LANTA

LANTA ON THE MOVE: NORTH

HPC EXPERIENCE DAY · NORTHERN REGION 2026

LANTA HPC HANDBOOK

LANTA HPC Handbook

A practical guide to Slurm jobs, scientific models, MPI, AI workflows, and applications of HPC

Login · files · modules · job scripts · monitoring · results · next experiment

CHIANG MAI, THAILAND

29 July–8 August 2026

NARIT, Mae Rim

Online pre-school and mentoring

Onsite: 3 August 2026

ThaiSC NARIT

BUILD DURING THE DAY

First job · MPI sample · GPU check · evidence pack

คำนำ

LANTA HPC Experience Day: On the Move

หนังสือเล่มนี้จัดทำขึ้นเพื่อเป็นคู่มือสำหรับผู้เข้าร่วมโครงการ LANTA HPC Experience Day: On the Move โดยเฉพาะ เนื้อหาจะพาทุกคนไปรู้จักและทดลองใช้การคำนวณสมรรถนะสูง หรือ HPC บนซูเปอร์คอมพิวเตอร์ “LANTA” ซึ่งเป็นโครงสร้างพื้นฐานด้านการคำนวณทางวิทยาศาสตร์ของประเทศไทย

เป้าหมายของโครงการนี้ คือ รวบรวมภาพปัจจุบันให้ทุกคนได้เห็นว่า งานคำนวณบนซูเปอร์คอมพิวเตอร์นั้นเป็นฐานของการพัฒนาเศรษฐกิจและสังคม ด้วยการนำวิทยาศาสตร์เชิงคำนวณ ข้อมูลคุณภาพสูง มาประยุกต์ร่วมกับปัญญาประดิษฐ์ เพื่อสร้างนวัตกรรม

ประเทศไทยลงทุนในโครงสร้างพื้นฐานดิจิทัลเพิ่มขึ้นอย่างรวดเร็ว ในปี 2568 มีคำขอส่งเสริมการลงทุนด้านศูนย์ข้อมูลรวม 36 โครงการ มูลค่ากว่า 728,000 ล้านบาท แปลว่าเราต้องมีคนที่สามารถนำทรัพยากรเหล่านี้ไปสร้างงานวิจัย ผลิตภัณฑ์ และนวัตกรรมได้จริง เพื่อใช้ประโยชน์จากเครื่องและศูนย์ข้อมูลที่ดึงทรัพยากรธรรมชาติไปสร้างนี้ให้คุ้มค่าที่สุด ในเวลาเดียวกัน แรงงานไทยร้อยละ 74.1 ยังมีทักษะดิจิทัลพื้นฐานต่ำกว่าเกณฑ์ และประเทศต้องการบุคลากรทักษะสูงอีกกว่า 280,000 คนภายในปี 2573 การเรียนรู้ HPC จึงไม่ใช่เรื่องไกลตัว แต่เป็นส่วนหนึ่งของการเตรียมกำลังคนให้พร้อมกับงานวิจัยและเศรษฐกิจที่ใช้ข้อมูลและการคำนวณมากขึ้น

คนไทยทำงานหนัก แต่สร้างผลผลิตต่ำ ประเทศไทยสร้างผลผลิตได้ประมาณ 17.9 ดอลล่าร์สหรัฐต่อชั่วโมงทำงาน ขณะที่ค่าเฉลี่ยของกลุ่ม OECD อยู่ที่ประมาณ 70 ดอลล่าร์สหรัฐ และสิงคโปร์อยู่ที่ประมาณ 100.4 ดอลล่าร์สหรัฐต่อชั่วโมง ทั้งที่คนไทยทำงานเฉลี่ยประมาณ 2,177 ชั่วโมงต่อปี ปัญหาจึงไม่ใช่การทำงานน้อย แต่เป็นการสร้างมูลค่าจากเวลา ความรู้ และเทคโนโลยีที่ยังทำได้ไม่เต็มที่ การพัฒนาการทำงานแบบตระหนักถึงสมรรถนะจึงสำคัญ

หัวใจการทำงานของ HPC คือ การตระหนักถึงสมรรถนะ การคิด การทำบนฐานทางวิทยาศาสตร์ การเปิดประสบการณ์ให้ลงมือทำจริงบน LANTA ในโครงการนี้ ผู้เรียนจะสัมผัสกระบวนการปกติของงานวิทยาศาสตร์และการพัฒนาซอฟต์แวร์ อาจจะเกิดผลงานจากการลงมือทำ จากงานเล็กหนึ่งงาน อาจต่อยอดไปสู่โจทย์ด้านอากาศ PM2.5 การเกษตร การแพทย์ ชีวสารสนเทศ ปัญญาประดิษฐ์ วิศวกรรม การผลิต หรือปัญหาของพื้นที่ได้

กิจกรรมนี้จัดขึ้นได้ด้วยการริเริ่มโครงการและแรงผลักดันให้เกิดการตั้งต้นจาก ดร.วิวรรณ จรีรัตนชาติ และทีมงาน NSTDA Supercomputer Center: ThaiSC ภายใต้สำนักงานพัฒนาวิทยาศาสตร์และเทคโนโลยีแห่งชาติ ที่ให้ทั้งการสนับสนุนงบประมาณ ทรัพยากร LANTA และการสนับสนุนด้านเทคนิค

ขอขอบคุณ ดร.อุเทน แสวงวิทย์ และทีมงาน สถาบันวิจัยดาราศาสตร์แห่งชาติ (องค์การมหาชน): NARIT สำหรับการสนับสนุนสถานที่ บุคลากร และทรัพยากร

ขอขอบคุณ Amazon Web Services (Thailand) สำหรับเครดิตคลาวด์และการสนับสนุนบริการออนไลน์ รวมถึง Association for Computing Machinery: ACM วิทยากร ผู้เชี่ยวชาญ และชุมชน HPC นานาชาติ สำหรับองค์ความรู้และสื่อที่เชื่อมโยงงานวิจัยแนวหน้าเข้ากับโจทย์ของประเทศไทย

ผู้ช่วยศาสตราจารย์ ดร.วรวรรณ ดีอัซ การ์บาโย

ผู้จัดโครงการ LANTA HPC Experience Day: On the Move

การใช้หนังสือเล่มนี้

เอกสารอ้างอิงสำหรับการเรียนและการออกแบบงานบน LANTA

หนังสือเล่มนี้เป็นแผนที่ของระบบ LANTA และวิธีคิดเมื่อต้องแปลงโจทย์วิทยาศาสตร์ ข้อมูล หรือ AI ให้เป็นงานคำนวณที่รันบน HPC ได้ แนวปฏิบัติที่ดีคืออ่านภาพรวมก่อนรันคำสั่ง แทนค่าตามบัญชีและพื้นที่โครงการของตน เก็บหมายเลขงาน บันทึกการรัน และผลลัพธ์ทุกครั้ง แล้วปรับชื่อไฟล์ ขนาดข้อมูล โปรแกรม และทรัพยากรให้เหมาะกับโจทย์ของตน

สัญลักษณ์ที่พบในตัวอย่าง

สัญลักษณ์	ความหมาย
`<username>`	ชื่อบัญชีผู้ใช้ LANTA
`<project-id>`	พื้นที่โครงการหรือโฟลเดอร์ที่ทีมใช้เก็บงาน
`<project-account>`	บัญชีโครงการที่ Slurm ใช้ผูกกับการใช้ทรัพยากร
`<partition>`	กลุ่มทรัพยากรที่เลือกใช้ เช่น debug, compute หรือ gpu
`<job-id>`	หมายเลขงานที่ระบบจัดคิวสร้างหลังส่งงานด้วย `sbatch`

ลำดับเนื้อหา

ช่วง	ใช้ตอบคำถามอะไร
พื้นฐานระบบ	HPC คืออะไร ทำไมต้องใช้ลินุกซ์ (Linux), Shell, ระบบไฟล์ และระบบจัดคิว
การเข้าใช้ LANTA	จะรู้ได้อย่างไรว่าบัญชี พื้นที่เก็บข้อมูล และซอฟต์แวร์พร้อมใช้งาน
Slurm และงานตัวอย่าง	สคริปต์งานประกอบด้วยอะไร และระบบจัดคิวอ่านคำขอทรัพยากรอย่างไร
งานวิทยาศาสตร์	จะจัดข้อมูลนำเข้า พารามิเตอร์ บันทึกการรัน และผลลัพธ์ของแบบจำลองอย่างไรให้ตรวจซ้ำได้
งาน AI/GPU	GPU มีบทบาทส่วนใด และควรดูสัญญาณใดในบันทึกการฝึกหรือการรันโมเดล

ใช้ช่วงเตรียมตัว

อ่านพื้นฐานลินุกซ์, SSH, การย้ายไฟล์, module และแนวคิดของงาน ทดสอบแรกก่อนวันลงมือจริง

ใช้วันลงมือจริง

เปิดเทียบกับงานจริงเมื่อต้องเลือก partition, เขียนสคริปต์งาน, อ่านบันทึกการรัน และอธิบายผลลัพธ์

แดชบอร์ดกิจกรรม

ใช้ตรวจเวลา ห้องเรียน ลิงก์ และข้อมูลล่าสุดของกิจกรรม

กิจกรรม

กำหนดการอบรม

เวลาและกิจกรรมจากแดชบอร์ด

วัน	เวลา	กิจกรรม
29 ก.ค. 2569	18:30–21:00	Platform and LANTA account readiness
30 ก.ค. 2569	18:30–21:00	ลินุกซ์, SSH และงาน Slurm แรก
1 ส.ค. 2569	18:30–21:00	Running scientific models, and AI Workflows
3 ส.ค. 2569	08:00–08:30	Registration and LANTA account verification
	08:30–09:00	Welcome and event opening
	09:00–09:30	From real problem to LANTA mini innovation
	09:30–10:15	Applied workload tracks and selection
	10:30–11:30	Workload Lab 1: Simulation, data, and parallel jobs
	11:30–12:00	Workload Lab 2: AI/GPU and performance comparison
	13:00–13:30	Mini Innovation Canvas
	13:30–15:00	Mini Innovation Build Sprint
	15:15–16:00	Results and Optimization Clinic
	16:00–16:30	Mini Innovation Demonstrations
	16:30–17:00	Prototype mentoring plan and onsite close
5 ส.ค. 2569	18:30–21:00	Workload performance mentoring
6 ส.ค. 2569	18:30–21:00	Training AI/ML models on LANTA: end-to-end workflow
7 ส.ค. 2569	18:30–21:00	Do High Performance Data Science on HPC
8 ส.ค. 2569	18:30–21:00	Create voice models for low-resource languages on LANTA

พื้นฐานลินุกซ์

ลินุกซ์, Shell และเส้นทางไฟล์

ภาษาพื้นฐานที่ใช้คุยกับซูเปอร์คอมพิวเตอร์จากระยะไกล

เครื่อง ซู เปอร์ คอมพิวเตอร์ ทุก เครื่อง ใน ปัจจุบัน ใช้ ระบบ ปฏิบัติ การ ลิ นุกซ์ ดัง นั้น ผู้ ใช้ จึง ต้อง รู้จัก คำ สั่ง และโครงสร้างของระบบปฏิบัติการลินุกซ์ในการทำงาน เราเข้าใช้ระบบซูเปอร์คอมพิวเตอร์ด้วยการล็อกอินจากทางไกล โดยเชื่อมต่อผ่าน `ssh` โดยใช้โปรแกรมเทอร์มินัล

คำศัพท์พื้นฐาน

คำ	ความหมายในบริบท LANTA
หน้าต่างคำสั่ง	โปรแกรมบนเครื่องของผู้ใช้สำหรับพิมพ์คำสั่งและอ่านข้อความตอบกลับ
`ssh`	คำสั่งที่สร้างการเชื่อมต่อระยะไกลอย่างปลอดภัยไปยัง LANTA
ช่วงเชื่อมต่อระยะไกล	ช่วงเวลาหลัง `ssh` สำเร็จ ซึ่ง prompt ที่เห็นเป็นเชลล์บนเครื่องปลายทาง
เชลล์	โปรแกรมบนลินุกซ์ที่รับคำสั่ง อ่านตัวแปร แปลสัญลักษณ์ และสั่งระบบปฏิบัติการทำงาน
เส้นทางไฟล์	ที่อยู่ของไฟล์หรือโฟลเดอร์ เช่น พื้นที่ส่วนตัว พื้นที่โครงการ หรือพื้นที่เก็บผลลัพธ์
เครื่องทางเข้า	จุดแรกหลังเข้าสู่ LANTA ใช้เตรียมไฟล์ โหลดซอฟต์แวร์ ตรวจข้อมูล แก้สคริปต์ และส่งงาน
เครื่องคำนวณ	เครื่องที่ระบบจัดคิวเลือกให้รันงานจริงตามทรัพยากรที่ร้องขอ
สคริปต์เชลล์	ไฟล์ที่รวบรวมคำสั่งลินุกซ์หลายคำสั่ง เพื่อให้เชลล์รันตามลำดับและรันซ้ำได้

