

Pertemuan 14 — Pola Paralel Tingkat Tinggi

Task dan *actor* terdistribusi dengan Dask dan Ray, serta I/O paralel di kluster

Hendri Karisma

Semester Ganjil 2026/2027

Program Studi Teknik Informatika, STMIK Tazkia

Alur pertemuan hari ini

1. Dari MPI ke abstraksi yang lebih tinggi
2. *Structured parallel patterns*
3. *Dynamic scheduling* dan *work stealing*
4. Dask: *task graph* di banyak mesin
5. Ray: *task*, *actor*, dan *object store*
6. *Distributed actors*: lanjutan Po6
7. MapReduce dan Spark: cukup disebut
8. *Parallel I/O* dan *checkpoint-restart*
9. *Strong* dan *weak scaling*
10. Praktikum: Ray dan Dask di beberapa *worker*

Sasaran pertemuan ini

Anda dapat mengenali pola paralel dalam sebuah soal, memilih antara MPI, Dask, dan Ray sesuai bentuk soalnya, menulis program *task* dan *actor* yang berjalan di beberapa mesin, dan mengukur *scaling*-nya dengan jujur — termasuk saat satu *worker* mati.

Dari MPI ke abstraksi yang lebih tinggi

Pertemuan 11: MPI itu cepat, tetapi semuanya urusan Anda

Yang Anda tulis sendiri di MPI

- ▶ Membagi data ke setiap *rank*
- ▶ Mengirim dan menerima pesan, atau memanggil *collective*
- ▶ Menyeimbangkan beban bila pekerjaan tidak rata
- ▶ Menangani kegagalan: biasanya **tidak ditangani**; satu *rank* mati, seluruh *job* berhenti

Yang diambil alih *framework* tingkat tinggi

- ▶ Pembagian pekerjaan menjadi *task* kecil
- ▶ Penjadwalan ke *worker* yang sedang kosong
- ▶ Pemindahan data antar-*worker*
- ▶ Mengulang *task* yang gagal secara otomatis

Intinya

Ongkosnya: *overhead* penjadwalan per *task* (orde ratusan mikrodetik sampai milidetik, tergantung *framework*) dan kendali yang lebih sedikit. MPI tetap raja untuk komputasi padat yang rapi; *framework task* unggul saat pekerjaannya tidak rata, dinamis, atau harus tahan gagal.

Tangga abstraksi

makin
tinggi



Library khusus domain: PyTorch DDP, Ray Train P15

Dataflow data: MapReduce, Spark dibahas di Data Engineering

Task dan actor: Dask, Ray, Akka/Pekko pekerjaan dinamis

Thread, socket, MPI kendali penuh, beban penuh

Makin ke atas, makin sedikit kode yang Anda tulis untuk paralelisme dan kegagalan, tetapi makin sempit bentuk soal yang cocok. Hari ini kita berdiri di anak tangga kedua.

Structured parallel patterns

Tujuh pola yang muncul berulang-ulang

Pola	Bentuknya	Contoh
Map	Fungsi yang sama untuk tiap elemen, saling bebas	Ubah ukuran 10.000 citra
Reduce	Gabungkan banyak nilai jadi satu dengan operator asosiatif	Jumlah, maksimum, histogram
Scan	Jumlah berjalan (<i>prefix sum</i>)	Saldo berjalan, <i>stream compaction</i>
Stencil	Nilai baru dari tetangga di sekitarnya	Konvolusi citra, simulasi panas
Task farm	<i>Master</i> membagi tugas, <i>worker</i> mengerjakan	Banyak simulasi dengan parameter beda
Pipeline	Tahap berurutan, tiap tahap paralel dengan tahap lain	Baca → olah → tulis
Fork-join	Pecah rekursif, kerjakan paralel, gabungkan	<i>Merge sort</i> , <i>quicksort</i> paralel

Sumber: McCool, Robison, dan Reinders, *Structured Parallel Programming*, 2012.

