

Parallel Python

This lecture is based on these resources:

http://calcul.math.cnrs.fr/Documents/Ecoles/2010/cours_multiprocessing.pdf

http://sebastianraschka.com/Articles/2014_multiprocessing.html

<https://code.tutsplus.com/articles/introduction-to-parallel-and-concurrent-programming-in-python--cms-28612>

https://support.scinet.utoronto.ca/education/staticpublic/course146/parallel_python.pdf

Example: Computing the value of $\pi=3.14159\dots$

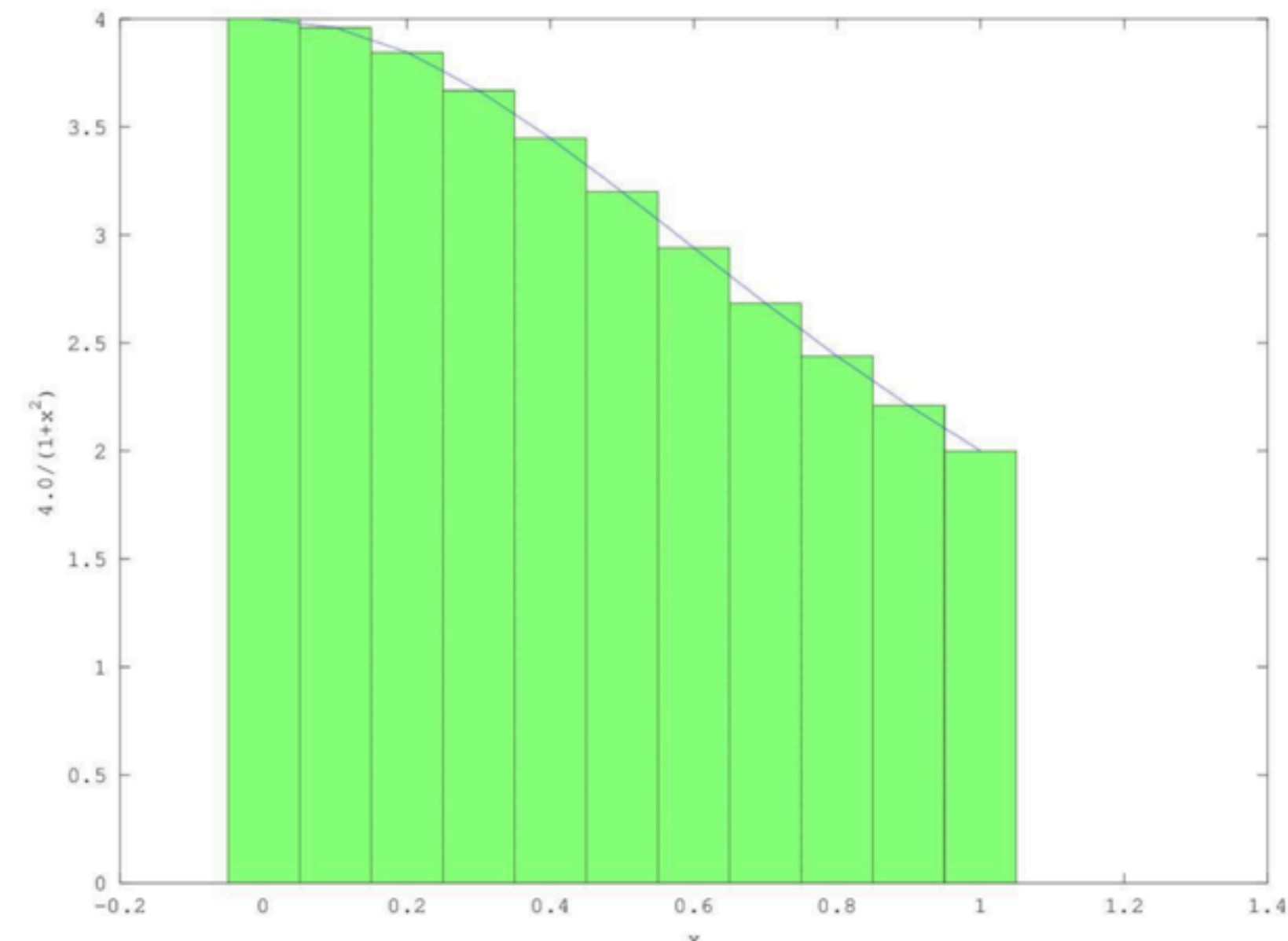
$$F(x) = \frac{4.0}{(1+x^2)}$$

it is known that the value of π can be computed by the numerical integration

$$\int_0^1 F(x) dx = \pi$$

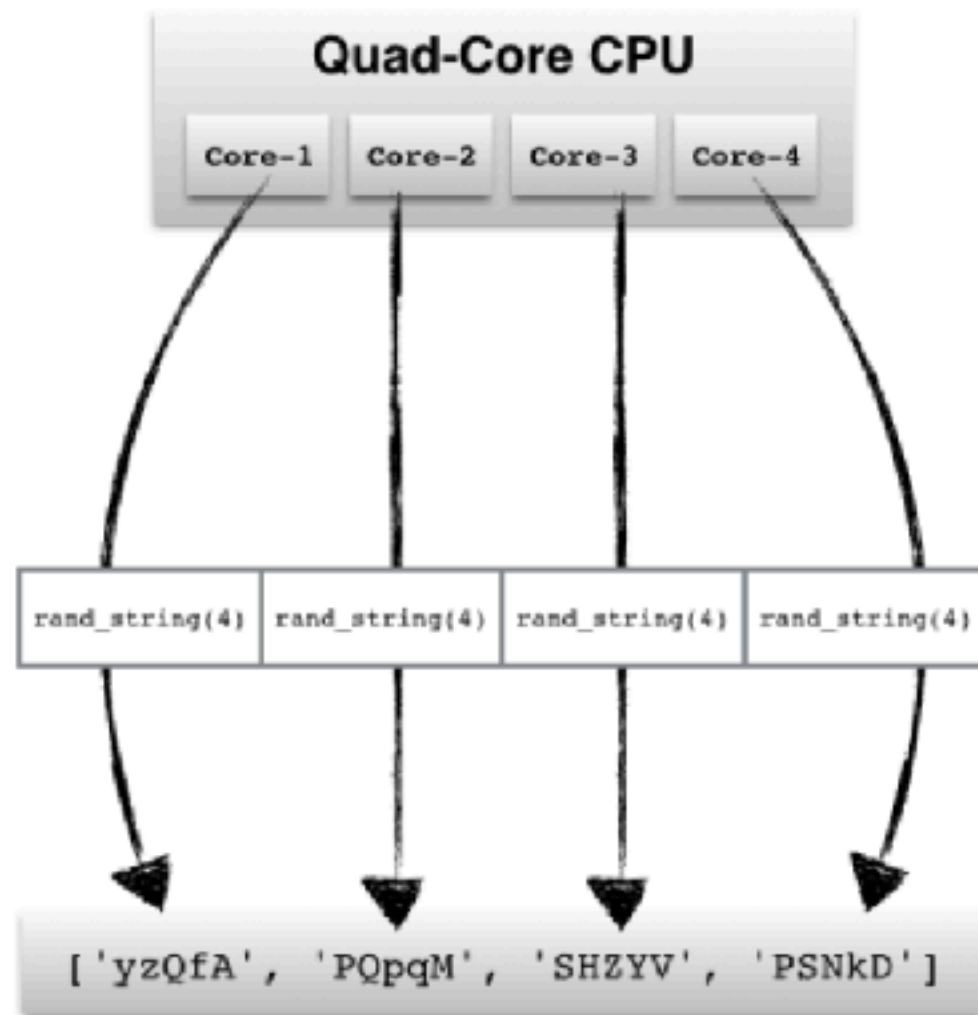
This can be approximated by

$$\sum_{i=0}^N F(x_i) \Delta x \approx \pi$$

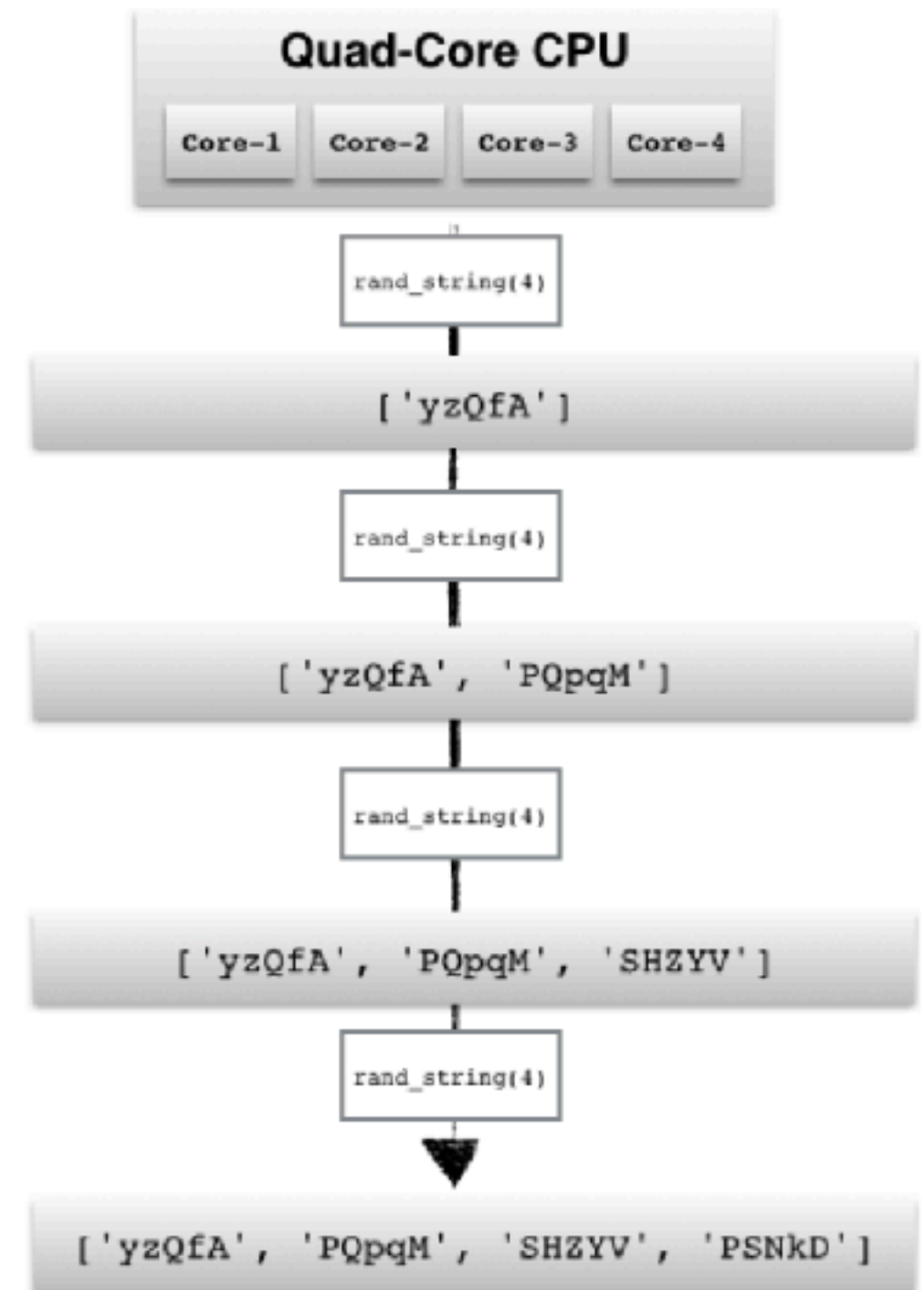


```
1 from __future__ import division
2 import time
3 import math
4
