Pertemuan 10 — CPU, Memori, dan OpenMP

Mengapa program cepat atau lambat, dan cara memakai semua *core*

Hendri Karisma

Semester Ganjil 2026/2027

Program Studi Teknik Informatika, STMIK Tazkia

Alur pertemuan hari ini

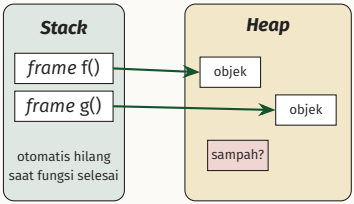
1. Pendahuluan: bagaimana bahasa mengelola memori (C/C++, GC, Rust)
2. Mengapa CPU berhenti makin cepat sendiri
3. Di dalam satu *core*: *pipeline*, *superscalar*, *out-of-order*, *branch prediction*
4. Hierarki memori dan *memory wall*
5. *Cache line*, *locality*, MESI, *false sharing*, NUMA
6. SIMD dan *vectorization*
7. Model *roofline*
8. OpenMP: *parallel for*, *reduction*, *schedule*
9. Studi kasus: perkalian matriks dari lambat ke cepat
10. Kaitan dengan Data Engineering: *columnar* dan *vectorized execution*
11. Praktikum

Sasaran pertemuan ini

Anda dapat menjelaskan mengapa sebuah *loop* cepat atau lambat dari sudut pandang *cache*, *bandwidth*, dan *vectorization*, memperkirakan batas performanya dengan *roofline*, lalu memparalelkannya dengan OpenMP dan mengukur GFLOPS-nya.

Pendahuluan: bagaimana bahasa mengelola memori

Stack, *heap*, dan tiga cara membersihkan *heap*



Yang sulit adalah *heap*: siapa yang memutuskan kapan objek boleh dibuang?

Cara	Siapa membersihkan	Contoh
Manual	Programmer (`free`, `delete`)	C, C++ lama
Garbage collection	*Runtime*, saat program berjalan	Java, Go, C#, Python, JavaScript
Ownership	*Compiler*, saat kompilasi	Rust (dan C++ modern lewat RAII, sebagian)

Ringkasan

Pilihan ini menentukan tiga hal yang penting di HPC dan sistem terdistribusi: **keamanan memori**, **jeda tak terduga**, dan **kendali atas tata letak data** di memori.

C dan C++ dulu: `malloc`/`free`, `new`/`delete`

```c
char *buat_nama(void) {
    char *p = malloc(16);
    strcpy(p, "nasabah-0001-xyz"); // butuh 17
    return p;
}
int main(void) {
    char *a = buat_nama();
    free(a);
    printf("%s\n", a);   // use-after-free
    free(a);             // double free
    char *b = malloc(1024);
    return 0;            // b tidak di-free: leak
}
```

- Programmer penuh kendali, tanpa *runtime cost*. Karena itu C dan C++ menjadi bahasa sistem operasi, *database*, dan HPC.
- Empat bug klasik: ***memory leak***, ***use-after-free***, ***double free***, ***buffer overflow***.
- Akibatnya bisa lebih parah dari *crash*: banyak serangan (eksekusi kode jarak jauh) berawal dari bug memori. Microsoft dan tim Chromium masing-masing melaporkan sekitar 70% celah keamanan serius mereka adalah masalah *memory safety*.
- Alat bantu: Valgrind, AddressSanitizer menemukan bug saat diuji, tidak mencegahnya.

Sumber: Microsoft Security Response Center (2019); Chromium Project, “Memory safety” (2020).

C++ modern: RAII dan *smart pointer* (C++11)

```cpp
#include <memory>
#include <vector>

void proses() {
  // dibebaskan otomatis saat keluar scope
  auto buf = std::make_unique<double[]>(1000000);
  std::vector<int> v(1000);          // juga RAII

  auto bersama = std::make_shared<Config>();
  auto salinan = bersama;            // refcount = 2
}   // buf, v, bersama, salinan dibebaskan di sini
```

- **RAII** (*Resource Acquisition Is Initialization*): sumber daya diikat ke umur objek; *destructor* melepasnya saat objek keluar *scope*.
- `unique_ptr`: satu pemilik, tanpa *overhead*. `shared_ptr`: banyak pemilik dengan *reference counting*.
- Jauh lebih aman, tetapi **tidak dipaksa**: *raw pointer* dan `new`/`delete` masih boleh, dan *compiler* tidak menolak *use-after-free*.
- Gagasan RAII inilah yang kelak dipaksakan oleh Rust.

Garbage collection: *runtime* yang memungut sampah

Reference counting

- Tiap objek menghitung berapa yang menunjuknya; saat nol, langsung dibebaskan.
- Python (CPython) memakainya, ditambah ***cycle collector*** untuk objek yang saling menunjuk. Swift memakai ARC yang disisipkan *compiler*.
- Kelemahan: siklus referensi, *cost* menaik-turunkan hitungan.

