

교육명

CPU&GPU기반 HPC 클러스터 구축과 최적화



목 차

1. 기초
2. 클러스터 구성
3. 클러스터 구축
4. 테스트
5. 기타

01 구성

기초



1. 기초

Single CPU 는 그 속도의 한계를 가지고 있다

1. 기초

1. 물리적 한계: 클럭 속도의 한계

발열(Heat): CPU 클럭을 높일수록 전력 소모와 발열이 기하급수적으로 증가.
일정 수준 이상에서는 냉각 기술로 해결할 수 없을 만큼 열이 발생함.

정전 용량과 누설 전류(Leakage Current): 공정 미세화가 진행될수록 누설 전류 증가로 인한 발열과 안정성 문제 발생.

전자 이동 속도의 한계: 실리콘 내에서 전자의 이동 속도(전자 속도)는 빛의 속도보다 훨씬 느리기 때문에 고클럭화에는 자연적 한계 존재.

1. 기초

2. 단일 스레드 실행 방식의 구조적 제약

- 대부분의 연산은 직렬(serial) 실행을 요구하며, 명령어 간 데이터 의존성 때문에 다음 명령어 실행까지 기다려야 함.
- Instruction-Level Parallelism(ILP)의 한계:
명령어 재배치 및 파이프라이닝 기법으로 어느 정도 속도 개선이 가능하나, 프로그램 특성에 따라 병목이 쉽게 발생

1. 기초

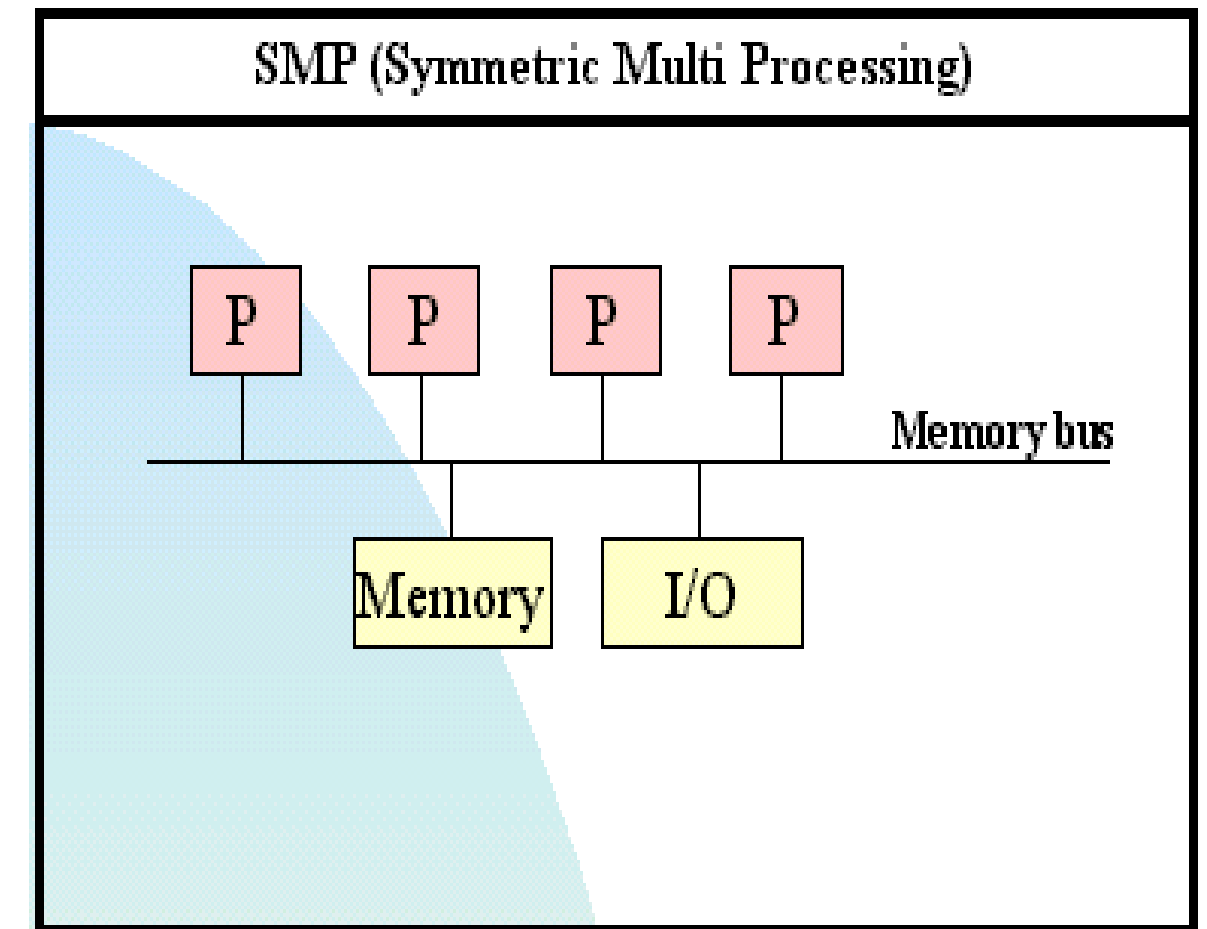
- 일반적으로 컴퓨터의 속도를 결정짓는 이론적인 요인
 - 컴퓨터의 CPU clock 속도
 - 단위 clock당 수행할 수 있는 instruction의 개수
- 보통 CPU clock 속도는 Hz로 표시
 - 1초당 진동하는 진동수를 의미
- 연산 성능 척도 => MFLOPS, GFLOPS

1GFLOPS = 초당 10억번의 실수 연산 능력

1. 기초

1980년대 ~ 1990년대 초반

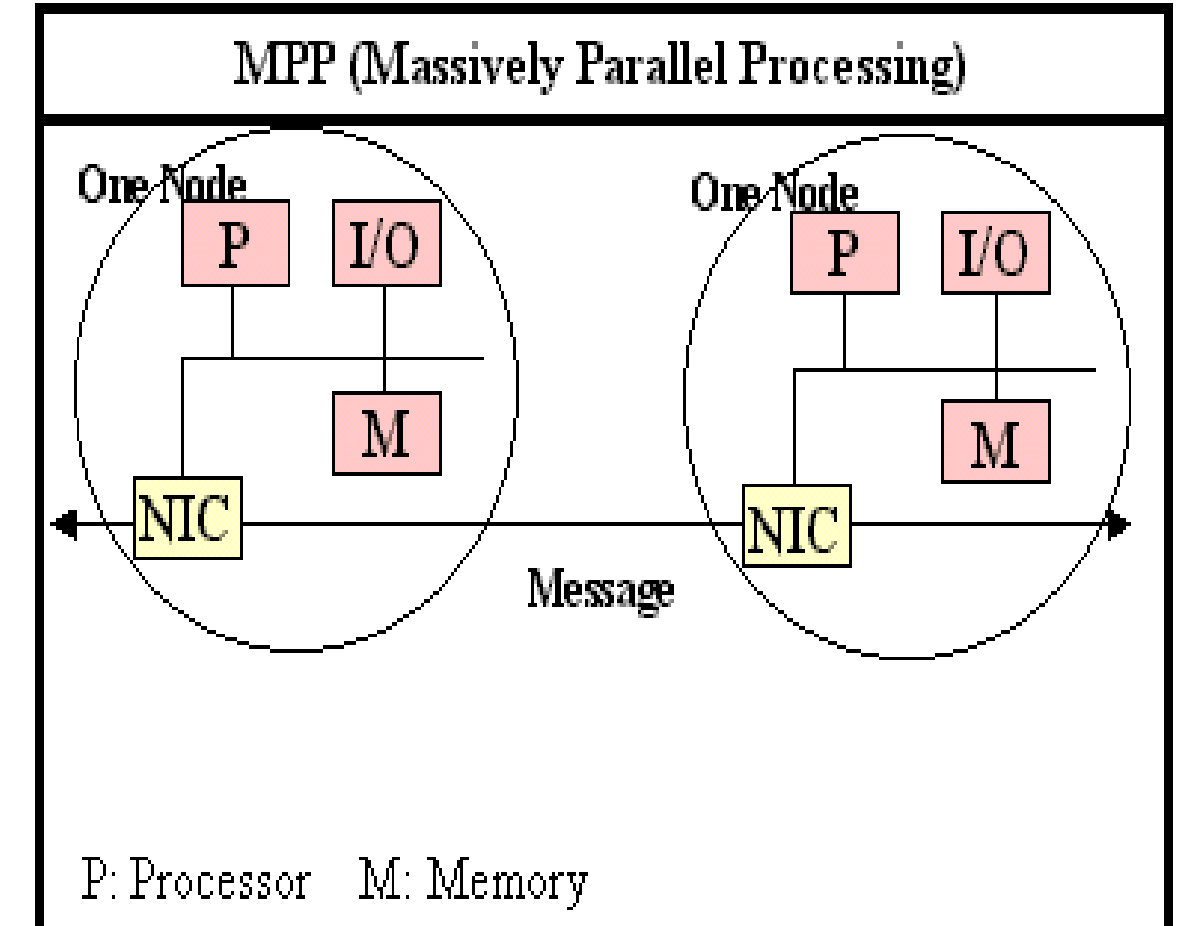
- 당시 슈퍼 컴퓨터들은 고성능의 단일 CPU를 탑재 하거나 벡터 레지스터(Vector register) 기능을 포함 해 CPU성능 향상을 통해 고성능 연산 능력을 제공하는 벡터형 SMP(Symmetric Multi Processor)가 주류
- 입출력 성능과 벡터 레지스터를 이용한 CPU의 계산 능력은 우수하였으나 가격이 매우 비싸다는 문제와 차후 시스템 확장성에 대한 한계를 가짐
- SMP의 경우, 시스템이 하나의 운영체제를 이용해 여러 개의 CPU를 작업량에 맞게 효율적으로 활용하는 기능으로써 여러 개의 CPU를 이용해 별다른 연결장치 없이 내부적으로 memory를 공유해 효율적인 성능을 높일 수 있었다.
- 2 ~ 4개의 프로세서를 갖는 SMP 시스템은 아주 간단하게 구축할 수 있지만, 그 이상의 프로세서를 갖는 시스템의 경우에는 문제가 모든 프로세서가 모든 I/O 및 메모리 자원을 액세스할 수 있어야 하기 때문 자치 공유 자원이 병목 현상에 빠지기 시작하며, CPU를 더 추가 할 경우 반환률이 감소 될 수도 있다



1. 기초

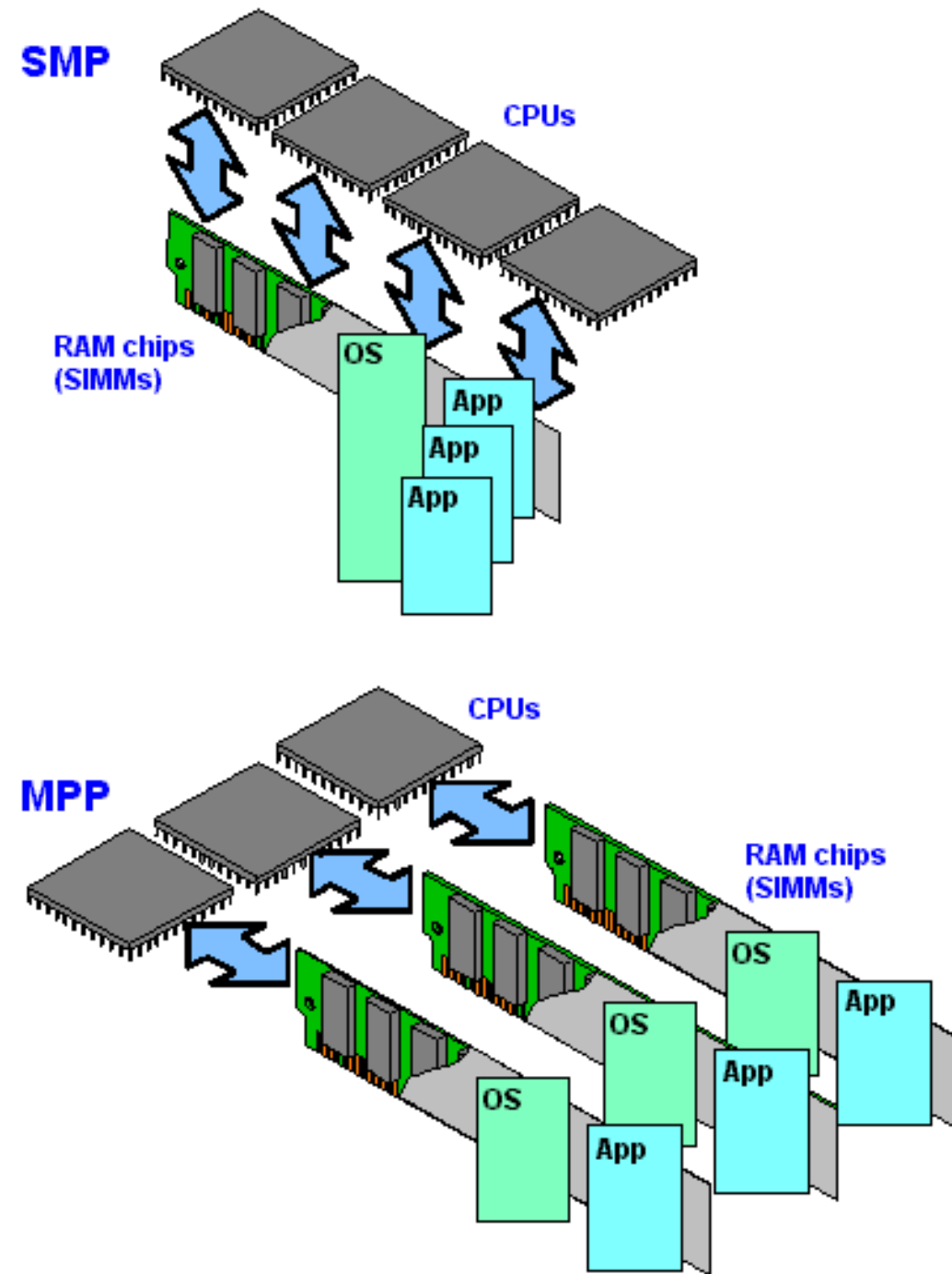
1994년대 ~ 1997년대 초반

- CPU에 대한 단가가 저렴 해졌기 때문에 다수의 일반적인 CPU(4 ~ 1024CPUs)를 이용하는 MPP(Massively Parallel Processor)형 병렬 슈퍼컴퓨터가 등장한 시기
- MPP란 프로그램을 여러 부분으로 나누어 여러 프로세서가 각 부분을 동시에 수행
- 각 프로세서는 운영체제와 메모리를 따로 가지며 프로세스 간에는 message passing로 통신
- 상용 소프트웨어들의 부재와 병렬 시스템을 사용하는 사용자들이 병렬화라는 특수한 개념의 프로그램 기법을 습득해야 한다는 치명적인 단점을 지님
- 사용자들이 병렬 슈퍼컴퓨터 시스템에 탑재되어 있는 CPU 수만큼의 워크스테이션들이라도 생각 할 수 있어 작업량이 많을 때는 여러 대의 워크스테이션들은 보유하고 있는 것처럼 Through-Put(단위 시간당 작업 소화 능력)개념으로 확대해 사용할 수 있다는 점.



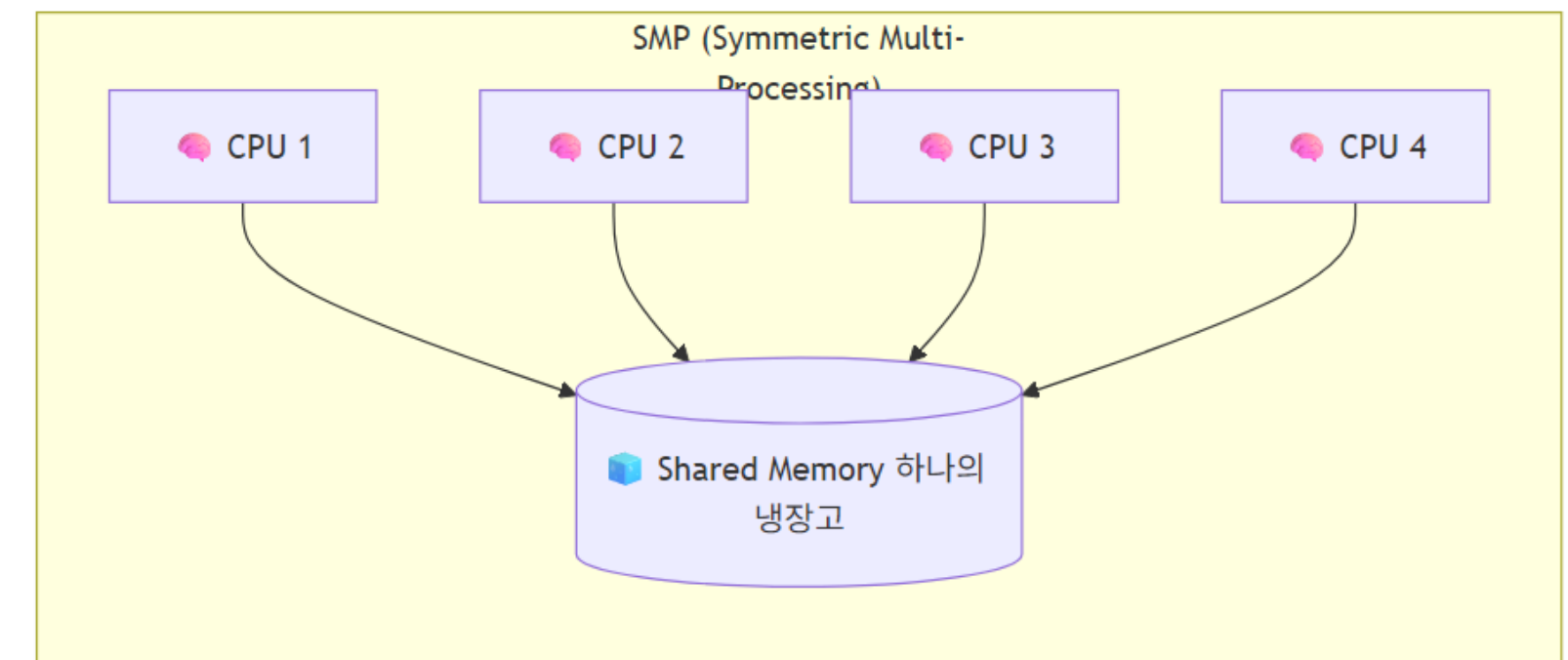
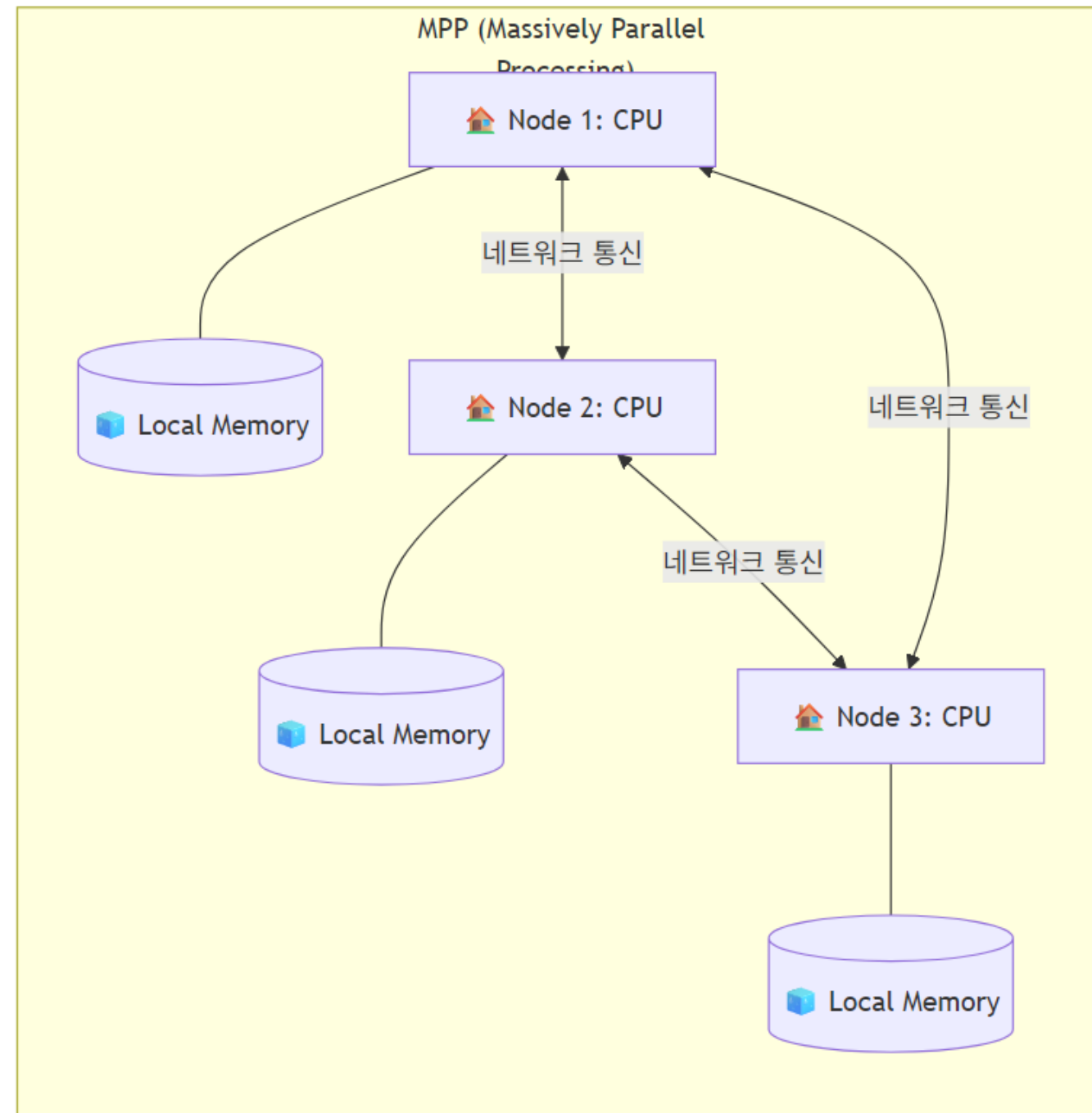
1. 기초

From Computer Desktop Encyclopedia
© 1998 The Computer Language Co. Inc.



비교 항목	SMP	MPP/(Cluster)
구 조	시스템 버스와 같은 인터 커넥션을 통해 프로세스, 메모리, I/O 등의 시스템 자원을 균등하게 공유하는 구조	개별 프로세스, 메모리, I/O 등의 시스템 자원을 가지는 노드들을 상호 연결로 결합한 구성
운영 형태	하나의 OS 커널이 존재 표준 개방형 OS 지원	각 노드별로 OS 커널 존재 표준 개방형 OS 지원 안함. /(표준 개방형 OS 지원).
Inter Connection	멀티 프로세싱, 멀티 스레드, 메모리 공유 프로그래밍 / 통신 방식	데이터 병렬 처리 또는 메시지 패싱 프로그래밍 방식
데이터 전달방식	공유 메모리에 직접 접근	명시적인 메시지 전달
원격데이터 접근	자동	주소 변환, 메시지 전달
프로그래밍	프로그래밍 용이 표준화된 개발 툴이 다양	프로그래밍이 어려움
확장성	업무량 증가 또는 추가시 단위 SMP 내에서 CPU, 메모리 등을 증설 가능	일정 규모까지 시스템 노드 내에서 증설 가능 또는 시스템 노드를 증가
안정성	HW 이중화 및 이중 운영 기능 장애 발생의 자동 감지 및 장애 부분 절단 가능 디스크 미러링	HW 이중화 및 상호 감시 장애의 자동 감지 및 자동 전환 디스크 미러링
Application	다양한 유틸리티 지원	극히 제한됨 / (다양한 유틸리티 지원)
가 격	저, 중가	고가/ (저,중가)

2. 클러스터 구성



2. 클러스터 구성

```
// mpp_mpi_example.c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char *argv[]) {
    int data;
    int local_sum, global_sum;

    MPI_Init(&argc, &argv);

    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // 각 프로세스는 자기만의 데이터를 가짐
    data = rank + 1;
    local_sum = data;

    // 모든 local_sum을 합산하여 global_sum에 저장
    MPI_Reduce(&local_sum, &global_sum, 1, MPI_INT, MPI_SUM, 0,
MPI_COMM_WORLD);

    if (rank == 0) {
        printf("합계 (MPP): %d\n", global_sum);
    }

    MPI_Finalize();
    return 0;
}
```

```
// smp_omp_example.c
#include <stdio.h>
#include <omp.h>

int main() {
    int data[4] = {1, 2, 3, 4};
    int sum = 0;

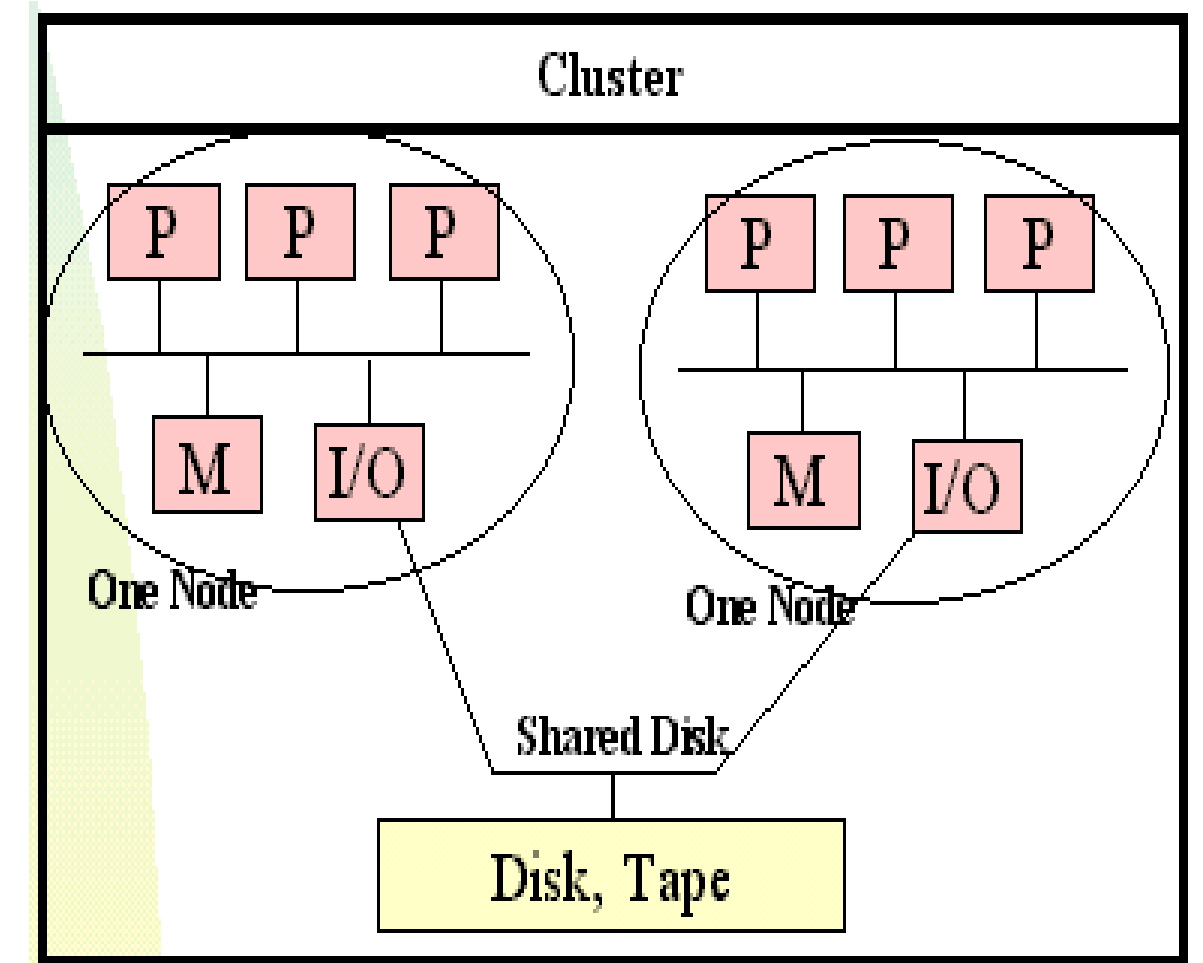
    #pragma omp parallel for reduction(+:sum)
    for (int i = 0; i < 4; i++) {
        sum += data[i]; // 모든 스레드가 data[]
배열 공유
    }

    printf("합계 (SMP): %d\n", sum);
    return 0;
}
```

1. 기초

1998년 ~ 현재

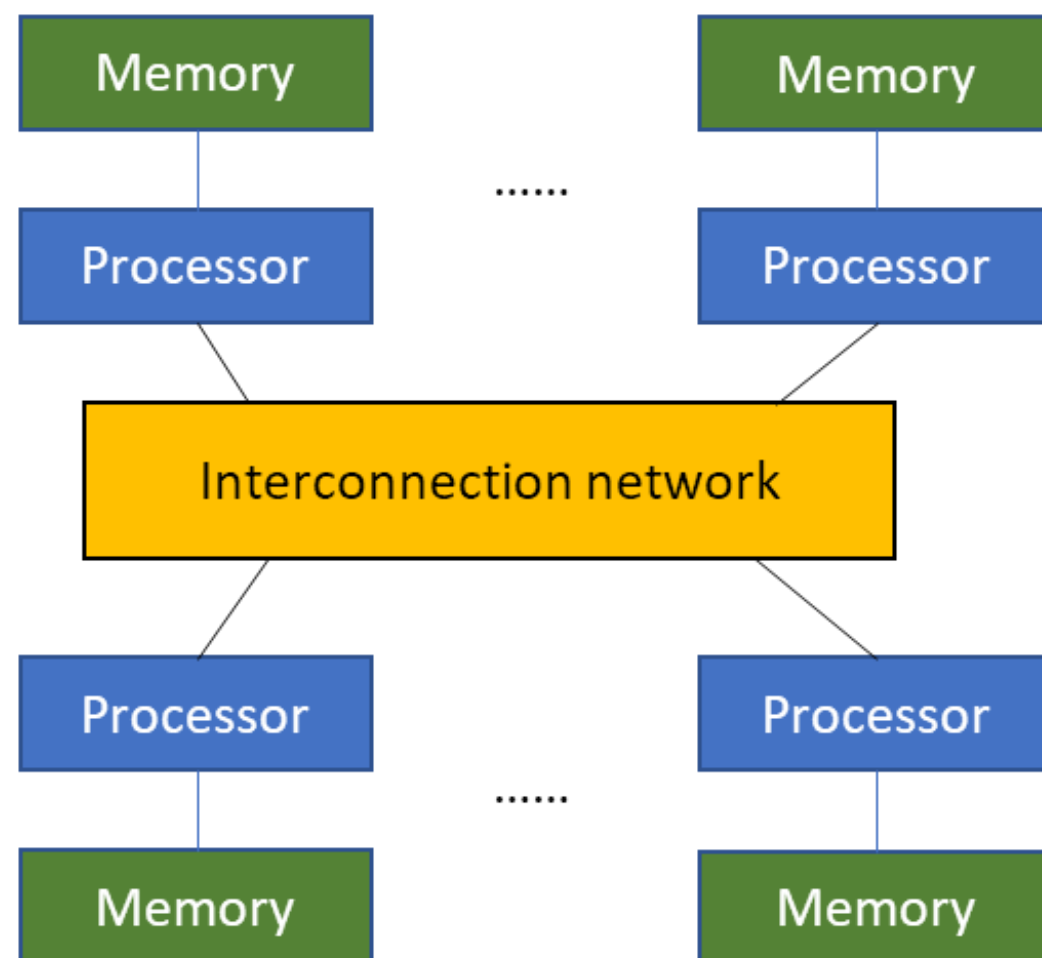
- 1996년부터 불어오기 시작한 리눅스 열풍은 기초과학 분야 및 연구소등에서 값비싼 슈퍼컴퓨터들을 구입해 활용하기보다 저비용으로 고성능을 보일 수 있는 리눅스 기반의 PC 클러스터 시스템이 등장하게 됨
- 클러스터(Cluster)는 MPP 보다 좀더 관리자 면에서 효율적인 운영체제가 필요한 시스템임,
- 사용자가 직접 machine를 관리/통제(병렬프로그래밍)해야 한다는 점이 MPP와 같다. 시스템들은 서로 memory를 제공 할 수 있는 연결 장치가 사용자의 기호(비용)에 맞게 구축 될 수 있으며, 문제가 발생한 machine은 바로 교체가 가능하며 low machine과 High machine을 혼합하여 사용할 수 있다는 특징이 있다.



