Introduction to Parallel Computing II

ZHANG, Rui

Department of Physics, HKUST

Oct 7^{th}, 2023

Materials referenced from the following resources:
Last year's 5001 course taught by Dr. Huijie Guan.
MPI course taught by Prof. Kevin Connington at CCNY.
https://www.youtube.com/channel/UCh9nVJoWXmFb7sLApWGcLPQ

Outline

- Recap of The Last Parallel Computing Lecture.
- Recurrence Relation and Master Theorem.
- Python for Parallel Computing
  - threading vs multiprocessing.
  - hands-on examples.

Introductory Parallel Computing I

- Definition & History
- Architecture & Platform
- Time Complexity
- Analysis of Speedup & Efficiency
- Design Strategy
- *Hello World* Example

Some Important Parallel Computing Concepts

- *Shared memory* versus *Distributed memory*.
- *Communications* - Parallel tasks typically need to exchange data.
- *Synchronization* - The coordination of parallel tasks in real time, very often associated with communications, usually involves waiting by at least one task, and can therefore cause a parallel application's wall clock execution time to increase.
- *Observed speedup* - $S = T(1)/T(P)$ with $T(P)$ being the computational time of using P processors.
- *Scalability* **-** a parallel system's (hardware and/or software) ability to demonstrate a proportionate increase in parallel speedup with the addition of more processors.
- *Parallelizability* - most problems contain parallelizable and unparallelizable components.

Brent’s Theorem and Amdahl’s Law

- Brent’s Theorem leads to the lower and upper bound of the runtime with P processors:
$$T_1/P \leq T_P \leq T_\infty + \frac{(T_1 - T_\infty)}{P} \leq \frac{T_1}{P} + T_\infty.$$
- Application: in parallel program of global sum for n numbers, the best runtime is $O(\log(n))$, which needs $n/2$ processors; One can reduce the number of processors to $O(n/\log(n))$ to retain the speedup.
- Amdahl’s Law implies that the speedup of a parallelized computation is bounded by its parallelizable portion p:
$$S = \frac{1}{(1-p) + p/s} \leq \frac{1}{1-p},$$

where s is the speedup of the parallelizable portion.

Example: Matrix Multiplication

- $C = AB$ or $C_{ij} = A_{ik}B_{kj}$.
- $n = 1$: $C_{11} = A_{11}B_{11}$.
- $n = 2$: $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$.
- $C = AB = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$.
- $n \geq 2$: $C = AB = \begin{bmatrix} A_{11} & \cdots & A_{1n} \\ \vdots & \ddots & \vdots \\ A_{n1} & \cdots & A_{nn} \end{bmatrix} \begin{bmatrix} B_{11} & \cdots & B_{1n} \\ \vdots & \ddots & \vdots \\ B_{n1} & \cdots & B_{nn} \end{bmatrix} = \begin{bmatrix} A_{1j}B_{j1} & \cdots & A_{1j}B_{jn} \\ \vdots & \ddots & \vdots \\ A_{nj}B_{j1} & \cdots & A_{nj}B_{jn} \end{bmatrix}$.

Example: Matrix Multiplication (Continued)

- Pseudo code:

MULT (C, A, B, n):

create tempory matrix $T_{n\times n}$
if $n = 1$
then $C_{11} = A_{11} \times B_{11}$
else
partition matrix // $O(1)$ *time*
spawn Mult($C_{11}, A_{11}, B_{11}, n/2$)
spawn Mult($C_{12}, A_{11}, B_{12}, n/2$)
spawn Mult($C_{21}, A_{21}, B_{11}, n/2$)
spawn Mult($C_{22}, A_{21}, B_{12}, n/2$)
spawn Mult($T_{11}, A_{12}, B_{21}, n/2$)
spawn Mult($T_{12}, A_{12}, B_{22}, n/2$)
spawn Mult($T_{21}, A_{22}, B_{21}, n/2$)
spawn Mult($T_{22}, A_{22}, B_{22}, n/2$)
sync
ADD(C, T, n)

$$C = AB = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$
$$= \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}.$$

ADD(D, A, B, n)
if n=1
then $D_{11} = A_{11} + B_{11}$
else
partition matrix
spawn ADD($D_{11}, A_{11}, B_{11}, n/2$)
spawn ADD($D_{12}, A_{12}, B_{12}, n/2$)
spawn ADD($D_{21}, A_{21}, B_{21}, n/2$)
spawn ADD($D_{22}, A_{22}, B_{22}, n/2$)
sync

Key words:
Spawn: run subroutines at the same time as parent.
Sync: wait until all subroutines are done.

Example: Matrix Multiplication (continued)

$\boxed{T(n) = aT(n/b) + O(1)}$

- $C = AB$ or $C_{ij} = A_{ik}B_{kj}$.

