

Pertemuan 15 — Klaster AI dan *Distributed Training*

Titik temu sistem terdistribusi dan *high performance computing*

Hendri Karisma

Semester Ganjil 2026/2027

Program Studi Teknik Informatika, STMIK Tazkia

Alur pertemuan hari ini

1. Mengapa satu GPU tidak cukup
2. *Data parallelism*: DDP dan *all-reduce*
3. Ring *all-reduce* dan NCCL
4. ZeRO dan FSDP
5. *Tensor parallelism* dan *pipeline parallelism*
6. 3D *parallelism* dan jaringan kluster AI
7. Keandalan pada skala ribuan GPU
8. Biaya, energi, dan jejak karbon
9. Rangkuman dua capaian semester
10. Praktikum DDP dan presentasi progres UAS

Sasaran pertemuan ini

Anda dapat membedakan *data*, *tensor*, dan *pipeline parallelism*, menjelaskan *all-reduce* dan peran NCCL, menjelaskan cara kluster ribuan GPU bertahan dari kegagalan, serta memperkirakan biaya dan energi satu skenario pelatihan.

Mengapa satu GPU tidak cukup

Masalah pertama: memori

Melatih dengan *mixed precision* dan *optimizer* Adam, setiap parameter butuh kira-kira:

Komponen	Byte/parameter
Bobot FP16/BF16	2
Gradien FP16/BF16	2
Salinan bobot FP32 (<i>master weights</i>)	4
Adam: <i>momentum</i> m (FP32)	4
Adam: <i>variance</i> v (FP32)	4
Total	16

Belum termasuk *activation*, yang tumbuh bersama ukuran *batch* dan panjang urutan.

Sumber: Rajbhandari dkk., *ZeRO*, SC 2020.

Model	Memori latih	GPU 80 GB
1 miliar param.	16 GB	muat
7 miliar	112 GB	tidak muat
70 miliar	1,12 TB	≥ 14 GPU
405 miliar	6,5 TB	≥ 81 GPU

Intinya

Model 7 miliar parameter — ukuran “kecil” untuk LLM hari ini — sudah tidak muat dilatih di satu GPU 80 GB, bahkan sebelum menghitung *activation*.

Masalah kedua: waktu

Aturan praktis untuk *transformer*: satu kali pelatihan butuh kira-kira

$$C \approx 6 \cdot N \cdot D \text{ FLOP}$$

dengan N = jumlah parameter dan D = jumlah token data latih.

Contoh: $N = 7 \times 10^9$, $D = 10^{12}$ token

$$C \approx 4,2 \times 10^{22} \text{ FLOP}$$

Bila satu GPU efektif menghasilkan 4×10^{14} FLOP/detik (asumsi):

$$\frac{4,2 \times 10^{22}}{4 \times 10^{14}} \approx 1,05 \times 10^8 \text{ detik} \approx \mathbf{3,3 \text{ tahun}}$$

Intinya

Dengan 1.000 GPU dan *scaling* yang baik, 3,3 tahun menjadi sekitar 1,2 hari. Itulah alasan ekonomi kluster AI: bukan hanya muat, tetapi juga **selesai tepat waktu**.

Faktor 6: sekitar 2 FLOP per parameter per token untuk *forward pass* dan sekitar 4 untuk *backward pass*. Angka ini hampiran kasar, cukup untuk perencanaan.

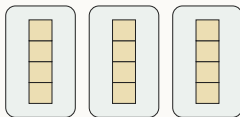
Tiga cara memotong pekerjaan

Data parallelism



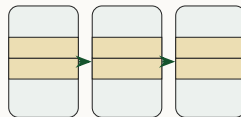
model **disalin** utuh, data dibagi

Tensor parallelism



setiap lapis dibelah
ke beberapa GPU

Pipeline parallelism



kelompok lapis dibagi
berurutan ke GPU

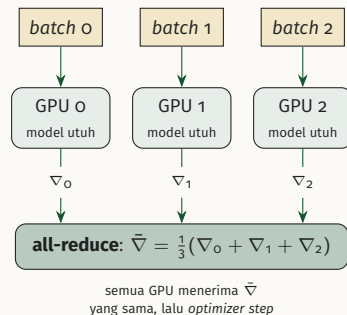
Kotak hijau = GPU, kotak emas = lapis model. Ketiganya bisa digabung; itulah **3D parallelism** yang dipakai melatih model terbesar.