ลำดับการทำงาน

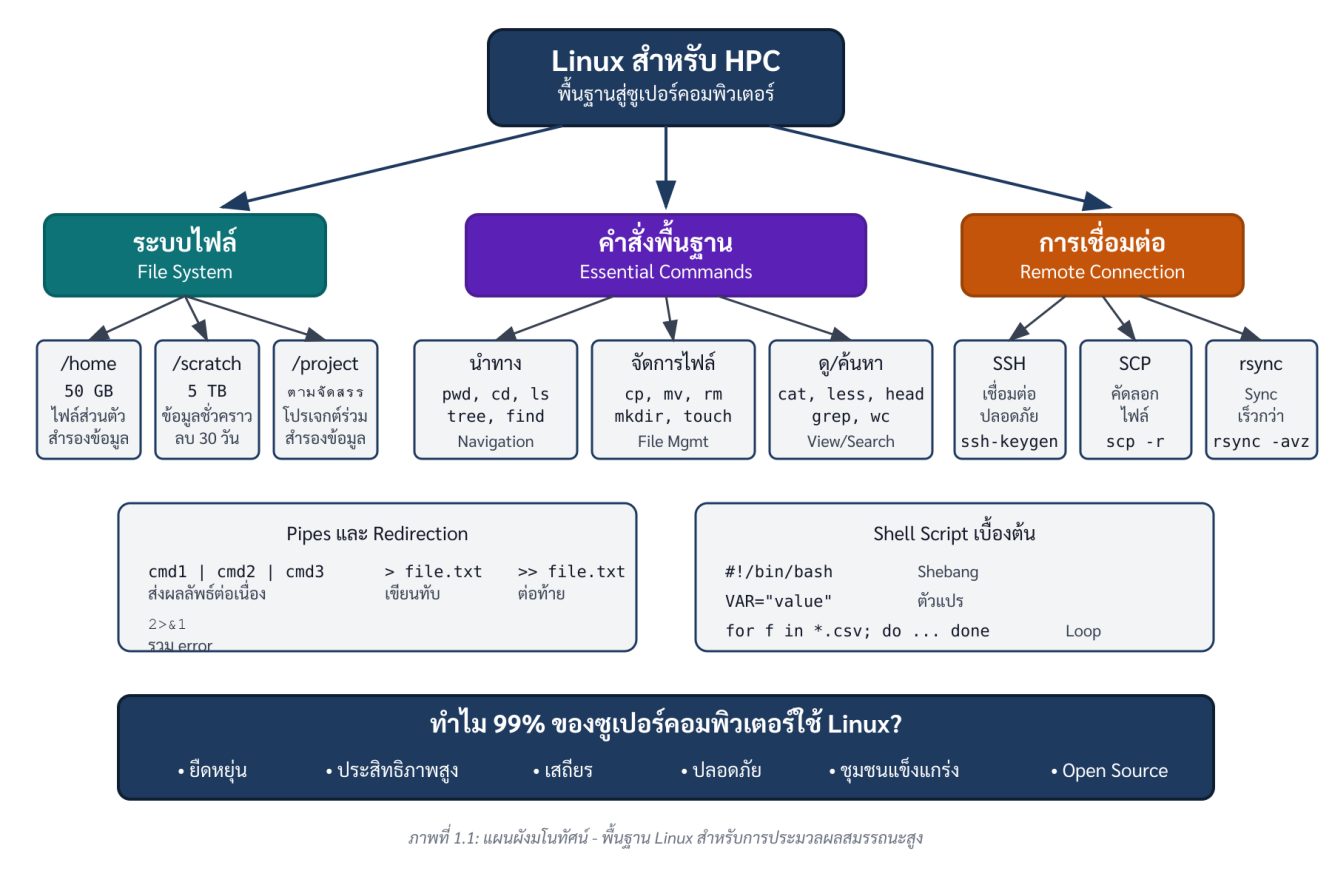
ช่วง	เกิดอะไรขึ้น
ก่อนเชื่อมต่อ	คำสั่งทำงานบนเครื่องของผู้ใช้ เช่น เตรียมไฟล์หรือเรียก `ssh`
หลัง `ssh` สำเร็จ	เชลล์ที่รับคำสั่งอยู่บน LANTA คำสั่ง `pwd`, `ls`, `module` จึงอ่านสภาพแวดล้อมของ LANTA
เมื่อรันสคริปต์	เชลล์อ่านไฟล์สคริปต์ทีละบรรทัด สร้างโฟลเดอร์ ตั้งค่าตัวแปร โหลด module หรือเรียกโปรแกรมตามที่เขียนไว้
เมื่อส่ง Slurm	`sbatch` ส่งสคริปต์เชลล์ให้ระบบจัดคิว แล้วคำสั่งหลักไปรันบนเครื่องคำนวณเมื่อได้ทรัพยากร

แผนผังลินุกซ์

ภาพรวมทักษะลินุกซ์สำหรับ HPC

เริ่มจากระบบไฟล์ คำสั่งพื้นฐาน การเชื่อมต่อ และสคริปต์

บนระบบลินุกซ์ ทุกคำสั่งสัมพันธ์กับตำแหน่งของไฟล์และโฟลเดอร์ การทำงานบน LANTA จึงต้องอ่านเส้นทางไฟล์ให้ถูก นำทางใน directory จัดการไฟล์ ตรวจดูเนื้อหา และค้นหาข้อมูลได้อย่างมั่นใจ เมื่อคำสั่งพื้นฐานเหล่านี้เริ่มเป็นระบบแล้ว การใช้ pipe, redirection, `ssh` และสคริปต์เชลล์จะกลายเป็นเครื่องมือสำหรับจัดลำดับงานให้รันซ้ำได้ และใช้ต่อเป็นฐานของ Slurm job script



ภาพที่ 1.1: แผนผังมโนทัศน์ - พื้นฐาน Linux สำหรับการประมวลผลสมรรถนะสูง

ภาพนี้บอกอะไร

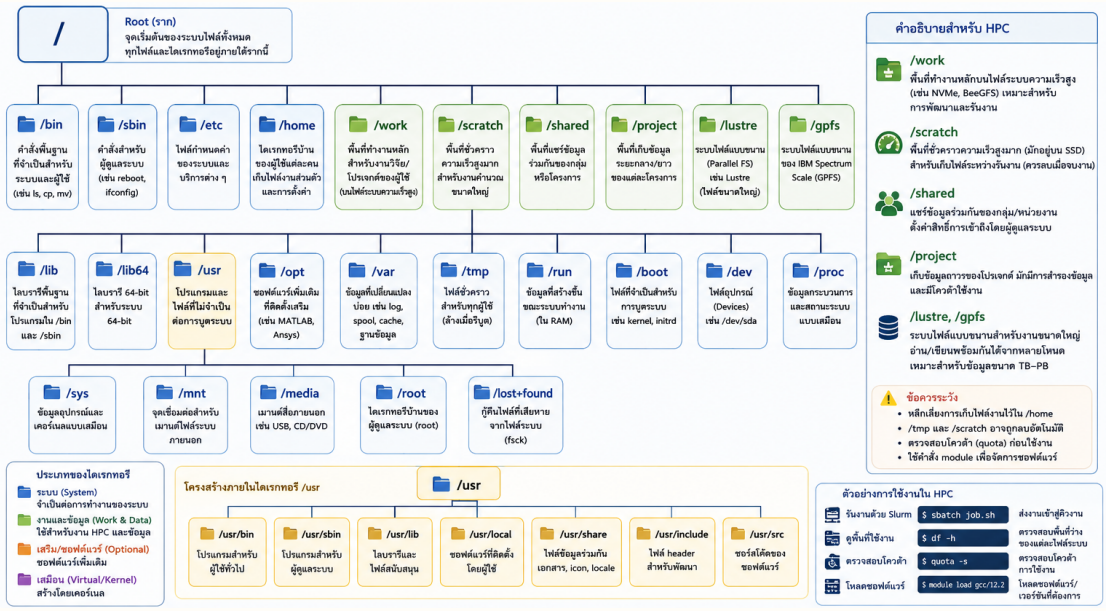
- ระบบไฟล์: แยกพื้นที่ส่วนตัว พื้นที่งานชั่วคราว และพื้นที่โครงการ
- คำสั่งพื้นฐาน: รู้ตำแหน่งปัจจุบัน ย้ายโฟลเดอร์ สร้าง ลบ คัดลอก ดู และค้นหาไฟล์
- การเชื่อมต่อและทำงานซ้ำ: ใช้ `ssh` เปิดช่วงเชื่อมต่อระยะไกล แล้วใช้สคริปต์เชลล์เก็บลำดับคำสั่งให้ตรวจซ้ำได้

คำสั่งพื้นฐาน

พื้นฐานลินุกซ์สำหรับ LANTA

คำสั่งพื้นฐานคือภาษาสำหรับจัดไฟล์ เรียกโปรแกรม และอ่านผลลัพธ์

ลินุกซ์จัดระเบียบไฟล์เป็นโครงสร้างแบบต้นไม้ เริ่มจาก / และมีโฟลเดอร์ย่อยสำหรับผู้ใช้ งานชั่วคราว และพื้นที่โครงการ บน LANTA การรู้ตำแหน่งปัจจุบันและเส้นทางไฟล์ทำให้คำสั่งอ่านข้อมูล เขียนผลลัพธ์ และส่งสคริปต์ทำงานได้ถูกที่



ภาพนี้บอกอะไร

- / คือจุดเริ่มของระบบไฟล์ ทุกเส้นทางบนลินุกซ์เริ่มจากโครงสร้างนี้
- พื้นที่ที่ใช้บ่อยบน LANTA คือ `/home/$USER`, `/scratch/$USER`, `/project/<group>` และพื้นที่ซอฟต์แวร์ที่ระบบเตรียมไว้
- ก่อนรันคำสั่งให้ตรวจตำแหน่งด้วย `pwd` และดูไฟล์ด้วย `ls` เพื่อให้ input, script, log และ output อยู่ในที่ที่ตั้งใจ

อ่าน prompt ก่อนอ่านคำสั่ง

เมื่อ เปิด หน้าต่าง คำ สั่ง บน เครื่อง ของ ตนเอง prompt แรก เป็น เชลล์ ของ เครื่อง นั้น หลัง ใช้ `ssh <username>@lanta.nstda.or.th` และ เข้า สู่ ระบบ สำเร็จ prompt จะ อยู่ ใน ช่วง เชื่อม ต่อ ระยะ ไกล บน LANTA คำสั่งในตัวอย่างต่อไปนี้จึงควรรันหลังเข้า LANTA แล้ว หรือหลังเข้าไปยังโฟลเดอร์งานที่ระบุ

คำสั่งหลัก

```
pwd
ls -lh
find . -maxdepth 2 -type f | sort
du -sh input results logs
head -5 input/data.csv
tail -20 logs/job.out
grep -i "error\\|failed\\|traceback" logs/*.err
```

อ่านรูปแบบคำสั่งที่ใช้ในตัวอย่าง

รู้สัญลักษณ์ก่อนเจอ heredoc, pipe, redirection และตัวแปร

บทลินุกซ์อธิบายว่า shell script คือไฟล์ที่รวบรวมคำสั่งลินุกซ์หลายคำสั่งเข้าด้วยกัน เพื่อให้ทำงานอัตโนมัติ ตัวอย่างในเล่มใช้ Bash สร้างไฟล์ เก็บผลลัพธ์ และส่งงานให้ Slurm ตารางนี้เป็นรายการอ้างอิงก่อนเข้าสู่หน้าที่มีคำสั่งยาว

รูปแบบคำสั่ง	ความหมาย
`cmd > file`	เขียนผลลัพธ์ปกติลงไฟล์ใหม่ ถ้าไฟล์เดิมมีอยู่จะถูกเขียนทับ
`cmd » file`	เขียนผลลัพธ์ต่อท้ายไฟล์เดิม
`cmd 2> err`	เก็บข้อความผิดพลาดลงไฟล์ `err`
`cmd > out 2> err`	แยกผลลัพธ์ปกติและข้อความผิดพลาดเป็นคนละไฟล์
`cmd1 | cmd2`	ส่งผลลัพธ์ของคำสั่งแรกไปเป็นข้อมูลเข้าให้คำสั่งถัดไป
`VAR=value` และ `$VAR`	ตั้งค่าตัวแปรและเรียกใช้ค่าของตัวแปร
`${VAR:-default}`	ใช้ค่าของ `VAR`; ถ้าไม่มีค่าให้ใช้ `default`
`$(cmd)`	รันคำสั่งย่อยแล้วนำผลลัพธ์มาแทนที่ในบรรทัดนั้น
`\{ ... \} > file`	รวมผลจากหลายคำสั่งแล้วเขียนลงไฟล์เดียว
`*`	แทนชื่อไฟล์หลายไฟล์ เช่น `logs/*.err`
`cmd || true`	ให้สคริปต์ทำงานต่อเมื่อคำสั่งเสริมไม่สำเร็จ เช่น module บางชื่อไม่มีในระบบ
`command -v name`	ตรวจว่ามีคำสั่ง `name` ให้เรียกใช้หรือไม่
`sed -n "Np" file`	อ่านบรรทัดที่ `N` จากไฟล์ ใช้กับ job array ที่อ่านพารามิเตอร์ตามหมายเลขงานย่อย

Heredoc คืออะไร

Heredoc คือวิธีเขียนข้อความหลายบรรทัดลงไฟล์หรือส่งเข้าโปรแกรมโดยตรง ในบทลินุกซ์ใช้รูปแบบนี้สร้างไฟล์ README หรือไฟล์ตั้งค่าโดยให้ shell อ่านทุกบรรทัดจนถึงคำปิดท้าย เช่น `EOF` รูปแบบ `cat > file «’EOF’` จะสร้างไฟล์ใหม่ และเครื่องหมายคำพูดรอบ `EOF` ทำให้ตัวแปรข้างในถูกเก็บเป็นข้อความตามที่พิมพ์

```
cat > configs/run-small.env <<’EOF’
INPUT=input/sample.csv
OUTPUT=results/sample-summary.csv
EOF
```

โครงสร้างไฟล์

จัดโครงสร้างไฟล์ให้รันซ้ำได้

ข้อมูลนำเข้า ผลลัพธ์ และไฟล์ชั่วคราวต้องแยกโฟลเดอร์ชัดเจน

งาน HPC อ่านง่ายขึ้นเมื่อเส้นทางไฟล์ชัดเจน ผู้ใช้ควรรู้ว่าไฟล์ใดเป็นข้อมูลต้นฉบับ ไฟล์ใดเป็นสคริปต์ ไฟล์ใดเป็นบันทึกการรัน และผลลัพธ์ของแต่ละ run ถูกเก็บไว้ที่ใด

สร้างรายการไฟล์และขนาดข้อมูล

```
cd /project/<project-id>/lanta-experience

find input -type f | sort > notes/input-files.txt
du -sh input > notes/input-size.txt
sha256sum input/* 2>/dev/null > notes/input-checksums.txt
```

