Unified Framework for Scalable Computing Systems and Algorithms · บทที่ 1

จาก Parallel and Distributed Computing สู่ Scalable Computing

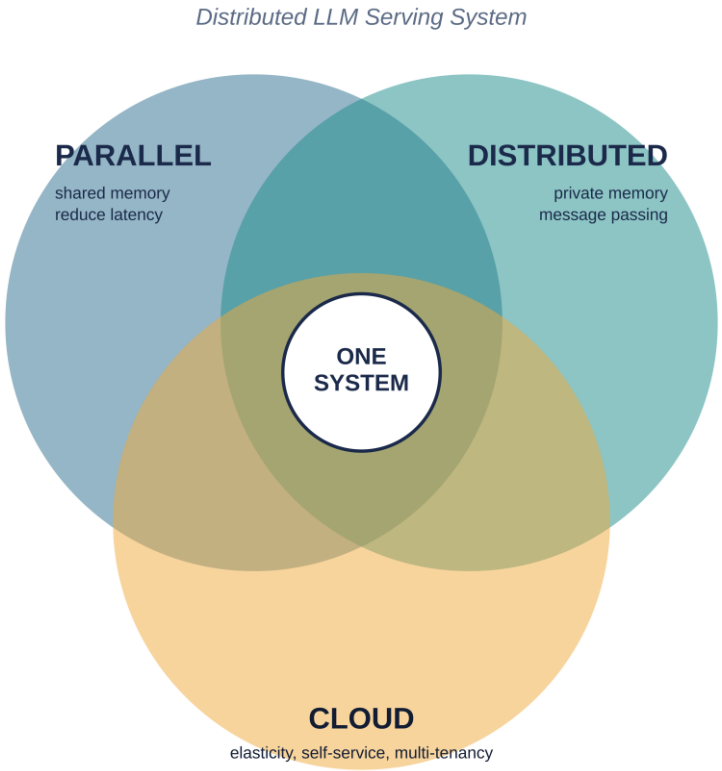
From Parallel and Distributed Computing to Scalable Computing
Part I: Foundations of Unified Scalable Computing

Warin Wattanapornprom PhD.

เนื้อหา

ลองจินตนาการ: แชทบอทระดับโลก

- บริการแชทบอทต้องตอบสนองผู้ใช้หลายล้านคนพร้อมกันตลอด 24 ชั่วโมง
- LLM เบื้องหลังมีพารามิเตอร์สูงถึงหลักแสนล้านตัว — ใหญ่เกินกว่าจะใส่ใน GPU ใบเดียว
- ต้องกระจายโมเดลออกเป็นชิ้นย่อยบน GPU หลายสิบถึงหลายร้อยตัว บางชิ้นอยู่เครื่องเดียวกัน บางชิ้นกระจายข้ามดาต้าเซ็นเตอร์ทั่วโลก
- เมื่อผู้ใช้พุ่งเข้ามาพร้อมกัน ระบบต้องเพิ่ม/ลดเครื่องอัตโนมัติเพื่อคุมต้นทุน
- **คำถาม: ระบบเช่นนี้จัดอยู่ในประเภทใดกันแน่?**



สามมุมมอง สามคำตอบ ระบบเดียวกัน

ถ้ามองผ่านมุม...

- GPU หลายตัวในเครื่องเดียวกัน ใช้ shared memory ร่วมกัน
- ต้องใช้หลายเครื่อง ต่างคน private memory สื่อสารผ่านเครือข่าย
- บริการแบบ on-demand ปรับทรัพยากรอัตโนมัติ ผู้ใช้ไม่รับรู้จำนวนเครื่อง

จะเห็นว่าเป็น...

- → Parallel System
- → Distributed System
- → Cloud System

เหตุใดระบบคอมพิวเตอร์สมัยใหม่จึงไม่สามารถจัดประเภทเป็น "parallel" หรือ "distributed" ได้อย่างเด็ดขาดอีกต่อไป และเราควรทำความเข้าใจด้วยกรอบความคิดใดแทน?

ทำไมป้ายกำกับกว้าง ๆ ถึงวินิจฉัยปัญหาไม่ได้

- **สถานการณ์: ระบบตอบสนองช้า และคำว่า "distributed" เพียงคำเดียวบอกอะไรไม่ได้เลย**
- ปัญหาเกิดจากโครงสร้างงาน (W)? การแบ่งงาน (D)? แบนด์วิดท์เครือข่าย (N)? จุดประสานงาน barrier (O)? หรือการจัดคิวงาน (P)?
- วิศวกรที่ยึดติดชื่อผลิตภัณฑ์ → เปลี่ยนเครื่องมือก่อนรู้สาเหตุจริง
- วิศวกรที่วิเคราะห์จากองค์ประกอบพื้นฐาน → ตั้งสมมติฐานที่ทดสอบได้ เช่น "tail latency เกิดจากรอ replica ที่ช้าที่สุด"
- **กระบวนการวิเคราะห์ 3 ขั้น:**
  1. **บันทึกพฤติกรรมจริงโดยไม่รีบตั้งชื่อ**
  2. **แยกแยะโครงสร้างงาน/ข้อมูล/สื่อสาร/ประสานงาน/ความผิดพลาด**
  3. **เชื่อมโยงกับ architecture, computational model, technology**

เป้าหมายการเรียนรู้ของบทที่ 1

- **1. อธิบายได้ว่าระบบเดียวมีมิติ parallel, distributed และ cloud พร้อมกันได้อย่างไร**
- 2. แยกแยะ Sequential / Concurrent / Parallel / Distributed ด้วยหลักการทางสถาปัตยกรรม ไม่ใช่ความรู้สึก
- 3. เข้าใจข้อจำกัดของ Amdahl's Law และเหตุผลที่เพิ่มโปรเซสเซอร์ไม่ได้แปลว่าเร็วขึ้นเสมอ
- 4. ใช้กรอบคิด Component–Algorithm–Technology และทูเพิล

$$S = \langle W, D, C, M, N, O, P, F, A \rangle \text{ วิเคราะห์ระบบจริง}$$

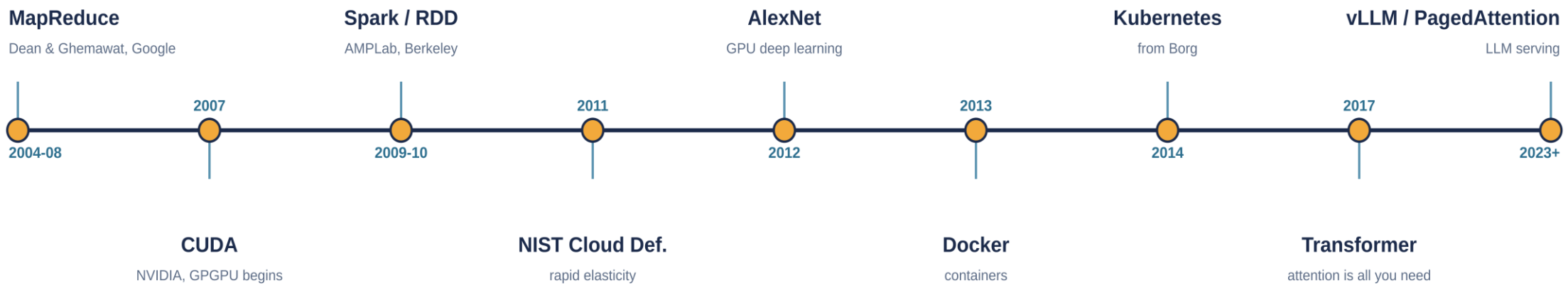
- 5. เดินตามเส้นทางคำขอหนึ่งรายการในระบบ LLM Serving เพื่อเห็นองค์ประกอบทำงานร่วมกัน
- **6. เปลี่ยนวิธีคิดจากการจำแนกชื่อเรียก ไปสู่การวิเคราะห์การตัดสินใจเชิงระบบ**

