Pertemuan 11 — *HPC cluster* dan MPI

Ketika satu mesin tidak cukup: *distributed memory*, pengiriman *message*, dan jaringan yang sangat cepat

Hendri Karisma

Semester Ganjil 2026/2027

Program Studi Teknik Informatika, STMIK Tazkia

Alur pertemuan hari ini

1. Dari *shared memory* ke *distributed memory*
2. Anatomi sebuah *HPC cluster*
3. MPI: *rank*, *communicator*, *send* dan *receive*
4. *Deadlock* dan cara menghindarinya
5. *Collective*: *broadcast* sampai *all-reduce*
6. *Communication cost model* alfa-beta
7. Topologi: *fat-tree*, *dragonfly*, *torus*
8. InfiniBand, RDMA, RoCE, *kernel bypass*
9. *Cluster scheduler*: Slurm vs Kubernetes
10. Praktikum MPI di *container*

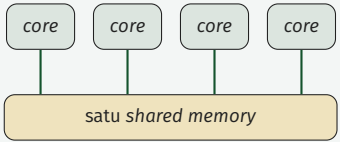
Sasaran pertemuan ini

Anda dapat menulis dan menjalankan program MPI (*point-to-point* dan *collective*), memperkirakan *communication cost* dengan model alfa-beta, dan menjelaskan mengapa *HPC cluster* memakai jaringan dan *scheduler* yang berbeda dari *cluster* layanan web.

Dari *shared memory* ke *distributed memory*

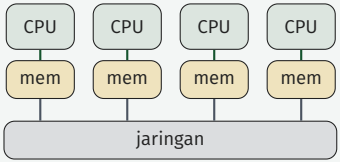
Pekan lalu satu mesin, hari ini banyak mesin

Shared memory (OpenMP, P10)



Semua *thread* membaca variabel yang sama. Mudah, tetapi terbatas pada jumlah *core* dan memori satu mesin.

Distributed memory (MPI, hari ini)



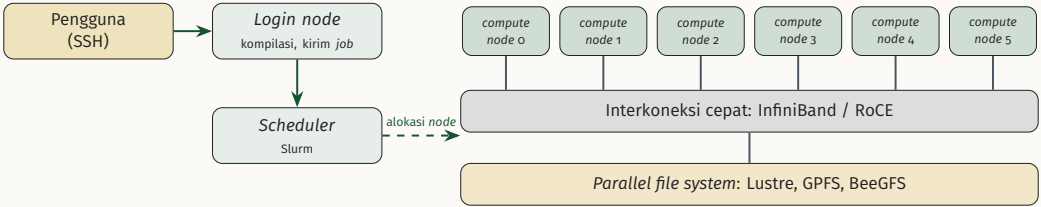
Tiap proses punya memori sendiri. Data milik proses lain hanya bisa didapat dengan **mengirim *message***.

Ringkasan

Ini *shared-nothing* yang sama dengan *microservices* di paruh pertama. Bedanya anggaran waktu: layanan web hidup dalam milidetik, program HPC dalam mikrodetik.

Anatomi sebuah *HPC cluster*

Bagaimana sebuah *HPC cluster* disusun



- Pengguna tidak pernah menjalankan komputasi berat di *login node*; ia mengirim *job* ke *scheduler*, yang memilih *compute node* yang kosong.
- *Compute node* identik satu sama lain (CPU banyak *core*, kadang GPU) dan biasanya tanpa disk penting: data ada di *parallel file system*.

Peran tiap komponen

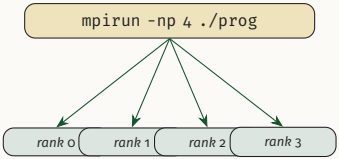
Komponen	Tugas	Mengapa berbeda dari *server* web
Login node	Pintu masuk: SSH, kompilasi, menyiapkan dan mengirim *job*	Dipakai bersama banyak pengguna; bukan untuk komputasi
Compute node	Menjalankan proses MPI, *thread* OpenMP, *kernel* GPU	Disewa utuh untuk satu *job* sampai selesai
Interkoneksi	Membawa *message* antar proses: *latency* orde mikrodetik, *bandwidth* ratusan Gb/s	Ethernet + TCP biasa terlalu lambat dan terlalu banyak salinan
Parallel file system	Membagi satu *file* besar ke banyak *server* penyimpanan agar dibaca paralel	Ribuan proses membaca *file* yang sama sekaligus
Scheduler (Slurm, PBS)	*Queue*, prioritas, jatah waktu, pembagian *node*	*Job* punya awal dan akhir; layanan web berjalan terus

Daftar TOP500 setiap tahun memuat *cluster* semacam ini: puluhan ribu *node*, satu jaringan khusus, satu *file system* paralel. Polanya sama, hanya skalanya yang berbeda.