คำสั่งหลัก

คำสั่ง	ใช้ทำอะไร
`find`	ค้นหาไฟล์ตามเงื่อนไข เช่น ค้นทุกไฟล์ใต้ `input/`
`sort`	เรียงรายการให้อ่านและเปรียบเทียบซ้ำได้
`du -sh`	สรุปขนาดพื้นที่ที่โฟลเดอร์ใช้
`sha256sum`	สร้างค่าตรวจสอบไฟล์ เพื่อรู้ว่าไฟล์เปลี่ยนหรือไม่
`cat «'EOF'`	สร้างไฟล์ข้อความหลายบรรทัดตามรูปแบบ heredoc

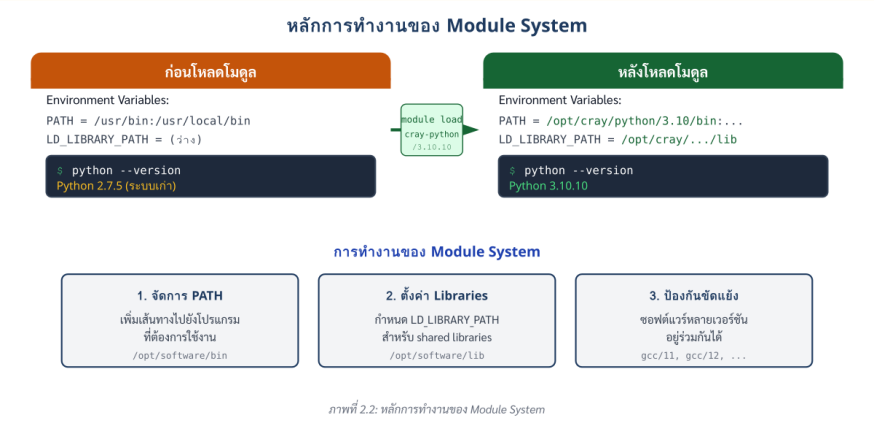
ตัวอย่าง config ที่อ่านง่าย

```
cat > configs/run-small.env <<'EOF'
INPUT=input/sample.csv
OUTPUT=results/sample-summary.csv
WORKERS=4
MODE=small
EOF
```

ตัวแปรสภาพแวดล้อมและ module

เชลล์ส่งค่าให้โปรแกรมผ่านตัวแปร และ module ใช้เปลี่ยนค่านั้นให้ตรงกับซอฟต์แวร์

เชลล์มีชุดค่าที่เรียกว่าตัวแปรสภาพแวดล้อม ค่าเหล่านี้ถูกส่งให้โปรแกรมตอนเริ่มรัน เช่น `PATH` บอกลำดับโฟลเดอร์ที่เชลล์ใช้หาโปรแกรม `LD_LIBRARY_PATH` บอกตำแหน่ง library และ `HOME` บอกพื้นที่ส่วนตัวของผู้ใช้ บน LANTA มีซอฟต์แวร์หลายเวอร์ชันอยู่ร่วมกัน ระบบ module จึงใช้เปลี่ยนค่าเหล่านี้อย่างเป็นระบบ เมื่อ `module load` เส้นทางของซอฟต์แวร์ที่เลือกจะถูกเพิ่มเข้าไปในสภาพแวดล้อม งานที่ต้องรันซ้ำควรเขียน `module purge`, `module load` และ `module list` ไว้ในสคริปต์หรือ log



ภาพที่ 2.2: หลักการทำงานของ Module System

คำสั่งและค่าที่ต้องตรวจ

สิ่งที่ใช้	ตรวจอะไร
`echo $PATH`	เชลล์จะเจอโปรแกรมจากโฟลเดอร์ใดก่อน
`env`	แสดงตัวแปรสภาพแวดล้อมทั้งหมดที่โปรแกรมจะได้รับเมื่อเริ่มรัน
`module avail/spider`	ซอฟต์แวร์และเวอร์ชันใดมีให้โหลดบนระบบ
`module purge/load`	เริ่มจากสภาพแวดล้อมสะอาด แล้วเลือกซอฟต์แวร์ที่งานต้องใช้
`module list`	บันทึก module ที่โหลดอยู่ เพื่อใช้ตรวจซ้ำเมื่ออ่าน log
`which python`	คำสั่ง `python` ที่กำลังเรียกใช้มาจากตำแหน่งใด

HPC Ignite: ปลดพลังซูเปอร์คอมพิวเตอร์

ภาพรวมพัฒนาการ สถาปัตยกรรม ผลกระทบ และการพัฒนาคนด้าน HPC

1 วิวัฒนาการและพลังของ HPC

1 เครื่องเดี่ยวสมรรถนะสูง (1970–80)

ระบบเดี่ยว ขนาดยังจำกัด

2 ประมวลผลขนานแบบกระจาย (1990–2000)

เชื่อมหลายหน่วยประมวลผลเข้าด้วยกัน

3 คลัสเตอร์ขนาดใหญ่และคลาวด์ HPC (2000–ปัจจุบัน)

ขยายระบบและเข้าถึงทรัพยากรตามต้องการ

ซูเปอร์คอมพิวเตอร์คืออะไร?

ระบบคอมพิวเตอร์สมรรถนะสูง วัดด้วย FLOPS และมักอยู่ในกลุ่ม Top500

ก้าวสู่ Exascale

ระดับ 10^{18} operations per second สำหรับงานวิทยาศาสตร์และ AI ขนาดใหญ่

2 ภายในสถาปัตยกรรม

ห้องและตู้แร็ก

แร็กและตู้เชื่อมต่อกันในพื้นที่เฉพาะ

แชสซี/เบลด

โหนดประมวลผล

แกน CPU/GPU

CPU GPU

หลายแกนต่อโหนด

Login node → Compute node

เข้าใช้งานผ่าน terminal/shell แล้วจองโหนดสำหรับรันงาน

โครงสร้างสนับสนุน

ไฟฟ้า ความเย็น เครือข่าย และความปลอดภัย

3 ผลกระทบทางเศรษฐกิจและการเติบโต

ลงทุน HPC

ช่วยเร่งนวัตกรรมและขีดความสามารถในการแข่งขัน

แหล่งการลงทุน

- ภาครัฐ
- กลาโหม
- การเงิน
- เอกชนอื่น

AI เป็นแรงขับสำคัญ

- AI/ML
- Smart City/ Digital Twin
- สุขภาพ/ชีววิทยาศาสตร์
- ฟิสิกส์/พลังงาน/วัสดุ

การเติบโต vs ความพร้อม

การลงทุน · ความพร้อมของระบบ · การเติบโตของตลาด · การใช้งานจริง

4 การพัฒนาคนด้าน HPC

เส้นทางพัฒนาทักษะ

- HPC Expert / Talent Zone
- Advanced User
- Intermediate User
- Novice User
- Pre-HPC User

ผู้เชี่ยวชาญแก้โจทย์ซับซ้อนได้เร็วขึ้น

ทักษะด้านขนานและการใช้งานระบบช่วยเพิ่มประสิทธิภาพ

เสาหลักทักษะ (3 เสาหลัก)

- ทักษะพื้นฐาน: Linux, Shell
- ทักษะ HPC: Job Scheduling, Scalability
- การคิดวิเคราะห์: Problem Solving

ผู้เชี่ยวชาญสาขา vs ผู้เชี่ยวชาญ IT

VS

งาน HPC ต้องอาศัยการเชื่อมโลกวิทยาศาสตร์กับวิศวกรรมคอมพิวเตอร์

สิ่งที่เห็นจากภาพนี้

1. HPC คือโครงสร้างพื้นฐานของงานวิจัยและ AI
2. สถาปัตยกรรมที่ดีช่วยให้รันงานได้มีประสิทธิภาพ
3. การลงทุนและตลาดผลักดันการเติบโต
4. การพัฒนาคนคือหัวใจของระบบนิเวศ

การคำนวณสมรรถนะสูงและซูเปอร์คอมพิวเตอร์

จากเครื่องคำนวณ สมรรถนะ การวัดผล และการออกแบบระบบ สู่การใช้งานจริง

ซูเปอร์คอมพิวเตอร์ (Supercomputer) คือคอมพิวเตอร์ประสิทธิภาพสูงในระดับเดียวกับเครื่องที่เร็วที่สุดของโลก โดยวัดจากความเร็วการคิดเลขทศนิยมเมื่อรัน Linpack และส่งผลไปจัดลำดับใน TOP500 คำว่า “ซูเปอร์คอมพิวเตอร์” มักชี้ไปที่ตัวเครื่องและเทคโนโลยีการสร้างเครื่อง ส่วนการคำนวณสมรรถนะสูง หรือ HPC (High-Performance Computing) โอบรวมระบบกลไกทั้งหมดที่ทำให้ซูเปอร์คอมพิวเตอร์รันโปรแกรมได้เร็ว มีประสิทธิภาพ ใช้ทรัพยากรคุ้มค่า และประหยัดพลังงาน

FLOPS, Linpack และงานจริง

ความเร็วของระบบ HPC วัดในหน่วย FLOPS หรือจำนวนการคำนวณเลขทศนิยมต่อวินาที การวัดด้วย Linpack เป็นภาษากลางสำหรับเปรียบเทียบสมรรถนะ แต่ประสิทธิภาพของงานจริงยังขึ้นกับรูปแบบข้อมูล หน่วยความจำ เครือข่าย การอ่านเขียนไฟล์ ซอฟต์แวร์ การแบ่งงาน และทักษะของผู้ใช้ หนังสือเล่มนี้จึงสอนทั้งการเข้าใจสมรรถนะและการใช้งานระบบจริง

ส่วนประกอบหลักของระบบ

ส่วนประกอบ	บทบาท
เครื่องทางเข้า	พื้นที่เข้าใช้ระบบ เตรียมไฟล์ แก้สคริปต์ และส่งงานเข้าสู่ระบบจัดคิว
เครื่องคำนวณ	เครื่องที่ใช้รันงานจริงหลังจากระบบจัดคิวจัดสรรทรัพยากร
ระบบไฟล์	พื้นที่เก็บข้อมูลนำเข้า โค้ดต้นฉบับ สคริปต์งาน log และผลลัพธ์
ระบบ module	วิธีเลือก compiler, Python, MPI, CUDA และ library เวอร์ชันต่าง ๆ
ระบบจัดคิว	ระบบรับคำขอทรัพยากรและสั่งให้รันจริงบนเครื่องคำนวณ
หลักฐานสมรรถนะ	เวลารัน หน่วยความจำ การขยายขนาดงาน log, metric และผลลัพธ์ที่ใช้ตรวจความถูกต้อง

จากเครื่องเร็วสู่ระบบนิเวศ

พัฒนาการของ HPC เติบโตจากเครื่องเดี่ยวประสิทธิภาพสูง ไปสู่ระบบหน่วยความจำแยก การประมวลผลขนานจำนวนมาก และ cluster ขนาดใหญ่ที่เชื่อม Cloud-HPC, Quantum-HPC hybrid และ GPU cluster สำหรับ AI ผู้ใช้จึงเข้ามาใช้ทั้งเครื่อง เครือข่าย ซอฟต์แวร์ ข้อมูล และกติกาการจัดสรรทรัพยากรในระบบนิเวศเดียวกัน

ตรวจสภาพแวดล้อม LANTA ก่อนส่งงาน

ดูคิว พื้นที่ บัญชีโครงการ และซอฟต์แวร์ที่จะใช้ในสคริปต์ Slurm

ก่อนส่งงานให้รู้ข้อมูลพื้นฐานหกอย่าง ได้แก่ โฟลเดอร์งาน partition เวลาที่ขอ พื้นที่เขียน log และผลลัพธ์ บัญชีโครงการ และซอฟต์แวร์ที่ต้องใช้ ข้อมูลเหล่านี้ทำให้สคริปต์ Slurm ระบุทรัพยากรและสภาพแวดล้อมของงานได้ครบ

partition เวลา CPU, GPU และบัญชีโครงการจะอยู่ในบรรทัด #SBATCH ซอฟต์แวร์จะตั้งด้วย `module load` โฟลเดอร์ที่ใช้ส่งงานจะกลับมาได้จาก `$SLURM_SUBMIT_DIR` และหมายเลขงานจะอยู่ใน `$SLURM_JOB_ID` เมื่อเขียนครบ งานทดสอบจะบอกได้ว่าบัญชี พื้นที่ ซอฟต์แวร์ และ Slurm พร้อมใช้งานหรือยัง

คำสั่งดูสภาพแวดล้อม

```
sinfo -o "%P %a %l %D %t %N"
squeue -u $USER
myquota
sbalance
echo $PATH
module list
module avail python
module avail mpi
module avail cuda
```

ผลลัพธ์จะบอกอะไรเรา

คำสั่ง	ใช้เพื่อดูอะไร
`sinfo`	ชื่อ partition สถานะ node และเวลาสูงสุดที่ควรใส่ใน #SBATCH
`squeue -u USER`	งานของเรากำลังรอคิวหรือรันอยู่ ช่วยตัดสินใจว่าจะส่งงานทดสอบเพิ่มหรือรอผลเดิม
`myquota`	พื้นที่และจำนวนไฟล์ที่ยังใช้ได้ ก่อนสร้าง log และผลลัพธ์จำนวนมาก
`sbalance`	สิทธิ์หรือเครดิตของบัญชีโครงการที่ใช้กับงานของทีม
`module avail`	ชื่อซอฟต์แวร์และเวอร์ชันที่จะใส่ในบรรทัด `module load`

เริ่มจากงานเล็ก

ใช้ข้อมูลนำเข้าขนาดเล็กเพื่อตรวจเส้นทางไฟล์ module และสคริปต์ Slurm ให้ผ่านก่อน จากนั้นจึงเพิ่มขนาดข้อมูล จำนวน CPU, MPI rank หรือ GPU ตามหลักฐานจาก log และผลลัพธ์