1. 기초

Distributed Memory Multiprocessor

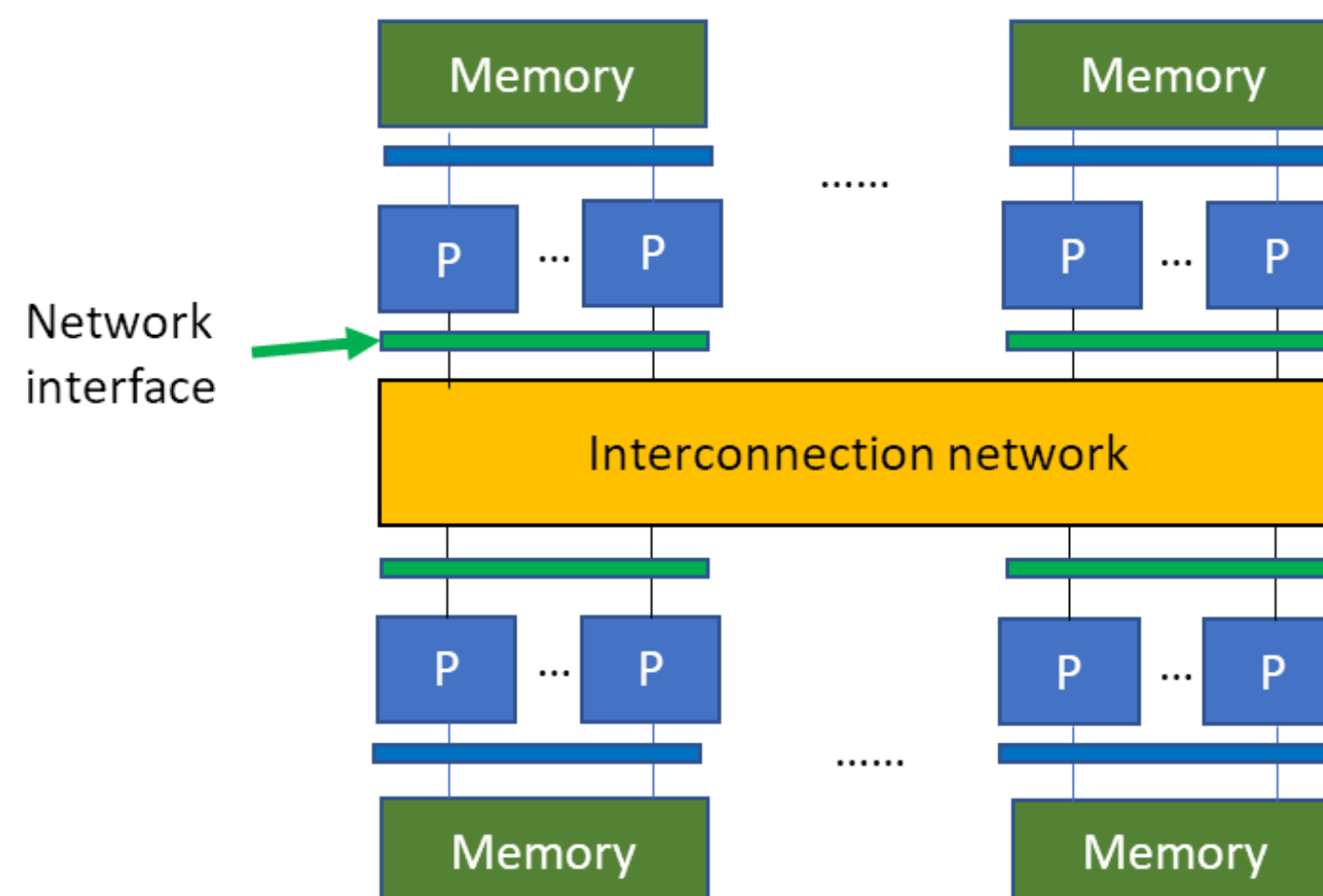
- Message passing between nodes



Massively Parallel Processor (MPP)

Cluster of SMPs

- Shared Memory addressing within SMP node
- Message passing between SMP Nodes



2. 클러스터 구성

“MPP(Massively Parallel Processing)는 고성능 병렬 계산을 위해 설계된 전용 아키텍처로, 고급 네트워크 인터커넥트와 tightly-coupled 노드 구조를 갖는 전용 시스템이다.

반면, Linux Cluster는 범용 서버(commodity hardware)를 고속 네트워크(예: InfiniBand, Omni-Path 등)로 연결하여 MPP와 유사한 병렬 환경을 소프트웨어적으로 구현한 구조이다.”

2025년 6월 기준 Top500에서 Linux Cluster 비중

- Top500 전체 100%는 Linux 기반 시스템
- 그중 x86 아키텍처 기반 시스템이 약 90% 이상
- 90% 이상이 x86 기반 일반 하드웨어 + InfiniBand/Slingshot 네트워크를 사용하는 Linux Cluster 환경
- 전통적인 전용 MPP 시스템(예: Fugaku, 일부 Cray EX/EX 시리즈 등)은 1~3% 수준

2. 클러스터 구성

항목	설명	MPP 환경 예	Cluster 환경 예
컴파일러	시스템 제공 또는 범용	ftn, cc, cray-mpich	mpicc, intelmpi, openmpi
모듈 환경	환경 모듈 로드 필요	module load cray-mpich	module load openmpi
실행 명령어	시스템별 실행기 다름	aprun, srun (Slurm)	mpirun, srun
파일 접근	공유 파일시스템 여부	Lustre, GPFS	NFS, BeeGFS, local
네트워크	고속 전용망 vs 일반망	Aries, Slingshot	InfiniBand, Ethernet
스케줄러	Slurm, PBS 등 다양	#SBATCH, #PBS	Slurm/PBS 동일하거나 유사

1. 기초

Beowulf

- Beowulf는 1994년 미 항공우주국 NASA 산하 연구소인 CESDIS(Center of Excellence in Space Data and information Science)에서 Super 컴퓨터 Cray의 임대기간 종료에 대비하기 위해 새로운 병렬처리 용 Super 컴퓨터의 개발을 시작하였고 16 노드의 클러스터를 이용하여 만든 병렬 컴퓨터의 프로젝트명
- 베어울프는 8세기 영국의 서사시의 주인공
- 도널드 베커, 토머스 스틸링
- Beowulf는 PC를 Ethernet과 같은 LAN으로 연결하여 만든 PC Cluster에서 병렬처리 한 프로그램을 실행시켜서 GFLOPS급의 Super 컴퓨터를 구현
- 1994년 16대의 66MHz Intel 80486 + 10Mbps Ethernet → 100MHz 486DX4 업그레이드 (74MFLOPS)



- 486DX4 100Mhz X 16node
- Each processor had 16M of 60ns DRAM.
- Each node had a 540M or 1G EIDE disk.
- Three 10Mbs bus-master ethernet cards

1. 기초

www.TOP500.ORG



1. 기초

R_{\max} = 실 성능

R_{peak} = 이론 성능

1. 기초

이론 성능 = $\text{Clock} * \text{Cores} * \text{IPC (Instruction Per Clock)}$

Microarchitecture	Instruction Set	SP FLOPS/cycle	DP FLOPS/cycle
Scalable Processor	Intel AVX-512 & FMA	64	32
Haswell/Bordwell	Intel AVX2 & FMA	32	16
Sandy bridge	Intel AVX (256b)	16	8
Nehalem	SSE(128b)	8	4

2. 클러스터 구성

용어 정리: DP FLOPS/cycle = 32

용어	의미
DP	Double Precision = 64-bit 부동소수점
FLOPS	Floating Point Operations (부동소수점 연산 수)
/cycle	CPU 클럭 사이클당 연산 가능 횟수
32 DP FLOPS/cycle	1 클럭 사이클에 Double Precision 연산 32번 수행 가능

항목	설명
64-bit 연산 (DP)	하나의 AVX-512 벡터 레지스터에 8개의 double 값 저장
FMA 지원	Fused Multiply Add 연산 → 1 instruction에 2 FLOPS 가능 ($a = b * c + d$)
레지스터 수	32개 (zmm0 ~ zmm31)
AVX-512 단위 연산기 (per core)	2개의 512-bit 벡터 유닛을 가정

• 1 사이클에 8 (DP 값) × 2 (FMA) × 2 (유닛) = 32 DP FLOPS/cycle

1. 기초

Intel Xeon Skylake CPU는 4가지 Clock 이 있음

- **AVX-512 모드** : AVX-512 / FMA 명령의 높은 요구 사항으로 인해 AVX-512 명령을 실행하는 동안 클럭 속도가 느려짐
- **AVX2 모드** : AVX2 / FMA 명령의 요구 사항이 높기 때문에 AVX 명령을 실행하는 동안 클럭 속도가 느려짐
- **비 AVX 모드** : AVX 명령어를 실행하지 않는 동안 프로세서는 베이스 기반 주파수에서 작동

Intel Turbo Boost Technology

활성 코어 수 (C0 또는 C1 상태)

- 각 모드 별 Turbo Boost Clock 존재
 - 프로세서의 예상 전류 소비량 (Imax)
 - 프로세서의 예상 전력 소비량 (TDP-Thermal Design Power)
 - 프로세서 온도

1. 기초

Performance : Clock : Non-AVX Core Frequency

SKU	Cores	LLC (MB)	TDP (W)	Base non-AVX Core Freq (GHz)	# of active cores / maximum core frequency in turbo mode (GHz)																												
					1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	
8280	28	38.5	205	2.7	4.0	4.0	3.8	3.8	3.7	3.7	3.7	3.7	3.7	3.7	3.7	3.7	3.7	3.7	3.7	3.7	3.7	3.7	3.7	3.5	3.5	3.5	3.5	3.3	3.3	3.3	3.3		
8276	28	38.5	165	2.2	4.0	4.0	3.8	3.8	3.7	3.7	3.7	3.7	3.7	3.7	3.7	3.7	3.7	3.7	3.7	3.7	3.4	3.4	3.4	3.4	3.1	3.1	3.1	3.1	3.0	3.0	3.0	3.0	
8270	26	35.75	205	2.7	4.0	4.0	3.8	3.8	3.7	3.7	3.7	3.7	3.7	3.7	3.7	3.7	3.7	3.7	3.7	3.7	3.7	3.7	3.7	3.5	3.5	3.5	3.5	3.4	3.4				
8268	24	35.75	205	2.9	3.9	3.9	3.7	3.7	3.6	3.6	3.6	3.6	3.6	3.6	3.6	3.6	3.6	3.6	3.6	3.6	3.6	3.6	3.6	3.5	3.5	3.5	3.5						
8260	24	35.75	165	2.4	3.9	3.9	3.7	3.7	3.6	3.6	3.6	3.6	3.6	3.6	3.6	3.6	3.6	3.6	3.6	3.6	3.3	3.3	3.3	3.3	3.1	3.1	3.1	3.1					
8256	4	16.5	105	3.8	3.9	3.9	3.9	3.9																									
8253	16	22	125	2.2	3.0	3.0	2.8	2.8	2.7	2.7	2.7	2.7	2.7	2.7	2.7	2.7	2.5	2.5	2.5	2.5													
6254	18	24.75	200	3.1	4.0	4.0	3.9	3.9	3.9	3.9	3.9	3.9	3.9	3.9	3.9	3.9	3.9	3.9	3.9	3.9	3.9	3.9											
6252	24	35.75	150	2.1	3.7	3.7	3.5	3.5	3.4	3.4	3.4	3.4	3.4	3.4	3.4	3.4	3.2	3.2	3.2	3.2	3.0	3.0	3.0	3.0	2.8	2.8	2.8	2.8					
6248	20	27.5	150	2.5	3.9	3.9	3.7	3.7	3.6	3.6	3.6	3.6	3.6	3.6	3.6	3.6	3.4	3.4	3.4	3.4	3.2	3.2	3.2	3.2									
6246	12	24.75	165	3.3	4.2	4.2	4.1	4.1	4.1	4.1	4.1	4.1	4.1	4.1	4.1	4.1																	
6244	8	24.75	150	3.6	4.4	4.4	4.3	4.3	4.3	4.3	4.3	4.3																					
6242	16	22	150	2.8	3.9	3.9	3.7	3.7	3.6	3.6	3.6	3.6	3.6	3.6	3.6	3.6	3.5	3.5	3.5	3.5													
6240	18	24.75	150	2.6	3.9	3.9	3.7	3.7	3.6	3.6	3.6	3.6	3.6	3.6	3.6	3.6	3.4	3.4	3.4	3.4	3.3	3.3											
6238	22	30.25	140	2.1	3.7	3.7	3.5	3.5	3.4	3.4	3.4	3.4	3.4	3.4	3.4	3.4	3.1	3.1	3.1	3.1	2.9	2.9	2.9	2.9	2.8	2.8							
6234	8	24.75	130	3.3	4.0	4.0	4.0	4.0	4.0	4.0	4.0	4.0																					
6230	20	27.5	125	2.1	3.9	3.9	3.7	3.7	3.6	3.6	3.6	3.6	3.4	3.4	3.4	3.4	3.0	3.0	3.0	3.0	2.8	2.8	2.8	2.8									
6226	12	19.25	125	2.7	3.7	3.7	3.5	3.5	3.5	3.5	3.5	3.5	3.5	3.5	3.5	3.5																	
5222	4	16.5	105	3.8	3.9	3.9	3.9	3.9																									
5220	18	24.75	125	2.2	3.9	3.9	3.7	3.7	3.6	3.6	3.6	3.6	3.1	3.1	3.1	3.1	2.8	2.8	2.8	2.8	2.7	2.7											
5218	16	22	125	2.3	3.9	3.9	3.7	3.7	3.6	3.6	3.6	3.6	3.1	3.1	3.1	3.1	2.8	2.8	2.8	2.8													

1. 기초

Performance : Clock : AVX2.0 Core Frequency

SKU	Cores	LLC (MB)	TDP (W)	Base AVX 2.0 Core Freq (GHz)	# of active cores / maximum core frequency in turbo mode (GHz)																											
					1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
8280	28	38.5	205	2.2	3.8	3.8	3.6	3.6	3.5	3.5	3.5	3.5	3.5	3.5	3.5	3.5	3.5	3.5	3.5	3.5	3.3	3.3	3.3	3.3	3.0	3.0	3.0	3.0	2.9	2.9	2.9	2.9
8276	28	38.5	165	1.7	3.8	3.8	3.6	3.6	3.5	3.5	3.5	3.5	3.5	3.5	3.5	3.5	3.2	3.2	3.2	3.2	2.9	2.9	2.9	2.9	2.7	2.7	2.7	2.7	2.6	2.6	2.6	2.6
8270	26	35.75	205	2.2	3.8	3.8	3.6	3.6	3.5	3.5	3.5	3.5	3.5	3.5	3.5	3.5	3.4	3.4	3.4	3.4	3.2	3.2	3.2	3.2	2.9	2.9	2.9	2.9	2.9	2.9		
8268	24	35.75	205	2.4	3.8	3.8	3.6	3.6	3.5	3.5	3.5	3.5	3.5	3.5	3.5	3.5	3.5	3.5	3.5	3.5	3.2	3.2	3.2	3.2	3.0	3.0	3.0	3.0				
8260	24	35.75	165	1.9	3.7	3.7	3.5	3.5	3.4	3.4	3.4	3.4	3.3	3.3	3.3	3.3	3.0	3.0	3.0	3.0	2.7	2.7	2.7	2.7	2.6	2.6	2.6	2.6				
8256	4	16.5	105	3.3	3.7	3.7	3.7	3.7																								
8253	16	22	125	1.7	2.7	2.7	2.5	2.5	2.4	2.4	2.4	2.4	2.2	2.2	2.2	2.2	2.0	2.0	2.0	2.0												
6254	18	24.75	200	2.7	3.8	3.8	3.6	3.6	3.5	3.5	3.5	3.5	3.5	3.5	3.5	3.5	3.5	3.5	3.5	3.5	3.4	3.4										
6252	24	35.75	150	1.7	3.6	3.6	3.4	3.4	3.3	3.3	3.3	3.3	3.1	3.1	3.1	3.1	2.8	2.8	2.8	2.8	2.5	2.5	2.5	2.5	2.4	2.4	2.4	2.4				
6248	20	27.5	150	1.9	3.8	3.8	3.6	3.6	3.5	3.5	3.5	3.5	3.4	3.4	3.4	3.4	3.0	3.0	3.0	3.0	2.8	2.8	2.8	2.8								
6246	12	24.75	165	2.9	4.0	4.0	3.8	3.8	3.8	3.8	3.8	3.8	3.8	3.8	3.8	3.8																
6244	8	24.75	150	3.0	4.0	4.0	3.9	3.9	3.9	3.9	3.9	3.9																				
6242	16	22	150	2.3	3.8	3.8	3.6	3.6	3.5	3.5	3.5	3.5	3.4	3.4	3.4	3.4	3.1	3.1	3.1	3.1												
6240	18	24.75	150	2.0	3.7	3.7	3.5	3.5	3.4	3.4	3.4	3.4	3.2	3.2	3.2	3.2	2.9	2.9	2.9	2.9	2.8	2.8										
6238	22	30.25	140	1.7	3.6	3.6	3.4	3.4	3.3	3.3	3.3	3.3	3.1	3.1	3.1	3.1	2.8	2.8	2.8	2.8	2.5	2.5	2.5	2.5	2.5	2.5						
6234	8	24.75	130	2.8	3.9	3.9	3.7	3.7	3.7	3.7	3.7	3.7																				
6230	20	27.5	125	1.6	3.8	3.8	3.6	3.6	3.4	3.4	3.4	3.4	2.9	2.9	2.9	2.9	2.6	2.6	2.6	2.6	2.4	2.4	2.4	2.4								
6226	12	19.25	125	2.3	3.6	3.6	3.4	3.4	3.3	3.3	3.3	3.3	3.1	3.1	3.1	3.1																
5222	4	16.5	105	3.3	3.8	3.8	3.8	3.8																								
5220	18	24.75	125	1.8	3.8	3.8	3.6	3.6	3.4	3.4	3.4	3.4	2.9	2.9	2.9	2.9	2.6	2.6	2.6	2.6	2.5	2.5										
5218	16	22	125	1.8	2.9	2.9	2.7	2.7	2.6	2.6	2.6	2.6	2.5	2.5	2.5	2.5	2.3	2.3	2.3	2.3												

1. 기초

Performance : Clock : AVX512 Core Frequency

SKU	Core s	LLC (MB)	TDP (W)	Base AVX- 512 Core Freq. (GHz)	# of active cores / maximum core frequency in turbo mode (GHz)																												
					1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	
8280	28	38.5	205	1.8	3.7	3.7	3.5	3.5	3.4	3.4	3.4	3.4	3.3	3.3	3.3	3.3	2.9	2.9	2.9	2.9	2.7	2.7	2.7	2.7	2.5	2.5	2.5	2.5	2.4	2.4	2.4	2.4	
8276	28	38.5	165	1.3	3.7	3.7	3.5	3.5	3.3	3.3	3.3	3.3	2.9	2.9	2.9	2.9	2.6	2.6	2.6	2.6	2.3	2.3	2.3	2.3	2.2	2.2	2.2	2.2	2.1	2.1	2.1	2.1	
8270	26	35.75	205	1.8	3.7	3.7	3.5	3.5	3.4	3.4	3.4	3.4	3.2	3.2	3.2	3.2	2.8	2.8	2.8	2.8	2.6	2.6	2.6	2.6	2.4	2.4	2.4	2.4	2.4	2.4			
8268	24	35.75	205	1.9	3.7	3.7	3.5	3.5	3.4	3.4	3.4	3.4	3.3	3.3	3.3	3.3	3.0	3.0	3.0	3.0	2.7	2.7	2.7	2.7	2.6	2.6	2.6	2.6					
8260	24	35.75	165	1.5	3.7	3.7	3.5	3.5	3.4	3.4	3.4	3.4	3.0	3.0	3.0	3.0	2.6	2.6	2.6	2.6	2.4	2.4	2.4	2.4	2.3	2.3	2.3	2.3					
8256	4	16.5	105	2.7	3.7	3.7	3.5	3.5																									
8253	16	22	125	1.2	2.6	2.6	2.4	2.4	2.0	2.0	2.0	2.0	1.7	1.7	1.7	1.7	1.6	1.6	1.6	1.6													
6254	18	24.75	200	2.2	3.6	3.6	3.4	3.4	3.3	3.3	3.3	3.3	3.3	3.3	3.3	3.3	3.0	3.0	3.0	3.0	2.9	2.9											
6252	24	35.75	150	1.3	3.5	3.5	3.3	3.3	3.0	3.0	3.0	3.0	2.6	2.6	2.6	2.6	2.3	2.3	2.3	2.3	2.1	2.1	2.1	2.1	2.0	2.0	2.0	2.0					
6248	20	27.5	150	1.6	3.8	3.8	3.6	3.6	3.5	3.5	3.5	3.5	3.0	3.0	3.0	3.0	2.7	2.7	2.7	2.7	2.5	2.5	2.5	2.5									
6246	12	24.75	165	2.4	3.9	3.9	3.7	3.7	3.6	3.6	3.6	3.6	3.4	3.4	3.4	3.4																	
6244	8	24.75	150	2.6	3.8	3.8	3.6	3.6	3.5	3.5	3.5	3.5																					
6242	16	22	150	1.9	3.7	3.7	3.5	3.5	3.2	3.2	3.2	3.2	2.7	2.7	2.7	2.7	2.5	2.5	2.5	2.5													
6240	18	24.75	150	1.6	3.7	3.7	3.5	3.5	3.4	3.4	3.4	3.4	2.9	2.9	2.9	2.9	2.6	2.6	2.6	2.6	2.5	2.5											
6238	22	30.25	140	1.3	3.6	3.6	3.4	3.4	3.0	3.0	3.0	3.0	2.6	2.6	2.6	2.6	2.3	2.3	2.3	2.3	2.1	2.1	2.1	2.1	2.1	2.1							
6234	8	24.75	130	2.3	3.7	3.7	3.5	3.5	3.1	3.1	3.1	3.1																					
6230	20	27.5	125	1.1	3.7	3.7	3.5	3.5	2.8	2.8	2.8	2.8	2.4	2.4	2.4	2.4	2.1	2.1	2.1	2.1	2.0	2.0	2.0	2.0									
6226	12	19.25	125	1.9	3.5	3.5	3.3	3.3	3.0	3.0	3.0	3.0	2.6	2.6	2.6	2.6																	
5222	4	16.5	105	2.7	3.7	3.7	3.5	3.5																									
5220	18	24.75	125	1.4	3.7	3.7	3.5	3.5	2.8	2.8	2.8	2.8	2.4	2.4	2.4	2.4	2.1	2.1	2.1	2.1	2.1	2.1											
5218	16	22	125	1.5	2.9	2.9	2.7	2.7	2.6	2.6	2.6	2.6	2.3	2.3	2.3	2.3	2.1	2.1	2.1	2.1													

1. 기초

```
[root@gt002 ~]# lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                 20
On-line CPU(s) list:   0-19
Thread(s) per core:    1
Core(s) per socket:    10
Socket(s):              2
NUMA node(s):          2
Vendor ID:              GenuineIntel
CPU family:             6
Model:                  79
Model name:             Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz
Stepping:               1
CPU MHz:                2195.083
BogoMIPS:               4390.16
Virtualization:         VT-x
L1d cache:              32K
L1i cache:              32K
L2 cache:               256K
L3 cache:               25600K
NUMA node0 CPU(s):      0-9
NUMA node1 CPU(s):      10-19
Flags:                  fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx
                        fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology
                        nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm
                        pcid dca sse4_1 sse4_2 x2apic movbe popcnt aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch epb cat_l3 cdp_l3 in
                        vpcid_single intel_ppin intel_pt tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2
                        erms invpcid rtm cqm rdt_a rdseed adx smap xsaveopt cqm_llc cqm_occup_llc cqm_mbm_total cqm_mbm_local dtherm arat p
                        ln pts
[root@gt002 ~]#
```

1. 기초

전력 관리 옵션 Enable : Default

Mon Sep 12 10:12:19 202

cpu MHz	: 1588.098
cpu MHz	: 2083.044
cpu MHz	: 1965.551
cpu MHz	: 1703.308
cpu MHz	: 2200.268
cpu MHz	: 1960.852
cpu MHz	: 2146.423
cpu MHz	: 2200.402
cpu MHz	: 1544.189
cpu MHz	: 1200.036
cpu MHz	: 1384.802
cpu MHz	: 1368.017
cpu MHz	: 1464.965
cpu MHz	: 1308.532
cpu MHz	: 1356.335
cpu MHz	: 1449.121
cpu MHz	: 2080.761
cpu MHz	: 1199.768
cpu MHz	: 1495.715
cpu MHz	: 2087.878

전력 관리 옵션 Disable

Mon Sep 12 10:05:59 202

cpu MHz	: 2195.083
cpu MHz	: 2195.083
cpu MHz	: 2195.083
cpu MHz	: 2195.083
cpu MHz	: 2195.083
cpu MHz	: 2195.083
cpu MHz	: 2195.083
cpu MHz	: 2195.083
cpu MHz	: 2195.083
cpu MHz	: 2195.083
cpu MHz	: 2195.083
cpu MHz	: 2195.083
cpu MHz	: 2195.083
cpu MHz	: 2195.083
cpu MHz	: 2195.083
cpu MHz	: 2195.083
cpu MHz	: 2195.083
cpu MHz	: 2195.083
cpu MHz	: 2195.083
cpu MHz	: 2195.083
cpu MHz	: 2195.083

1. 기초

SKU	Core	Base Non-AVX core Freq (GHz)	Base AVX 2.0 core Freq (GHz)	Base AVX 512 core Freq (GHz)
8280	28	2.7	2.2	1.8

이론 성능 = Clock * Core * IPC (Instruction Per Clock)

Microarchitecture	Instruction Set	SP FLOPS/cycle	DP FLOPS/cycle
Scalable Processor	Intel AVX-512 & FMA	64	32
Haswell/Bordwell	Intel AVX2 & FMA	32	16
Sandy bridge	Intel AVX (256b)	16	8
Nehalem	SSE(128b)	8	4

Maximum AVX 512 core Freq 2.4 GHz * 28 * 32 = 2,150.4 GFLOPS

Base Non-AVX core Freq 2.7 GHz * 28 * 4 = 302.4 GFLOPS

Base AVX 2.0 core Freq 2.2 GHz * 28 * 16 = 985 GFLOPS

Base AVX 512 core Freq 1.8 GHz * 28 * 32 = 1,612.8 GFLOPS

1. 기초

HPC 고객 워크로드의 경우 다양한 세트를 실행합니다.



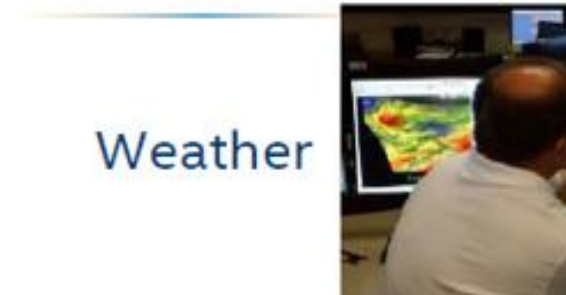
More compute
=
More Value

Per Core Perf
AVX-512 ✓
Mem B/W
Intel Software ✓



More compute
=
More Value

Per Core Perf
AVX-512 ✓
Mem B/W
Intel Software ✓



More Mem B/W
=
More Value

Per Core Perf
AVX-512
Mem B/W ✓
Intel Software ✓



Per Core Perf
=
More Value

Per Core Perf ✓
AVX-512 ✓
Mem B/W
Intel Software ✓



Compute + Mem B/W
=
More Value

Per Core Perf
AVX-512 ✓
Mem B/W ✓
Intel Software ✓

1. 기초

INTEL AVX-512 CODES

Life & Materials Sciences



- ✓ **VASP***
Studying the movements of atoms & molecules
- ✓ **GROMACS***
Studying the movements of atoms & molecules
- ✓ **LAMMPS***
Studying the movements of atoms & molecules
- ✓ **NAMD***
Studying the movements of atoms & molecules
- ✓ **Amber***
Classical molecular dynamics & structural analysis
- ✓ **CP2K***
Quantum Chemistry

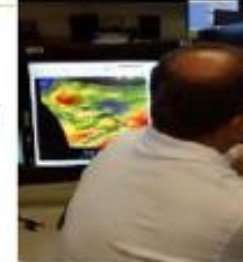
Finance



Options Pricing

- ✓ **Binomial***
Fast calculation for options trading
- ✓ **Black-Scholes***
Fast calculation for options trading
- ✓ **Monte-Carlo***
Fast calculation for options trading

Weather



- ✓ **COSMO***
Non-hydrostatic atmospheric prediction system

Manufacturing (CAE)



Structural Analysis

- ✓ **Abaqus***
CAE package ideal for static & low-speed dynamic events

Crash Simulation

- ✓ **RADIOSS***
Front car crash model

Oil & Gas



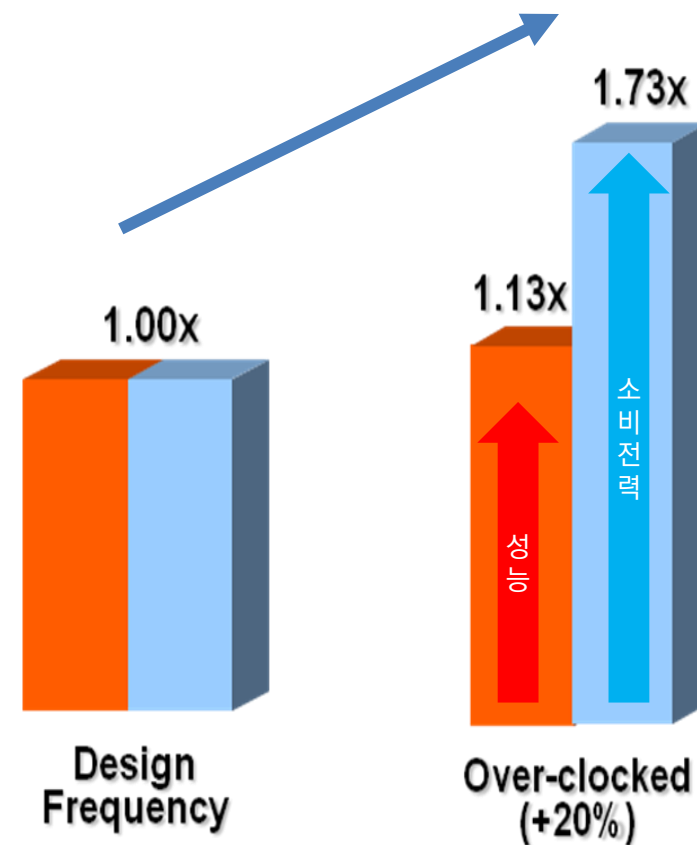
Seismic Analysis

- ✓ **Proprietary Codes**
Applications used to model earth and find oil

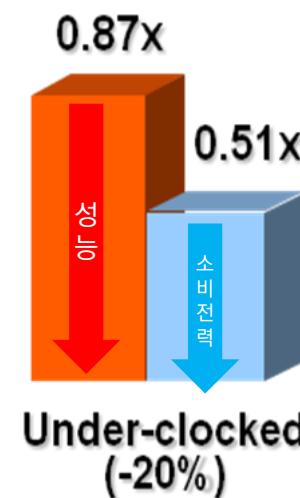
1. 기초

Performance : CPU : Clock

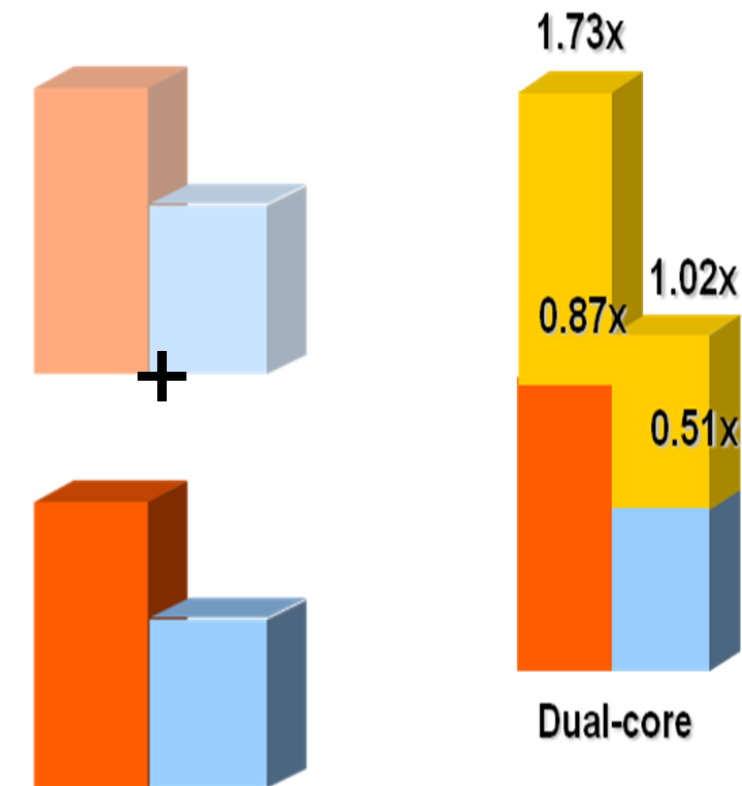
- CPU의 Clock을 20%을 한다면 실제적으로 얻을 수 있는 성능 향상은 13% 정도 이다. 반면, 소비전력 73%가 증가를 하게 된다..
- 이에 반하여 Clock 을 20%를 낮춘다면 성능은 13%가 낮아 지며, 소비전력은 49%나 낮아지게 된다.
- 이때, Core를 증가를 하게 되면, 73%의 성능 증가와 에너지 효율을 수 있게 되는 것이다.
- 이 때문에 Clock 을 높이는 것보다 Core 를 증가를 시키는 것이다.



CPU의 Clock을 20% UP



CPU의 Clock을 20% Down



Dual Core 로 구성

1. 기초

캐시 일관성(cache coherence)

- 각 CPU 코어는 자체 L1/L2 캐시를 가지며 일부는 L3 캐시를 공유
- 동일한 메모리 주소에 대해 여러 캐시에 복사본이 존재할 경우, 한 캐시에서의 변경이 다른 캐시에 반영되지 않으면 데이터 불일치가 발생
- 이를 해결하기 위한 프로토콜이 캐시 일관성 프로토콜(Coherence Protocol)

프로토콜	특징
MESI	Modified, Exclusive, Shared, Invalid 상태 기반. 대부분의 CPU에서 사용됨.
MOESI	MESI + Owned 상태 추가 (AMD EPYC 등)
MESIF	MESI + Forward (Intel의 최신 프로세서에서 사용)

1. 기초

상태	Full Name	설명
M	Modified	이 캐시에만 있는 데이터이며, 메모리에는 없음. 다른 캐시는 invalid 상태. (Dirty 상태)
E	Exclusive	메모리와 값이 동일하며, 이 캐시만 복사본을 가지고 있음. 다른 캐시는 없음.
S	Shared	여러 캐시에 복사본이 있음. 메모리와의 값이 동일. 읽기만 가능.
I	Invalid	이 캐시의 데이터는 더 이상 유효하지 않음. 접근하면 다시 읽어와야 함.