MPI: *rank*, *communicator*, *send* dan *receive*

Apa itu MPI

- **Message Passing Interface** — standar (bukan program) untuk pertukaran *message* antar proses, disusun oleh MPI Forum sejak awal 1990-an.
- Implementasinya banyak: **Open MPI**, **MPICH**, Intel MPI, MVAPICH. Kode yang sama dapat dikompilasi ulang di *cluster* mana pun.
- API resmi untuk C dan Fortran; Python lewat `mpi4py`.
- Model **SPMD** (*Single Program, Multiple Data*): satu program yang sama dijalankan *N* kali; tiap salinan tahu nomornya dan memilih bagian kerjanya sendiri.



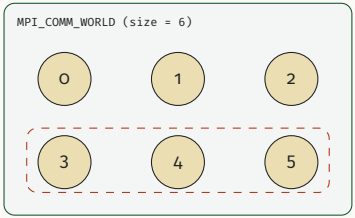
empat proses, kode sama, `rank` berbeda

Ringkasan

MPI tidak menyembunyikan jaringan. Pemrogram sendiri yang memutuskan siapa mengirim apa kepada siapa. Itu membuatnya sulit, tetapi juga membuatnya sangat cepat dan dapat diskalakan.

Rank, *size*, dan *communicator*

- ***Communicator*** — kelompok proses yang boleh saling berkirim *message*. Kelompok bawaan berisi semua proses: `MPI_COMM_WORLD`.
- ***Rank*** — nomor proses di dalam *communicator*, dari 0 sampai *size* − 1.
- ***Size*** — jumlah proses di dalam *communicator*.
- *Communicator* baru dapat dibuat dari sebagian proses, misalnya satu per baris matriks, dengan `MPI_Comm_split`.



communicator kecil hasil `MPI_Comm_split`

Program MPI pertama

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

int main(int argc, char **argv) {
    MPI_Init(&argc, &argv);

    int rank, size, len;
    char nama[MPI_MAX_PROCESSOR_NAME];
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Get_processor_name(nama, &len);

    printf("Halo dari rank %d dari %d, di %s\n",
           rank, size, nama);

    MPI_Finalize();
    return 0;
}
```

Kompilasi dan jalankan:

```
mpicc halo.c -o halo
mpirun -np 4 ./halo
```

Output (urutannya bisa berbeda setiap kali):

```
Halo dari rank 2 dari 4, di node01
Halo dari rank 0 dari 4, di node01
Halo dari rank 3 dari 4, di node01
Halo dari rank 1 dari 4, di node01
```

`mpicc` hanyalah *wrapper* untuk `gcc` yang menambahkan *header* dan *library* MPI. Urutan cetak acak adalah pengingat pertama: tidak ada jam bersama antar proses.

Point-to-point: `MPI_Send` dan `MPI_Recv`

```
int MPI_Send(const void *buf, int count, MPI_Datatype type,
             int dest, int tag, MPI_Comm comm);
int MPI_Recv(void *buf, int count, MPI_Datatype type,
             int source, int tag, MPI_Comm comm, MPI_Status *status);
```

```
double data[100];
if (rank == 0) {
    isi(data, 100);
    MPI_Send(data, 100, MPI_DOUBLE, 1, 7,
             MPI_COMM_WORLD);
} else if (rank == 1) {
    MPI_Recv(data, 100, MPI_DOUBLE, 0, 7,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

- Sebuah *message* dikenali dari tiga hal: pengirim (*source*), *tag*, dan *communicator*.
- *Receive* harus cocok dengan *send*: tipe dan jumlah elemen yang sama (atau lebih besar).
- `MPI_ANY_SOURCE` dan `MPI_ANY_TAG` menerima dari siapa saja.
- Antara dua proses yang sama, *message* dengan *tag* dan *communicator* yang sama tiba sesuai urutan kirim.

