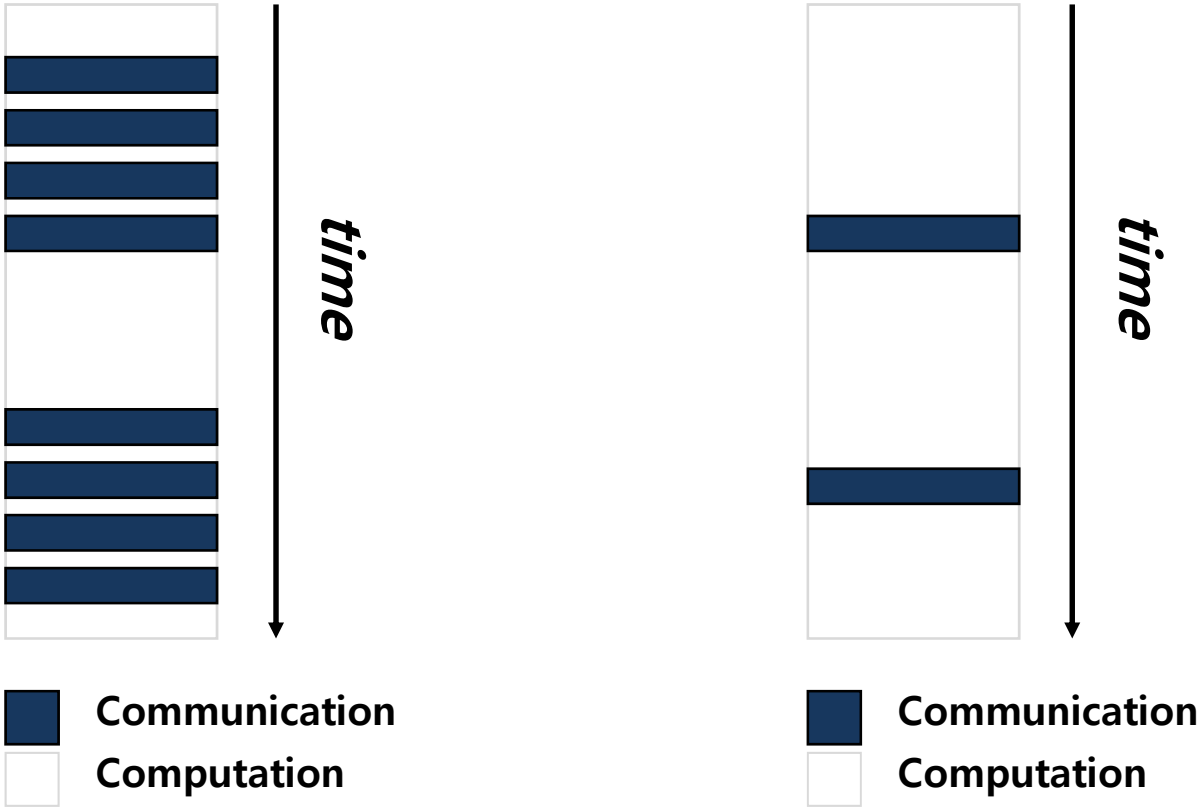
통신부하 (4/4)


Effective Bandwidth
effective bandwidth (MB/sec)
1000
100
10
1
0.1
0.01
network bandwidth
• latency = 22 μs
• bandwidth = 133 MB/sec
1
10
100
1000
10000
100000
1000000
message size(bytes)

세분성 (1/2)

- 병렬 프로그램내의 통신시간에 대한 계산시간의 비
  - Fine-grained 병렬성
    - 통신 또는 동기화 사이의 계산작업이 상대적으로 적음
    - 로드 밸런싱에 유리
  - Coarse-grained 병렬성
    - 통신 또는 동기화 사이의 계산작업이 상대적으로 많음
    - 로드 밸런싱에 불리
- 일반적으로 Coarse-grained 병렬성이 성능면에서 유리
  - 계산시간 < 통신 또는 동기화 시간
  - 알고리즘과 하드웨어 환경에 따라 다를 수 있음