5 def Pi(num_steps):
6     start = time.time()
7     step = 1.0/num_steps
8     sum = 0
9     for i in xrange(num_steps):
10         x= (i+0.5)*step
11         sum += 4.0/(1.0+x*x)
12     pi = step * sum
13     end = time.time()
14     print "Pi with {} steps is {:.20f} in {} secs".format(num_steps, pi, end-start)
15     print "Error: {}".format(math.fabs(pi - math.pi))
16
17 if __name__ == '__main__':
18     Pi(1000000000)
19
```


[parallel processing]



[serial processing]



PARALLEL COMPUTERS

- **Multiprocessor/multicore:**
 - several processors work on data stored in shared memory
- **Cluster:**
 - several processor/memory units work together by exchanging data over a network
- **Co-processor:**
 - a general-purpose processor delegates specific tasks to a special-purpose processor (GPU, FPGA, ...)
- **Other:**
 - Cluster of multicore nodes with GPUs
 - NUMA (non-uniform memory access) architectures
 - ...

Almost all computers made today are parallel!

PARALLELISM VS. CONCURRENCY

Parallelism:

use multiple processors to make a computation faster.

Concurrency:

permit multiple tasks to proceed without waiting for each other.

Different goals that share implementation aspects.

Scientific computing cares more about *parallelism*.
Concurrency is rarely needed.

PARALLEL PROGRAMMING

- ✻ Decomposition of the complete task into independent subtasks and the data flow between them.
- ✻ Distribution of the subtasks over the processors minimizing the total execution time.
- ✻ For clusters: distribution of the data over the nodes minimizing the communication time.
- ✻ For multiprocessors: optimization of the memory access patterns minimizing waiting times.
- ✻ Synchronization of the individual processes.