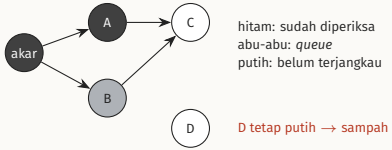
Tracing GC

- Mulai dari akar (*stack*, variabel global), tandai semua yang masih terjangkau, buang sisanya (*mark-sweep*).
- ***Generational***: kebanyakan objek mati muda, jadi *young generation* dipungut sering dan murah, *old generation* jarang.
- JVM: **G1** adalah *default*; **ZGC** dan **Shenandoah** bekerja hampir seluruhnya bersamaan dengan program agar jedanya sangat pendek. .NET juga *generational*.

Ringkasan

GC membuat program aman dari *use-after-free* dan *double free* tanpa usaha programmer. Harganya: memori ekstra, kerja CPU tambahan, dan **jeda** yang waktunya tidak kita tentukan.

GC di Go: dirancang untuk jeda pendek



Algoritma ***tri-color mark-sweep***, berjalan ***concurrent*** dengan program.

Sumber: Dokumentasi Go, “A Guide to the Go Garbage Collector”.

- Go memilih jeda pendek di atas segalanya: GC berjalan bersamaan dengan *goroutine*, hanya berhenti total (*stop-the-world*) sesaat.
- **Tidak** *generational* dan **tidak** memadatkan memori (*non-compacting*); lebih sederhana, tetapi lebih sering memindai.
- Penyetelan: `GOGC` (*default* 100: GC dipicu saat *heap* tumbuh 100% sejak GC terakhir) dan `GOMEMLIMIT` (batas memori lunak, sejak Go 1.19).
- Di layanan dengan *heap* besar, GC berkala bisa menaikkan *tail latency* (ingat contoh Discord dari pertemuan 1).

Rust tidak punya *garbage collector*

```rust
fn main() {
    let a = vec![1, 2, 3]; // a pemilik
    let b = a;             // pindah ke b
    // println!("{:?}", a); // DITOLAK:
                           // value moved
    let r = &b;            // borrow (baca)
    println!("{:?}", r);
}  // b keluar scope: dibebaskan (Drop)

// Banyak pemilik? pilih eksplisit:
// Rc<T> (satu thread), Arc<T> (antar-thread)
```

- ***Ownership***: tiap nilai punya tepat satu pemilik; saat pemilik keluar *scope*, memori dibebaskan (`Drop`, sama seperti RAII).
- ***Borrowing*** dan ***lifetime***: boleh banyak pembaca **atau** satu penulis, dan referensi tidak boleh hidup lebih lama dari datanya. Semua dicek **saat kompilasi**.
- Hasilnya: tanpa GC dan tanpa jeda, tetapi *use-after-free*, *double free*, dan *data race* ditolak sebelum program berjalan.
- `Rc`/`Arc`: *reference counting* opsional bila memang butuh banyak pemilik. `unsafe`: pintu darurat untuk kode tingkat rendah, dan tanggung jawabnya kembali ke programmer.

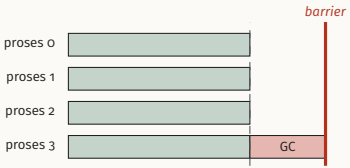
Membandingkan enam bahasa

Bahasa	Pengelolaan memori	Jeda GC	*Memory safety*	Kendali tata letak data
C	Manual	Tidak ada	Tidak dijamin	Penuh
C++	Manual + RAII, *smart pointer*	Tidak ada	Lebih baik, tidak dipaksa	Penuh
Java	*Tracing GC generational* (G1, ZGC)	Ada, bisa sangat pendek	Dijamin	Terbatas (objek di *heap*)
Go	*Tracing GC concurrent*	Ada, pendek	Dijamin (kecuali `unsafe`)	Sedang (`struct` berurutan)
Python	*Reference counting* + *cycle collector*	Ada	Dijamin	Rendah (pakai NumPy)
Rust	*Ownership*, dicek *compiler*	Tidak ada	Dijamin (kecuali `unsafe`)	Penuh

Sumber: Ringkasan kualitatif; besar jeda GC bergantung pada versi, ukuran *heap*, dan penyetelan.

Mengapa HPC memakai C/C++, Fortran, dan kini Rust

- **Kendali tata letak memori.** Hari ini kita akan melihat bahwa kecepatan ditentukan *cache line* dan SIMD. Itu hanya bisa dioptimalkan bila programmer menentukan susunan data persis.
- **Tanpa jeda tak terduga.** Program MPI sering menunggu semua proses di satu titik (*barrier*, *all-reduce*, pertemuan 11). Satu proses yang sedang GC menahan ribuan proses lain.
- ***Legacy code* dan *library*.** BLAS, LAPACK, banyak kode iklim dan fisika ditulis dalam Fortran dan C, dan masih yang tercepat.
- **Rust** masuk karena menawarkan kendali yang sama tanpa bug memori; Python tetap dipakai sebagai perekat di atas *library* C/C++/Fortran.



tiga proses menunggu satu proses yang sedang GC

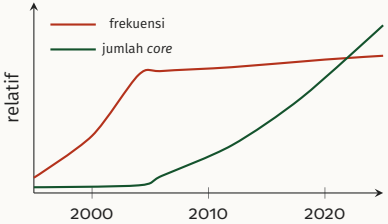
Ringkasan

Di layanan web, jeda GC menaikkan P99. Di HPC, jeda satu proses memperlambat **seluruh** *cluster*.