MULT (C, A, B, n):

  create tempory matrix $T_{n\times n}$
  if $n = 1$
  then $C_{11} = A_{11} \times B_{11}$
  else
    partition matrix // $O(1)$ *time*
    spawn Mult($C_{11}, A_{11}, B_{11}, n/2$)
    spawn Mult($C_{12}, A_{11}, B_{12}, n/2$)
    spawn Mult($C_{21}, A_{21}, B_{11}, n/2$)
    spawn Mult($C_{22}, A_{21}, B_{12}, n/2$)
    spawn Mult($T_{11}, A_{12}, B_{21}, n/2$)
    spawn Mult($T_{12}, A_{12}, B_{22}, n/2$)
    spawn Mult($T_{21}, A_{22}, B_{21}, n/2$)
    spawn Mult($T_{22}, A_{22}, B_{22}, n/2$)
    sync
    ADD(C, T, n)

Q: how much are a & b for matrix multiplication in serial & parallel computations?

$$A_1(n) = 4A_1\left(\frac{n}{2}\right) + O(1)$$

$$M_1(n) = 8M_1\left(\frac{n}{2}\right) + A_1(n)$$

$$A_\infty(n) = A_\infty\left(\frac{n}{2}\right) + O(1)$$

$$M_\infty(n) = M_\infty\left(\frac{n}{2}\right) + A_\infty(n)$$

ADD(D, A, B, n)
  if n=1
  then $D_{11} = A_{11} + B_{11}$
  else
    partition matrix
    spawn ADD($D_{11}, A_{11}, B_{11}, n/2$)
    spawn ADD($D_{12}, A_{12}, B_{12}, n/2$)
    spawn ADD($D_{21}, A_{21}, B_{21}, n/2$)
    spawn ADD($D_{22}, A_{22}, B_{22}, n/2$)
    sync

Example: Matrix Multiplication (continued)

$$
\begin{aligned}
T(n) &= aT(n/b) + O(1) && (1)\\
&= a(aT(n/b^2) + O(1)) + O(1) && (2)\\
&= a^2T(n/b^2) + O(1) + O(1) && (3)\\
&\vdots && (4)\\
&= a^kT(n/b^k) + \overbrace{O(1) + O(1) + \ldots + O(1)}^{k} && (5)
\end{aligned}
$$

$T(1) = 1$

If $b^k = n$, i.e., $k = \log_b n$, then we have

$$
\begin{aligned}
T(n) &= O\left(a^{\log_b n} + \log_b n\right)\\
&= O\left(n^{\log_b a} + \log_b n\right)
\end{aligned}
$$

$$
\begin{aligned}
&(\log_b a)(\log_b n) = (\log_b n)(\log_b a)\\
&\Rightarrow b^{(\log_b a)(\log_b n)} = b^{(\log_b n)(\log_b a)}\\
&\Rightarrow \left(b^{\log_b a}\right)^{\log_b n} = \left(b^{\log_b n}\right)^{\log_b a}\\
&\Rightarrow a^{\log_b n} = n^{\log_b a}
\end{aligned}
$$

Recurrence Relation (Master Theorem)

- More generally speaking, we have the master theorem:
- for $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ and $f(n) = n^c$

$$T(n) = \begin{cases} \mathrm{O}\left(n^{\log_b a}\right) & c < \log_b a \\ \mathrm{O}(n^c) & c > \log_b a \end{cases}$$

$A_1(n)$=4 $A_1\left(\frac{n}{2}\right) + O(1) \;\; = \mathrm{O}(n^2)$

$M_1(n) = 8M_1\left(\frac{n}{2}\right) + A_1(n) \;\; = \mathrm{O}(n^3)$

https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)

Recurrence Relation (Master Theorem)

$$T(n) = aT(n/b) + \theta(n^k \log^p n)$$

where n = size of the problem

a = number of subproblems in the recursion and a >= 1

n/b = size of each subproblem

b > 1, k >= 0 and p is a real number.

Then,

1. if $a > b^k$, then $T(n) = \theta(n^{\log_b a})$

2. if $a = b^k$, then
   (a) if $p > -1$, then $T(n) = \theta(n^{\log_b a} \log^{p+1} n)$
   (b) if $p = -1$, then $T(n) = \theta(n^{\log_b a} \log\log n)$
   (c) if $p < -1$, then $T(n) = \theta(n^{\log_b a})$

3. if $a < b^k$, then
   (a) if $p >= 0$, then $T(n) = \theta(n^k \log^p n)$
   (b) if $p < 0$, then $T(n) = \theta(n^k)$

for $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ and $f(n) = n^c$

$$T(n) = \begin{cases} O\left(n^{\log_b a}\right) & c < \log_b a \\ O(n^c) & c > \log_b a \end{cases}$$

By taking $k = c, p = 0$, these two theorems are consistent with each other.

Recurrence Relation (Master Theorem) Application

$$T(n) = aT(n/b) + \theta(n^k \log^p n)$$

where n = size of the problem

a = number of subproblems in the recursion and a >= 1

n/b = size of each subproblem

b > 1, k >= 0 and p is a real number.