Data parallelism: DDP dan all-reduce

Data parallelism: cara paling sederhana

1. Setiap GPU memegang **salinan utuh** model.
2. Setiap GPU menerima potongan *batch* yang berbeda.
3. Masing-masing menghitung *forward* dan *backward*, menghasilkan gradien lokal.
4. Gradien semua GPU **dirata-rata** lewat **all-reduce**.
5. Semua GPU memperbarui bobot dengan gradien yang sama, sehingga salinannya tetap identik.

Di PyTorch: `DistributedDataParallel (DDP)`. Syaratnya: model harus muat di satu GPU.



All-reduce: operasi kolektif yang paling penting

All-reduce: setiap peserta punya satu vektor; di akhir, setiap peserta memegang **jumlah** (atau rata-rata) semua vektor.

Cara naif: semua kirim ke satu GPU pusat, pusat menjumlah, lalu mengirim balik.

- ▶ GPU pusat menerima $(N - 1)$ kali ukuran gradien
- ▶ *Bandwidth* pusat menjadi *bottleneck*, makin parah ketika N membesar
- ▶ Sama persis dengan masalah “satu server pusat” di Capaian A

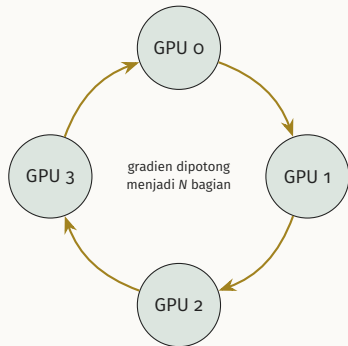
Kita sudah bertemu ini di pertemuan 11

MPI_Allreduce adalah operasi yang sama. Pelatihan AI terdistribusi mewarisi puluhan tahun riset HPC tentang operasi kolektif: *broadcast*, *reduce*, *all-gather*, *reduce-scatter*.

Intinya

Pada DDP, satu langkah pelatihan = komputasi lokal + satu *all-reduce* seukuran seluruh gradien. Kecepatan *all-reduce* menentukan apakah *scaling*-nya baik.

Ring all-reduce



Gradien dibagi menjadi N potongan. Setiap GPU hanya bicara dengan tetangganya.

1. **Reduce-scatter** ($N - 1$ langkah): potongan berputar sambil dijumlahkan; di akhir, tiap GPU memegang satu potongan yang sudah lengkap jumlahnya.
2. **All-gather** ($N - 1$ langkah): potongan lengkap diedarkan sampai semua GPU memegang semuanya.

Data yang dikirim tiap GPU, untuk gradien berukuran S :

$$2 \cdot \frac{N - 1}{N} \cdot S \xrightarrow{N \rightarrow \infty} 2S$$

Hampir **tidak bergantung pada** N : menambah GPU tidak menambah beban kirim tiap GPU. Yang bertambah adalah jumlah langkah, sehingga *latency* per langkah tetap penting.

NCCL: pustaka komunikasi antar GPU

- ▶ **NCCL** (NVIDIA *Collective Communications Library*) menyediakan *all-reduce*, *all-gather*, *reduce-scatter*, *broadcast* untuk GPU.
- ▶ Ia mengenali topologi: NVLink di dalam node, InfiniBand atau RoCE antar node, lalu memilih pola *ring* atau *tree* yang sesuai.
- ▶ Data bergerak langsung GPU ke GPU, tanpa disalin dulu ke memori CPU, bila perangkat keras mendukung (GPUDirect RDMA).
- ▶ PyTorch memakai NCCL lewat `backend="nccl"`; untuk CPU dipakai `backend="gloo"`.

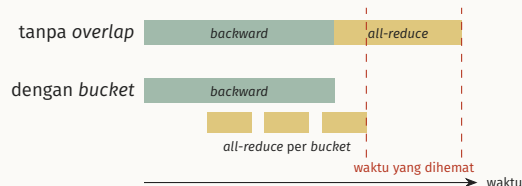
Operasi	Dipakai untuk
<i>All-reduce</i>	Rata-rata gradien (DDP); <i>tensor parallelism</i>
<i>All-gather</i>	Merakit bobot yang dipecah (FSDP/ZeRO-3)
<i>Reduce-scatter</i>	Membagi gradien yang sudah dijumlah (ZeRO-2/3)
<i>Broadcast</i>	Menyamakan bobot awal semua GPU
<i>Send/recv</i>	Mengalirkan <i>activation</i> antar tahap <i>pipeline</i>

Menyembunyikan komunikasi di balik komputasi

DDP tidak menunggu *backward* selesai baru mengirim gradien.