PART 2

3. จาก Parallel และ Distributed สู่ Scalable Computing

วิวัฒนาการของข้อจำกัดเชิงระบบ · นิยามสี่คำ · Scale-up/out/Elasticity · เพดานความเร็ว

เส้นเวลาแห่งข้อจำกัดทางวิศวกรรม



อ่านเส้นเวลานี้ในฐานะ "ประวัติศาสตร์ของข้อจำกัดเชิงกายภาพ" ไม่ใช่รายชื่อซอฟต์แวร์

บทเรียนจากเส้นเวลา: หน่วยของการขยายขนาดเปลี่ยนไปเรื่อย ๆ

- **เครื่องคอมพิวเตอร์เดี่ยว → โหนดในคลัสเตอร์ → การ์ดเร่งความเร็ว → กลุ่ม GPU ข้ามเครื่องที่ทำงานร่วมกันเสมือนเครื่องเดียว**
- คำถามพื้นฐานทางสถาปัตยกรรมยังเหมือนเดิมทุกยุค:
  - จะแบ่งงานอย่างไร?
  - มอบหมายให้ใครทำ?
  - สื่อสารกันอย่างไร?
  - รับมือความผิดพลาดและความต้องการที่เปลี่ยนแปลงอย่างไร?
- MapReduce สำเร็จเพราะสร้าง abstraction ให้โปรแกรมเมอร์เขียนแค่ map/reduce — รันไทม์จัดการกระจายงาน, retry, รวบรวมผลให้เอง
- **เหตุผลที่ตำราเล่มนี้อธิบายระบบผ่าน "องค์ประกอบ" แทนชื่อผลิตภัณฑ์: เทคโนโลยีเปลี่ยนเร็ว แต่คำถามเชิงองค์ประกอบไม่เปลี่ยน**

สี่คำที่มักถูกใช้ปะปนกัน — แต่นิยามแยกขาดจากกัน

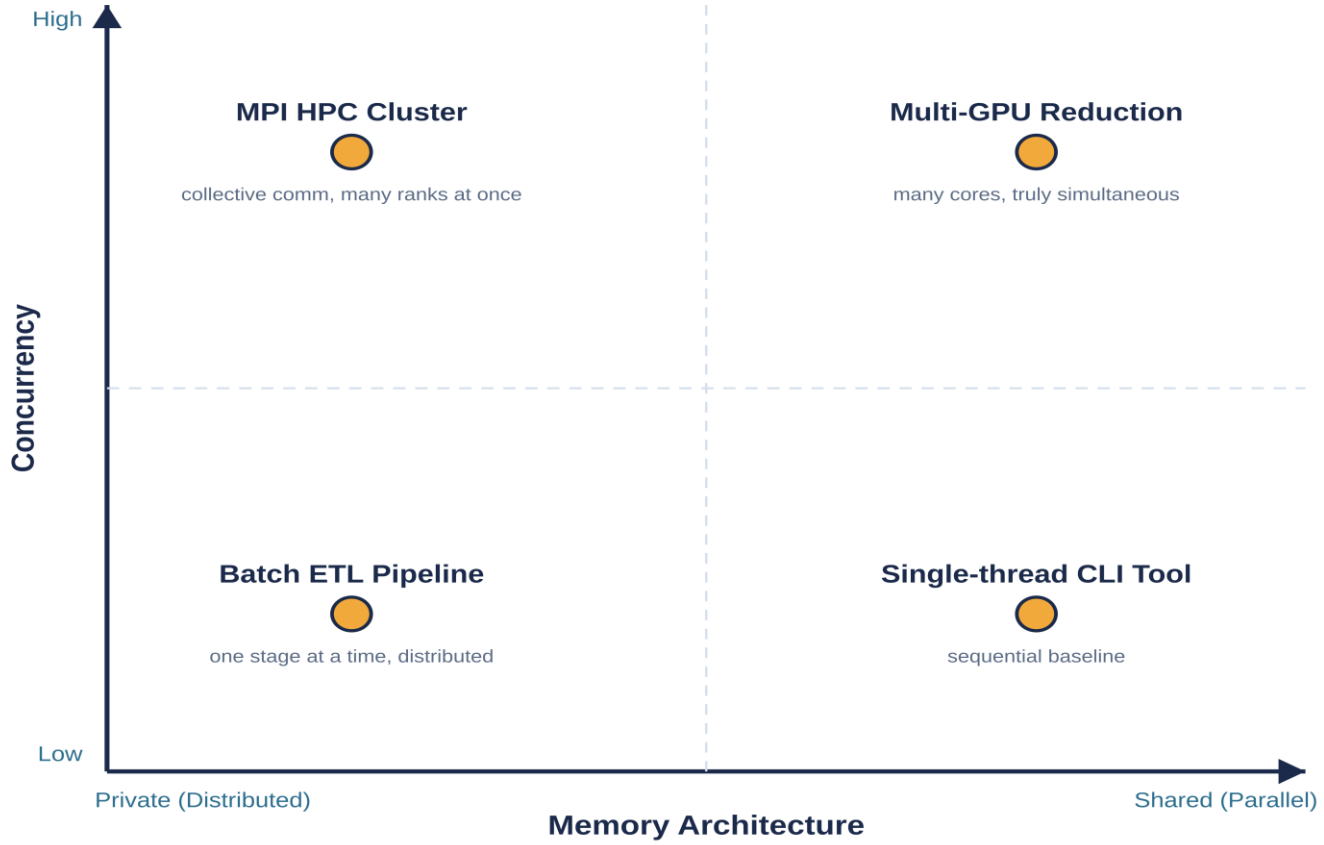
Sequential & Concurrent

- Sequential: คำสั่งทำงานทีละคำสั่งบน CPU เดี่ยว ไม่มีการซ้อนทับเวลา — เป็นเส้นฐาน (baseline) ของทุกการเปรียบเทียบ
- Concurrent: งานหลายสายซ้อนทับกันทางเวลา ไม่จำเป็นต้องเกิดพร้อมกันจริงทางฟิสิกส์ — เรื่องของการจัดการลำดับและ coordination
- ตัวอย่าง: event-driven server รับการเชื่อมต่อหลายพันรายการด้วยเธรดเดียว (single-thread event loop)

Parallel & Distributed

- Parallel: หน่วยประมวลผลหลายตัวที่เข้าถึง shared memory ทำงานพร้อมกันจริง เพื่อลดเวลาของงานเดี่ยว
- Distributed: หน่วยประมวลผลแยกขาด private memory สื่อสารผ่าน message passing เป้าหมายกว้างกว่าแค่ลดเวลา
- **เส้นแบ่งที่แท้จริง: สถาปัตยกรรมหน่วยความจำ (shared vs private) ไม่ใช่จำนวนโหนดหรือความเร็วบัส**

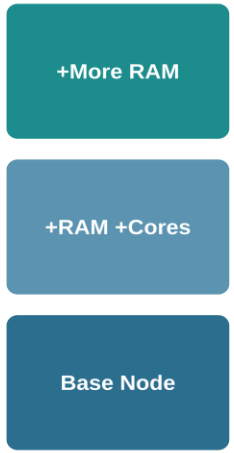
Memory Architecture × Concurrency: สองแกนที่ตั้งฉากกัน



ตัวอย่างที่ตั้งฉาก: MPI cluster (parallel, high concurrency) ต่างจาก batch ETL (distributed, low concurrency)