Mengapa CPU berhenti makin cepat sendiri

The free lunch is over

- Sampai awal 2000-an, frekuensi CPU naik terus. Program lama otomatis makin cepat tanpa diubah.
- Sekitar pertengahan 2000-an kenaikan frekuensi tertahan di kisaran beberapa GHz: daya dan panas naik terlalu cepat (berakhirnya *Dennard scaling*).
- Transistor tetap bertambah, tetapi dipakai untuk **lebih banyak *core***, *cache* lebih besar, dan unit vektor lebih lebar.
- Herb Sutter (2005) menyebutnya *"the free lunch is over"*: performa kini harus diambil sendiri oleh pemrogram lewat paralelisme.



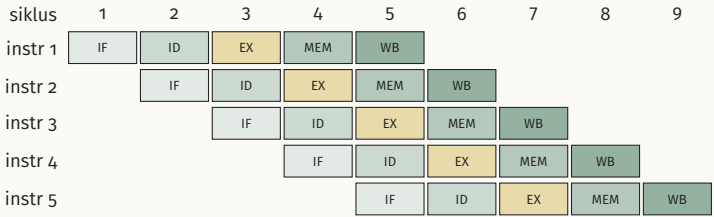
Ilustrasi bentuk tren, bukan data produk tertentu.

Ringkasan

Program serial hari ini hanya akan sedikit lebih cepat di CPU lima tahun lagi. Yang bertambah banyak adalah *core* dan lebar unit vektor yang tidak dipakainya.

Di dalam satu *core*

Pipeline: lini perakitan instruksi



- Lima tahap klasik: *fetch*, *decode*, *execute*, *memory*, *write back*
- Satu instruksi tetap butuh 5 siklus, tetapi setelah *pipeline* penuh, satu instruksi **selesai tiap siklus**
- CPU modern punya *pipeline* yang jauh lebih panjang

Hazard

Pipeline macet bila instruksi menunggu hasil instruksi sebelumnya (*data hazard*) atau bila CPU tidak tahu ke mana harus melompat (*control hazard*, yaitu percabangan).

Superscalar dan *out-of-order*: ILP

- ***Superscalar*** — beberapa unit eksekusi, sehingga lebih dari satu instruksi bisa selesai per siklus (IPC > 1).
- ***Out-of-order execution*** — CPU menjalankan instruksi yang datanya sudah siap lebih dulu, lalu menyusun ulang hasilnya agar tampak berurutan.
- ***Register renaming*** — menghapus ketergantungan palsu antar instruksi yang kebetulan memakai register yang sama.
- Semua ini disebut ***instruction-level parallelism*** (ILP): paralelisme di dalam satu *thread*, tanpa Anda tulis.

Satu akumulator: setiap penjumlahan menunggu yang sebelumnya.

```
double s = 0;
for (long i = 0; i < n; i++)
    s += x[i];          /* rantai bergantung */
```

Empat akumulator: empat rantai bebas, unit eksekusi terisi.

```
double s0 = 0, s1 = 0, s2 = 0, s3 = 0;
for (long i = 0; i + 3 < n; i += 4) {
    s0 += x[i];     s1 += x[i + 1];
    s2 += x[i + 2]; s3 += x[i + 3];
}
double s = s0 + s1 + s2 + s3;  /* + sisa */
```

Untuk *floating point*, *compiler* tidak boleh mengubah urutan penjumlahan sendiri (hasil pembulatannya bisa berbeda) kecuali diizinkan, misalnya dengan `-ffast-math`.

Branch prediction: data terurut bisa lebih cepat

```
/* data: 32 juta int acak 0..255 */
long jumlah = 0;
for (int r = 0; r < 10; r++)
    for (long i = 0; i < n; i++)
        if (data[i] >= 128)    /* cabang */
            jumlah += data[i];

/* versi tanpa cabang */
for (long i = 0; i < n; i++) {
    int m = -(data[i] >= 128);  /* 0 atau -1 */
    jumlah += data[i] & m;
}
```

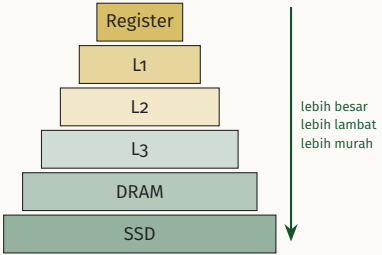
- CPU menebak arah `if` sebelum hasil perbandingan diketahui, lalu terus bekerja di jalur tebakan.
- Tebakan salah membuang pekerjaan di *pipeline* (belasan siklus).
- Data **acak**: tebakan salah sekitar separuh waktu. Data **terurut**: cabangnya berpola (“tidak, tidak, ..., ya, ya”), tebakan hampir selalu benar.
- Pada `-O1` perbedaannya biasanya terlihat jelas. Pada `-O3` *compiler* sering sudah mengubahnya menjadi kode tanpa cabang atau vektor, sehingga perbedaannya hilang.

