OS report 2025

1. Intro to Parallel Programming

Algorithm Exlanation

ใช้ Algorithm แยกตัวประกอบของ n แบบลองไปเรื่อยๆตั้งแต่ 1 ถึง $\sqrt{n}$
หลักการที่ใช้ของ parallel คือการแบ่งจำนวนเป็น chunk แล้วให้แค่ละ process รันในส่วนของตัวเอง

Pseudo-code

```
- Input: n, process
- แบ่งช่วง $[2, sqrt(n)]$ ออกเป็นจำนวน process ช่วง
- ไล่เช็คหาว่า n หารในช่วงตัวเองและบันทึกตัวที่
- ให้ "MPI.Gather" รวมกันแล้วจัดรูปผลเป็นชุดของตัวประกอบ
```

Implementation notes

- **Libraries**: python 3.13.9, mpi4py
- **I/O**: อ่าน n จาก argument ของไฟล์บันทึกเวลาและหน่วยความจำ
- **Correctness**: cross-check เทียบ version sequenctial ตอน process = 1

Experiment setup

- **Hardware**: 13th Gen Intel(R) Core(TM) i5-13500H, RAM 16 GB
- **OS & Runtime**: Fedora Linux 42 (Workstation Edition) x86_64
- **Parameters**: n หลายขนาด และจำนวน process ตั้งแต่ 1 ถึง 15

Performance Analysis

การวัดผลที่ใช้เวลาจำนวน processes เทียบกับจำนวนวินาทีและ speed up เมื่อเทียบกับ 1 process และวัด efficiency ต่อ 1 core
ตาม Amdahl's Law สูตรคือ $S = \frac{1}{(1-p)+\frac{P}{N}}$
โดยที่

- S คือ speed up
- N คือ จำนวน process
- p คือ จำนวน task ที่ parallel ได้
- $1-p$ คือ จำนวน task ที่ serquencial

Result เป้าหมาย: หาว่า program มีส่วนของ parallel part และ sequential part เท่าไร จากความสัมพันธ์ $S(N) = \frac{1}{(1-p)+\frac{p}{N}}$ จะหา p

1. หาค่า baseline $T(1)$ โดยดูจากค่า csv ตรงที่ $T(1)$
2. จัดรูปสมการให้หาค่า p_n
    1. จาก $S(N) = \frac{1}{(1-p)+\frac{P}{N}}$
    2. $\frac{1}{S(N)} = 1 - p\left(1 - \frac{1}{N}\right) \rightarrow p = \frac{1-\frac{1}{S(N)}}{1-\frac{1}{N}}$
3. หาค่า p_n สำหรับ $n = 1, 2, 3, 4, ...$

$$p_n = \frac{1 - \frac{1}{S(N)}}{1 - \frac{1}{N}}$$

1. รวมค่า p_N เพื่อค่า p ที่กลางที่สุด วิธีคือเอาค่าเฉลี่ยของ median ของ p_N ด้วย $p_N = \frac{1-\frac{1}{S(N)}}{1-\frac{1}{N}}$

Summary จากการคำนวณพบว่า parallelizable = 72.43% และส่วนของ serial = 27.57 % โดยใช้

