Pertemuan 9 — Pengantar *High Performance Computing*

Metric, hukum Amdahl dan Gustafson, dan cara mengukur dengan jujur

Hendri Karisma

Semester Ganjil 2026/2027

Program Studi Teknik Informatika, STMIK Tazkia

Alur pertemuan hari ini

1. Jembatan: dari Capaian A ke Capaian B
2. Mengapa HPC: cuaca, iklim, simulasi, AI
3. FLOPS dan daftar TOP500
4. Taksonomi Flynn
5. *Shared memory* vs *distributed memory*
6. *Latency* vs *throughput*
7. *Speedup*, efisiensi, *strong* dan *weak scaling*
8. Hukum Amdahl, Gustafson, dan Karp–Flatt
9. Mengapa *speedup* tidak pernah linear
10. Mengukur dengan jujur, lalu praktikum

Sasaran pertemuan ini

Anda dapat menghitung *speedup* dan efisiensi dari hasil pengukuran sendiri, membedakan *strong* dan *weak scaling*, menaksir fraksi serial program dengan hukum Amdahl, dan menyusun eksperimen performa yang hasilnya dapat dipercaya.

Jembatan: dari Capaian A ke Capaian B

Pertanyaan yang sama, anggaran yang berbeda

Capaian A — Sistem Terdistribusi

"Bagaimana sistem tetap **benar dan hidup** walau terpecah di banyak mesin?"

- Ukuran: *availability*, *tail latency* P_{99}, *throughput* permintaan
- Banyak pekerjaan kecil yang saling bebas (satu permintaan = satu pekerjaan)
- Kegagalan dianggap normal; sistem tetap melayani

Capaian B — *High Performance Computing*

"Bagaimana memakai banyak prosesor agar **satu soal besar** selesai secepat mungkin?"

- Ukuran: waktu selesai, FLOPS, *speedup*, efisiensi
- Satu pekerjaan besar dipecah menjadi bagian yang saling bergantung
- Prosesor saling menunggu; komunikasi adalah musuh utama

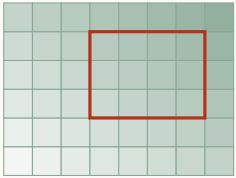
Ringkasan

Di paruh pertama, komunikasi antar mesin diukur dalam milidetik dan kita menoleransinya. Di paruh kedua, setiap mikrodetik komunikasi adalah waktu prosesor yang terbuang.

Mengapa HPC

Soal yang tidak selesai di satu komputer

- **Prakiraan cuaca** — atmosfer dibagi menjadi jutaan sel grid; tiap langkah waktu menghitung ulang semua sel. Prakiraan besok yang selesai lusa tidak berguna.
- **Iklim** — simulasi puluhan sampai ratusan tahun dengan banyak skenario emisi.
- **Simulasi ilmiah dan rekayasa** — dinamika molekul untuk obat, aliran udara pada sayap, uji tabrakan mobil, reaktor.
- **AI** — melatih model besar berarti triliunan operasi matriks yang diulang jutaan kali.



satu kotak merah = jatah satu prosesor

Pola khas HPC: domain dibagi (*domain decomposition*), tiap prosesor menghitung bagiannya, lalu bertukar nilai di tepi dengan tetangganya setiap langkah waktu.

Ringkasan

Ciri soal HPC: ada **tenggat** (hasil harus keluar sebelum tidak berguna) dan ada **ketergantungan** antar bagian (prosesor harus saling bertukar data).

FLOPS: satuan kerja HPC

FLOPS = *floating-point operations per second*.

Awalan	FLOPS	Gambaran kasar
GFLOPS	10^9	satu *core* CPU
TFLOPS	10^{12}	satu GPU modern
PFLOPS	10^{15}	satu *cluster* besar
EFLOPS	10^{18}	superkomputer *exascale*

Sejak 2022 superkomputer teratas sudah menembus skala *exaflops* pada uji LINPACK *double precision*.