Deadlock dan cara menghindarinya

Blocking send bisa membuat program macet

Dua proses bertukar *array* besar:

```
int lawan = 1 - rank;   /* 0 <-> 1 */
MPI_Send(kirim,  N, MPI_DOUBLE, lawan, 0,
         MPI_COMM_WORLD);
MPI_Recv(terima, N, MPI_DOUBLE, lawan, 0,
         MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Untuk *N* kecil program ini sering berjalan, untuk *N* besar macet. Mengapa?

- *Message* kecil biasanya dikirim secara ***eager***: disalin ke *buffer* internal, lalu `MPI_Send` langsung kembali.
- *Message* besar memakai protokol ***rendezvous***: `MPI_Send` menunggu sampai penerima memanggil `MPI_Recv`.
- Kedua proses sama-sama menunggu di `MPI_Send`, tidak ada yang sampai ke `MPI_Recv`: ***deadlock***.
- Standar MPI tidak menjamin `MPI_Send` punya *buffer*. Program yang "kebetulan jalan" tetap program yang salah.



Tiga cara memperbaikinya

1. Atur urutan

```
if (rank % 2 == 0) {
    MPI_Send(kirim, ...);
    MPI_Recv(terima, ...);
} else {
    MPI_Recv(terima, ...);
    MPI_Send(kirim, ...);
}
```

Sederhana, tetapi rawan salah bila polanya rumit.

2. `MPI_Sendrecv`

```
MPI_Sendrecv(
  kirim,  N, MPI_DOUBLE,
  lawan, 0,
  terima, N, MPI_DOUBLE,
  lawan, 0,
  MPI_COMM_WORLD,
  MPI_STATUS_IGNORE);
```

Library yang mengurus urutannya. Cocok untuk pertukaran berpasangan.

3. *Non-blocking*

```
MPI_Request req[2];
MPI_Irecv(terima, N, MPI_DOUBLE,
  lawan, 0, MPI_COMM_WORLD,
  &req[0]);
MPI_Isend(kirim, N, MPI_DOUBLE,
  lawan, 0, MPI_COMM_WORLD,
  &req[1]);
/* boleh menghitung di sini */
MPI_Waitall(2, req,
  MPI_STATUSES_IGNORE);
```

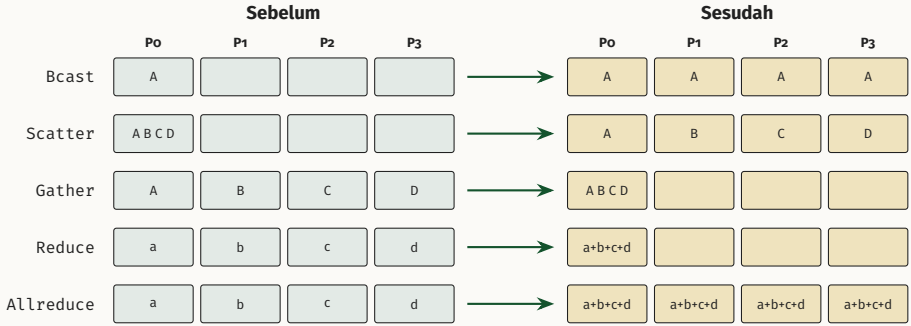
Paling fleksibel, dan memberi peluang menumpuk komputasi dengan komunikasi.

Ringkasan

Buffer yang dipakai `MPI_Isend`/`MPI_Irecv` tidak boleh disentuh sebelum `MPI_Wait` selesai. Itu jenis *race condition* versi MPI.

Collective: *broadcast* sampai *all-reduce*

Lima *collective* yang paling sering dipakai



Collective dipanggil oleh **semua** proses di *communicator*. Bila satu proses lupa memanggilnya, semua proses lain menunggu selamanya.

Kapan memakai yang mana

Fungsi	Arti	Contoh pemakaian
`MPI_Bcast`	Satu proses menyebarkan data yang sama ke semua	Parameter simulasi dibaca *rank* 0, lalu disebar
`MPI_Scatter`	Satu proses membagi *array* menjadi potongan	Baris-baris matriks dibagi ke setiap *rank*
`MPI_Gather`	Kebalikan *scatter*: potongan dikumpulkan ke satu proses	Hasil per *rank* dikumpulkan untuk ditulis ke *file*
`MPI_Reduce`	Gabungkan nilai semua proses dengan operasi (jumlah, maks, min) ke satu proses	Total *error*, jumlah partikel, estimasi π (*pi*)
`MPI_Allreduce`	*Reduce* lalu hasilnya dikirim ke semua	Syarat berhenti iterasi; rata-rata *gradient* saat melatih model AI (P15)
`MPI_Barrier`	Semua menunggu sampai semua tiba	Menyelaraskan sebelum mengukur waktu

Ringkasan

Pakai *collective* bawaan, jangan menulis ulang dengan `Send`/`Recv` dalam *loop*. Implementasi MPI memilih algoritma (pohon, cincin) yang sesuai dengan topologi jaringannya.

Contoh lengkap: estimasi π dengan MPI di C

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv) {
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    long n = (argc > 1) ? atol(argv[1]) : 100000000L;
    double h = 1.0 / n, lokal = 0.0, pi = 0.0;
    MPI_Barrier(MPI_COMM_WORLD);
    double t0 = MPI_Wtime();
    for (long i = rank; i < n; i += size) {  /* bergiliran */
        double x = h * (i + 0.5);
        lokal += 4.0 / (1.0 + x * x);
    }
    lokal *= h;
    MPI_Reduce(&lokal, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    double t1 = MPI_Wtime();
    if (rank == 0)
        printf("pi ~ %.12f  waktu %.3f s  (%d proses)\n", pi, t1 - t0, size);
    MPI_Finalize();
    return 0;
}
```