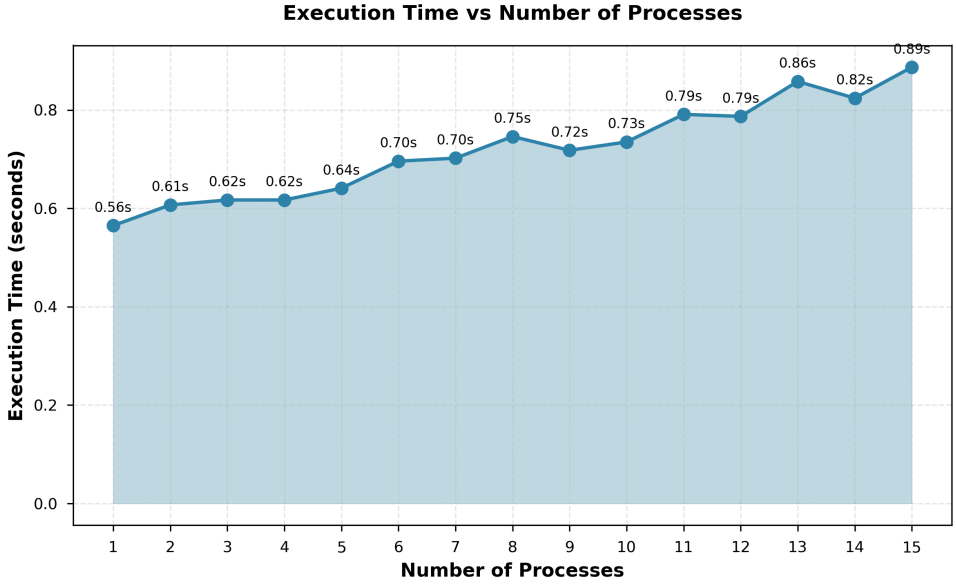

Figure 1: Speed-up comparison for small input sizes

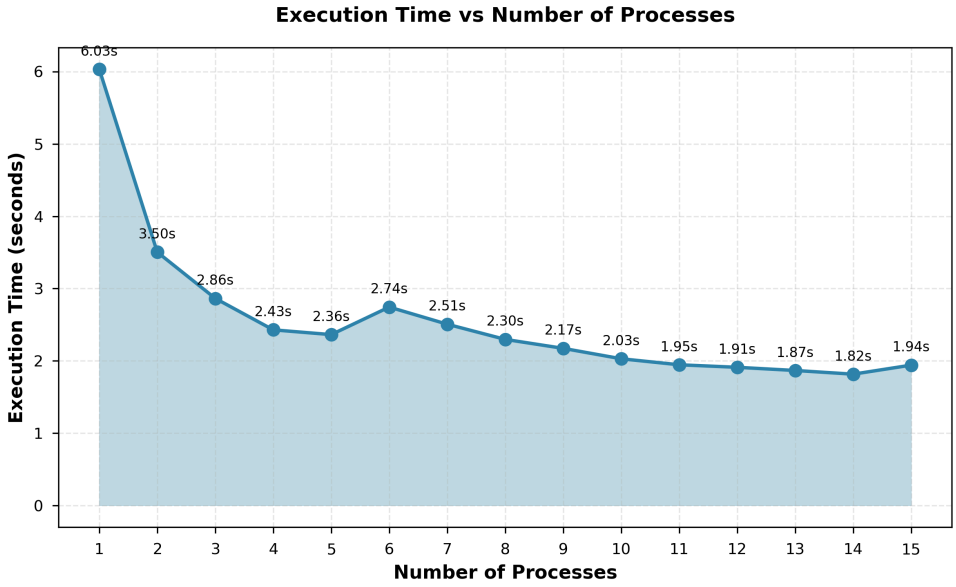

Figure 2: Speed-up comparison for large input sizes

Analysis

- Speed up เพิ่มขึ้นตามจำนวน process แต่เริ่มชะลอเมื่อเกินขนาด X เพราะ overhead (คอขวดตรง communication และ sync)
- Efficience ต่อ core ลดลงเมื่อ process มากขึ้นตาม **Amdahl's Law**: $S = \frac{1}{(1-p)+\frac{P}{N}}$

2. deadlock

เป็นการจำลอง deadlock โดยมี 3 แนวทางคือ

1. Avodiance (Banker's),
2. Detection (Wait-for Graph),
3. Resolution (find process and abort)

Simulated Resource

- total: เวกเตอร์ของทรัพยากรแต่ละชนิดทั้งหมด
- available: เวกเตอร์ของทรัพยากรที่ยังว่างอยู่
- max[pid]: เวกเตอร์ความต้องการสูงสุด each process
- alloc[pid]: เวกเตอร์ของทรัพยากรที่ยังจัดสรรค์อยู่
- need[pid]: max[pid] - alloc[pid]
- alive[pid]: สถานะว่า process ว่ายังมีชีวิตอยู่หรือ abort ไปแล้ว
- waiting_req[pid]: คำขอปัจจุบันในการสร้าง WFG

Coffman's Deadlock Conditions ใน deadlock.py สร้าง thread T_1 และ T_2 และล็อค A/B:

- Mutual Exclusion: Lock A/B เขาถึงทีละ thread
- Hold and wait: T_1 ถือ A แล้วรอ B ในตอนนั้นเอง T_2 ถือ B แล้วรอ A
- No preemption: ล็อคไม่ถูกยึดคืนอัตโนมัติ
- Circular wait: $T_1 \to A \text{ wait } B$, $T_2 \to B \text{ wait } A$ เป็นวงจร

Deadlock Avoidance using Banker's Algorithm ในโค้ด python เมื่อ ResourceManager(use_bankers=True)

1. request(pid, req) ถ้าจะลองทำให้ update available alloc need ชั่วคราว
2. เรียก is_safe_state() หา safe sequence:
   - work = available.copy()
   - finish[pid] = False by default
   - วนหา process ที่ need[pid] $\leq$ work แล้วก็ทำจนจบแล้ว finish[pid] = True
   - ถ้าทำจน finish ทุกโปรเซส = True ก็คือ safe ไม่งั้น unsafe
3. ระบบนี้ไม่เข้าสู้ unsafe state เสี่ไม่มี deadlock แน่นอน

Deadlock Detectoin using Wait-for Graph เมื่อ ResourceManager(use_bankers=False) และมีคำขอที่รอ:

- build_wait_for_graph() ส่วน wait for graph โหนด = process, edge = p รอ q
- detect_cycle(wait for graph) ใช้ 2 color problem ตรวจวงวรแล้วพบว่า deadlock เกิดขึ้นแล้ว

Deadlock Resolution using Victim Selection and abort เมื่อพบ cycle:

- เลือก victim แล้วแต่นโยบายเช่น
  1. ถือทรัพยากรเยอะ
  2. อยู่นาน
- release_all_and_abort(vicim) คืนทรัพยากรทั้งหมดของเหยื่อโดยการ alive[victim]=False, ล้าง waiting_req
- ปลุกตัว tread อื่นแล้วดำเนินการต่อ

Reference:

1. “mpi4py — MPI for Python,” mpi4py Documentation. [Online]. Available: https://mpi4py.readthedocs.io/en/stable/. Accessed: Oct. 29, 2025.
2. “Simple deadlock examples,” Stack Overflow. [Online]. Available: https://stackoverflow.com/questions/1385843/simple-deadlock-examples. Accessed: Oct. 29, 2025.
3. “Program for deadlock free condition in operating system,” GeeksforGeeks. [Online]. Available: https://www.geeksforgeeks.org/operating-systems/program-for-deadlock-free-condition-in-operating-system/. Accessed: Oct. 29, 2025.
4. “Using mpi4py to parallelize a for loop on a compute cluster,” Stack Overflow. [Online]. Available: https://stackoverflow.com/questions/50979373/using-mpi4py-to-parallelize-a-for-loop-on-a-compute-cluster. Accessed: Oct. 29, 2025.
5. “Parallelizing a for loop in Python,” Computational Science Stack Exchange. [Online]. Available: https://scicomp.stackexchange.com/questions/19586/parallelizing-a-for-loop-in-python. Accessed: Oct. 29, 2025.
6. “Parallel programming in Python – mpi4py part 1,” PDC Center for High Performance Computing, KTH Royal Institute of Technology. [Online]. Available: https://www.kth.se/blogs/pdc/2019/08/parallel-programming-in-python-mpi4py-part-1/. Accessed: Oct. 29, 2025.
7. D. Doshi, “Large file splitting with Python threading for improved performance,” Medium. [Online]. Available: https://medium.com/@darshan_doshi/large-file-splitting-with-python-threading-for-improved-performance-616f03e4622c. Accessed: Oct. 29, 2025.
8. J. Limoff, “Understanding deadlocks in Python with examples,” Medium. [Online]. Available: https://medium.com/@jaklimoff/understanding-deadlocks-in-python-with-examples-20b5e00f1eb8. Accessed: Oct. 29, 2025.
9. “Parallelize a function call with mpi4py,” Stack Overflow. [Online]. Available: https://stackoverflow.com/questions/37159923/parallelize-a-function-call-with-mpi4py. Accessed: Oct. 29, 2025.
10. “Introduction of deadlock in operating system,” GeeksforGeeks. [Online]. Available: https://www.geeksforgeeks.org/operating-systems/introduction-of-deadlock-in-operating-system/. Accessed: Oct. 29, 2025.