상태	Full Name	설명
M	Modified	MESI와 동일. 이 캐시에만 있고 메모리와 불일치.
O	Owned	데이터는 메모리와 불일치하며, 이 캐시가 대표로 유지/공유함. 다른 캐시도 복사본 있음.
E	Exclusive	동일. 유일 복사본. 메모리와 일치.
S	Shared	동일. 여러 캐시와 메모리와 일치. 읽기 전용.
I	Invalid	동일. 무효 데이터.

상태	Full Name	설명
M	Modified	동일. 메모리와 불일치. 유일한 복사본.
E	Exclusive	동일. 메모리와 일치, 유일한 복사본.
S	Shared	동일. 메모리와 일치, 여러 캐시가 공유 가능.
I	Invalid	동일. 무효 데이터.
F	Forward	Shared 상태 중 특정 캐시가 대표로 다른 캐시에 데이터 전달을 담당

1. 기초

// false_sharing.c

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h> // OpenMP 병렬 처리용 헤더

#define N 100000000 // 각 스레드가 수행할 반복 횟수
#define THREADS 4 // 사용할 스레드 수

// 각 스레드가 접근할 공유 구조체
typedef struct {
    int value; // 문제점: 구조체가 서로 가까이 위치하여 캐시 라인을 공유하게 됨
} SharedData;

SharedData data[THREADS]; // THREADS 개수만큼 배열 생성

int main() {
    double start = omp_get_wtime(); // 시작 시간 측정

    // 병렬 영역 시작
    #pragma omp parallel num_threads(THREADS)
    {
        int tid = omp_get_thread_num(); // 현재 스레드 ID

        // 각 스레드가 자신에게 할당된 데이터에 대해 반복 증가
        for (int i = 0; i < N; i++) {
            data[tid].value++; // false sharing 발생 가능
        }
    }

    double end = omp_get_wtime(); // 종료 시간 측정
    printf("False Sharing Time: %.5f seconds\n", end - start);
    return 0;
}
```

// padding.c

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#define N 100000000
#define THREADS 4
#define CACHE_LINE_SIZE 64 // 일반적인 캐시 라인 크기

// 구조체 패딩 적용: 캐시 라인을 독립적으로 점유
typedef struct {
    int value;
    char padding[CACHE_LINE_SIZE - sizeof(int)]; // 나머지를 padding으로 채움
} PaddedData;

PaddedData data[THREADS];

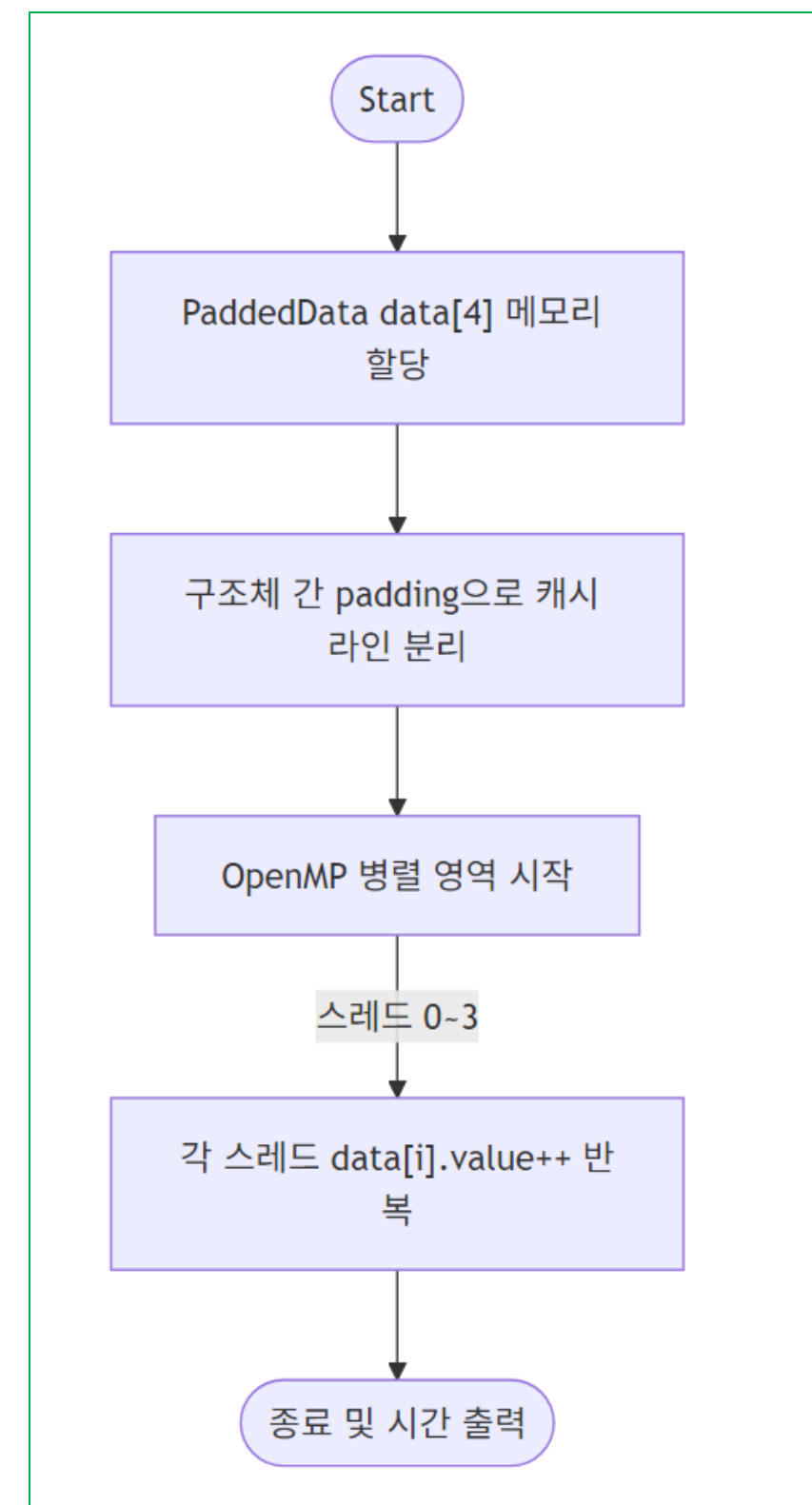
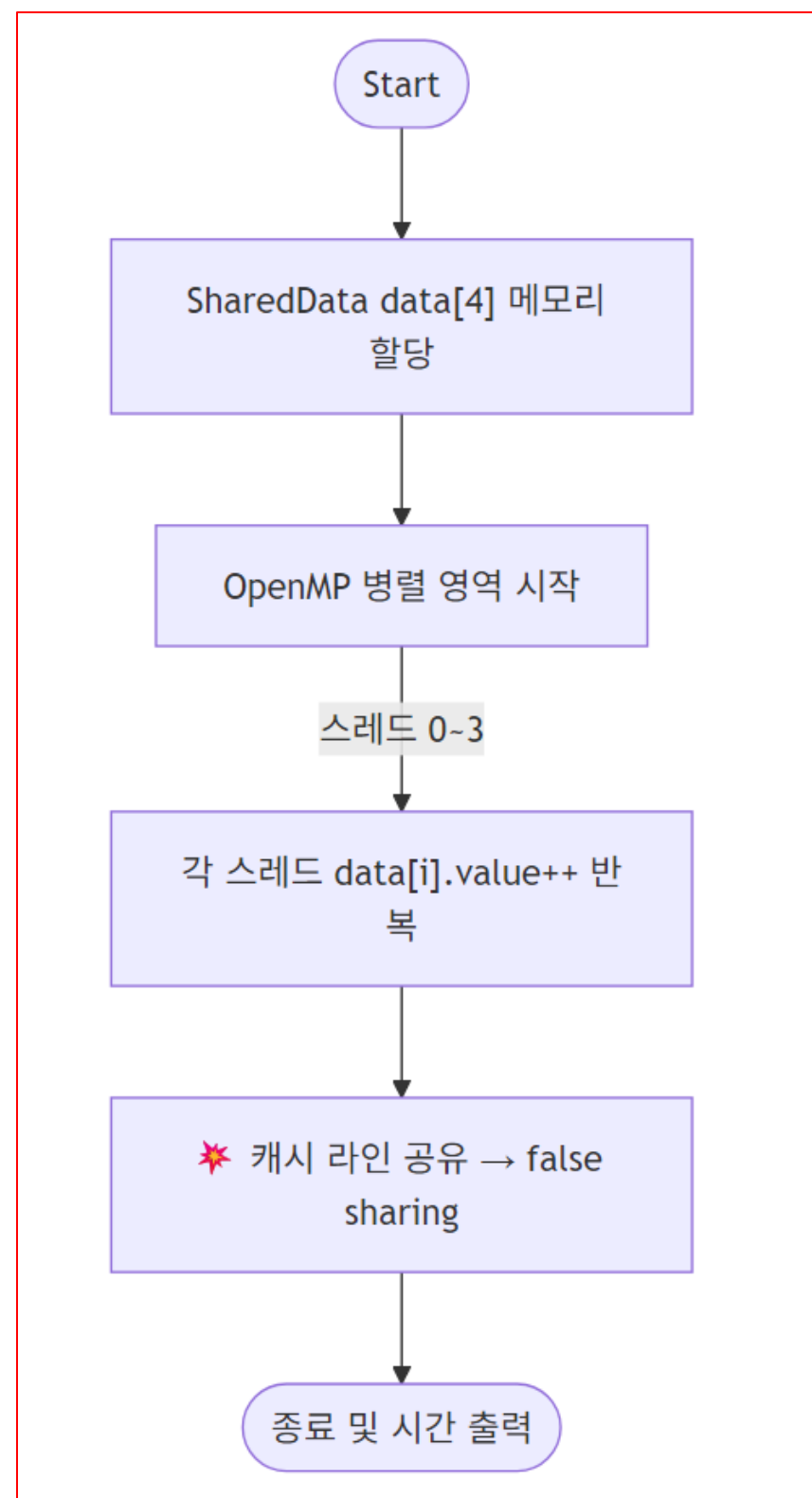
int main() {
    double start = omp_get_wtime();

    #pragma omp parallel num_threads(THREADS)
    {
        int tid = omp_get_thread_num();

        // 각 스레드가 자신만의 독립된 캐시 라인에 접근하므로 성능 향상
        for (int i = 0; i < N; i++) {
            data[tid].value++;
        }
    }

    double end = omp_get_wtime();
    printf("With Padding Time: %.5f seconds\n", end - start);
    return 0;
}
```

1. 기초



2. 클러스터 구성

캐시 일관성(cache coherence)

구분	Intel Xeon (5세대)	AMD EPYC (Zen 4, Turin)	ARM (Graviton, Ampere)
대표 아키텍처	Granite Rapids, Sapphire Rapids	Genoa, Bergamo, Turin	Neoverse N1/N2, V1
캐시 일관성 프로토콜	MESIF	MOESI	MESI-like or custom
캐시 계층	L1/L2 (Private), L3 (shared by tile/core)	L1/L2 (Private), L3 (per CCD)	다양한 구조, L3 없음도 존재
멀티 다이 구성	1~2개의 큰 다이, MCM 가능	칩렛 기반 CCD	단일 다이 또는 MCM
Coherence 범위	전체 SoC + UPI/QPI	CCD 단위로 독립, IFoP로 연결	일반적으로 전체 SoC 내 제한적

2. 클러스터 구성

캐시 일관성(cache coherence)

구분	Intel Xeon (5세대)	AMD EPYC (Zen 4, Turin)	ARM (Graviton, Ampere)
캐시 coherence 방식	MESIF (L3 공유)	MOESI (L3 분산)	Directory 기반 또는 MESI-like
L3 캐시 구성	큰 공유 L3	CCD별 독립 L3	단일 L3 또는 없음
노드 간 coherence	UPI로 유지	IFoP/Infinity Fabric	SoC 내 한정
스케일링	Mesh 상에서 중앙화 성능 저하 가능	CCD 스케일링 유리	Directory 방식으로 확장성 높음
false sharing 발생 가능성	높음 (공유 L3에서)	낮음 (L3 분산, CCD 격리)	낮음 (L2가 크고 독립적)

2. 클러스터 구성

캐시 일관성(cache coherence)

- Intel Xeon: 대형 공유 L3와 MESIF로 인해 cache contention이 발생할 수 있음
→ OpenMP 사용 시 주의
- AMD EPYC: CCD 구조와 분산 L3 덕분에 cache coherence 비용이 낮고 NUMA 최적화 시 성능 우수
- ARM: 전력 효율 위주이지만, Directory 기반 coherence 설계로 오히려 큰 코어 수에서 확장성이 뛰어남

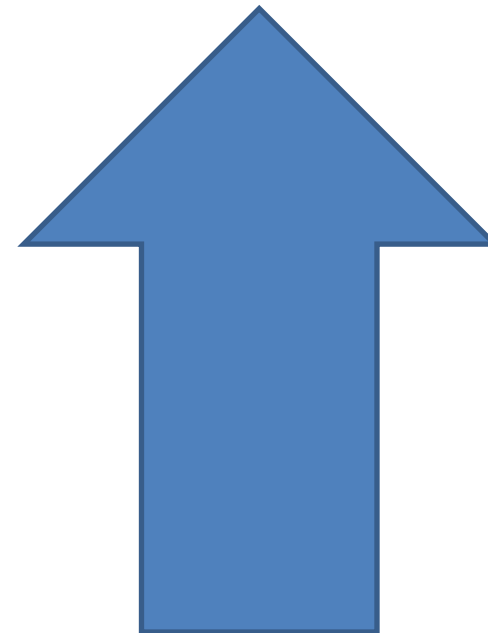
2-way 시스템, 즉 듀얼 소켓(Dual Socket) 서버 환경에서의 캐시 일관성(Cache Coherence)은 단일 소켓보다 훨씬 더 복잡하고 성능에 미치는 영향도 큼**

특히 Intel Xeon, AMD EPYC, ARM 아키텍처는 각기 다른 방식으로 **소켓 간 일관성(Coherence)**을 유지하며, 이 구조적 차이가 HPC에서의 NUMA 최적화, OpenMP, MPI 하이브리드 성능, 메모리 병목 등에 결정적 역할

1. 기초

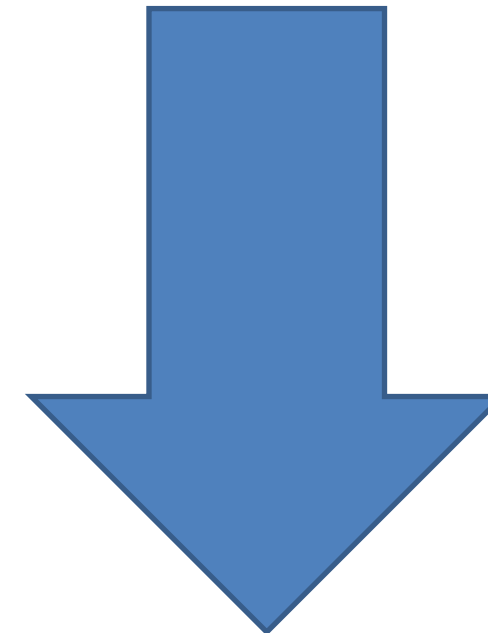
MEMORY

Large Size
Problems



문제
사이즈를 더욱 더
키우고

High Speed



해석 시간을
더욱 더
줄이고

1. 기초

MEMORY

항목	설명
작은 메모리 → 작은 문제 크기	대규모 행렬이나 데이터셋을 담을 수 없음
작은 문제 크기 → 연산량 부족	연산량이 적으면 CPU/GPU의 파이프라인이 덜 활용됨
작은 문제 크기 → 통신 오버헤드 증가	HPC에서 작은 문제는 오히려 통신 비용이 상대적으로 커짐
메모리 부족 → 페이지 폴트, 스와핑 발생	느린 디스크 I/O까지 사용하게 됨 → 성능 급감

1. 기초

MEMORY & Mesh

Mesh

- 해석 도메인을 작은 격자(셀)로 나눈 총 개수
- 단위: 보통 만 단위 (10^4), 백만 단위 (10^6), **억 단위 (10^8)**로 표현
- 셀 1개는 아주 작은 공간의 속도, 압력, 온도를 계산하는 최소 단위

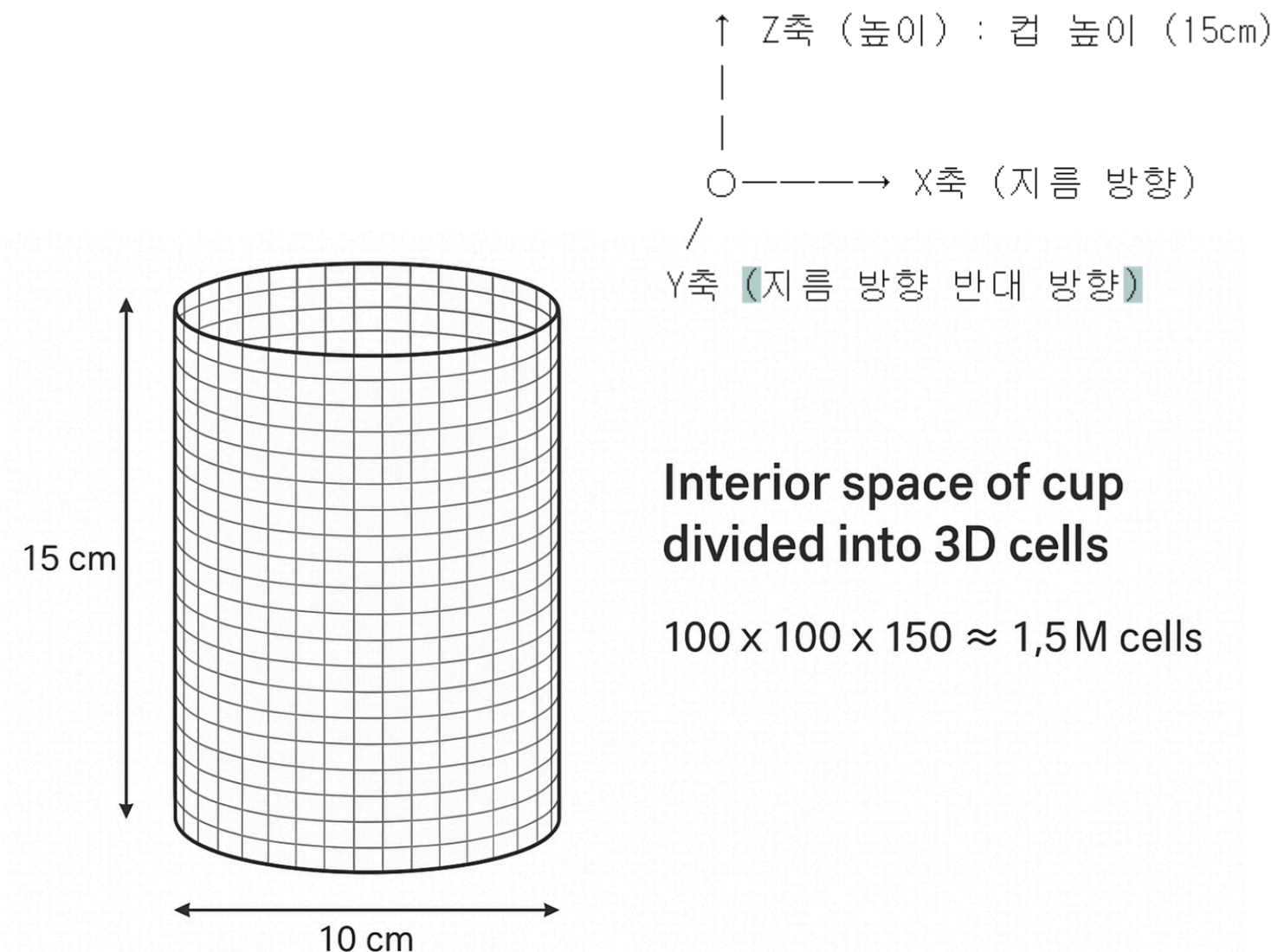
항목	설명
정의	해석 대상 물리 영역을 작은 셀(Cell) 또는 요소(Element)로 분할한 격자 구조
종류	Structured (정형 격자), Unstructured (비정형 격자), Hybrid
셀 형태	2D: 사각형, 삼각형 / 3D: 사면체, 육면체, 프리즘 등
역할	각 셀마다 방정식을 풀고 전체 시스템을 구성하여 해를 구함
Mesh Size	전체 셀 수 (예: 10 million = 10M cells), 또는 각 셀의 해상도

2. 클러스터 구성

MEMORY & Mesh

컵 내부 공간을 3D 셀로 나눈다고 가정 예: 지름 10cm, 높이 15cm → 0.001m 해상도로 쪼개면

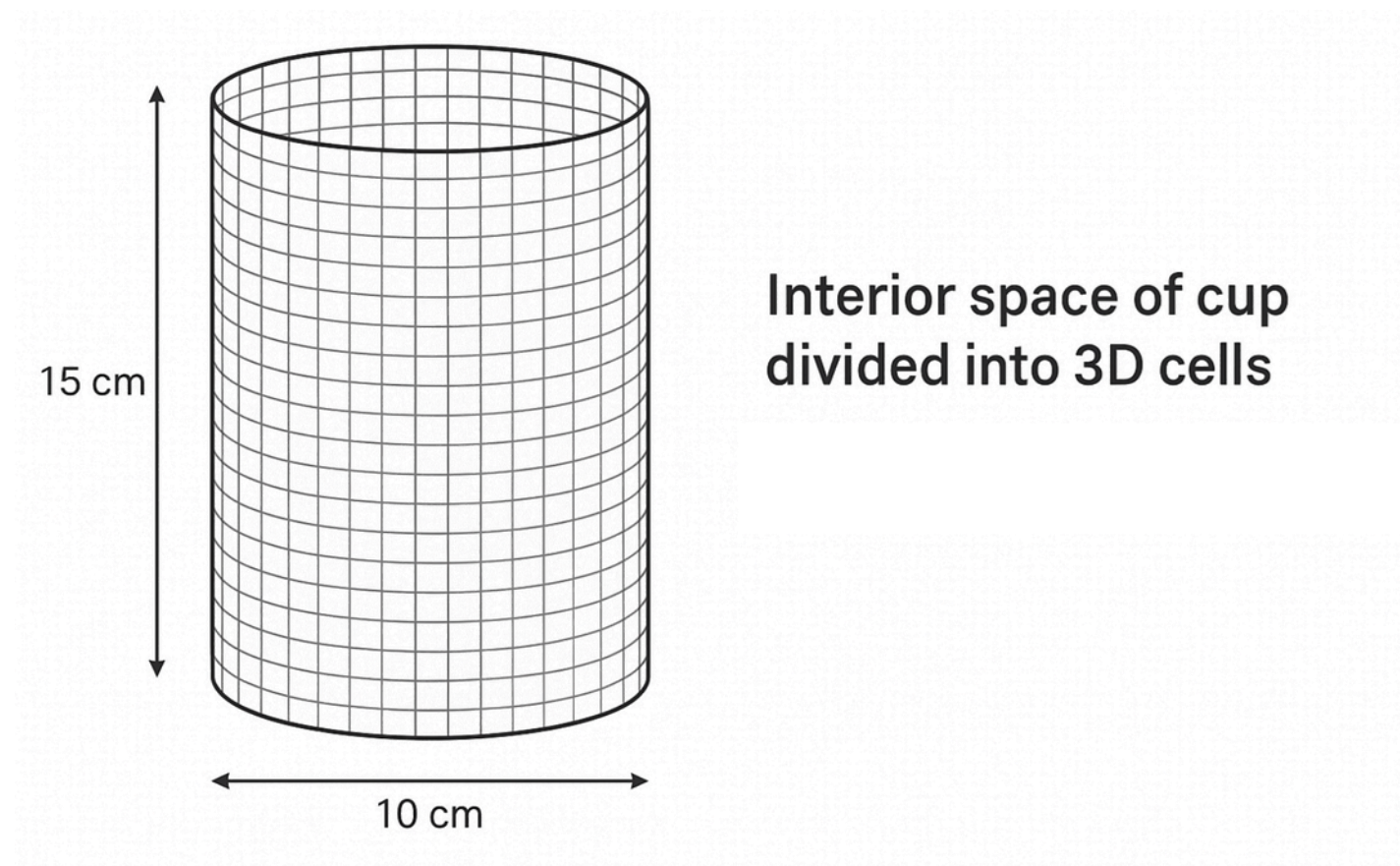
100 x 100 x 150 ≈ 1.5M 셀



항목	의미	수치	설명
100	X 방향 분할 수	컵의 지름을 0.001m (= 1mm) 단위로 나눈 개수	컵 지름이 10cm → 100mm → 100개
100	Y 방향 분할 수	위와 동일 (Y 방향도 지름에 해당)	컵 단면은 원형이라 X=Y
150	Z 방향 분할 수 (높이 방향)	컵 높이 15cm = 150mm → 1mm 간격으로 나누면 150개	

2. 클러스터 구성

해상도를 2배 높이면, 셀 수는 **8배**
(3D이므로 $x2 \times x2 \times x2 = x8$)



해상도 (단위 셀 크기)	Mesh 크기
1cm	$10 \times 10 \times 15 = 1,500$
1mm (0.001m)	$100 \times 100 \times 150 = 1.5M$
0.5mm	$200 \times 200 \times 300 = 12M$
0.25mm	$400 \times 400 \times 600 = 96M$

1. 기초

MEMORY & Mesh

- 매쉬 크기는 곧 해당 해석에 필요한 메모리량과 계산량을 결정

$$\text{필요 메모리} \approx (\text{Cell 수}) \times (\text{변수 수}) \times (\text{자료형 크기}) \times (\text{안전 계수})$$

예시 계산

매쉬 크기: 10M 셀 (10,000,000)

변수 수: 압력(P), 속도(Ux, Uy, Uz), 온도(T) = 5개 변수

자료형 크기: double = 8바이트

안전 계수: 2~3 (경계조건, 내부 버퍼, 유도 변수 등 포함)

$$\text{예상 메모리} = 10,000,000 \times 5 \times 8 \text{ Byte} \times 3 \approx 1.2 \text{ G}$$

1. 기초

MEMORY & Mesh

해석 유형	변수 수 예시	포함 변수
3 Vars (속도만)	3	Ux, Uy, Uz
5 Vars (속도+P+T)	5	Ux, Uy, Uz, P, T
10 Vars (난류+화학종)	10~20	+ k, ε, species, v_t, Y_i 등

Mesh Size	3 Vars	5 Vars	10 Vars
1M 셀	0.07 GB	0.11 GB	0.22 GB
10M 셀	0.67 GB	1.12 GB	2.24 GB
100M 셀 (1억)	6.7 GB	11.2 GB	22.4 GB
1B 셀 (10억)	67 GB	112 GB	224 GB

1. 기초

MEMORY & floating-point

정밀도	명칭	바이트 크기	설명
Single Precision	REAL*4, float32	4 byte	소수점 이하 약 7자리 정밀도
Double Precision	REAL*8, float64	8 byte	소수점 이하 약 15자리 정밀도
Quad Precision	REAL*16, float128	16 byte	소수점 이하 약 34자리 정밀도

1. 기초

MEMORY & floating-point

항목	FP16 (Half)	FP32 (Single)	FP64 (Double)
비트 수	16 bit	32 bit	64 bit
IEEE754 명칭	half precision	single precision	double precision
메모리 사용량	2 byte	4 byte	8 byte
유효 자릿수	약 3~4 자리	약 7 자리	약 15~16 자리
속도	가장 빠름 (GPU 기준)	빠름	느림
연산 장치	대부분 GPU 전용	CPU & GPU	CPU 전용 많음
사용 사례	딥러닝 추론, 대규모 그래픽	딥러닝 학습, CFD 간단 해석	정밀 물리 해석, FDS, OpenFOAM 등

1. 기초

MEMORY

$$C = A \times B$$

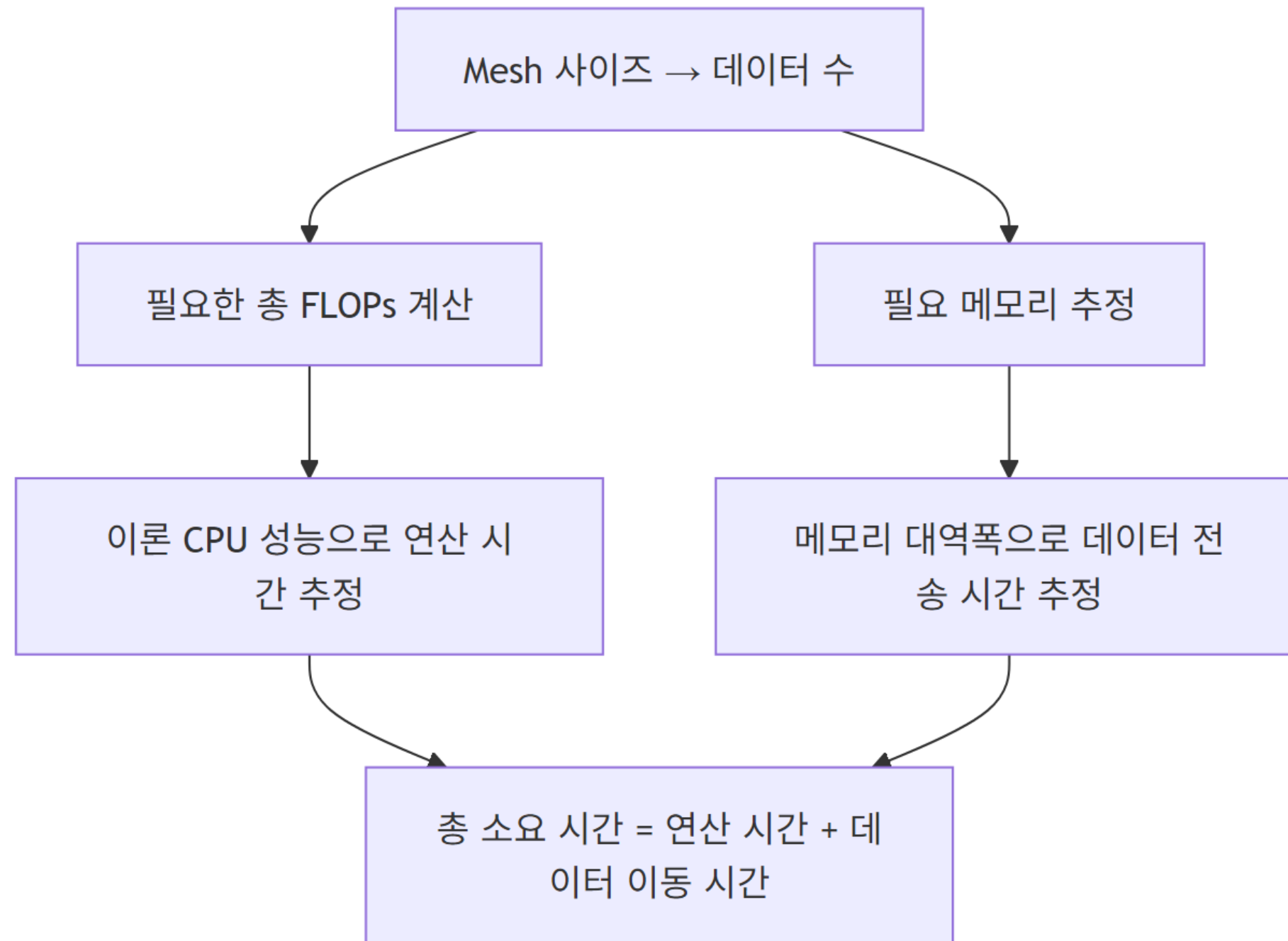
A: $M \times K$, B: $K \times N$, C: $M \times N$

총 필요 메모리 $\approx (M \times K + K \times N + M \times N) \times \text{sizeof(float)}$

- 즉, 행렬의 크기를 늘리려면 그만큼 메모리 공간이 충분해야 함
- 문제 크기를 늘릴 수 없다면, 하드웨어 자원을 덜 활용하게 되어 성능이 떨어짐

1. 기초

MEMORY



1. 기초

MEMORY

① 문제 크기 → 메모리 사용량 추정

예: FDS 또는 CFD/FEA 해석에서 3D Mesh:

$$N_x \times N_y \times N_z = \text{총 셀 수}$$

예: $100 \times 100 \times 150 = 1,500,000$ 셀

각 셀당 저장할 데이터 수 (온도, 압력, 속도 등) = D

데이터 타입: FP32 (4 Byte) 또는 FP64 (8 Byte)

② 총 연산량 (FLOPs) 추정

해석 유형에 따라 차이:

단순 포물선 방정식 → 셀당 50~100 FLOPs

복잡한 연산 (e.g. LES, Radiative Transfer) → 1,000~10,000 FLOPs/cell/ste

$$\text{총 FLOPs} = \# \text{cells} \times \text{FLOPs_per_cell} \times \# \text{iterations}$$

1. 기초

MEMORY

③ CPU 이론 성능

이론 FLOPS = Clock × Cores × IPC × FLOPs/cycle

$$\begin{aligned} & 5 \text{ GHz, } 8 \text{ cores, IPC}=2, \text{ FP32 } 8 \text{ FLOPs/cycle (AVX2)} \\ & = 3.5\text{e9} \times 8 \times 2 \times 8 = 448 \text{ GFLOPS} \end{aligned}$$

④ 연산 시간 예측

연산 시간 T_{compute} = 총 FLOPs / 이론 FLOPS

⑤ 메모리 대역폭 병목 고려 (optional)

DRAM Bandwidth = 예: 80 GB/s

총 데이터 이동량 = 셀 수 × 1회당 이동 바이트

총 시간 $T_{\text{total}} \approx \max(T_{\text{compute}}, T_{\text{memory}})$

대부분은 둘 중 병목이 되는 쪽이 전체 시간 결정

1회 step의 시간: 위를 iteration 수로 나누거나, FLOPs/step 기준 계산

1. 기초

MEMORY

예시 계산:

Mesh: $100 \times 100 \times 150 = 1.5\text{M}$ 셀

1 cell당 연산량: 1,000 FLOPs

총 step: 1,000

총 FLOPs:

$$1.5\text{M} \times 1,000 \times 1,000 = 1.5 \times 10^{12}$$

FLOPs

CPU 이론 성능: 300 GFLOPS

$$T_{\text{compute}} = 1.5e12 / 3e11 = 5 \text{ seconds}$$

메모리 요구량: $1.5\text{M} \text{ 셀} \times 10 \text{ floats} \times 4\text{B} = 60 \text{ MB}$
(메모리 병목 아님 → 연산 지배적)

항목	값	의미
1.5M 셀	$100 \times 100 \times 150 = 1,500,000$	3차원 해석 영역을 세 방향으로 나눈 총 셀(격자) 개수입니다. 이게 해석의 "문제 크기"
10 floats	예시: 속도(3), 압력(1), 온도(1), 난류계수(2), 기타 3개 등	한 셀에 저장되는 물리량(변수)의 개수. 셀마다 10개의 float형 데이터(부동소수점 수)가 저장된다고 가정.
4B (Byte)	FP32 = 32bit = 4 Byte	float (단정밀도 부동소수점 수)의 크기. ※ FP64 (double)일 경우는 8 Byte로 바뀜
60 MB	$1,500,000 \times 10 \times 4 = 60,000,000 \text{ Byte} = 60 \text{ MB}$	최종적으로 요구되는 메모리 용량. 이 정도면 단일 CPU에서도 메모리 문제 없이 처리 가능

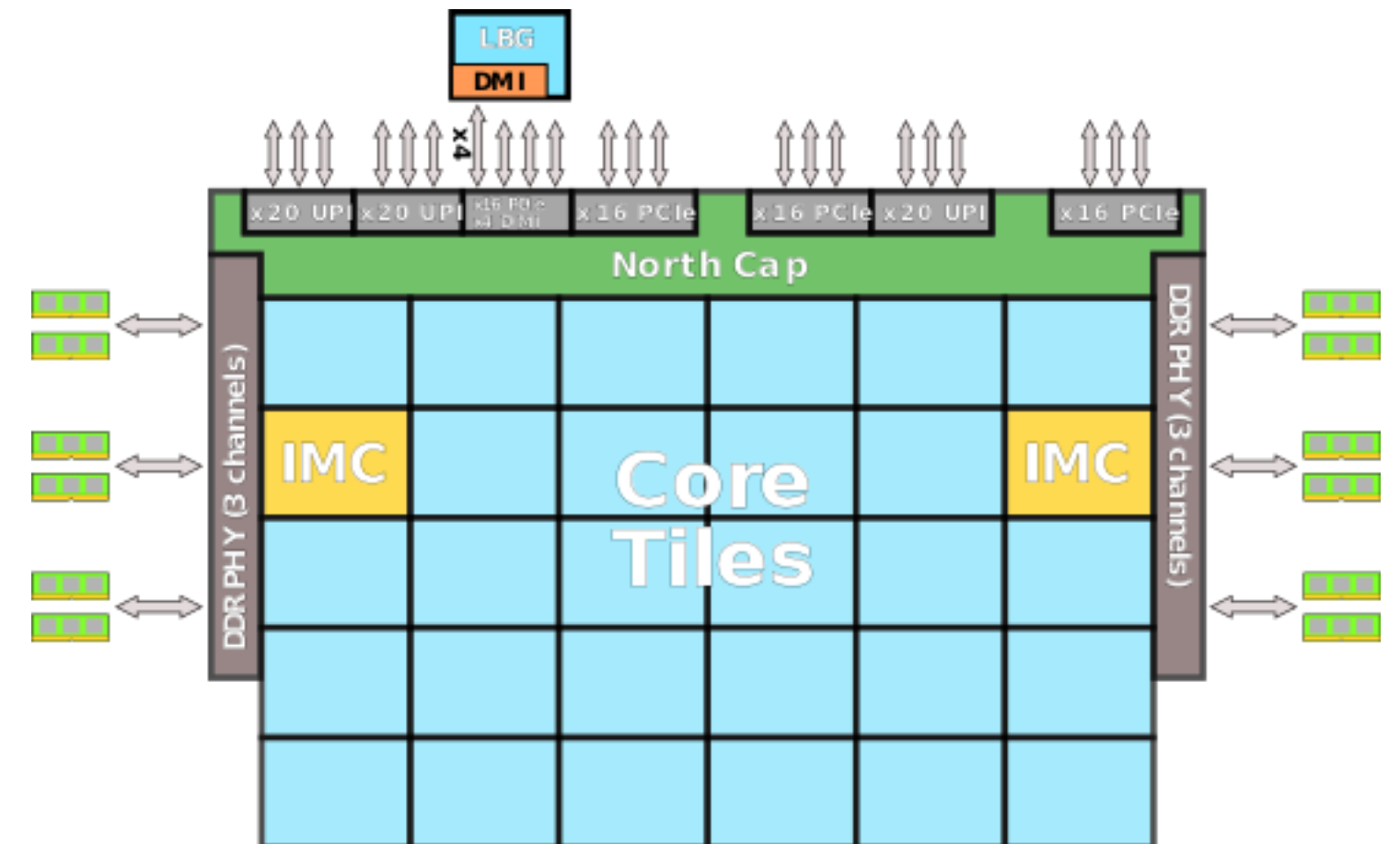
1. 기초

MEMORY : The Intel Second Generation Xeon Scalable: Cascade Lake

Memory

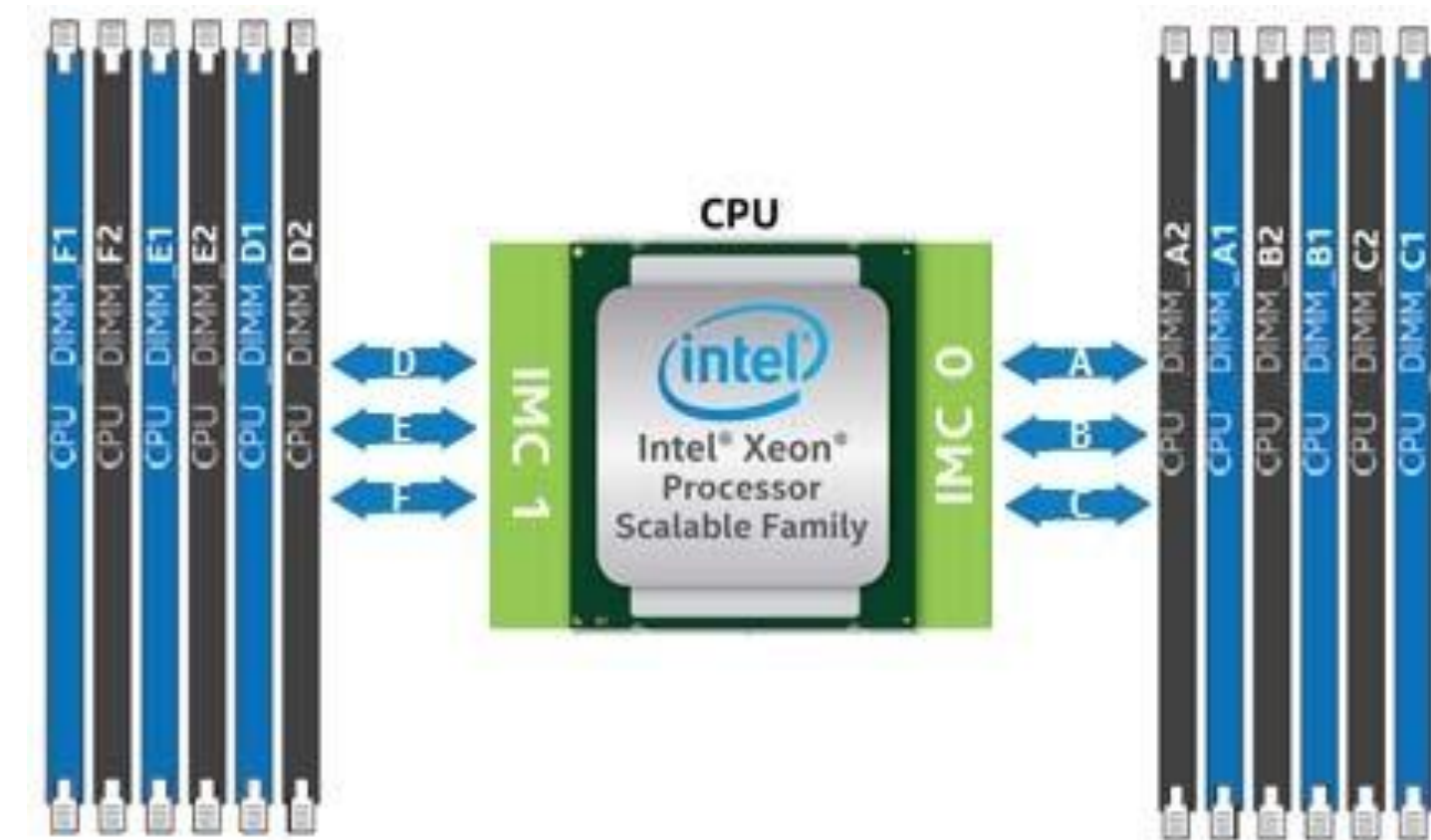
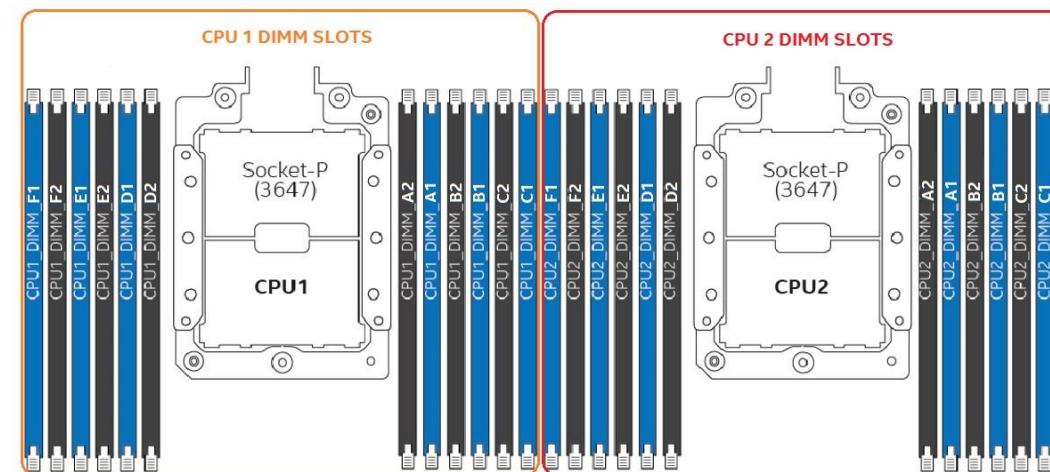
- Higher data rate (2933 MT/s, up from 2666 MT/s)
- Standard support for up to 1 TiB per socket (up from 768 GiB)
- Extended memory support for up to 2 TiB per socket (up from 1.5 TiB)
- Large memory support for up to 4.5 TiB per socket

- DRAM6 channels of DDR4, up to 2666 MT/s
- RDIMM and LRDIMM
- bandwidth of 21.33 GB/s
- aggregated bandwidth of 128 GB/s



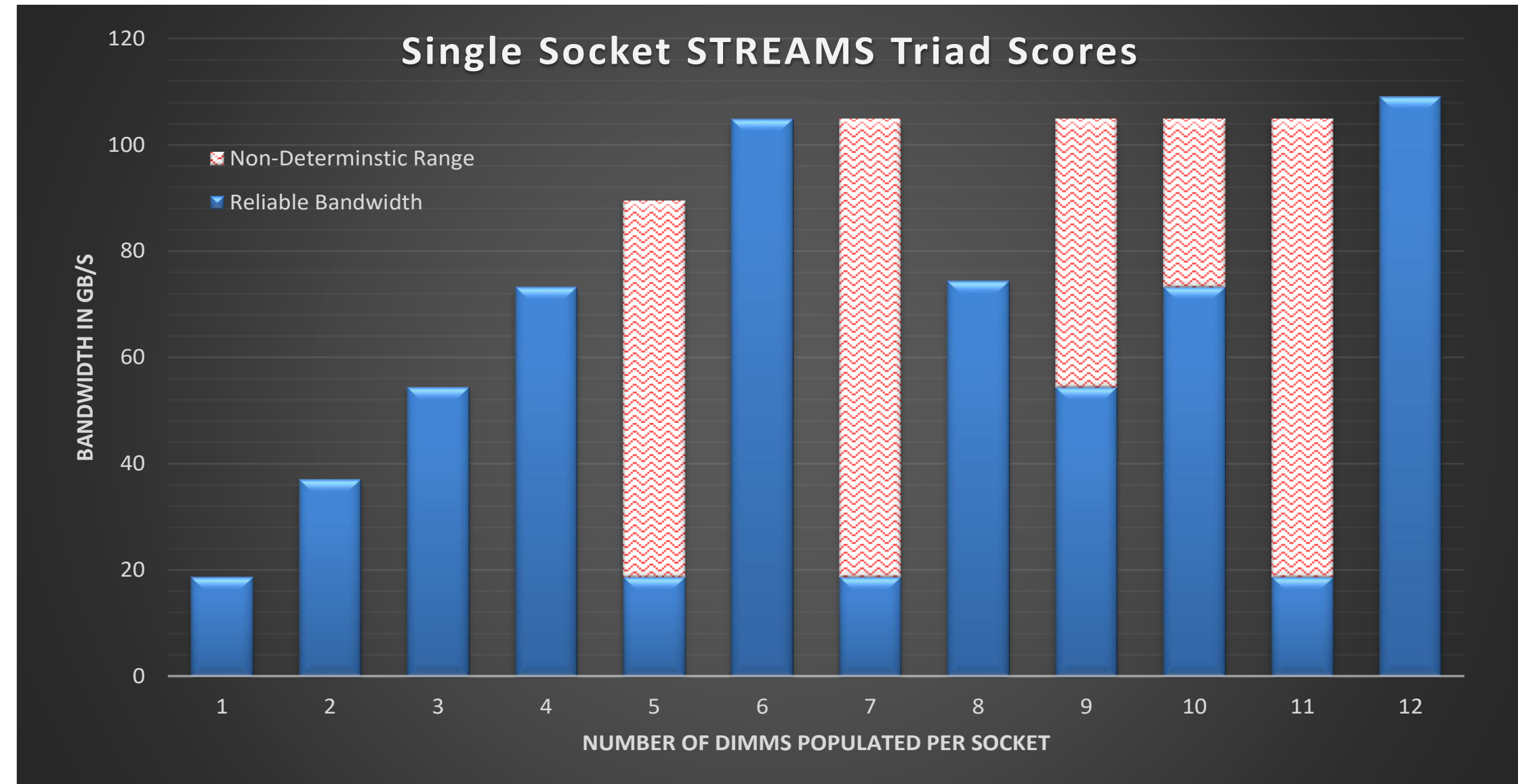
1. 기초

MEMORY : The Intel Second Generation Xeon Scalable: Cascade Lake



1. 기초

STREAMS Triad Performance vs DIMM Population



- X축: NUMBER OF DIMMS POPULATED PER SOCKET
→ 소켓당 DIMM(메모리 모듈) 장착 개수 (1~12개)
- Y축: BANDWIDTH IN GB/S
→ STREAM Triad 벤치마크로 측정한 메모리 대역폭 (GB/s)

- 파란색 부분: Reliable Bandwidth
→ 권장 Population 구성에서 얻을 수 있는 안정적(예측 가능) 대역폭
- 빨간색 지그재그 부분: Non-Deterministic Range
→ 비권장 DIMM 구성에서 대역폭이 들쭉날쭉할 수 있는 불확정 범위

1. 기초

STREAMS Triad Performance vs DIMM Population 결과

- DIMM 개수가 늘수록 대역폭은 증가한다.
DIMM을 채울수록 채널/뱅크가 더 많이 활성화되어 대역폭이 커짐.
- 일부 DIMM 개수에서는 Non-Deterministic 대역폭이 발생한다.
예: 5, 7, 9, 11개 DIMM에서는 대역폭이 불확정(붉은 영역).
이건 메모리 채널/뱅크가 불균형해져서 워크로드에 따라 성능 편차가 커지기 때문.
- 6, 8, 12개 DIMM은 가장 안정적인 최대 대역폭을 제공한다.
채널당 균형 있는 슬롯 배치로, STREAM 벤치마크에서 최대 대역폭이 예측 가능.

메모리 채널 구조(예: 6채널, 8채널 등)에 맞춰 짝수/정배수로 DIMM을 구성

비권장 개수(불균형)는 대역폭이 크게 흔들려 HPC나 고성능 워크로드에 부적합

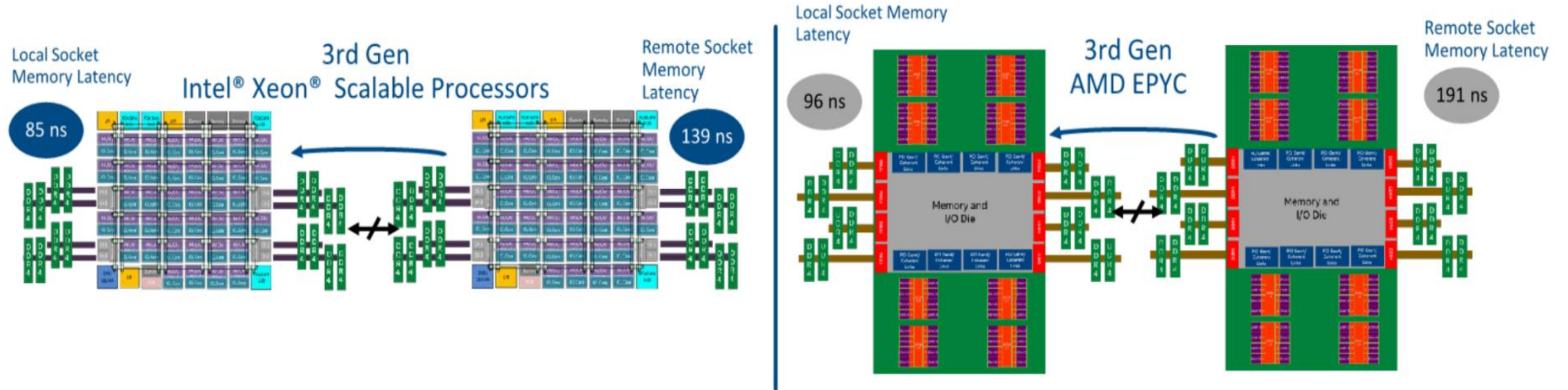
1. 기초

권장 DIMM 배치 가이드라인 표

항목	권장 배치 원칙	이유 / 주의사항
싱글 채널	1개 채널만 사용	성능 낮음, 최소 구성
듀얼 채널	2개 슬롯, 채널별 1개씩 동일한 용량 DIMM	대역폭 2배, 슬롯 색상 매칭
쿼드 채널	4개 슬롯, 채널별 1개씩 동일 용량 DIMM	고성능 CPU(HEDT, 서버) 기본
Hexa/Octa 채널	채널 수만큼 동일 DIMM 설치	최신 서버/워크스테이션(Hexa=6, Octa=8)
2DPC (Dual DIMM Per Channel)	2랭크/4랭크 DIMM 권장, 채널별 동일	슬롯 2개씩 짝을 땀 같은 속도/용량/랭크
NUMA 소켓별 균형	CPU 소켓별 동일 DIMM 개수	NUMA 불균형 방지, 원격 메모리 접근 최소화
DIMM 속도/랭크 혼용 금지	동일 속도/랭크 DIMM만 혼용	혼용 시 낮은 속도로 동작, 비결정적 성능
빈 슬롯 방지	같은 채널/뱅크는 모두 채우기	비균형하면 Non-deterministic 대역폭 발생

1. 기초

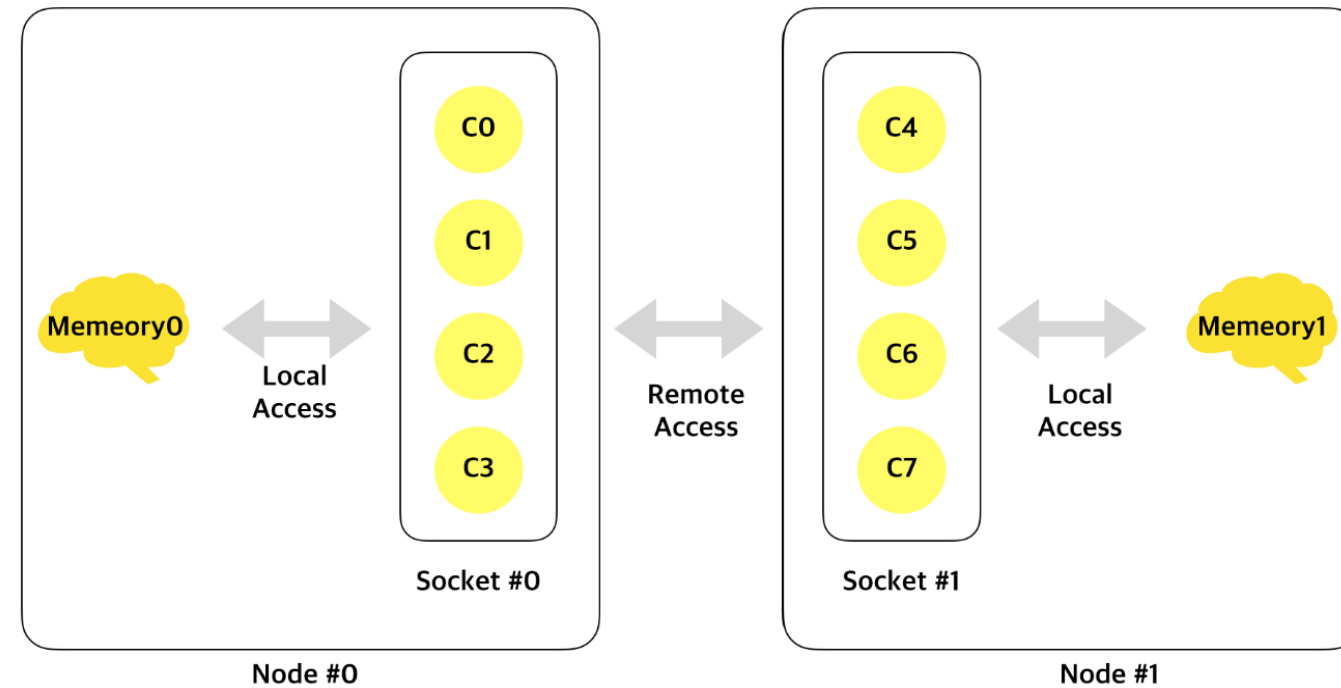
MEMORY : Local & Remote Latency



	Intel® Xeon® Platinum 8380 Processor (Ice Lake)	AMD EPYC 7763 Processor (Milan)	Intel® Xeon® Platinum 8280 Processor (Cascade Lake)
Memory Controller	On die – 8 ch	Multi chip module – 8 ch	On Die – 6ch
Max DIMM Capability	2 DPC 3200/2933/2666 (SKU dependent) <small>New: PMem runs at memory channel speed</small>	1 DPC 3200 2 DPC 2933/2666	1 DPC 2933/2 DPC 2666 (SKU dependent)
DRAM read latency local socket, ns	85	96	81
DRAM read latency (remote socket), ns	139	191	138
Max Memory Capacity per Socket	6TB (DDR+PMem). 4TB (DDR)	4TB (DDR)	4.5 TB (DDR+PMem). 3TB (DDR)

1. 기초

MEMORY : NUMA(불균일 기억 장치 접근, Non-Uniform Memory Access)



- 0번 CPU가 자신의 로컬 메모리에 접근하는 동안 1번 CPU도 자신의 메모리에 접근할 수 있어서 성능이 향상된다.
- 로컬 메모리의 양이 모자라면 다른 CPU에 붙어있는 메모리에 접근이 필요하게 되고, 이때 메모리 접근에 시간이 소요되어 예상치 못한 성능 저하가 발생
- 로컬 메모리에서 얼마나 많이 메모리 접근이 일어나느냐가 성능향상의 가장 중요한 포인트
- 각각의 CPU마다 별도의 메모리가 있는데 이와 같이 메모리에 접근하는 방식을 로컬 액세스(Local Access)라고 한다. 그리고 이렇게 CPU와 메모리를 합쳐서 노드라고 부른다. NUMA에서는 자신의 메모리가 아닌 다른 노드에 있는 메모리에도 접근할 수 있으며 이것을 리모트 액세스(Remote Access)라고 부른다.

1. 기초

MEMORY : NUMA(불균일 기억 장치 접근, Non-Uniform Memory Access)

A Performance-Stable NUMA Management Scheme for Linux-Based HPC Systems

JAEHYUN SONG¹, MINWOO AHN¹, GYUSUN LEE¹, EUISEONG SEO²,
AND JINKYU JEONG³, (Member, IEEE)

¹Department of Electrical and Computer Engineering, Sungkyunkwan University, Suwon 16419, Republic of Korea

²Department of Computer Science and Engineering, Sungkyunkwan University, Suwon 16419, Republic of Korea

³Department of Semiconductor Systems Engineering, Sungkyunkwan University, Suwon 16419, Republic of Korea

Corresponding author: Jinkyu Jeong (jinkyu@skku.edu)

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korean Government [Ministry of Science and ICT (MSIT)] under Grant NRF-2016M3C4A7952587 and Grant NRF-2020R1A2C2102406.

ABSTRACT Linux is becoming the de-facto standard operating system for today's high-performance computing (HPC) systems because it can satisfy the demands of many HPC systems for rich operating system (OS) features. However, owing to features intended for the general-purpose OS, Linux has many OS noise sources such as page faults or thread migrations that can result in the unstable performance of HPC application. Furthermore, in the case of the non-uniform memory access (NUMA) architecture, which has different memory access latencies to local and remote memory nodes, the performance stability of the application can be more exacerbated by the OS noise. In this paper, we address the OS noise caused by Linux in the NUMA architecture and propose a novel performance-stable NUMA management scheme called *Stable-NUMA*. *Stable-NUMA* comprises three techniques for improving performance stability: two-level thread clustering, state-based page placement, and selective page profiling. Our proposed *Stable-NUMA* scheme significantly alleviates OS noise and enhances the local memory access ratio of the NUMA system as compared to Linux. We implemented *Stable-NUMA* in Linux and experimented with various HPC workloads. The evaluation results demonstrated that *Stable-NUMA* outperforms Linux with and without its NUMA-aware feature by up to 25% in terms of average performance and 73% in terms of performance stability.

이 문서는 Linux 기반 HPC 시스템을 위한 성능이 안정적인 NUMA 관리 체계를 제시합니다. 우리의 분석에 따르면 CPU 로드 밸런서와 NUMA 인식 기능 간의 스레드 마이그레이션 충돌은 스레드 및 메모리 페이지의 핑퐁 마이그레이션으로 인해 성능을 저하시킬 수 있습니다. 이 충돌은 스레드 및 페이지 마이그레이션의 실행 간 변형으로 인해 응용 프로그램의 성능 가변성을 증가시킵니다.

따라서 우리의 계획은 CPU 로드 밸런서에서 노드 간 CPU 로드 밸런싱을 분리하고 이를 NUMA 인식 페이지 및 스레드 배치와 정렬하여 이 문제를 해결합니다.

메모리 액세스 프로파일링 정보는 애플리케이션에 대한 성능 영향을 최소화하여 신중하게 수집됩니다. 수집된 정보는 그룹 수가 물리적 NUMA 노드 수에 맞춰진 그룹 스레드에 사용됩니다. 스레드는 또한 스레드에 의한 원격 메모리 액세스 수를 최소화하기 위해 신중하게 그룹화됩니다.

앞서 언급한 메커니즘을 통해 평가 결과는 NUMA 인식 기능이 있거나 없는 Linux 커널과 비교하여 *Stable-NUMA*가 성능 변동을 훨씬 억제한다는 것을 보여주었습니다.

NUMA 아키텍처의 이 기능을 처리하기 위해 Linux 커널에는 백그라운드에서 실행되고 스레드의 메모리 액세스를 특성화하고 메모리 액세스 지역성을 개선하기 위해 스레드 및/또는 페이지를 마이그레이션하는 *Auto-NUMA*라는 NUMA 인식 기능이 있습니다. [26].

그러나 *Auto-NUMA*는 (1) 메모리 액세스의 런타임 프로파일링으로 인해 높은 오버헤드가 발생하고 애플리케이션 성능의 실행 간 변동이 증가하고 (2) 다음과 같은 두 가지 이유로 성능 안정성이 저하됩니다. CPU 로드 밸런서와 *Auto-NUMA* 간의 의사 결정 정책은 NUMA 노드 간에 스레드와 페이지의 핑퐁 마이그레이션을 초래하여 모든 결정을 무용지물로 만듭니다. 이러한 오버헤드는 OS 노이즈로 작용하여 애플리케이션의 성능 안정성을 악화시킵니다.

2. 클러스터 구성

Context switching

cache coherence (캐시 일관성)

Cache Conflict vs Cache Contention

1. 기초

Accelerator

NVIDIA V100



NVIDIA Tesla V100 액셀러레이터는 7.8 TFLOP / s (배정 밀도)의 최대 성능을 가지며 Summit에서 수행되는 대부분의 계산 작업에 기여합니다. 각 V100에는 80 개의 스트리밍 멀티 프로세서 (SM), 16GB의 고 대역폭 메모리 (HBM2) 및 SM에 사용 가능한 6MB L2 캐시가 포함되어 있습니다. GigaThread Engine은 SM간에 작업을 분배하고 16 비트 HBM2 메모리에 대한 액세스를 제어하는 (8) 512 비트 메모리 컨트롤러를 담당합니다. V100은 NVIDIA의 NVLink 인터커넥트를 사용하여 GPU간에 그리고 CPU에서 GPU로 데이터를 전달합니다.