สองแนวทางหลักในการขยายขนาด

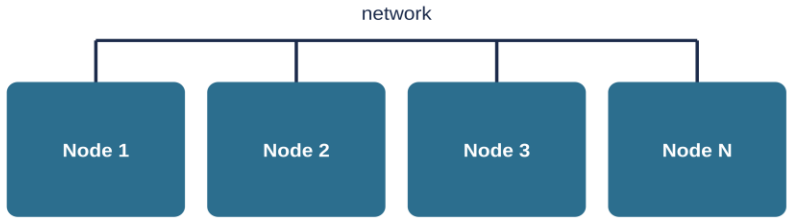
SCALE-UP (Vertical)



one host grows bigger
no cross-node coordination
ceiling: hardware limit, cost curve

then

SCALE-OUT (Horizontal)



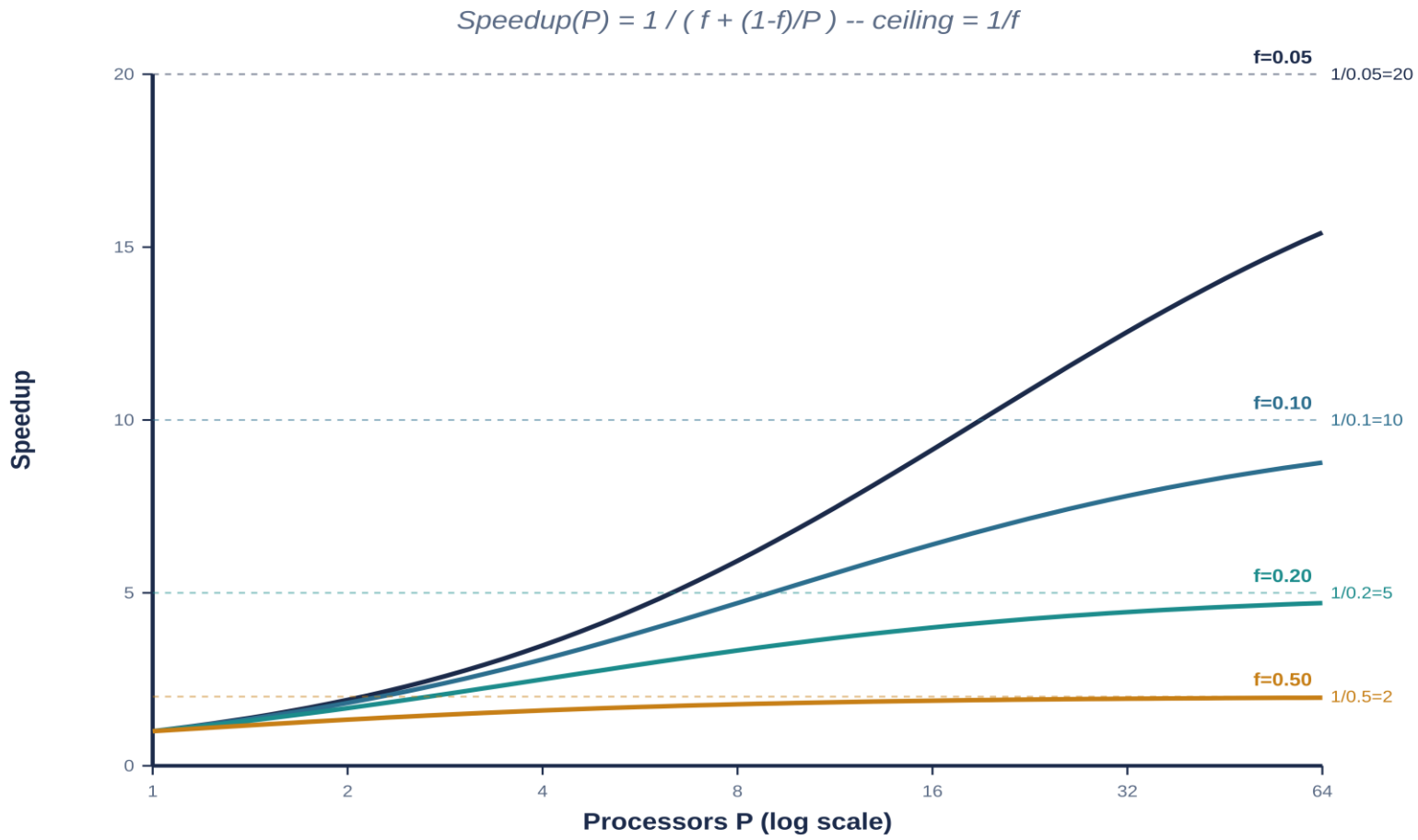
many independent hosts
needs routing, partitioning, replication
near-unlimited, but coordination cost grows

Scale-up: เพิ่มพลังในโฮสต์เดิม (แนวตั้ง) | Scale-out: เพิ่มจำนวนโหนด (แนวนอน)

Elasticity ≠ Scale-out แต่ต่อยอดขึ้นไปอีกชั้น

- **นิยาม NIST: ความสามารถปรับเพิ่ม/ลดทรัพยากรอย่างรวดเร็ว อัตโนมัติ ให้ผู้ใช้รู้สึกเหมือนทรัพยากรไม่จำกัด (Mell & Grance, 2011)**
- Elasticity เพิ่มมิติ "ความเร็วเชิงเวลา" (time responsiveness) และ "กลไกอัตโนมัติแบบปิด" (closed automation loop) เข้าไปบน scale-out
- วงจร Observe → Decide → Actuate เสมอมี actuation delay — เครื่องใหม่ต้องบูตและ warm cache ก่อนพร้อมใช้งาน
- ตั้ง cooldown window สั้นเกินไป → controller สั่งเพิ่ม/ลดสลับกันไม่หยุด (oscillation)
- **กฎเหล็ก: เริ่มจาก Scale-up จนต้นทุนไม่คุ้ม → หา partition boundary เพื่อ Scale-out → ค่อยสวมทับด้วย Elasticity**

กฎของ Amdahl: เพดานเหล็กของ Speedup



Speedup(P) = 1 / (f + (1-f)/P) → เพดานสูงสุด = 1/f ไม่ว่า P จะมากเพียงใด

Gustafson's Law: มุมมองตรงข้ามที่ไม่ขัดแย้งกัน

- **Amdahl (Strong Scaling): ตรึงขนาดปัญหาให้เท่าเดิม แล้วถามว่าเร็วขึ้นแค่ไหนเมื่อเพิ่ม P**
- **Gustafson & Barsis (1988) (Weak Scaling): เมื่อมี P เพิ่ม ผู้ใช้มักขยายขนาดปัญหาตามไปด้วย เพื่อความละเอียดที่สูงขึ้นในเวลาที่ยอมรับได้**
- สัดส่วนงานที่ขนานได้จึงขยายตามขนาดปัญหา ไม่ได้คงที่ตายตัวแบบข้อสมมติของ Amdahl
- *ทั้งสองแบบจำลองไม่ขัดแย้งกัน — เป็นการตั้งคำถามจากมุมมองต่างกัน รายละเอียดเชิงคำนวณเจาะลึกในบทที่ 2*