Gagasannya:

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx$$

dihampiri dengan jumlah *n* persegi panjang.

- Setiap *rank* menghitung persegi panjang ke-`rank`, `rank+size`, ...
- Komunikasinya hanya **satu** `MPI_Reduce` di akhir: hampir seluruh waktu dipakai menghitung.
- Karena itu *speedup*-nya mendekati ideal. Bandingkan dengan Amdahl (P9): bagian serialnya sangat kecil.

```
mpicc -O2 pi.c -o pi
mpirun -np 4 ./pi 1000000000
```

Versi Python dengan `mpi4py`

```python
from mpi4py import MPI
import numpy as np
import sys

comm = MPI.COMM_WORLD
rank, size = comm.Get_rank(), comm.Get_size()

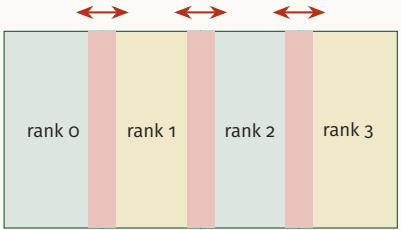
n = int(sys.argv[1]) if len(sys.argv) > 1 else 10_000_000
h = 1.0 / n
i = np.arange(rank, n, size, dtype=np.float64)
x = h * (i + 0.5)
lokal = h * np.sum(4.0 / (1.0 + x * x))

# semua rank menerima hasil yang sama
pi = comm.allreduce(lokal, op=MPI.SUM)
if rank == 0:
    print(f"pi ~ {pi:.12f} dari {size} proses")
```

```
mpirun -np 4 python3 pi_mpi.py 100000000
```

- `mpi4py` membungkus *library* MPI yang sama dengan versi C.
- Fungsi huruf kecil (`allreduce`, `send`) mengirim objek Python apa pun lewat *pickle*: mudah tetapi lebih lambat.
- Fungsi huruf besar (`Allreduce`, `Send`) mengirim *buffer* NumPy langsung: hampir secepat C.
- Komputasi tetap di NumPy; Python hanya sebagai perekat. Ini pola yang sama dengan pembahasan dampak bahasa di P1.

Pola umum: *domain decomposition* dan *halo exchange*



kolom *halo* ditukar setiap langkah

- Simulasi panas, cuaca, atau fluida memakai *grid* besar. *Grid* dibagi menjadi potongan, satu per *rank*.
- Titik di tepi potongan membutuhkan nilai tetangganya yang dimiliki *rank* lain. Kolom tepi itu (*halo*) dipertukarkan setiap langkah waktu, biasanya dengan `MPI_Sendrecv` atau *non-blocking*.
- Komputasi sebanding dengan **luas** potongan, komunikasi sebanding dengan **kelilingnya**. Potongan besar berarti rasio komunikasi kecil.