TOP500

- Daftar 500 superkomputer tercepat, diperbarui dua kali setahun sejak 1993
- Peringkat memakai **HPL** (*High Performance LINPACK*): menyelesaikan sistem persamaan linear padat yang sangat besar
- Dua angka: **Rpeak** (puncak teoretis) dan **Rmax** (yang benar-benar dicapai HPL)
- Ada daftar pendamping: Green500 (FLOPS per watt) dan HPCG (beban yang lebih mirip aplikasi nyata, angkanya jauh di bawah HPL)

Sumber: `top500.org`

Ringkasan

Angka puncak adalah janji pabrik. Angka yang Anda capai bergantung pada seberapa baik program memberi makan prosesor dengan data.

Menghitung FLOPS puncak satu mesin

$$\text{FLOPS}_{\text{puncak}} = \text{jumlah } \textit{core} \times \text{frekuensi} \times \frac{\text{FLOP}}{\text{siklus per } \textit{core}}$$

Contoh CPU dengan AVX2 dan dua unit FMA per *core*:

- satu register AVX2 = 256 bit = 4 angka *double*
- FMA ($a \times b + c$) dihitung sebagai 2 FLOP
- per siklus: 2 unit $\times$ 4 $\times$ 2 $=$ 16 FLOP *double precision*
- 8 *core* $\times$ 3 GHz $\times$ 16 $=$ **384 GFLOPS**

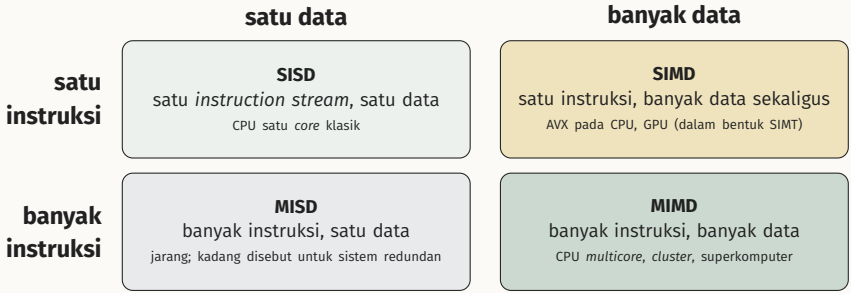
Jarang tercapai

Angka itu hanya tercapai bila setiap siklus semua unit FMA terisi data. Loop biasa yang ditulis apa adanya sering hanya mencapai beberapa persen. Pertemuan 10 membahas mengapa: *cache*, *memory bandwidth*, dan *vectorization*.

Coba di laptop Anda: `lscpu` menampilkan jumlah *core*, frekuensi, dan *flag* seperti `avx2`, `fma`, atau `avx512f`.

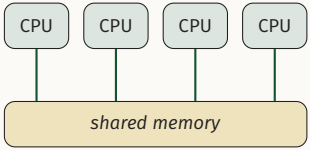
Taksonomi Flynn

Taksonomi Flynn (1966): instruksi × data

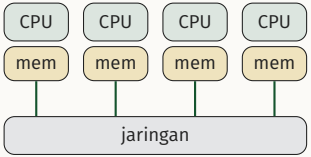


Mesin nyata adalah campuran: *cluster* (MIMD) berisi *node multicore* (MIMD) yang tiap *core*-nya punya unit vektor (SIMD), ditambah GPU (SIMT). Di HPC, gaya pemrograman yang paling umum disebut **SPMD**: satu program yang sama dijalankan banyak proses, tiap proses menangani potongan datanya sendiri.

Shared memory vs distributed memory



- Semua *core* membaca alamat memori yang sama
- Komunikasi = baca/tulis variabel; perlu *synchronization* (*lock*, *atomic*)
- Model: *thread*, OpenMP, *goroutine*
- Batas: satu mesin (puluhan sampai ratusan *core*)



- Tiap *node* punya memori sendiri
- Komunikasi = kirim *message* lewat jaringan
- Model: MPI, Ray/Dask, *actor*
- Batas: hampir tak terbatas (ribuan *node*)

Ringkasan

Superkomputer nyata memakai keduanya (***hybrid***): OpenMP atau CUDA di dalam *node*, MPI antar *node*. Pertemuan 10 membahas yang kiri, pertemuan 11 yang kanan.