Ringkasan

Ukur, jangan menebak: `perf stat -e branch-misses ./prog`

Hierarki memori dan *memory wall*

Hierarki memori: makin dekat, makin cepat, makin kecil



Tingkat	Kapasitas khas	*Latency* kira-kira
Register	ratusan byte	$<$ 1 ns
L1 (per *core*)	32–64 KB	$\sim$1 ns (beberapa siklus)
L2 (per *core*)	0,25–2 MB	$\sim$3–5 ns
L3 (bersama)	beberapa–puluhan MB	$\sim$10–20 ns
DRAM	GB	$\sim$80–100 ns
SSD NVMe	TB	$\sim$20–100 µs

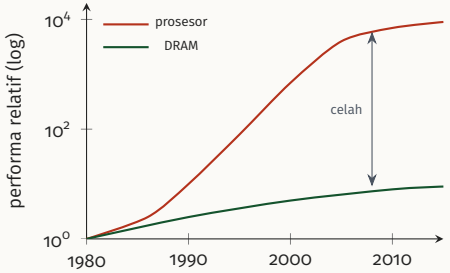
Sumber: angka berbeda tiap generasi; cek CPU Anda dengan `lscpu -C`.

Ringkasan

Satu akses ke DRAM setara ratusan operasi aritmetika yang bisa dikerjakan *core* di waktu yang sama. Program cepat adalah program yang jarang menunggu DRAM.

Memory wall

- Selama puluhan tahun, kecepatan prosesor tumbuh jauh lebih cepat daripada *latency* DRAM membaik.
- Celahnya terus melebar: istilah ***memory wall*** dipopulerkan pada pertengahan 1990-an untuk menggambarkan hal ini.
- Dengan banyak *core*, masalahnya bertambah: semua *core* berebut *memory bandwidth* yang sama.
- Jawaban arsitektur: *cache* bertingkat, *prefetcher*, lebih banyak kanal memori, dan HBM pada GPU (pertemuan 13).



Ilustrasi mengikuti bentuk grafik di Hennessy & Patterson.

Cache line dan *locality*

- Memori dipindahkan ke *cache* per ***cache line***, umumnya 64 byte pada x86 (8 angka *double*). Membaca satu byte berarti membawa 64 byte.
- ***Spatial locality*** — data yang berdekatan dipakai berdekatan waktunya: satu *cache line* termanfaatkan penuh.
- ***Temporal locality*** — data yang sama dipakai lagi tak lama kemudian: masih ada di *cache*.
- Array 2D di C disimpan *row-major*: `a[i][j]` dan `a[i][j+1]` bersebelahan; `a[i+1][j]` berjarak satu baris penuh.

```c
/* cepat: berjalan sepanjang baris */
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
        s += a[i][j];

/* lambat: melompat N*8 byte tiap akses */
for (int j = 0; j < N; j++)
    for (int i = 0; i < N; i++)
        s += a[i][j];
```

Untuk N besar (matriks jauh lebih besar dari L3), versi kedua bisa beberapa kali lebih lambat walau jumlah operasinya identik. Bandingkan dengan `perf stat -e cache-misses`.

Cache coherence: MESI secara ringkas

Tiap *core* punya *cache* sendiri. Bila dua *core* menyimpan salinan alamat yang sama, siapa yang benar? Protokol koherensi menjaga agar semua melihat nilai yang konsisten. Tiap *cache line* berada di salah satu *state*:

Modified	hanya di sini, sudah diubah, memori utama usang
Exclusive	hanya di sini, sama dengan memori
Shared	ada di beberapa *cache*, hanya dibaca
Invalid	salinan ini tidak berlaku

Skenario

1. *Core* 0 membaca $x \rightarrow$ E
2. *Core* 1 membaca $x \rightarrow$ keduanya S
3. *Core* 0 menulis $x \rightarrow$ *core* 0 M, salinan *core* 1 I
4. *Core* 1 membaca x lagi $\rightarrow$ *cache miss*, data diambil dari *core* 0, keduanya kembali S

Ringkasan

Menulis ke data bersama memaksa salinan di *core* lain dibatalkan. Itu lalu lintas yang mahal dan menjadi sumber *false sharing*.

