Parallel computation in Python

André Silva

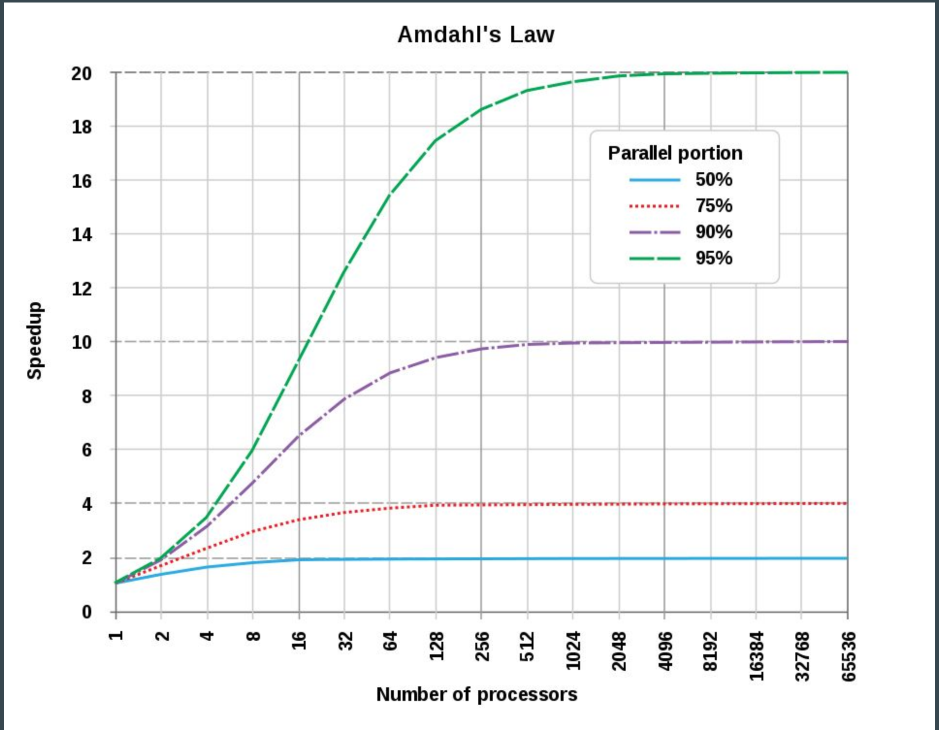
Overview

- Concurrent vs Parallel computation
- Problems with Parallelism
- How are tasks scheduled?
- How does Python tackle this problem?
- The multiprocessing module
- What we will (probably) have in the future

Parallelism vs concurrency

- Concurrency: A condition that exists when at least two threads are making progress.
- Parallelism: A condition that arises when at least two threads are executing simultaneously.

Why should(n't) we go parallel ? - Amdahl's law



- Used in parallel computing to predict the theoretical speedup when using multiple processors;
- Parallel computing with many processors is useful only for highly parallelizable programs.

Fig: Amdahl's law. Taken from: https://en.wikipedia.org/wiki/File:AmdahlsLaw.svg

Problems with Parallelism - I

- Can we even parallelize our problem?
    a. Do we need previous results to go forward?
- Will we run into race conditions?
    a. Does the order of computation change the output?
- Shared Resources, e.g. access to files or data in memory
    a. Locks: block access within the process
    b. Mutex: block access system wide
    c. Semaphores: similar to Mutex, but multiple threads can access (number is capped)

Problems with Parallelism - I

- Load balancing
  a. If we create too many chunks: the overheads of managing and scheduling the chunks will be large.
  b. If we create too few chunks: some cores on the machine will have nothing to do.
- Harder to debug...
- Do we have enough memory ??

