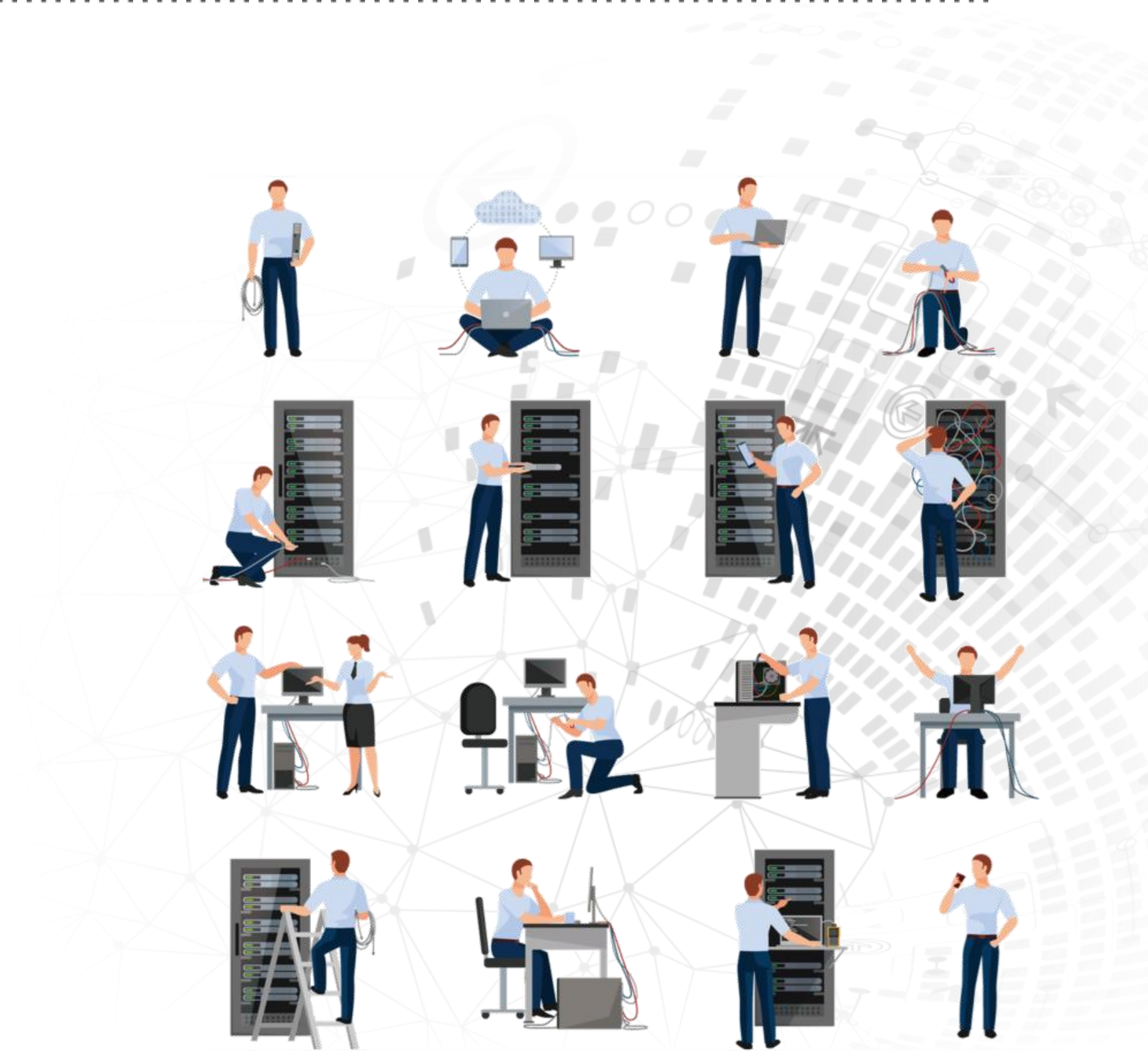


05 기타

AVX2 & AVX512



5. 기타 :