Ringkasan

Inilah alasan *strong scaling* selalu mentok: makin banyak *rank*, potongan makin kecil, keliling relatif terhadap luas makin besar, dan komunikasi mulai mendominasi.

Model *communication cost* alfa-beta

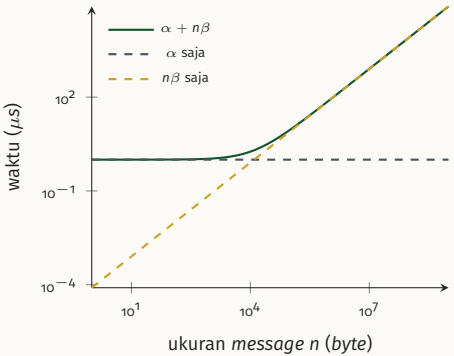
Berapa lama sebuah *message*? Model alfa-beta

$$T(n) = \alpha + n\,\beta$$

- α (dibaca *alpha*) — ***latency***: *fixed cost* setiap *message* (*software*, kartu jaringan, *switch*), berapa pun ukurannya.
- $\beta = 1/\textit{bandwidth}$ (*beta*) — *cost* per *byte*.
- n — ukuran *message* dalam *byte*.

Contoh angka (ilustrasi): $\alpha = 1\,\mu$s (mikrodetik; μ dibaca *mu*), *bandwidth* 100 Gb/s = 12,5 GB/s, sehingga $\beta = 0{,}08$ ns/*byte*.

n	$T(n)$	didominasi
8 B	$\approx 1\,\mu$s	α
12,5 KB	$2\,\mu$s	seimbang
1 MB	$\approx 81\,\mu$s	β
1 GB	≈ 80 ms	β



Titik seimbang $n_{1/2} = \alpha/\beta = 12{,}5$ KB: *message* yang lebih kecil dari ini “membayar” terutama *latency*.

Akibat praktis model alfa-beta

- **Gabungkan *message* kecil.** Seribu *message* 8 B $\approx 1000\alpha$ = 1 ms; satu *message* 8 KB $\approx \alpha + 8000\beta \approx 1{,}6\,\mu$s. Datanya sama, waktunya ratusan kali berbeda.
- ***Overlap* komputasi dan komunikasi.** Dengan `MPI_Isend`/`MPI_Irecv`, proses dapat menghitung bagian dalam potongan sambil menunggu *halo* datang.
- ***Collective* punya rumus sendiri.** Misalnya *ring all-reduce* pada p proses dan data n *byte*:

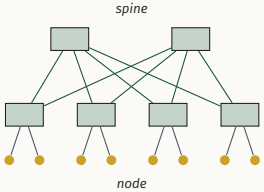
  $$T \approx 2(p-1)\,\alpha + 2\,\frac{p-1}{p}\,n\,\beta$$

  Suku *bandwidth* hampir tidak bertambah dengan p, tetapi suku *latency* tumbuh linear. Kita bertemu rumus ini lagi saat melatih model AI di banyak GPU (P15).
- **Ethernet + TCP** punya α jauh lebih besar daripada InfiniBand karena *message* melewati *kernel* dan disalin berkali-kali. Itu topik bagian berikutnya.

Topologi: *fat-tree*, *dragonfly*, *torus*

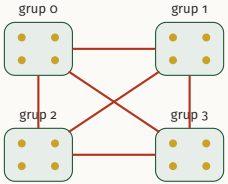
Tiga cara menyambungkan ribuan *node*

Fat-tree



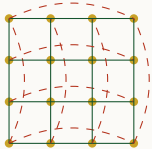
Mirip *spine-leaf* di *data center*: banyak jalur sama panjang, *bandwidth* besar ke segala arah. Umum pada InfiniBand *cluster* dan *AI cluster*.

Dragonfly



Node dikelompokkan; di dalam grup tersambung rapat, antar grup cukup satu lompatan kabel optik panjang. Hemat kabel mahal pada skala sangat besar; dipakai pada banyak superkomputer Cray/HPE.