Mengapa pola penting: Anda sudah memakainya

- ▶ **P10 OpenMP:** `parallel for` adalah *map*; `reduction(+:sum)` adalah *reduce*.
- ▶ **P11 MPI:** `MPI_Scatter` lalu `MPI_Reduce` adalah *map-reduce*; *halo exchange* adalah *stencil*.
- ▶ **P12–P13 CUDA:** *reduction* dan *scan* di *shared memory*; konvolusi adalah *stencil*.
- ▶ **Hari ini:** *task farm* dan *fork-join* di banyak mesin.

Cara pakai pola

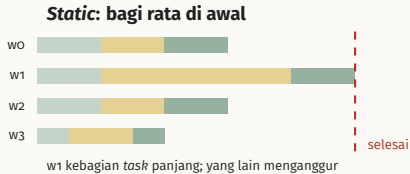
1. Kenali polanya dari soal
2. Cek ketergantungan data: bebas, tetangga, atau berurutan?
3. Pilih alat yang menyediakan pola itu
4. Ukur: apakah biaya komunikasi lebih kecil dari komputasi?

Catatan

Operator *reduce* harus asosiatif agar boleh dikerjakan dalam urutan apa pun. Penjumlahan *floating point* hanya “hampir” asosiatif, jadi hasil paralel bisa berbeda di digit terakhir.

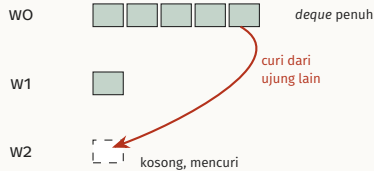
Dynamic scheduling dan work stealing

Pembagian statis vs dinamis



- **Static scheduling:** setiap *worker* mendapat bagian yang sama sejak awal. Murah, tetapi buruk bila lama tiap *task* tidak rata.
- **Dynamic scheduling:** *task* dibagikan saat *worker* kosong. Beban lebih rata, ada ongkos koordinasi.
- Ingat P10: `schedule(static)` vs `schedule(dynamic)` di OpenMP adalah pilihan yang sama di satu mesin.
- Ingat P9: *load imbalance* langsung memotong *speedup*.

Work stealing: worker yang mengganggu mencuri

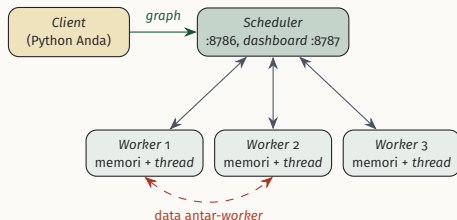


- ▶ Tiap *worker* punya *deque* (antrean dua ujung) sendiri. Ia mengambil *task* dari ujungnya sendiri.
- ▶ *Worker* yang kosong mencuri dari ujung **lain** milik *worker* acak. Tabrakan jarang terjadi.
- ▶ Dirumuskan untuk Cilk (Blumofe dan Leiserson); dipakai di Go *scheduler*, Java *ForkJoinPool*, Rust Tokio dan Rayon, Intel TBB.
- ▶ Di klaster, Dask juga memindahkan *task* dari *worker* sibuk ke *worker* yang mengganggu.

Sumber: Blumofe dan Leiserson, *Scheduling Multithreaded Computations by Work Stealing*, JACM, 1999.

Dask: *task graph* di banyak mesin

Dask dalam satu gambar



- ▶ Kode Anda membangun **task graph**: *node* = fungsi, *sisi* = ketergantungan data. Belum ada yang dihitung (*lazy*).
- ▶ **Scheduler** membagi *task* ke *worker*, mengutamakan *worker* yang sudah memegang datanya.
- ▶ Hasil antara tinggal di memori *worker*; dipindah antar-*worker* hanya bila perlu.
- ▶ API akrab: `dask.array` (mirip NumPy), `dask.dataframe` (mirip pandas), `dask.delayed` dan `futures` untuk kode bebas.