Then,

1. if $a > b^k$, then $T(n) = \theta(n^{\log_b a})$

2. if $a = b^k$, then
   (a) if $p > -1$, then $T(n) = \theta(n^{\log_b a} \log^{p+1} n)$
   (b) if $p = -1$, then $T(n) = \theta(n^{\log_b a} \log\log n)$
   (c) if $p < -1$, then $T(n) = \theta(n^{\log_b a})$

3. if $a < b^k$, then
   (a) if $p >= 0$, then $T(n) = \theta(n^k \log^p n)$
   (b) if $p < 0$, then $T(n) = \theta(n^k)$

$$A_1(n) = 4\, A_1\left(\frac{n}{2}\right) + O(1) = O(n^2)$$

$$M_1(n) = 8M_1\left(\frac{n}{2}\right) + A_1(n) = O(n^3)$$

$$A_\infty(n) = A_\infty\left(\frac{n}{2}\right) + O(1) = O(\log n)$$

$$M_\infty(n) = M_\infty\left(\frac{n}{2}\right) + A_\infty(n) = O((\log n)^2)$$

Parallelism

$$\frac{A_1(n)}{A_\infty(n)} = \frac{O(n^2)}{O(\log n)}$$

$$\frac{M_1(n)}{M_\infty(n)} = \frac{O(n^3)}{O((\log n)^2)}$$

It is highly possible to obtain super linear speedup
The number of processors needed also grows quickly with n

Parallel Computing in Python

- Threading vs Multiprocessing
- Hands-on programming:
  - Simple Examples
  - Threading
  - Multiprocessing
  - Lock
  - Queue
  - Pool

Threading vs Multiprocessing

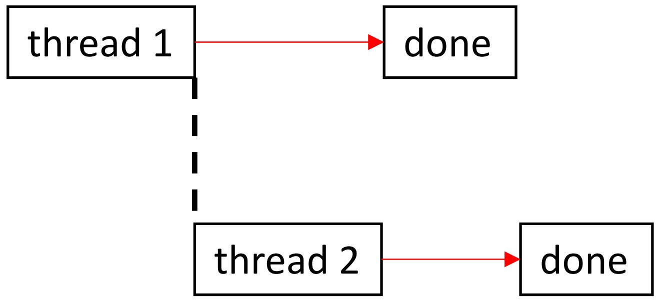
- Threading:
  - A new thread is spawned within the existing process;
  - Starting a thread is faster than starting a process;
  - Memory is shared between all threads;
  - Mutexes are often necessary to control access to shared data;
  - One GIL (Global Interpreter Lock) for all threads.
- Multiprocessing:
  - A new process is started independent from the first process;
  - Starting a process is slower than starting a thread;
  - Memory is not shared between processes;
  - IPC (inter-process communication) is more complicated.
  - One GIL for each process.

A *Mutex* or a *lock* is a synchronization mechanism for enforcing limits on access to a resource in an environment where there are many threads of execution.

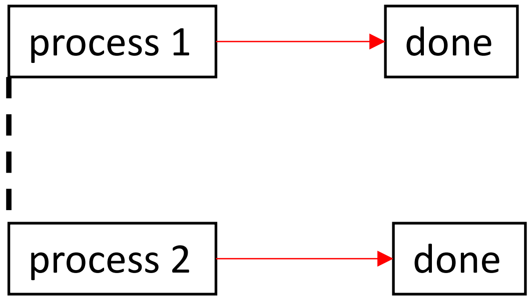
Global Interpreter Lock: a mutex (or a lock) that allows only one thread to hold control of the Python interpreter. This means that the GIL allows only one thread to execute at a time even in a multi-threaded architecture. It is needed because CPython's (reference implementation of Python) memory management is not thread-safe.

Threading vs Multiprocessing (continued)