การเข้าเครื่องและการจัดไฟล์

หน้าต่างคำสั่งคือจุดควบคุมงานบนระบบคำนวณสมรรถนะสูง

การจัดโฟลเดอร์ให้ชัดเจนช่วยให้รู้ว่าไฟล์ข้อมูลต้นทางอยู่ที่ใด ย้ายขึ้น LANTA ด้วยวิธีใด ไฟล์คำสั่งใดใช้รันงาน และไฟล์ผลลัพธ์ถูกเก็บไว้ที่โฟลเดอร์ใด `ssh` ใช้เข้าเครื่อง, `scp` และ `rsync` ใช้ย้ายไฟล์ผ่านเครื่องสำหรับรับส่งข้อมูล ส่วน `find`, `du` และ `wc` ใช้ตรวจว่าไฟล์ขึ้นครบและขนาดเป็นไปตามที่คาด

เข้าเครื่องและย้ายไฟล์

```
ssh <username>@lanta.nstda.or.th

scp local_input.csv <username>@lanta-xfer.nstda.or.th:/project/<project-id>/lanta-experience/input/
rsync -avz --progress local_folder/ <username>@lanta-xfer.nstda.or.th:/project/<project-id>/lanta-experience/input/local_folder/
```

ตรวจไฟล์หลังย้าย

```
cd /project/<project-id>/lanta-experience
find input -maxdepth 2 -type f | sort | head
du -sh input
wc -l input/*.csv 2>/dev/null
```

ไฟล์คำสั่ง

เก็บไฟล์คำสั่งที่แก้ด้วยมือไว้ใน `src/` และบันทึกเวอร์ชันหรือสำรองเมื่อเปลี่ยนส่วนสำคัญ

ไฟล์ผลลัพธ์

เก็บบันทึกการรันใน `logs/` และผลลัพธ์ใน `results/` ให้แยกจากข้อมูลต้นทาง

จากงานคำนวณสู่ผลลัพธ์ที่ใช้ต่อได้

การมีทรัพยากรสมรรถนะสูง เปิดโอกาสในการแก้โจทย์ปัญหา ด้วยข้อมูลและโมเดลวิทยาศาสตร์คุณภาพสูง

งานบน LANTA ควรเริ่มจากคำถามที่ชัดเจน เช่น ต้องการวิเคราะห์ข้อมูลชุดใด จำลองระบบใด ฝึกโมเดลใด หรือมีการค้นพบใดที่ต้องการตรวจสอบ จากนั้นระบุ input โปรแกรมหรือสคริปต์ สภาพแวดล้อมซอฟต์แวร์ ทรัพยากรที่ขอ log และผลลัพธ์ เมื่อข้อมูลเหล่านี้อยู่ด้วยกัน คนอื่นจะตรวจได้ว่าผลลัพธ์เกิดจากเงื่อนไขใด และควรปรับการทดลองรอบถัดไปจุดใด

แปลงโจทย์เป็นงานคำนวณ

โจทย์ปลายทาง	งานบน LANTA	ต้องการหลักฐานอะไรบ้าง
คุณภาพ อากาศ และเกษตร	แบบ จำลอง ภูมิ อากาศ ข้อมูลเซนเซอร์ ภาพถ่ายดาวเทียม	เวลารัน พื้นที่ศึกษา แหล่งข้อมูล ผลสรุป และกราฟตรวจผล
ชีวภาพและสุขภาพ	วิเคราะห์ ลำดับ พันธุกรรม โมเลกุล ภาพทางการแพทย์ หรือข้อมูลกลุ่มตัวอย่าง	รุ่นฐานข้อมูล เวอร์ชันโปรแกรม พารามิเตอร์ และตัวชี้วัดความถูกต้อง
AI และข้อมูลขนาดใหญ่	เตรียมข้อมูล ฝึกโมเดล ประเมินผล หรือรัน inference	ชุดข้อมูล รุ่นโมเดล seed ค่า metric และ log ของ GPU/CPU
วิศวกรรมและวัสดุ	การ จำลอง การ สำรวจ หลายแบบ และการลองหลายพารามิเตอร์	ไฟล์ตั้งค่า mesh/grid ขนาดปัญหา และผลเปรียบเทียบกับ baseline

รูปแบบคิดสำหรับงานของเรา

โจทย์ปลายทาง → ข้อมูลและแบบจำลอง → งานบน LANTA → ผลลัพธ์ที่ตรวจได้ → การตัดสินใจหรือการทดลองถัดไป

เลือกทรัพยากรให้ตรงกับงาน

CPU, MPI, GPU และการอ่านเขียนไฟล์เหมาะกับงานต่างกัน

การขอทรัพยากรควรเริ่มจากลักษณะงานและสัญญาณว่าจุดใดจะเกิดคอขวด ก่อนเลือก CPU, GPU, หน่วยความจำ หรือเวลาใช้งาน งานที่อ่านเขียนไฟล์หนักมักได้ผลจากการจัดไฟล์และแบ่งข้อมูลให้ดี งาน GPU ได้ผลเมื่อโค้ดใช้ GPU จริงและควบคุมการย้ายข้อมูลได้

ลักษณะงาน	ตัวเลือกแรก	สัญญาณที่ต้องดู
งาน Python บน CPU	1 node, 1 task, หลาย CPU ต่อ task	การใช้ CPU เวลารัน และหน่วยความจำ
งานเดียว หลายชุดข้อมูล	job array	log แยกตาม task และงานย่อยที่ล้มเหลว
งาน MPI	หลาย tasks และใช้ `srun`	scaling, จำนวน rank และเวลาสื่อสาร
งาน OpenMP	1 task, หลาย CPU ต่อ task	`OMP_NUM_THREADS` ตรงกับ CPU ที่ขอ
งาน AI/GPU	gpu partition และคำขอ GPU	`nvidia-smi`, หน่วยความจำ GPU และ training log
งานอ่านเขียนหนัก	จัดไฟล์และ chunk ให้ดี	จำนวนไฟล์ อัตราอ่านเขียน และไฟล์ชั่วคราว

อ่านผลการรัน

- งานขนาดเล็กให้ผลตอบกลับเร็วกว่าในช่วงสำรวจคอขวดของโปรแกรม
- การเพิ่มขนาดงานแยกทีละแกน เช่น ข้อมูลนำเข้า CPU, MPI rank หรือ GPU ทำให้อ่านผลได้ชัดกว่า
- การเปลี่ยนทีละอย่างทำให้บอกได้ว่าอะไรทำให้เวลา หน่วยความจำ หรือผลลัพธ์เปลี่ยน
- คำขอทรัพยากร เวลารัน หน่วยความจำ และผลลัพธ์ เป็นข้อมูลพื้นฐานสำหรับเปรียบเทียบ run

บันทึกหลักฐานของงาน

งานที่รันได้มีเส้นทางให้ตรวจย้อนกลับ

งาน HPC หนึ่งงานประกอบด้วยไฟล์ผลลัพธ์และบริบทกำกับ ได้แก่ สภาพแวดล้อม คำสั่ง ทรัพยากร ข้อความผิดพลาด บันทึกการรัน และเวลารัน ข้อมูลเหล่านี้ทำให้ผู้อ่านคนอื่นเข้าใจว่า run นั้นเกิดขึ้นภายใต้เงื่อนไขใด และใช้เปรียบเทียบกับ run ถัดไปได้

ตัวอย่างข้อมูลกำกับ run

```
mkdir -p notes
{
  echo "workspace=$(pwd)"
  echo "date=$(date -Is)"
  echo "user=$(whoami)"
  echo "host=$(hostname)"
} > notes/evidence.txt

module list 2> notes/modules-current.txt
squeue -u $USER > notes/squeue-current.txt
```

หลังส่งงาน ให้เพิ่มข้อมูลนี้

```
JOBID=<job-id>
sacct -j $JOBID --format=JobID,JobName,State,Elapsed,AllocCPUS,MaxRSS,ExitCode \
  > notes/sacct_${JOBID}.txt
cp jobs/<job-script>.sbatch notes/
cp logs/*${JOBID}* notes/ 2>/dev/null || true
```

ข้อมูลที่ทำให้ run อ่านได้

เส้นทางของพื้นที่งาน หมายเลขงาน สคริปต์งาน บันทึกการรัน ข้อความผิดพลาด ตัวอย่างผลลัพธ์ และคำถามเฉพาะเจาะจง เช่น ต้องการลดเวลา ต้องการแก้ข้อผิดพลาดจากหน่วยความจำ หรือต้องการย้ายไป GPU

แผนที่เข้าสู่โลก HPC:
จากพื้นฐาน Linux สู่งานซูเปอร์คอมพิวเตอร์

เรียนรู้ทีละขั้น ใช้งานได้จริง

โซน 1: ตั้งตัวในสภาพแวดล้อมผู้ใช้

1 รับตัวตนและเช็กสิทธิ์

```
$ whoami
> whoami
hpc_user
>id
uid=1001
gid=1001
```

- บัญชีผู้ใช้
- กลุ่มและสิทธิ์
- คำสั่ง: whoami, id

2 เขียนสคริปต์ให้ทำงานเอง

```
#!/bin/bash
echo "Hello"
INPUT=$1
echo $0 $1
```

.sh

- ใช้ .sh
- รับค่า $0, $1
- ลดงานซ้ำ

3 เลือกพื้นที่เก็บข้อมูล

- /home — งานส่วนตัว
- /project — งานร่วมกัน
- /scratch — งานชั่วคราวเร็ว

โซน 2: ใช้งานคลัสเตอร์ให้ถูกทาง

1 เข้าโหนดล็อกอิน

- แก้ไฟล์
- คอมไพล์
- ไม่รันงานหนัก

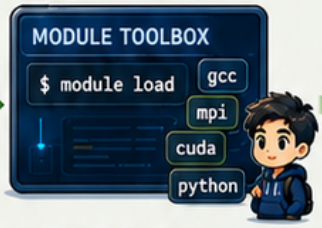
2 จองทรัพยากรผ่าน Slurm

```
$ sbatch job.sh
☑ CPUs ✓
☑ GPUs ✓
☑ Memory ✓
```

slurm

- sbatch
- CPUs / GPUs / Memory
- ตรวจคิวงาน

3 โหลดสภาพแวดล้อม



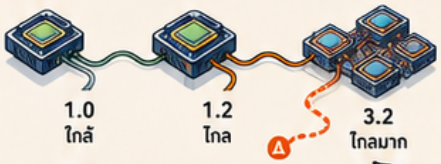
- module load
- compiler / mpi / cuda / python
- ใช้เวอร์ชันให้ตรง

4 ส่งงานไปโหนดประมวลผล

- รันบน compute nodes
- เหมาะกับงานหนัก
- ใช้ทรัพยากรคุ้มค่า

โซน 3: เข้าใจสถาปัตยกรรมและสมรรถนะ

1 ดูระยะห่างของ NUMA

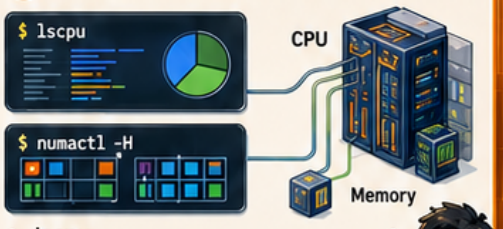


- ใกล้ / ไกล
- กระทบ latency
- ช่วยจัดวางงาน

2 อ่านผลกระทบต่อประสิทธิภาพ

ค่าระยะทาง	ผลกระทบ
1.0	พื้นฐาน
1.2	ช้าลงเล็กน้อย
3.2	ช้าลงมาก

3 ปรับจูนตามสถาปัตยกรรม



- lscpu
- numactl -H
- จับคู่ CPU / Memory

เมื่อทำได้ จะเกิดอะไรขึ้น

1. ใช้งานระบบได้ถูกทาง
2. งานรันได้และตรวจสอบได้
3. ใช้ทรัพยากรคุ้มและต่อยอดงานได้

เมื่อใช้ระบบเป็น งานจะน่าเชื่อถือขึ้น และต่อยอดไปสู่งานที่ช่วยชีวิต ความปลอดภัย พลังงาน และบริการสาธารณะได้

งานแรกต้องเล็กและตรวจได้

เริ่มจากตรวจไฟล์และคำสั่ง แล้วส่งงานทดสอบขนาดเล็กผ่าน Slurm

การเริ่มใช้ LANTA ควรเดินเป็นสามระดับตามลำดับของบทสภาพแวดล้อมระบบ: ตรวจไฟล์และคำสั่งบนเครื่องทางเข้า ส่งงานทดสอบขนาดเล็กผ่าน Slurm และขยายเป็นงานจริงเมื่อรู้ทรัพยากรที่ต้องใช้แล้ว ลำดับนี้ทำให้เห็นความสัมพันธ์ระหว่างเครื่องทางเข้า เครื่องคำนวณ ระบบจัดคิว module และพื้นที่เก็บข้อมูล

ขั้นตอนของงานทดสอบแรก

1. สร้างสคริปต์ Python ที่เขียนไฟล์ผลลัพธ์หนึ่งไฟล์
2. สร้างสคริปต์ Slurm ที่ขอทรัพยากรน้อยและใช้เวลาสั้น
3. ส่งด้วย `sbatch`
4. อ่าน `squeue`, `sacct`, log และไฟล์ผลลัพธ์
5. บันทึกหมายเลขงานลง notes

สัญญาณที่ถือว่าผ่าน

หลักฐาน	ควรเห็นอะไร
หมายเลขงาน	ได้ตัวเลขจาก `sbatch`
Slurm state	`COMPLETED` หรือ state อื่นพร้อม error ที่อ่านได้
output log	มีชื่อ node, เวลาเริ่ม, เวลาเสร็จ
result file	มีไฟล์ที่สคริปต์เขียนสำเร็จ

ประเภทงาน Slurm ที่ส่งบน LANTA

เลือกชนิดงานให้เหมาะกับเป้าหมาย ทรัพยากร และรูปแบบการรัน

1 งานโต้ตอบ (Interactive Job)