세분성 (2/2)



(a) Fine-grained

(b) Coarse-grained

입출력

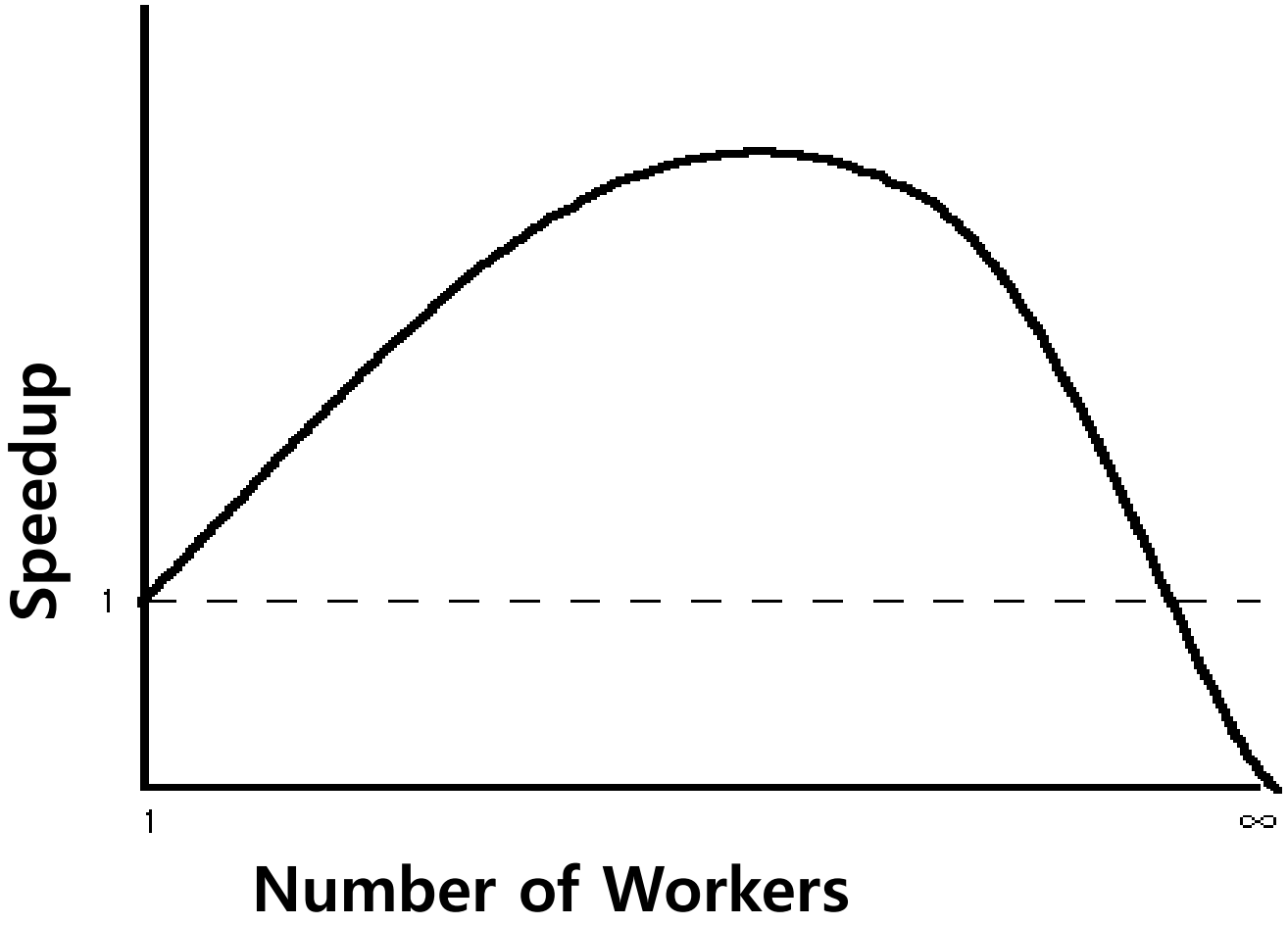
- 일반적으로 병렬성을 방해함
  - 쓰기 : 동일 파일공간을 이용할 경우 겹쳐 쓰기 문제
  - 읽기 : 다중 읽기 요청을 처리하는 파일서버의 성능 문제
  - 네트워크를 경유(NFS, non-local)하는 입출력의 병목현상
- 입출력을 가능하면 줄일 것
  - I/O 수행을 특정 순차영역으로 제한해 사용
  - 지역적인 파일공간에서 I/O 수행
- 병렬 파일시스템의 개발 (GPFS, PVFS, PPFS…)
- 병렬 I/O 프로그래밍 인터페이스 개발 (MPI-2 : MPI I/O)