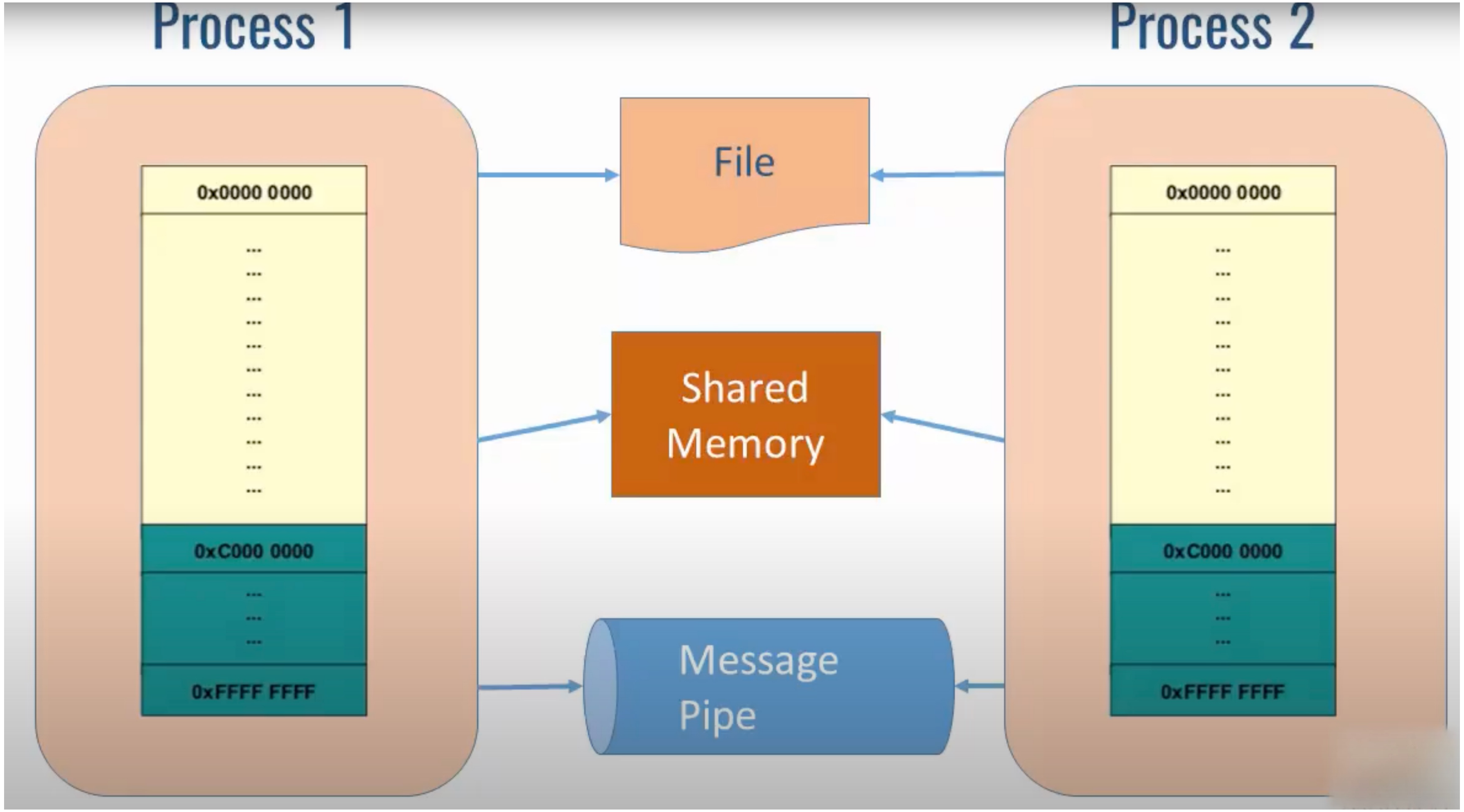
- Threading is good for I/O-bound tasks
  - Despite the GIL it is useful for I/O-bound tasks when your program has to talk to slow devices, like a hard drive or a network connection. With threading the program can use the time waiting for these devices and intelligently do other tasks in the meantime.
  - Example: Download website information from multiple sites. Use a thread for each site.



- Multiprocessing is good for CPU-bound tasks.
  - It is useful for CPU-bound tasks that have to do a lot of CPU operations for a large amount of data and require a lot of computation time. With multiprocessing you can split the data into equal parts and do parallel computing on different CPUs.
  - Example: Calculate the square numbers for all numbers from 1 to 1000000. Divide the numbers into equal sized parts and use a process for each subset.



Data Sharing in Multiprocessing



https://www.youtube.com/watch?v=sp7EhjLkFY4

Hands-on Parallel Coding with Python

- Your own computer installed with Python.
  - Python 3.X (3.8 & 3.9 confirmed) is recommended.
  - For Windows, Anaconda is recommended.
  - For Mac OS, python 3.8 or 3.9 can be installed using package manager, e.g., anaconda, mac ports, home brew, etc.
- Online python platform:
  - google: http://colab.research.google.com/
  - cocalc: https://cocalc.com/app?anonymous=jupyter
  - binder: https://mybinder.org/v2/gh/ipython/ipython-in-depth/master?filepath=binder/Index.ipynb
- Relevant materials: https://docs.python.org/3/library/multiprocessing.html

Hands-on Parallel Coding with Python

- check_cpu_number.py
- If you are using python 2.X, you may not be able to use this command.
- Q: is the result the number of physical cores or logical cores?
  - Logical.
- Q: what is the alternative way of calling method os.cpu_count()?
  - from os import cpu_count
  - print(cpu_count())

```
import os
print(os.cpu_count())
```

Simple Examples - Hello World

__name__ is one such special variable. If the source file is executed as the main program, the interpreter sets the __name__ variable to have a value "__main__". If this file is being imported from another module, __name__ will be set to the module's name.

- HelloWorld.py
  - p.start(): start the process's activity; can be called at most once.
  - p.join(): block until the process terminates.
  - p.join([*timeout*]): block at most *timeout seconds* if it is positive.
  - "if __name__ == '__main__'" is to safeguard your code from unwanted executions.
  - try run foo.py and import foo in python.
  - In target, try using do_something().
- If you are using python 2.7, try
  - print("Hello World from",os.getuid())

```
import multiprocessing
from multiprocessing import Process
import os

def do_something():
  print(f"Hello World from {os.getpid()}")

if __name__ == '__main__':
  p1 = multiprocessing.Process(target=do_something)
  p2 = multiprocessing.Process(target=do_something)

  p1.start()
  p2.start()

  p1.join()
  p2.join()
```

Simple Examples - Hello World (Cont’d)