Sumber: Rocklin, *Dask: Parallel Computation with Blocked Algorithms and Task Scheduling*, SciPy, 2015.

dask.delayed: mengubah fungsi biasa jadi *task graph*

```
import dask
from dask.distributed import Client

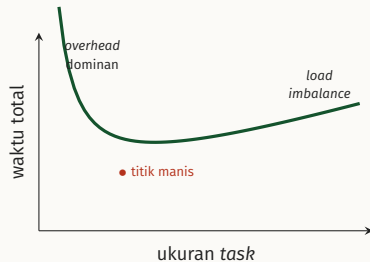
client = Client("tcp://scheduler:8786")

@dask.delayed
def baca(path):    return muat_citra(path)
@dask.delayed
def olah(img):     return ekstrak_fitur(img)
@dask.delayed
def gabung(xs):    return rangkum(xs)

fitur = [olah(baca(p)) for p in daftar_berkas]
hasil = gabung(fitur)           # belum dihitung
hasil.visualize("graph.png")    # lihat graph-nya
print(hasil.compute())         # baru dijalankan
```

- ▶ Pola: *map* (baca, olah per berkas) lalu *reduce* (gabung).
- ▶ `visualize()` menggambar *task graph*: berguna untuk mengecek apakah paralelismenya sesuai dugaan.
- ▶ Buka *dashboard* di `:8787`: tampak *task stream*, memori tiap *worker*, dan *progress*.
- ▶ Untuk pekerjaan yang muncul dinamis, pakai `client.submit` dan `client.map (futures)`.

Ukuran *task*: terlalu kecil atau terlalu besar sama-sama rugi

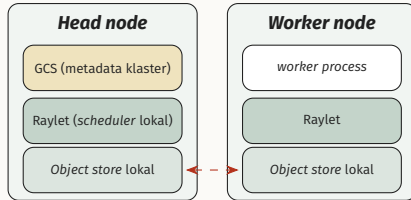


Sumber: Kurva ilustrasi.

- ▶ Setiap *task* membawa ongkos tetap: penjadwalan, serialisasi, catatan di *scheduler*.
- ▶ *Task* terlalu kecil: *scheduler* jadi *bottleneck*.
- ▶ *Task* terlalu besar: sedikit *task*, sebagian *worker* menganggur, dan memori satu *worker* bisa penuh.
- ▶ Aturan praktis: buat *task* yang bekerja orde ratusan milidetik atau lebih, dan jumlahnya beberapa kali lipat jumlah *core*.
- ▶ Ukur sendiri; titik manisnya bergantung soal dan klaster.

Ray: *task*, *actor*, dan *object store*

Ray dalam satu gambar



- ▶ **Task**: fungsi tanpa keadaan yang dijalankan di mana saja (`@ray.remote def`).
- ▶ **Actor**: objek berkeadaan yang hidup di satu *worker* (`@ray.remote class`).
- ▶ **Object store**: memori bersama per *node*; hasil besar disimpan sekali dan dibaca tanpa salin di *node* yang sama.
- ▶ Penjadwalan **bertingkat**: Raylet lokal mencoba dulu, baru melempar ke *node* lain. Ini yang membuat Ray sanggup menangani jutaan *task* kecil.

Sumber: Moritz dkk., *Ray: A Distributed Framework for Emerging AI Applications*, OSDI, 2018.

Ray task: remote, ObjectRef, dan ray.get

```
import ray, random
ray.init(address="auto")      # sambung ke klaster

@ray.remote
def sampel_pi(n):
    masuk = 0
    for _ in range(n):
        x, y = random.random(), random.random()
        masuk += (x*x + y*y <= 1.0)
    return masuk

N, T = 10_000_000, 64
refs = [sampel_pi.remote(N // T) for _ in range(T)]
total = sum(ray.get(refs))    # tunggu semua
print(4 * total / N)
```

- ▶ `.remote(...)` langsung kembali dengan `ObjectRef`, semacam “janji” hasil (*future*). Pekerjaannya berjalan di latar.
- ▶ `ray.get` menunggu hasil. Panggil sesudah semua *task* dikirim, jangan di dalam *loop* — itu membuatnya serial lagi.
- ▶ `ObjectRef` bisa diberikan ke *task* lain tanpa `get`: Ray merangkai ketergantungannya sendiri.
- ▶ Pola: *task farm + reduce*. Satu *task* di sini bekerja orde detik, ukuran yang sehat.