- ทดลองคำสั่ง
- debug
- เช็ก environment
- เหมาะกับงานสั้น

```
srun --pty bash
```

ใช้ทดสอบแบบโต้ตอบ ไม่เหมาะกับงานยาว

2 งานแบตช์ CPU

- ส่งสคริปต์ด้วย sbatch
- ใช้ CPU
- งานทั่วไป
- วิเคราะห์ข้อมูล preprocessing

```
sbatch cpu_job.sh
```

1 node | หลาย core

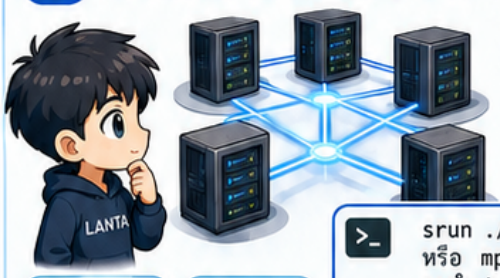
3 งานแบตช์ GPU

- ใช้เร่งงาน AI/ML
- deep learning
- simulation ที่ใช้ GPU

```
sbatch gpu_job.sh
--gres=gpu:1
```

GPU node

4 งานหลายโหนด / MPI



- รันข้ามหลายโหนด
- process หลายตัวสื่อสารกัน
- เหมาะกับ distributed memory

```
srun ./a.out
หรือ mpirun -np ...
ภายใน batch job
```

หลาย node | MPI ranks

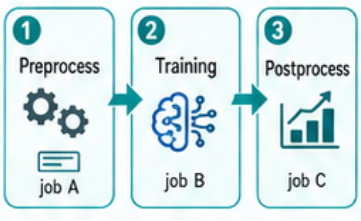
5 งานแบบ Job Array

- งานคล้ายกันหลายชุด
- เปลี่ยนพารามิเตอร์
- sweep
- หลาย input file

```
sbatch --array=1-100 array_job.sh
```

ช่วยจัดการงานซ้ำจำนวนมาก

6 งานต่อเนื่อง / Dependency Workflow



- รันเป็นลำดับ
- งาน B รอ A
- pipeline
- preprocessing -> training -> postprocess

```
sbatch --dependency=afterok:12345 next_job.sh
```

เหมาะกับ workflow หลายขั้น

เลือกแบบไหนดี

1 Interactive	2 CPU batch	3 GPU batch	4 MPI	5 Job Array	6 Dependency
ทดลอง / debug	งานทั่วไป	งานใช้ตัวเร่ง	งานหลายโหนด	งานซ้ำหลายชุด	งานเป็นลำดับ

สิ่งที่ควรเช็กก่อนส่งงาน

- **จำนวน CPU / GPU** ขอให้พอดีกับงาน
- **เวลา walltime** ตั้งเวลาให้เหมาะสม
- **หน่วยความจำ** พอสำหรับงาน
- **ไฟล์เข้า-ออก** พาธถูกต้องและมีสิทธิ์
- **ตรวจผลด้วย sacct / logs** เช็กสถานะและผลลัพธ์

ใช้ทรัพยากรอย่างคุ้มค่า • เคารพกติกาการใช้งาน • ช่วยกันให้ระบบทำงานได้เต็มประสิทธิภาพ

LANTA SUPERCOMPUTER

โครงสร้างสคริปต์งาน Slurm

บอกทรัพยากรที่ต้องการและคำสั่งที่จะรันบนเครื่องคำนวณ

สคริปต์งาน Slurm เป็นไฟล์ Bash ที่มีสองส่วน ส่วนแรกคือบรรทัด #SBATCH สำหรับบอกระบบจัดคิวว่าต้องการทรัพยากรอะไร ส่วนที่สองคือคำสั่งที่จะรันหลังจาก Slurm จัดสรรเครื่องคำนวณให้แล้ว

โครงสร้างมาตรฐาน

```
#!/bin/bash
#SBATCH --job-name=<name>
#SBATCH --partition=<partition>
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=4
#SBATCH --time=00:30:00
#SBATCH --output=logs/%j.out
#SBATCH --error=logs/%j.err
##SBATCH --account=<project-account>

module purge
module load <module-name>
cd "$SLURM_SUBMIT_DIR"

echo "job=$SLURM_JOB_ID node=$SLURM_JOB_NODELIST"
srun <program> <arguments>
```

อ่านบรรทัดสำคัญ

บรรทัด	ความหมาย
`-partition`	เลือกกลุ่มเครื่องที่จะส่งงานเข้าไป
`-nodes`, `-ntasks`	ระบุจำนวนเครื่องและจำนวน process ที่ต้องการ
`-cpus-per-task`	ระบุจำนวน CPU ต่อหนึ่ง process
`-time`	ระบุเวลาสูงสุดที่งานใช้ได้
`-output`, `-error`	ระบุไฟล์สำหรับผลลัพธ์ปกติและข้อความผิดพลาด
`module purge/load`	ตั้งค่าสภาพแวดล้อมซอฟต์แวร์ในสคริปต์ ไม่อาศัย shell เดิมของผู้ใช้
`cd "$SLURM_SUBMIT_DIR"`	กลับไปยังโฟลเดอร์ที่ใช้ส่งงาน

ส่งงานแรก

ตัวอย่าง Slurm job

คัดลอก สร้างไฟล์ ส่งงาน แล้วตรวจผล

ตัวอย่างนี้สร้างไฟล์ Python หนึ่งไฟล์และสคริปต์ Slurm หนึ่งไฟล์ด้วย heredoc จากนั้นส่งงานด้วย `sbatch` ผลลัพธ์ที่ต้องเห็นคือหมายเลขงาน log และไฟล์ใน `results/`

```
cd /project/<project-id>/lanta-experience

cat > src/hello_lanta.py <<'PY'
from pathlib import Path
import os, platform, time

Path("results").mkdir(exist_ok=True)
job_id = os.environ.get("SLURM_JOB_ID", "manual")
out = Path("results") / f"hello_{job_id}.txt"

lines = [
    f"job_id={job_id}",
    f"host={platform.node()}",
    f"user={os.environ.get('USER', 'unknown')}",
    f"submit_dir={os.environ.get('SLURM_SUBMIT_DIR', os.getcwd())}",
    f"time={time.strftime('%Y-%m-%d %H:%M:%S')}",
]
out.write_text("\n".join(lines) + "\n")
print(out)
PY

cat > jobs/hello_lanta.sbatch <<'SLURM'
#!/bin/bash
#SBATCH --job-name=hello_lanta
#SBATCH --partition=debug
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
#SBATCH --time=00:05:00
#SBATCH --output=logs/hello_%j.out
#SBATCH --error=logs/hello_%j.err
##SBATCH --account=<project-account>

module purge
module load cray-python/3.10.10 2>/dev/null || module load python 2>/dev/null || true
cd "$SLURM_SUBMIT_DIR"
python src/hello_lanta.py
SLURM

sbatch jobs/hello_lanta.sbatch
```

เมื่อใช้ partition อื่น

เปลี่ยน `-partition=debug` เป็น partition ที่ `sinfo` แสดงว่าใช้งานได้ และปรับเวลาให้สั้น

อ่านสถานะ log และผลลัพธ์

สถานะ COMPLETED บอกว่า job จบ แต่ความถูกต้องอยู่ที่ output และ log

หลังส่งงาน

```
squeue -u $USER

JOBID=<job-id>
sacct -j $JOBID --format=JobID,JobName,State,Elapsed,AllocCPUS,MaxRSS,ExitCode
tail -50 logs/hello_${JOBID}.out
tail -50 logs/hello_${JOBID}.err
cat results/hello_${JOBID}.txt
```

อ่าน state ที่พบบ่อย

State	ความหมายและสิ่งที่ต้องทำ
PENDING	รอคิว ตรวจ reason ด้วย `squeue -j JOBID -o "\%.18i \%.9P \%.20j \%.8T \%.20R"`
RUNNING	งานกำลังรัน ดู log ถ้าสคริปต์เขียนความคืบหน้าไว้
COMPLETED	process จบปกติ ต้องตรวจไฟล์ผลลัพธ์ต่อ
FAILED	อ่าน error log และ exit code
TIMEOUT	เวลาที่ขอสั้นกว่ารันจริง หรือข้อมูลนำเข้าใหญ่เกินสำหรับทรัพยากรที่ขอ
OOM	หน่วยความจำที่ขอต่ำกว่าการใช้จริง ลดขนาดข้อมูลนำเข้าหรือเพิ่มหน่วยความจำตามหลักฐาน

บันทึก

เพิ่มหมายเลขงานและคำสรุปหนึ่งบรรทัดลง `notes/job-history.tsv` หลังรันทุกครั้ง

งาน CPU

ตัวอย่างงาน Python บน CPU

ใช้หลาย CPU ในเครื่องเดียวเพื่อประมาณค่า pi

ตัวอย่าง นี้ ใช้ Python แบ่ง งาน คำนวณ เป็น หลาย worker ใน เครื่อง เดียว จำนวน worker อ่าน จาก `SLURM_CPUS_PER_TASK` เพื่อให้โค้ดใช้ CPU เท่ากับที่สคริปต์ Slurm ขอไว้

```
cat > src/parallel_pi.py <<'PY'
import argparse, math, multiprocessing as mp, os, random, time
from pathlib import Path

def count_inside(seed_and_n):
    seed, n = seed_and_n
    rng = random.Random(seed)
    inside = 0
    for _ in range(n):
        x, y = rng.random(), rng.random()
        inside += (x*x + y*y) <= 1.0
    return inside

parser = argparse.ArgumentParser()
parser.add_argument("--samples", type=int, default=1000000)
parser.add_argument("--workers", type=int, default=int(os.environ.get("SLURM_CPUS_PER_TASK", "1")))
args = parser.parse_args()

workers = max(1, args.workers)
chunk = math.ceil(args.samples / workers)
t0 = time.time()
with mp.Pool(processes=workers) as pool:
    inside = sum(pool.map(count_inside, [(1000+i, min(chunk, args.samples-i*chunk)) for i in range(workers)]))
pi = 4.0 * inside / args.samples

Path("results").mkdir(exist_ok=True)
job_id = os.environ.get("SLURM_JOB_ID", "manual")
Path(f"results/pi_{job_id}.txt").write_text(f"samples={args.samples}\nworkers={workers}\npi={pi}\nelapsed={time.time()-t0:.3f}\n")
print(f"pi={pi} workers={workers}")
PY
```

```
cat > jobs/parallel_pi.sbatch <<'SLURM'
#!/bin/bash
#SBATCH --job-name=parallel_pi
#SBATCH --partition=compute
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=8
#SBATCH --time=00:15:00
#SBATCH --output=logs/pi_%j.out
#SBATCH --error=logs/pi_%j.err
##SBATCH --account=<project-account>

module purge
module load cray-python/3.10.10 2>/dev/null || module load python 2>/dev/null || true
cd "$SLURM_SUBMIT_DIR"
export OMP_NUM_THREADS=1
python src/parallel_pi.py --samples 2000000 --workers ${SLURM_CPUS_PER_TASK:-1}
SLURM

sbatch jobs/parallel_pi.sbatch
```

งานหลายพารามิเตอร์

ส่งงานเดียวหลายชุดพารามิเตอร์

ใช้เมื่อคำสั่งเหมือนกัน แต่ข้อมูลหรือพารามิเตอร์ต่างกัน

Job array หรือการส่งงานย่อยหลายชุดจากสคริปต์เดียว ช่วยให้ Slurm แยก log ตามหมายเลขงานย่อยได้ชัดเจน เหมาะกับการลองหลายพารามิเตอร์ หลายพื้นที่ หลายไฟล์ หรือหลาย seed ไฟล์ `pi-params.csv` ในตัวอย่างนี้เก็บค่า `samples` และ `workers` หนึ่งชุดต่อหนึ่งบรรทัด

```
cat > configs/pi-params.csv <<'EOF'
100000,1
200000,2
400000,4
800000,4
1600000,8
EOF

cat > src/array_pi.py <<'PY'
import argparse, subprocess
parser = argparse.ArgumentParser()
parser.add_argument("--line", required=True)
args = parser.parse_args()
samples, workers = args.line.split(",")
subprocess.check_call([
    "python", "src/parallel_pi.py",
    "--samples", samples,
    "--workers", workers,
])
PY
```

```
cat > jobs/pi_array.sbatch <<'SLURM'
#!/bin/bash
#SBATCH --job-name=pi_array
#SBATCH --partition=compute
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=8
#SBATCH --time=00:20:00
#SBATCH --array=1-5
#SBATCH --output=logs/pi_array_%A_%a.out
#SBATCH --error=logs/pi_array_%A_%a.err
##SBATCH --account=<project-account>

module purge
module load cray-python/3.10.10 2>/dev/null || module load python 2>/dev/null || true
cd "$SLURM_SUBMIT_DIR"
LINE=$(sed -n "${SLURM_ARRAY_TASK_ID}p" configs/pi-params.csv)
python src/array_pi.py --line "$LINE"
SLURM

sbatch jobs/pi_array.sbatch
```

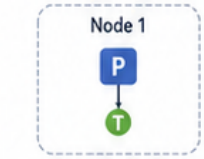
LANTA HPC ENVIRONMENT

เมทริกซ์รูปแบบการเขียนโปรแกรมขนาน

เลือกแนวทางให้เหมาะกับโหนด โครงสร้างโปรแกรม และเป้าหมายของงาน

1 Serial

ฮาร์ดแวร์ที่เหมาะ: 1 โหนด



P = Process T = Thread

รูปแบบการทำงาน: 1 process | 1 thread

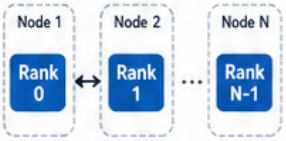
เหมาะกับงานแบบใด:
- งานเล็ก
- งานทดสอบ
- งานที่ยังไม่ต้องขนาน