ต้นทุนการสื่อสารและการประสานงาน — เพดานเหล็กตัวที่สอง

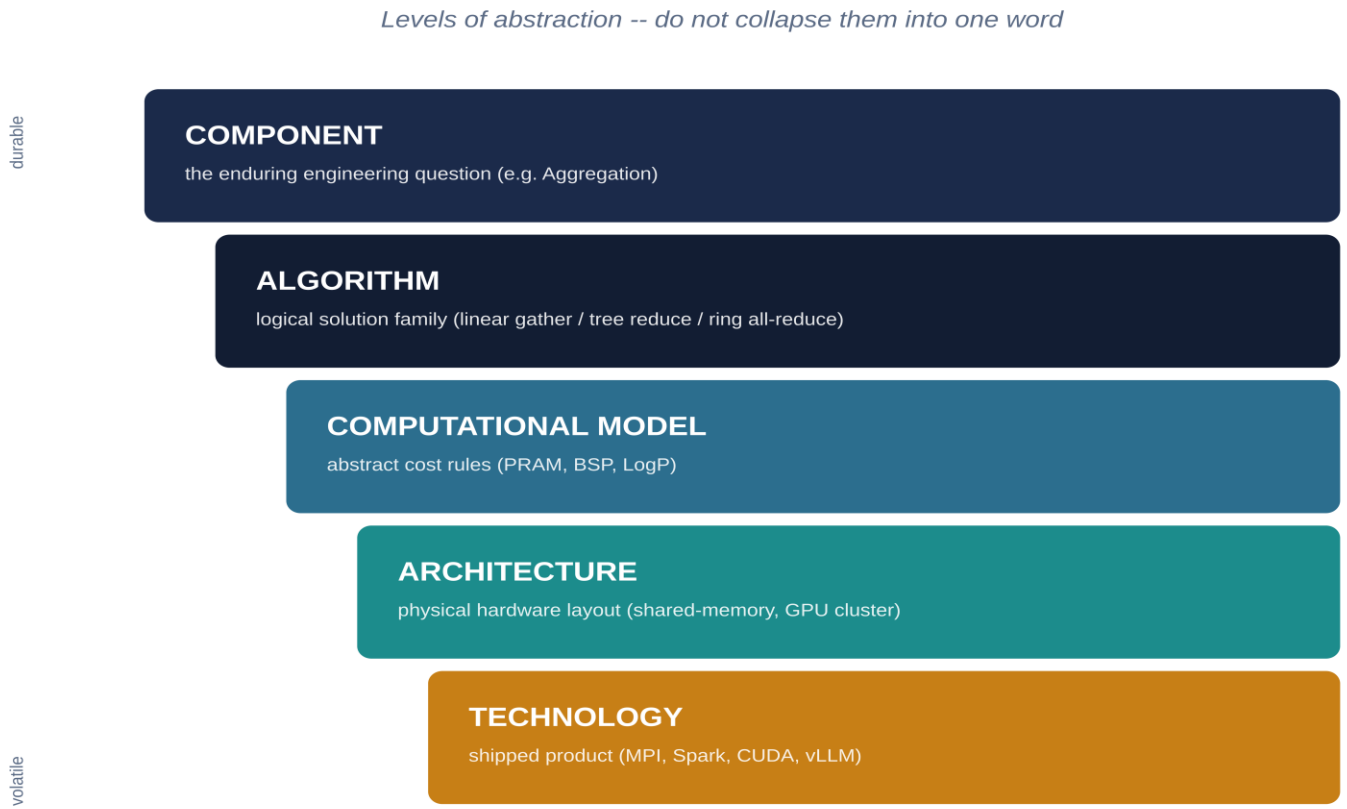
- **เพิ่มโปรเซสเซอร์ → ต้นทุนการสื่อสาร (communication) และการประสานงาน (coordination) เติบโตตามไปด้วย**
- เมื่อระบบสเกลเกินจุดวิกฤต ต้นทุนแฝงนี้โตเร็วกว่าการลดลงของเวลาคำนวณบริสุทธิ์ — เพิ่มเครื่องแล้วช้าลงกว่าเดิม
- **ตัวอย่างตัวเลข: งาน 100 วินาที (20s ลำดับ + 80s ขนานได้) บน 8 โปรเซสเซอร์**

  แบบจำลองอุดมคติ: 20 + 80/8 = 30s → Speedup ≈ 3.33 เท่า

  ระบบจริง (+เครือข่าย 8s +ไม่สมดุล 5s +จัดตาราง 2s): 45s → Speedup เหลือ ≈ 2.22 เท่า

  คอขวดแฝง: lock ของ memory allocator, metadata service จุดเดียว, straggler ที่ทุกคนต้องรอ (barrier)

สี่ระดับชั้นของสิ่งนามธรรม



ยิ่งอยู่บนสุด ยิ่งคงทนข้ามยุคสมัย — ยิ่งอยู่ล่างสุด ยิ่งเปลี่ยนเร็วที่สุด

แยกระดับชั้นของปัญหา "การรวบรวมผลลัพธ์" (Aggregation)

- **Component: โครงสร้างงาน (W) ของการลดรูปข้อมูล, พาร์ทิชัน (D), เครือข่าย (N) ส่งผลย่อย, ประสานงาน (O) ควบคุมลำดับรวมผล**
- Algorithm: linear gather / tree reduction / ring all-reduce — เลือกได้หลายแบบสำหรับปัญหาเดียวกัน
- Computational Model / Architecture: multicore shared memory หรือ GPU cluster private memory
- Technology: MPI collective primitives หรือเฟรมเวิร์กสมัยใหม่
- **ผิด: "Apache Spark scale ได้ดีเพราะ Spark เป็นระบบ Scalable" — นี่คือการให้เหตุผลวนเป็นวงกลม (circular reasoning)**

PART 3

4. กรอบคิดและกระบวนการตัดสินใจเชิงสถาปัตยกรรม

Component–Algorithm–Technology · ทูเพิลนิยามระบบ S = ⟨W,D,C,M,N,O,P,F,A⟩

สามชั้นความคิดที่เชื่อมโยงกันเป็นระบบ

- **Component (องค์ประกอบพื้นฐาน): โจทย์วิศวกรรมสากลที่ทุกระบบต้องตอบ ไม่ว่าจะปี 2008 หรือ 2026**
- **Algorithm (ขั้นตอนวิธี): คำตอบเชิงวิธีการสำหรับแต่ละ component — เลือกได้หลายตระกูลตามข้อสมมติ**
- **Technology (เทคโนโลยี): การประกอบผลลัพธ์ของหลายอัลกอริทึมเข้าด้วยกันเป็นผลิตภัณฑ์ใช้งานจริง**
- MapReduce, Spark, Ray ล้วนตอบคำถามชุดเดียวกันเรื่อง W และ D เพียงแต่เลือกอัลกอริทึมและเป้าหมายต่างกันไปตามยุค
- *ข้อควรระวัง: ไม่ใช่สายโซ่หนึ่งต่อหนึ่ง — component หนึ่งอาจพึ่งพาหลายอัลกอริทึม และอัลกอริทึมเดียวกระทบหลาย component พร้อมกัน*

จากหลักฐานเชิงพฤติกรรม สู่การเติมทูเพิล

- **รวบรวมหลักฐาน: task graph, data ownership, network trace, scheduler logs, failure logs**
- เติมค่าลงในช่องว่างทูเพิล พร้อมระบุส่วนที่ยังไม่รู้ (unknowns) — หลีกเลี่ยงการเดาจากคำโฆษณา
- กรณีศึกษา: Distributed Cache ใช้ Consistent Hashing → ลด P, N แต่ถ้าเจอ Hot Key ต้องเพิ่ม Replication → เพิ่มต้นทุน O
- Recompose ตัวอย่าง: ข้อมูลห้ามออกจากอุปกรณ์ผู้ใช้ → Centralized Training แตกหัก → ต้อง Federated Learning (ปรับ N, เพิ่ม C ท้องถิ่น, กระทบ O และ F)
- **อย่าตกหลุมพราง "Cargo-cult Architecture" — ลอกสถาปัตยกรรมสำเร็จโดยไม่เข้าใจข้อสมมติเบื้องหลัง**

S = ⟨W, D, C, M, N, O, P, F, A⟩

ทูเพิลนี้คือรายการตรวจสอบการตัดสินใจ (checklist of decisions) ไม่ใช่สมการที่คำนวณตัวเลข

W, D, C, M, N — จากงานสู่การสื่อสาร

โครงสร้างและการจัดวางงาน

- W — Work Structure: โครงสร้างงานและความสัมพันธ์รอกัน (dependency graph, critical path)
- D — Decomposition: หน่วยของการแบ่งงาน/ข้อมูลและระดับ granularity
- C — Compute Organization: โครงสร้างฮาร์ดแวร์ประมวลผล (CPU/GPU/multicore)