Performance metric

Latency vs throughput

- ***Latency*** — berapa lama **satu** pekerjaan selesai (detik per tugas).
- ***Throughput*** — berapa **banyak** pekerjaan selesai per satuan waktu (tugas per detik, FLOPS, GB/s).

Analogi: satu mobil sport membawa 2 orang Bandung–Jakarta dalam 2 jam. Satu bus membawa 50 orang dalam 3 jam. Mobil menang *latency*; bus menang *throughput*.

Dalam HPC

- CPU dirancang menekan *latency* satu *thread*; GPU dirancang memaksimalkan *throughput*
- Komunikasi antar prosesor punya keduanya: *latency* (*fixed cost* tiap *message*) dan *bandwidth* (*cost* per byte)
- Paralelisme hampir selalu menaikkan *throughput*; ia menurunkan *latency* satu soal hanya jika soalnya bisa dipecah

Ringkasan

Sebelum mengoptimalkan, tanyakan dulu: kita mengejar *latency* (satu soal lebih cepat) atau *throughput* (lebih banyak soal per jam)? Desainnya berbeda untuk keduanya.

Speedup dan efisiensi

$$S(p) = \frac{T(1)}{T(p)} \qquad E(p) = \frac{S(p)}{p}$$

- $T(1)$: waktu dengan 1 prosesor, sebaiknya versi **serial terbaik**, bukan versi paralel yang dijalankan dengan 1 *thread*
- $T(p)$: waktu dengan p prosesor
- $S(p) = p$ disebut *linear speedup*; $E = 1$ berarti tidak ada prosesor yang menganggur

p	$T(p)$ (detik)	$S(p)$	$E(p)$
1	64,0	1,00	100%
2	33,5	1,91	96%
4	18,0	3,56	89%
8	10,4	6,15	77%
16	6,8	9,41	59%

Data contoh untuk latihan, bukan hasil mesin tertentu.

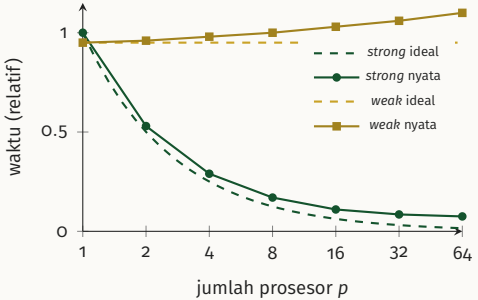
Ringkasan

Speedup yang terus naik belum tentu kabar baik: efisiensi turun dari 96% ke 59%. Pada 16 prosesor, hampir separuh waktu prosesor terbuang.

Strong scaling vs *weak scaling*

- ***Strong scaling*** — ukuran soal **tetap**, prosesor ditambah. Ideal: waktu turun sebanding $1/p$. Pertanyaannya: "seberapa cepat soal ini bisa selesai?"
- ***Weak scaling*** — ukuran soal **per prosesor** tetap; soal total membesar bersama p. Ideal: waktu tetap datar. Pertanyaannya: "seberapa besar soal yang bisa diselesaikan dalam waktu yang sama?"

Prakiraan cuaca dengan grid lebih halus pada mesin lebih besar adalah contoh *weak scaling*.



Ilustrasi bentuk kurva.

Hukum Amdahl (1967): *strong scaling* punya batas

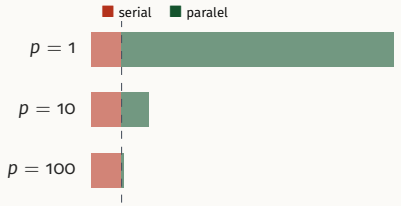
Misalkan f = fraksi waktu serial (tidak bisa diparalelkan):

$$S(p) = \frac{1}{f + \dfrac{1-f}{p}} \qquad \lim_{p\to\infty} S(p) = \frac{1}{f}$$

$\lim_{p\to\infty}$ (dibaca *limit*; ∞: *infinity*): nilai $S(p)$ bila p sangat besar.