จุดเด่น:
- ง่าย
- ตรงไปตรงมา
- ดีสำหรับเริ่มต้นและ debug

ข้อควรระวัง:
- ช้าเมื่อข้อมูลใหญ่หรือคำนวณหนัก

2 MPI

ฮาร์ดแวร์ที่เหมาะ: หลายโหนด



↔ = Message

รูปแบบการทำงาน: หลาย ranks สื่อสารผ่านเครือข่าย

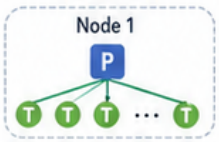
เหมาะกับงานแบบใด:
- distributed memory
- cluster scaling
- งานที่ต้องกระจายข้อมูลข้ามโหนด

จุดเด่น:
- ขยายไปหลายโหนดได้ดี
- ควบคุมการสื่อสารละเอียด

ข้อควรระวัง:
- ต้องจัดการ data distribution และ communication cost

3 OpenMP

ฮาร์ดแวร์ที่เหมาะ: 1 โหนด



P = Process T = Thread

รูปแบบการทำงาน: 1 process | many threads

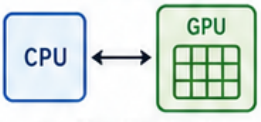
เหมาะกับงานแบบใด:
- shared memory ภายในโหนดเดียว
- ลูปขนาน
- งาน CPU-bound

จุดเด่น:
- เพิ่มความเร็วได้ง่ายกว่าการกระจายข้ามโหนด
- ใช้กับ C/C++/Fortran ได้ดี

ข้อควรระวัง:
- จำกัดอยู่ในหน่วยความจำร่วมของโหนดเดียว
- ต้องระวัง race condition

4 GPU / CUDA

ฮาร์ดแวร์ที่เหมาะ: GPU node



รูปแบบการทำงาน: CPU process + GPU kernels

CPU = Process
GPU = Kernels

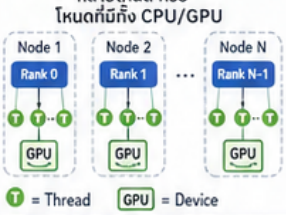
เหมาะกับงานแบบใด:
- งานที่ขนานสูง
- AI/ML
- matrix operations
- simulation kernels

จุดเด่น:
- เร่งงานได้มากเมื่ออัลกอริทึมเหมาะกับ GPU

ข้อควรระวัง:
- ต้องจัดการ memory transfer (host ↔ device)
- ออกแบบ kernel ให้เหมาะ

5 Hybrid

ฮาร์ดแวร์ที่เหมาะ: หลายโหนด หรือ โหนดที่มีทั้ง CPU/GPU



T = Thread GPU = Device

รูปแบบการทำงาน: MPI ranks + threads หรือ GPUs

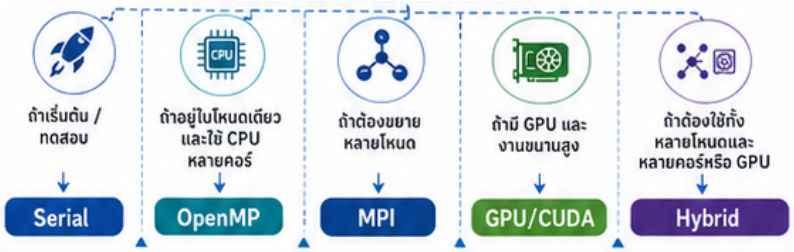
R = MPI Rank
T = Thread
GPU = Device

เหมาะกับงานแบบใด:
- ใช้ทรัพยากรของแต่ละโหนดให้เต็ม
- เช่น MPI + OpenMP, MPI + CUDA

จุดเด่น:
- ยืดหยุ่นสูง
- ใช้ระบบได้คุ้มที่สุด

ข้อควรระวัง:
- ซับซ้อนที่สุด
- ต้องวางโครงสร้างโปรแกรมดี

เลือกแบบไหนดี?



คำสำคัญ

- **node** — เครื่องคอมพิวเตอร์หนึ่งเครื่องในระบบ ซึ่งมี CPU, หน่วยความจำ และ I/O
- **process** — การทำงานของโปรแกรมหนึ่งตัวที่รันอยู่ มีพื้นที่หน่วยความจำของตนเอง
- **thread** — หน่วยการทำงานย่อยภายใน process ใช้หน่วยความจำร่วมกัน
- **rank** — ตัวตนของ process ใน MPI ใช้สื่อสารกับ ranks อื่น
- **shared memory** — หน่วยความจำร่วมกันในโหนดเดียวกัน เข้าถึง ร่วมกัน
- **distributed memory** — หน่วยความจำที่แยกกันในแต่ละโหนด ต้องสื่อสารผ่านเครือข่าย

หลักคิดสำคัญ: เลือกแนวทางที่เหมาะกับสถาปัตยกรรมของระบบและลักษณะงาน เพื่อให้ได้ประสิทธิภาพสูงสุดใน LANTA HPC ENVIRONMENT

เลือกรูปแบบงานขนาน

เลือกวิธีแบ่งงานให้ตรงกับหน่วยความจำและการสื่อสาร

แนวทางแรกคือรันแบบ serial เพื่อมีผลลัพธ์ฐานสำหรับเปรียบเทียบ จากนั้นเลือกวิธีขนานตามรูปแบบของข้อมูลและการสื่อสาร หน่วยความจำร่วมใช้หลาย thread หรือหลาย process ในเครื่องเดียว หน่วยความจำแยกใช้หลาย process ที่สื่อสารกันด้วย MPI ส่วน GPU ใช้หน่วยประมวลผลเฉพาะสำหรับงานเมทริกซ์ tensor และ kernel จำนวนมาก

ตารางสำหรับเลือกแนวทาง

แนวทาง	เหมาะกับ	สัญญาณที่ต้องตรวจ
Serial	งานเล็กหรือ baseline	ต้องมี baseline ก่อนเพิ่มงานขนานเสมอ
Multiprocessing	Python ที่ใช้ CPU ในเครื่องเดียว	หน่วยความจำต่อ worker และต้นทุนจากการส่งข้อมูล
OpenMP	C/ C+ +/ Fortran แบบหน่วยความจำร่วม	จำนวน thread และ race condition
MPI	หลาย process หรือหลาย node	การสื่อสาร, load balance และ I/O
GPU	tensor, matrix, deep learning, CUDA	โค้ดต้องใช้ GPU จริงและการย้ายข้อมูลมีต้นทุน

Amdahl แบบใช้งาน

ถ้าโค้ดมีส่วนที่ parallel ได้ 90% ต่อให้เพิ่มทรัพยากรมากเท่าใด speedup สูงสุดทางทฤษฎีคือประมาณ 10 เท่า ดังนั้นก่อนขยายงานต้องรู้ว่าส่วนใด serial ส่วนใด parallel และเวลาเสียไปกับ I/O หรือ communication เท่าใด

OPENMP

ตัวอย่าง C/OpenMP ขนาดเล็ก

ใช้หน่วยความจำร่วมในเครื่องเดียวและควบคุมด้วย OMP_NUM_THREADS

ตัวอย่างนี้คอมไพล์โปรแกรม C ขนาดเล็กแล้วให้ Slurm จัด CPU ภายในเครื่องเดียว จำนวน thread อ่านจาก `SLURM_CPUS_PER_TASK` เพื่อให้ตรงกับทรัพยากรที่ขอ

```
cat > src/omp_hello.c <<'C'
#include <stdio.h>
#include <omp.h>

int main() {
  #pragma omp parallel
  {
    int tid = omp_get_thread_num();
    int nthreads = omp_get_num_threads();
    printf("hello from thread %d of %d\n", tid, nthreads);
  }
  return 0;
}
C

cat > jobs/omp_hello.sbatch <<'SLURM'
#!/bin/bash
#SBATCH --job-name=omp_hello
#SBATCH --partition=compute
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=8
#SBATCH --time=00:10:00
#SBATCH --output=logs/omp_%j.out
#SBATCH --error=logs/omp_%j.err
##SBATCH --account=<project-account>

module purge
module load gcc 2>/dev/null || true
cd "$SLURM_SUBMIT_DIR"
mkdir -p results

cc -fopenmp src/omp_hello.c -o results/omp_hello
export OMP_NUM_THREADS=${SLURM_CPUS_PER_TASK:-1}
srun results/omp_hello | sort
SLURM

sbatch jobs/omp_hello.sbatch
```

เลือก compiler wrapper

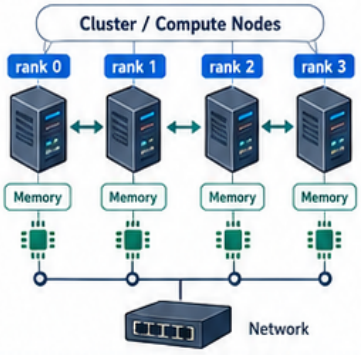
ตรวจ `module avail gcc` และดูว่า partition นั้นใช้ compiler wrapper ชื่อ `cc`, `gcc` หรือชื่ออื่น

พื้นฐานการเขียนโปรแกรม MPI

สื่อสารระหว่างโปรเซส เพื่อให้งานขนานทำงานร่วมกันเป็นระบบ

1. MPI คืออะไร

- MPI = Message Passing Interface
- มาตรฐานสำหรับโปรแกรมขนานแบบกระจาย
- แต่ละ process มีหน่วยความจำของตนเอง
- สื่อสารกันด้วยการส่งข้อความ



rank	size	communicator
ลำดับที่ของ process ใน communicator	จำนวน process ทั้งหมด ใน communicator	กลุ่มของ process ที่สื่อสารกันได้

2. โครงสร้างพื้นฐานที่ต้องรู้

A	เริ่มต้นโปรแกรม	`MPI_Init`	เริ่มสภาพแวดล้อม MPI
B	ตัวตนของ process	`MPI_Comm_rank` / `MPI_Comm_size`	รู้ว่า process นี้เป็นลำดับที่เท่าไร และมีทั้งหมดกี่ตัว
C	จบโปรแกรม	`MPI_Finalize`	ปิดการทำงานอย่างถูกต้อง
D	รูปแบบการทำงาน	SPMD	โค้ดเดียวกันรันหลาย process

3. การสื่อสารหลัก

A. Point-to-Point

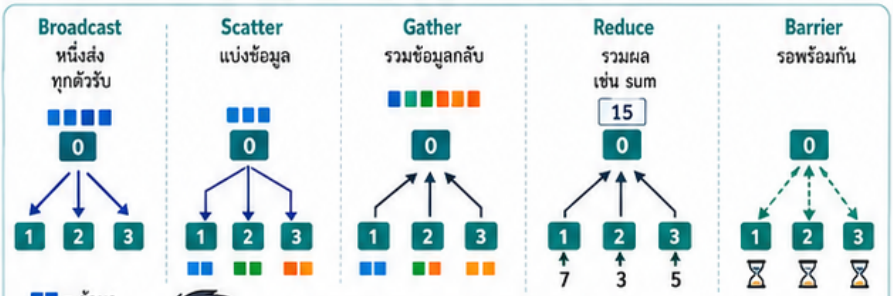
MPI_Send ส่งข้อมูล

ส่งข้อมูลจาก rank 0 → rank 1

MPI_Recv รับข้อมูล

รับข้อมูลที่ส่งจาก rank 0

B. Collective Communication



Collective ช่วยลดโค้ดและเพิ่มประสิทธิภาพ เมื่อ process หลายตัวทำงานร่วมกัน

4. เวิร์กโฟลว์การใช้งาน

1. เขียนโค้ด
2. คอมไพล์
3. รัน
4. ตรวจผล
5. ต่อยอด

```
#include <mpi.h>
int main(int argc, char **argv)
{
  ...
}
```

```
mpicc hello.c -o hello
```

```
mpirun -np 4 ./hello
```

```
Hello from rank 0 of 4
Hello from rank 1 of 4
Hello from rank 2 of 4
Hello from rank 3 of 4
```

- โดเมน decomposition
- parallel I/O
- MPI + OpenMP

ตัวอย่าง Hello World

```c
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv) {
    MPI_Init(&argc, &argv);                    // เริ่มต้น MPI
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); // ลำดับของ process นี้
    MPI_Comm_size(MPI_COMM_WORLD, &size); // จำนวน process ทั้งหมด
    printf("Hello from rank %d of %d\n", rank, size);
    MPI_Finalize();                            // จบ MPI
    return 0;
}
```

เริ่มง่าย ๆ ด้วย Hello World ก่อนจะไปสู่โปรแกรมที่ซับซ้อน

เมื่อเข้าใจพื้นฐานแล้ว

- ออกแบบงานขนานได้เป็นระบบ
- เข้าใจการสื่อสารของ process
- ใช้คลัสเตอร์ได้มีประสิทธิภาพขึ้น
- พร้อมต่อยอดสู่โปรแกรมวิทยาศาสตร์และ AI

งาน MPI

MPI สำหรับงานหลายโพรเซส

หน่วยความจำกระจายใช้การส่งข้อความระหว่างโพรเซส

MPI หรือ Message Passing Interface เป็นมาตรฐานสำหรับการเขียนโปรแกรมแบบขนานบนระบบหน่วยความจำกระจาย ใช้กับงานที่ต้องแบ่งปัญหาใหญ่ให้หลายโพรเซสรับผิดชอบคนละส่วน เช่น แบบจำลองสภาพอากาศ น้ำท่วม ไฟป่า การจำลองโมเลกุล หรือโปรแกรมวิทยาศาสตร์ที่ออกแบบมาสำหรับหลาย node ประสิทธิภาพของงาน MPI จึงขึ้นกับการแบ่งงานให้สมดุล ต้นทุนการสื่อสาร และการวัดผลเมื่อเพิ่มจำนวนโพรเซส