หน่วยความจำและการสื่อสาร

- M — Memory and State: ข้อมูลถูกเก็บที่ไหน ใครเป็นเจ้าของ ระดับความสอดคล้อง
- N — Communication: รูปแบบข้อความ ขนาด topology และแบนด์วิดท์
- **W ต้องพิจารณาก่อน D เสมอ — หั่นข้อมูลโดยไม่รู้เส้นทางวิกฤตของ W คือการสร้างงานย่อยที่ไม่ช่วยเพิ่มความขนานจริง**

O, P, F, A — จากการประสานงานสู่การปรับตัว

ประสานงานและจัดวาง

- O — Coordination: การประสานลำดับ, barrier, ความถูกต้องเชิงตรรกะของการเรียงลำดับ
- P — Placement & Scheduling: จัดวางงาน/ข้อมูลลงโฮสต์ไหน เมื่อไร ด้วยนโยบายใด

ความผิดพลาดและการปรับตัว

- F — Failure and Recovery: ข้อสมมติความล้มเหลว ขอบเขตกักกันความเสียหาย กลไกกู้คืน
- A — Adaptation & Elasticity: สลับอัลกอริทึม, repartition ระหว่างรัน, adaptive batching, load shedding
- **ห้ามเขียนแค่ "M = Redis" ในช่องทูเพิล — ต้องเขียนคุณสมบัติเชิงวิศวกรรมที่ตรวจสอบได้จริง**

Distributed LLM Serving System ในทูเพิล S

ตัวแปร	ประเด็น	รายละเอียด
W	Autoregressive Dependency	การสร้างคำถัดไปต้องรอคำก่อนหน้าเสมอ จำกัดเพดานความเร็วการสร้างคำ
D	Tensor + Pipeline Parallelism	กระจายพารามิเตอร์โมเดลภายในเครื่อง (tensor) และข้ามโหนด (pipeline)
C	GPU Cluster	ทำงานบนกลุ่มฮาร์ดแวร์ประมวลผลกราฟิก
M	PagedAttention	บริหาร KV-cache ผ่านสิ่งนามธรรมแบ่งหน้าหน่วยความจำ
N	All-reduce Collective	ขนส่งข้อมูลผ่านการสื่อสารกลุ่มความเร็วสูง
O	Token & Pipeline Ordering	ควบคุมลำดับถูกต้องของโทเค็นและจังหวะแต่ละสเตจ
P	Continuous Batching	จัดคิวคำขอและรวมชุดคำขอคำนวณแบบต่อเนื่อง
F	Retry + Fault Tolerance	รันงานใหม่และทนทานต่อความล้มเหลวของ GPU
A	Replica Autoscaling	เพิ่ม/ลดเครื่องสำเนาตามความหนาแน่นคำขอ

PART 4

5. จากแบบจำลอง สู่ระบบที่ใช้งานจริง

20 ขั้นตอนวินิจฉัย · ตระกูลอัลกอริทึม · เส้นทางคำขอ · 6 กรณีศึกษาอุตสาหกรรม

20 ขั้นตอนการวินิจฉัยเชิงรูปนัย — สี่ระยะหลัก

Twenty-step formal diagnostic -- turn "it doesn't scale" into a testable hypothesis

เป้าหมาย: เปลี่ยนคำว่า "ระบบไม่ scale" ให้กลายเป็นสมมติฐานที่วัดค่าได้จริง

ขั้นที่ 1–4: อย่ารีบแก้ ก่อนจะรู้ว่าอาการคืออะไร

ขั้น 1 — เริ่มจากอาการ ไม่ใช่วิธีแก้: บันทึกพฤติกรรมเชิงประจักษ์ (เช่น p99 latency พุ่งเมื่อ >2,000 req/s) หลีกเลี่ยงคำว่า "ไม่ scale" ลอย ๆ

ขั้น 2 — กำหนดลักษณะภาระงาน: arrival pattern, input-size distribution, read/write ratio, burstiness — ค่าเฉลี่ยเดียวซ่อน skew และ tail latency

ขั้น 3 — วัตถุประสงค์และข้อจำกัดเข้มงวด: latency/throughput/ต้นทุน/พลังงาน — ข้อจำกัดทางกายภาพห้ามแปลงเป็นคะแนนเฉลี่ยชดเชย

ขั้น 4 — สร้างเส้นฐานที่ยุติธรรม: เปรียบเทียบกับ sequential version ที่ปรับแต่งดีแล้ว ไม่ใช่โค้ดหละหลวมที่ทำให้ speedup ดูสูงเกินจริง

ขั้นที่ 5–8: วาดโครงสร้าง ตรวจ Compute/Memory และ Network/Coordination

ขั้น 5 — วาดโครงสร้างงาน (W): dependency, critical path — ถ้าเส้นทางวิกฤตเป็นลำดับ เพิ่มเครื่องไม่ช่วย

ขั้น 6 — ตรวจการแบ่งงาน/ข้อมูล (D): หน่วยพาร์ทิชัน granularity ปัญหา data skew — ไล่กวาดขนาดพาร์ทิชันแทนการเดา

ขั้น 7 — ตรวจ Compute (C) และ Memory (M) ร่วมกัน: arithmetic intensity เทียบแบนด์วิดท์หน่วยความจำจริง

ขั้น 8 — แยก Network (N) ออกจาก Coordination (O): ข้อความเล็กแต่ช้าเพราะรอ barrier ≠ ปัญหาแบนด์วิดท์เครือข่าย

ขั้นที่ 9–11: การจัดวาง ความผิดพลาด และการปรับตัว

ขั้น 9 — ตรวจการจัดวางและจัดตาราง (P): queue, placement, affinity, stragglers — ข้อมูลใหญ่เกินไป→repartition D, ฮาร์ดแวร์ช้า →speculative execution

ขั้น 10 — ตรวจกลไกความผิดพลาดและการกู้คืน (F): steady-state metric นับรวมเวลา timeout/retry/recovery แล้วหรือยัง?

ขั้น 11 — ตรวจระบบปรับตัวยืดหยุ่น (A): control signal, observation window, hysteresis, actuation delay — autoscaler ที่บูตช้ากว่าโหลดพุ่งเข้ามา ช่วยอะไรไม่ได้

ขั้นที่ 12–20: จากสมมติฐาน สู่บันทึกการตัดสินใจ

ขั้น 12 ตั้งสมมติฐานเชิงเหตุผล
ขั้น 13 แยกตัวแปรทดลองทีละกลุ่ม
ขั้น 14 วิเคราะห์ผลต่างที่เหลือ (residual)

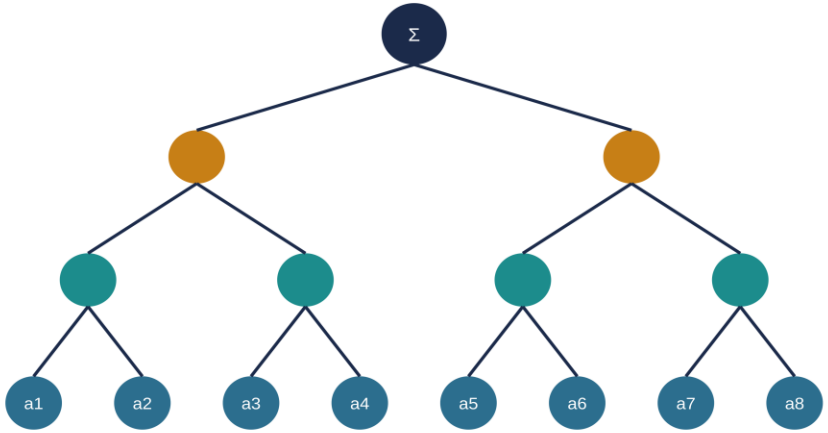
ขั้น 15 เขียนข้อดี/เสียข้ามองค์ประกอบ (benefit, direct cost, shifted cost)
ขั้น 16 ประเมินขอบเขตภูมิภาคการทำงาน (operating envelope)