False sharing: tidak berbagi data, tetap berebut

```c
#include <omp.h>
#define T 8
#define ITER 100000000L

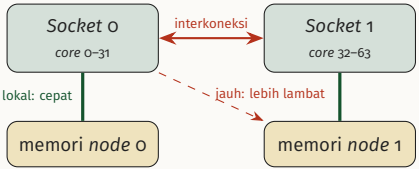
/* buruk: 8 penghitung berdempetan di
   satu cache line (8 x 8 byte = 64) */
_Alignas(64) struct { volatile long c; } hit[T];

/* baik: tiap penghitung di line sendiri */
struct { _Alignas(64) volatile long c; } hit2[T];

int main(void) {
    #pragma omp parallel num_threads(T)
    {
        int t = omp_get_thread_num();
        for (long i = 0; i < ITER; i++)
            hit[t].c++;      /* ganti hit2 */
    }
    return 0;
}
```

- Tiap *thread* hanya menulis penghitungnya sendiri; secara logika tidak ada data bersama.
- Tetapi penghitungnya satu *cache line*. Setiap tulisan membatalkan *line* itu di *core* lain (MESI), sehingga *line* terus "dilempar" antar *core*.
- Perbaikan: beri jarak (*padding*/*alignment* 64 byte), atau hitung di variabel lokal lalu tulis sekali di akhir, seperti program Go pertemuan 9.
- `volatile` di sini hanya agar *compiler* tidak menyimpan penghitung di register selama *loop*, sehingga efeknya terlihat.

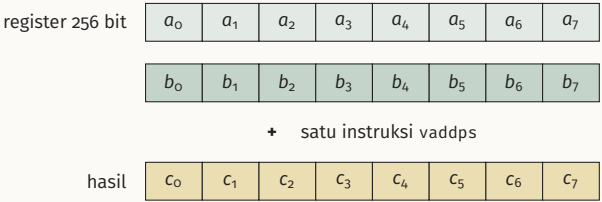
NUMA: tidak semua memori sama jauhnya



- ***Non-Uniform Memory Access***: *server* dengan beberapa *socket* (dan sebagian CPU besar satu *socket*) memiliki memori yang "dekat" dan "jauh".
- Linux umumnya memakai kebijakan ***first touch***: halaman memori ditempatkan di *node* milik *thread* yang **pertama kali menulisnya**.
- Akibatnya: *initialization* array secara serial menaruh semua data di satu *node*. Inisialisasikan secara paralel dengan pembagian yang sama seperti saat menghitung.
- Alat: `numactl --hardware`, `lscpu`, variabel `OMP_PROC_BIND` dan `OMP_PLACES`.

SIMD dan *vectorization*

SIMD: satu instruksi, banyak angka



Set instruksi	Lebar	float	double
SSE (x86)	128 bit	4	2
AVX, AVX2 (x86)	256 bit	8	4
AVX-512 (x86)	512 bit	16	8
NEON (ARM)	128 bit	4	2

Tanpa SIMD, Anda hanya memakai sebagian kecil lebar unit aritmetika. Pada AVX2 dengan *float*, kode skalar memakai 1 dari 8 jalur.

Auto-vectorization oleh *compiler*

```c
/* saxpy: y = a*x + y */
void saxpy(int n, float a,
           const float *restrict x,
           float *restrict y) {
    for (int i = 0; i < n; i++)
        y[i] = a * x[i] + y[i];
}
```

```
gcc -O3 -march=native -fopt-info-vec \
    -c saxpy.c
# saxpy.c:5:23: optimized: loop vectorized
#   using 32 byte vectors
```

Syarat agar *loop* mudah di-*vectorize*:

- Akses memori berurutan (*stride* 1)
- Tidak ada ketergantungan antar iterasi (`y[i]` tidak memakai `y[i-1]`)
- Tidak ada *aliasing*: `restrict` menjanjikan `x` dan `y` tidak tumpang tindih
- Jumlah iterasi bisa dihitung; tidak ada `break` atau *function call* rumit
- Bentuk data *structure of arrays* lebih ramah SIMD daripada *array of structures*

`-march=native` mengizinkan instruksi milik CPU ini (AVX2/AVX-512); hasilnya mungkin tidak jalan di CPU lain.

Model *roofline*

Roofline: dibatasi hitungan atau dibatasi memori?

Arithmetic intensity:

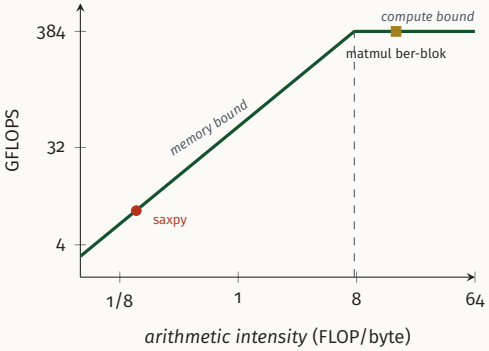
$$I = \frac{\text{FLOP}}{\text{byte dipindahkan dari memori}}$$

Performa yang bisa dicapai:

$$P = \min\left(P_{\text{puncak}},\ I \times BW\right)$$

Diperkenalkan Williams, Waterman, dan Patterson (2009).