- Possible outcome:

```
Hello World from 609
Hello World from 610
```

- Q: in the above example, is it possible to see an inversed order of 609 and 610 on screen?
  - yes.
- Q: is there a difference between single and double quotes?
  - You can use both in print().
- Q: How to include both single and double quotes in the same string?
  - print("'a'=\"a\"") or print('"a"=\'a\’')

```
import multiprocessing
from multiprocessing import Process
import os

def do_something():
  print(f”Hello World from {os.getpid()}”)

if __name__ == '__main__’:
  p1 = multiprocessing.Process(target=do_something)
  p2 = multiprocessing.Process(target=do_something)

  p1.start()
  p2.start()

  p1.join()
  p2.join()
```

Simple Examples - Hello World (Cont’d)

- HelloWorld2.py & HelloWorld3.py
- Depending on your platform, you may expect two possible outcomes

Possible Outcome 1:	Possible Outcome 2:
Hello World from 609 Hello World from 610	Hello Hello World from 609 World from 610

- Q: how to enforce a sequential execution of multiple processes?
  - exchange p2.start() and p1.join(); see HelloWorld4.py as an example.

```
import multiprocessing
from multiprocessing import Process
import os

def do_something():
  print('Hello')
  print(f"World from {os.getpid()}")

if __name__ == '__main__':
  p1 = multiprocessing.Process(target=do_something)
  p2 = multiprocessing.Process(target=do_something)

  p1.start()
  p2.start()

  p1.join()
  p2.join()
```

Simple Examples (Cont'd)

- serial.py
- sleep1.py
- Q: What's the outcome and why?
  - Finished in 0.0 second(s).
  - Unlike serial function, processes are not executed if start() is not called.

round(*number, digits*) returns a floating point number that is a rounded version of the specified number, with the specified number of decimals.
The default number of decimals is 0, meaning that the function will return the nearest integer.

```
import time
import multiprocessing

start = time.perf_counter()

def func():
  time.sleep(1)
  print('slept 1 second...')

p1 = multiprocessing.Process(target=func, args=())
p2 = multiprocessing.Process(target=func, args=())

finish = time.perf_counter()

print(f'Finished in {round(finish-start,2)} second(s)')
```

Simple Examples (Cont’d)

- sleep2.py
- Q: What’s the outcome and why?
  - Output starts with “Finished in 0.1 second(s)”
  - Main process ends before child process.
- sleep3.py
  - call join() to wait for child processes to end.
- Q: why the printed lines are always in the same order?
  - longer slept processes end later.

```
import time
import multiprocessing

start = time.perf_counter()

def func(t):
  time.sleep(t)
  print(f'slept {t} second...')

if __name__ == '__main__':
  processes = []

  for x in range(1,10):
    p = multiprocessing.Process(target=func, args=(x/10,))
    processes.append(p)
    p.start()

  finish = time.perf_counter()

  print(f'Finished in {round(finish-start,2)} second(s)')
```

Threading

- multitasking.py vs thread1.py
- Q: Which one is faster?
  - thread1.py
- thread2.py
  - A race condition occurs when two or more threads can access shared data and they try to change it at the same time.
- Q: what is an easy fix to this race condition and what is the price?
  - exchange t2.start() and t1.join(); the price is that they are no longer parallel.

```
from threading import Thread
import time

database_value = 0

def increase():
    global database_value
    local_copy = database_value
    local_copy += 1
    time.sleep(0.1)
    database_value = local_copy

if __name__ == "__main__":
    print('Start value: ', database_value)

    t1 = Thread(target=increase)
    t2 = Thread(target=increase)

    t1.start()
    t2.start()

    t1.join()
    t2.join()

    print('End value:', database_value)
    print('end main')
```

Lock in Threading

- thread3.py & thread4.py
  - A lock (also known as mutex) is a synchronization mechanism for enforcing limits on access to a resource in an environment where there are many threads of execution.
  - lock.acquire(), lock.release()
  - use lock as a context “with lock:” (see thread4.py).

```
from threading import Thread, Lock
import time

database_value = 0

def increase(lock):
    global database_value
    lock.acquire()
    local_copy = database_value
    local_copy += 1
    time.sleep(0.1)
    database_value = local_copy
    lock.release()

if __name__ == "__main__":
    lock = Lock()

    print('Start value: ', database_value)
    t1 = Thread(target=increase, args=(lock,))
    t2 = Thread(target=increase, args=(lock,))

    t1.start()
    t2.start()

    t1.join()
    t2.join()

    print('End value:', database_value)
    print('end main')
```

Threading vs Multiprocessing

```python
from threading import Thread


def square_numbers():
  for i in range(1000):
    result = i * i

if __name__ == "__main__":
  threads = []
  num_threads = 10

# create threads and asign a function for each thread
  for i in range(num_threads):
    thread = Thread(target=square_numbers)
    threads.append(thread)

# start all threads for thread in threads:
    thread.start()

# wait for all threads to finish
# block the main thread until these threads are finished for thread in threads:
  thread.join()
```

```python
from multiprocessing import Process
import os

def square_numbers():
  for i in range(1000):
    result = i * i

if __name__ == "__main__":
  processes = []
  num_processes = os.cpu_count()

# create processes and asign a function for each process
  for i in range(num_processes):
    process = Process(target=square_numbers)
    processes.append(process)

# start all processes for process in processes:
    process.start()

# wait for all processes to finish
# block the main thread until these processes are finished for process in processes:
    process.join()
```