Single core behaviour/Concurrent computation

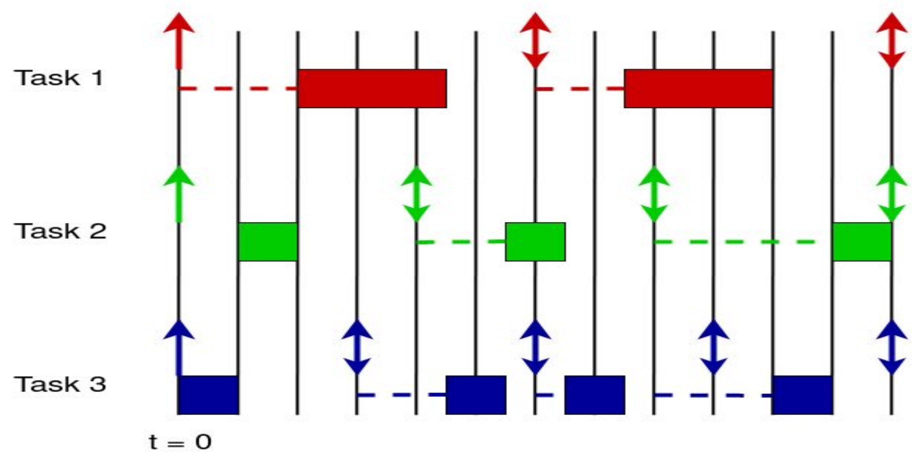


Fig: Task scheduling with an Early Deadline First (EDF) approach, without interrupts.

Single core behaviour/Concurrent computation

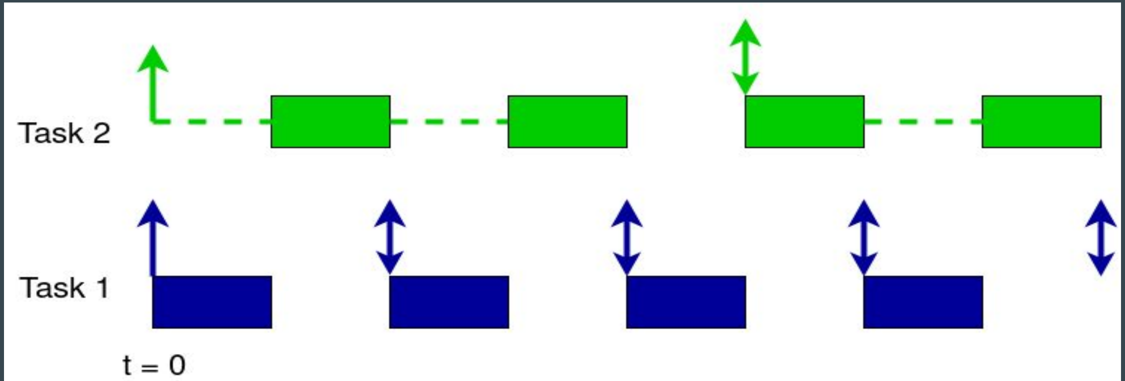


Fig: Task scheduling with an Early Deadline First (EDF) approach, with interrupts.

Single core behaviour/Concurrent computation

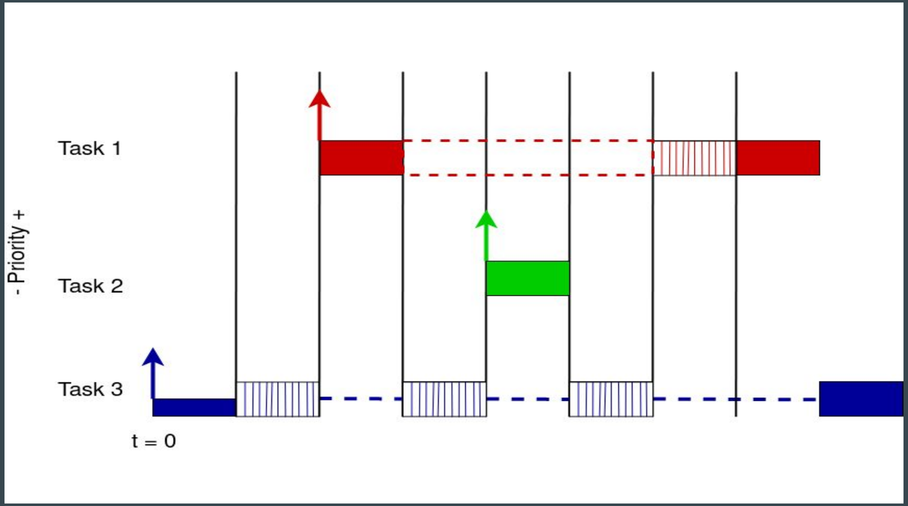


Fig: Task scheduling with interrupts and blocking regions.