Contoh mesin: $P_{\text{puncak}} = 384$ GFLOPS, $BW = 50$ GB/s. Titik belok (*ridge point*): $384/50 \approx 7{,}7$ FLOP/byte.



Menghitung *arithmetic intensity*

saxpy (*float*)

Per elemen: 2 FLOP (kali dan tambah). Data: baca `x[i]` 4 byte, baca `y[i]` 4 byte, tulis `y[i]` 4 byte = 12 byte.

$$I = 2/12 \approx 0{,}17 \;\Rightarrow\; P \leq 0{,}17 \times 50 \approx 8{,}3 \text{ GFLOPS}$$

Hanya sekitar 2% dari puncak. Menambah *core* tidak banyak menolong; yang membatasi adalah *memory bandwidth*.

Perkalian matriks $n \times n$ (*double*)

Total $2n^3$ FLOP atas $3n^2$ angka. Secara teori data bisa dipakai ulang sampai $O(n)$ kali ($O(n)$: tumbuh sebanding dengan n).

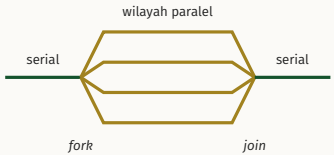
- Versi naif: tiap perkalian-tambah membaca ulang `A` dan `B` dari memori $\Rightarrow$ I kira-kira 2/16 FLOP/byte
- Versi ber-blok ukuran b: tiap angka yang dimuat dipakai sekitar b kali $\Rightarrow$ I naik sebanding b sampai melewati titik belok

Ringkasan

Roofline memberi tahu ***optimization* mana yang masuk akal**: kode *memory bound* perlu memperbaiki pemakaian ulang data dan *locality*; kode *compute bound* perlu *vectorization* dan lebih banyak *core*.

OpenMP

OpenMP: model *fork–join*



- OpenMP = *directive* `#pragma` untuk C, C++, Fortran. Kode tetap sah bila `-fopenmp` tidak dipakai.
- *Thread* utama membuat tim *thread*, membagi pekerjaan, lalu menunggu semua selesai (*implicit barrier*).
- Variabel di luar wilayah paralel bersifat *shared*; yang dideklarasikan di dalamnya *private*.

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

int main(void) {
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        int n  = omp_get_num_threads();
        printf("halo dari %d / %d\n", id, n);
    }
    return 0;
}
```

```
gcc -O2 -fopenmp halo.c -o halo
OMP_NUM_THREADS=4 ./halo
```

Urutan baris *output* berbeda setiap dijalankan: itulah tanda *concurrency*.

Parallel for dan *reduction*: menghitung π

Salah: semua *thread* menulis `jumlah` bersamaan (*race condition*). Hasilnya berubah-ubah.

```
double h = 1.0 / N, jumlah = 0.0;
#pragma omp parallel for
for (long i = 0; i < N; i++) {
    double x = (i + 0.5) * h;
    jumlah += 4.0 / (1.0 + x * x); /* race! */
}
```

Benar: tiap *thread* punya salinan, digabung di akhir.

```
#pragma omp parallel for reduction(+:jumlah)
for (long i = 0; i < N; i++) {
    double x = (i + 0.5) * h;
    jumlah += 4.0 / (1.0 + x * x);
}
double pi = jumlah * h;
```

- Ini integrasi numerik $\int_0^1 \frac{4}{1+x^2}\,dx = \pi$ (π dibaca *pi*, $\approx 3{,}14159$; $\int_0^1$ dibaca *integral* dari 0 sampai 1: luas di bawah kurva dari 0 sampai 1) dengan aturan titik tengah.
- `reduction(+:jumlah)` setara dengan pola "penghitung lokal lalu gabung" yang kita tulis manual di Go.
- Alternatif `#pragma omp atomic` atau `critical` juga benar, tetapi jauh lebih lambat: setiap iterasi berebut satu variabel.
- Hasil *reduction floating point* bisa berbeda di digit terakhir dengan versi serial karena urutan penjumlahannya berbeda. Itu wajar.

Schedule: cara membagi iterasi

Klausa	Cara kerja	Cocok untuk
`schedule(static)`	Iterasi dibagi rata menjadi blok berurutan sebelum mulai	*Cost* tiap iterasi seragam; *overhead* paling kecil
`schedule(static, c)`	Potongan berukuran *c* dibagikan bergiliran (*cyclic*)	*Cost* naik turun secara teratur
`schedule(dynamic, c)`	*Thread* yang selesai mengambil potongan *c* berikutnya dari *queue*	*Cost* iterasi tidak bisa ditebak; ada *overhead* per potongan
`schedule(guided, c)`	Seperti *dynamic*, tetapi potongan dimulai besar lalu mengecil sampai *c*	Kompromi antara keseimbangan dan *overhead*

Ringkasan

Kasus bilangan prima pertemuan 9: `schedule(static)` memberi *thread* terakhir bilangan terbesar yang paling mahal diuji (*load imbalance*). Coba `schedule(dynamic, 1000)` dan ukur perbedaannya.

Studi kasus: perkalian matriks

Langkah 1 dan 2: urutan *loop* `ijk` vs `ikj`

ijk (naif) — `B` dibaca per kolom: lompatan $n \times 8$ byte.

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++) {
        double s = 0;
        for (int k = 0; k < n; k++)
            s += A[i*n + k] * B[k*n + j];
        C[i*n + j] = s;
    }
```

ikj — semua akses di *loop* terdalam berurutan.