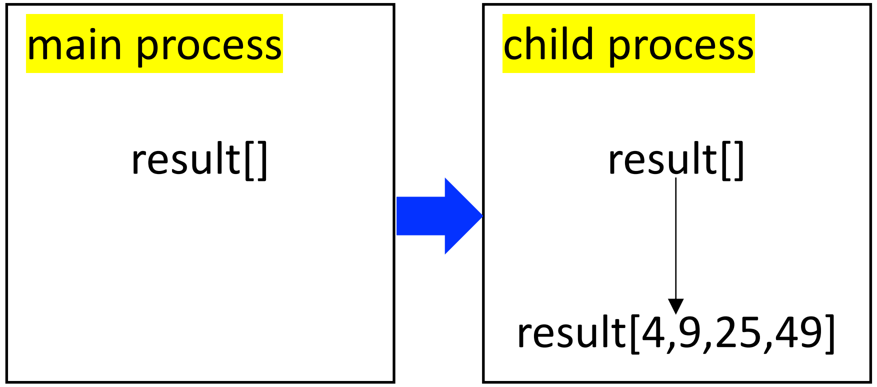
Threading vs Multiprocessing

- global1.py
  - Global variables are shared by different threads.
- Q: what are the different ways of printing a mixture of string and numbers?
  - print(f’abc {x}’) or print(‘abc’+str(x)) or print(‘abc’,x).
- Q: how to clean/clear a variable result in python?
  - del result

```
import threading

result=[]

def cal_square(numbers):
  global result
  for n in numbers:
    print('square '+str(n*n))
    result.append(n*n)

if __name__ == "__main__":
  array = [2,3,5,7]
  p1 = threading.Thread(target=cal_square, args=(array,))

  p1.start()
  p1.join()

  print('result '+str(result))
  print('done')
```

Threading vs Multiprocessing (Cont'd)

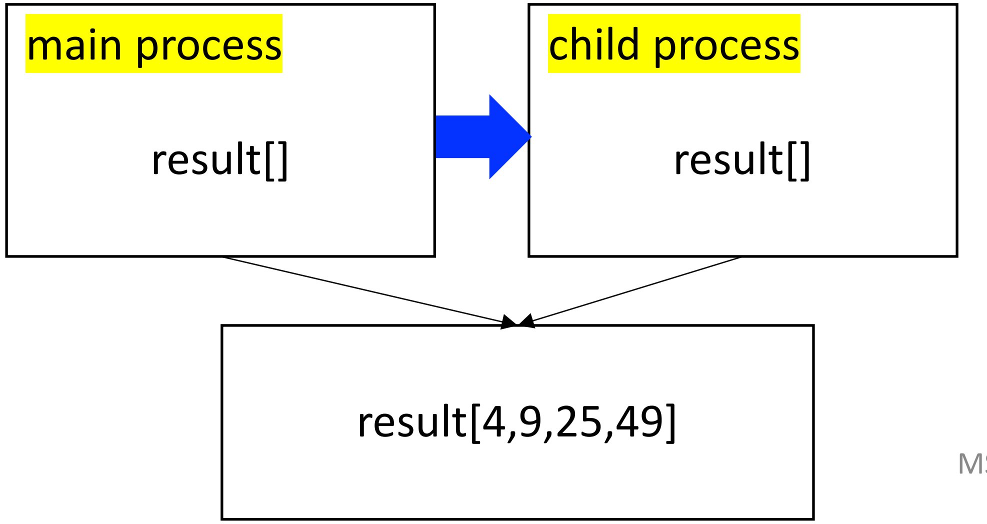
- global2.py
  - Global variables are not shared by different processes.
  - Every process has its own address space. Interprocess communication techniques are needed to share data between processes.
- Q: what if result is nonempty, e.g., [1,2]?
  - Child process inherits variable from main process. See global3.py.




```
import multiprocessing

result=[]

def cal_square(numbers):
  global result
  for n in numbers:
    print('square '+str(n*n))
    result.append(n*n)

if __name__ == "__main__":
  array = [2,3,5,7]
  p1 = multiprocessing.Process(target=cal_square, args=(array,))

  p1.start()
  p1.join()

  print('result '+str(result))
  print('done')
```

Sharing Data Between Processes - Array

- array1.py
- Q: how to make an integer array s?
  - s = Array(‘i’,[1,2,3])
- Q: how to make a string array s?
  - s = Array(‘c’,b’abc’)
- Q: how to change a byte string s to an ascii string?
  - import codecs
  - codecs.decode(s.value)




```
from multiprocessing import Process, Array

def cal_square(numbers,result):
  for id,n in enumerate(numbers):
    print('square '+str(n*n))
    result[id] = n*n

if __name__ == "__main__":
  array = [2,3,5,7]
  result = Array('d',array)
  p1 = Process(target=cal_square, args=(array,result,))

  p1.start()
  p1.join()

  print('result ',result[:])
  print('done')
```