ขั้น 17 ตรวจกรณีศึกษาสำเร็จอย่างวิพากษ์ (5 มิติ: โจทย์, ข้อจำกัดยุคนั้น, การตัดสินใจ, กลไกรันไทม์, ขอบเขต)
ขั้น 18 Recomposition — จงใจเปลี่ยนข้อจำกัดแล้วดูว่าอัลกอริทึมเดิมยังเหมาะสมไหม
ขั้น 19 สื่อสารความไม่แน่นอนตรงไปตรงมา — เลี่ยงคำว่า "เสมอ" "ดีที่สุด" "รับประกัน"
ขั้น 20 บันทึกการตัดสินใจ (ADR): เหตุผล ทางเลือกที่ปฏิเสธ และตัวกระตุ้นให้หมดอายุ

หาผลรวมตัวเลข n ตัว: 3 แบบจำลองความคิด

Sequential: span = O(n) -- n-1 additions, one after another

Parallel Tree Reduction (shared memory): span = O(log n) -- trades hardware for time



Work stays O(n);
span shrinks to O(log n)
but pays synchronization
overhead at every level

Sequential O(n) span → Parallel tree reduction O(log n) span (แลกด้วยต้นทุน sync) → Distributed MapReduce-style (แลกด้วยต้นทุนเครือข่าย + ความเสี่ยงเครื่องล่ม)

เส้นทางเดินของหนึ่งคำขอคำนวณ (Request Trace Path)

One request, ten components -- observed latency is the sum of every stage

Total latency = queue wait + prefill + decode + coordination wait + network -- not just kernel time

Distributed LLM Serving System — 10 สเตจ ตั้งแต่ Edge Gateway จนถึง Adaptation & Scaling

จาก Edge Gateway ถึง Continuous Batching

1. Edge Gateway: ตรวจสอบสิทธิ์ จำกัดอัตรา จัดเส้นทางภูมิภาค — ต้องรักษา idempotency ป้องกันคำขอซ้ำจาก retry
2. Service Discovery & Routing: routing table เป็นข้อมูลแฝงเก่าเสมอ (stale) ต้องมี staleness tolerance และเส้นทางสำรอง
3. Tokenization & Admission Control: tokenize บน CPU อาจกลายเป็นคอขวดถ้า GPU เร็วมาก; admission ประมาณ KV-cache ล่วงหน้าเพื่อป้องกัน OOM
4. Continuous Batching Scheduler: FCFS ยุติธรรมแต่เสี่ยงคำขอยาวถ่วง Throughput; SJF ลด wait time เฉลี่ยแต่เสี่ยง starvation

จาก Paged Memory ถึง Collective Communication

5. Paged Memory Allocation: แบ่ง KV-cache เป็นบล็อกหน้าความจำ ลด external fragmentation แต่ต้องแลกกับ metadata lookup overhead

6. Prefill Stage & Tensor Parallelism: คำนวณเมทริกซ์ขนานสูง — ต้องใช้ topology-aware scheduler ไม่ใช่แค่นับ GPU ว่าง

7. Collective Messaging (All-reduce): ทุก rank ต้องรันคำสั่งตรงจังหวะกัน — หลุดจังหวะแม้เสี้ยววินาทีเดียว = deadlock ทั้งคลัสเตอร์

- **แยกให้ชัด: N ถามเรื่องไบต์/แบนด์วิดท์ ส่วน O ถามเรื่องความถูกต้องของลำดับ (ordering correctness)**

จาก Decode Stage ถึง Adaptation & Scaling

8. Decode Stage & Backpressure: Autoregressive บังคับรอทีละคำ → span ยาวตามความยาวคำตอบ; pipeline bubble ที่ต้นและปลายท่อ

9. Observability, Failure & Recovery: ต้องวัดแยกตาม diagnostic equation ไม่ใช่แค่ GPU utilization เดี่ยว ๆ — ต้องตัดสิน fail-open vs checkpoint

10. Adaptation Control & Drain Protocol: โหลดโมเดลใหม่ใช้เวลานาที (startup delay สูง) → ต้องมี warm pool; scale-in ต้องรอ drain คำขอเดิมให้เสร็จก่อนปิดเครื่อง

- **บทสรุปเส้นทางคำขอ: latency ที่ผู้ใช้เห็น = ผลรวมซ้อนทับของทุกสเตจ — ปรับ kernel เร็วขึ้น 20% ไม่มีความหมายถ้า 90% เวลาไปติดคิว**

หกกรณีศึกษา: วัตถุประสงค์ต่างกัน → ทูเพิลต่างกัน

กรณีศึกษาเหล่านี้ไม่ได้มีไว้สรรเสริญผลิตภัณฑ์ใด แต่แสดงว่าเมื่อวัตถุประสงค์และข้อจำกัดเปลี่ยน การตัดสินใจในทูเพิล S ต้องเปลี่ยนตามอย่างหลีกเลี่ยงไม่ได้

1. **Flash Sale E-commerce — ห้ามขายเกินสต็อก + ต้องชำระเงินสำเร็จ**
2. **Climate Simulation HPC — ลดเวลาคำนวณให้สั้นที่สุด (Time-to-Solution)**
3. **Genomics Pipeline — Throughput สะสมสูงสุด + reproducibility 100%**
4. **Global Video Streaming — ไม่มีอาการภาพชะงัก + startup delay ต่ำ**
5. **Financial Ledger & Payments — ความถูกต้องสัมบูรณ์ของยอดเงิน**
6. **Edge AI Factory Detection — แจ้งเตือนระดับมิลลิวินาที + ทำงานได้แม้เน็ตขาด**

Flash Sale · Climate HPC · Genomics

Flash Sale E-commerce

- W/D: แยก critical path ชำระเงินออกจากงานเสริม; แบ่งสต็อกเป็นถังย่อยแก้ hot key
- O/F: Idempotency Key + Conditional Update; Outbox/Inbox pattern กันธุรกรรมซ้ำ
- A: Pre-scale warm fleet ก่อนเทศกาล + queue depth signal

Climate HPC & Genomics Pipeline

- Climate: Stencil dependency ข้ามก้าวเวลา → เพิ่มคอร์ไม่ช่วยตัด dependency; domain decomposition ชั่งพื้นที่ผิว vs จำนวนชิ้นงาน
- Genomics: DAG งานอิสระข้ามตัวอย่าง แต่ precedence เข้มงวดในตัวอย่างเดียว; recompute เฉพาะพาร์ทิชันที่หาย ไม่รันใหม่ทั้งหมด

Video Streaming · Financial Ledger · Edge AI

Video Streaming & Financial Ledger

- Video: ตัดเป็น segment สั้นเพื่อปรับ bitrate ไว; Hierarchical CDN placement — ยอดนิยมกระจายทั่วโลก เก่าเก็บกลับส่วนกลาง
- Financial: Append-only ledger เท่านั้นที่เป็น source of truth; Saga pattern แทน distributed transaction ใหญ่; Fail-open vs Fail-closed ตามความเสี่ยง

Edge AI Factory Detection

- Local inference ลด N ให้เหลือน้อยที่สุด ส่งเฉพาะ feature/event สรุป ไม่ส่งวิดีโอดิบ
- Store-and-forward รองรับเครือข่ายติด ๆ ขัด ๆ; สัญญาณเตือนภัยแซงคิวข้อมูลสถิติ
- A ปรับที่ตัวอุปกรณ์เอง: ลด frame rate, ลดความละเอียด, สลับโมเดลเล็กลง แทนการเพิ่มเครื่อง (ติดข้อจำกัดกายภาพ)