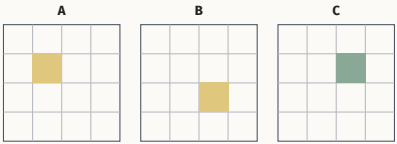
```
/* C harus diisi nol lebih dulu */
for (int i = 0; i < n; i++)
    for (int k = 0; k < n; k++) {
        double a = A[i*n + k];
        for (int j = 0; j < n; j++)
            C[i*n + j] += a * B[k*n + j];
    }
```

- Jumlah FLOP keduanya persis sama: $2n^3$. Yang berbeda hanya **pola akses memori**.
- Pada `ikj`, *loop* terdalam adalah saxpy atas satu baris: berurutan, mudah di-*vectorize*, dan satu *cache line* `B` dipakai penuh.
- Untuk n besar, `ikj` sering beberapa kali lebih cepat. Itu jawaban pertanyaan pemantik pertemuan lalu.

Langkah 3: *blocking* (*tiling*)

```
#define BS 64     /* coba 32, 64, 128 */
#define MIN(a, b) ((a) < (b) ? (a) : (b))

for (int ii = 0; ii < n; ii += BS)
  for (int kk = 0; kk < n; kk += BS)
    for (int jj = 0; jj < n; jj += BS)
      for (int i = ii; i < MIN(ii+BS, n); i++)
        for (int k = kk; k < MIN(kk+BS, n); k++) {
          double a = A[i*n + k];
          for (int j = jj; j < MIN(jj+BS, n); j++)
            C[i*n + j] += a * B[k*n + j];
        }
```



- Kerjakan matriks per blok $BS \times BS$ yang muat di *cache* (3 blok *double* $64 \times 64 \approx 96$ KB, kira-kira ukuran L2).
- Blok yang sudah dimuat dipakai berulang sebelum dibuang: *temporal locality*, *arithmetic intensity* naik.
- Ukuran blok terbaik bergantung CPU; tentukan dengan eksperimen.

Langkah 4: OpenMP, lalu ukur GFLOPS

```c
#pragma omp parallel for schedule(static)
for (int ii = 0; ii < n; ii += BS)
  for (int kk = 0; kk < n; kk += BS)
    /* ... isi sama dengan versi blok ... */

/* pengukuran */
double t0 = omp_get_wtime();
matmul(n, A, B, C);
double t = omp_get_wtime() - t0;
double gflops = 2.0 * n * n * n / t / 1e9;
printf("%d,%d,%.3f,%.2f\n", n,
       omp_get_max_threads(), t, gflops);
```

- Paralelkan *loop* terluar `ii`: tiap *thread* menulis baris C yang berbeda, sehingga tidak ada *race*.
- GFLOPS $= \dfrac{2n^3}{t \cdot 10^9}$. Contoh $n = 2048$: $2n^3 \approx 1{,}72 \times 10^{10}$ FLOP; bila selesai dalam 2 detik, berarti $\approx 8{,}6$ GFLOPS.
- Selalu periksa kebenaran: bandingkan C dengan versi naif pada n kecil (selisih maksimum $< 10^{-9}$).
- Pembanding akhir: *library* BLAS (OpenBLAS, `numpy.dot`). Seberapa dekat versi Anda?

Tabel yang harus Anda isi dan jelaskan

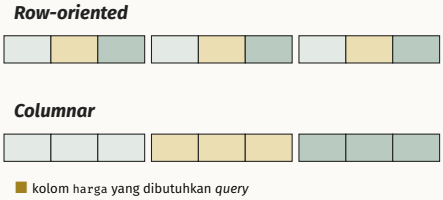
Versi	Waktu $n = 2048$	GFLOPS	% puncak	Penjelasan utama
1. `ijk`, `-O2`	...	...	...	...
2. `ikj`, `-O2`	...	...	...	...
3. `ikj`, `-O3 -march=native`	...	...	...	...
4. ber-blok	...	...	...	...
5. ber-blok + OpenMP (*p thread*)	...	...	...	...
6. OpenBLAS / NumPy	...	...	...	...

Ringkasan

Setiap baris harus disertai **alasan** dari materi hari ini: *locality*, *vectorization*, *cache reuse*, paralelisme, atau batas *roofline*. Angka tanpa penjelasan tidak dinilai.