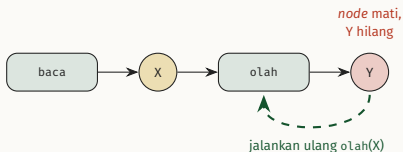
Ray actor: keadaan yang hidup di kluster

```
@ray.remote(max_restarts=2, max_task_retries=1)
class Penghitung:
    def __init__(self):
        self.n = 0
    def tambah(self, k=1):
        self.n += k
        return self.n
    def nilai(self):
        return self.n

p = Penghitung.remote()          # actor di satu worker
refs = [p.tambah.remote() for _ in range(1000)]
ray.get(refs)
print(ray.get(p.nilai.remote())) # 1000
```

- ▶ Panggilan ke satu *actor* diproses **satu per satu** sesuai urutan dari pemanggil yang sama — persis *actor model* di PO6, tanpa *lock*.
- ▶ Cocok untuk keadaan bersama: *cache*, model yang sudah dimuat, penghitung, koneksi ke basis data.
- ▶ `max_restarts`: *actor* dihidupkan ulang bila prosesnya mati, tetapi keadaan di memorinya **hilang**. Simpan *checkpoint* sendiri bila keadaan itu penting.
- ▶ Satu *actor* adalah satu titik serial; perbanyak *actor* bila menjadi *bottleneck*.

Ownership dan pemulihan lewat lineage



- Pemanggil yang membuat sebuah *task* menjadi **owner** hasilnya: ia tahu lokasi objek dan cara membuatnya ulang.
- Bila objek hilang karena *node* mati, *owner* menjalankan ulang *task* pembuatnya (**lineage reconstruction**).
- *Task* biasa dicoba ulang otomatis bila *worker*-nya mati (`max_retries`).
- Syaratnya: *task* harus **deterministik** dan bebas efek samping, sama seperti *idempotency* di P4.

Sumber: Wang dkk., *Ownership: A Distributed Futures System for Fine-Grained Tasks*, NSDI, 2021.

Placement group: mengatur di mana sumber daya diletakkan

```
from ray.util.placement_group import placement_group
from ray.util.scheduling_strategies import \
    PlacementGroupSchedulingStrategy

pg = placement_group([{"CPU": 2}] * 4,
                     strategy="SPREAD")
ray.get(pg.ready())

@ray.remote(num_cpus=2)
def kerja(i): ...

refs = [kerja.options(
    scheduling_strategy=
        PlacementGroupSchedulingStrategy(pg)
).remote(i) for i in range(4)]
```

- PACK: kumpulkan di sesedikit mungkin *node*. Komunikasi murah, cocok untuk *distributed training* (P15).
- SPREAD: sebar ke banyak *node*. Tahan gagal: satu *node* mati tidak membawa semuanya.
- STRICT_PACK dan STRICT_SPREAD: versi wajib, gagal bila tidak terpenuhi.
- Ini wujud nyata *fault domain* dari P1 dan P7: penempatan menentukan apa yang ikut mati bersama.

Dask atau Ray?