1. 기초

Accelerator

GPU Programming

```
// Host code
int main()
{
    // Allocate vectors in device memory
    size_t size = N * sizeof(float);
    float* d_A;
    cudaMalloc((void**)&d_A, size);
    float* d_B;
    cudaMalloc((void**)&d_B, size);
    float* d_C;
    cudaMalloc((void**)&d_C, size);

    // Copy vectors from host memory to device memory
    // h_A and h_B are input vectors stored in host memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int threadsPerGrid =
        (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<threadsPerGrid, threadsPerBlock>>>(d_A, d_B, d_C);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```

Time

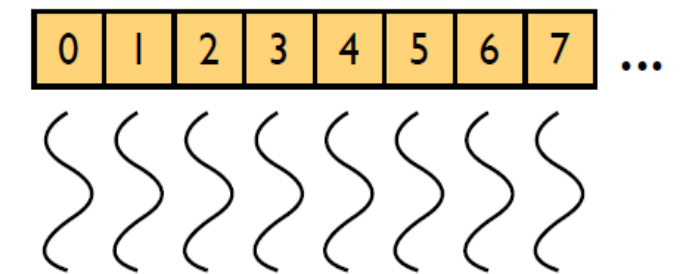
Memory
Allocation

Memory
Copy : Host → GPU

Kernel
Call

Memory
Copy : GPU → Host

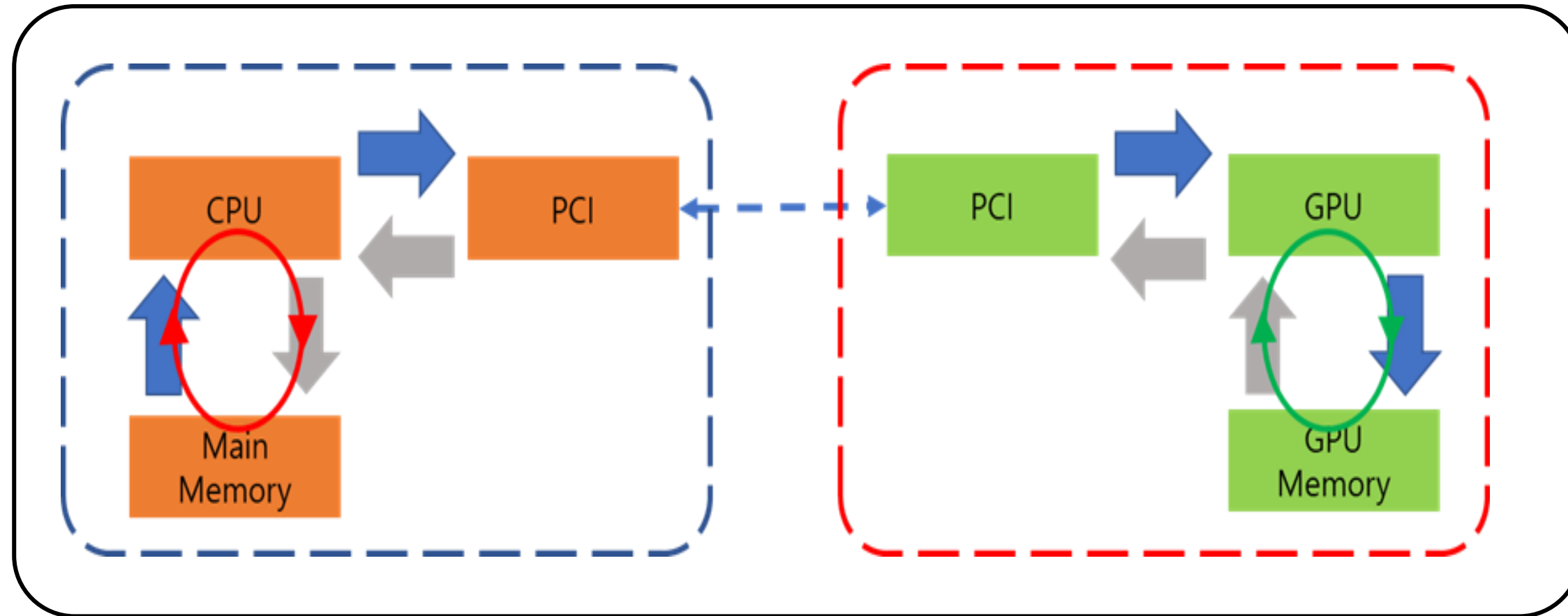
Free GPU
Memory