AVX2

```
// gemm_avx2.c
#include <immintrin.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define M 512
#define N 512
#define K 512

float *A, *B, *C;

void init_matrices() {
    A = (float *) aligned_alloc(32, M * K * sizeof(float));
    B = (float *) aligned_alloc(32, K * N * sizeof(float));
    C = (float *) aligned_alloc(32, M * N * sizeof(float));

    for (int i = 0; i < M * K; i++) A[i] = 1.0f;
    for (int i = 0; i < K * N; i++) B[i] = 1.0f;
    for (int i = 0; i < M * N; i++) C[i] = 0.0f;
}

void gemm_avx2() {
    for (int i = 0; i < M; i++) {
        for (int j = 0; j < N; j += 8) {
            __m256 c_vec = _mm256_load_ps(C + i * N + j);
            for (int k = 0; k < K; k++) {
                __m256 b_vec = _mm256_load_ps(B + k * N + j);
                __m256 a_val = _mm256_set1_ps(A[i * K + k]);
                c_vec = _mm256_fmadd_ps(a_val, b_vec, c_vec);
            }
            _mm256_store_ps(C + i * N + j, c_vec);
        }
    }
}
```

```
int main() {
    init_matrices();

    clock_t start = clock();
    gemm_avx2();
    clock_t end = clock();

    // 검증용 일부 출력
    printf("C[0][0] = %.2f\n", C[0]);
    printf("Expected = %.2f\n", K * 1.0f); // A와 B 모두 1이므로
    printf("Time = %.6f sec\n", (double)(end - start) / CLOCKS_PER_SEC);

    free(A);
    free(B);
    free(C);
    return 0;
}
```

5. 기타 :

AVX512

```
// gemm_avx512.c
#include <immintrin.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define M 512
#define N 512
#define K 512

float *A, *B, *C;

void init_matrices() {
    A = (float *) aligned_alloc(64, M * K * sizeof(float));
    B = (float *) aligned_alloc(64, K * N * sizeof(float));
    C = (float *) aligned_alloc(64, M * N * sizeof(float));

    for (int i = 0; i < M * K; i++) A[i] = 1.0f;
    for (int i = 0; i < K * N; i++) B[i] = 1.0f;
    for (int i = 0; i < M * N; i++) C[i] = 0.0f;
}

void gemm_avx512() {
    for (int i = 0; i < M; i++) {
        for (int j = 0; j < N; j += 16) {
            __m512 c_vec = _mm512_load_ps(C + i * N + j);
            for (int k = 0; k < K; k++) {
                __m512 b_vec = _mm512_load_ps(B + k * N + j);
                __m512 a_val = _mm512_set1_ps(A[i * K + k]);
                c_vec = _mm512_fmadd_ps(a_val, b_vec, c_vec);
            }
            _mm512_store_ps(C + i * N + j, c_vec);
        }
    }
}
```

```
int main() {
    init_matrices();

    clock_t start = clock();
    gemm_avx512();
    clock_t end = clock();

    printf("C[0][0] = %.2f\n", C[0]);
    printf("Expected = %.2f\n", K * 1.0f); // A와 B 모두 1이므로
    printf("Time = %.6f sec\n", (double)(end - start) / CLOCKS_PER_SEC);

    free(A);
    free(B);
    free(C);
    return 0;
}
```

5. 기타 :

AVX2 전용

```
gcc -O3 -mavx2 -mfma gemm_avx2.c -o gemm_avx2
```

AVX512 전용

```
gcc -O3 -mavx512f -mfma gemm_avx512.c -o gemm_avx512
```

실행

```
./gemm_avx2
```

```
./gemm_avx512
```

속도 차이 AVX-512가 AVX2보다 약 1.2~2배 빠름 (CPU 따라 다름)
더 큰 M,N,K 로 확장하면 차이가 더 분명합니다.

5. 기타 :

GEMM 벡터화 연산 흐름
(AVX2 / AVX-512)

5. 기타 :

GEMM 행렬곱 구조 개요

$$C[M \times N] = A[M \times K] \times B[K \times N]$$

$$A: 512 \times 512$$

$$B: 512 \times 512$$

$$C: 512 \times 512$$

$$C[i][j] = \sum_{k=0}^{511} A[i][k] \times B[k][j]$$

5. 기타 :

데이터 흐름 블록 다이어그램

- 1) A Row[i][k] \rightarrow Broadcast (스칼라)
- 2) B[k][j:j+VL] \rightarrow 벡터 Load
- 3) FMA: 스칼라 \times 벡터 + 누적 (Σ)
- 4) Store: C[i][j:j+VL]

VL = 8 (AVX2), VL = 16 (AVX-512)

5. 기타 :