แนวคิดที่ต้องตรวจ

คำ	ความหมาย
communicator	กลุ่มของโพรเซสที่สื่อสารกันได้ โดย `MPI_COMM_WORLD` รวมทุกโพรเซสที่โปรแกรมเริ่มมา
rank	หมายเลขประจำตัวของโพรเซสใน communicator เริ่มจาก 0 ถึง size-1
size	จำนวนโพรเซสทั้งหมดใน communicator
point-to-point	การส่งข้อมูลระหว่างสองโพรเซสโดยตรง เช่น send และ receive
collective	คำสั่งสื่อสารที่ทุกโพรเซสใน communicator ต้องเข้าร่วม เช่น broadcast, scatter, gather และ reduce

อ่านคำขอทรัพยากร

ในสคริปต์ Slurm สำหรับ MPI ค่า `-nodes=2` และ `-ntasks-per-node=4` หมายถึงขอ 2 node และเปิดโพรเซส MPI 4 ตัวต่อ node รวม 8 โพรเซส ค่า `-cpus-per-task=1` หมายถึงแต่ละโพรเซส MPI ใช้ CPU หนึ่ง core ส่วน `srun` เป็นคำสั่งเปิดงานที่อ่านทรัพยากรจาก Slurm แล้ววางโพรเซสให้ตรงกับทรัพยากรที่ได้รับจัดสรร จึงเป็นวิธีที่ควรใช้ในงานจริงบน LANTA

```
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=4
#SBATCH --cpus-per-task=1

srun -n $SLURM_NTASKS results/mpi_hello
```

ตัวอย่าง MPI ที่คอมไพล์และรันได้

เลือก compiler wrapper ให้ตรงกับสภาพแวดล้อมจริง

ตัวอย่าง C เดินตามโครงสร้างพื้นฐานของ MPI: `MPI_Init` เริ่มระบบ MPI, `MPI_Comm_rank` อ่านหมายเลขของโพรเซส, `MPI_Comm_size` อ่านจำนวนโพรเซสทั้งหมด, `MPI_Get_processor_name` อ่านชื่อเครื่องที่โพรเซสนั้นรันอยู่ และ `MPI_Finalize` ปิดงาน MPI ผลลัพธ์ควรมีหนึ่งบรรทัดต่อหนึ่ง rank ถ้าจำนวนบรรทัดไม่ตรงกับ `-ntasks` ให้ตรวจ module, compiler wrapper และคำสั่ง `srun`

```
cat > src/mpi_hello.c <<'C'
#include <mpi.h>
#include <stdio.h>

int main(int argc, char **argv) {
  MPI_Init(&argc, &argv);
  int rank, size;
  char name[MPI_MAX_PROCESSOR_NAME];
  int len = 0;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Get_processor_name(name, &len);
  printf("rank %d of %d on %s\n", rank, size, name);
  MPI_Finalize();
  return 0;
}
C

cat > jobs/mpi_hello.sbatch <<'SLURM'
#!/bin/bash
#SBATCH --job-name=mpi_hello
#SBATCH --partition=compute
#SBATCH --nodes=1
#SBATCH --ntasks=4
#SBATCH --cpus-per-task=1
#SBATCH --time=00:10:00
#SBATCH --output=logs/mpi_%j.out
#SBATCH --error=logs/mpi_%j.err
##SBATCH --account=<project-account>

module purge
module load OpenMPI/4.1.4 2>/dev/null || module load cray-mpich/8.1.25 2>/dev/null || true
cd "$SLURM_SUBMIT_DIR"
mkdir -p results

if command -v mpicc >/dev/null 2>&1; then
  mpicc src/mpi_hello.c -o results/mpi_hello
else
  cc src/mpi_hello.c -o results/mpi_hello
fi

srun -n ${SLURM_NTASKS:-4} results/mpi_hello | sort
SLURM

sbatch jobs/mpi_hello.sbatch
```

การคำนวณเชิงวิทยาศาสตร์บน LANTA HPC

จากโจทย์วิจัย สู่การรันงาน วิเคราะห์ผล และต่อยอดการค้นพบ

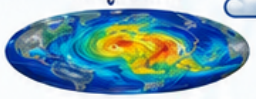
LANTA HPC คือโครงสร้างพื้นฐานสมรรถนะสูง ที่ช่วยให้นักวิจัยสามารถคำนวณเชิงวิทยาศาสตร์ ขนาดใหญ่ ซับซ้อน และใช้เวลานานได้อย่างมีประสิทธิภาพ ตั้งแต่โจทย์วิจัยจนถึงการค้นพบใหม่

LANTA HPC

ทรงพลัง | ยืดหยุ่น | ปลอดภัย | พร้อมสนับสนุน

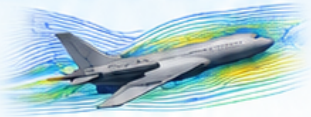
1 งานวิทยาศาสตร์ที่เหมาะกับ LANTA

แบบจำลองสภาพอากาศ และภูมิอากาศ



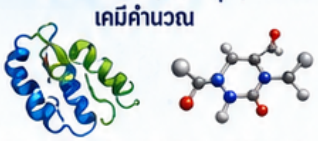
พยากรณ์อากาศ ภูมิอากาศ และการเปลี่ยนแปลงสภาพภูมิอากาศ ด้วยแบบจำลองความละเอียดสูง

พลศาสตร์ของไหล / CFD



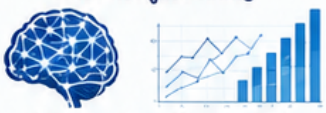
จำลองการไหล แรงกระทำ การถ่ายเทความร้อน เพื่อออกแบบและเพิ่มประสิทธิภาพ ระบบและอุปกรณ์

ชีวสารสนเทศ / โมเลกุล / เคมีคำนวณ



วิเคราะห์จีโนม จำลองโปรตีนและปฏิกิริยาเคมี เพื่อทำความเข้าใจและออกแบบโมเลกุลใหม่

AI for Science / วิเคราะห์ข้อมูลขนาดใหญ่



ใช้ AI และการวิเคราะห์ข้อมูลขนาดใหญ่ เพื่อค้นหาแพทเทิร์นและองค์ความรู้ใหม่

2 เวิร์กโฟลว์การทำงาน

1 เตรียมข้อมูลและโค้ด

- จัดเตรียมข้อมูล
- ตรวจสอบความถูกต้อง
- เขียน/ปรับโค้ด

2 เข้าใช้งานระบบ (SSH, login node)

- SSH เข้าใช้งาน
- ยืนยันตัวตน (OTP/Key)
- เริ่มต้นที่ login node

3 โหลด environment และ modules

```
$ module avail
$ module load
  gcc/12.2.0
  openmpi/4.1.5 5
  python/3.11
```

- module load ซอฟต์แวร์
- ตั้งค่า environment
- ตรวจสอบเวอร์ชัน

4 คอมไพล์ / ทดสอบ

- คอมไพล์โปรแกรม
- ทดสอบขนาดเล็ก
- ตรวจสอบผลลัพธ์เบื้องต้น

5 ส่งงานผ่าน Slurm

```
$ sbatch job.sh
Submitted batch job
123456
```

- เขียนสคริปต์งาน (job.sh)
- ส่งงานด้วย sbatch
- งานเข้า Queue

6 ติดตามงานและจัดเก็บผลลัพธ์

- ตรวจสอบสถานะด้วย sacct, squeue
- ดู logs และไฟล์ผลลัพธ์
- จัดเก็บใน /home, /project, /scratch

7 วิเคราะห์ผลและทำซ้ำได้

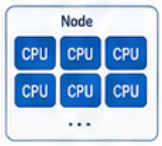
- วิเคราะห์และแปลผล
- สรุปผล สร้างภาพ
- ทำซ้ำ ปรับปรุง ทำซ้ำได้

คำสั่งที่ใช้บ่อย: module load | sbatch | squeue | sacct | scancel | less / tail (logs)

พื้นที่จัดเก็บข้อมูล: /home (งานส่วนตัว) | /project (งานวิจัยร่วม) | /scratch (พื้นที่ชั่วคราวความเร็วสูง)

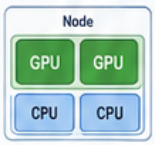
3 รูปแบบการรันงาน

CPU batch job



เหมาะกับงานทั่วไป ที่ใช้เฉพาะ CPU เช่น การวิเคราะห์ข้อมูล การประมวลผลเชิงสถิติ

GPU job

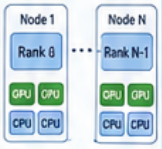


เหมาะกับงานที่มีการคำนวณแบบขนานจำนวนมาก เช่น AI, Deep Learning, Monte Carlo

MPI multi-node job

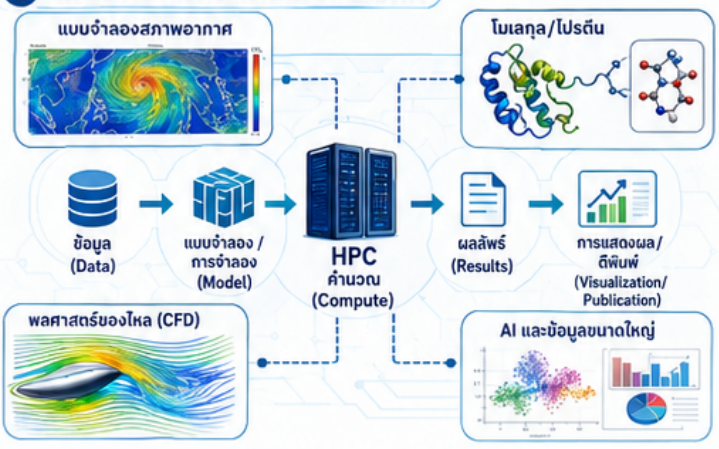
เหมาะกับงานจำลองขนาดใหญ่ ที่ต้องการสื่อสารระหว่างโหนด เช่น แบบจำลองภูมิอากาศ, CFD

Hybrid MPI + GPU



ผสาน MPI และ GPU เพื่อประสิทธิภาพสูงสุด สำหรับงานขนาดใหญ่และซับซ้อน

4 ตัวอย่างงานวิทยาศาสตร์บน LANTA



5 สิ่งที่ต้องคิดก่อนส่งงาน

- จำนวน cores / GPUs เหมาะสมกับงาน
- walltime กำหนดเวลาให้เพียงพอ
- memory ต่อ node เหมาะสม
- input-output files จัดเตรียมครบ
- software version ใช้เวอร์ชันที่ถูกต้อง
- queue / partition เลือกให้เหมาะสม
- scaling test ทดสอบประสิทธิภาพก่อนรันใหญ่
- reproducibility บันทึกขั้นตอนทำซ้ำได้

เตรียมพร้อมดี = รันงานราบรื่น ได้ผลลัพธ์เชื่อถือได้

6 ผลที่ได้เมื่อใช้อย่างเป็นระบบ

รันงานได้เสถียร

ลดการสิ้นเหลวของงาน ความเสถียรสูง

ใช้ทรัพยากรคุ้มค่า

ใช้ CPU/GPU/หน่วยความจำ อย่างมีประสิทธิภาพ

วิเคราะห์ได้เร็วขึ้น

ลดเวลาในการคำนวณ และวิเคราะห์ผล

ต่อยอดสู่งานวิจัย และนวัตกรรม

สร้างองค์ความรู้ใหม่ สู่ผลกระทบเชิงบวก

LANTA HPC

ระบบสมรรถนะสูงเพื่อวิจัยและนวัตกรรมของไทย

คิด → จำลอง → คำนวณ → ค้นพบ → สร้างคุณค่า

ทำงานเป็นทีม • ใช้ทรัพยากรอย่างรับผิดชอบ • มุ่งสู่การค้นพบที่ยิ่งใหญ่

สายงานวิทยาศาสตร์บน LANTA

โจทย์วิทยาศาสตร์ต้องมีข้อมูลนำเข้า แบบจำลอง พารามิเตอร์ ผลลัพธ์ และการตรวจผล

งานวิทยาศาสตร์บน HPC มักประกอบด้วยการเตรียมข้อมูล การรันแบบจำลอง การสรุปผล และการทำภาพประกอบ หัวใจของสายงานคือการกำหนดข้อมูลนำเข้าและผลลัพธ์ของแต่ละขั้นให้ชัดเจน และทำให้รันซ้ำได้โดยเปลี่ยนพารามิเตอร์อย่างควบคุมได้

โครงสร้างสายงาน

ขั้น	ตัวอย่างหลักฐาน
เตรียมข้อมูล	สคริปต์ รายการข้อมูลนำเข้า ข้อมูลที่ทำความสะอาดแล้ว และค่า checksum
รันแบบจำลอง	ไฟล์พารามิเตอร์ สคริปต์ Slurm รายการ module และบันทึกการรัน
สรุปผล	ไฟล์สรุป CSV ตัวชี้วัด รูปภาพ และบันทึกการตรวจผล
ขยายขนาดงาน	เวลารันตั้งต้น คำขอทรัพยากรใหม่ และตารางเปรียบเทียบ

คำสั่งจัดโฟลเดอร์ต่อการรันหนึ่งครั้ง

```
RUN=diffusion_small_$(date +%Y%m%d_%H%M%S)
mkdir -p results/$RUN
cp configs/diffusion-small.env results/$RUN/
cp jobs/diffusion.sbatch results/$RUN/
echo "$RUN" > notes/latest-run.txt
```

ตัวอย่างโมเดล

ตัวอย่างแบบจำลอง diffusion 1 มิติ

ใช้เป็นสะพานจากสคริปต์ Slurm ไปสู่งานวิทยาศาสตร์ที่มีพารามิเตอร์และผลลัพธ์ตรวจได้