```
// Device code
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}
```

1. 기초

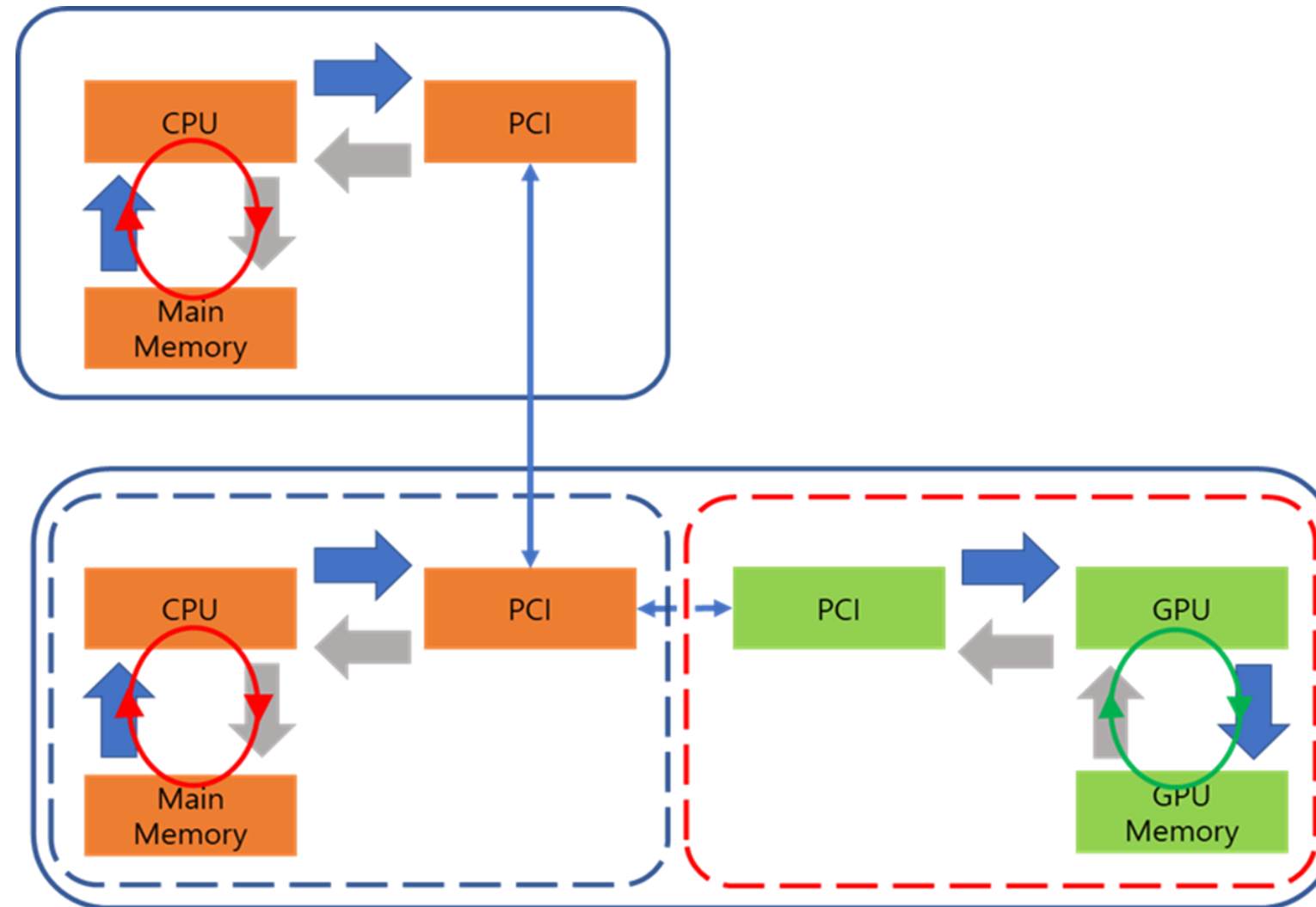
Accelerator



1. 프로세싱 처리는 Host CPU와 메인 메모리에서 GPU 메모리로의 복사를 통해 진행됩니다.
2. 복사된 데이터가 GPU 내부에서 실행됩니다.
3. 모든 결과물은 GPU의 메모리에 저장됩니다.
4. Host에서는 결과물 데이터만 GPU 메모리에서 다시 메인 메모리로 복사하여 모든 처리가 완료됩니다.

1. 기초

Accelerator



Cluster
Server

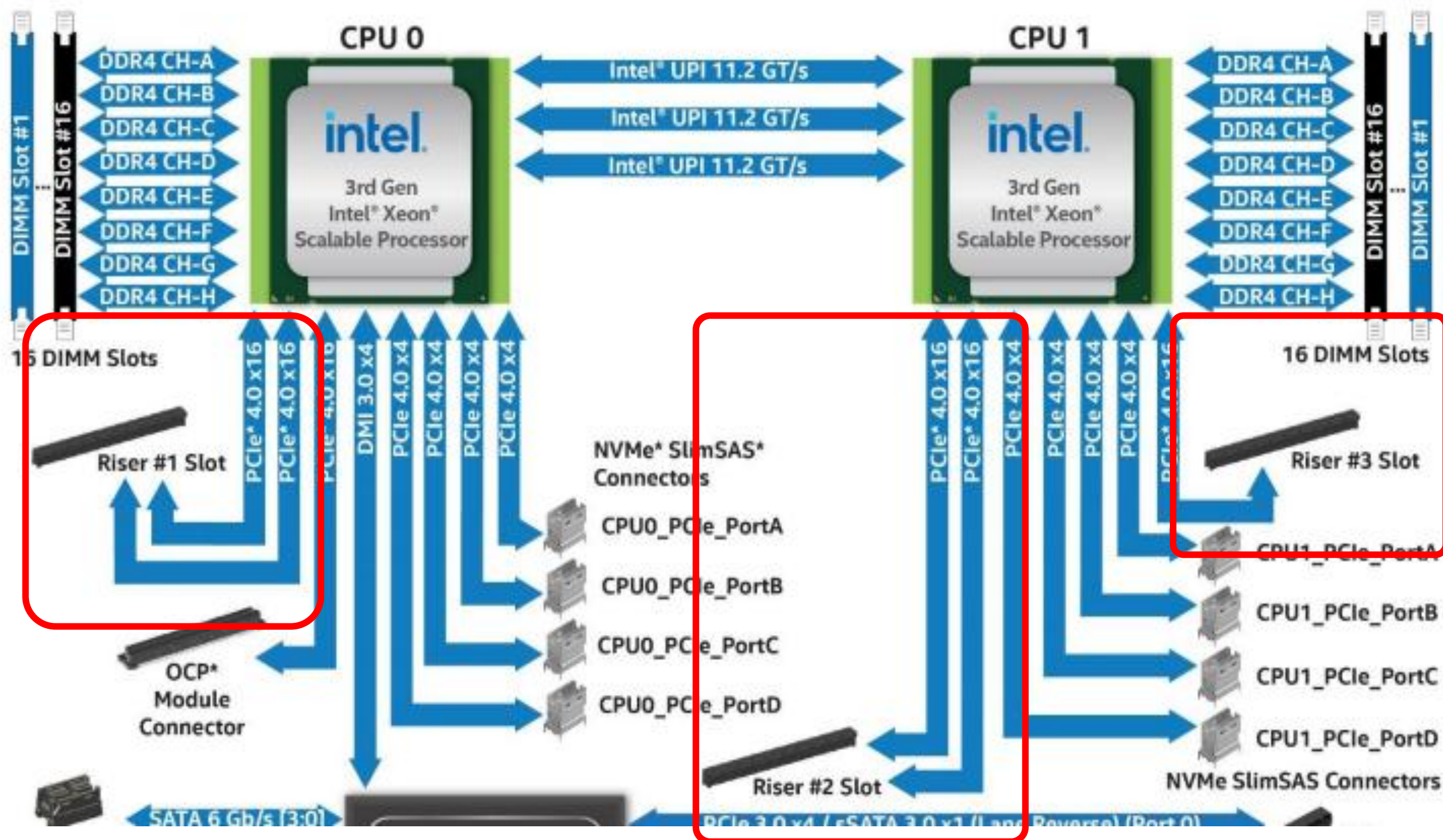
GPU
Server

프로세싱 처리는 Host CPU와 메인 메모리에서 네트워크를 통하여 타 GPU 서버의 CPU 메모리로 복사 진행

1. Host CPU와 메인 메모리에서 GPU 메모리로의 복사진행
2. GPU 메모리로 복사 진행
3. 복사된 데이터가 GPU 내부에서 실행
4. 모든 결과물은 GPU의 메모리에 저장
5. GPU Host에서는 결과물 데이터만 GPU 메모리에서 다시 메인 메모리로 복사
6. GPU 서버에서 네트워크를 통하여 다시 서버 CPU 로 복사되어 모든 처리가 완료

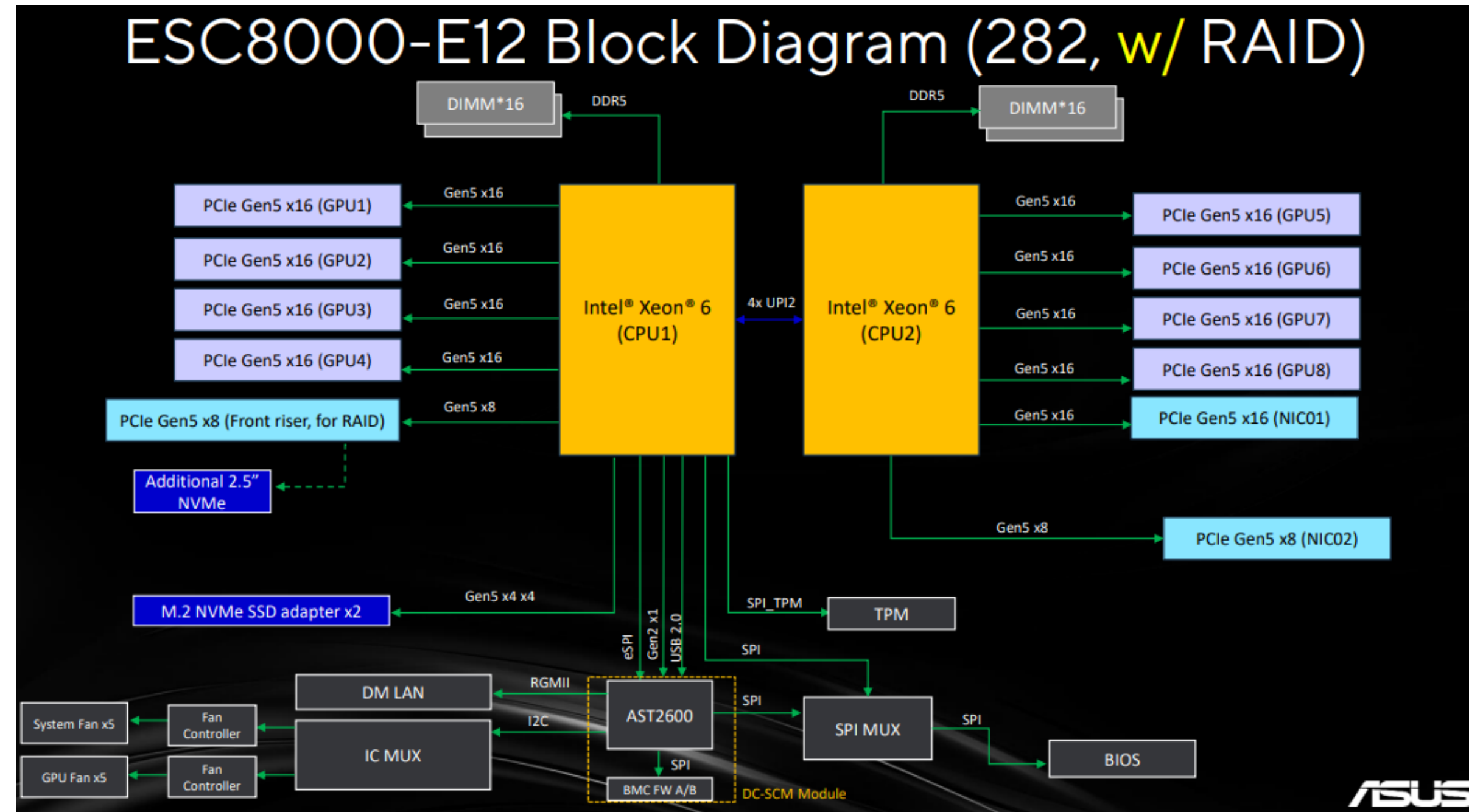
1. 기초

Accelerator



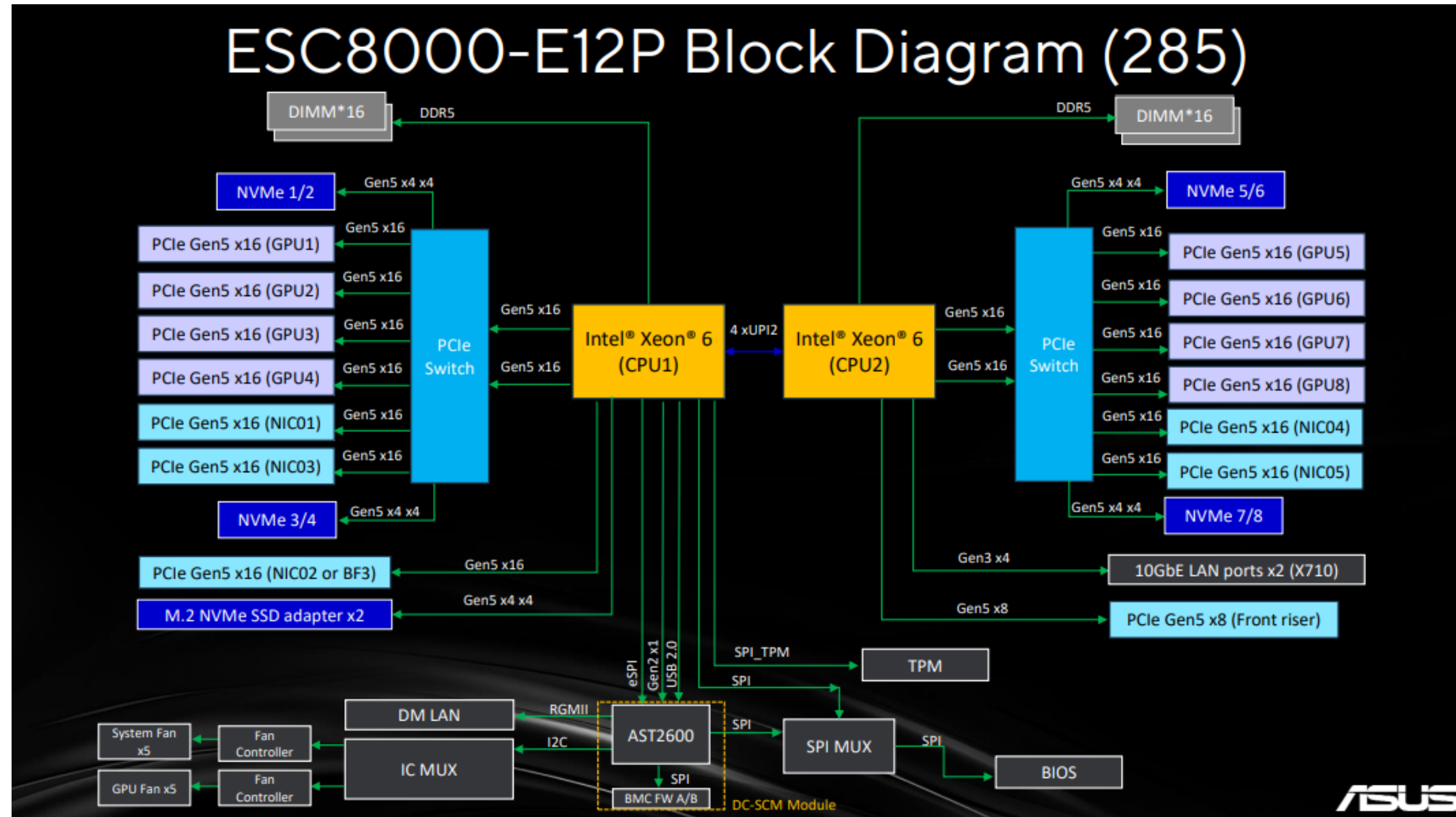
1. 기초

Accelerator



1. 기초

Accelerator



1. 기초

Accelerator

구분	PCIe 스위치 없이	PCIe 스위치 사용
CPU↔GPU 대역폭	최대	스위치 포트 대역폭 한계
GPU↔GPU P2P	CPU 버스를 거침 예를 들어 CPU0-GPU0, CPU1-→GPU1와 같이 구성시 GPU0 <-> GPU1 통신은 GPU0 <-> CPU0 0 <-> UPI <-> CPU1 <-> GPU1 과 같이 통신 필요	스위치 내부에서 바로 교환 가능 (효율적) 예를 들어 CPU0-GPU0, CPU1-→GPU1와 같이 구성시 GPU0 <-> GPU1 통신은 GPU0 <-> PCI SWitch <-> GPU1 과 같이 통신이 이루어짐
레이턴시	가장 낮음	스위치 hop 만큼 소폭 증가
확장성	CPU PCIe 레인 수에 의존	스위치 추가로 유연 확장
멀티 노드 NVMe, NIC 등 공유 I/O	레인 부족 시 병목	스위치로 공유 I/O 조율 가능
추천 워크로드	GPU 간 독립적으로 사용 되는 워크로드	GPU 와 GPU 간 빈번한 워크로드 : AI 관련 스위치가 내부 P2P 경로 제공
비추천 워크로드	AI 관련	HPC 워크로드 HPC 워크로드는 CPU <-> GPU 통신이 많음

1. 기초

Accelerator

ASUS ESC8000-E12P 같은 8GPU 고밀도 서버는 실제로 내부에 두 개의 PCIe 스위치(PLC) 로 구성이 됩니다.

CPU 소켓 1 ---- PCIe uplink ---- PCIe 스위치 A , CPU 소켓 2 ---- PCIe uplink ---- PCIe 스위치 B

스위치 A ---- GPU1, GPU2, GPU3, GPU4 , 스위치 B ---- GPU5, GPU6, GPU7, GPU8

이때 스위치 A와 스위치 B는 일반적으로 직접 연결되지 않습니다!

- CPU ↔ GPU 통신은 CPU↔스위치 A/B↔GPU 구조로 hop 1개
- GPU ↔ GPU가 같은 스위치 내에 있으면 direct P2P 가능 (스위치 내부 routing)
- GPU ↔ GPU가 다른 스위치에 있으면 **CPU interconnect(QPI/UPI)**를 경유해야 함

이 경우에는 CPU와 CPU간 통신이 되므로 NUMA hop이 발생이 됩니다.

이런 구조적인 문제로 스케줄링이라는 기술이 필요 합니다. (핵심은 PCI switch 가 있든 없든, 같은 프로세서내에 격리를 시킨다)

예들어 slurm 을 가지고 gpu job을 스케줄링을 할 수 있는데,

예: 4GPU job을 같은 스위치 그룹에 묶어 배치

#SBATCH --gres=gpu:4

#SBATCH --cpus-per-task=32

export CUDA_VISIBLE_DEVICES=0,1,2,3 # 같은 스위치 그룹

numactl --cpunodebind=0 --membind=0 python train.py

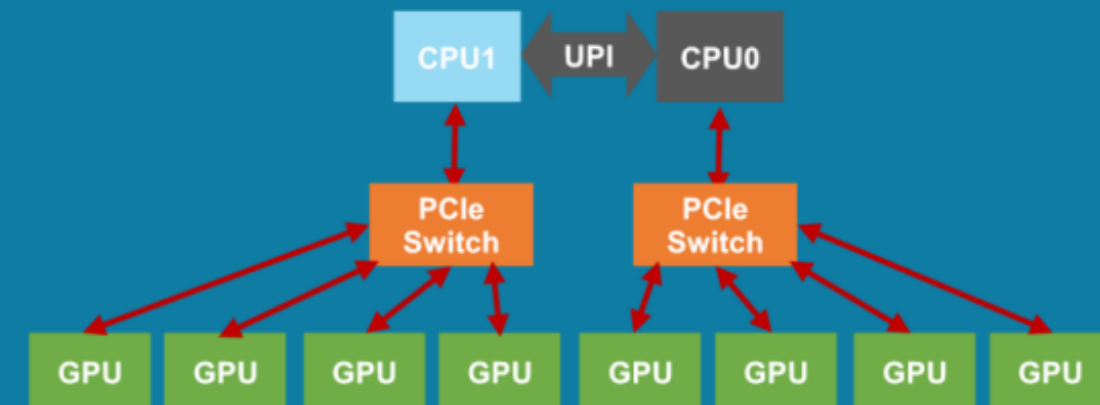
와 같이 GPU를 해당 스위치 그룹과 해당 프로세서 (numa를 사용 하여 메모리 바인딩)내에 격리

1. 기초

Accelerator

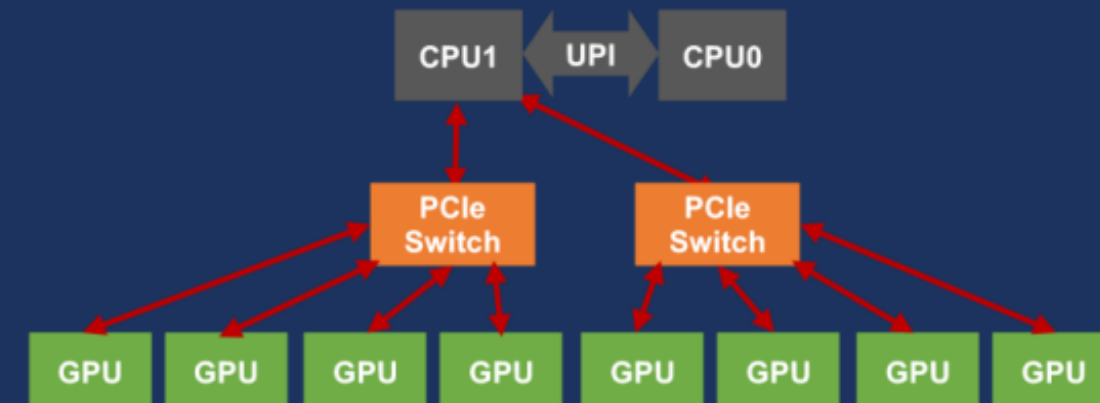
Balance

- Suitable for GPU pass-through virtualization
- GPU cloud application
- Small and medium-sized deep learning training
- Inference, public cloud and HPC scenarios



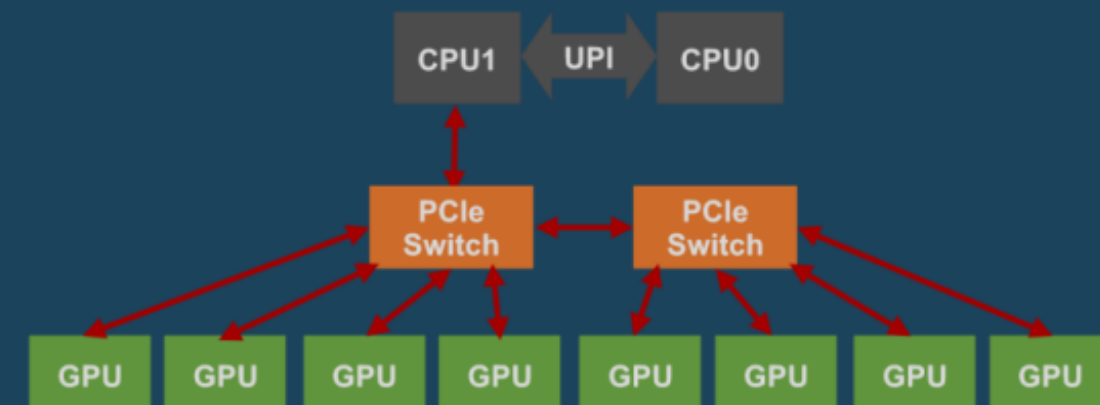
Common

- Excellent AI training performance
- GPU P2P communication
- Most deep learning application scenarios



Cascade

- Some AI training models have the best performance
- GPU P2P communication
- Large-scale deep learning application scenarios



2. 클러스터 구성

GPU Memory

문제 : AI 모델 사이즈가 671B가 돼, 이를 위한 최적의 시스템 구성을 해줘

모델 파라미터 메모리 요구량 계산

671B 파라미터 \times 2byte(16bit, FP16) \approx 1.36TB

671B 파라미터 \times 4byte(32bit, FP32) \approx 2.72TB

훈련 시:

Optimizer state까지 고려해야 하므로 파라미터 크기 \times 2~4배가 필요

예: Adam Optimizer \rightarrow 4배 이상.

\rightarrow 훈련 메모리만 최소 5TB 이상 필요.

추론 시:

FP16 inference라면 파라미터 메모리만 로드하면 됨 \rightarrow 1.36TB 수준.

KV Cache까지 고려하면 시퀀스 길이 \times 배치 사이즈에 따라 수백 GB의 추가 GPU 메모리가 필요

2. 클러스터 구성

GPU Memory

문제 : AI 모델 사이즈가 671B가 돼, 이를 위한 최적의 시스템 구성을 해줘

항목	추정치
파라미터 수	671B (billion parameters)
FP16 모델 크기	약 1.34TB
FP32 모델 크기	약 2.68TB
Optimizer state	48배 → 훈련 시 약 510TB VRAM 필요
KV Cache	시퀀스 길이/배치 크기에 따라 추가로 수백 GB ~ TB

2. 클러스터 구성

GPU Memory

문제 : AI 모델 사이즈가 671B가 돼, 이를 위한 최적의 시스템 구성을 해줘

1) 추론(Inference) 시

- FP16 기준 모델 파라미터 1.34TB → 최소 1.5TB GPU HBM 필요
- 실시간 서비스 KV Cache까지 고려 → 2~3TB 이상 필요
- 대기시간(latency) 줄이려면 NVLink, NVSwitch 필수

2) 훈련(Training) 시

- Optimizer state + Activation Checkpoint → 파라미터 메모리의 최소 4배
→ 5~10TB GPU 메모리 필요

2. 클러스터 구성

GPU Memory

문제 : AI 모델 사이즈가 671B가 돼, 이를 위한 최적의 시스템 구성을 해줘

GPU	FP16 메모리	HBM	NVLink/InfiniBand
NVIDIA H100 SXM	80GB	HBM2e	NVLink/NVSwitch
NVIDIA B100	192GB	HBM3e	NVLink/NVSwitch
NVIDIA GB200 NVL72	144TB 시스템 메모리	HBM3e	NVLink Switch

추론만 할 경우

- NVIDIA B100 192GB HBM × 최소 8~16장 → 모델 파라미터만 해도 1.5TB 필요.
- KV Cache 때문에 더 많은 GPU 필요.
- GB200 NVL72 같은 모듈화 슈퍼클러스터는 72 GPU당 144TB HBM과 3.5TB/s NVLink 대역폭.

훈련 시

- 최신 LLM 훈련은 기본적으로 수백~수천개의 GPU를 사용
- Microsoft, OpenAI, Google DeepMind는 NVIDIA HGX 기반 클러스터 (10,000개 GPU 규모) 운영.

2. 클러스터 구성

GPU Memory – LLM 파라미터 관련 메모리 구조

구성	설명	보통 크기
모델 파라미터	가중치 (Weights)	1 × 파라미터 수
옵티마이저 상태	Adam이면 4 × 파라미터 수 (m, v, optional)	4 ×
그라디언트	역전파 시 필요	1 ×
Activation Memory	순전파/역전파 중 중간 값	보통 파라미터의 1~2배
KV Cache	(추론 시) Attention 저장	배치/시퀀스 길이에 따라 추가

훈련시 메모리

가중치 + 옵티마이저 + 그라디언트 + 액티베이션 $\approx 1 \times + 4 \times + 1 \times + 1 \sim 2 \times$

2. 클러스터 구성

GPU Memory – 모델별 메모리 및 GPU (예시)

모델구분	파라미터 (B)	FP16 파라미터(GB)	훈련 VRAM 요구(GB)	추론 VRAM 요구(GB, 양자화/4bit)	H200 필요수(훈련)	H200 필요수(추론)
SLM (7B)	7B	14GB	56~112GB	3.5GB + KV	1	1
LLM (70B)	70B	140GB	560~1120GB	35GB + KV	4~8	1~2
SLLM (671B)	671B	1,340GB	510TB	335GB + KV	35~75	3~5

1. 기초

Storage

클러스터 구성에서 가장 어려운 부분

역할	설명
유저 홈	유저가 가지고 있는 개인 홈 디렉토리를 말하며, 각 노드들이 유저 홈을 서로 공유
유저 작업 공간	유저가 가진 job을 공유 하며, job에 대한 결과 역시 이곳에 쓰여지게 됨. 문제의 사이즈가 커짐에 따라. 이 부분 역시 커져야 하며, 안정성 및 속도가 고려가 되어야 함
중간 데이터 저장	계산 도중 생성이 되는 임시 데이터 파일이 저장이 되는 공간. 이에, 가장 빠른 디스크 IO가 필요

- 사용자가 원하는 것
큰 용량, 높은 성능, 안정성
- 많은 기술, 프로젝트, 회사들이 존재
- 리눅스 클러스터에서 스토리지를 위한 간단한 솔루션은 아직 없음

1. 기초

RAID

Redundant Array of Independent Disks

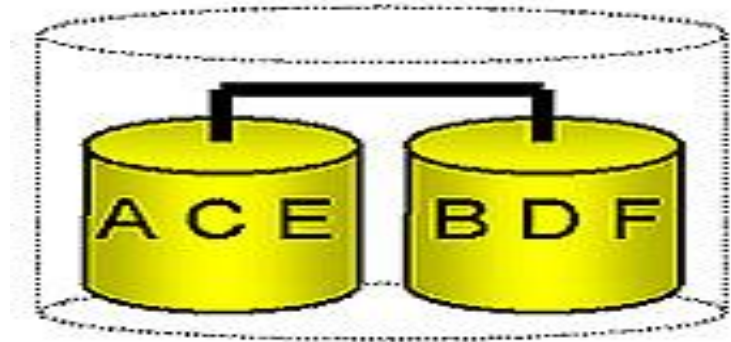
Redundant Array of Inexpensive Disks

1. 기초

RAID Level

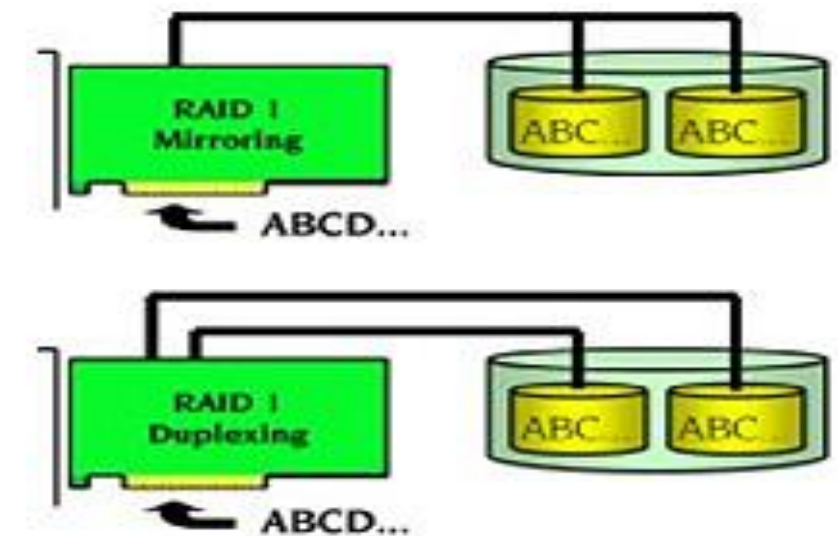
RAID 0 - 스트라이핑

스트라이핑이란 데이터 스트림을 똑같은 크기의 집합체로 분할해서 그것을 동등한 사이즈의 집합체(스트라이프 블록)로서 복수의 디스크에 배치를 한다. 장점으로는 가장 빠른 속도를 보일 수 있으나, 단점으로 디스크 하나가 불량일 생기면, 전체 스토리지 사용이 불가능 해 진다.



RAID 1 - 미러링

이 타입의 어레이는 종래의 폴트 톨러런트 어플리케이션으로 사용된 쉘도우 또는 미러링이라고 한다. RAID 1은 속도보다도 데이터의 가용성을 최적화하기 위해서 설계되었으며 각 기록 트랜잭션을 한 개 이상의 미러 디스크로 복제한.

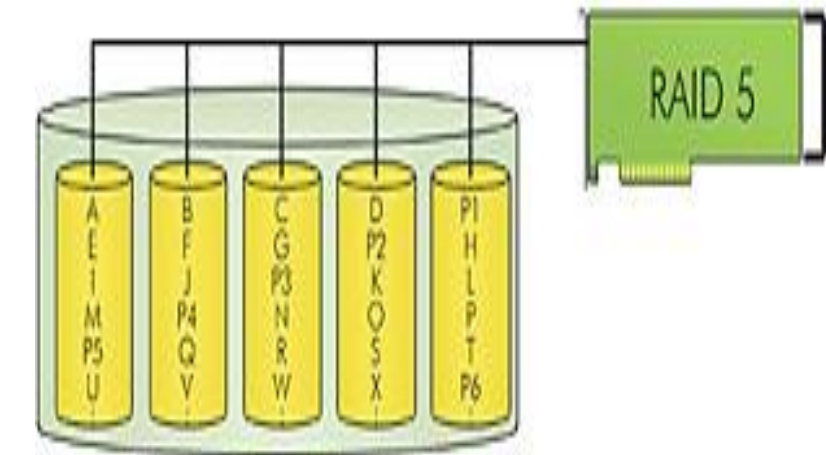


1. 기초

RAID Level

RAID 5 - 스트라이핑과 패리티 삽입

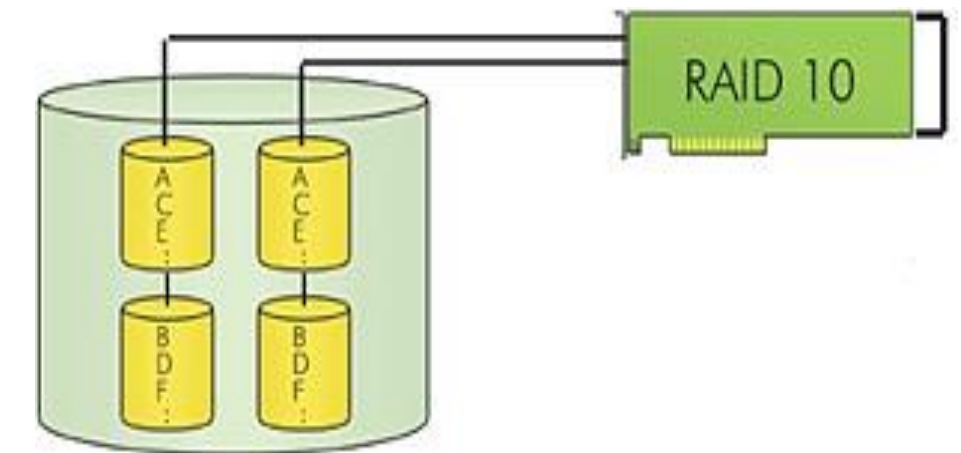
RAID 5는 스트라이핑의 성능상 이점을 살리면서 데이터에 여러 수정 정보(패리티)를 추가함으로써 데이터 손실에 대한 약점을 줄일 수 있다



RAID 10 - 스트라이핑 + 미러링

기존의 2가지 RAID 레벨을 결합시킴으로써 몇 가지 효과적인 구성을 만들어낼 수 있다.

스트라이핑과 미러링을 적용한 시스템의 신뢰성은 미러링에 의해서 가능해지는 높은 리더던트 때문에 매우 뛰어나다. RAID 10 시스템은 각각의 단독 디스크의 장애를 견딜 수 있으며 게다가 RAID 5와는 달리 성능의 약화를 거의 초래하지 않고 데이터 입. 출력을 고속으로 처리할 수 있습니다. 그러나 보호를 필요로 하는 데이터는 단순한 독립 스피들의 2배의 디스크 공간을 점유함으로써 RAID 10 시스템은 미러링 시스템과 똑같은 비용이 필요하게 되면 RAID 5보다 고가가 됩니다.



1. 기초

RAID Level : 예상 성능 개요

RAID Level	디스크 수 최소	읽기 성능	쓰기 성능	특징	이유
RAID 0 (Stripe)	2	매우 빠름	매우 빠름	성능 극대화, 무중복	데이터가 디스크 여러 개에 분산되어 동시 I/O
RAID 1 (Mirror)	2	빠름	느림~보통	데이터 중복 저장	읽기는 여러 디스크에서 동시에 가능, 쓰기는 동일 데이터 복제 때문에 디스크 수만큼 중복
RAID 5 (Stripe + Parity)	3	빠름	느림~보통	성능+중복 절충	읽기는 Striping 덕에 빠름, 쓰기는 Parity 계산 + 디스크에 분산 쓰기로 느려짐
RAID 6 (Stripe + Double Parity)	4	빠름	느림	Parity 이중화	RAID 5보다 Parity가 더 많아 쓰기 부하가 커짐
RAID 10 (1+0, Mirror of Stripes)	4	매우 빠름	빠름	성능+중복 극대화	Stripe로 읽기/쓰기 성능 확보 + 미러로 안전성
RAID 50 (5+0)	6 이상	빠름	보통~빠름	RAID 5 그룹을 Stripe	RAID 5 그룹으로 중복 확보 + Stripe로 성능 확보

1. 기초

RAID Level : 예상 성능 개요

RAID Level	Throughput Read	Throughput Write
RAID 0	$N \times \text{Disk}$	$N \times \text{Disk}$
RAID 1	$N \times \text{Disk}$	$1 \times \text{Disk}$
RAID 5	$(N-1) \times \text{Disk}$	$(N-1) \times \text{Disk} * 0.75$
RAID 6	$(N-2) \times \text{Disk}$	$(N-2) \times \text{Disk} * 0.5 \sim 0.7$
RAID 10	$(N/2) \times \text{Disk}$	$(N/2) \times \text{Disk}$

1. 기초

RAID Level : 계산 예 (총 디스크 수: 10 디스크 1개 성능: 150 IOPS, 200 MB/s)

RAID0

- 읽기 IOPS: $10 \times 150 = 1,500$ IOPS
- 쓰기 IOPS: 1,500 IOPS
- 읽기 MB/s: $10 \times 200 = 2,000$ MB/s
- 쓰기 MB/s: 2,000 MB/s

RAID10 (10개 디스크 → 5개 데이터 세트)

- 읽기 IOPS: $10 \times 150 = 1,500$ IOPS
- 쓰기 IOPS: $5 \times 150 = 750$ IOPS
- 읽기 MB/s: 2,000 MB/s
- 쓰기 MB/s: 1,000 MB/s

RAID5 (N-1 데이터 디스크)

- 데이터 디스크: 9
- 읽기 IOPS: $10 \times 150 = 1,500$ IOPS
- 쓰기 IOPS: $9 \times 150 \div 4 = 337.5$ IOPS
- 읽기 MB/s: 2,000 MB/s
- 쓰기 MB/s: $9 \times 200 \times 0.25 = 450$ MB/s

RAID6 (N-2 데이터 디스크)

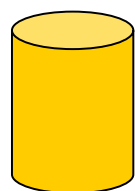
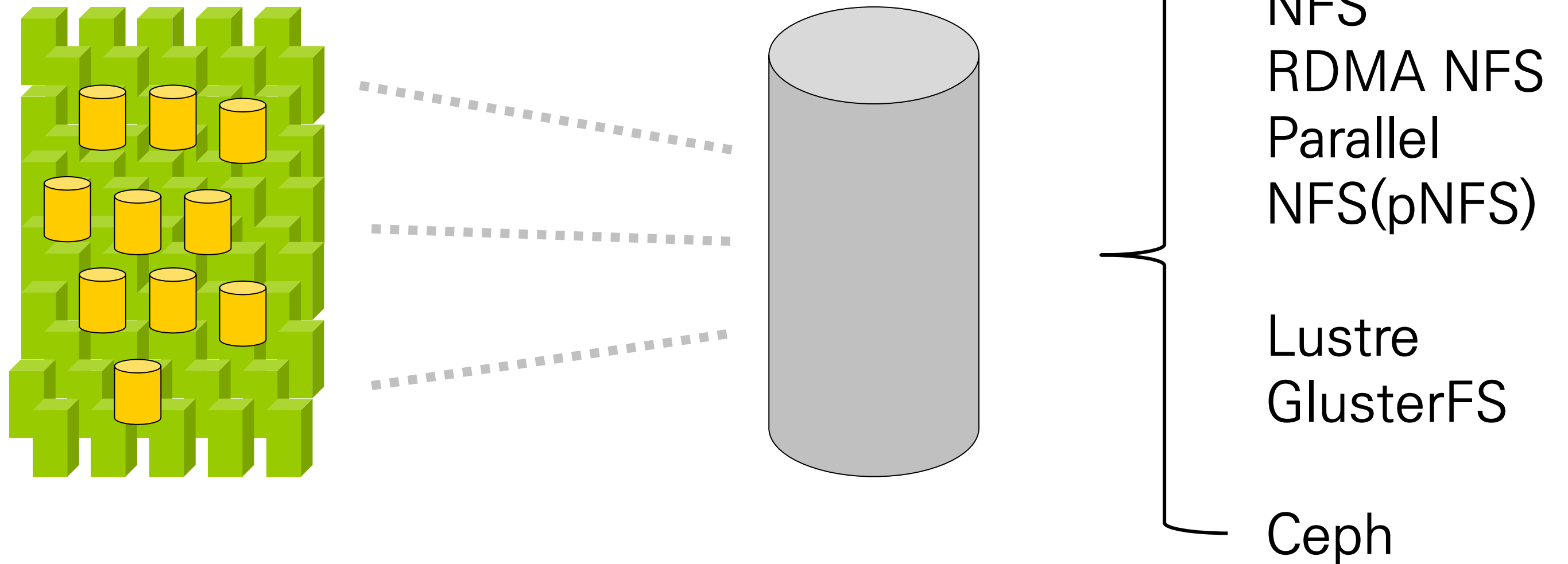
- 데이터 디스크: 8
- 읽기 IOPS: $10 \times 150 = 1,500$ IOPS
- 쓰기 IOPS: $8 \times 150 \div 6 \approx 200$ IOPS
- 읽기 MB/s: 2,000 MB/s
- 쓰기 MB/s: $8 \times 200 \times 0.25 \approx 400$ MB/s

RAID Level	읽기 IOPS	쓰기 IOPS	읽기 MB/s	쓰기 MB/s
RAID0	1,500	1,500	2,000	2,000
RAID10	1,500	750	2,000	1,000
RAID5	1,500	338	2,000	450
RAID6	1,500	200	2,000	400

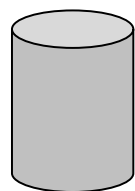
1. 기초

병렬 파일 시스템

File system Terminology



Local Disk/ Filesystem
Only accessible on its node



Global File systems
Accessible across entire cluster

Parallel File System

서버 또는 클라이언트를 추가하면 성능이 향상될 수 있는 글로벌 파일 시스템입니다. 단일 파일에 대한 여러 클라이언트의 협력 액세스 지원

1. 기초

병렬 파일 시스템 - Lustre

MGS(Management Server)

MGS는 하나 이상의 Lustre 파일 시스템에 대한 구성 정보를 저장하고 이 정보를 다른 Lustre 호스트에 제공.

MDS(Metadata Server)

MDS는 Lustre 파일 시스템에 대한 인덱스 또는 네임스페이스를 제공. 메타데이터 콘텐츠는 MDT(Meta Data Targets)라는 볼륨에 저장. Lustre 파일 시스템의 디렉토리 구조와 파일 이름, 권한, 확장 속성 및 파일 레이아웃이 MDT에 기록, Lustre 파일 시스템마다 MDT가 하나 이상 있어야 함

OSS(Object Storage Server)

OSS는 Lustre 파일 시스템의 모든 파일 콘텐츠에 대한 대량 데이터 저장 영역을 제공. 각 OSS는 OST(Object Storage Targets)라는 스토리지 볼륨 세트에 대한 액세스를 제공.

1. 기초

병렬 파일 시스템 - Lustre

Lustre 예상 가용 공간(TiB) $\approx 0.99 * \text{어레이 수} * (\text{어레이당 디스크 수}) * 0.8 * \text{HDD 크기(TB)} * 10^{12}/2^{40}$

- Lustre는 사용 가능한 용량을 2 단위의 제곱으로 표시하므로 사용 가능한 공간은 TiB로 계산된다
- 0.99는 파일 시스템의 1% 오버 헤드를 고려한 요소이며, (어레이당 디스크 수)은 핫 스페어를 제외한 어레이 당 HDD 수
- 0.8은 데이터 드라이브인 RAID 6 (8 + 2) HDD의 80 %입니다 (RAID 볼륨의 나머지 20 %는 패리티 드라이브이며 사용 가능한 공간으로 고려되지 않음)
- 공식 $10^{12} / 2^{40}$ 의 마지막 요소는 사용 가능한 공간을 TB에서 TiB로 변환

1. 기초

병렬 파일 시스템 - Lustre

- 어레이 수: 4
- 어레이당 디스크 수: 10개 (스페어 제외)
- RAID6 (8+2) 구성
- HDD 크기: 20TB

Lustre 예상 가용 공간(TiB) $\approx 0.99 * \text{어레이 수} * (\text{어레이당 디스크 수}) * 0.8 * \text{HDD 크기(TB)} * 10^{12}/2^{40}$

요소	의미	이유
0.99	파일 시스템 메타데이터 오버헤드 (1% 손실)	Lustre는 inode, 저널 등 파일 시스템 구조물 유지
어레이 수	디스크 어레이(스토리지 셀프) 개수	총 스토리지 확장도
어레이당 디스크 수	스페어(예비 디스크) 제외	RAID 그룹에 실제 사용되는 디스크만 포함
0.8	RAID6 사용 가능 비율	RAID6 (8+2) 구성을 가정하면 2개의 패리티로 80%만 데이터 저장
HDD 크기 (TB)	디스크 1개 용량	TB 단위로 입력
$10^{12}/2^{40}$	TB \rightarrow TiB 변환	Lustre는 TiB(2의 제곱 기준)로 관리

$$\begin{aligned}
 \text{가용 공간 (TiB)} &= 0.99 \times 4 \times 10 \times 0.8 \times 20 \times 10^{12}/2^{40} \\
 &= 0.99 \times 4 \times 10 \times 0.8 \times 20 \times 0.9095 \\
 &\approx 0.99 \times 4 \times 10 \times 0.8 \times 20 \times 0.9095 \\
 &\approx 576\text{TiB}
 \end{aligned}$$

1. 기초

병렬 파일 시스템 - Lustre

- 목표 가용량 : 576TiB
- 어레이당 디스크 수: 10개 (스페어 제외)
- RAID6 (8+2) 구성
- HDD 크기: 20TB

$$\text{어레이수} \approx \text{목표 가용 용량 (TiB)} / \{0.99 \times (\text{어레이당 디스크 수}) \times 0.8 \times \text{HDD 크기 (TB)} \times (10^{12} / 2^{40})\}$$

$$\begin{aligned} \text{어레이수} &= 576.2592 \text{ TiB} / (0.99 * 10 * 0.8 * 20 * 0.9095) \\ &\approx 576.2592 \text{ TiB} / 143.12 \\ &\approx 4.02 \end{aligned}$$

1. 기초

병렬 파일 시스템 - Lustre

Lustre 예상 가용 공간(TiB) $\approx 0.99 * \text{어레이 수} * (\text{어레이당 디스크 수}) * 0.8 * \text{HDD 크기(TB)} * 10^{12}/2^{40}$

TB	어레이	디스크 수	가용용량	TB	어레이	디스크 수	가용용량
4	1	80	230.5023	4	2	80	461.0047
8	1	80	461.0047	8	2	80	922.0093
10	1	80	576.2558	10	2	80	1152.512
12	1	80	691.507	12	2	80	1383.014
최대읽기 성능		~5.6GB/s		최대읽기 성능		~11.3GB/s	
최대 쓰기 성능		~5.3GB/s		최대 쓰기 성능		~10.6GB/s	

1. 기초

병렬 파일 시스템 – Lustre – RAID Level 6

이유	설명