- ▶ Gradien dikelompokkan ke dalam **bucket**.
- ▶ Begitu satu *bucket* siap (lapis-lapis terakhir lebih dulu), *all-reduce* untuk *bucket* itu langsung dimulai.
- ▶ Sementara itu GPU tetap menghitung gradien lapis yang lebih awal.

Ini prinsip yang sama dengan *pipelining* di CPU (pertemuan 10) dan *asynchronous I/O* (pertemuan 6): kerjakan hal lain selagi menunggu.



Intinya

Scaling yang baik bukan berarti komunikasinya nol, tetapi komunikasinya **tersembunyi** di balik komputasi.

Kode: DDP minimal di PyTorch

```
import os, torch, torch.distributed as dist
import torch.nn.functional as F
from torch.nn.parallel import DistributedDataParallel as DDP
from torch.utils.data import DataLoader, DistributedSampler

def main():
    pakai_gpu = torch.cuda.is_available()
    dist.init_process_group(backend="nccl" if pakai_gpu else "gloo")
    local_rank = int(os.environ["LOCAL_RANK"]) # diisi oleh torchrun
    device = torch.device(f"cuda:{local_rank}" if pakai_gpu else "cpu")
    if pakai_gpu:
        torch.cuda.set_device(device)

    model = buat_model().to(device)
    model = DDP(model, device_ids=[local_rank] if pakai_gpu else None)
    sampler = DistributedSampler(dataset) # tiap proses dapat potongan berbeda
    loader = DataLoader(dataset, batch_size=64, sampler=sampler)
    opt = torch.optim.AdamW(model.parameters(), lr=1e-3)

    for epoch in range(5):
        sampler.set_epoch(epoch) # acak ulang secara konsisten
        for x, y in loader:
            x, y = x.to(device), y.to(device)
            loss = F.cross_entropy(model(x), y)
            opt.zero_grad()
            loss.backward() # all-reduce gradien terjadi di sini
            opt.step()
        dist.destroy_process_group()
```

Menjalankan dan menyimpan *checkpoint*

Menjalankan dua proses di satu mesin

```
torchrun --nproc_per_node=2 latih_ddp.py
```

Dua mesin, masing-masing 4 GPU

```
# di setiap mesin
torchrun --nnodes=2 --nproc_per_node=4 \
  --rdzv_backend=c10d \
  --rdzv_endpoint=10.0.0.1:29500 \
  latih_ddp.py
```

torchrun mengisi RANK, LOCAL_RANK, dan WORLD_SIZE, lalu mempertemukan semua proses lewat *rendezvous*.

Simpan dari satu proses saja

```
if dist.get_rank() == 0:
    torch.save({
        "model": model.module.state_dict(),
        "opt": opt.state_dict(),
        "epoch": epoch,
    }, "ckpt.pt")
dist.barrier() # tunggu sampai selesai
```

`model.module` membuka bungkus DDP. Pada DDP semua salinan identik, jadi cukup *rank* 0 yang menyimpan. Pada FSDP setiap *rank* memegang potongan berbeda, sehingga dipakai *sharded checkpoint* (`torch.distributed.checkpoint`).

**Ketika model tidak muat: ZeRO,
FSDP, *tensor* dan *pipeline***

ZeRO dan FSDP: berhenti menyimpan salinan yang sama

Pada DDP, N GPU menyimpan N salinan identik dari *optimizer state*, gradien, dan bobot. Itu boros. **ZeRO** (*Zero Redundancy Optimizer*) memecahnya:

Tahap	Yang dipecah	Memori/GPU
DDP biasa	—	16Ψ
ZeRO-1	<i>optimizer state</i>	$4\Psi + 12\Psi/N$
ZeRO-2	+ gradien	$2\Psi + 14\Psi/N$
ZeRO-3 / FSDP	+ bobot	$16\Psi/N$

Ψ = jumlah parameter; tanpa *activation*.

- ▶ **FSDP** (*Fully Sharded Data Parallel*) adalah versi PyTorch dari gagasan ZeRO-3.
- ▶ Sebelum menghitung satu lapis, bobot lapis itu dirakit dengan *all-gather*; setelah dipakai, dibuang lagi.
- ▶ Gradien dibagi dengan *reduce-scatter*.
- ▶ Harganya: komunikasi lebih banyak daripada DDP.