หกอุตสาหกรรม หกชุดการตัดสินใจ

อุตสาหกรรม	SLO หลัก	ข้อจำกัดกายภาพ	การตัดสินใจแกนนำ	ต้นทุนที่ยอมจ่าย
LLM Serving	TTFT + Inter-token latency	แรม GPU + autoregressive order	PagedAttention / Continuous Batching	Metadata lookup + scheduler ซับซ้อน
Flash Sale	อัตราชำระเงินสำเร็จ	ความถูกต้องสต็อก + Hot Key	แยกงานเสริม / Idempotency Key	ชะลอ/ปฏิเสธคำขอ
Climate HPC	Time-to-Solution สั้นสุด	Topology บัส + critical path	Domain Decomposition + block layout	แรงงานเขียนโค้ดระดับล่าง
Genomics	Throughput สะสมสูงสุด	ไฟล์ใหญ่ + reproducibility	Data Lineage + Atomic Commit	พื้นที่เก็บไฟล์กลาง + บีบอัด
Video Streaming	Zero Rebuffering	แบนด์วิดท์ + สัญญาณผันผวน	Segment split + Hierarchical CDN	พื้นที่สำเนาหลายระดับ
Edge AI Factory	แจ้งเตือน ms + local autonomy	พลังงาน/ความร้อน/เน็ตขาด	Local inference + Dynamic policy	ความแม่นยำโมเดลลดลง

PART 5

6–9. Recomposition, ข้อแลกเปลี่ยน และบทสรุป

เมื่อบริบทเปลี่ยน ทูเพิลต้องเปลี่ยนตาม · ความเข้าใจผิดที่พบบ่อย · ไวยากรณ์แห่งการแลกเปลี่ยนต้นทุน

ย้าย LLM Serving ไปอยู่บน Edge Device เครื่องเดียว

- **ข้อจำกัดใหม่: แรม 8GB, พลังงานจำกัด, ไม่มีเครือข่ายภายนอกเลย, ความเป็นส่วนตัวสูงสุด, ยังต้องรักษา interactive latency SLO**
- N, O → ลดเหลือศูนย์ระหว่างเครื่อง (ไม่มีเครื่องอื่นให้คุยด้วย) แต่ internal communication ภายในบอร์ดยังมีอยู่จริง
- D, M → เลิก Tensor/Pipeline Parallelism ข้ามเครื่อง → ใช้ Model Quantization / Pruning / Weight Offloading แทน
- W → เพิ่ม Speculative Decoding (draft model เล็กนำ + model ใหญ่ตรวจสอบขนาน) ลด span แต่เพิ่ม work รวม
- F → จาก node failure ข้ามเครือข่าย เปลี่ยนเป็น OOM crash, ไฟล์โมเดลเสีย, แบตหมด, ความร้อนสูง — restart จาก immutable image
- A → ไม่มีสิทธิ์เรียกเครื่องเพิ่มจากคลาวด์ → ปรับ batch size, ตัด context length, ลด precision อัตโนมัติแทน

จุดอิ่มตัวเชิงประสิทธิภาพ (Performance Saturation Point)

- **เพิ่ม GPU ในระบบ LLM Serving ไม่ได้แปลว่า latency ลดลงเสมอไป**
- Tensor Parallelism บังคับให้เกิด All-reduce ทุกชั้นของโมเดล — ยิ่ง GPU มาก ต้นทุนสื่อสารต่อรอบยิ่งพุ่งตามสถาปัตยกรรมเครือข่าย
- ถ้าอัตราลดของงานคำนวณต่อชิปช้ากว่าอัตราเพิ่มของต้นทุนสื่อสาร → ระบบชนจุดอิ่มตัว → เพิ่ม GPU กลับทำให้ช้าลงกว่าเดิม
- **นี่คือ Amdahl's Law ในบริบทซอฟต์แวร์สเกลสูง: f ไม่ได้จำกัดแค่โค้ดลำดับ แต่รวมเวลารอ synchronization ด้วย**
- *แยก TTFT (compute-bound, ได้ประโยชน์จาก batch ใหญ่) ออกจาก Inter-Token Latency (memory-bandwidth-bound, ไวต่อกลยุทธ์คิว) — อย่ารายงานค่าเฉลี่ยรวม*

สี่ความเข้าใจผิดที่พบบ่อยที่สุด

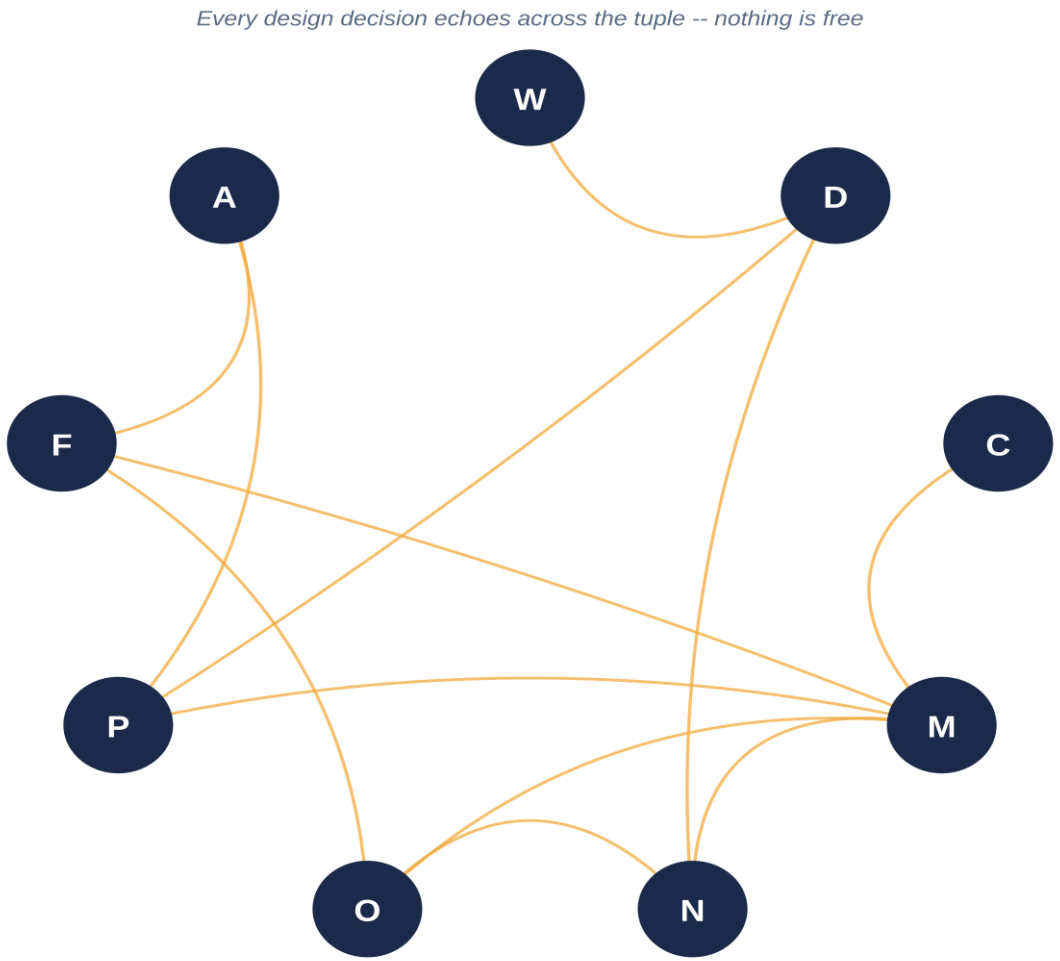
เข้าใจผิดที่ 1–2

1. "Parallel กับ Distributed คือเรื่องเดียวกัน" — ผิด: แบ่งด้วย memory architecture ไม่ใช่จำนวนโหนด
2. "เพิ่มโหนด/GPU จะเร็วขึ้นเสมอ" — ผิด: ขึ้นกับ critical path ratio และอัตราเติบโตของต้นทุนแฝง