Contoh: program 100 detik, 10 detik di antaranya membaca *file* secara serial ($f = 0{,}1$).

- $p = 10$: $T = 10 + 90/10 = 19$ detik, $S \approx 5{,}3$
- $p = 100$: $T = 10 + 0{,}9 = 10{,}9$ detik, $S \approx 9{,}2$
- $p = \infty$: $T = 10$ detik, $S = 10$, tidak lebih



Bagian paralel menyusut, bagian serial tetap. Lama-lama yang tersisa hanya bagian merah.

Ringkasan

Sebelum menambah prosesor, perkecil dulu bagian serial. Mengurangi f dari 10% ke 5% menaikkan batas *speedup* dari $10\times$ ke $20\times$.

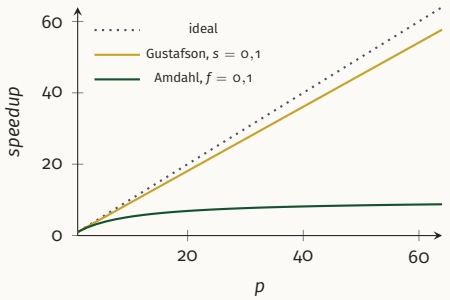
Hukum Gustafson (1988): *weak scaling* lebih optimis

Amdahl memegang **ukuran soal** tetap. Gustafson memegang **waktu** tetap dan bertanya: soal sebesar apa yang selesai dalam waktu itu?

$$S(p) = s + p\,(1 - s) = p - s\,(p - 1)$$

s = fraksi serial yang diukur **pada mesin paralel**.

Contoh $s = 0{,}1$, $p = 100$:
$S = 100 - 0{,}1 \times 99 = 90{,}1$.



Ringkasan

Keduanya benar; mereka menjawab pertanyaan berbeda. Amdahl: “bisakah soal **ini** selesai lebih cepat?” Gustafson: “bisakah kita menyelesaikan soal yang **lebih besar**?” Superkomputer hidup dari jawaban kedua.

Menaksir fraksi serial dari data: Karp–Flatt

Dari *speedup* terukur S pada p prosesor, fraksi serial eksperimental:

$$e = \frac{\frac{1}{S} - \frac{1}{p}}{1 - \frac{1}{p}}$$

Rumus ini didapat dengan membalik hukum Amdahl: “kalau hasil ini dijelaskan dengan Amdahl, berapa f-nya?”

Cara membaca:

- e **tetap** saat p naik $\Rightarrow$ penghambatnya bagian serial
- e **naik** saat p naik $\Rightarrow$ penghambatnya *overhead*: komunikasi, *synchronization*, *load imbalance*

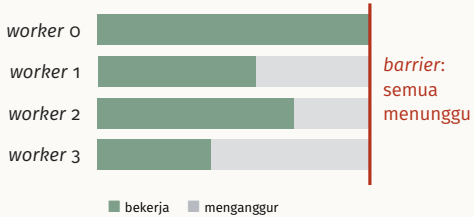
Memakai data contoh tadi:

p	$S(p)$	e
2	1,91	0,047
4	3,56	0,042
8	6,15	0,043
16	9,41	0,047

e relatif tetap di sekitar 4–5%. Kesimpulan: ada kira-kira 4–5% pekerjaan serial; batas *speedup* sekitar 20–25×. Menambah prosesor di atas 32 tidak sepadan sebelum bagian serial itu diperbaiki.

Mengapa *speedup* tidak pernah linear

Penghambat 1: *load imbalance*



Waktu total ditentukan *worker* yang paling lambat.