Intinya

Model 70 miliar parameter butuh 1,12 TB. Dengan ZeRO-3 di 64 GPU: $1,12 \text{ TB}/64 \approx 17,5 \text{ GB per GPU}$.

Tensor parallelism: membelah satu lapis

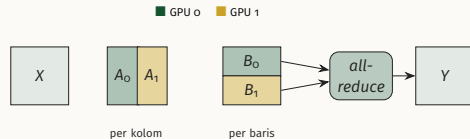
Gagasan Megatron-LM untuk blok MLP

$$Y = \text{GeLU}(XA)B:$$

- ▶ Matriks A dibelah per **kolom**: GPU i menghitung $\text{GeLU}(XA_i)$ tanpa perlu bicara dengan GPU lain.
- ▶ Matriks B dibelah per **baris**: GPU i menghitung $\text{GeLU}(XA_i)B_i$, sebuah hasil parsial.
- ▶ Satu *all-reduce* menjumlahkan hasil parsial menjadi Y .

Hal serupa dilakukan pada *attention*: tiap GPU memegang sebagian *head*.

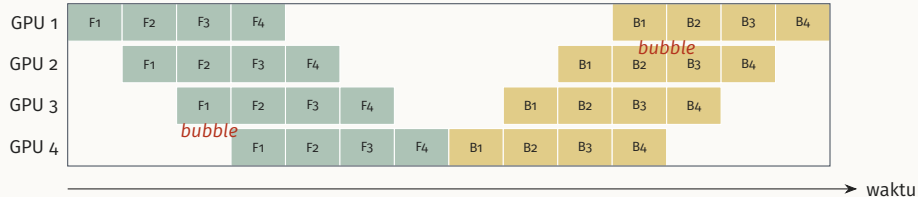
Sumber: Shoeybi dkk., *Megatron-LM*, 2019.



Harganya

Ada *all-reduce* di **setiap lapis**, pada *forward* maupun *backward*. Komunikasinya sangat sering, sehingga *tensor parallelism* biasanya hanya dipakai di dalam satu node yang tersambung NVLink.

Pipeline parallelism: lapis dibagi ke tahap

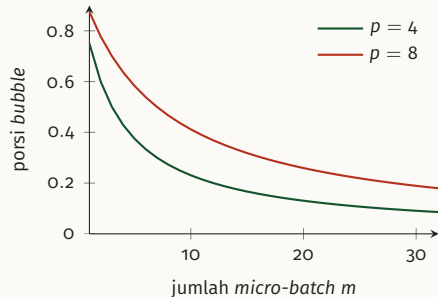


Contoh 4 tahap ($p = 4$) dan 4 *micro-batch* ($m = 4$). F = *forward*, B = *backward*. Kotak kosong adalah **bubble**: GPU menganggur menunggu tahap lain.

$$\text{porsi bubble} = \frac{p - 1}{m + p - 1} = \frac{3}{7} \approx 43\%$$

Mengecilkan *bubble*

- ▶ **Perbanyak micro-batch.** Dengan $p = 4$ dan $m = 32$: porsi *bubble* = $3/35 \approx 9\%$. Tetapi *activation* untuk semua *micro-batch* harus disimpan.
- ▶ **Jadwal 1F1B.** Begitu *pipeline* penuh, setiap tahap bergantian satu *forward*, satu *backward*. *Bubble* sama, tetapi *activation* yang ditahan jauh lebih sedikit.
- ▶ **Interleaved.** Tiap GPU memegang beberapa potong lapis yang tidak bersebelahan, sehingga *bubble* mengecil, dibayar dengan komunikasi lebih banyak.



Pipeline mengirim *activation* antar tahap dengan *send/recv*: volumenya kecil dibanding *all-reduce*, sehingga cocok melintasi node.