เข้าใจผิดที่ 3–4

3. "Cloud คือแค่ชื่อใหม่ของ Distributed" — ผิด: Cloud เพิ่ม elasticity, multi-tenancy, pay-per-use
4. "LLM Serving ต้องใช้ความรู้ชุดใหม่ทั้งหมด" — ผิด: PagedAttention คือโจทย์ M แบบเดิมภายใต้ข้อจำกัดใหม่

The Grammar of Cross-Component Trade-offs



ทุกการตัดสินใจในทูเพิลส่งเสียงสะท้อนไปยังตัวแปรอื่นเสมอ — ไม่มีการปรับปรุงใดที่ไม่มีต้นทุน

ทุกความสำเร็จ = ยอมแลกอะไรบางอย่างเสมอ

- MapReduce: ยอมจำกัดความยืดหยุ่นการเขียนโปรแกรม แลกกับการจัดตารางงาน + กู้คืนความเสียหายอัตโนมัติ
- Apache Spark: ปรับ W และ M เพื่อเก็บ working set ไว้ในแรม ลดต้นทุนงานวิเคราะห์แบบวนซ้ำมหาศาล
- CUDA: เปิด Throughput มวลชนใน C แต่บังคับโปรแกรมเมอร์จัดระเบียบ M ให้สม่ำเสมอ (coalesced access)
- Kubernetes: จัดระเบียบ P ผ่าน Reconciliation Loop บนสิ่งนามธรรม container
- vLLM: พลิกโฉม M ด้วย PagedAttention เปิดทางให้ P ทำ Continuous Batching เต็มประสิทธิภาพ
- **นวัตกรรมไม่ได้เกิดจากการคิดค้นสิ่งใหม่จากความว่างเปล่า แต่เกิดจากการเข้าใจหลักการเดิมอย่างลึกซึ้ง แล้ว Recompose ใหม่ให้เหมาะกับข้อจำกัดยุคนั้น**

คำตอบใหม่สำหรับกรณีศึกษา LLM Serving

- *คำถามเปิดบท: ระบบนี้เป็น parallel, distributed หรือ cloud กันแน่?*
- **คำตอบใหม่: "ระบบนี้ใช้สถาปัตยกรรม distributed-memory ระหว่างโฮสต์ ขับเคลื่อนด้วยวัตถุประสงค์ parallel computation เพื่อลดเวลาโมเดลเดี่ยว บริหาร concurrency ของคำขอที่เข้ามาพร้อมกัน และพึ่งพา cloud provisioning เพื่อปรับเครื่องสำเนาตามเวลาจริง"**
- คำอธิบายระดับนี้คมชัดพอที่จะชี้ทางตั้งคำถามวิจัยและออกแบบการทดลองต่อยอดได้อย่างมีหลักการ

แบบฝึกหัดสามระดับ

- ระดับพื้นฐาน: จำแนก sequential/concurrent/parallel/distributed ของ 4 ระบบตัวอย่าง (event-driven server, 4-GPU workstation, Hadoop cluster, CDN) พร้อมเหตุผลเชิงสถาปัตยกรรม
- ระดับประยุกต์: วิเคราะห์ระบบแนะนำสินค้าอีคอมเมิร์ซว่าควร scale-up/out/ผสม พร้อมบทบาทของ Elasticity ช่วง flash sale; ระบุ bottleneck ที่เสี่ยงที่สุดในทูเพิล LLM Serving
- ระดับสังเคราะห์: ถอดรหัสเทคโนโลยีจริง (Kubernetes/Spark/Ray) เป็นทูเพิล S คร่าว ๆ; ออกแบบสถานการณ์ที่ระบบเดียวกันถูกจัดเป็น parallel หรือ distributed สลับกันตามมุมมอง
- *ทุกคำตอบต้องมี: หลักฐานเชิงประจักษ์ ≥ 1, สมมติฐานเชิงสถาปัตยกรรม ≥ 1, คู่ trade-off ≥ 1*

เลือกเส้นทางเรียนรู้ตามความเชี่ยวชาญของตัวเอง

Algorithm & Systems Focus

- สาย Algorithm: เริ่มจาก W และ D — ขอบเขตล่างทางทฤษฎีของงาน แล้วค่อยดูว่า C เปลี่ยน cost model อย่างไร
- สาย Systems: เดินตาม Request Path Trace จริง บันทึกพิกัดที่คิวงาน สถานะ ข้อความ และความเสียหายปรากฏตัว แล้วจับคู่ลงทูเพิล

Cloud & Data/AI Focus

- สาย Cloud: แยก Provisioning Plane (A) ออกจาก Application Behavior — autoscaling บน stateful system ที่ไม่มี drain protocol ไม่ปลอดภัย
- สาย Data/AI: แยก Training (throughput-driven, collective gradient) ออกจาก Serving (queue-driven, tail latency) อย่างเด็ดขาด

เกณฑ์การประเมินผลเชิงวิเคราะห์

Assessment Rubric — ให้น้ำหนักที่คุณภาพเหตุผล ไม่ใช่ตัวเลขเดี่ยว

มิติการประเมิน	คำอธิบาย
ความแม่นยำเชิงแนวคิด (Conceptual Precision)	แยกปัญหาเครือข่ายจริง (N) ออกจากปัญหาการประสานงานเชิงตรรกะ (O) ได้อย่างมีหลักการ
ประกาศข้อสมมติชัดเจน (Explicit Assumptions)	ระบุคุณสมบัติและขอบเขตของภาระงาน/ฮาร์ดแวร์ก่อนวิเคราะห์ หลีกเลี่ยงการสรุปเหมารวม
เชื่อมโยงต้นทุนข้ามองค์ประกอบ (Cross-Component Mapping)	อธิบายได้ว่าการตัดสินใจที่โหนดหนึ่งส่งผลดี/ร้ายแฝงกลับไปโหนดอื่นในทูเพิลอย่างไร
ซื่อสัตย์ต่อขอบเขตความไม่แน่นอน (Uncertainty Awareness)	รายงาน operating region อย่างมีหลักการ หลีกเลี่ยงถ้อยคำการันตีแบบไร้เงื่อนไข

รายการอ้างอิง

References

- Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. AFIPS Conference Proceedings, 30, 483–485.
- Burns, B., Grant, B., Oppenheimer, D., Brewer, E., & Wilkes, J. (2016). Borg, Omega, and Kubernetes. Communications of the ACM, 59(5), 50–57.
- Dean, J., & Ghemawat, S. (2008). MapReduce: Simplified data processing on large clusters. Communications of the ACM, 51(1), 107–113.
- Grama, A., Gupta, A., Karypis, G., & Kumar, V. (2003). Introduction to Parallel Computing (2nd ed.). Addison-Wesley.
- Gustafson, J. L. (1988). Reevaluating Amdahl's law. Communications of the ACM, 31(5), 532–533.
- Kwon, W., et al. (2023). Efficient memory management for LLM serving with PagedAttention. Proceedings of SOSP '23.
- Mell, P., & Grance, T. (2011). The NIST Definition of Cloud Computing (NIST SP 800-145).
- Rauber, T., & Rünger, G. (2013). Parallel Programming: For Multicore and Cluster Systems (2nd ed.). Springer.
- Zaharia, M., et al. (2010). Spark: Cluster computing with working sets. Proceedings of HotCloud '10.

จบบทที่ 1

คำถามและอภิปราย

สัปดาห์หน้า: บทที่ 2 — แปลงเหตุผลเชิงคุณภาพให้เป็นเครื่องมือเชิงปริมาณ (Latency, Throughput, Speedup, Efficiency, Strong/Weak Scaling)