확장성 (1/2)

- 확장된 환경에 대한 성능이득을 누릴 수 있는 능력
  - 하드웨어적 확장성
  - 알고리즘적 확장성

- 확장성에 영향을 미치는 주요 하드웨어적 요인
  - CPU-메모리 버스 대역폭
  - 네트워크 대역폭
  - 메모리 용량
  - 프로세서 클럭 속도

확장성 (2/2)



의존성과 교착

- 데이터 의존성 : 프로그램의 실행 순서가 실행 결과에 영향을 미치는 것

```
DO k = 1, 100
   F(k + 2) = F(k +1) + F(k)
ENDDO
```

- 교착 : 둘 이상의 프로세스들이 서로 상대방의 이벤트 발생을 기다리는 상태

Process 1	Process 2
X = 4 SOURCE = TASK2 RECEIVE (SOURCE,Y) DEST = TASK2 SEND (DEST,X) Z = X + Y	Y = 8 SOURCE = TASK1 RECEIVE (SOURCE,X) DEST = TASK1 SEND (DEST,Y) Z = X + Y

의존성

F(1)	F(2)	F(3)	F(4)	F(5)	F(6)	F(7)	…	F(n)
1	2	3	4	5	6	7	…	n

```
DO k = 1, 100
   F(k + 2) = F(k +1) + F(k)
ENDDO
```

Serial

F(1)	F(2)	F(3)	F(4)	F(5)	F(6)	F(7)	…	F(n)
1	2	3	5	8	13	21	…	…

Parallel

F(1)	F(2)	F(3)	F(4)	F(5)	F(6)	F(7)	…	F(n)
1	2	3	5(4)	7	11	18	…	…

병렬 프로그램 작성 순서

① 순차코드 작성, 분석(프로파일링), 최적화

- hotspot, 병목지점, 데이터 의존성 등을 확인
- 데이터 병렬성/태스크 병렬성 ?