3D parallelism: memetakan ke perangkat keras

Dimensi	Komunikasi	Frekuensi	Ditempatkan di
<i>Tensor parallelism</i>	<i>All-reduce activation</i>	Setiap lapis, sangat sering	Dalam satu node (NVLink, NVSwitch)
<i>Pipeline parallelism</i>	<i>Send/recv activation</i>	Antar tahap, sedang	Antar node berdekatan
<i>Data parallelism</i> (DDP/FSDP)	<i>All-reduce</i> atau <i>reduce-scatter</i> gradien	Sekali per langkah, besar tetapi bisa di-overlap	Lapis terluar, menyeberangi kluster

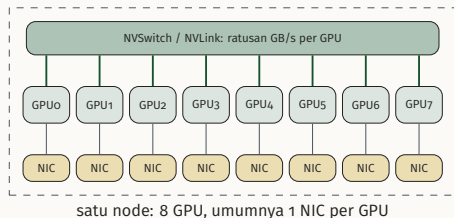
Contoh susunan: 8 GPU per node untuk *tensor*, 16 tahap *pipeline*, dan sisanya *data parallel*. Total GPU = $TP \times PP \times DP$, misalnya $8 \times 16 \times 128 = 16.384$.

Intinya

Prinsipnya sama dengan pertemuan 10 tentang *cache*: letakkan komunikasi yang paling sering di jalur yang paling cepat dan paling dekat.

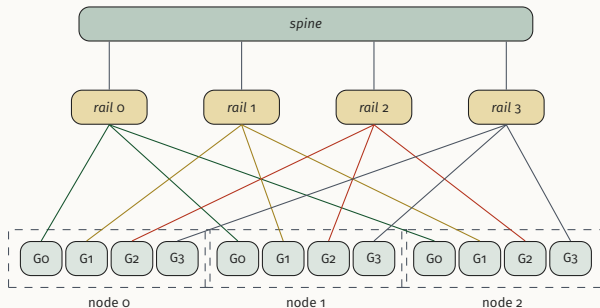
Jaringan klaster AI

Di dalam satu node, dan di antara node



- ▶ **Di dalam node:** NVLink dan NVSwitch menghubungkan GPU langsung, jauh lebih cepat daripada PCIe.
- ▶ **Antar node:** setiap GPU punya NIC sendiri berkecepatan ratusan Gb/s (InfiniBand atau Ethernet RoCE), memakai RDMA.
- ▶ Selisih *bandwidth* dalam node dan antar node bisa sekitar satu orde besaran. Itulah dasar penempatan 3D *parallelism*.

Topologi rail-optimized



- ▶ GPU nomor k di **setiap** node disambungkan ke *switch* yang sama, disebut *rail k*.
- ▶ Komunikasi kolektif antar node umumnya terjadi antar GPU bernomor sama, sehingga cukup satu lompatan *switch*.
- ▶ Lalu lintas lain melewati *spine*, seperti *spine-leaf* di data center biasa.

Intinya

Topologi jaringan dirancang mengikuti **pola komunikasi** beban kerjanya. Pada *microservices* polanya acak; pada pelatihan AI polanya kolektif dan dapat ditebak.

InfiniBand atau RoCE?

	InfiniBand	RoCE (RDMA over Converged Ethernet)
Dasar	Jaringan khusus HPC dengan <i>switch</i> dan NIC sendiri	RDMA di atas Ethernet
<i>Lossless</i>	Bawaan, lewat <i>credit-based flow control</i>	Perlu konfigurasi: PFC, ECN, <i>congestion control</i>
Kelebihan	<i>Latency</i> rendah, matang di HPC, mendukung operasi kolektif di dalam <i>switch</i>	Memakai ekosistem dan keahlian Ethernet yang luas
Tantangan	Ekosistem lebih sempit, umumnya satu vendor dominan	<i>Tuning</i> jaringan lebih sulit agar tetap stabil pada skala besar

Keduanya dipakai di klaster AI besar. Meta, misalnya, melaporkan dua klaster sekitar 24 ribu GPU: satu dengan RoCE dan satu dengan InfiniBand. Pelajaran pertemuan 11 berlaku: RDMA melewati kernel sistem operasi agar *latency* mikrodetik tercapai.

Keandalan pada skala ribuan GPU

Matematika kegagalan di kluster besar

Bila tiap komponen gagal saling bebas dengan rata-rata MTBF M , maka kluster berisi N komponen gagal rata-rata setiap

$$\text{MTBF}_{\text{kluster}} \approx \frac{M}{N}$$

Ilustrasi (asumsi): satu GPU rata-rata gagal sekali dalam 5 tahun ≈ 43.800 jam.

$$\frac{43.800 \text{ jam}}{16.000 \text{ GPU}} \approx \mathbf{2,7 \text{ jam}}$$

Dan itu baru GPU: belum memori, NIC, kabel optik, *switch*, catu daya, pendingin, dan perangkat lunak.