Torus



Tiap *node* hanya tersambung ke tetangganya; tepi kiri disambung ke tepi kanan (garis putus). Cocok untuk soal yang memang bertetangga seperti *halo exchange*. Dipakai pada Blue Gene, Fugaku, dan TPU *pod*.

Membandingkan topologi

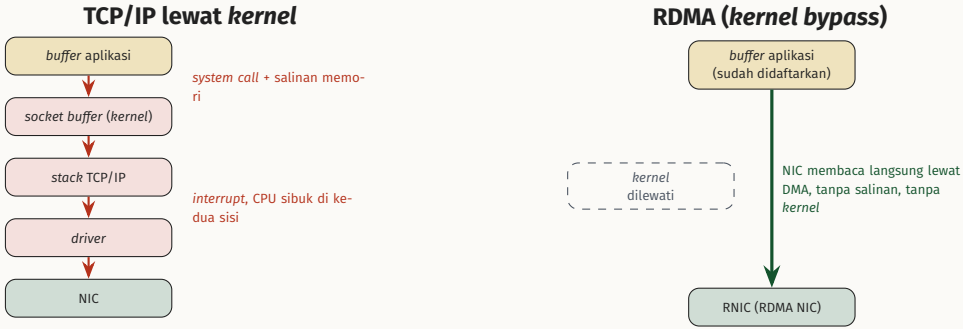
	Fat-tree	**Dragonfly**	**Torus**
Jumlah lompatan	Kecil dan seragam	Kecil (sekitar 3 lompatan global)	Tumbuh dengan ukuran sisi *grid*
Cost	*Switch* dan kabel banyak	Lebih hemat kabel panjang	Paling hemat: tanpa *switch* pusat
Lalu lintas acak	Sangat baik	Baik, perlu *adaptive routing*	Kurang baik
Lalu lintas tetangga	Baik	Baik	Sangat baik
Contoh	InfiniBand *cluster*, *AI cluster*	Superkomputer Cray/HPE	Blue Gene, Fugaku, TPU

Ringkasan

Tidak ada topologi terbaik. Pilihan topologi adalah taruhan tentang **pola komunikasi** aplikasi yang akan dijalankan dan anggaran yang tersedia.

InfiniBand, RDMA, RoCE, *kernel bypass*

Jalan data: TCP biasa vs RDMA



- **RDMA** (*Remote Direct Memory Access*): kartu jaringan membaca atau menulis memori aplikasi di mesin lain **tanpa melibatkan CPU dan sistem operasi** mesin itu (operasi *one-sided*).
- Hasilnya: α orde mikrodetik, salinan memori nol (*zero-copy*), dan CPU bebas untuk menghitung.

InfiniBand, RoCE, dan kawan-kawannya

InfiniBand

- Jaringan yang dirancang untuk HPC, RDMA bawaan
- *Lossless*: pengirim hanya mengirim bila penerima punya ruang (*credit-based flow control*), sehingga *packet* tidak dibuang
- Dikelola *subnet manager*; generasi baru bergerak dari 200 ke 400 Gb/s per port dan seterusnya
- Banyak dipakai pada TOP500 dan *AI training cluster*

RoCE dan iWARP

- **RoCE** (*RDMA over Converged Ethernet*): RDMA di atas Ethernet; RoCEv2 dibungkus UDP/IP sehingga bisa di-*route*
- Butuh Ethernet yang "hampir *lossless*": PFC dan ECN harus disetel benar, bila tidak kinerjanya jatuh
- **iWARP**: RDMA di atas TCP, lebih jarang dipakai
- Daya tariknya: memakai *switch* Ethernet yang sudah dikenal tim jaringan

Ringkasan

MPI dan NCCL memakai RDMA secara otomatis bila perangkatnya ada (lewat *library* seperti UCX atau *libfabric*). Program Anda tidak berubah; yang berubah adalah α dan β pada *cost model*.

Konsep inti RDMA

- ***Memory registration*** — aplikasi mendaftarkan dan "mengunci" area memori agar NIC boleh mengaksesnya; NIC mendapat kunci akses.
- ***Queue pair*** — *send queue* dan *receive queue* di NIC; aplikasi menaruh permintaan kerja langsung ke *queue* ini dari *user space*.
- ***Two-sided*** (*send*/*receive*) — kedua pihak ikut serta, mirip `MPI_Send`/`MPI_Recv`.
- ***One-sided*** (*RDMA read*/*write*) — hanya satu pihak yang bertindak; CPU di sisi lain tidak tahu apa-apa.
- ***Completion queue*** — NIC memberi tahu aplikasi bahwa pekerjaan selesai.