Penyebab umum:

- Potongan data sama besar, tetapi *cost* per elemen berbeda (contoh: menguji bilangan prima besar lebih mahal daripada yang kecil)
- Prosesor tidak seragam (*core* performa dan *core* efisiensi pada CPU laptop modern)
- Gangguan proses lain di mesin yang sama

Obat:

- Pembagian *cyclic* atau acak, bukan blok berurutan
- Potongan kecil yang diambil dinamis (*work queue*, *work stealing*)
- OpenMP `schedule(dynamic)` — pertemuan 10

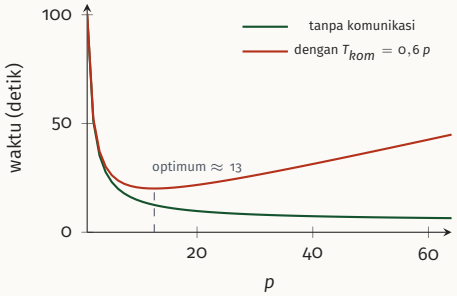
Penghambat 2: komunikasi dan *synchronization*

Model sederhana waktu paralel:

$$T(p) = \underbrace{T_s}_{\text{serial}} + \underbrace{\frac{T_{par}}{p}}_{\text{dibagi}} + \underbrace{T_{kom}(p)}_{\text{komunikasi}}$$

T_{kom} biasanya **naik** bersama p: lebih banyak *message*, lebih banyak *lock* yang diperebutkan, *reduction* yang bertingkat ($\sim \log p$, sebanding dengan logaritma p: dua kali lipat prosesor hanya menambah satu tingkat).

Akibatnya ada **titik optimum**: setelah itu menambah prosesor justru memperlambat.



Angka ilustrasi.

Prinsip yang akan berulang sampai pertemuan 15: **perbanyak hitung, perkecil bicara**. Rasio komputasi terhadap komunikasi adalah kunci HPC.

Kadang *speedup* melebihi p: *superlinear*

- Data 64 MB tidak muat di *cache* satu *core*; dibagi ke 8 *core*, tiap potongan 8 MB mungkin muat. Akses memori mendadak jauh lebih cepat.
- Pada pencarian (misalnya mencari satu solusi), salah satu prosesor bisa beruntung menemukan jawaban jauh lebih awal.

Ini **bukan** pelanggaran Amdahl. Amdahl berasumsi pekerjaan per elemen tetap; di sini pekerjaannya ikut berubah karena perilaku *cache*.

Curigai dulu

Sebelum bangga dengan *speedup* superlinear, periksa:

- Apakah versi serial Anda memang yang terbaik?
- Apakah hasil perhitungannya sama?
- Apakah pengukuran versi serial terganggu (*cold cache*, CPU sedang turun frekuensi)?

Mengukur dengan jujur

Lima aturan eksperimen performa

1. **Pemanasan** (*warm-up*) — jalankan sekali dan buang hasilnya. Putaran pertama membayar *page fault*, *cold cache*, kompilasi JIT, dan pemuatan pustaka.
2. **Ulangi** — minimal 5 kali per *configuration*. Satu angka bukan pengukuran.
3. **Laporkan median dan sebaran** — median tahan terhadap satu putaran yang terganggu; sertakan minimum–maksimum atau simpangan baku.
4. **Ubah satu variabel** — bila jumlah *thread* dan ukuran data berubah bersamaan, Anda tidak tahu mana yang berpengaruh.
5. **Catat lingkungan** — model CPU, jumlah *core* fisik vs logis, *compiler* dan *flag*-nya, versi bahasa, apakah laptop dicolok ke listrik.

Ringkasan

Laptop dengan baterai, *browser* terbuka, dan *turbo boost* aktif adalah laboratorium yang berisik. Tutup aplikasi lain, colok listrik, dan ulangi pengukuran.