Fail-stop

Pelatihan sinkron bersifat *fail-stop*: satu GPU mati, semua *rank* terhenti di *all-reduce* berikutnya. Satu komponen dari 16 ribu menghentikan seluruh pekerjaan.

Intinya

Ini kebalikan dari *microservices* di Capaian A, yang tetap melayani saat satu replika mati. Di sini strateginya: **deteksi cepat, ganti, muat ulang checkpoint, lanjutkan.**

Studi kasus: pelatihan Llama 3

- ▶ Model terbesar, 405B, dilatih pada sekitar **16 ribu GPU H100**.
- ▶ Dalam periode pengamatan **54 hari**, tercatat **466 gangguan** pekerjaan — rata-rata kira-kira satu gangguan setiap 3 jam.
- ▶ Sebagian kecil terencana (pemeliharaan); sebagian besar tidak terduga, dan mayoritas penyebabnya **perangkat keras**: GPU, memori HBM, dan komponen lain.
- ▶ Meski begitu, pelatihan tetap berjalan dengan porsi waktu efektif yang tinggi, karena deteksi, diagnosis, dan pemulihan dibuat otomatis.

Sumber: Meta, *The Llama 3 Herd of Models*, 2024.

Cocokkan dengan hitungan tadi

Hitungan kasar MTBF M/N menghasilkan “beberapa jam sekali”. Data nyata menunjukkan hal yang sama. Kerusakan pada skala ini bukan kecelakaan, melainkan **cuaca**.

Pelajarannya untuk Anda: rancangan yang benar dimulai dari **asumsi kegagalan**, bukan ditambahkan setelah sistem jatuh.

Silent data corruption: kegagalan yang tidak berteriak

Silent data corruption (SDC): perangkat keras menghasilkan angka yang **salah** tanpa pesan galat apa pun.

- ▶ Penyebab: cacat produksi kecil pada prosesor, penuaan komponen, *bit flip* di memori yang lolos dari ECC.
- ▶ Google dan Meta pernah melaporkan bahwa pada armada ribuan sampai jutaan mesin, sebagian kecil prosesor menghasilkan kesalahan komputasi seperti ini.
- ▶ Pada pelatihan AI, akibatnya bisa berupa *loss spike*, NaN, atau — yang terburuk — model yang diam-diam lebih buruk.

Cara bertahan

- ▶ Pantau *loss* dan norma gradien; lonjakan aneh memicu pemeriksaan
- ▶ Bandingkan hasil antar replika *data parallel* yang seharusnya identik
- ▶ Jalankan uji diagnostik berkala dan karantina mesin yang mencurigakan
- ▶ *Checksum* pada *checkpoint* dan data yang disimpan
- ▶ Simpan beberapa *checkpoint* lama untuk mundur ke titik yang sehat

Intinya

Fail-stop masih mudah: sistem berhenti dan kita tahu. Yang sulit adalah komponen yang tetap jalan tetapi berbohong — gema dari masalah *Byzantine* di sistem terdistribusi.

Checkpoint: seberapa sering?

Dua ongkos yang saling tarik:

- ▶ *Checkpoint* terlalu jarang → saat gagal, banyak pekerjaan yang hilang.
- ▶ *Checkpoint* terlalu sering → waktu habis untuk menyimpan.

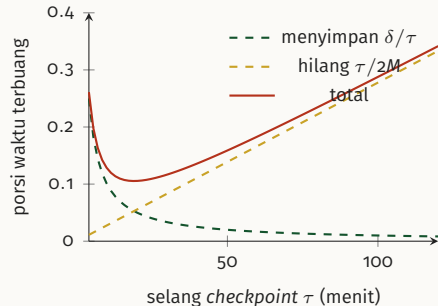
Hampiran klasik Young untuk selang optimal:

$$\tau_{\text{opt}} \approx \sqrt{2 \delta M}$$

δ = lama menyimpan satu *checkpoint*, M = MTBF klaster.

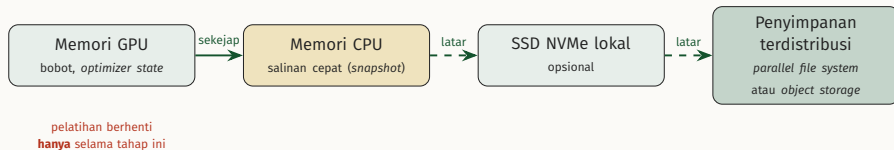
Contoh: $\delta = 1$ menit, $M = 180$ menit

$$\tau_{\text{opt}} \approx \sqrt{2 \cdot 1 \cdot 180} \approx 19 \text{ menit}$$



Makin cepat *checkpoint* (δ kecil), makin sering ia boleh diambil, dan makin sedikit kerja yang hilang.