Hubungan dengan paruh pertama

Ingat delapan *fallacies* (P1): "*transport cost* itu nol" dan "*latency* itu nol". RDMA adalah usaha paling serius untuk mendekati dua asumsi itu, dengan harga *hardware* khusus dan jaringan yang harus dijaga *lossless*.

Gagasan yang sama muncul lagi di P13 (NVLink, GPUDirect RDMA) dan P15 (komunikasi antar GPU saat melatih model besar).

Cluster scheduler: Slurm vs Kubernetes

Slurm: mengirim *job* ke *cluster*

File `pi.sbatch`:

```
#!/bin/bash
#SBATCH --job-name=pi-mpi
#SBATCH --partition=compute
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=32
#SBATCH --time=00:10:00
#SBATCH --output=pi-%j.out

module load openmpi
srun ./pi 10000000000
```

Perintah sehari-hari:

```
sbatch pi.sbatch      # kirim job
squeue -u $USER       # lihat antrean
scancel 12345         # batalkan job
sinfo                 # status partisi
```

- ***Partition***: kelompok *node* (misalnya `compute`, `gpu`).
- Job meminta 4 *node* $\times$ 32 proses = 128 *rank* MPI, paling lama 10 menit.
- `srun` menjalankan program di semua *node* yang dialokasikan; perannya mirip `mpirun`.
- Job yang melewati batas `–time` dihentikan paksa. Itulah alasan *checkpoint* penting.

Slurm vs Kubernetes: dua *scheduler*, dua dunia

	Slurm (HPC)	**Kubernetes (layanan, P7)**
Unit kerja	*Job batch*: mulai, menghitung, selesai	*Pod* layanan yang berjalan terus
Tujuan utama	Memakai *cluster* sepenuh mungkin; *fair queue*	Menjaga *desired state*; *availability*
Scheduling	*Gang scheduling*: 128 proses jalan bersama atau tidak sama sekali	Per *pod*, satu per satu
Bila *node* mati	*Job* gagal; muat ulang dari *checkpoint* (*fail-stop*)	*Pod* dijadwal ulang otomatis (*self-healing*)
Lingkungan	*Module* di sistem bersama, MPI, RDMA	*Container image*, jaringan *overlay*
Kekuatan	Kinerja dan topologi jaringan	Otomasi dan ekosistem

Keduanya kini saling mendekat: ada operator Kubernetes untuk *job* MPI (misalnya dari Kubeflow) dan *scheduler batch* seperti Volcano, sementara pusat HPC mulai menerima *container*. Ini contoh nyata titik temu Jalur A dan Jalur B.

Praktikum MPI di *container*

Praktikum: menyiapkan MPI di laptop

1. Jalankan *container* Ubuntu dan pasang Open MPI serta `mpi4py`:

```
docker run --rm -it -v "$PWD":/kerja -w /kerja ubuntu:24.04 bash
apt-get update && apt-get install -y build-essential openmpi-bin \
    libopenmpi-dev python3-mpi4py python3-numpy
```

2. Kompilasi dan jalankan estimasi π dengan 1, 2, 4, dan 8 proses:

```
mpicc -O2 pi.c -o pi
for np in 1 2 4 8; do
  mpirun --allow-run-as-root --oversubscribe -np $np ./pi 1000000000
done
```

3. Setelah `pingpong` pada slide berikutnya dikompilasi, paksa komunikasi lewat TCP (bukan *shared memory*) untuk melihat perbedaan α:

```
mpirun --allow-run-as-root --mca pml ob1 --mca btl self,tcp -np 2 ./pingpong
```

`–allow-run-as-root` diperlukan karena di dalam *container* kita `root`; `–oversubscribe` mengizinkan proses lebih banyak dari jumlah *core*. Hasil *np* melebihi jumlah *core* fisik justru melambat: catat dan jelaskan.