ตัวอย่างแบบจำลอง diffusion เป็นงานฝึกขนาดเล็กสำหรับรูปแบบงานวิทยาศาสตร์: Python รับพารามิเตอร์จากบรรทัดคำสั่ง เขียนผลลัพธ์เป็น CSV และ Slurm ใส่หมายเลขงานไว้ในชื่อไฟล์ สิ่งที่ควรตรวจคือหัวตาราง จำนวนบรรทัด เวลารัน และบันทึกการรัน

```
cat > src/diffusion_1d.py <<'PY'
import argparse, csv, math, time
from pathlib import Path

parser = argparse.ArgumentParser()
parser.add_argument("--n", type=int, default=200)
parser.add_argument("--steps", type=int, default=500)
parser.add_argument("--alpha", type=float, default=0.15)
parser.add_argument("--output", default="results/diffusion.csv")
args = parser.parse_args()

u = [0.0] * args.n
for i in range(args.n):
    x = i / (args.n - 1)
    u[i] = math.exp(-200.0 * (x - 0.5) ** 2)

t0 = time.time()
for _ in range(args.steps):
    v = u[:]
    for i in range(1, args.n - 1):
        v[i] = u[i] + args.alpha * (u[i-1] - 2*u[i] + u[i+1])
    u = v

Path(args.output).parent.mkdir(parents=True, exist_ok=True)
with open(args.output, "w", newline="") as f:
    w = csv.writer(f)
    w.writerow(["i", "value"])
    for i, value in enumerate(u):
        w.writerow([i, f"{value:.8f}"])
print(f"output={args.output} elapsed={time.time()-t0:.3f}")
PY

cat > jobs/diffusion.sbatch <<'SLURM'
#!/bin/bash
#SBATCH --job-name=diffusion
#SBATCH --partition=compute
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
#SBATCH --time=00:10:00
#SBATCH --output=logs/diffusion_%j.out
#SBATCH --error=logs/diffusion_%j.err
##SBATCH --account=<project-account>

module purge
module load cray-python/3.10.10 2>/dev/null || module load python 2>/dev/null || true
cd "$SLURM_SUBMIT_DIR"
python src/diffusion_1d.py --n 400 --steps 1000 --output results/diffusion_${SLURM_JOB_ID}.csv
SLURM

sbatch jobs/diffusion.sbatch
```

งานข้อมูล

ตัวอย่างงานประมวลผลข้อมูล

สร้างข้อมูล สรุปผล และเก็บผลลัพธ์สำหรับตรวจซ้ำ

งานประมวลผลข้อมูลอ่านง่ายขึ้นเมื่อแยกขั้นสร้างข้อมูล สรุปผล และตรวจผลออกจากกัน ข้อมูลขนาดเล็กช่วยเปิดดูรูปแบบไฟล์ก่อนขยายไปสู่ข้อมูลจริงขนาดใหญ่

```
cat > src/make_sensor_data.py <<'PY'
import csv, math, random
from pathlib import Path
Path("input").mkdir(exist_ok=True)
random.seed(7)
with open("input/sensor.csv", "w", newline="") as f:
    w = csv.writer(f)
    w.writerow(["minute", "station", "pm25"])
    for minute in range(1440):
        for station in range(5):
            value = 18 + 10*math.sin(minute/1440*6.283) + random.random()*5 + station
            w.writerow([minute, f"S{station}", f"{value:.2f}"])
PY

cat > src/summarize_sensor.py <<'PY'
import csv, statistics
from collections import defaultdict
from pathlib import Path

groups = defaultdict(list)
with open("input/sensor.csv") as f:
    for row in csv.DictReader(f):
        groups[row["station"]].append(float(row["pm25"]))

Path("results").mkdir(exist_ok=True)
with open("results/sensor_summary.csv", "w", newline="") as f:
    w = csv.writer(f)
    w.writerow(["station", "count", "mean", "max"])
    for station, values in sorted(groups.items()):
        w.writerow([station, len(values), f"{statistics.mean(values):.2f}", f"{max(values):.2f}"])
print("results/sensor_summary.csv")
PY
```

```
python src/make_sensor_data.py
python src/summarize_sensor.py
head results/sensor_summary.csv
```

ทำให้คนอื่นรันซ้ำได้

ผลลัพธ์ที่ดีต้องมีคำสั่งและเงื่อนไขประกอบ

ไฟล์ README ต่อ run

```
RUN=diffusion_${SLURM_JOB_ID:-manual}
mkdir -p results/$RUN

cat > results/$RUN/README.txt <<EOF
question: 1D diffusion baseline
workspace: $(pwd)
date: $(date -Is)
input: generated by src/diffusion_1d.py
script: src/diffusion_1d.py
job: jobs/diffusion.sbatch
result: results/diffusion_${SLURM_JOB_ID:-manual}.csv
validation: CSV has header i,value and n rows
EOF
```

ตรวจผลแบบเร็ว

```
head results/diffusion_<job-id>.csv
wc -l results/diffusion_<job-id>.csv
tail logs/diffusion_<job-id>.out
sacct -j <job-id> --format=JobID,State,Elapsed,MaxRSS,ExitCode
```

สำหรับ scientific model

เก็บ parameter file, input subset, mesh/grid size, timestep, module และ compiler

สำหรับงาน AI

เก็บการแบ่งชุดข้อมูล seed สภาพแวดล้อม ชนิด GPU, metric และตำแหน่ง checkpoint

ออกแบบอนาคตด้วย AI

AI ถูกสร้าง ผสาน และใช้กับงานวิทยาศาสตร์อย่างไร

แผนภาพสรุปกลยุทธ์การขนานงาน วงจรงานวิทยาศาสตร์ และการเชื่อม AI กับแบบจำลองเชิงฟิสิกส์

1. กลยุทธ์การขยายสเกล: ขนานงานและประสิทธิภาพ

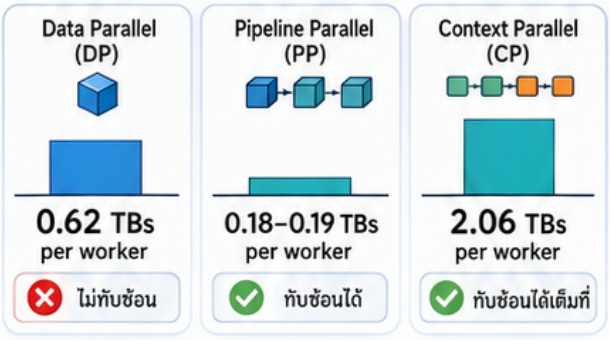
- รวมการขนานหลายมิติ เพื่อรองรับโมเดลขนาดใหญ่
- แบ่งงาน แบ่งพารามิเตอร์ และแบ่งบริบทอย่างเป็นระบบ

การทับซ้อนระหว่าง Compute และ Communication

- ทำงานคำนวณและส่งข้อมูลพร้อมกัน
- ลดเวลารอของระบบ

2. วงจรการพัฒนา AI สำหรับงานวิทยาศาสตร์

3. ความแม่นยำและการใช้หน่วยความจำ



8-bit Precision และ KV-cache Compression

- ลดคอขวดของการฝึกและการอนุมาน
- ใช้หน่วยความจำคุ้มขึ้น

4. ใส่ความรู้ฟิสิกส์ลงใน AI

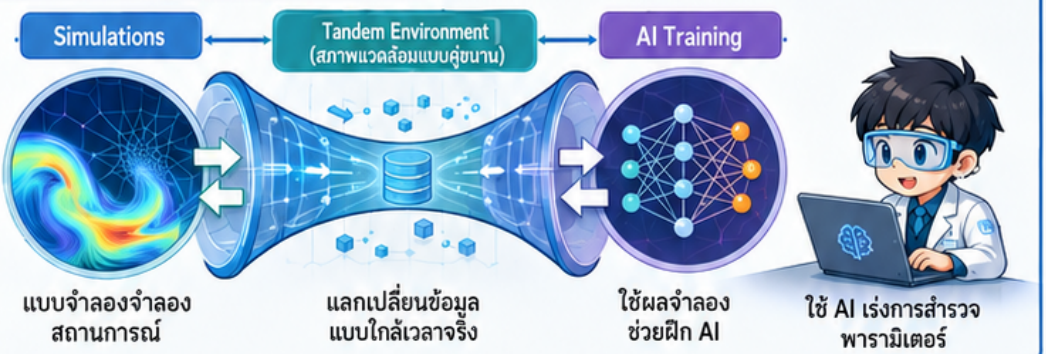
$$E_0 = -\frac{\hbar^2}{2m}\nabla^2\psi + V\psi$$

$$\nabla \cdot E = \frac{\rho}{\varepsilon_0}$$

$$\frac{\partial u}{\partial t} + \vec{v}\cdot\nabla u = 0$$

- ไม่ใช่แค่จำรูปแบบ
- ใส่ข้อจำกัดและกฎฟิสิกส์ลงในสถาปัตยกรรม
- ทำให้ผลสอดคล้องกับโลกจริงมากขึ้น

5. เชื่อม AI กับ ModSim



จุดที่ควรจำ

1. AI วิทยาศาสตร์ต้องพึ่งทั้งข้อมูล โมเดล และระบบคำนวณ
2. การขนานงานที่ดีช่วยให้โมเดลใหญ่ทำงานได้จริง
3. การทับซ้อน compute/communication ช่วยลดเวลาสูญเปล่า
4. การเชื่อม AI กับฟิสิกส์ และ ModSim ช่วยให้งานวิทยาศาสตร์แม่นยำและใช้ได้จริง

งาน AI และ GPU

GPU ช่วยได้เมื่อโค้ดใช้ GPU จริงและข้อมูลเคลื่อนย้ายอย่างเหมาะสม

ขั้นแรกของงาน AI/GPU คือยืนยันว่า job ได้ GPU จริง เห็น CUDA และ framework ใช้ GPU ได้ จากนั้นจึงค่อยฝึกโมเดลหรือรัน inference ด้วยข้อมูลขนาดเล็ก

งานตรวจ GPU

```
cat > jobs/gpu_check.sbatch <<'SLURM'
#!/bin/bash
#SBATCH --job-name=gpu_check
#SBATCH --partition=gpu
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=4
#SBATCH --gpus-per-node=1
#SBATCH --time=00:10:00
#SBATCH --output=logs/gpu_%j.out
#SBATCH --error=logs/gpu_%j.err
##SBATCH --account=<project-account>

module purge
module load CUDA/11.7.0 2>/dev/null || true
module load PyTorch/1.13.1-CUDA-11.7.0 2>/dev/null || module load cray-python/3.10.10 2>/dev/null || true
cd "$SLURM_SUBMIT_DIR"

nvidia-smi || true
python - <<'PY'
try:
    import torch
    print("torch", torch.__version__)
    print("cuda_available", torch.cuda.is_available())
    if torch.cuda.is_available():
        x = torch.randn(2000, 2000, device="cuda")
        y = x @ x
        print("matrix_sum", float(y.sum().cpu()))
except Exception as exc:
    print("python_gpu_check_error", repr(exc))
PY
SLURM

sbatch jobs/gpu_check.sbatch
```

รายการคำสั่งและแหล่งอ้างอิง

ใช้เทียบคำสั่ง โครงสร้างไฟล์ และแหล่งอ้างอิงของเนื้อหา

รายการคำสั่ง

งาน	คำสั่ง
เข้าสู่ระบบ	`ssh <username>@lanta.nstda.or.th`
ย้ายไฟล์	`scp`, `rsync`, SFTP ผ่านเครื่องรับส่งไฟล์
ตรวจระบบ	`sinfo`, `myquota`, `sbalance`, `module avail`
ส่งงาน	`sbatch jobs/name.sbatch`
ดูคิว	`squeue -u USER`
ดูประวัติ	`sacct -j JOBID -format=JobID,State,Elapsed,MaxRSS,ExitCode`
ยกเลิกงาน	`scancel JOBID`
อ่านบันทึก	`tail -50 logs/name_JOBID.out` และ `tail -50 logs/name_JOBID.err`

แหล่งเรียนรู้และอ้างอิงเนื้อหา

แหล่งเรียนรู้	ใช้ต่อเรื่องใด
ThaiSC และ LANTA User Guide	บัญชีผู้ใช้ การเข้า LANTA พื้นที่งาน module, Slurm, การย้ายไฟล์ และช่องทางขอความช่วยเหลือ
The Art of HPC	หนังสือเรียนฟรีของ Victor Eijkhout สำหรับพื้นฐาน scientific computing, architecture, parallel programming และ MPI/OpenMP
HPC Carpentry: Using HPC Systems	บทเรียนเปิดสำหรับผู้เริ่มใช้ shell, remote login, scheduler, file transfer และ software environment บนระบบ HPC
LLNL HPC Tutorials	คอร์สออนไลน์ฟรีเรื่อง parallel computing, Slurm, MPI, OpenMP และแนวคิดการใช้คลัสเตอร์
SchedMD Slurm Quick Start	คำสั่ง `sbatch`, `squeue`, `sacct`, `srun` และหลักการขอทรัพยากรผ่าน scheduler
Open MPI Documentation	การคอมไพล์และรันโปรแกรม MPI รวมถึงแนวทางใช้ launcher ร่วมกับระบบจัดคิว
OpenMP Tutorials and Articles	พื้นฐาน shared-memory parallelism, thread, directive และตัวอย่าง OpenMP สำหรับ C/C++/Fortran
NVIDIA CUDA Programming Guide และ NVIDIA DLI self-paced courses	แนวคิด GPU computing, CUDA, accelerated computing, AI และคอร์ส self-paced บางรายการที่เปิดให้เรียนฟรี

LANTA

LANTA HPC

Handbook

เข้าเครื่องครั้งแรก · สคริปต์ Slurm · งานวิทยาศาสตร์ · MPI · GPU · หลักฐาน

สายงานบน LANTA

โจทย์ ข้อมูล สภาพแวดล้อม งาน Slurm และหลักฐาน

เริ่มจากงานเล็ก บันทึกคำสั่งที่ใช้ เก็บหมายเลขงานและ log แล้วค่อยขยายงานเมื่อผลลัพธ์ถูกต้องและรันซ้ำได้

แดชบอร์ดกิจกรรม

lanta-experience.hpc-ignite.org

วันลงมือจริง

NARIT Chiang Mai

วันจันทร์ที่ 3 สิงหาคม 2569
Andromeda Room

Mae Rim, Chiang Mai

หน่วยงานที่ร่วมจัด

ThaiSC · NARIT