Checkpoint cepat ke penyimpanan terdistribusi



- ▶ **Asynchronous checkpoint:** GPU hanya berhenti selama menyalin ke memori CPU; penulisan ke disk dan jaringan berlangsung di latar, sambil pelatihan berlanjut.
- ▶ **Sharded checkpoint:** setiap *rank* menulis potongannya sendiri secara paralel, sehingga *bandwidth* penyimpanan seluruh kluster terpakai.
- ▶ Penyimpanannya sendiri adalah sistem terdistribusi dengan replikasi atau *erasure coding* (pertemuan 14): *checkpoint* tidak boleh hilang bersama node yang mati.

Biaya, energi, dan jejak karbon

Menghitung kasar biaya dan energi

Skenario ilustrasi: 16.000 GPU selama 54 hari.

$$16.000 \times 54 \times 24 \approx 2,07 \times 10^7 \text{ GPU-jam}$$

Bila satu GPU menarik sekitar 0,7 kW (asumsi daya puncak kelas H100 SXM):

$$2,07 \times 10^7 \times 0,7 \text{ kW} \approx 14,5 \text{ GWh}$$

Itu baru GPU. CPU, memori, jaringan, dan terutama pendingin menambah lagi; rasio total daya data center terhadap daya perangkat IT disebut **PUE**.

Dari energi ke biaya dan karbon

- ▶ **Biaya sewa** $\approx$ GPU-jam $\times$ harga per GPU-jam
- ▶ **Energi total** $\approx$ energi IT $\times$ PUE
- ▶ **Emisi** $\approx$ energi total $\times$ intensitas karbon listrik (kg CO₂e per kWh), yang sangat bergantung pada sumber listrik di lokasi data center

Intinya

Lokasi, waktu, dan efisiensi perangkat lunak bisa mengubah jejak karbon pelatihan yang sama berkali-kali lipat.

Efisiensi adalah penghematan

MFU (*Model FLOPs Utilization*):

$$\text{MFU} = \frac{\text{FLOP/detik model yang tercapai}}{\text{FLOP/detik puncak GPU}}$$

- ▶ MFU rendah berarti GPU banyak menunggu: komunikasi, *bubble*, I/O data, kegagalan.
- ▶ Menaikkan MFU dari 30% ke 45% setara dengan menambah 50% GPU **tanpa** membeli apa pun.

Ke mana waktu GPU hilang

- ▶ Komunikasi yang tidak tersembunyi (bagian serial Amdahl)
- ▶ *Pipeline bubble*
- ▶ Menunggu data (I/O, *data loader*)
- ▶ *Checkpoint* sinkron
- ▶ Waktu pulih setelah kegagalan
- ▶ *Kernel* yang *memory-bound*, bukan *compute-bound* (*roofline*)

Intinya

Setiap teknik semester ini — *overlap*, *cache*, *tiling*, topologi, *checkpoint* — pada akhirnya diterjemahkan ke uang, waktu, dan energi.

Rangkuman dua capaian semester

Satu klaster AI, dua capaian

Yang terjadi di klaster AI	Konsep yang dipakai	Dipelajari di
Ribuan proses saling menemukan (<i>rendezvous</i>)	<i>Service discovery</i> , koordinasi	Capaian A, P5, P7
Satu GPU mati, pekerjaan pulih	<i>Fail-stop</i> , MTBF/MTTR, <i>fault domain</i> , Kubernetes/Slurm	Capaian A, P1, P7, P11
<i>Checkpoint</i> tersimpan aman	Replikasi, <i>erasure coding</i> , konsistensi	Capaian A, P7, P14
SDC dan komponen “berbohong”	<i>Partial failure</i> , <i>Byzantine</i>	Capaian A, P1, P7
<i>All-reduce</i> gradien	Operasi kolektif MPI, NCCL	Capaian B, P11
Satu lapis secepat mungkin	<i>Cache</i> , <i>tiling</i> , <i>roofline</i> , Tensor Core	Capaian B, P10, P12–13
Komunikasi tersembunyi	<i>Overlap</i> , Amdahl, <i>scaling</i>	Capaian B, P9
Jaringan <i>rail-optimized</i>	InfiniBand, RoCE, RDMA, NVLink	Capaian B, P11, P13