Multi-core behaviour/Parallel computation

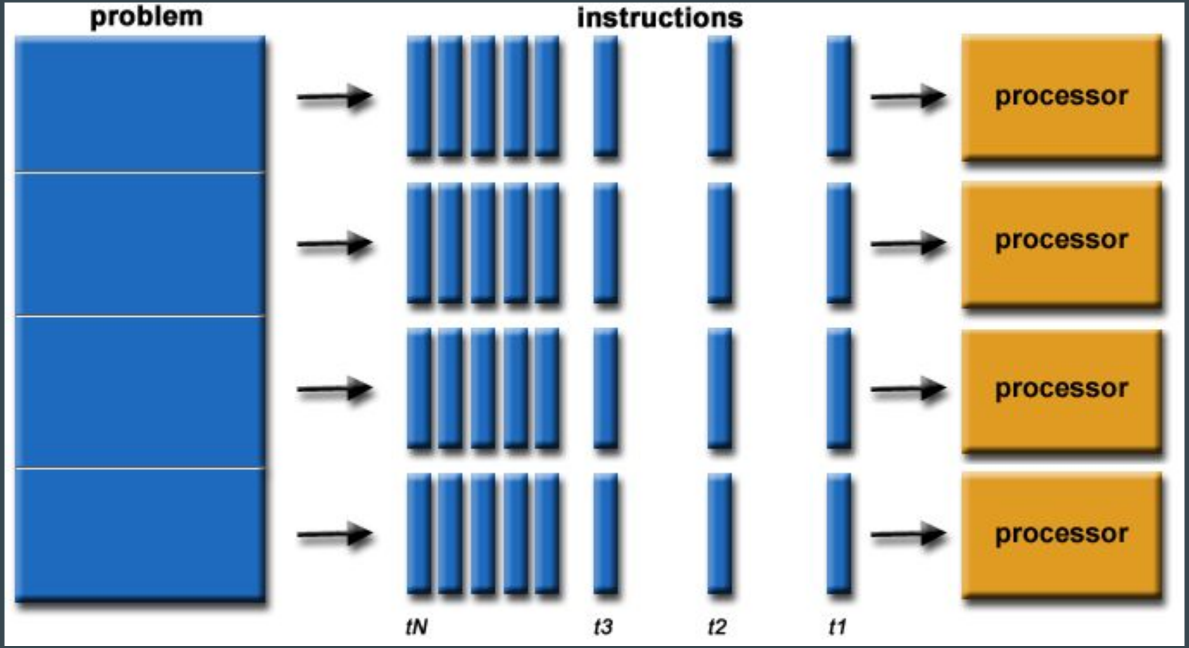


Fig: Schematic of parallel computation, for a general problem.

Taken from: https://computing.llnl.gov/tutorials/parallel_comp/

How does Python achieve parallelism

- Global Interpreter Lock (GIL)
  - async code, multi-threaded code and never have to worry about acquiring locks on any variables
  - avoid having processes crash from deadlocks.
  - Only 1 thread can be executing at any given time.

> Past efforts to create a “free-threaded” interpreter (one which locks shared data at a much finer granularity) have not been successful because performance suffered in the common single-processor case. It is believed that overcoming this performance issue would make the implementation much more complicated and therefore costlier to maintain.

Fig: Problems encountered when removing the GIL, from python documentation pages

The multiprocessing module - Process

```python
from multiprocessing import Process

def f(name):
    print('hello', name)

if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

- Represent activity that is run in a separate process
- Has equivalents of all the methods of threading.Thread

The multiprocessing module - Pool

```
from multiprocessing import Pool

def f(x):
    return x*x

if __name__ == '__main__':
    with Pool(5) as p:
        print(p.map(f, [1, 2, 3]))
```

- Pool of worker processes to which jobs can be submitted;
- The pool will distribute those tasks to the worker processes and collects the return values;

The multiprocessing module - exchanging data I

- Pipe - only two endpoints;
  - Better performance;

```python
from multiprocessing import Process, Pipe

def f(conn):
    conn.send([42, None, 'hello'])
    conn.close()

if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(child_conn,))
    p.start()
    print(parent_conn.recv())   # prints "[42, None, 'hello']"
    p.join()
```

The multiprocessing module - exchanging data II

- Queues - can have multiple endpoints

```
from multiprocessing import Process, Queue

def f(q):
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print(q.get())    # prints "[42, None, 'hello']"
    p.join()
```

The multiprocessing module - using shared memory - I

