# Threads and processes

An alternative option for parallelization is the built-in `threading` module in python.

In [1]:
```
from threading import Thread
```

In [2]:
```
# repeating from before
import random

def calc_pi(N, name=None):
    "Compute pi using N random samples"
    printing = name is not None
    if printing:
        print(f"{name}: starting")
    M = 0
    for i in range(N):
        # Simulate random coordinates
        x = random.uniform(-1, 1)
        y = random.uniform(-1, 1)
        if x**2 + y**2 < 1: # don't need sqrt b/c 1**2 = 1
            M += 1

    if printing:
        print(f"{name}: Done")
    return 4*M/N
```

In [3]:
```
%%time
calc_pi(10**7)
```

```
CPU times: user 5.48 s, sys: 0 ns, total: 5.48 s
Wall time: 5.48 s
```

Out[3]:
```
3.1425108
```

In [4]:
```
%%time
n=int((10**7)/2)
t1 = Thread(target=calc_pi, args=(n, "Thread 1", ))
t2 = Thread(target=calc_pi, args=(n, "Thread 2", ))

t1.start()
t2.start()

a1 = t1.join()
a2 = t2.join()

# TODO: collect the result and show number of computations and compute pi? -- see examples later on?
# https://stackoverflow.com/questions/6893968/how-to-get-the-return-value-from-a-thread
# Is it correct that the point here is that this is run sequentially and there is no speedup?
```

```
Thread 1: starting
Thread 2: starting
Thread 1: Done
Thread 2: Done
CPU times: user 5.86 s, sys: 12 ms, total: 5.87 s
Wall time: 5.85 s
```

## Discussion: where is the speedup

- ask in the group?

**Solution**

- Python only allows one thread to acces the interpreter at any given time. In other words, if we have a python session and start two threads, only one of them can execute python code at the time.
- This means that the two threads are waiting for the other to finish their work

## Notes from realpython

- In python, a daemon thread shuts down immediately when a program exits. We can achieve this behavior when using `daemon=True` on the `threading.Thread` function.
- If it is not specified, a thread is running in the background. It is only finished when `join` is called on it. Either this is done by us, or it is done when the program exits because python will close all objects, and on `threading` objects, the `_shutdown` method calls join under the hood.
- If we call `join`, the main thread waits for the thread to finish. Whether `daemon=True` does not matter because the process is waited for to finish.

## Note on the Global Interpreter Lock

- The above did not run in parallel. The reaons: python's global interpreter lock
- It prevents us from using multiple cores from a single python instance.
- This makes programming in python safer (intuition???), but leads us to waste precious CPU resources.
- We can circumvent or lift the GIL with two types of solutions
  1. run multiple python instances: `multiprocessing`
  2. have important code outside python: C++ extensions, cython, numba

**Having multiple python instances** The problem is that we need to replicate program state between processes -- that is, if we load up a big dataset into memory and then run parallel processes on it, we need to transfer this big dataset to each child process. This is done with serialization (pickle, json) and creates large overhead. This is why multiprocessing should not be the first choice for parallelization.

**Code outside python** Numpy has many routines that are situated outside of the GIL. Also numba makes this very easy, as we will show now.

**Lesson learned: try out and profile your application!**

In [5]:
```
import numba
```

In [6]:
```
@numba.jit(nopython=True, nogil=True)
def calc_pi_nogil(N):
    M = 0
    for i in range(N):
        x = random.uniform(-1, 1)
        y = random.uniform(-1, 1)
        if x**2 + y**2 < 1:
            M += 1
    return 4 * M / N
```

In [7]:
```
time_nogil = %timeit -o calc_pi_nogil(10**6)
```

```
7.5 ms ± 790 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

In [8]:
```
time_forloop = %timeit -o calc_pi(10**6)
```

```
673 ms ± 30.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

**Note**

In general, it is good practice to use `@numba.jit` with `noypython=True`, to make sure the code is run without running any python objects. There is also a more direct way to do this: `numba.njit`. If we use `nopython=True` and numba is not able to run the required code without any python, it will raise an error. If we do not specify `nopython=True`, it may fall back to object code (which is in python?), causing slowdown.

In [9]:
```
speedup = time_forloop.average / time_nogil.average
print(f"Using nogil is {speedup:.2f} times faster than using the for loop")
```

```
Using nogil is 89.82 times faster than using the for loop
```

In [10]:
```
@numba.jit(nopython=False)
def calc_pi_nogil_wrong(N):
    M = 0
    for i in range(N):
        x = random.uniform(-1, 1)
        y = random.uniform(-1, 1)
        if x**2 + y**2 < 1:
            M += 1
    return 4 * M / N
```

In [11]:
```
calc_pi_nogil_wrong(10**4)
```

Out[11]:
```
3.132
```

In [12]:
```
time_wrong = %timeit -o calc_pi_nogil_wrong(10**6)
```

```
6.92 ms ± 88.6 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

We see that `time_wrong` and `time_nogil` take about the same time. Why is that?

Note on numba options

- `nogil` does not have any effect if we run it just like that above -- it's only when we start threading that we can see the speedup.
  - This will be shown in the asyncio section; would it make sense to also show it here?
- `nopython=False` **still tries to compile to machine code**, but *does not require it*; but because the simple example above succeeds in compiling to use no python objects at all, we do not see a difference in speed when using `nopython=False` and `nopython=True`.

-> the exercise below is kind of in this spirit: numpy always (?) releases the GIL. So I could make this more clear then

### Exercise: Threading on a numpy function

In [13]:
```
import numpy as np

a = np.random.random(10**6)
b = np.random.random(10**6)
```

In [14]:
```
%%timeit -n 10 -r 10
np.sort(a)
np.sort(b)
```