Praktikum: mengukur α dan β dengan *ping-pong*

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#define ULANG 100
int main(int argc, char **argv) {
    MPI_Init(&argc, &argv);
    int rank;  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    char *buf = malloc(1 << 24);
    for (int n = 1; n <= (1 << 24); n *= 4) {
        MPI_Barrier(MPI_COMM_WORLD);
        double t0 = MPI_Wtime();
        for (int k = 0; k < ULANG; k++) {
            if (rank == 0) {
                MPI_Send(buf, n, MPI_BYTE, 1, 0, MPI_COMM_WORLD);
                MPI_Recv(buf, n, MPI_BYTE, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            } else if (rank == 1) {
                MPI_Recv(buf, n, MPI_BYTE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                MPI_Send(buf, n, MPI_BYTE, 0, 0, MPI_COMM_WORLD);
            }
        }
        double t = (MPI_Wtime() - t0) / (2.0 * ULANG);  /* satu arah */
        if (rank == 0) printf("%9d byte  %10.2f us\n", n, t * 1e6);
    }
    free(buf);  MPI_Finalize();  return 0;
}
```

- Rank 0 dan 1 saling memantulkan *message*; waktu satu arah adalah waktu pulang-pergi dibagi dua.
- Gambarkan hasilnya pada skala log-log, seperti grafik model alfa-beta.
- Taksir α dari *message* terkecil, dan *bandwidth* dari *message* terbesar: $1/\beta \approx n/T(n)$.
- Jalankan dua kali: *shared memory* (bawaan) dan `–mca pml ob1 –mca btl self,tcp`. Bandingkan α keduanya.

```
mpicc -O2 pingpong.c -o pingpong
mpirun --allow-run-as-root -np 2 ./pingpong
```

Praktikum: tabel yang harus diisi

A. *Scaling* estimasi π ($n = 10^9$)

np	waktu (s)	*speedup*	efisiensi
1	…	1,00	100%
2	…	…	…
4	…	…	…
8	…	…	…

Ulangi tiap *configuration* minimal 3 kali dan laporkan median (P9).

B. Model alfa-beta

Transport	α (μs)	*bandwidth* (GB/s)
Shared memory	…	…
TCP (*loopback*)	…	…

Nilai lebih: jalankan dua *container* di dua laptop dalam satu jaringan (dengan *hostfile* dan SSH) dan isi baris ketiga "TCP antar mesin".

Ringkasan

Pertanyaan analisis: berapa ukuran *message* minimum agar *bandwidth* yang Anda dapat mencapai separuh dari maksimum? Cocokkan dengan $n_{1/2} = \alpha/\beta$.

Penutup

Ringkasan

- *HPC cluster* = *login node* + *compute node* + interkoneksi cepat + *parallel file system* + *scheduler*.
- MPI memakai model SPMD: satu program, banyak *rank*, data dipertukarkan dengan *message* eksplisit.
- *Blocking send* pada dua arah bisa *deadlock*; gunakan `MPI_Sendrecv` atau *non-blocking*.
- *Collective* (*Bcast*, *Scatter*, *Gather*, *Reduce*, *Allreduce*) dipanggil semua *rank* dan dioptimalkan oleh *library*.
- Model $T = \alpha + n\beta$ menjelaskan mengapa *message* kecil mahal dan mengapa *strong scaling* mentok.
- Topologi, InfiniBand, dan RDMA ada untuk menekan α dan β; Slurm mengatur siapa memakai *node* mana dan berapa lama.

Tugas untuk pertemuan berikutnya

1. Selesaikan dua tabel praktikum (*scaling* π dan alfa-beta). Sertakan grafik log-log hasil *ping-pong* dan kurva *speedup* dibandingkan hukum Amdahl.
2. Ubah program π agar memakai `MPI_Bcast` untuk menyebarkan n dari *rank* 0 dan `MPI_Allreduce` sebagai pengganti `MPI_Reduce`. Jelaskan perbedaannya dalam dua kalimat.
3. Baca Pacheco dan Malensek, *An Introduction to Parallel Programming*, Bab 3 (MPI).
4. Siapkan akun Google Colab dan pastikan Anda bisa memilih *runtime* GPU.

Pertemuan depan

Kita pindah dari ratusan *core* besar ke ribuan *core* kecil: arsitektur GPU, model SIMT, dan *kernel* CUDA pertama Anda.

Terima kasih

Pertanyaan?