	Dask	Ray
Kekuatan utama	Array dan <i>dataframe</i> besar dengan API mirip NumPy/pandas	<i>Task</i> kecil dalam jumlah sangat banyak, <i>actor</i> berkeadaan
<i>Scheduler</i>	Terpusat, dengan <i>work stealing</i>	Bertingkat (lokal dulu, lalu global)
Keadaan	Kebanyakan tanpa keadaan; ada <i>actor</i> terbatas	<i>Actor</i> sebagai warga kelas satu
Ekosistem	SciPy, xarray, analitik ilmiah	AI: Ray Train, Tune, Serve, Data
Cocok untuk	Analitik numerik melebihi memori satu mesin	Simulasi, <i>hyperparameter search</i> , <i>servings</i> , <i>pipeline</i> AI

Intinya

Bukan soal mana yang lebih baik, tetapi bentuk soal Anda. Kalau soalnya “tabel atau *array* raksasa”, mulai dari Dask. Kalau soalnya “banyak pekerja kecil dengan keadaan”, mulai dari Ray. Kalau soalnya komputasi padat yang rapi, MPI masih juaranya.

Distributed actors: lanjutan PO6

Dari PO6: *actor* di satu mesin ke banyak mesin

Yang sudah kita bahas di PO6

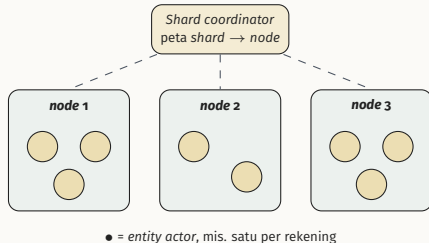
- ▶ *Actor* = keadaan pribadi + *mailbox*, satu pesan pada satu waktu
- ▶ Di satu mesin: jutaan *actor* di atas *thread pool*
- ▶ *Location transparency*: alamat sama, lokal atau jarak jauh

Pertanyaan baru di banyak mesin

- ▶ *Actor* untuk rekening #123 tinggal di *node* mana?
- ▶ Bagaimana pesan menemukannya?
- ▶ Apa yang terjadi saat *node*-nya mati?
- ▶ Bagaimana mencegah **dua** *actor* untuk rekening yang sama?

Tiga jawaban industri: **Akka/Pekko Cluster Sharding**, **Orleans virtual actor**, dan **Ray actor**.

Akka/Pekko Cluster Sharding



- ▶ Setiap *entity* punya ID (mis. nomor rekening). ID di-hash ke **shard**; *shard* ditempatkan di satu *node*.
- ▶ Pengirim cukup menyebut ID; *shard region* di tiap *node* meneruskan pesan ke *node* yang benar.
- ▶ *Node* mati → *shard*-nya dipindah ke *node* lain dan *entity* dibuat ulang di sana. Keadaan dipulihkan dari *persistence* (*event sourcing*) bila diaktifkan.
- ▶ Agar tidak ada dua salinan saat jaringan terbelah, kluster memakai **split brain resolver** — ingat partisi jaringan dan CAP di P7.

Orleans: *virtual actor*

- ▶ Dari Microsoft Research; dipakai untuk layanan berskala besar (misalnya *backend* gim daring).
- ▶ Actor di Orleans disebut **grain**, berjalan di **silo** (*server*).
- ▶ Gagasan kuncinya: *grain* **selalu ada secara logis**. Anda tidak membuat atau menghapusnya; cukup memanggil berdasarkan ID.
- ▶ *Runtime* mengaktifkan *grain* saat dipanggil, menonaktifkannya saat lama menganggur, dan mengaktifkan ulang di *silo* lain bila *silo*-nya mati.

Bandingkan dengan Po6

Akka klasik

Anda membuat dan mengawasi *actor*

Pohon *supervisor*
Penempatan diatur aplikasi

Orleans

Runtime yang mengelola siklus aktif

Aktivasi ulang otomatis
Penempatan diatur *runtime*

Sumber: Bykov dkk., *Orleans: Cloud Computing for Everyone*, ACM SoCC, 2011.

