Parallel Programming

ภาษาที่ใช้, lib ที่ใช้, compiler มีเทคนิค
เพื่อนำเสนอ parallelism ที่ซ่อนใน algorithm
เช่น มี GPU ก็ควรเขียน code ที่รองรับ GPU
มี multi-core ควรเขียน code แบบ thread programming
มี cluster เขียน code แบบ multi-processor

Framework การเขียนโปรแกรม

Parallel Programming Models

- Parallel programming allows programmers to express parallelism in the algorithm that can match with the underlying parallel architecture.
- Parallel programming models
  - Vectorization ใช้ประโยชน์จาก vector computing ที่มีคำสั่งเฉพาะ vector operation
    - Usually based on compiler optimization dev ไม่ค่อยยุ่งมาก มันเป็น low level ที่ compiler จัดการให้
  - Thread programming
    - Threads communicate via shared memory
    - Run on single computer อาจเป็น multi-core, multi-processor ก็ได้ แต่อยู่บนเครื่องเดียว
  - Multi-process programming มีหลายเครื่อง ไม่ได้ share memory ร่วมกัน
    - Processes communicate via message passing สื่อสารด้วยการส่ง message ข้าม process
    - Run on single computer or cluster of computers

      แต่ยังต้องส่ง message passing อยู่ดี เพราะไม่ได้ใช้ตัวแปร, memory ร่วมกัน
  - High-level parallel programming
    - E.g. parallel libraries, compiler directives, functional programming
    - Implicit parallelism บอกแค่ว่าจะทำงานแบบ parallelism
    - Implemented with low-level libraries

      Explicit parallelism ต้องระบุว่าสร้างกี่ thread กี่ process สื่อสารกันอย่างไร (Detail การสร้าง sub task)

Getting System Information in Python

```
!pip install -q psutil
!pip install -q py-cpuinfo
import platform
import psutil
import cpuinfo

print('platform       :', platform.platform())
print('cpu model      :', cpuinfo.get_cpu_info()['brand_raw'])
print('physical cpu   :', psutil.cpu_count(logical=False))
print('logical cpu    :', psutil.cpu_count(logical=True))
print('virtual memory :', psutil.virtual_memory())
```

notebook

```
platform       : Windows-11-10.0.22631-SP0
cpu model      : AMD Ryzen 7 6800HS Creator Edition
physical cpu   : 8 cores
logical cpu    : 16
virtual memory : svmem(total=14702026752, available=5497176064, percent=62.6, used=9204850688, free=5497176064)
```

จ.น. core จริง

ควรเขียน code โดยไม่ควรแตก thread > logical cpu (16) แต่มากกว่า 16 นิดหน่อยได้ เพราะมีบาง thread มีการหยุดรอ อ่านไฟล์, CPU ไม่ได้ทำงานตลอด ก็สามารถ switch thread ไปรันได้

server

```
platform       : Linux-4.19.0-18-amd64-x86_64-with-glibc2.35
cpu model      : Intel(R) Xeon(R) Gold 6248R CPU @ 3.00GHz
physical cpu   : 48
logical cpu    : 96
virtual memory : svmem(total=405303078912, available=197320822784, percent=51.3, used=202437398528, free=39546245120,
588412416, buffers=5755064320, cached=157564370944, shared=4242927616, slab=8781287424)
```

2x

Google Colab

```
platform       : Linux-6.1.58+-x86_64-with-glibc2.35
cpu model      : Intel(R) Xeon(R) CPU @ 2.00GHz
physical cpu   : 1
logical cpu    : 2
virtual memory : svmem(total=13609451520, available=12530159616, percent=7.9, used=786333696, free=10113830912,
```

Kaggle

```
platform       : Linux-5.15.133+-x86_64-with-debian-bullseye-sid
cpu model      : Intel(R) Xeon(R) CPU @ 2.20GHz
physical cpu   : 2
logical cpu    : 4
virtual memory : svmem(total=33669914624, available=32371695616, percent=3.9, used=820023296, free=31915687936,
```

Getting GPU Information in Python

```
import torch
import tensorflow as tf
import subprocess

def check_gpu_info():
    """Checks for GPU availability and prints information if found.
    """

    # Check with PyTorch
    if torch.cuda.is_available():
        gpu_index = 0  # Adjust if you have multiple GPUs
        device = torch.device(gpu_index)
        print("GPU Found (PyTorch):")
        print(f"  - Device Name: {torch.cuda.get_device_name(gpu_index)}")
        print(f"  - Compute Capability: {torch.cuda.get_device_capability(gpu_index)}")
        return

    # Check with TensorFlow
    if tf.test.gpu_device_name():
        print("GPU Found (TensorFlow):")
        # TensorFlow might provide more detailed information (if you need it)
        # Explore the tf.test.gpu_device_name() output
        print(tf.test.gpu_device_name())
        return

    # System-level Check (nvidia-smi)
    try:
        info = subprocess.check_output('nvidia-smi', stderr=subprocess.STDOUT).decode()
        print("GPU Found (nvidia-smi):")
        print(info)  # Print the full output from nvidia-smi
        return
    except Exception:
        pass

    # No GPU found
    print("No GPU detected.")


check_gpu_info()
```