Kaitan dengan Data Engineering

Mengapa format *columnar* dan *vectorized execution* cepat



`SELECT avg(harga) FROM penjualan`: pada format *columnar*, semua nilai `harga` bersebelahan.

- ***Cache line* penuh data berguna** — tidak ikut memuat kolom lain (*spatial locality*).
- **SIMD** — nilai satu tipe yang berurutan bisa dijumlahkan 8 atau 16 sekaligus.
- ***Compression*** — nilai sejenis mudah dimampatkan, sehingga byte yang dibaca dari disk dan memori lebih sedikit (*memory bound* jadi lebih ringan).
- ***Vectorized execution*** — mesin *query* memproses *batch* ribuan nilai per panggilan fungsi, bukan satu baris per panggilan: *overhead function call* dan *branch* berkurang.

Contoh: Parquet dan ORC (penyimpanan), Apache Arrow (memori), DuckDB dan mesin *query* modern lain. Gagasan *vectorized execution* berasal dari riset MonetDB/X100 (Boncz dkk., CIDR 2005).

Ringkasan

Di Data Engineering (P2 dan P5) Anda sudah **mengukur** bahwa Parquet, Arrow, DuckDB, dan Polars cepat. Di sini Anda tahu **mengapa**: *cache*, SIMD, dan *bandwidth*.

Praktikum

Langkah praktikum

1. Kenali mesin: `lscpu`, `lscpu -C`, `numactl –hardware`. Hitung FLOPS puncak seperti pertemuan 9.
2. Tulis `mm.c` dengan versi 1–5 pada tabel; pilih versi lewat argumen.
3. Uji kebenaran pada $n = 256$ terhadap versi naif.
4. Ukur $n = 1024$ dan 2048, 5 ulangan, laporkan median.
5. Versi 5: ukur $p = 1, 2, 4, \ldots$ *thread*; gambar *speedup* seperti pertemuan 9.
6. Lihat *cache miss* versi 1 vs 2 dengan `perf stat`.
7. Tempatkan setiap versi pada grafik *roofline* mesin Anda.
8. Bonus: jalankan contoh *false sharing* dan *branch prediction*.

```
gcc -O2 -fopenmp mm.c -o mm_o2   # v1-2
gcc -O3 -march=native -fopenmp \
    mm.c -o mm                   # v3-5
./mm 2 256 --cek          # versi 2, cek
for v in 1 2 3 4; do
  b=mm; [ $v -le 2 ] && b=mm_o2
  OMP_NUM_THREADS=1 ./$b $v 2048
done
for p in 1 2 4 8; do
  OMP_NUM_THREADS=$p \
  OMP_PROC_BIND=close ./mm 5 2048
done
perf stat -e cycles,instructions,\
cache-misses ./mm_o2 1 1024
```

Pada WSL, `perf` kadang tidak tersedia; laporkan bila begitu dan pakai pengukuran waktu saja.

Penutup

Ringkasan

- Frekuensi berhenti naik; performa kini datang dari banyak *core*, SIMD, dan pemakaian *cache* yang baik.
- Satu *core* sudah paralel di dalam: *pipeline*, *superscalar*, *out-of-order*; percabangan yang sulit ditebak memacetkannya.
- DRAM ratusan kali lebih lambat dari aritmetika (*memory wall*); *cache line*, *locality*, dan pemakaian ulang data menentukan kecepatan.
- Banyak *core* membawa masalah baru: koherensi (MESI), *false sharing*, NUMA.
- *Roofline*: $P = \min(P_{\text{puncak}}, I \times BW)$ memberi tahu apakah kode dibatasi hitungan atau memori.
- OpenMP: `parallel for`, `reduction`, `schedule`; selalu periksa *race condition* dan kebenaran hasil.
- Perkalian matriks: urutan *loop*, *blocking*, *vectorization*, lalu paralelisme; tiap langkah punya alasan yang bisa dijelaskan.

Tugas untuk pertemuan berikutnya

1. Selesaikan praktikum perkalian matriks: tabel 6 versi, grafik *speedup* versi 5, grafik *roofline*, dan penjelasan tiap baris (3–4 halaman).
2. Baca Hennessy & Patterson Bab 2 (hierarki memori) dan Pacheco & Malensek Bab 5 (OpenMP).
3. Pasang MPI di laptop (`sudo apt install openmpi-bin libopenmpi-dev` atau MPICH) dan pastikan `mpirun -np 2 hostname` berjalan.
4. Pertanyaan pemantik: bila satu mesin sudah memakai semua *core*-nya, bagaimana cara membagi perkalian matriks ke **empat mesin** yang tidak berbagi memori?

Pertemuan depan

Kita keluar dari satu mesin: *HPC cluster*, MPI (*send*, *receive*, *collective*), *communication cost model*, InfiniBand dan RDMA, serta *scheduler* Slurm.

Terima kasih

Pertanyaan?