Tiga *distributed actor*, satu tabel

	Akka/Pekko Cluster	Orleans	Ray actor
Bahasa Penempatan	Scala, Java <i>Shard coordinator</i>	C#/.NET <i>Runtime</i> otomatis	Python (juga Java, C++) <i>Scheduler</i> Ray, <i>placement group</i>
<i>Node</i> mati	<i>Shard</i> pindah, <i>entity</i> dibuat ulang	<i>Grain</i> diaktifkan ulang	<i>max_restarts</i>
Keadaan tahan gagal Cocok untuk	<i>Event sourcing</i> (opsional) Sistem transaksi, IoT, layanan berkeadaan	<i>Grain storage</i> <i>Backend</i> interaktif	<i>Checkpoint</i> sendiri AI, simulasi, <i>serving</i> model

Yang tidak berubah dari Po6

Pesan antar-mesin bisa hilang atau terlambat. Akka secara bawaan menjanjikan *at-most-once delivery*; jaminan lebih kuat dibangun aplikasi dengan *ack*, *retry*, dan penanganan pesan ganda (*idempotency*). Akka berlisensi BSL sejak 2022; Apache Pekko adalah *fork* berlisensi Apache 2.0.

MapReduce dan Spark: cukup disebut

MapReduce dan Spark: pola *map-reduce* untuk data

Dua nama ini pasti Anda dengar. Di peta pola hari ini, keduanya adalah:

- ▶ **Map** di setiap potongan data, lalu **shuffle** (komunikasi *all-to-all*, mirip `MPI_Alltoall` di P11), lalu **reduce**.
- ▶ Tahan gagal lewat *lineage*: menjalankan ulang potongan yang hilang — gagasan yang sama dengan Ray tadi.
- ▶ *Scheduler task* dengan *data locality*: kirim program ke data.

Dibahas tuntas di Data Engineering

HDFS, ekosistem Hadoop, MapReduce, dan Spark (termasuk RDD, DAG, *lineage*, dan *query plan*) adalah materi Data Engineering (HDFS di P3, Hadoop dan Spark di P9–P10). Di sini cukup tahu posisinya di tangga abstraksi: anak tangga *dataflow* data.

Intinya

Dask dan Ray adalah *framework task* umum; Spark adalah *framework dataflow* khusus data tabular. Pola dasarnya sama.

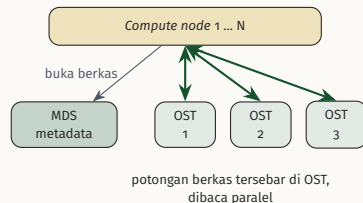
Parallel I/O dan checkpoint-restart

Masalah yang sering dilupakan: membaca dan menulis data

- ▶ 1.000 proses yang masing-masing membuka berkas kecil di satu *server* NFS = antrean panjang di satu titik.
- ▶ Komputasi bisa diskalakan; *storage* sering tidak, kecuali dirancang paralel.
- ▶ Gejalanya: *speedup* bagus untuk komputasi, lalu mendatar karena semua menunggu disk.

Parallel file system

Lustre dan **GPFS** (kini IBM Storage Scale) memecah satu berkas ke banyak *storage server* (**striping**), sehingga banyak proses membaca dan menulis bersamaan.



Arsitektur Lustre: *metadata* (MDS) terpisah dari data (OSS/OST), mirip pemisahan NameNode dan DataNode yang Anda temui di Data Engineering.

- ▶ MPI-IO: setiap *rank* membaca atau menulis bagiannya dari **satu** berkas bersama, bukan satu berkas per *rank*.
- ▶ **Collective I/O** (`MPI_File_write_all`): beberapa *rank* menjadi *aggregator* yang menggabungkan potongan kecil menjadi tulisan besar berurutan. Disk menyukai tulisan besar berurutan.
- ▶ *Library* di atasnya: HDF5 paralel, NetCDF — format standar data ilmiah dan cuaca.

Aturan praktis I/O di kluster

1. Sedikit berkas besar lebih baik daripada banyak berkas kecil
2. Tulis besar dan berurutan, hindari acak dan kecil
3. Jangan semua proses membuka berkas yang sama di saat bersamaan tanpa koordinasi
4. Ukur I/O terpisah dari komputasi

Masalah “banyak berkas kecil” yang sama juga muncul di HDFS dan *object storage*.

