Multiprocessing in Python

Tucson Python Meetup
July 7th 2020

Ian Bertolacci
ian-bertolacci.github.io
(Available for hire!)
Code and materials available at
github.com/ian-bertolacci/Multiprocessing-in-Python-Tucson-Python-Meetup

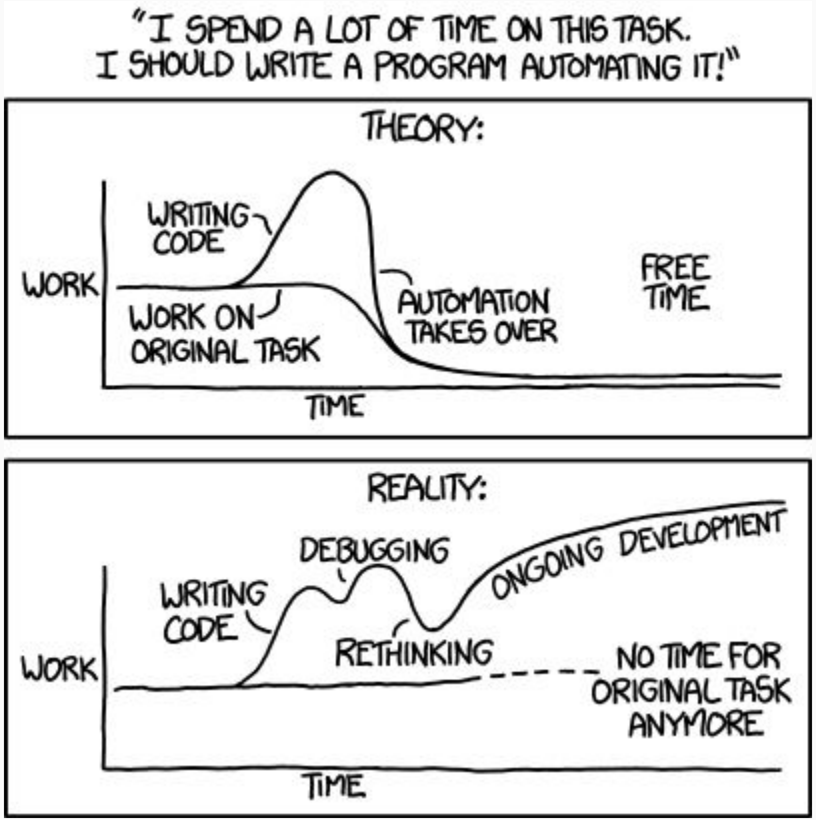
Python Performance Disclaimer

- Python is not a high performance programming language.
  - Write in a faster language if you need extremely fast applications.
- Use fast libraries!
  - Have fast implementations (though unlikely to be parallel).
  - NumPy, SciPy, Pandas, etc

Parallel Programming Disclaimer

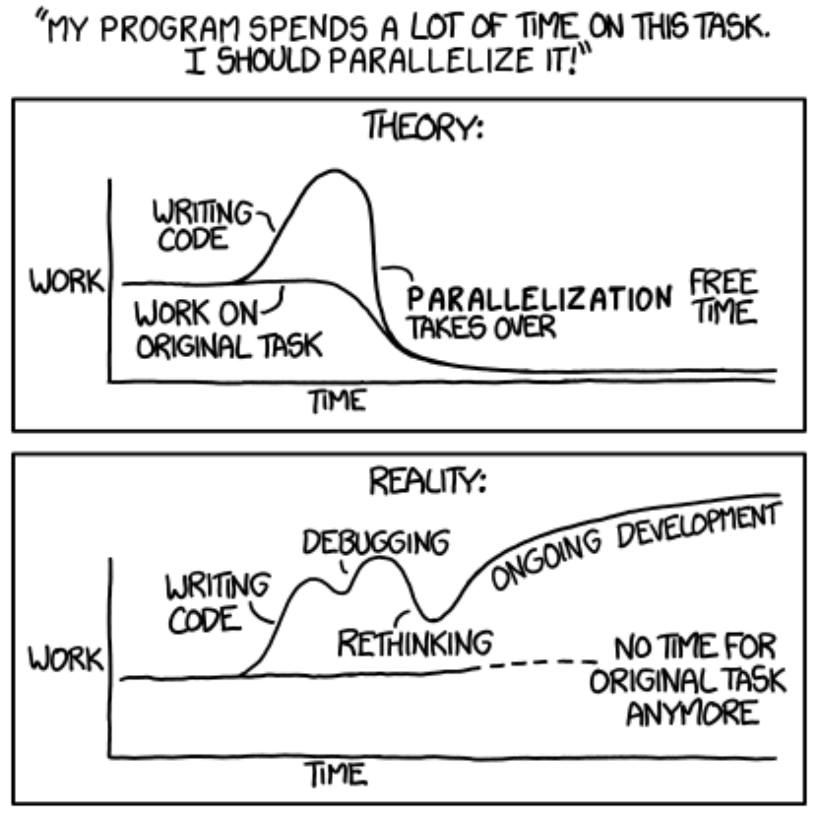
- Parallelism is not always the answer
  - Amdahl's law: speedup <= total_time / (time_in_serial + (time_in_parallel / processes))
  - If x% of a program is parallelizable, the ***BEST CASE*** is an x% reduction in runtime.
    - Period.
    - You will get less than that.
- More Parallelism != Faster Program
  - Overhead of spawning parallel tasks.
  - Resource contention.
- Profile! Experiment!
  - Is program slow because of amount of work? Or because implementation is algorithmically slow?