DIFFICULTIES

Correctness:

- ✱ Verifying that subtasks are indeed independent.
- ✱ Making synchronization patterns deadlock-free.
- ✱ Clusters: verifying communication patterns.

Efficiency:

- ✱ Attributing equal workloads to all processors.
- ✱ Taking into account computation *and* communication.
- ✱ Optimizing for a specific parallel computer.

SYNCHRONIZATION ISSUES

Deadlock:

- Two processes are waiting for each other to finish.
- Usually caused by **locks** or by **blocking communication**.

Race condition:

- Two or more processes modify a shared resource (variable, file, ...)
- Result depends on which process comes first.
- Can be avoided using **locks**, but...
- ... handling locks is *very* difficult and mistakes often cause **deadlocks**.

PARALLEL PROGRAMMING MODELS

Popular models in scientific computing:

- ✻ **Message-passing:** a low-level non-deterministic general model for programming clusters. Used by the very popular MPI library. Suitable but less efficient for multiprocessors.
- ✻ **Multi-threading:** a low-level non-deterministic general model for multiprocessors. Not suitable for clusters. Popular implementations: Posix threads, OpenMP.
- ✻ **Data-parallel:** a high-level deterministic specialized model. Suitable for all parallel computers.
- ✻ **Task farming:** a high-level almost-deterministic specialized model. Suitable for all parallel computers.

PARALLEL COMPUTING LIBRARIES FOR PYTHON

Lots of Python libraries:

<http://wiki.python.org/moin/ParallelProcessing>

- **In the Python standard library:**
 - threading: thread handling and locks
 - multiprocessing (Python 2.6): process-based multithreading
- **In ScientificPython:**
 - Scientific.BSP: “Bulk Synchronous Parallel” model
 - Scientific.DistributedComputing: task farming
- **In IPython:**
 - interactive shell for working with clusters
- **Other:**
 - pypar, pyMPI, mpi4py implement MPI-like message passing

We explore

- **Why is parallelism tricky in Python** (hint: it's because of the GIL—the global interpreter lock).
- **Threads vs. Processes:** Different ways of achieving parallelism. When to use one over the other?
- **Parallel vs. Concurrent:** Why in some cases we can settle for concurrency rather than parallelism.

```
1 import os
2 import time
3 import threading
4 import multiprocessing
5
6 NUM_WORKERS = 4
7
8 def only_sleep():
9     """ Do nothing, wait for a timer to expire """
10    print("PID: %s, Process Name: %s, Thread Name: %s" % (
11        os.getpid(),
12        multiprocessing.current_process().name,
13        threading.current_thread().name)
14    )
15    time.sleep(1)
16
17
18 def crunch_numbers():
19     """ Do some computations """
20    print("PID: %s, Process Name: %s, Thread Name: %s" % (
21        os.getpid(),
22        multiprocessing.current_process().name,
23        threading.current_thread().name)
24    )
25    x = 0
26    while x < 100000000:
27        x += 1
28
```

running only_sleep

```
1 from example import *
2
3 ## Run tasks serially
4 start_time = time.time()
5 for _ in range(NUM_WORKERS):
6     only_sleep()
7 end_time = time.time()
8
9 print("Serial time=", end_time - start_time)
10
11 # Run tasks using threads
12 start_time = time.time()
13 threads = [threading.Thread(target=only_sleep) for _ in range(NUM_WORKERS)]
14 [thread.start() for thread in threads]
15 [thread.join() for thread in threads]
16 end_time = time.time()
17
18 print("Threads time=", end_time - start_time)
19
20 # Run tasks using processes
21 start_time = time.time()
22 processes = [multiprocessing.Process(target=only_sleep()) for _ in range(NUM_WORKERS)]
23 [process.start() for process in processes]
24 [process.join() for process in processes]
25 end_time = time.time()
26
27 print("Parallel time=", end_time - start_time)
28
```

```

01 PID: 95726, Process Name: MainProcess, Thread Name: MainThread
02 PID: 95726, Process Name: MainProcess, Thread Name: MainThread
03 PID: 95726, Process Name: MainProcess, Thread Name: MainThread
04 PID: 95726, Process Name: MainProcess, Thread Name: MainThread
05 Serial time= 4.018089056015015
06
07 PID: 95726, Process Name: MainProcess, Thread Name: Thread-1
08 PID: 95726, Process Name: MainProcess, Thread Name: Thread-2
09 PID: 95726, Process Name: MainProcess, Thread Name: Thread-3
10 PID: 95726, Process Name: MainProcess, Thread Name: Thread-4
11 Threads time= 1.0047411918640137
12
13 PID: 95728, Process Name: Process-1, Thread Name: MainThread
14 PID: 95729, Process Name: Process-2, Thread Name: MainThread
15 PID: 95730, Process Name: Process-3, Thread Name: MainThread
16 PID: 95731, Process Name: Process-4, Thread Name: MainThread
17 Parallel time= 1.014023780822754