Checkpoint-restart: jembatan ke P15

- ▶ *Job* yang berjalan sehari-hari di ratusan *node* hampir pasti mengalami kegagalan.
- ▶ **Checkpoint**: simpan keadaan secara berkala ke *parallel file system*. **Restart**: lanjutkan dari *checkpoint* terakhir, bukan dari awal.
- ▶ Terlalu sering → waktu habis untuk menulis; terlalu jarang → banyak kerja yang diulang.

Rumus Young (perkiraan pertama)

Interval *checkpoint* optimal $\approx \sqrt{2CM}$, dengan C = lama menulis satu *checkpoint* dan M = MTBF klaster.

Contoh hitungan (ilustrasi)

$C = 5$ menit, $M = 24$ jam = 1.440 menit:

$$\sqrt{2 \times 5 \times 1.440} = \sqrt{14.400} = 120 \text{ menit}$$

Checkpoint kira-kira tiap 2 jam.

Intinya

Ingat P1: MTBF klaster turun saat jumlah *node* naik. Klaster dua kali lebih besar berarti *checkpoint* harus lebih sering. Di P15 kita lihat ini pada *training* model AI di ribuan GPU.

Sumber: Young, *A First Order Approximation to the Optimum Checkpoint Interval*, CACM, 1974.

Strong dan weak scaling

Dua cara mengukur *scaling*

Strong scaling (Amdahl)

Ukuran soal **tetap**, jumlah *worker* ditambah.
Pertanyaannya: seberapa cepat soal yang sama selesai?

Ideal: p *worker* $\rightarrow$ waktu T_1/p .

$$\text{Efisiensi } E_p = \frac{T_1}{p T_p}.$$

Weak scaling (Gustafson)

Ukuran soal **ikut membesar** sebanding jumlah *worker*.
Pertanyaannya: bisakah kita menyelesaikan soal p kali lebih besar dalam waktu yang sama?

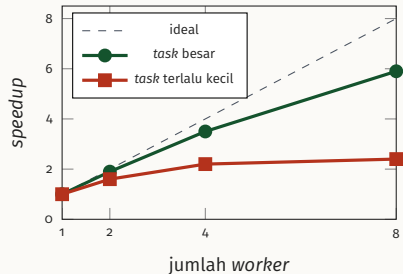
Ideal: waktu **tetap** saat p naik.

$$\text{Efisiensi } E_p = \frac{T_1}{T_p}.$$

Worker	Strong: sampel Monte Carlo total	Weak: sampel per worker
1, 2, 4, 8	10^8 tetap	$2,5 \times 10^7$ tetap (total $2,5 \times 10^7 \cdot p$)

Sumber: Kembali ke Pg: Amdahl untuk *strong scaling*, Gustafson untuk *weak scaling*.

Membaca grafik *scaling* dengan jujur



Sumber: Angka ilustrasi; ganti dengan hasil ukur Anda.

- ▶ Selalu tampilkan garis ideal sebagai pembanding.
- ▶ Kurva mendatar? Cari penyebabnya: *overhead scheduler*, data yang dipindah antar-*worker*, I/O, atau bagian serial.
- ▶ Laporkan median dari beberapa kali jalan, bukan sekali jalan terbaik.
- ▶ Pisahkan waktu: kirim data, komputasi, kumpulkan hasil. *Dashboard* Dask dan Ray membantu melihatnya.
- ▶ Satu mesin dengan banyak *container* bukan kluster sungguhan: semua berbagi CPU dan memori yang sama. Tuliskan batasan ini di laporan.

Benang merah: pola yang sama di GPU

- ▶ Dask dan Ray bisa menjadwalkan *task* ke GPU: `num_gpus=1` di Ray, `dask-cuda` di Dask.
- ▶ RAPIDS (cuDF, cuML) menjalankan operasi tabel dan ML di GPU, dan bisa disebar ke banyak GPU lewat Dask.
- ▶ Polanya tetap: *map* di tiap potongan, *reduce* di akhir, komunikasi sesedikit mungkin.