Parallel Programming Disclaimer



https://xkcd.com/1319/

Parallel Programming Disclaimer



Based on https://xkcd.com/1319/

Breaking Down a Problem into Parallelism

- What is "work"?
  - Multiple <u>different</u> tasks?
    - Implies *task* <u>parallelism</u>
  - Multiple <u>identical</u> tasks on <u>different</u> data?
    - Implies <u>data parallelism</u>
- Dependencies
  - What tasks depend on other each other?
    - Implies <u>ordering</u> between tasks
  - What tasks share <u>objects</u>?
    - Implies <u>communication</u> between tasks.
  - What tasks are modifying shared objects?
    - Implies <u>synchronization</u> between tasks

Example: Adding Two Arrays

```
added_AB = [ a + b for a, b in zip(A, B) ]
```

- What kind of work is happening here?
    - Same task, different data.
- Are there dependencies between tasks?
    - No
    - All tasks operate independently.
- Do tasks share data?
    - No
    - All tasks have their own data.

Example: Modifying, Sending, and Writing Data

```
fooized_data = fooize( data )
send_data( fooized_data, server_url )
json_write( fooized_data, "fooized.json" )
```

- What kind of work is happening here?
  - Different tasks
- Are there dependencies between tasks?
  - Yes
  - The send_data communication and json_write depend on the statement creating fooized_data
- Do tasks share data?
  - Yes
  - The server communication and file output share fooized_data
- Are tasks are modifying shared objects?
  - No
  - Because I made this example up and assume neither send_data or json_write modify the object

Python Multiprocessing

- Standard Python library with API for parallel python programming.
  - Same API as Python Threading library (which is *not* parallel).
- Excellent docs: docs.python.org/3/library/multiprocessing.html
- Provides:
  - Parallelism via Processes and Process pools
  - Synchronization via Locks, Events, Barriers, etc...
  - Inter-process communication via Pipes, Queues, Shared Memory

Example: Data Parallelism using Pool.map

Original:

```
added_AB = [ a + b for a, b in zip(A, B) ]
```

Parallel:

```
with Pool( processes ) as pool:
  added_AB = pool.starmap( operator.add, zip(A, B)  )
```

Example: Parallel Reduction using Pool.map

Original:

```
sum_data = sum( data )
```

Parallel:

```
with Pool( processes ) as pool:
  chunk_size = int( ceil( len(data)/processes ) )
  chunked_work = [
    data[ i : i + chunk_size ]
    for i in range( 0, len(data), chunk_size )
  ]
  partial_sum_data = pool.imap( sum, chunked_work, chunksize=1 )
  sum_data = sum( partial_sum_data )
```

Some Notes on Pool.map

- Careful about order!
  - if order of matters, make sure you're using the correct map function
  - Especially true for reduction operations.
- A number of different map functions for different needs.
  - Asynchronous map
  - Starmap
  - Unordered map
  - Iterable map
  - Combinations of all
-

Example: Task Parallelism with Pool.apply

Original:

```
fooized_data = fooize( data )
send_data( fooized_data, server_url )
json_write( fooized_data, "fooized.json" )
```

Parallel:

```
fooized_data = fooize( data )
with Pool( processes ) as pool:
  send_async = pool.apply_async( send_data, (fooized_data, server_url) )
  write_async = pool.apply_async( json_write, (fooized_data, "fooized.json") )

 send_result = send_async.get()
 write_result = write_async.get()
```