Jebakan khas saat membaca hasil

- ***Hyper-threading*** — 8 *core* logis sering hanya 4 *core* fisik. *Speedup* di atas jumlah *core* fisik biasanya kecil.
- **Turbo dan termal** — frekuensi satu *core* lebih tinggi daripada saat semua *core* aktif. $T(1)$ diuntungkan, s*peedup* terlihat lebih buruk.
- **Pembanding yang curang** — membandingkan versi paralel teroptimasi dengan versi serial yang dikompilasi tanpa *optimization*.
- **Mengukur yang salah** — waktu total termasuk membaca *file* dan mencetak, padahal yang diparalelkan hanya inti hitungnya. Ukur keduanya, laporkan terpisah.
- ***Compiler* terlalu pintar** — hasil yang tidak pernah dipakai bisa dihapus oleh *compiler*; cetak atau periksa hasilnya.
- **Skala sumbu** — grafik *speedup* dengan sumbu log dapat menyembunyikan efisiensi yang buruk; tampilkan juga efisiensi.

Cek cepat di Linux: `lscpu` (*core* fisik, *thread per core*), `nproc`, dan `watch -n1 "grep MHz /proc/cpuinfo"` untuk melihat frekuensi.

Praktikum: *speedup* 1 sampai *N core*

Program uji (Go): menghitung bilangan prima

```go
func prima(n int) bool {
    if n < 2 { return false }
    for d := 2; d*d <= n; d++ {
        if n%d == 0 { return false }
    }
    return true
}

func main() {
    p, _ := strconv.Atoi(os.Args[1]) // worker
    const N = 20_000_000
    runtime.GOMAXPROCS(p)  // batasi core
    hasil := make([]int, p)
    var wg sync.WaitGroup
    mulai := time.Now()
    for w := 0; w < p; w++ {
        wg.Add(1)
        go func(w int) {
            defer wg.Done()
            c := 0                  // penghitung lokal
            for i := w; i < N; i += p { // cyclic
                if prima(i) { c++ }
            }
            hasil[w] = c            // tulis sekali
        }(w)
    }
    wg.Wait()
    total := 0
    for _, c := range hasil { total += c }
    fmt.Printf("%d,%.4f,%d\n", p,
        time.Since(mulai).Seconds(), total)
}
```

Package yang diimpor: `fmt`, `os`, `runtime`, `strconv`, `sync`, `time`. Perhatikan dua keputusan desain: pembagian *cyclic* (bukan blok) dan penghitung lokal. Alasannya dibahas pertemuan 10 (*false sharing*).

Versi Python: `multiprocessing`

```python
import sys, time
from multiprocessing import Pool

def prima(n):
    if n < 2:
        return False
    d = 2
    while d * d <= n:
        if n % d == 0:
            return False
        d += 1
    return True

def hitung(args):
    awal, n, p = args
    return sum(1 for i in range(awal, n, p)
               if prima(i))

if __name__ == "__main__":
    p, N = int(sys.argv[1]), 2_000_000
    mulai = time.perf_counter()
    with Pool(p) as pool:
        total = sum(pool.map(hitung,
                    [(w, N, p) for w in range(p)]))
    dt = time.perf_counter() - mulai
    print(f"{p},{dt:.4f},{total}")
```

- Dipakai *process*, bukan *thread*: *thread* Python terhalang GIL untuk kerja CPU (pertemuan 1 dan 6).
- N dibuat 10 kali lebih kecil daripada versi Go karena Python murni jauh lebih lambat.
- Waktu membuat *pool* ikut terukur. Itu bagian serial nyata yang akan terlihat di Karp–Flatt.

Ringkasan

Bandingkan juga $T(1)$ Go vs Python. Pilihan bahasa sering memberi *speedup* lebih besar daripada menambah *core*.

Menjalankan eksperimen

Satu putaran pemanasan, lalu 5 ulangan untuk tiap *p*. Setiap baris *output*: `p,detik,jumlah_prima`.

```
go build -o prima prima.go
./prima 1 > /dev/null                    # pemanasan, dibuang
for p in 1 2 4 8 $(nproc); do
  for r in 1 2 3 4 5; do ./prima $p >> hasil.csv; done
done
```

Ringkasan

Kolom `jumlah_prima` harus sama untuk semua *p*. Program paralel yang cepat tetapi hasilnya berbeda adalah program yang salah.