Google Colab

```
GPU Found (PyTorch):
  - Device Name: Tesla T4
  - Device Properties: _CudaDeviceProperties(name='Tesla T4', major=7, minor=5, total_memory=15102MB, multi_processor_count=40)
```

Measure execution time in Python

```
import time
start = time.process_time()
sum(range(100000000))
end = time.process_time()
print(end - start, "seconds")
```

```
2.578125 seconds
```

```
import datetime
start = datetime.datetime.now()
sum(range(100000000))
end = datetime.datetime.now()
print(end - start, "h:mm:ss")
```

```
0:00:02.180773 h:mm:ss
```

```
%time sum(range(100000000))
```

```
CPU times: total: 2.34 s
Wall time: 2.35 s
```

นาฬิกาของเครื่อง (import time)

```
4999999950000000
```

จับเวลาแบบทำหลายรอบ เอามาหาค่าเฉลี่ย มักจะเป็นคำสั่งที่เสร็จภายในเสี้ยววินาที

```
%timeit sum(range(100000000))
```

```
2.13 s ± 46.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

$\bar{x}$ ทำ 1 รอบ

```
%%timeit -r 1
sum(range(100000000))
sum(range(100000000))
```

%% คือจับเวลาแบบ block of code (หลายบรรทัด)

```
4.31 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

In Jupyter Notebook

Multithreading in Python

Windows thread
Pthread } รันบน multi-core จริงๆ

- Multithreading in Python does not exploit multicore/multiprocessor, due to Global Interpreter Lock (GIL). So, threads run concurrently but not in parallel. Probably good for threads with I/O.

Python เขียน multi-thread ได้ แต่ thread ต่างๆ ไม่ได้ทำงานแบบ Parallelism แต่มีประโยชน์กับโปรแกรมที่ใช้ I/O

```
import threading
import time

def thread(tid):
    for i in range(5):
        print("thread ", tid, " iteration ", i)
        time.sleep(1)

t1 = threading.Thread(target=thread, args=(1,))
t2 = threading.Thread(target=thread, args=(2,))
```

create thread

```
# Will execute both threads concurrently
t1.start()
t2.start()
```

2 thread สลับกันทำงาน สามารถแทรกกันได้

```
# Joins threads back to the parent process, which is this program
t1.join()
t2.join()
```

} ทำงานเสร็จ คืนทรัพยากรที่ใช้ใน memory

```
thread  1  iteration  0
thread  2  iteration  0
thread thread  2  iteration  1
 1  iteration  1
thread thread  1  iteration  2
 2  iteration  2
thread  1  iteration  3
thread  2  iteration  3
thread thread  1  iteration  4
 2  iteration  4
```

concurrency

→ output ของ thread 1 ที่ถูกแทรกโดย output thread 2

Multithreading in Python

- Multithreading in Python does not exploit multicore/multiprocessor, due to Global Interpreter Lock (GIL). So, threads run concurrently but not in parallel. Probably good for threads with I/O.

```
import threading

def calc_partial_pi(rank, nthreads, nsteps, dx, partial_pi):
    partial_pi[rank] = 0.0
    for i in range(rank, nsteps, nthreads):
        x = (i + 0.5) * dx
        partial_pi[rank] += 4.0 / (1.0 + x * x)
    partial_pi[rank] *= dx

nsteps = 10000000
dx = 1.0 / nsteps
nthreads = 10
partial_pi = np.zeros(nthreads)

inputs = [(rank, nthreads, nsteps, dx, partial_pi) for rank in range(nthreads)]
threads = [threading.Thread(target=calc_partial_pi, args=inp) for inp in inputs]

for t in threads:
    t.start()

for t in threads:
    t.join()

pi = partial_pi.sum()
```

Partial results are stored in shared memory.

Try changing *nthreads* and measure execution time.