1. 고가용성	Lustre는 HPC/병렬 IO 환경이라 대량의 HDD를 사용 HDD는 용량이 크고 실패 확률이 높기 때문에 이중 패리티(2개 디스크 장애 허용)가 가능한 RAID6가 적합
2. 큰 스토리지 풀 효율	RAID6는 N-2의 데이터 디스크를 사용할 수 있어, 큰 스토리지 풀을 만들 때 RAID5보다 장애 허용성이 높고, RAID10보다 저장 효율이 높다
3. 재빌드 시간 대비 안정성	HDD 용량이 수십 TB로 커지면서 RAID5의 단일 패리티는 디스크 장애 시 재빌드 중 추가 장애가 발생할 위험이 큼 RAID6는 두 번째 패리티 덕분에 이를 방지
4. 비용 효율성	Lustre는 대용량 데이터 아카이빙·병렬 처리 목적이므로 RAID10의 높은 미러링 비용 대신 RAID6의 공간 효율성을 선호
5. Hot Spare 연동	대형 Lustre 환경에선 Hot Spare 디스크를 두고 RAID6 재빌드 시 빠르게 복구할 수 있도록 설계

1. 기초

병렬 파일 시스템 – Lustre – RAID Level 권장

RAID Level	권장 여부	주요 특징
RAID0	× (권장하지 않음)	고속 IO는 가능하나 장애 발생 시 데이터 손실. 메타데이터 서버에 극히 일부 사용.
RAID1/10	제한적 사용	Lustre MDT(Metadata Target)에 주로 사용. 메타데이터는 용량이 작고 무결성이 중요해서 RAID1/10으로 미러링.
RAID5	(거의 사용하지 않음)	디스크 대용량화로 재빌드 위험성이 큼. 싱글 패리티로 안전성이 부족함.
RAID6	적극 권장	OST(Object Storage Target)에 가장 많이 사용. 2중 패리티로 대용량 HDD에 적합.
RAIDZ (ZFS)	가능	Lustre + ZFS 환경에서는 RAIDZ2(2중 패리티)나 RAIDZ3(3중 패리티)도 사용.
Erasure Coding	일부 연구적용	Object Storage에서 HDD 대신 Ceph 등 EC로 확장하는 시도도 있으나 Lustre에서는 표준은 아님.

Erasure Coding(이레이저 코딩) 은 요즘 대용량 분산 스토리지(특히 클라우드 Object Storage)에서 자주 등장하는 데이터 무결성 보장 기술
Ceph, MinIO, Hadoop, Azure Blob Storage, Google Cloud Storage
S3 호환 오브젝트 스토리지 솔루션

1. 기초

병렬 파일 시스템 – Lustre – MDT

MDT는 파일/디렉토리 이름, 퍼미션, 속성, 레이아웃 정보(inode) 등을 저장
실제 데이터(파일 내용)는 OST(Object Storage Target)에 저장되고, MDT는 파일 시스템의 ‘주소록’ 역할을 수행
용량 설계가 부족하면 메타데이터가 꽉 차서 전체 파일시스템 확장성이 막힘

MDT 용량 ~ 총 예상 파일 수 × inode당 평균 메타데이터 크기

요소	설명	예시 값
총 파일 수	최대 파일/디렉토리 수	예: 10억 개
inode당 평균 메타데이터 크기	Lustre 권장 평균: 4KB ~ 8KB	디렉토리 구조 복잡할수록 커짐
예비 공간	파일시스템 reserve + 성능 여유	일반적으로 10%~20% 추가

1. 기초

병렬 파일 시스템 – Lustre – MDT 계산 예

MDT 예상 용량 (GB)=(총 파일 수×inode 평균 크기 (Byte) / 2³⁰)×(여유율 %)

- 총 파일 수 = 1,000,000,000 (10억)
- inode 평균 크기 = 4KB
- 여유율 = 20%

1,000,000,000×4,096Byte=4,096,000,000,000Byte≈3.72TB

여유율 포함:

3.72TB×1.2≈4.5TB

3.72TB×1.2≈4.5TB

환경	파일 평균 크기	권장 inode 크기	MDT 용량 비중
대형 HPC Scratch	대형 바이너리/시뮬레이션 결과	4KB	낮음
수많은 작은 파일	로그, 소스 코드 저장소	8KB+	높음
일반 연구 클러스터	혼합	4KB~8KB	중간

1. 기초

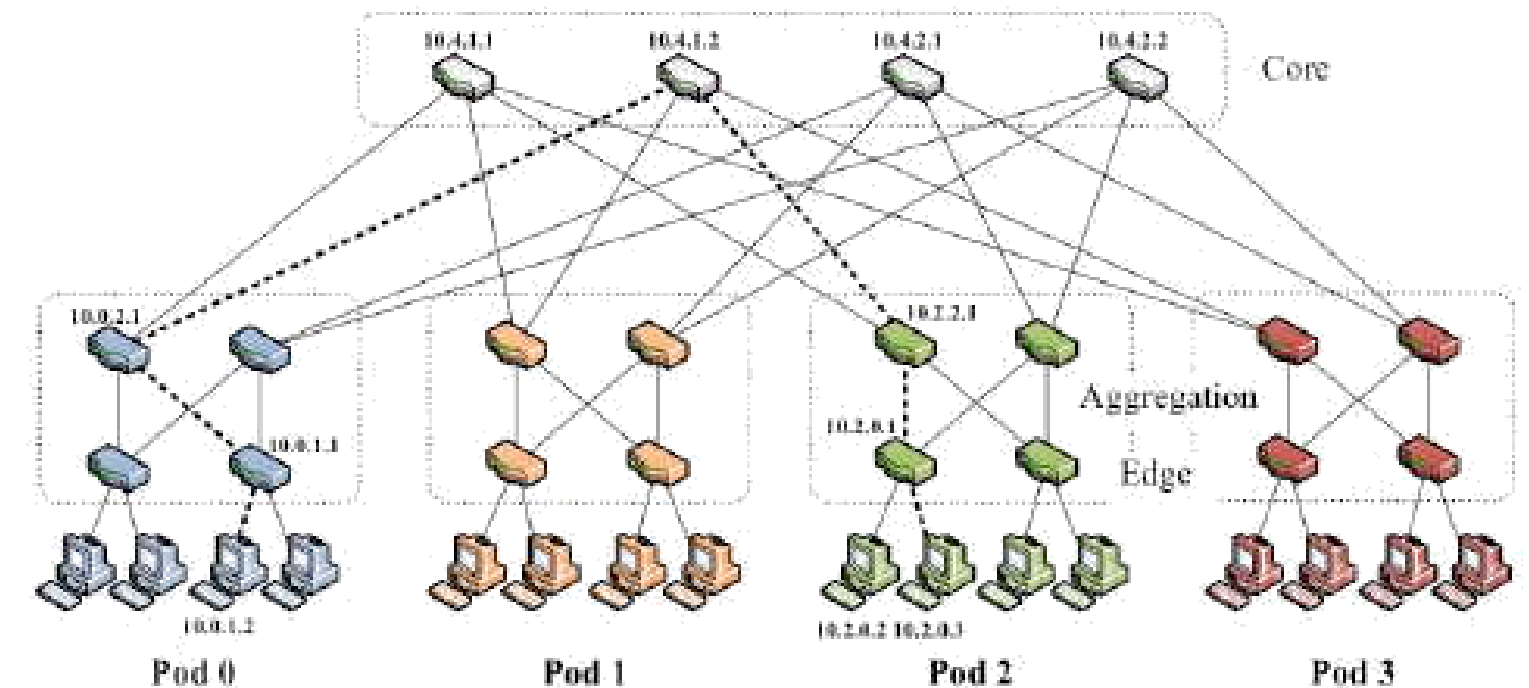
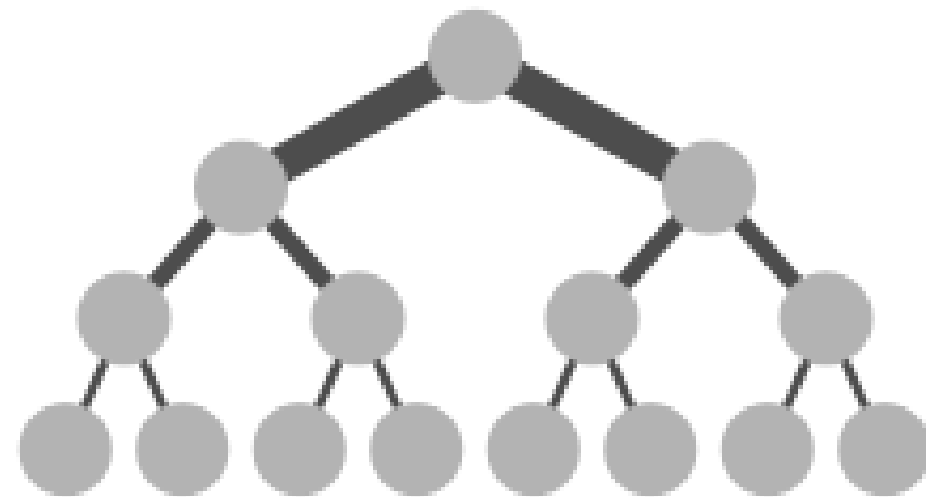
Network

	용도 및 기능	device
IPMI 관리 네트워크	<ul style="list-style-type: none"> IPMI(Intelligent Platform Management Interface) 네트워크, 하드웨어를 원격으로 관리 서버의 전원 제어 원격 전원 켜기, 전원 끄기 및 서버 재부팅과 같은 작업을 수행 하드웨어 모니터링 IPMI를 사용하여 서버의 중요한 하드웨어 부분을 모두 모니터링. 팬 상태에서 팬 속도, CPU 온도 및 상태, 서버 하드웨어의 모든 측면에 대한 전원 공급 상태를 원격으로 액세스 및 변경. 	IPMI BMC
메니지먼트 네트워크	<ul style="list-style-type: none"> HPC 시스템의 라이선스 및 모니터링 HPC 메니지먼트 소프트웨어와 연동 하여 시스템 운영 상태 모니터링 라이선스 관리 HPC Application 라이선스 운영, 모니터링 SW 모니터링 서버의 SW 운영, 모니터링. CPU 자원 상태, Memory 자원 상태, Network 자원 상태 등 소프트웨어 모든 측면에 대한 모니터링 	TCP/IP
계산 네트워크	<ul style="list-style-type: none"> HPC 계산을 위한 전용 네트워크 	Infnniband

1. 기초

Network

일반적인 tree 데이터 구조에서는 각각의 branch 가 자신의 depth 에 상관없이 같은 굵기를 가지게 된다. 하지만 fat tree 에서는 같이 상위 계층에 있는 branch 일수록 더 fatter (두꺼운) 성질을 가지게 되며, depth 가 커질수록 branch 는 가늘어 지게 된다.



Simple fat-tree topology. Using the two-level routing tables packets from source 10.0.1.2 to destination 10.2.0.3 would take the dashed path.

1. 기초

Network

항목	Tree	Fat Tree
구조	루트 → 중간 스위치 → 리프(서버)로 내려가는 전통적 계층형 구조	Tree와 유사하지만 상위 계층으로 갈수록 링크/스위치 대역폭을 '굵게(Fat)' 설계
대역폭	하위(Leaf)에서 상위(Core)로 갈수록 링크 수가 줄어 병목 발생	상위 계층 대역폭을 늘려 상위 병목 완화
트래픽 흐름	상위 코어 스위치에 통신이 집중되어 혼잡해지기 쉬움	모든 노드 간 경로에 충분한 대역폭을 제공하여 All-to-All 통신 최적화
확장성	수직적 계층 추가로 확장하면 코어 스위치가 병목	계층별로 스위치/링크 수를 늘려 수평 확장 가능
예시 사용처	소규모 LAN, 기업용 일반 네트워크	HPC 클러스터, 대형 데이터센터(Clos/Fat Tree), 클라우드 인프라

1. 기초

Network : 왜 HPC에서 Fat Tree를 사용하나?

HPC 환경은 노드 간 통신이 All-to-All 또는 Many-to-Many로 발생하므로 상위 계층에서의 병목이 HPC 전체 성능에 큰 영향을 미칩

Fat Tree가 적합한 이유

1. Non-blocking bisection bandwidth

네트워크를 절반으로 잘라도 절반 간의 통신에 충분한 대역폭을 유지할 수 있어 대규모 병렬 계산에 필수적

2. 스케일아웃에 최적

노드 수가 늘어나면 상위 계층 스위치 수와 링크를 늘려서 병목 없이 확장 가능

3. Top500 표준 설계

InfiniBand, Omni-Path, 400G Ethernet 같은 HPC 네트워크는 거의 Fat Tree(Clos Network)를 기반으로 설계됨

4. 다양한 트래픽 패턴에 강함

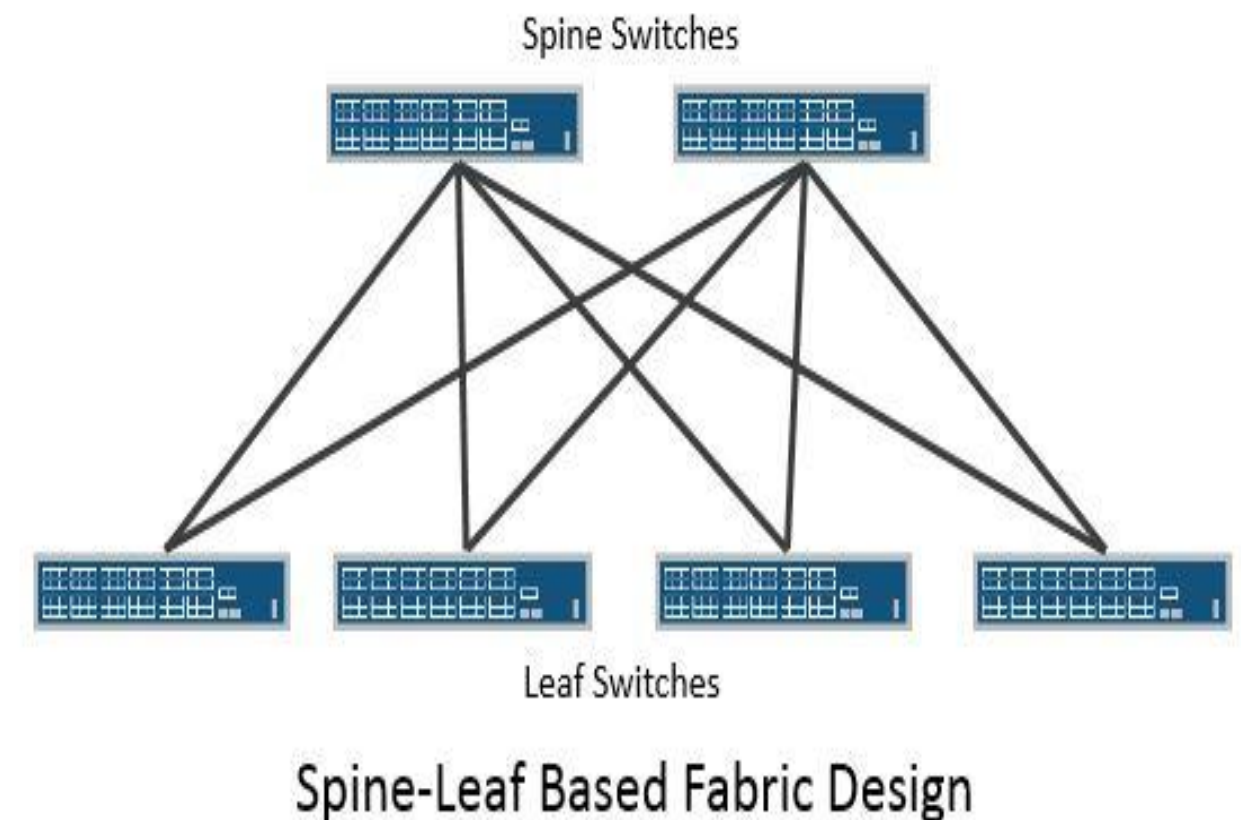
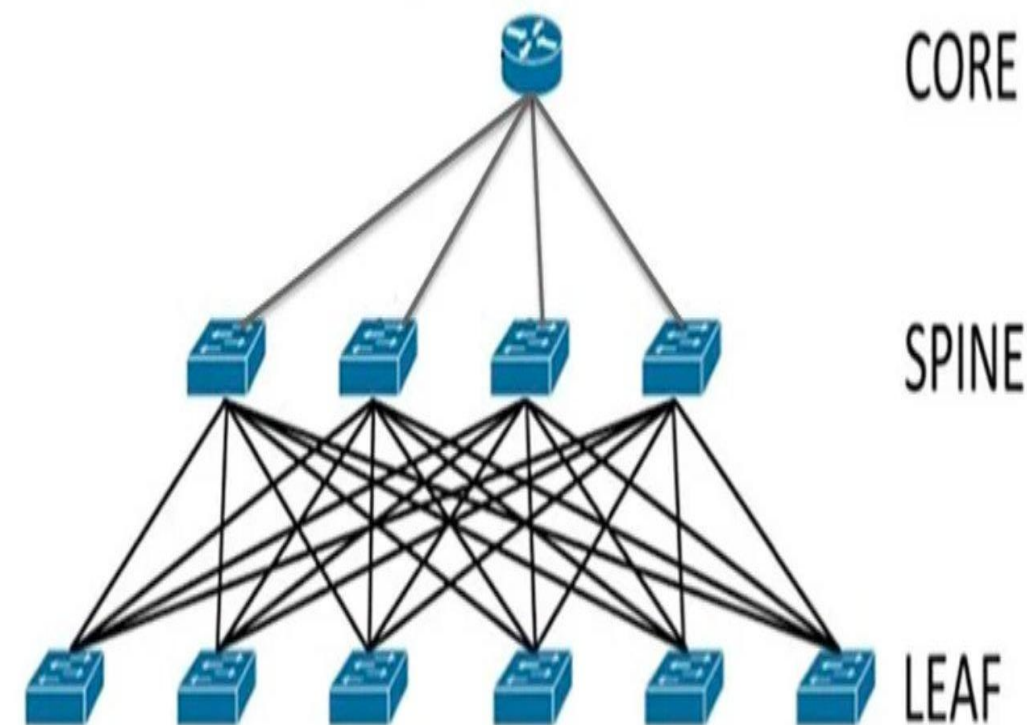
불규칙 통신(MPI Allreduce 등)에도 안정적인 대역폭 제공

1. 기초

Network : Fat Tree 배선 예시와 레벨별 스위치 용량(포트 수, 대역폭) 계산 방법

기본 용어

- Leaf Switch: Compute 노드가 직접 연결되는 스위치 (Top of Rack, TOR)
- Spine Switch(Core/Agg): Leaf 스위치를 상호 연결해 트래픽을 상위로 전달하는 스위치
- Blocking Factor: Non-blocking 설계라면 1:1, 일부 Blocking 설계는 2:1 등으로 설계 가능



1. 기초

Network : Fat Tree 배선 예시와 레벨별 스위치 용량(포트 수, 대역폭) 계산 방법

예시: 64개 노드를 QDR InfiniBand로 Non-blocking Fat Tree 구성

항목	값
노드 수	64
Leaf 스위치 포트 수	36포트 (Mellanox 4036Q)
코어/스파인 스위치	Leaf 스위치와 동일한 모델 사용 가능
Blocking Factor	1:1 (Non-blocking)

1단계: Leaf 스위치

- Leaf 스위치는 절반은 노드 연결, 절반은 Spine 연결로 사용. 예: 36포트 스위치 → 18포트는 Compute 노드 연결, 18포트는 Spine 연결.

- 필요한 Leaf 스위치 수:

$$\begin{aligned} \text{leaf 스위치 수} &= \text{전체 노드 수} \div (\text{Leaf 스위치의 노드 연결 포트 수}) \\ &= 64 \div 18 \approx 3.56 \rightarrow 4\text{대} \end{aligned}$$

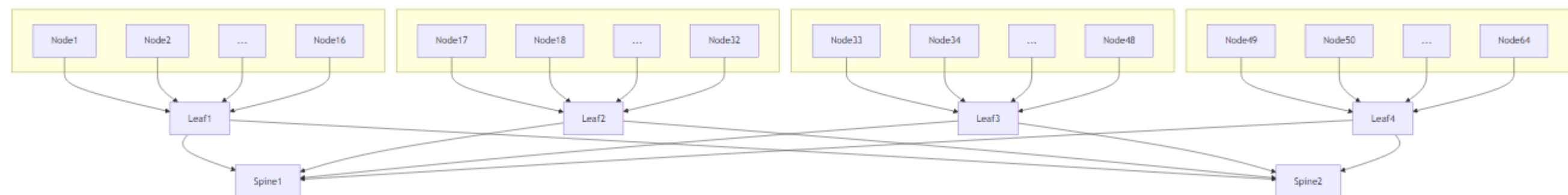
2단계: Spine 스위치

- Leaf 스위치는 절반은 노드 연결, 절반은 Spine 연결로 사용.
- Leaf 스위치 총 Uplink 포트 수:

$$4 \text{ Leaf 스위치} \times 18 \text{ Uplink 포트} = 72$$

- Spine 스위치 1대당 36포트로 받는다면,

$$\begin{aligned} \text{Spine 스위치 수} &= \text{총 Uplink} \div \text{Spine 스위치 포트 수} \\ &= 72 \div 36 = 2\text{대} \end{aligned}$$



1. 기초

Network : Clos 네트워크란?

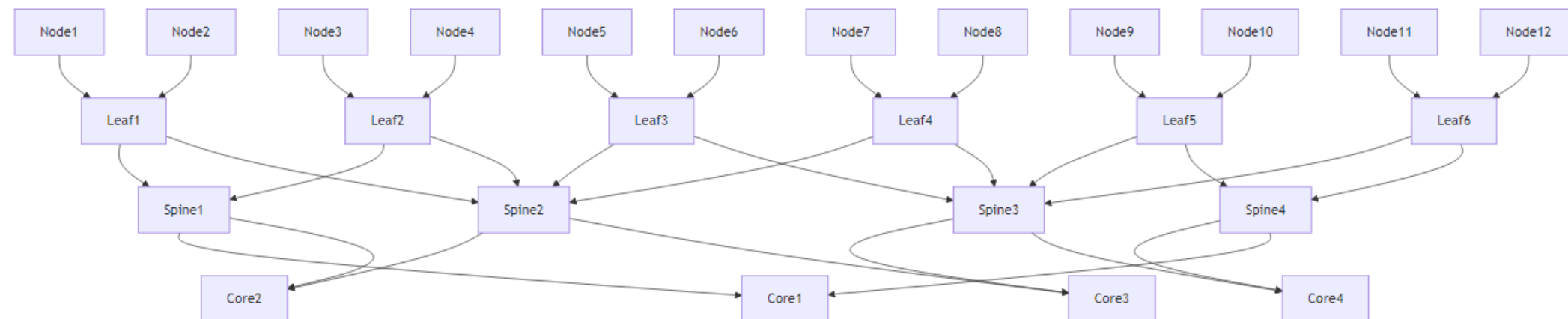
Clos(클로즈) 네트워크는 1950년대 전자 교환기 설계자 Charles Clos가 제안한 멀티스테이지 스위칭 네트워크 구조
핵심 아이디어는 스위치를 여러 계층(Stage) 으로 쌓아서 Non-blocking 또는 Near Non-blocking 성능을 만들 수 있음
Clos는 Fat Tree의 원형이며, Fat Tree는 Clos 설계를 HPC/데이터센터용으로 발전시킨 버전

3-Tier Clos (3단계 Clos) 기본 구조3단계:

- Leaf (Access) 레벨 → Compute 노드/서버가 직접 연결됨
- Spine (Aggregation) 레벨 → 여러 Leaf를 상호 연결
- Core (Super-Spine) 레벨 → 대규모 시스템에서 여러 Spine Pod(Clos Fabric)을 상호 연결

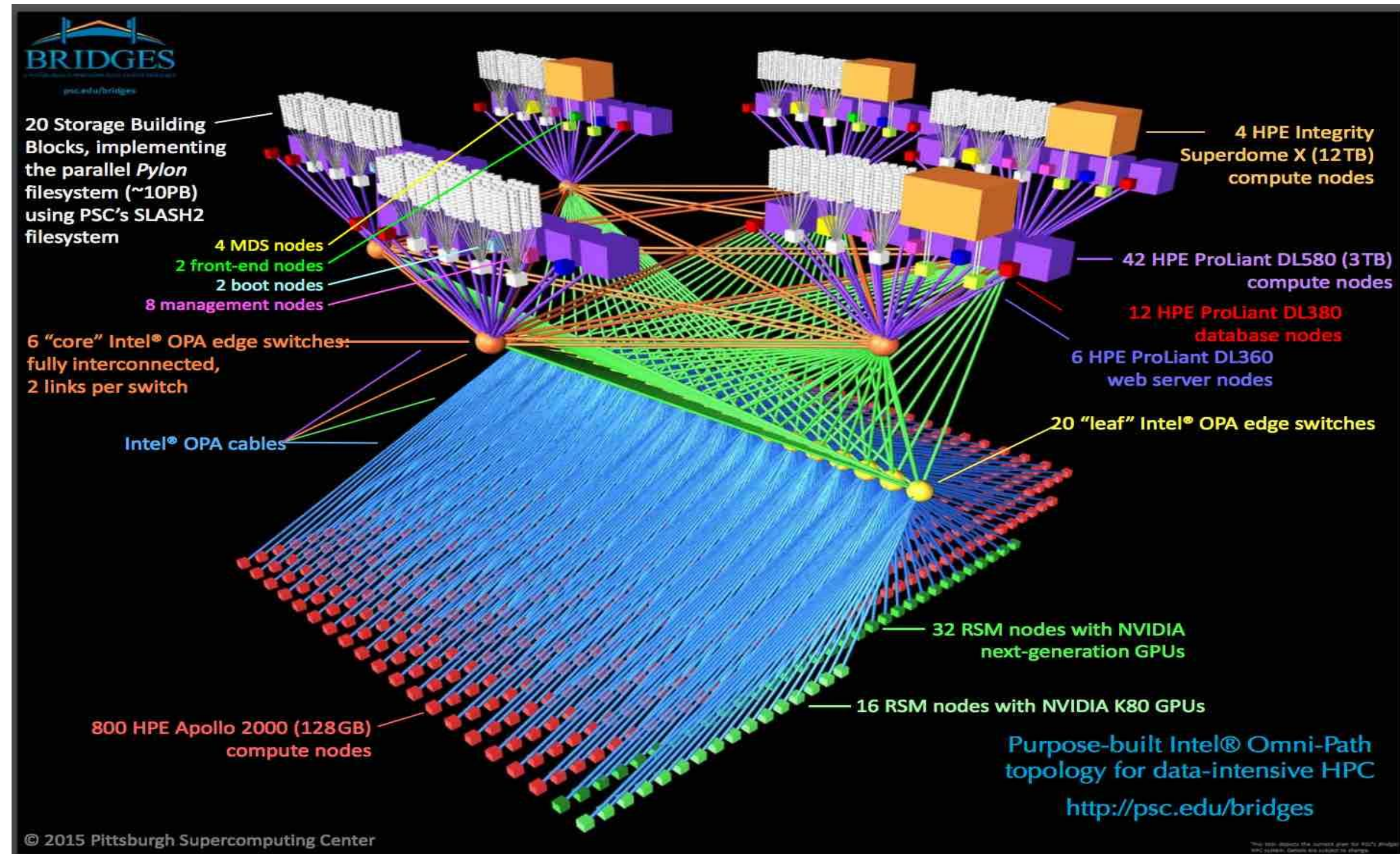
3-Tier Clos가 필요한 이유

- 소규모 클러스터라면 2-Tier(Leaf-Spine)로 충분
- 수천~수만 노드로 확장하려면 2-Tier만으로는 Spine 스위치 포트 수가 한계에 부딪힘
- Spine-계층도 여러 그룹(Pod)으로 나누고 Core 레벨을 추가해서 Pod 간 대역폭을 연결해줌
- 이를 통해 Full bisection bandwidth, Non-blocking Fabric을 유지하면서 대규모 확장이 가능!



1. 기초

Network :



1. 기초

공조

- 소비전력에 대비하여 발열
- 클러스터 시스템의 발열량은 소비전력으로 계산
- BTU(British Thermal Unit- 영국 온도단위)

$$\text{전체 소비전력(W)} \times 3.41214 = \text{BTU/hr}$$

$$\text{전체 소비전력(W)} \times 0.860 = \text{kcal/h}$$

$$1 \text{ RT} = 12,000 \text{ BTU/hr 또는 } 1 \text{ RT} \approx 3.517 \text{ kW}$$

1. 기초

공조 계산 예시

가정

소비전력: 5,000 W (5 kW)

이 전력을 모두 냉방에 사용한다고 가정

W → BTU/hr 변환

$$5,000 \text{ W} \times 3.41214 = 17,060.7 \text{ BTU/hr}$$

BTU/hr → RT(냉동톤) 변환

냉동톤(RT, Refrigeration Ton)의 기준:

$$1 \text{ RT} = 12,000 \text{ BTU/hr}$$

따라서,

$$17,060.7 \text{ BTU/hr} \div 12,000 = 1.42 \text{ RT}$$

W → kcal/hr 변환 (참고)

$$5,000 \text{ W} \times 0.860 = 4,300 \text{ kcal/hr}$$

1. 기초

공조 계산 : 룰 오브섬(Rule of Thumb) 계산 방식 응용

- 총 열부하 =

방 면적 BTU + Windows BTU + 총 거주자 BTU + 장비 BTU + 조명 BTU

① 면적 BTU = 길이 (m) x 너비 (m) x 337

② 총 창문 BTU = 북쪽창 + 남쪽창

③ 총 거주자 BTU = 거주자 수 x 400

④ 장비 BTU = 모든 장비의 총 전력량 x 3.5 (~ 3.41214

⑤ 조명 BTU = 모든 조명에 대한 총 전력 x 4.25

- 필요한 a / c 장치 수 = 총 열 부하 BTU / 냉각 용량 BTU

1. 기초

공조 계산 : 룰 오브섬(Rule of Thumb) 계산 방식 응용

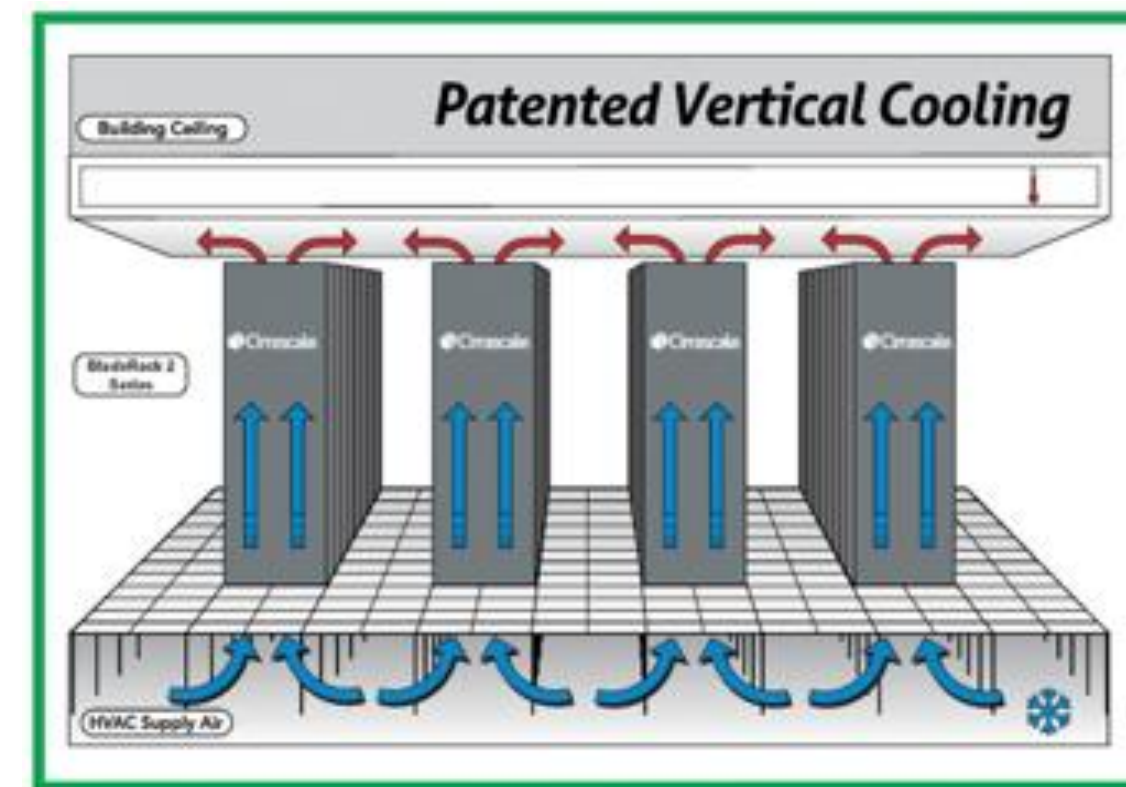
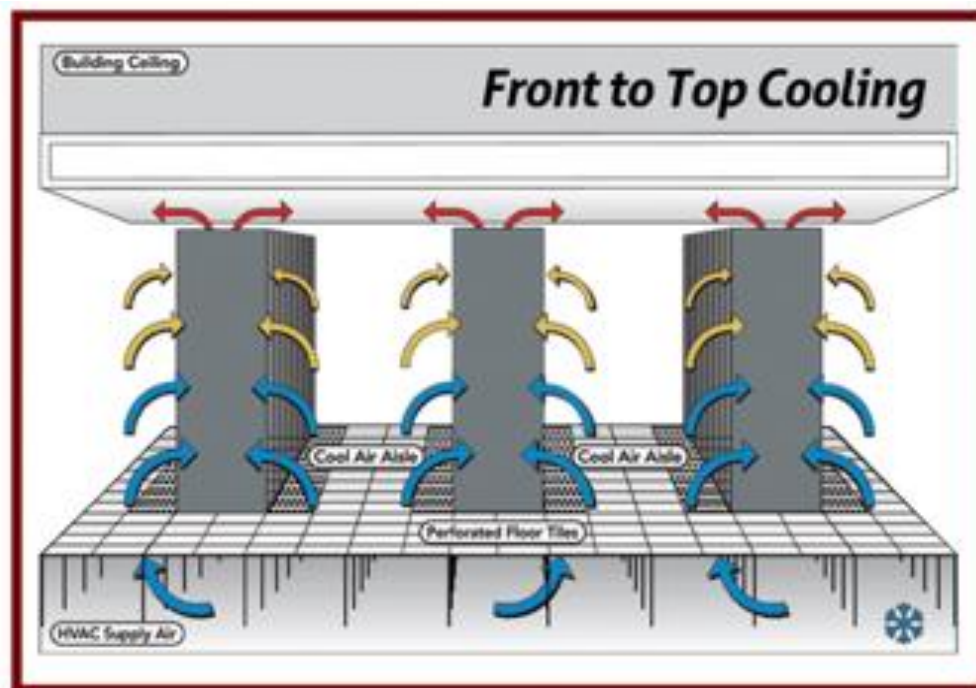
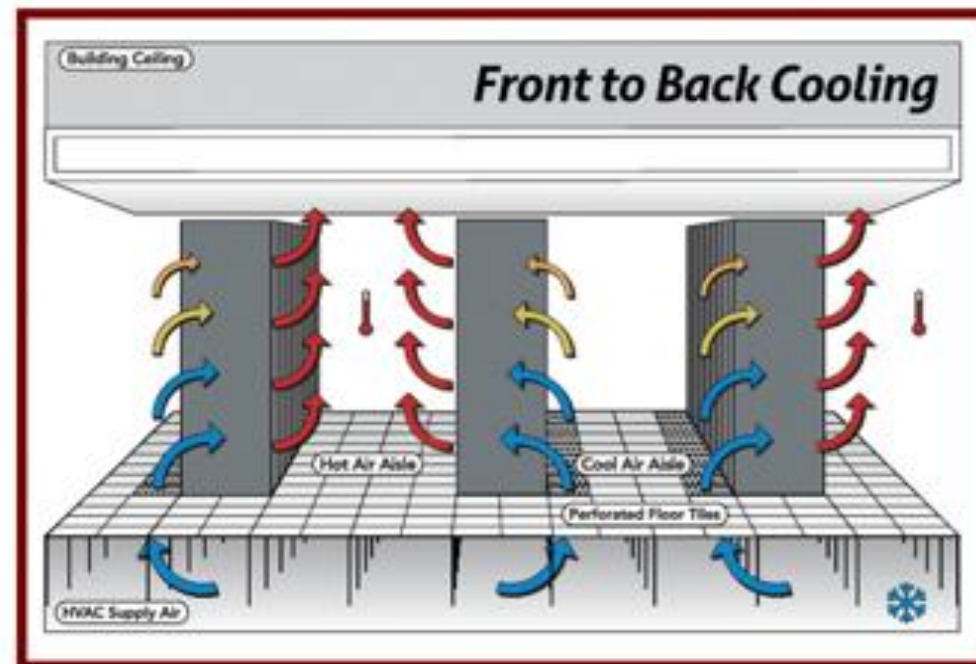
- 총 열부하 =

방 면적 BTU + Windows BTU + 총 거주자 BTU + 장비 BTU + 조명 BTU

항목	계산식	실무 적합성	참고
면적 BTU	길이(m) × 너비(m) × 337	보통 평방피트(ft²)로 25~35 BTU/ft²로 잡는데, 미터 기준으로 환산한 값으로 OK.	1m² ≈ 10.76 ft², 337 BTU/m²는 31.3 BTU/ft² 정도.
창문 BTU	북쪽+남쪽	실무에서도 방향별 일사 부하 따로 계산	보통 남향 창은 더 큰 값 사용
거주자 BTU	인원수 × 400	1인당 400 BTU/hr는 표준에 근접	250~450 BTU/hr 범위
장비 BTU	총전력(W) × 3.5	3.41214가 공식값, 3.5는 여유치 반영	실무에서 3.4~3.5 사용
조명 BTU	총전력(W) × 4.25	이 부분은 주의 필요	일반적으로 1W × 3.41214 BTU/hr가 표준, 4.25는 여유 반영 (발열 손실을 고려)

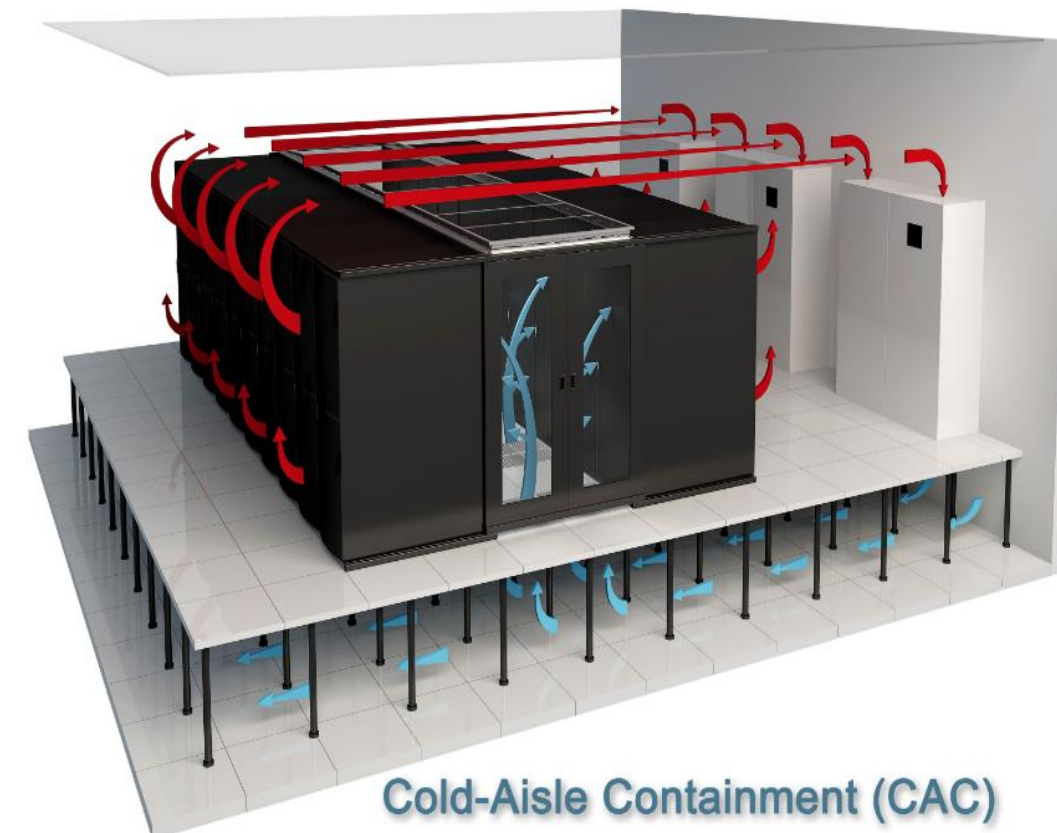
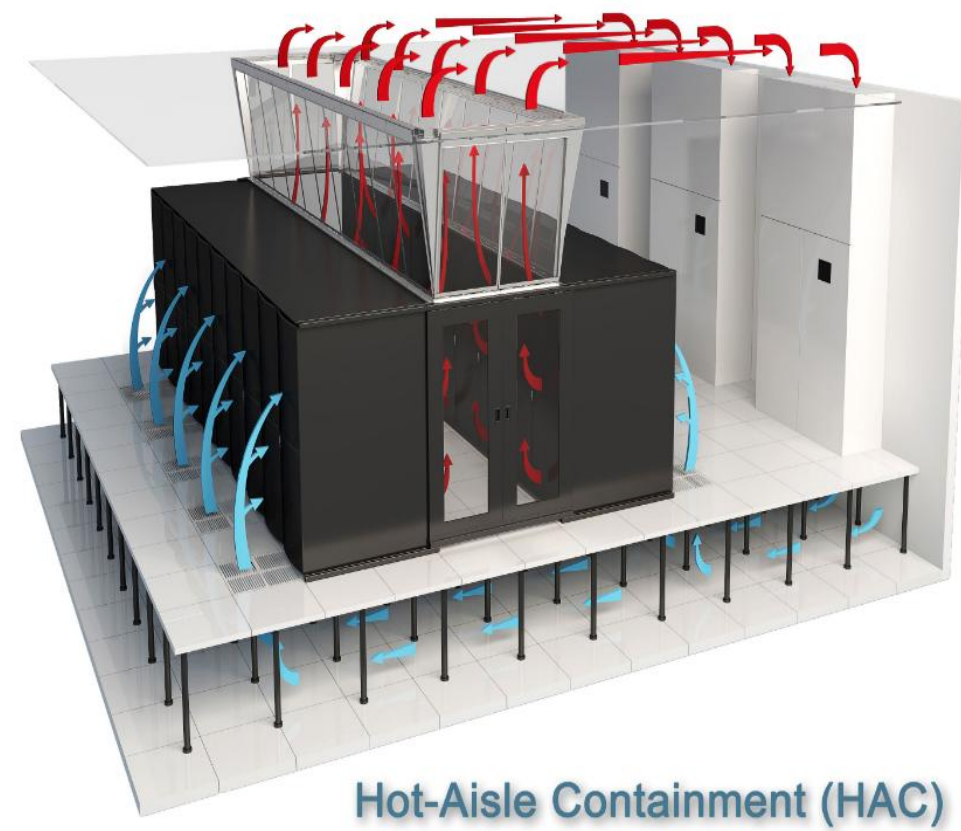
1. 기초

공조 계산 : Cooling



1. 기초

공조 계산 : Cooling



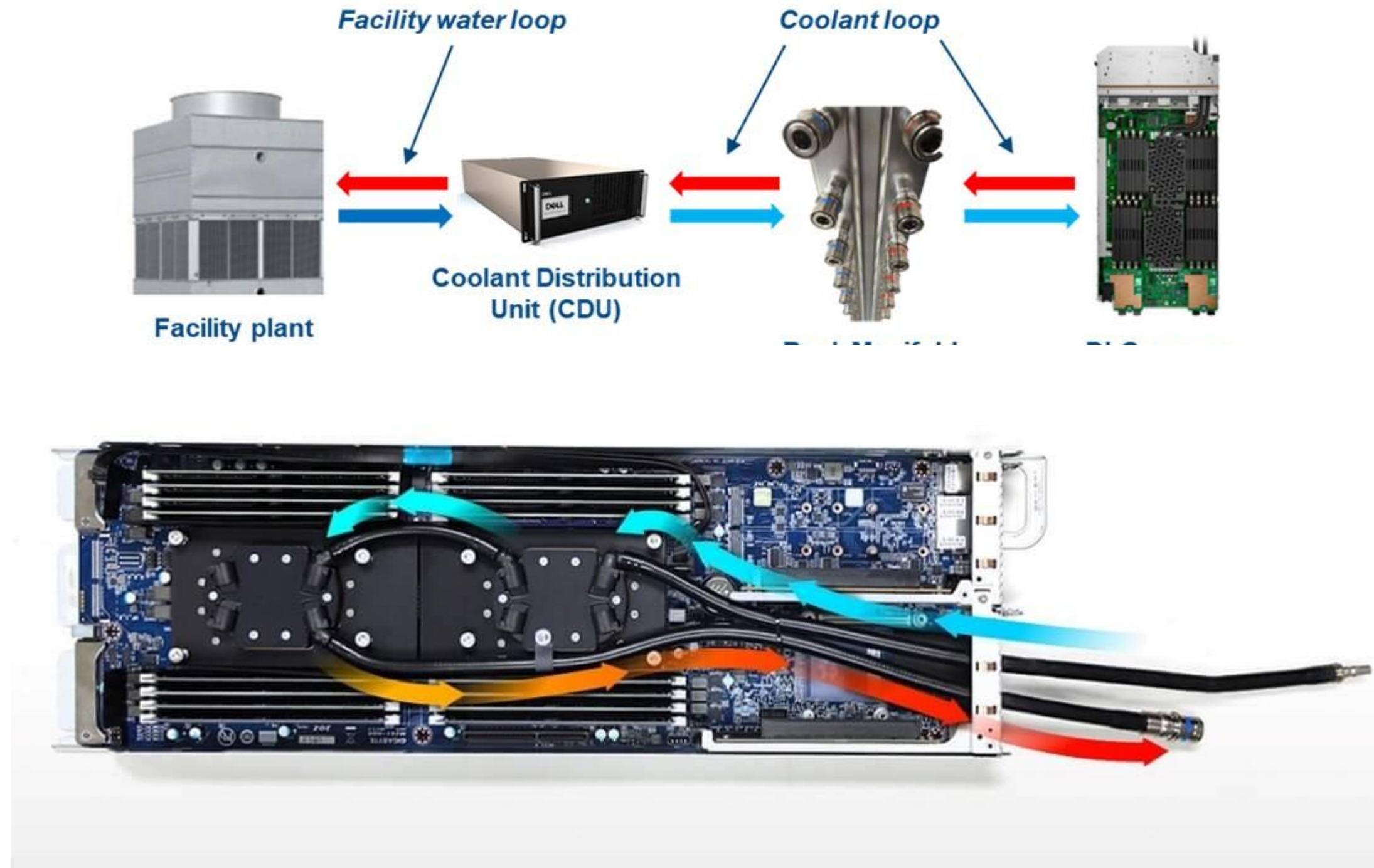
1. 기초

공조 계산 : Cooling

구분	Cold-Aisle Containment (CAC)	Hot-Aisle Containment (HAC)
정의	차가운 공기가 흐르는 통로(Cold Aisle)를 밀폐해 차가운 공기가 서버 인렛(흡입구)으로 집중되도록 함	뜨거운 공기가 흐르는 통로(Hot Aisle)를 밀폐해 뜨거운 공기가 섞이지 않고 리턴 플레넘으로 효율적으로 배출되도록 함
배치	랙의 인렛(전면)이 서로 마주보도록 하고 그 공간을 밀폐	랙의 아웃렛(후면)이 서로 마주보도록 하고 그 공간을 밀폐
혼합 방지 효과	차가운 공기가 유출될 수 있어 Hot Aisle로 일부 혼합될 가능성 있음	Hot Aisle를 완전히 밀폐하여 Hot/Cold 공기 혼합 방지 효과가 더 좋음
공조 효율(COP)	중간	상대적으로 더 높음
PUE (Power Usage Effectiveness):	약 1.5~1.6 수준	약 1.3~1.4 수준까지 가능
에너지 절감	20~30%	최대 30~40%까지 가능
구성 난이도	설치가 상대적으로 쉽고 비용도 적음	설계 및 리턴 덕트 설비 추가 등 복잡할 수 있음
인체 작업 환경	작업자가 Cold Aisle에서 작업	Hot Aisle 내부는 뜨거워서 점검/유지보수 작업이 쉽고 불편함

1. 기초

공조 계산 : Cooling

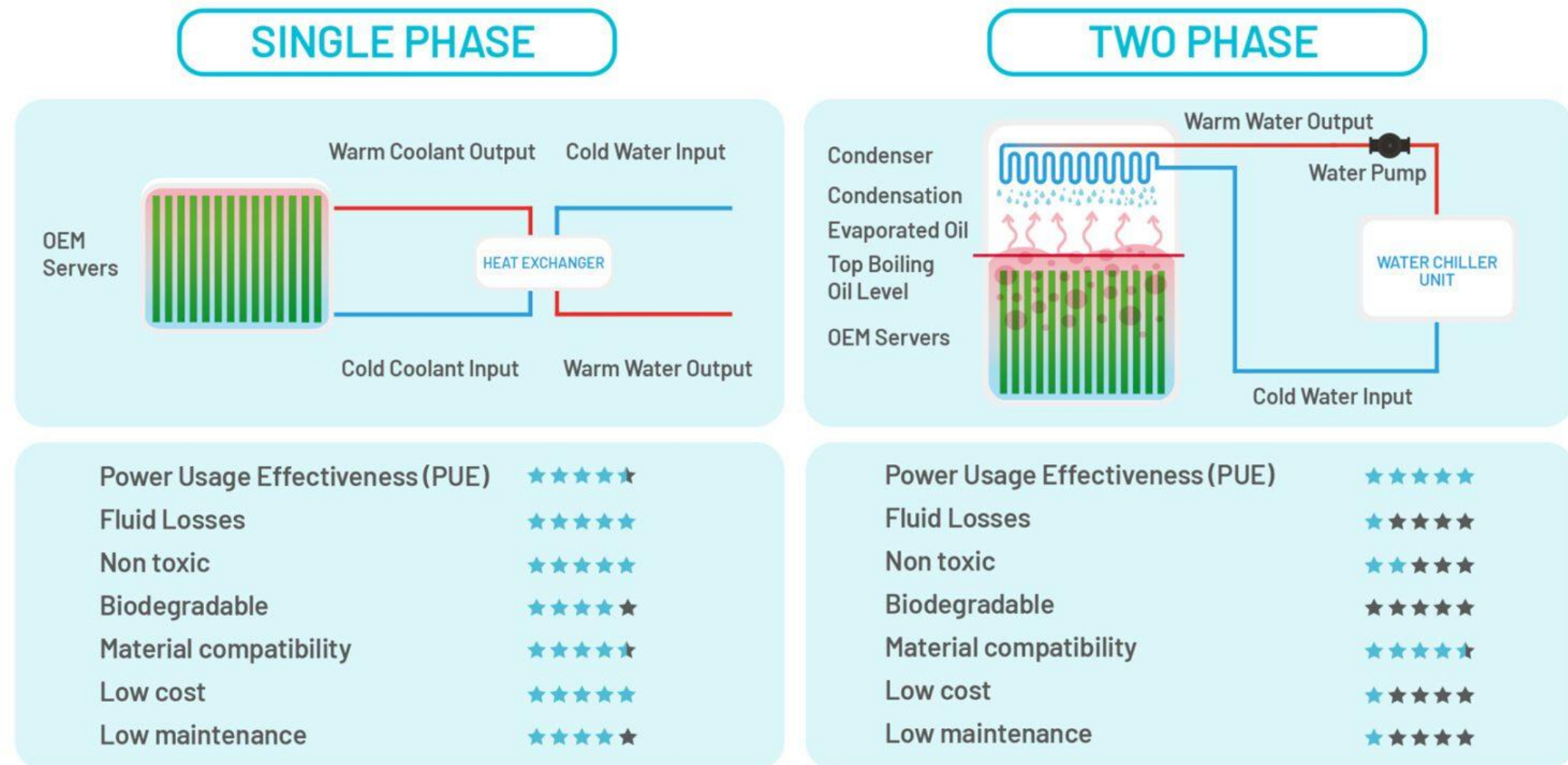


1. 기초

공조 계산 : Cooling

SINGLE-PHASE .V.S. TWO-PHASE IMMERSION COOLING WHICH ONE IS BEST FOR YOU?

There are two techniques which can be used with Immersion Cooling but how do they compare against one another? Let's take a look!



LEGEND

Measurements are made using a 1-5 star scale. 1 = Not very good, 5 = Very good

1. 기초

공조 계산 : PUE

PUE

Power Usage Effectiveness (PUE)

1. 기초

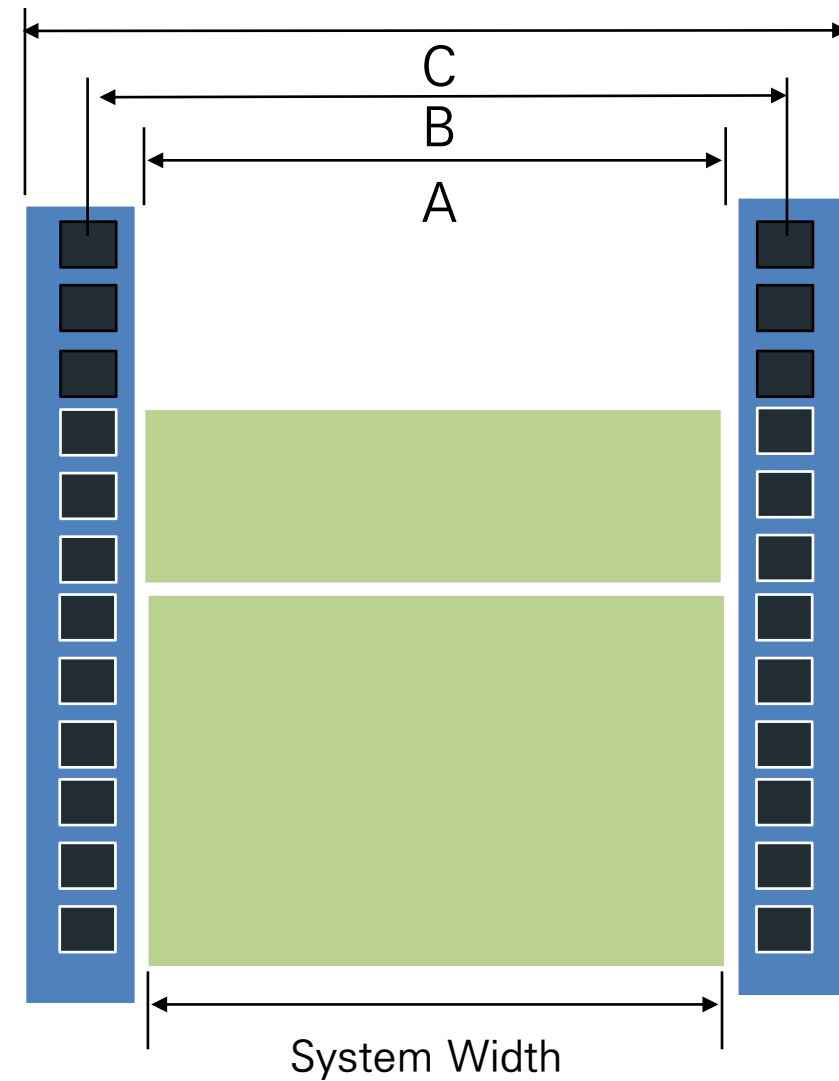
공조 계산 : PUE

평균 전력 사용 효율 지수
Power Usage Effectiveness (PUE)

$$\text{PUE} = \frac{\text{Total Datacenter Power(총 전력)}}{\text{Actual IT Power (IT 장비 전력)}}$$

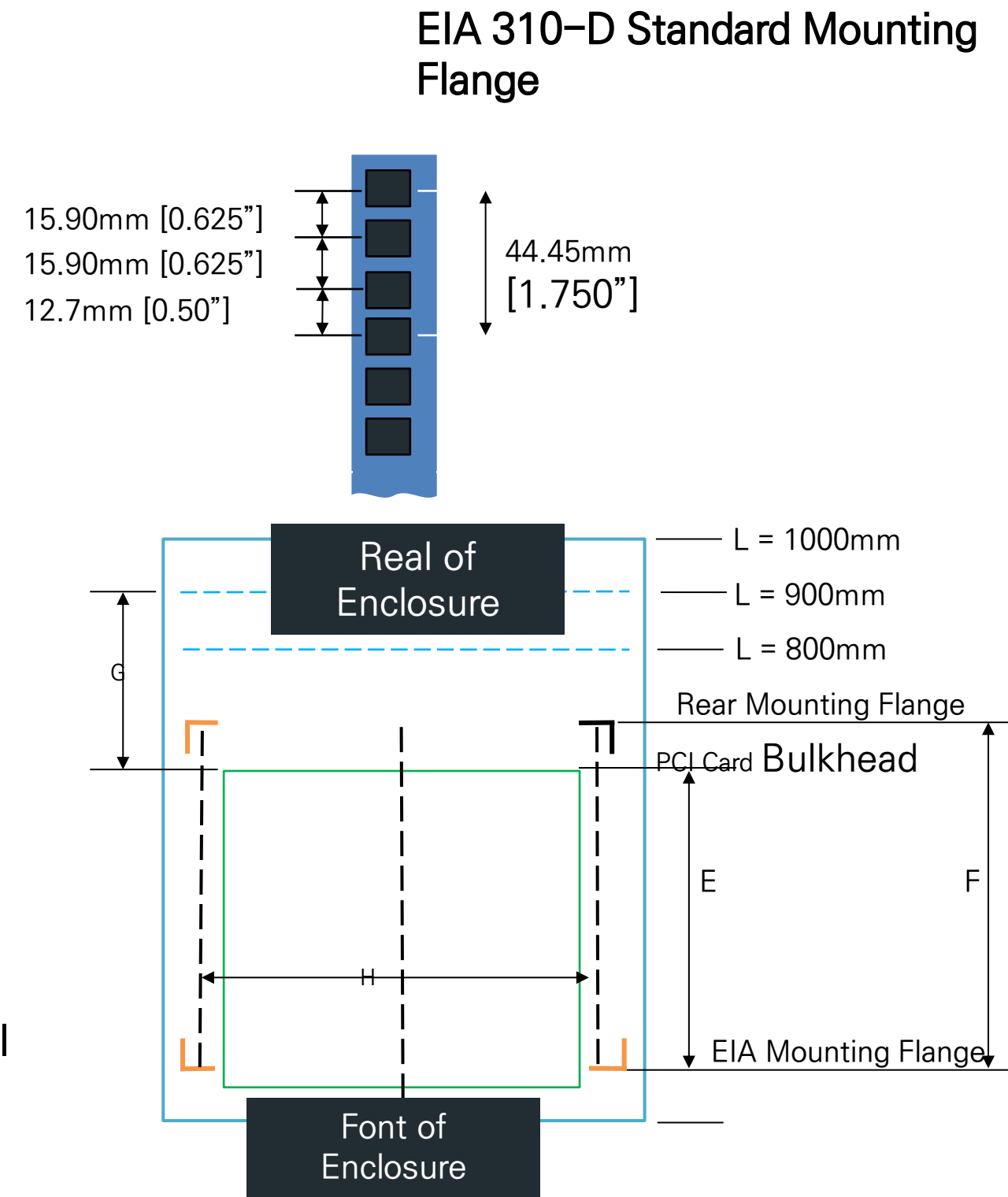
1. 기초

공조 계산 : RACK



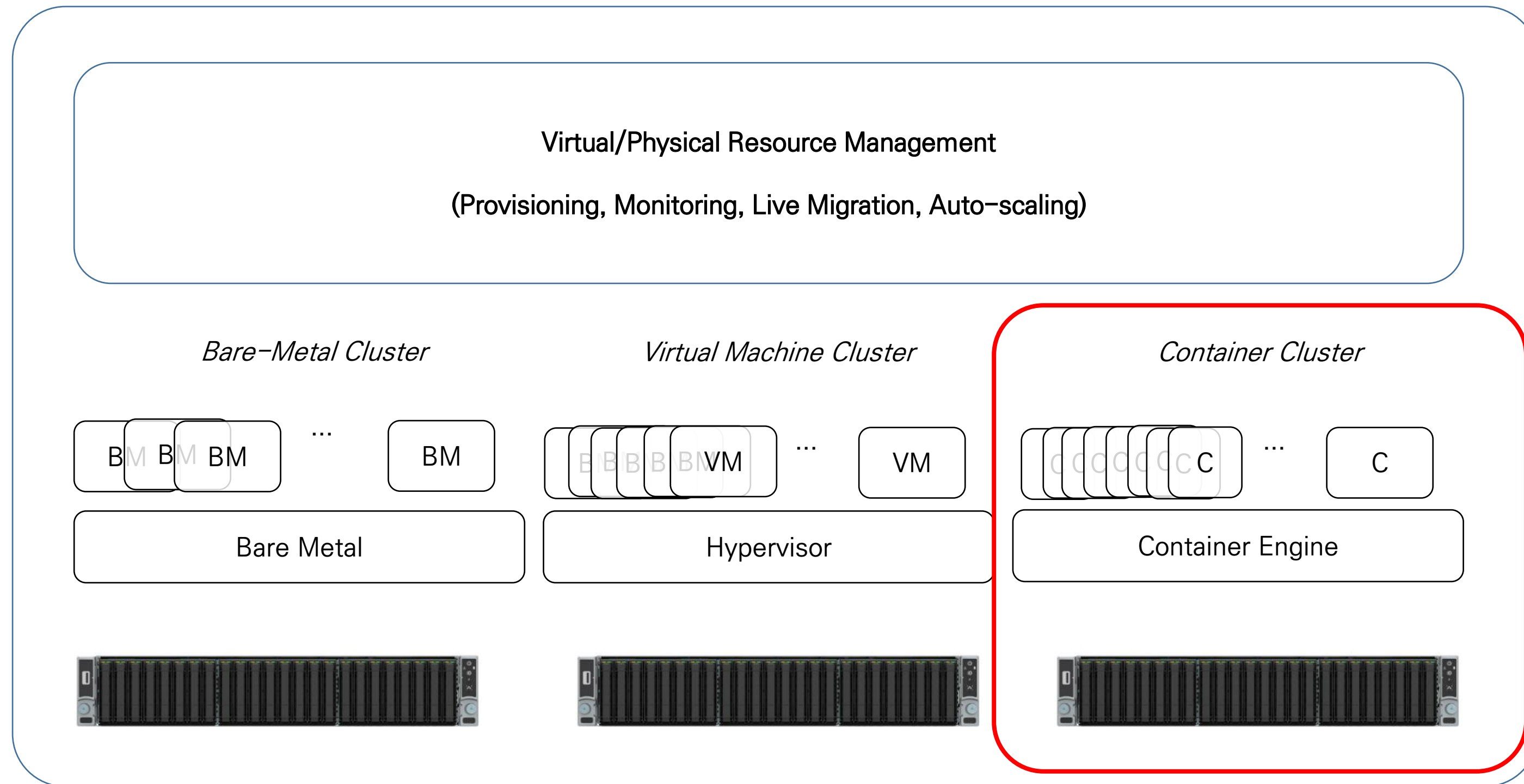
Notable Dimensions

Dimension A = 450mm [17.717"] min
Dimension B = 465mm [18.30"] nominal
Dimension C = 483.4mm [19.03"] min



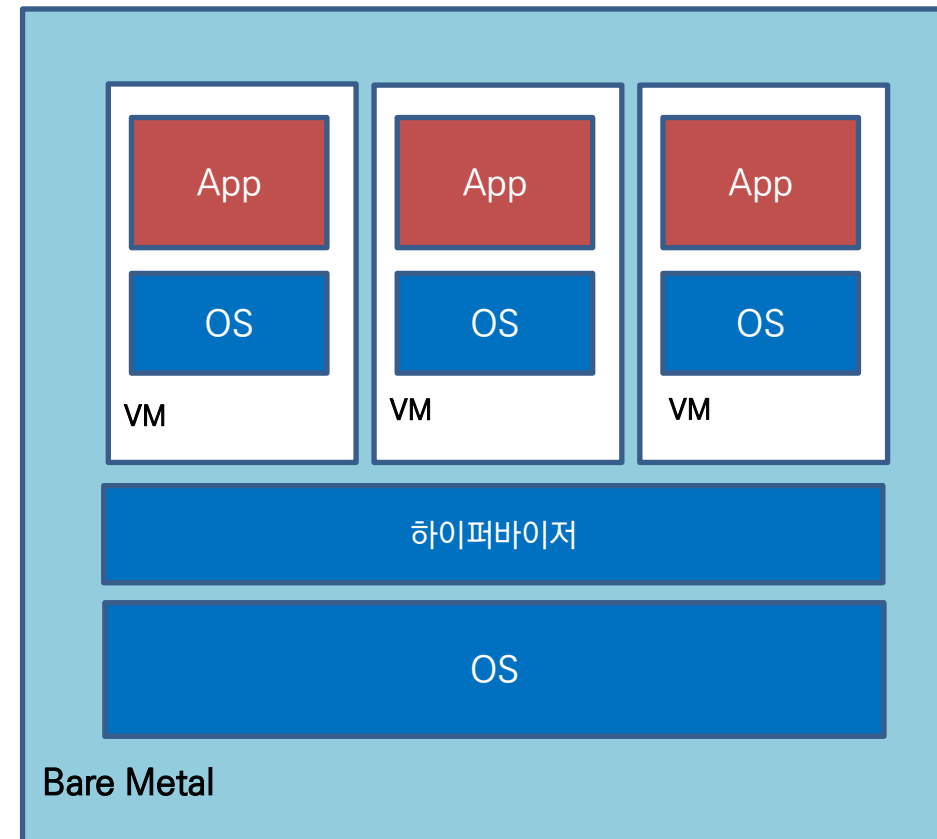
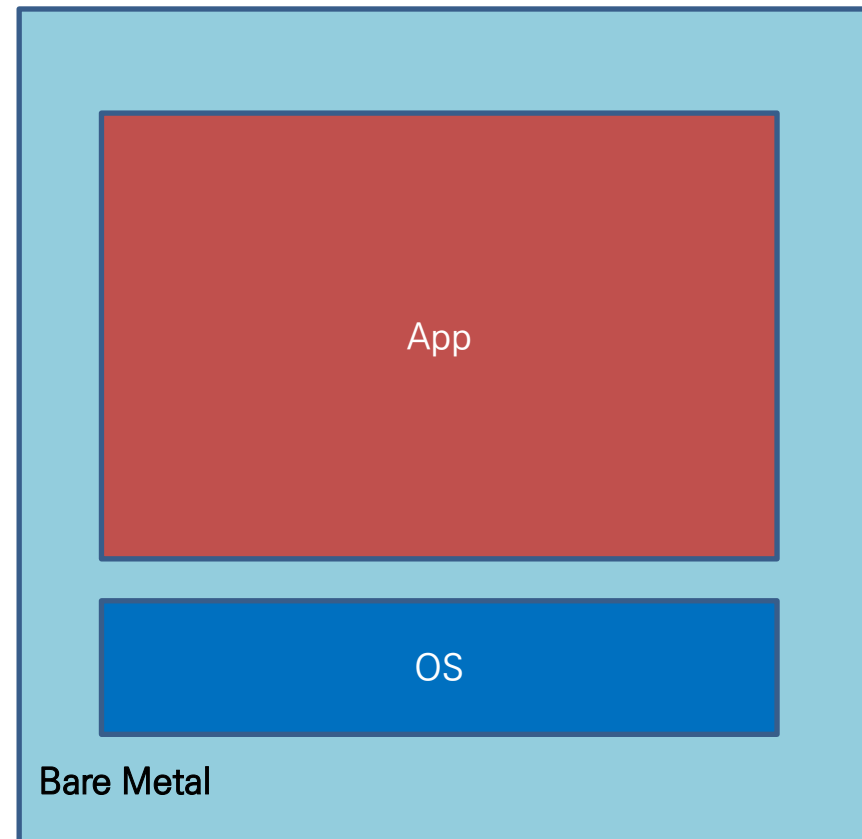
1. 기초

트렌드



1. 기초

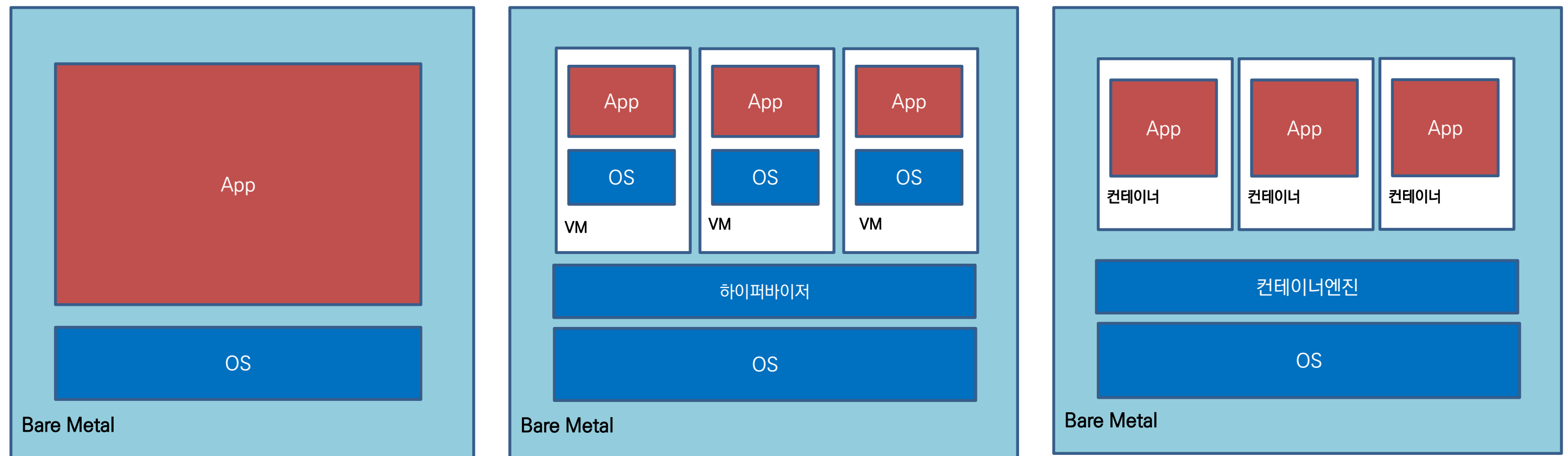
베어메탈/가상화/컨테이너



베어메탈	가상화	컨테이너
전통적인 클러스터	Bare Metal 자원을 가상화	프로세서화
스케줄러 slurm SGE PBS 등을 사용 하여 작업 관리	생성된 VM 단위로 스케줄러 사용	K8s 등을 사용 최근 기존 스케줄러와 연동
변화에 적용이 느림	필요에 따라 자유로운 구성 가능	필요에 따라 자유로운 구성 가능

1. 기초

베어메탈/가상화/컨테이너



	가상화	컨테이너
하이퍼바이저	O	X
Guest OS	O	X
커널 자원 분리	O	X
시작 및 종료 시간	느림	빠름
자원효율성	낮음	높음

1. 기초

HPC 컨테이너 장점과 단점

- 컨테이너화 혹은 운영체제 수준의 가상화는 하나의 호스트 머신에서 격리를 사용 하여 호스트 시스템 구성 설정과의 충돌 없이 특화된 새로운 시스템 환경을 구축할 수 있다는 장점
- HPC 분야에서는 연산의 규모가 크기(극한으로) 때문에 사용되는 기술들 사이의 미세한 성능 차이가 결국 큰 차이를 보일 수 있다.
- 한국컴퓨터정보학회논문지 Journal of The Korea Society of Computer and Information Vol. 25 No. 9, pp. 11–20, September 2020 <https://doi.org/10.9708/jksci.2020.25.09.011>
- 최근 HPC 컨테이너는 네이티브와 유사하거나 수행하는 어플리케이션에 따라 오히려 더 빠른 성능을 보임 (NPB : EP)
- 호스트 네트워크 사용 조건에서 Docker 컨테이너와 HPC 컨테이너들은 대부분의 벤치마크에 대해 비슷한 성능을 보임

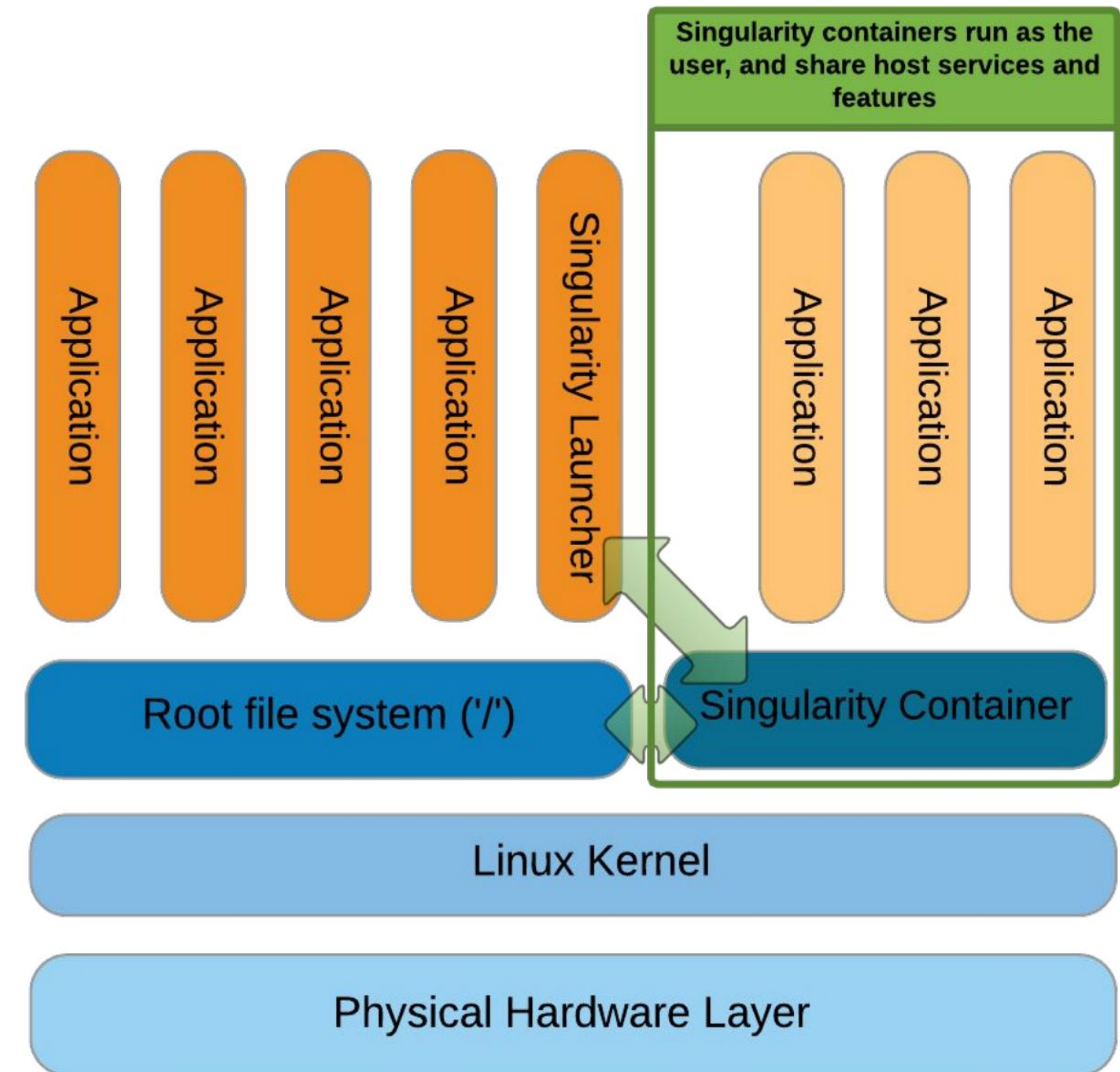
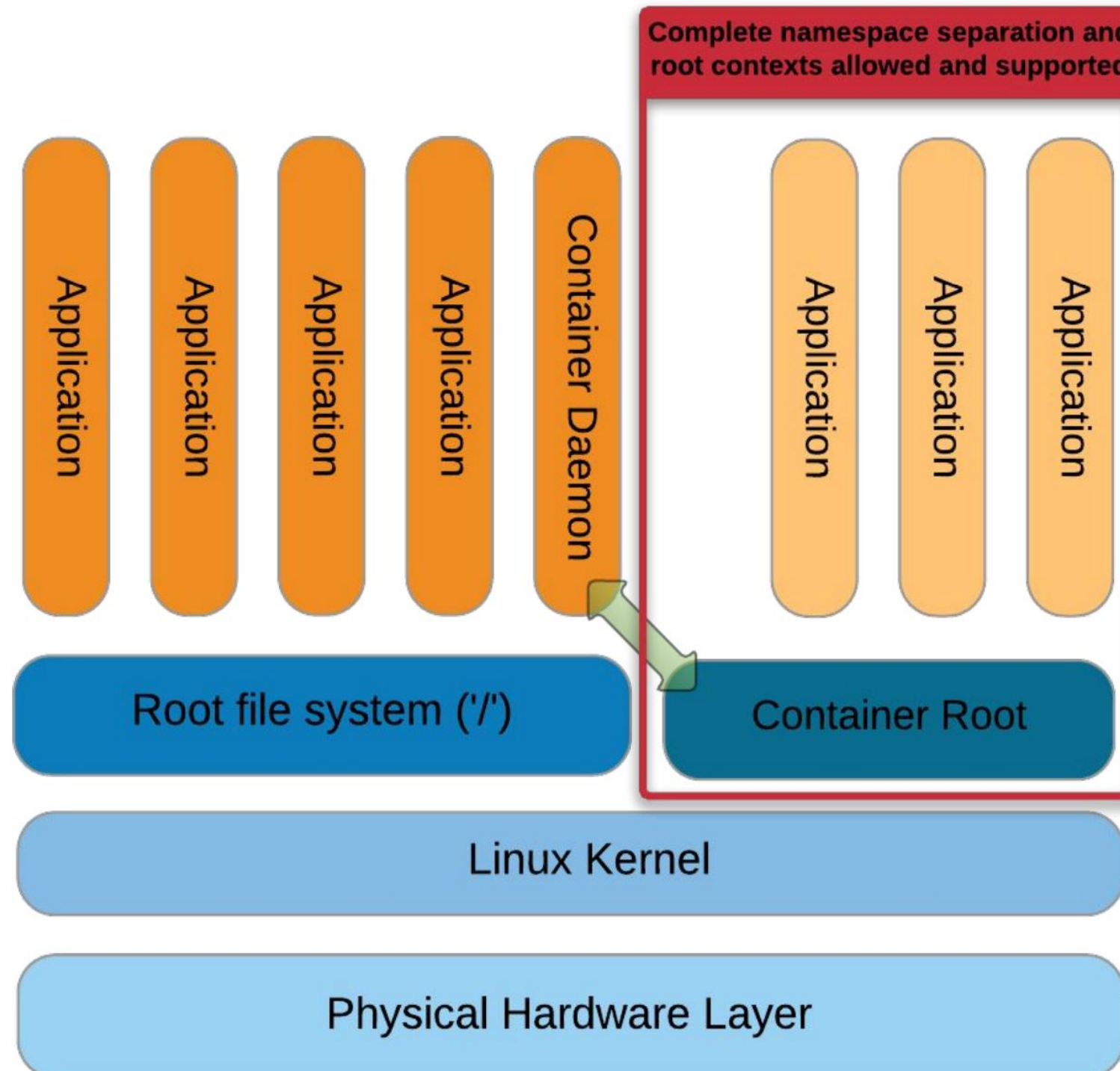
1. 기초

Docker vs Singularity

- Docker는 가장 널리 사용되는 컨테이너 런타임이지만 몇몇 이유로 인해 HPC 환경에서는 권장되지 않는다.
- 가장 큰 이유는 보안에 취약하다는 점이다.
- Docker가 기존 HPC 환경에 통합이 쉽지 않다는 것이다. HPC 환경은 대부분 RM을 사용하여 클러스터의 리소스를 사용할 job에 할당 하는 batch 시스템을 사용한다. 하지만 Docker의 경우 이러한 batch 시스템의 도입에 대한 고려 없이 설계되어 대부분의 RM들은 현재 Docker를 지원하지 않고 있다.
- Singularity는 Docker 컨테이너를 관리하기 위한 특별한 권한을 필요로 하는 Docker와 달리 HPC 센터의 보안성을 충족시키기 위해 특정한 권한 없이도 컨테이너 사용 을 할 수 있도록 개발
- Singularity는 기존 전통적으로 HPC 환경에서 사용하는 기술과의 호환을 위해 MPI, InfiniBand, Lustre 같이 HPC 환경에 필수적으로 사용되는 기술들을 지원
- RM과의 호환성을 위해 자체적으로 Slurm 플러그인을 지원
- Singularity는 낮은 버전의 OS가 사용되는 HPC 환경에서도 동작할 수 있도록 호환성을 지원

1. 기초

Docker vs Singularity



1. 기초

Apptainer란?

- Apptainer는 컨테이너 기술 중 하나로, 주로 HPC(High Performance Computing)와 연구 컴퓨팅 환경에서 신뢰성과 보안, 이식성을 위해 개발된 Singularity의 후속 오픈소스 프로젝트
- 원래 이름은 Singularity였으나, 현재는 Apptainer라는 이름으로 진행

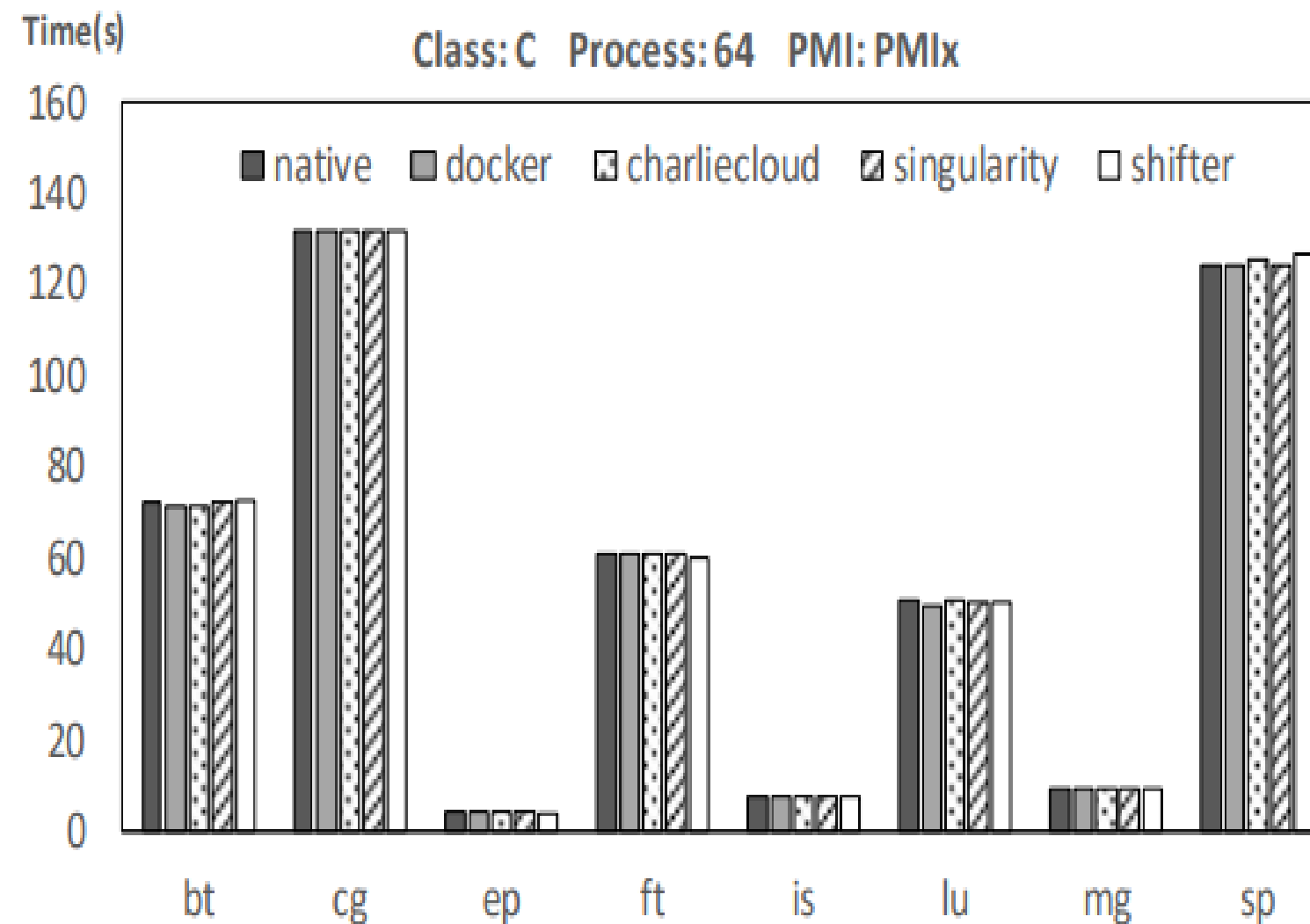
특징	내용
루트 권한 없음	컨테이너 내부에서 루트 권한이 필요하지 않음. 사용자 권한으로 실행하여 보안성을 높임.
HPC 친화적	MPI, GPU, IB(InfiniBand)와 같은 HPC 요소와 쉽게 통합 가능.
단일 이미지 파일 (SIF)	컨테이너를 단일 .sif 파일로 패키징해 이식성과 버전 관리 용이.
기존 환경과 통합	기존 파일 시스템, 유저 계정, 모듈과 쉽게 연동 가능.
재현성	연구 재현성을 위해 동일한 이미지로 어디서나 동일한 실행 환경 제공.