② 병렬코드 개발

- MPI/OpenMP/… ?
- 태스크 할당과 제어, 통신, 동기화 코드 추가

③ 컴파일, 실행, 디버깅

④ 병렬코드 최적화

- 성능측정과 분석을 통한 성능개선

디버깅과 성능분석

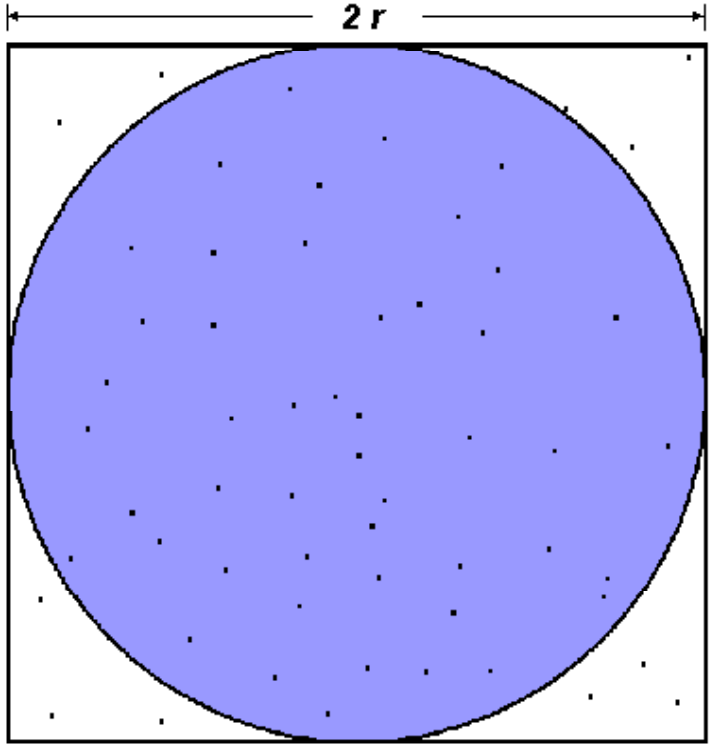
- 디버깅
  - 코드 작성시 모듈화 접근 필요
  - 통신, 동기화, 데이터 의존성, 교착 등에 주의
  - 디버거 : TotalView

- 성능측정과 분석
  - timer 함수 사용
  - 프로파일러 : prof, gprof, pgprof, TAU

병렬 프로그램 작성 예 (1/4)

PI 계산 알고리즘

1. 정사각형에 원을 내접 시킴
2. 정사각형내에서 무작위로 점 추출
3. 추출된 점들 중 원안에 있는 점의 개
4. $PI = 4 \times \frac{\text{원안의 점 개수}}{\text{전체 점의 개수}}$



$A_S = (2r)^2 = 4r^2$

$A_C = \pi r^2$

$\pi = 4 \times \frac{A_C}{A_S}$

병렬 프로그램 작성 예 (2/4)

순차코드 작성 (Pseudo)

```
tpoints = 10000
in_circle = 0
do j = 1,tpoints
   x = rand()
   y = rand()
   if (x, y) inside circle then
   in_circle = in_circle + 1
end do
PI = 4.0*in_circle/tpoints
```

- 거의 모든 계산이 루프에서 실행 → 루프의 반복을 여러 프로세스로 나누어 동시 수행
- 각 프로세스는 담당한 루프 반복만을 실행
- 다른 프로세스의 계산에 대한 정보 불필요 → 의존성 없음
- SPMD 모델사용, 마스터 프로세스가 최종적으로 계산 결과를 취합해야 함

병렬 프로그램 작성 예 (3/4)

병렬코드 작성 (Pseudo)

```
tpoints = 10000
in_circle = 0
p = number of process
num = tpoints/p
find out if I am MASTER or WORKER
do j = 1,num
   x = rand()    y = rand()
   if (x, y) inside circle then
   in_circle = in_circle + 1
end do
if I am MASTER receive from WORKERS their in_circle
   compute PI (use MASTER and WORKER calculations)
else if I am WORKER send to MASTER in_circle
end if
```

기울임 글꼴(붉은색) 부분이 병렬화를 위해 첨가된 부분

병렬 프로그램 작성 예 (4/4)