Example: Task Parallelism with Process

Original:

```
fooized_data = fooize( data )
send_data( fooized_data, server_url )
json_write( fooized_data, "fooized.json" )
```

Parallel:

```
fooized_data = fooize( data )
processes = [
  Process( target=send_data, args=(fooized_data, server_url) ),
  Process( target=json_write, args=(fooized_data, "fooized.json") )
]
for process in processes: process.start()
for process in processes: process.join()
```

Process Communication and Shared Data

- Different tasks may be independent until they need to share data.
- Direct communication:
  - Pipes: connects two processes, can be bi-directional (Not covered here)
  - Queues: many-to-many FIFO queue
- Shared Data/Memory
  - Values: wraps a C type variable with a lock.
  - Arrays: wraps an array of C type values with a lock.

Example: Queue

```
def make_work( queue, N ):
  for work in range(N):
    queue.put( work )
  queue.put("dead")
```

```
def do_work( queue ):
  while True:
    work = queue.get()
    if work == "dead":
      break
    print( f"got {work}" )
```

```
# main
items = 100
work_queue = Queue()
Process( target = make_work, args =(work_queue, items) ).start()
Process( target = do_work,     args =(work_queue, )         ).start()
```

Example: Value

```
def writer( counter  ):
  with counter.get_lock():
    while True:
      counter.value += 1
```

```
def reader( counter ):
  while True:
    print( counter.value )
```

```
# main
shared_value = Value( "i", 0 )
Process( target = make_work, args =(shared_value, ) ).start()
Process( target = do_work,     args =(shared_value, ) ).start()
```

But What About Non-ctype Shared Memory?

Synchronization

- Lock
  - Mutual exclusion (of code sections and/or data)
  - A process acquires the lock <u>then</u> releases the lock
- Event:
  - Many-on-one ordering
  - One-or-more processes wait on the event, a single process triggers it, unblocking all
- Condition:
  - Conditional mutual exclusion (think lock + event)
  - Process waits, another process unblocks
- Barriers:
  - Many-on-many ordering
  - All processes* must reach and wait at the barrier before all can proceed
- Semaphore/BoundedSemaphore:
  - Capacity exclusion (think lock that can be opened by at most N keys simultaneously)
  - Only N processes may acquire the semaphore simultaneously

Example: Lock

```
def worker( id, array ):
  while True:
    array[id] += 1
```

```
def printer( id, lock, array ):
  while True:
    sleep( 1 )
    with lock:
      for v in array:
        print( f"({id}) {v}" )
```

```
# main
array = Array( "i", [0] * workers )
lock = Lock()
[ Process( target = worker, args = (id, array) ).start() for id in … ]
[ Process( target = printer, args = (id, array, lock ) ).start() for id in … ]
```

Deadlocking

- Processes can acquire multiple locks.
- Deadlock occurs when two processes hold locks (a, b) while trying to acquire the other (b, a)
- Neither can progress, essentially halting the program.
- Easy to avoid if process will ever only hold one lock simultaneously.
- Hard to avoid otherwise.

```
def foo( lock_a, lock_b ):
 with lock_a:
  with lock_b:
   … stuff ...
```

```
def bar( lock_a, lock_b ):
 with lock_b:
  with lock_a:
   … stuff ...
```

Example: Event and Barrier

```
def writer( id, array, barrier, event ):
  while True:
    event.wait()
    while event.is_set():
      array[id] += 1
    barrier.wait()
```

```
def safe_reader( array, wbarrier,
rbarrier, event ):
  while True:
    sleep( 1 )
    event.clear()
    wbarrier.wait()
    local_copy = [ v for v in array ]
    rbarrier.wait()
    event.set()
```

```
# main
event = mp.Event()
write_barrier = mp.Barrier( writers +
readers_barrier = mp.Barrier( readers )
event.set()
# … the rest of the program ...
```

Is this safe for multiple readers?

NO!

Parting Thoughts

- Is multiprocessing for everyone? Certainly not.
  - Still hard to use
  - Parallel engineering is weirder than in other languages (shared memory in particular)
- Is it useful? Mmmaybe?
  - Probably best for data parallelism
  - Hard to see where multiprocessing would be the best mechanism for task parallelism
    - Rapidly descends into microservices land
- Hints:
  - Read those docs!!!!
  - Python.threading and Python.multiprocessing have a very common API (intentional)
  - Be ready to kill some processes manually