Value and Array

value1.py

```
from multiprocessing import Process, Value

def cal_square(v, v2):
  c = v.value
  v2.value = c*c
  v.value = 3.14

if __name__ == "__main__":
  v = Value('d',0)
  v2= Value('d',1)
  print(f"after: {v.value} {v2.value}")

  p = Process(target=cal_square, args=(v,v2,))

  p.start()
  p.join()

  print(f"after: {v.value} {v2.value}")
```

array1.py

```
from multiprocessing import Process, Array

def cal_square(numbers,result):
  for id,n in enumerate(numbers):
    print('square '+str(n*n))
    result[id] = n*n

if __name__ == "__main__":
  array = [2,3,5,7]
  result = Array('d',array)
  p1 = Process(target=cal_square, args=(array,result,))

  p1.start()
  p1.join()

  print('result ',result[:])
  print('done')
```


https://www.youtube.com/watch?v=uWbSc84he2Q&t=16s

Sharing Data Between Processes - Value

- value1.py
- value2.py
  - Race condition.
- Q: what are the ways to access an Array variable a?
  - a[:] or a.value.
- Q: how to avoid race condition in multiprocessing?
  - use Lock, see the next slide.

```
from multiprocessing import Process, Value
import time

def add_100(number):
  for i in range(100):
    time.sleep(0.01)
    number.value += 1

if __name__ == "__main__":
  shared_num = Value('i', 0)
  print(f"Number at beginning is {shared_num.value}")

  p1 = Process(target=add_100, args=(shared_num,))
  p2 = Process(target=add_100, args=(shared_num,))

  p1.start()
  p2.start()

  p1.join()
  p2.join()

  print(f"Number at end is {shared_num.value}")
```

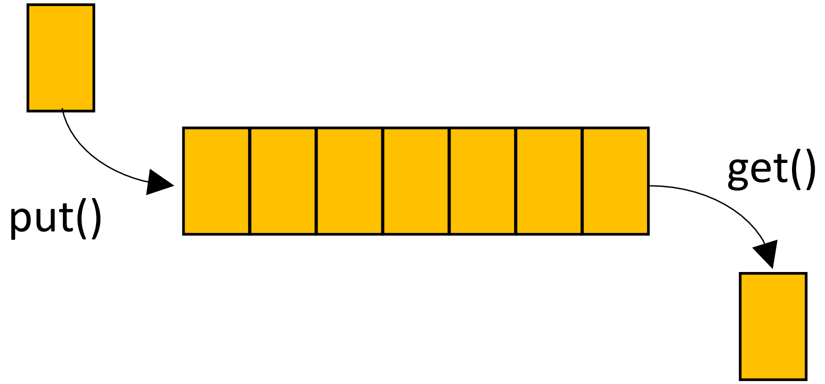
Sharing Data Between Processes – Value (Cont'd)

- value3.py
  - use Lock to deal with race condition.
- Q: is this Lock the same as that in threading?
  - although they share the same name, they cannot be cross-used.
- Q: if both threading and multiprocessing are involved, how to use both types of locks?
  - specify them as multiprocessing.Lock() and threading.Lock().

```
from multiprocessing import Process, Value, Lock
import time

def add_100(number, lock):
  for i in range(100):
    time.sleep(0.01)
    lock.acquire()
    number.value += 1
    lock.release()

def sub_100(number, lock):
  for i in range(100):
    time.sleep(0.01)
    with lock:
      number.value -= 1

if __name__ == "__main__":
  lock = Lock()
  shared_num = Value('i', 0)
  print(f"Number at beginning is {shared_num.value}")

  p1 = Process(target=add_100, args=(shared_num,lock,))
  p2 = Process(target=sub_100, args=(shared_num,lock,))

  p1.start()
  p2.start()

  p1.join()
  p2.join()

  print(f"Number at end is {shared_num.value}")
```

Queue

- Queues can be used for thread-safe/process-safe data exchanges and data processing both in a multithreaded and a multiprocessing environment.
- A queue is a linear data structure that follows the First In First Out (FIFO) principle. A good example is a queue of customers that are waiting in line, where the customer that came first is served first.



https://www.youtube.com/watch?v=sp7EhjLkFY4

Queue (Cont'd)

Caution: avoid naming any of your script files using a module name.