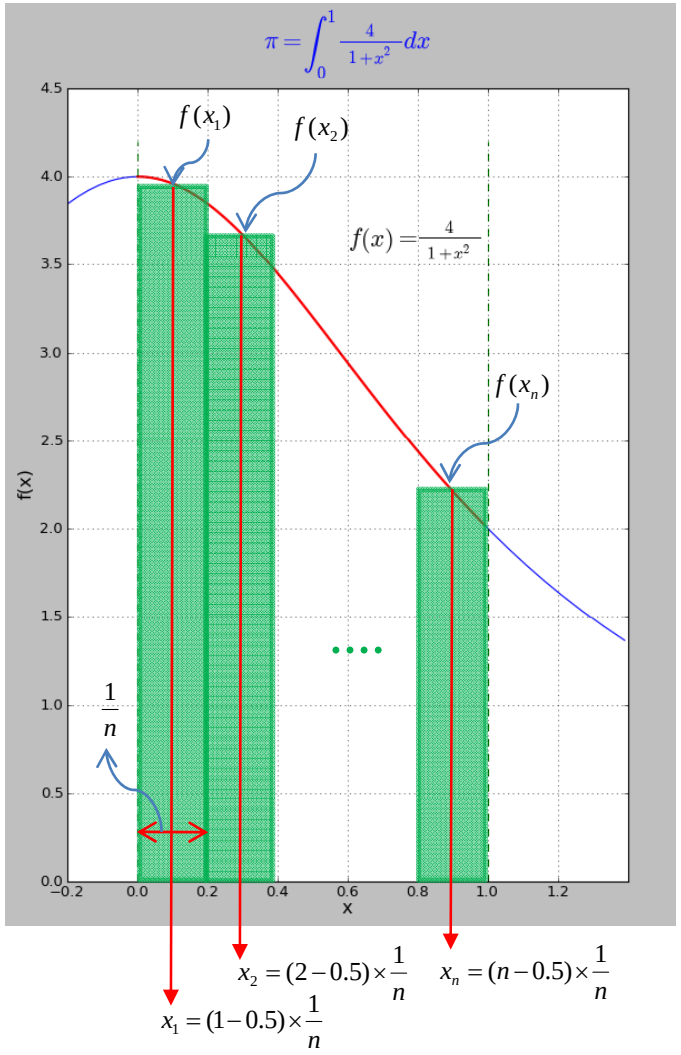
Pi 계산 알고리즘 : 적분을 통한 Pi 계산

$$\int_0^1 \frac{4}{1+x^2}\,dx \qquad x = \tan\theta \qquad dx = \sec^2\theta\ d\theta$$

$$\Rightarrow \int_0^{\frac{\pi}{4}} \frac{4}{1+\tan^2\theta}\sec^2\theta\ d\theta$$

$$\Rightarrow \int_0^{\frac{\pi}{4}} 4\times\cos^2\theta\frac{1}{\cos^2\theta}\,d\theta$$

$$\Rightarrow \int_0^{\frac{\pi}{4}} 4\ d\theta = \pi$$



Example : PI 계산 (1/4)

순차코드 : Fortran

```
      program main
      implicit none
      integer*8,parameter :: num_step = 500000000
      integer*8           :: i
      real(kind=8)        :: sum,step,pi,x
      real(kind=8)        :: stime,etime,rtc
      step = (1.0d0/dble(num_step))
      sum  = 0.0d0
      write(*,400)
      stime=rtc()  !starting time
      do i=1,num_step
        x   = (dble(i)-0.5d0)*step
        sum = sum + 4.d0/(1.d0+x*x)  ! F(x)
      enddo
      etime=rtc()  !ending time
      pi = step * sum
      write(*,100) pi,dabs(dacos(-1.0d0)-pi)
      write(*,300) etime-stime
      write(*,400)
  100 format(' PI = ', F17.15,' (Error =',E11.5,')')
  300 format(' Elapsed Time = ',F8.3,' [sec] ')
  400 format('-----------------------------------------')
      stop
      end program
```

Example : PI 계산 (3/4)

순차코드 : C

```
#include <stdio.h>
#include <math.h>
#include <time.h>
int main(){
  const long num_step = 500000000;
  long i;
  double sum, step, pi, x;
  time_t st, et;
  step = (1.0/(double)num_step);
  sum = 0.0;
  time(&st);
  printf("----------------------------------------₩n");
  for(i=1;i<=num_step;i++){
    x = ((double)i-0.5)*step;
    sum = sum + 4.0/(1.0 + x*x);
  }
  time(&et);
  pi = step * sum;
  printf("PI = %.15f (Error = %e)₩n", pi, fabs(acos(-1.0)-pi));
  printf("Elapsed Time = %.3f [sec]₩n", difftime(et,st));
  printf("----------------------------------------₩n");
}
```