루프 구조 흐름

for i in 0..511 (행)

for j in 0..511 step VL (열, SIMD 벡터)

c_vec = C[i][j:j+VL]

for k in 0..511 (내적)

b_vec = B[k][j:j+VL]

a_val = A[i][k] (브로드캐스트)

c_vec += a_val * b_vec

Store c_vec → C[i][j:j+VL]

5. 기타 :

SIMD 벡터 너비 비교

AVX2:

VL = 8 floats (256-bit)

Alignment: 32 bytes

AVX-512:

VL = 16 floats (512-bit)

Alignment: 64 bytes

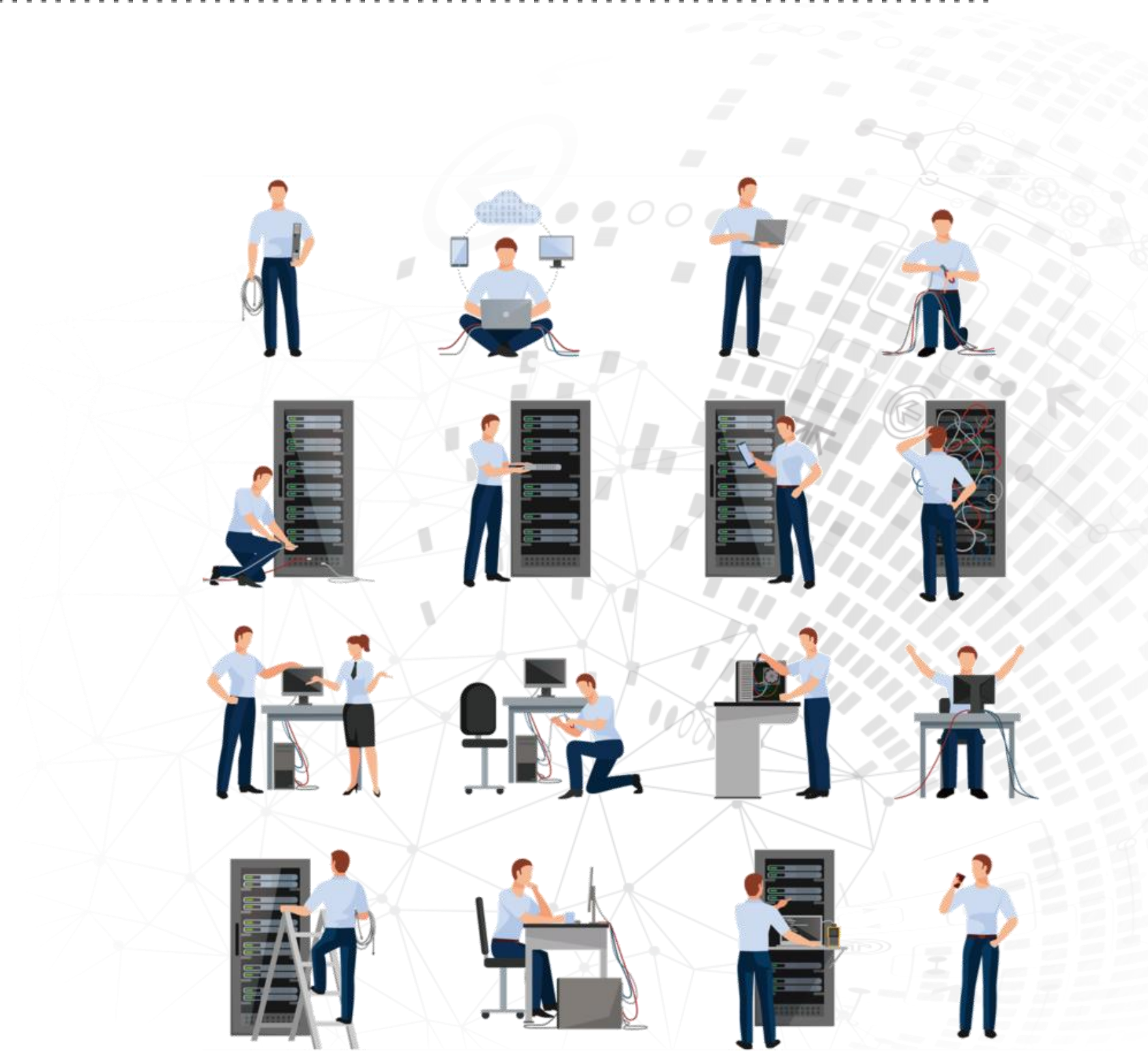
05 기타

HPC Latency



05 기타

NUMA-aware MPI Rank Mapping



05 기타

RFP 분석



5. 기타 :

현실적인 성능 예측

$$HPL_N = HPL_1 * N$$

5. 기타 :

현실적인 성능 예측

현실의 병렬 스케일링 문제들

통신 오버헤드 증가

노드 수가 늘어나면 MPI 통신량 증가.