Vectorization and multithreading in Numpy

- Numpy array is faster than Python list due to more efficient memory access.
- Numpy implements many functions using C libraries, with vectorization and multithreading, so they can run in parallel.
- Numpy element-wise multiply is not multithreaded, but dot product is multithreaded.

```
import random
import numpy as np
```

```
## element-wise multiplication of Python lists
xs = [random.random() for i in range(10000000)]
ys = [random.random() for i in range(10000000)]
%timeit [x*y for x, y in zip(xs, ys)]
```

xs : [1, 2, 3, 4]
ys : [5, 6, 7, 8] x
= [5, 12, 21, 32]

```
1.54 s ± 312 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
## element-wise multiplication of Numpy arrays (vectorized)
np_xs = np.random.rand(10000000)
np_ys = np.random.rand(10000000)
%timeit np.multiply(np_xs, np_ys)
```

1 thread

```
40.8 ms ± 2.19 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
## dot product of Numpy arrays (vectorized and threaded)
np_xs = np.random.rand(10000000)
np_ys = np.random.rand(10000000)
%timeit np.dot(np_xs, np_ys)
```

numpy technique

dot product ใช้ multi thread เลยยิ่ง optimized อีก

```
8.77 ms ± 343 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Dot product runs more operations than element-wise multiply, but faster due to multithreading.

Matrix Multiplication in Python

```
def matrix_multiplication(A, B, result):
    for i in range(result.shape[0]):
      for j in range(result.shape[1]):
          temp = 0.
          for k in range(A.shape[1]):
            temp += A[i, k] * B[k, j]
          result[i, j] = temp
```

```
C = np.empty([100, 100], np.float32)
%timeit matrix_multiplication(A, B, C)
```

```
1 loop, best of 3: 471 ms per loop
```

- Try larger matrices

Matrix Multiplication with Numpy

```
A = np.random.rand(100, 100)
B = np.random.rand(100, 100)
%timeit C = np.dot(A, B)
```

[100x100] · [100x100]
A B

```
The slowest run took 5.89 times longer than the fastest.
10000 loops, best of 3: 136 µs per loop
```

```
A = np.random.rand(1000, 1000)
B = np.random.rand(1000, 1000)
%timeit C = np.dot(A, B)
```

[1000x1000] · [1000x1000]
A B

```
10 loops, best of 3: 49.8 ms per loop
```

Numba

- Numba translates Python functions to optimized machine code at runtime. (Just In Time Compilation) แปลง code python เป็น parallel code
- It offers options for parallelizing Python code for CPUs and GPUs, often with only minor code changes (e.g. compiler directives).

  ใช้ประโยชน์จาก CPU / GPU

https://numba.pydata.org/

```python
def matrix_multiplication(A, B, result):
    for i in range(result.shape[0]):
        for k in range(A.shape[1]):
            for j in range(result.shape[1]):
                result[i, j] += A[i, k] * B[k, j]
```

```python
!pip install -q numba
from numba import jit
@jit
def matrix_multiplication_numba(A, B, result):
    for i in range(result.shape[0]):
        for k in range(A.shape[1]):
            for j in range(result.shape[1]):
                result[i, j] += A[i, k] * B[k, j]
```

jit - Just in time

code python การคูณเมทริกซ์เหมือนเดิม แต่ครอบด้วย numba annotate

```python
A = np.random.rand(1000, 1000)
B = np.random.rand(1000, 1000)
```

```python
C = np.empty([1000, 1000], float)
%timeit matrix_multiplication(A, B, C)
```

```
14min 39s ± 9.62 s per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```python
## matrix multiplication with Numpy
%timeit C = np.dot(A, B)
```

```
22.5 ms ± 1.73 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

very fast

```python
## matrix multiplication with Numba
C = np.empty([1000, 1000], float)
%timeit matrix_multiplication_numba(A, B, C)
```

```
370 ms ± 16.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Multiprocessing in Python

Multiprocessing in Python can exploit multicore/multiprocessor.

ถ้ามีหลาย core เขียน python เพื่อสร้างหลาย process แทน
เพราะ python ไม่ได้สร้าง multi-thread

```
import multiprocessing as mp
import numpy as np

def dotprod(a, b, q):
  q.put(np.dot(a, b))

nprocs = 10   10 processes
size = 10000000
a = np.random.rand(size)
b = np.random.rand(size)
q = mp.Queue() เก็บผลลัพธ์แต่ละ process
procs = [] list ของ process object

for i in range(nprocs) :
  start = int(i*size/nprocs)
  end = int((i+1)*size/nprocs - 1)
  procs.append(mp.Process(target=dotprod, args=(a[start:end], b[start:end], q)))

# Will execute both in parallel
for p in procs:
  p.start()

# Joins process back to the parent process, which is this program
for p in procs:
  p.join()

c = [q.get() for p in procs]
print(sum(c))
```