• Multiprocessing Queue	• Queue Module
import multiprocessing q = multiprocessing.Queue()	import queue q = queue.Queue()
• Lives in shared memory • used to share data between processes.	• Lives in in-process memory • used to share data between threads.

https://www.youtube.com/watch?v=sp7EhjLkFY4

Using Queue

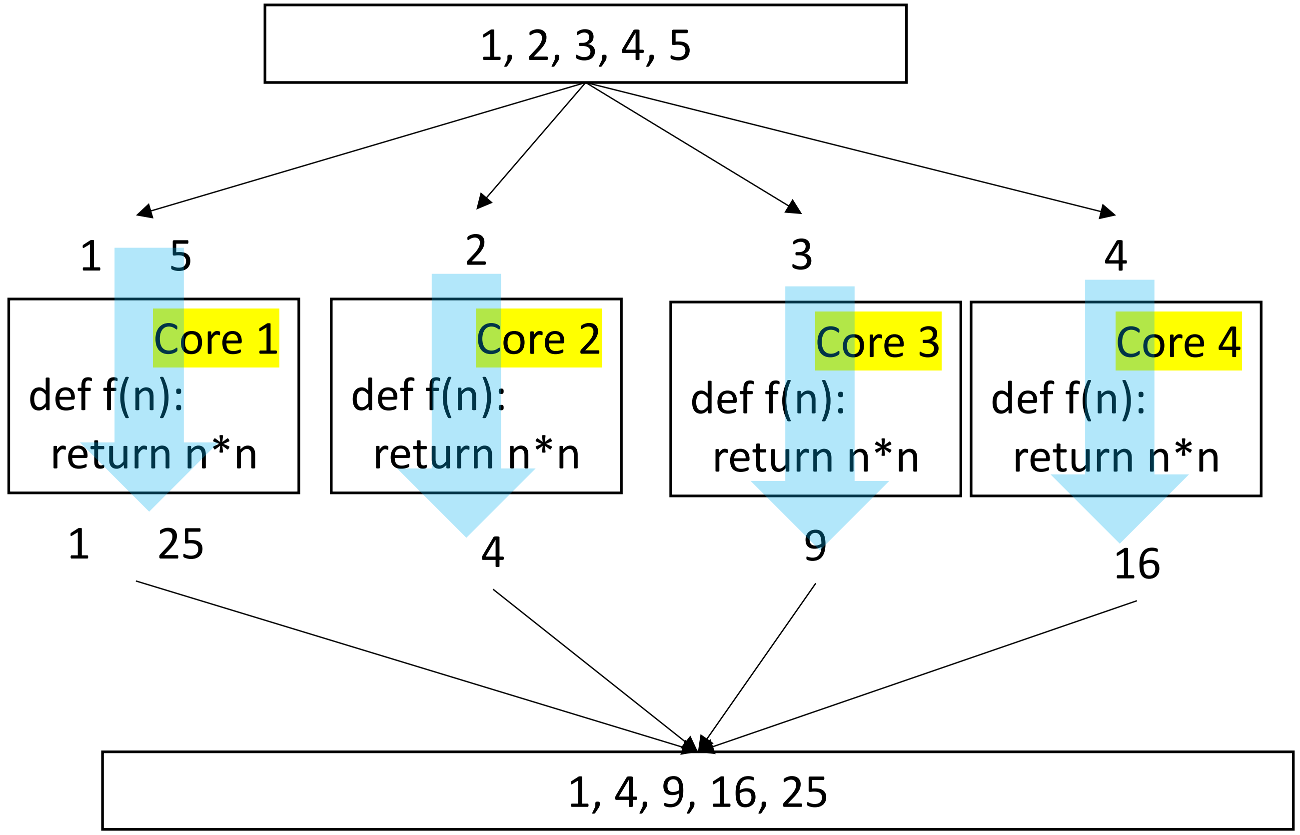
- q1.py
- Q: how to use Queue in Threading module for the above script?
    - Load queue and Thread; see q2.py.
- Caution: don’t mix the usage of the two types of Queue.

```
from multiprocessing import Process, Queue

def cal_square(numbers, q):
  for n in numbers:
    q.put(n*n)

#  print('inside process ',str(result))

if __name__ == "__main__":
  numbers = [2,3,5,7]
  q = Queue()
  p = Process(target=cal_square, args=(numbers,q,))

  p.start()
  p.join()

  while q.empty() is False:
    print(q.get())
```

https://www.youtube.com/watch?v=_1ZwkCY9wxk

Pool (Map and Reduce)

- Pool object offers a convenient means of parallelizing the execution of a function across multiple input values, distributing the input data across processes.



https://www.youtube.com/watch?v=_1ZwkCY9wxk

Using Map and Pool

- map1.py
- map2.py
  - close(): Prevents any more tasks from being submitted to the pool. Once all the tasks have been completed the worker processes will exit.
- p = Pool(processes=3)

```
from multiprocessing import Pool
import time

def f(n):
  sum = 0
  for x in range(1000):
    sum += x*x
  return sum

if __name__ == '__main__':
  t1 = time.time()
  array = [1,2,3,4,5]
  p = Pool(processes=8)
  result = p.map(f,range(10000))
  p.close()
  p.join()

  print(f"Pool took {time.time()-t1}")

  t2 = time.time()
  result=[]
  for x in range(10000):
    result.append(f(x))

  print(f"Serial processing took {time.time()-t1}")
```

https://www.youtube.com/watch?v=_1ZwkCY9wxk

Summary

- Recap of The Last Parallel Computing Lecture.
- Recurrence Relation.
- Python for Parallel Computing Examples.

Next Lecture

- Remaining examples.
- MPI for python.
- GPGPU computing.