Perjalanan satu semester

Capaian A — tetap benar dan hidup

- ▶ Dua komputer bicara: *socket*, TCP, UDP, QUIC, HTTP/1.1–3, TLS
- ▶ Cara memanggil: RPC, RMI, REST, gRPC, GraphQL, *messaging*
- ▶ Bentuk sistem: *n-tier*, *monolith*, *microservices*
- ▶ Banyak hal sekaligus: *thread*, *goroutine*, *actor*, Rx
- ▶ Tahan gagal: replikasi, konsistensi, Raft, Kubernetes

Capaian B — secepat mungkin

- ▶ Mengukur dengan jujur: *speedup*, *scaling*, Amdahl
- ▶ Satu mesin: *cache*, NUMA, SIMD, OpenMP
- ▶ Banyak mesin: MPI, InfiniBand, RDMA
- ▶ Ribuan *core* kecil: GPU, CUDA, akselerator
- ▶ Skala besar: MapReduce, Spark, *distributed training*

Intinya

Pertanyaan pertemuan 1 masih sama: **bagaimana banyak komputer berkomunikasi**. Yang berubah adalah anggaran waktunya — milidetik untuk layanan web, mikrodetik untuk HPC — dan sikap terhadap kegagalan. Sistem terbesar hari ini membutuhkan keduanya sekaligus.

Praktikum dan presentasi progres UAS

Praktikum: DDP dua proses dan *speedup*

1. Siapkan model kecil (MLP atau CNN kecil) dan dataset seperti MNIST atau CIFAR-10.
2. Jalankan versi satu proses biasa; catat waktu per *epoch* dan akurasi.
3. Jalankan dengan `torchrun -nproc_per_node=2`:
 - Colab atau Kaggle dengan 2 GPU: `backend="nccl"`
 - Tanpa 2 GPU: `backend="gloo"` di CPU, dengan `torch.set_num_threads` dibagi agar adil
4. Hitung *speedup* dan efisiensi. Ulangi minimal 3 kali, laporkan median.
5. Ubah ukuran *batch* per proses. Kapan komunikasi mulai mendominasi?
6. Tambahkan *checkpoint* tiap *epoch*, matikan satu proses di tengah jalan (`kill`), lalu lanjutkan dari *checkpoint*. Berapa kerja yang hilang?

Intinya

Hasil yang wajar: *speedup* di bawah 2. Tugas Anda menjelaskan **mengapa**: ukuran gradien, *bandwidth*, ukuran *batch*, dan bagian serial.

Setiap kelompok, 7 menit:

1. Komputasi yang dipilih dan ukurannya
2. Versi yang sudah jalan: serial, *multicore*, banyak node, GPU
3. Angka sementara: waktu, *speedup*, efisiensi
4. Satu hambatan terbesar dan rencana mengatasinya

Yang akan ditanyakan

- ▶ Bagaimana Anda tahu hasil paralelnya **benar**?
- ▶ Berapa bagian serialnya, dan cocokkah dengan Amdahl?
- ▶ Apakah *bottleneck*-nya komputasi, memori, atau komunikasi?
- ▶ Apakah pengukuran diulang dan kondisinya dikendalikan?

Tugas untuk pertemuan berikutnya

1. Selesaikan laporan praktikum DDP: tabel waktu, *speedup*, efisiensi, dan analisis penyebabnya.
2. Finalisasi tugas besar UAS “Satu soal, makin banyak prosesor”:
 - semua versi (serial, *multicore*, banyak node, GPU) menghasilkan jawaban yang sama;
 - grafik *strong scaling* dan *weak scaling*, dibandingkan dengan kurva Amdahl;
 - penjelasan *bottleneck* dengan bukti (*profiling* atau *roofline*);
 - repositori yang bisa dijalankan ulang dan data pengukuran mentah.
3. Siapkan presentasi 10 menit dan demo singkat.
4. Bacaan pengayaan: *The Llama 3 Herd of Models* bagian infrastruktur, dan makalah ZeRO (Rajbhandari dkk., 2020).

Pertemuan depan

UAS: presentasi dan demo tugas besar. Tunjukkan bahwa setiap klaim “lebih cepat” didukung angka yang Anda ukur sendiri.

Terima kasih

Pertanyaan?