각 노드 간의 메시지 전달, 동기화, 집계 연산 등의 오버헤드가 급격히 늘어납니다.

네트워크 병목

스위치, 링크 대역폭, 토폴로지에 따라 트래픽이 병목(Bottleneck)을 일으킬 수 있음.

특히 All-to-All 통신 패턴이 많은 HPL 특성상, 링크 사용량 증가 → 대기 시간 증가.

프로세스 간 부하 불균형

프로세서나 노드 간 HPL 성능이 균일하지 않으면, 느린 노드가 전체 성능을 떨어뜨립니다.

메모리 대역폭 차이, CPU binning 차이 등도 원인.

HPL의 Global Reduction 단계

LU 분해 중 특정 단계는 모든 노드 간의 연산 결과를 공유해야 하므로 병렬화가 제한됨.

5. 기타 :

현실적인 성능 예측

HPL 예측 모델 개요

(1) 단일 노드 기준 성능: HPL_1

AMD 및 Lenovo의 내부 벤치마크 데이터를 활용.

제조 공정 차이에 따라 단일 노드 HPL 성능이 편차를 보이므로 **최고 성능값이 아닌 평균 또는 최저 기준값을 사용.**

일반적으로 노드 메모리의 절반 이상을 HPL에 할당하여 최대 성능을 유도함.

(2) 스케일링 계수: s

노드 수를 2배로 늘릴 때마다 전체 성능이 얼마나 떨어지는지를 나타내는 계수.

네트워크 구성 (Interconnect), 아키텍처 (topology, blocking ratio), HPL 메모리 사용량 등을 종합 고려.

경험적으로 측정하거나, TOP500 데이터, 내부 클러스터 측정 (Lenox) 기반으로 설정됨.

예: $s = 0.99 \rightarrow$ 노드 수 2배 증가 시 성능 1% 감소

5. 기타 :

현실적인 성능 예측

$$HPL_N = HPL_1 * N * S^{\log_2 N}$$

- N: 전체 노드 수
- HPL_N: N개 노드에서 기대되는 HPL 성능 (GFlops)
- HPL_1: 단일 노드 HPL 성능
- S: 스케일링 계수

5. 기타 :

현실적인 성능 예측

$$S^{\log_2 N}$$

노드 수가 증가할수록 성능 효율이 감소하는 현실적 요소를 반영.

항목	설명
S	노드 수 2배 증가 시 성능 효율 감소율 (0.99 → 1% 감소)
$\log_2 N$	노드 증가로 인해 발생하는 통신/동기화 단계 수
$S^{\{\log_2 N\}}$	전체 성능 저하를 지수적으로 반영하는 항

5. 기타 :

현실적인 성능 예측

N (노드 수)	$\log_2 N$	$S^{\log_2 N}$ (S=0.99)	스케일링 손실
1	0	1.0000	0%
2	1	0.9900	1%
4	2	0.9801	2%
8	3	0.9703	3%
64	6	0.9415	5.85%
1024	10	0.9044	9.56%

5. 기타 :

현실적인 성능 예측

HPL 예측 모델 개요

$$\text{HPL}_1 = 5000 \text{ GFlops}$$

$$S = 0.99$$

$$N = 64 \text{ nodes}$$

$$\log_2 64 = 6$$

$$\begin{aligned}\text{HPL}_{64} &= 5000 \times 64 \times 0.99^6 \\ &= 320000 \times 0.94148 \\ &\approx 301273 \text{ GFlops}\end{aligned}$$

5. 기타 :

Nodes (N)	log2(N)	$S^{\log_2(N)}$	Predicted HPL_N (GFlops)
1	0	1	5000
2	1	0.99	9900
4	2	0.9801	19602
8	3	0.970299	38811.96
16	4	0.96059601	76847.6808
32	5	0.95099005	152158.408
64	6	0.941480149	301273.6478
128	7	0.932065348	596521.8227
256	8	0.922744694	1181113.209
512	9	0.913517247	2338604.154
1024	10	0.904382075	4630436.224

5. 기타 :