```

- In the case of the **serial approach**, things are pretty obvious. We're running the tasks one after the other. All four runs are executed by the same thread of the same process.
- **Using processes** we cut the execution time down to a quarter of the original time, simply because the tasks are executed in parallel. Notice how each task is performed in a different process and on the `MainThread` of that process.

- **Using threads** we take advantage of the fact that the tasks can be executed concurrently. The execution time is also cut down to a quarter, even though nothing is running in parallel. Here's how that goes: we spawn the first thread and it starts waiting for the timer to expire. We pause its execution, letting it wait for the timer to expire, and in this time we spawn the second thread. We repeat this for all the threads. At one moment the timer of the first thread expires so we switch execution to it and we terminate it. The algorithm is repeated for the second and for all the other threads. At the end, the result is as if things were run in parallel. You'll also notice that the four different threads branch out from and live inside the same process: `MainProcess`.
- You may even notice that the threaded approach is quicker than the truly parallel one. That's due to the overhead of spawning processes. As we noted previously, spawning and switching processes is an expensive operation.

PID: 96285, Process Name: MainProcess, Thread Name: MainThread
PID: 96285, Process Name: MainProcess, Thread Name: MainThread
PID: 96285, Process Name: MainProcess, Thread Name: MainThread
PID: 96285, Process Name: MainProcess, Thread Name: MainThread
Serial time= 2.705625057220459
PID: 96285, Process Name: MainProcess, Thread Name: Thread-1
PID: 96285, Process Name: MainProcess, Thread Name: Thread-2
PID: 96285, Process Name: MainProcess, Thread Name: Thread-3
PID: 96285, Process Name: MainProcess, Thread Name: Thread-4
Threads time= 2.6961309909820557
PID: 96289, Process Name: Process-1, Thread Name: MainThread
PID: 96290, Process Name: Process-2, Thread Name: MainThread
PID: 96291, Process Name: Process-3, Thread Name: MainThread
PID: 96292, Process Name: Process-4, Thread Name: MainThread
Parallel time= 0.8014059066772461

running crunch_numbers

```
1 from example import *
2
3 ## Run tasks serially
4 start_time = time.time()
5 for _ in range(NUM_WORKERS):
6     crunch_numbers()
7 end_time = time.time()
8
9 print("Serial time=", end_time - start_time)
10
11 # Run tasks using threads
12 start_time = time.time()
13 threads = [threading.Thread(target=crunch_numbers) for _ in range(NUM_WORKERS)]
14 [thread.start() for thread in threads]
15 [thread.join() for thread in threads]
16 end_time = time.time()
17
18 print("Threads time=", end_time - start_time)
19
20 # Run tasks using processes
21 start_time = time.time()
22 processes = [multiprocessing.Process(target=crunch_numbers, args=(i,)) for i in range(NUM_WORKERS)]
23 [process.start() for process in processes]
24 [process.join() for process in processes]
25 end_time = time.time()
26
27 print("Parallel time=", end_time - start_time)
28
```

The Python Parallel/Concurrent Programming Ecosystem

Python has rich APIs for doing parallel/concurrent programming. In this tutorial we're covering the most popular ones, but you have to know that for any need you have in this domain, there's probably something already out there that can help you achieve your goal.

In the next section, we'll build a practical application in many forms, using all of the libraries presented. Without further ado, here are the modules/libraries we're going to cover:

- `threading`: The standard way of working with threads in Python. It is a higher-level API wrapper over the functionality exposed by the `_thread` module, which is a low-level interface over the operating system's thread implementation.
- `concurrent.futures`: A module part of the standard library that provides an even higher-level abstraction layer over threads. The threads are modeled as asynchronous tasks.
- `multiprocessing`: Similar to the `threading` module, offering a very similar interface but using processes instead of threads.
- `gevent` and `greenlets`: Greenlets, also called micro-threads, are units of execution that can be scheduled collaboratively and can perform tasks concurrently without much overhead.
- `celery`: A high-level distributed task queue. The tasks are queued and executed concurrently using various paradigms like `multiprocessing` or `gevent`.

Examples using the module multiprocessing

see the zip file parallel.zip