```
srun apptainer exec --mpi --nv /shared/containers/my_app.sif ./my_app_executable
```

1. 기초

Comparative Analysis of Container for High Performance Computing

한국컴퓨터정보학회논문지 Journal of The Korea Society of
Computer and Information Vol. 25 No. 9, pp. 11-20,
September 2020
<https://doi.org/10.9708/jksoci.2020.25.09.011>



Bench	Min	Max	NT/Min	DK/Min	CC/Min	SG/Min	SF/Min
BT	DK	SF	+0.7%	-	+0.3%	+0.8%	+1.4%
CG	CC	DK	+0.1%	+0.2%	-	+0.1%	+0.1%
EP	SF	SG	+12.2%	+12.0%	+17.4%	+17.6%	-
FT	SF	NT	+1.5%	+1.0%	+0.7%	+0.8%	-
IS	SG	DK	+1.6%	+1.9%	+1.2%	-	+0.4%
LU	DK	NT	+2.6%	-	+2.1%	+1.8%	+1.4%
MG	SF	SG	+0.4%	+0.3%	+0.1%	+1.3%	-
SP	SG	SF	+0.1%	+0.1%	+1.0%	-	+2.2%

* NT(native), DK(docker), CC(charliecloud), SG(singularity),
SF(shifter)

2. 클러스터 구성

감사합니다