```python
from multiprocessing import Process, Value, Array

def f(n, a):
    n.value = 3.1415927
    for i in range(len(a)):
        a[i] = -a[i]

if __name__ == '__main__':
    num = Value('d', 0.0)
    arr = Array('i', range(10))

    p = Process(target=f, args=(num, arr))
    p.start()
    p.join()

    print(num.value)
    print(arr[:])
```

```
3.1415927
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

The multiprocessing module - using shared memory - II

- Initialize array and shared memory block

```
import numpy as np
a = np.array([1, 1, 2, 3, 5, 8])  # Start with an existing NumPy array
from multiprocessing import shared_memory
shm = shared_memory.SharedMemory(create=True, size=a.nbytes)
# Now create a NumPy array backed by shared memory
b = np.ndarray(a.shape, dtype=a.dtype, buffer=shm.buf)
b[:] = a[:]  # Copy the original data into shared memory
```

- Retrieve the name attributed to the shared memory

```
>>> shm.name  # We did not specify a name so one was chosen for us
'psm_21467_46075'
```

The multiprocessing module - using shared memory - III

- On a new shell/process, connect to the memory and retrieve the numpy array

```
>>> import numpy as np
>>> from multiprocessing import shared_memory
>>> # Attach to the existing shared memory block
>>> existing_shm = shared_memory.SharedMemory(name='psm_21467_46075')
>>> # Note that a.shape is (6,) and a.dtype is np.int64 in this example
>>> c = np.ndarray((6,), dtype=np.int64, buffer=existing_shm.buf)
```

- Lastly: clean up the shared memory block on both shells/processes

```
>>> # Clean up from within the second Python shell
>>> del c  # Unnecessary; merely emphasizing the array is no longer used
>>> existing_shm.close()
```

```
>>> # Clean up from within the first Python shell
>>> del b  # Unnecessary; merely emphasizing the array is no longer used
>>> shm.close()
>>> shm.unlink()  # Free and release the shared memory block at the very end
```

The multiprocessing module - using shared memory - IV

- Managers
  - create data which can be shared between different processes;
  - A manager object controls a server process which manages shared objects;
  - Other processes can access the shared objects by using proxies.;

```
>>> with SharedMemoryManager() as smm:
...     sl = smm.ShareableList(range(2000))
...     # Divide the work among two processes, storing partial results in sl
...     p1 = Process(target=do_work, args=(sl, 0, 1000))
...     p2 = Process(target=do_work, args=(sl, 1000, 2000))
...     p1.start()
...     p2.start()  # A multiprocessing.Pool might be more eficient
...     p1.join()
...     p2.join()   # Wait for all work to complete in both processes
...     total_result = sum(sl)  # Consolidate the partial results now in sl
```

The multiprocessing module - using shared memory - V

```
>>> from multiprocessing import shared_memory
>>> a = shared_memory.ShareableList(['howdy', b'HoWdY', -273.154, 100, None, True, 42])
```

```
>>> a[2] = 'dry ice'  # Changing data types is supported as well
>>> a[2]
'dry ice'
>>> a[2] = 'larger than previously allocated storage space'
Traceback (most recent call last):
  ...
ValueError: exceeds available storage for existing str
```

What we will (probably) have in the future

- PEP 554 -- Multiple Interpreters in the Stdlib - still **provisional**
  - Wrapper for the C-api feature available since 1997;
  - Sub-interpreters operate in relative isolation from one another, which provides the basis for an alternative concurrency model.
  - Each sub-interpreter has a GIL;
  - Still share data with “channels”, similar to queues, but only pass data;
    - Communicate-via-shared-memory approach doesn't work!!
  - Sub-interpreters are not intended as a replacement for any method.
  - Certainly they overlap, but the benefits of sub-interpreters include isolation and (potentially) performance.

Other tools for parallel computation

- Dask: https://dask.org/
  - Dask's schedulers scale to thousand-node clusters
  - Dask arrays support most of the NumPy interface
  - Dask DataFrame is used when Pandas fails due to data size or computation speed
  - Dask-ML provides scalable machine learning in Python
- Vaex: https://github.com/vaexio/vaex/
  - incredibly fast and memory efficient support for all common string manipulations
  - Compared to Pandas, string operations are up to ~30–100x faster on your quadcore laptop (up to a 1000 times faster on a 32 core machine)
  - dask.dataframe was actually slower than pure Pandas (~2x). Pandas string operations do not release the GIL, Dask cannot effectively use multithreading
  - Solved by using processes but: slowed down the operations by 40x compared to Pandas, which is 1300x slower (!) compared to Vaex

To keep in mind

- Avoid early optimizations
  - Do we really need to parallelize our problem?
  - Is it worthy, when taking into account the introduced complexity?
- Avoid passing too much data between processes
- We will always be limited by the slowest non-parallelizable function