Mengolah hasil: median, *speedup*, Karp–Flatt, Amdahl

```python
import csv, statistics as st
import matplotlib.pyplot as plt
data = {}
for p, t, _ in csv.reader(open("hasil.csv")):
    data.setdefault(int(p), []).append(float(t))
P = sorted(data); T = {p: st.median(data[p]) for p in P}
S = {p: T[1] / T[p] for p in P}
e = [(1/S[p] - 1/p) / (1 - 1/p) for p in P if p > 1]
f = st.median(e)                                  # fraksi serial taksiran
plt.plot(P, [S[p] for p in P], "o-", label="terukur")
plt.plot(P, [1/(f + (1-f)/p) for p in P], "--", label=f"Amdahl f={f:.3f}")
plt.plot(P, P, ":", label="ideal")
plt.xlabel("p"); plt.ylabel("speedup"); plt.legend(); plt.savefig("speedup.png")
```

Langkah praktikum

1. Catat lingkungan: *output* `lscpu`, versi Go dan Python, laptop dicolok listrik atau tidak.
2. Jalankan versi Go untuk $p = 1, 2, 4, \ldots$ sampai jumlah *core* logis, 5 ulangan setelah pemanasan.
3. Jalankan versi Python dengan cara yang sama.
4. Buat tabel p, median $T(p)$, $S(p)$, $E(p)$, dan Karp–Flatt e untuk keduanya.
5. Gambar kurva *speedup* terukur, ideal, dan Amdahl dengan f taksiran.
6. Eksperimen tambahan: ganti pembagian *cyclic* menjadi blok (`for i := w*N/p; i < (w+1)*N/p; i++`). Apa yang terjadi pada *speedup*? Mengapa?
7. Eksperimen *weak scaling*: buat N sebanding dengan p. Apakah waktunya datar?

Yang dinilai

Seberapa **jujur** pengukuran Anda dan seberapa tepat Anda **menjelaskan** bentuk kurvanya. Besar kecilnya *speedup* tidak menentukan nilai.

Penutup

Ringkasan

- HPC mengejar **waktu selesai** satu soal besar; ukurannya FLOPS, *speedup*, dan efisiensi.
- Flynn: SISD, SIMD, MISD, MIMD. Mesin nyata adalah campuran; gaya pemrograman umum adalah SPMD.
- *Shared memory* (OpenMP, *thread*) di dalam *node*, *distributed memory* (MPI) antar *node*; superkomputer memakai keduanya.
- $S = T(1)/T(p)$, $E = S/p$. *Strong scaling*: soal tetap. *Weak scaling*: soal per prosesor tetap.
- Amdahl memberi batas $1/f$; Gustafson menunjukkan soal yang lebih besar dapat diselesaikan; Karp–Flatt menaksir f dari data.
- Penghambat nyata: bagian serial, *load imbalance*, komunikasi, *synchronization*.
- Pengukuran yang jujur: pemanasan, ulangan, median dan sebaran, satu variabel, catat lingkungan.

Tugas untuk pertemuan berikutnya

1. Selesaikan praktikum hari ini: laporan 2–3 halaman berisi tabel, grafik *speedup* dan efisiensi, taksiran *f*, serta penjelasan bentuk kurvanya.
2. Baca Pacheco & Malensek Bab 2 (*parallel hardware and software*) dan Hennessy & Patterson Bab 1 bagian kinerja.
3. Pasang `gcc` dengan dukungan OpenMP (`gcc -fopenmp`) dan pastikan `perf` dapat dijalankan (Linux/WSL).
4. Pertanyaan pemantik: mengapa program yang sama bisa 10 kali lebih cepat hanya dengan menukar urutan dua *loop*?

Pertemuan depan

Kita membuka “kotak hitam” CPU: *pipeline*, *cache*, *memory wall*, SIMD, dan OpenMP, lalu mempercepat perkalian matriks langkah demi langkah.

Terima kasih

Pertanyaan?