Yang dibawa dari P12–P13

GPU tidak otomatis lebih cepat. Bila *task* kecil atau datanya harus bolak-balik CPU–GPU lewat PCIe, waktu transfer bisa menelan keuntungannya. Ukur dengan dan tanpa biaya transfer, seperti di P12.

Praktikum

Praktikum: klaster Ray dan Dask kecil dengan Docker

```
services:
  ray-head:
    image: rayproject/ray:latest
    command: ray start --head --port=6379
              --dashboard-host=0.0.0.0 --block
    ports: ["8265:8265"]
    shm_size: 2gb
  ray-worker:
    image: rayproject/ray:latest
    command: ray start --address=ray-head:6379 --block
    shm_size: 2gb
    deploy: { replicas: 3 }
  dask-scheduler:
    image: ghcr.io/dask/dask:latest
    command: dask scheduler
    ports: ["8786:8786", "8787:8787"]
  dask-worker:
    image: ghcr.io/dask/dask:latest
    command: dask worker tcp://dask-scheduler:8786
    deploy: { replicas: 3 }
```

- ▶ `docker compose up --scale ray-worker=N` untuk mengubah jumlah *worker* (1, 2, 4).
- ▶ *Dashboard*: Ray di `localhost:8265`, Dask di `localhost:8787`.
- ▶ `shm_size` perlu cukup besar karena *object store* Ray memakai memori bersama.
- ▶ Pakai versi *image* yang sama untuk semua *node*, dan versi Python klien yang cocok dengan *image*.
- ▶ Nama dan versi *image* disesuaikan dengan repositori kelas.

Praktikum: eksperimen yang diukur

1. **Monte Carlo π dengan Ray task:** ukur *strong scaling* (10^8 sampel tetap) dan *weak scaling* ($2,5 \times 10^7$ sampel per *worker*) untuk 1, 2, 4 *worker*. Coba 8, 64, dan 4.096 *task*; catat pengaruh ukuran *task*.
2. **Banyak berkas dengan Dask:** proses 500 berkas citra atau CSV kecil (*map*), lalu rangkum (*reduce*). Bandingkan `dask.delayed` dengan satu proses Python biasa.
3. **Ray actor berkeadaan:** buat Penghitung atau *cache* hasil. Kirim 10.000 panggilan dari banyak *task*; ukur *throughput*-nya, lalu bagi menjadi 4 *actor* dan ukur lagi.
4. **Uji kegagalan:** saat eksperimen 1 berjalan, matikan satu *worker* (`docker kill`). Apakah hasil akhirnya tetap benar? Berapa tambahan waktunya? Lakukan juga pada *actor*: apa yang terjadi pada nilai penghitungnya?

Isi laporan

Tabel dan grafik *strong* dan *weak scaling* (dengan garis ideal), grafik waktu terhadap jumlah *task*, tangkapan *dashboard* saat satu *worker* dimatikan, dan penjelasan mengapa nilai *actor* hilang atau tidak.

Tugas untuk pertemuan berikutnya

1. Kumpulkan laporan praktikum: grafik *scaling*, pengaruh ukuran *task*, dan hasil uji kegagalan.
2. Baca makalah Ray (Moritz dkk., OSDI 2018) bagian 3–4, dan dokumentasi Ray tentang *fault tolerance*.
3. Lanjutkan tugas besar UAS. Bila memilih Ray atau Dask untuk tahap T2 (banyak *node*), versi itu harus sudah berjalan dan terukur.

Pertemuan depan

Titik temu kedua jalur: melatih model AI di banyak GPU — *data*, *tensor*, dan *pipeline parallelism*, *all-reduce*, dan bertahan dari kegagalan harian dengan *checkpoint* pada skala ribuan akselerator.

Terima kasih

Pertanyaan?