```
127 ms ± 2.23 ms per loop (mean ± std. dev. of 10 runs, 10 loops each)
```

In [15]:
```
%%timeit -n 10 -r 10
t1 = Thread(target=np.sort, args=(a, ))
t2 = Thread(target=np.sort, args=(a, ))

t1.start()
t2.start()

a1 = t1.join()
a2 = t2.join()
```

```
67.1 ms ± 910 µs per loop (mean ± std. dev. of 10 runs, 10 loops each)
```

We see a speed-up of about 50%, which we get because we process the threads in parallel in numpy releases the GIL. The same we could achieve with numba (or with numpy) for the `calc_pi` function. Should we do this as an additional exercise?

## Multiprocessing

We can run multiple processes in parallel with the `multiprocessing` module. Its API is similar to the one from threading, but its behavior is quite different.

In [16]:
```
from multiprocessing import Process
```

In [17]:
```
%%time

if __name__ == "__main__":
    # n = 10**6
    n=int((10**7)/2)
    p1 = Process(target=calc_pi, args=(n, "Process 1"))
    p2 = Process(target=calc_pi, args=(n, "Process 2"))

    p1.start()
    p2.start()

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

```
Process 1: starting
Process 2: starting
Process 1: Done
Process 2: Done
CPU times: user 8.38 ms, sys: 4 ms, total: 12.4 ms
Wall time: 3.49 s
```

### What is going on?

- In contrast to `threading`, we managed to get a speed up here of a bit less than 50 percent
- **But**, under the hood, two new processes with a fresh copy of the python interpreter are created; and all resources associated to the parent are transferred
- Creating a process is resources intensive, multiprocessing is only beneficial if running the function is larger than the overhead of creating a new process
  - In the present context, this seems to be true since there are few objects to be transferred between processes.

### Skipped for now: passing objects & sharing state; using contexts

In [18]:
```
%%time

!python teaching/notes/mp_queue.py
```

```
python: can't open file '/home/flavio/repositories/teaching/parallel-python-workshop/teaching/notes/teaching/notes/mp_queue.py': [Errno 2] No such file or directory
CPU times: user 4.6 ms, sys: 0 ns, total: 4.6 ms
Wall time: 117 ms
```

## Submitting tasks to a pool: role of overhead

Describe task; write to exercise document

### Solution

Make two python files

To vary the amount of work: `mp_pool.py`

```
"Vary the amount of work"


from itertools import repeat
import multiprocessing as mp
from timeit import timeit
from calc_pi import calc_pi


def submit(ctx, N):
    with ctx.Pool() as pool:
        pool.starmap(calc_pi, repeat((N,), 4))


if __name__ == "__main__":
    ctx = mp.get_context("spawn")
    for i in (100, 1_000, 10_000, 1_000_000, 10_000_000): # note true N is 4*this input, but same
order of magnitude
        res = timeit(lambda: submit(ctx, i), number=5)
        print(f"Using {i} samples took {res} seconds.")
```

To vary the amount of workers: `mp_pool_vary_processes`

```
"Vary the amount of workers"
from itertools import repeat
import multiprocessing as mp
from timeit import timeit

from calc_pi import calc_pi


def submit(ctx, n_procs):
    with ctx.Pool() as pool:
        pool.starmap(calc_pi, repeat((1_000_000//n_procs,), n_procs))


if __name__ == "__main__":
    ctx = mp.get_context("spawn")
    for i in (1, 2, 4, 8, 16):
        res = timeit(lambda: submit(ctx, i), number=5)
        print(f"Using {i} workers took {res} seconds.")
```

In [19]:
```
!python teaching/notes/mp_pool.py
```

```
python: can't open file '/home/flavio/repositories/teaching/parallel-python-workshop/teaching/notes/teaching/notes/mp_pool.py': [Errno 2] No such file or directory
```

In [20]:
```
!python teaching/notes/mp_pool_vary_processes.py
```

```
python: can't open file '/home/flavio/repositories/teaching/parallel-python-workshop/teaching/notes/teaching/notes/mp_pool_vary_processes.py': [Errno 2] No such file or directory
```

## Lessons learned

**Varying the amount of work**

- overhead from creating processes. fixed cost -> for low `N`, the overhead dominates, and we need high enough `N` to make this worthwhile.
- with larger samples ( `1_000_000` -> `10_000_000` ), a 10x increase in the amount of work results in a more than 10x increase in the time taken.

**Varying the number of processes**

- If our task is large enough ( `N = 1_000_000` ), it makes sense to spread across multiple workers. We get a speed-up.
  - Compare this to `N = 100`, where there is barely any speed-up as we increase the number of workers. This is because of the overhead from creating the python processes.

- There are limits because of CPU congestion (only have 4 physical cores)

In [21]:
```
%%time
calc_pi(100_000)
```

```
CPU times: user 63.7 ms, sys: 0 ns, total: 63.7 ms
Wall time: 63.3 ms
```

Out[21]: 3.1538