Note: time unit conversions

- 1ns = 1e-6 ms = 0.000001 millisecond
- 1 millisecond (ms) = 0.001 seconds
- 1 microsecond = 0.001 millseconds
- alternatively
    - 10e-9 = nanosecond (ns)
    - 10e-6 = microsecond (mu s)
    - 10e-3 = millisecond (ms)

Note: interpreting timings

- `user` + `sys` tell us whow much actual CPU time the process has used. `user` is the time taken on the CPU and `sys` is additional time taken such as allocating memory, which the code cannot do from "user" mode.
- I think wall time is `real`, and could be confounded by other processes

Computing π

We now start with the first paralellization tasks. We will

- see how a python application can be parallelized
- discuss **data parallelism** and **task parallelism**

We use the example of computing π because it is a simple example that still takes some time to compute, so perfect for our instructional purposes.

The way this works is with **Monte-Carlo** methods--this means approximating exact results with random numbers and simulation.

The geometry behind

add figure from online material

- link to online material
- assume a unit radius
- then the surface of the square is `4r^2`
- the surface of the circle is `pi r^2`
- if we can compute the surfaces, we can call them `M` for the circle and `N` for the square
- then we have `M / N = pi r^2 / 4 r^2`
- solving for `pi` we get `pi = 4 M / N`

How do we get `M` and `N`?

Exercise: implement the algorithm (time?)

- using only standard python and the `random.uniform` function

In [1]:
```python
import random
import math

def calc_pi(N):
    "Compute pi using N random samples"
    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

    return 4*M/N
```

In [2]:
```python
time_forloop = %timeit -o calc_pi(10**6)
```

```
560 ms ± 15 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Before starting to parallelize, we should make the inner function (the for loop) as efficient as possible. We discuss two options

1. vectorization with `numpy`
2. native code generation with `numba`

Vectorization works by applying the computation over an array of numbers, instead of each number individually.

What does `numba` do?

- when we execute python code, the line is *interpreted* and directly performed. the speed of this depends on how the programming language is designed *(interpretation)*
- an alternative is *compiled* code, where we write in some language and the translate the code to machine code. Only after this we execute the program. *(ahead-of-time compilation)*
- numba sits in between the two and performs *just-in-time compilation*. It generates optimized machine code---that is faster than python code---, at runtime. This means that our code is compiled to machine code as we define the function.
    - this is faster than python code but has additional overhead from compiling

Numpy

In [3]:
```python
# can use this to show how to get to the function below
import numpy as np
np.random.uniform(-1, 1, (1, 1))
points = np.random.uniform(-1, 1, (2, 100))
(points**2).sum(axis=0)
(points**2).sum(axis=0) < 1
np.count_nonzero((points**2).sum(axis=0) < 1)

M = np.count_nonzero((points**2).sum(axis=0) < 1)

4 * M / 100
```

Out[3]:
```
2.88
```

In [4]:
```python
def calc_pi_numpy(N):
    "Compute pi using N samples with numpy"
    points = np.random.uniform(-1, 1, (2, N))
    # M = np.count_nonzero((points**2).sum(axis=0) < 1)
    M = np.count_nonzero((points**2).sum(axis=0) < 1)
    return 4 * M / N
```

In [5]:
```python
calc_pi_numpy(10**6)
```

Out[5]:
```
3.139856
```

In [6]:
```python
time_numpy = %timeit -o calc_pi_numpy(10**6)
```

```
15.5 ms ± 23.6 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In [7]:
```python
speedup = time_forloop.average / time_numpy.average
print(f"Using numpy is {speedup:.2f} times faster than using the for loop")
```

```
Using numpy is 36.14 times faster than using the for loop
```

This is a *vectorized* version of the algorithm: It demonstrates **data parallelism** where a single operation is applied over a collection of data.

This contrasts to **task parallelism** where different independent procedures are applied in parallel. Example: cutting vegetables while simmering the split peas.

Discussion: Is this all better?

What are the downside of the vectorized implementation, and of data parallelism in general?

- it uses more memory
- it is perhaps less intuitive
- it is more monolithic and it cannot easily be broken up into parts -- in contrast to the example below.

Challenge: Daskify

Write `calc_pi_dask` to make the numpy version parallel. Compare speed and memory performance with the numpy version. Remember that dask.array mimics the numpy API.

In [8]:
```python
#can use this to illustrate how get to to the function below
import dask.array as da
points = da.random.uniform(-1, 1, (2, 100))
(points**2).sum(axis=0).compute() < 1
work = da.count_nonzero((points**2).sum(axis=0) < 1)
work.compute()
```

Out[8]:
```
76
```

In [9]:
```python
def calc_pi_dask(N):
    "Compute pi using N samples with dask"
    points = da.random.uniform(-1, 1, (2, N))
    work = da.count_nonzero((points**2).sum(axis=0) < 1)

    # using the physical number of cores, as we found out in previous episode
    M = work.compute(num_workers=4)
    return 4 * M / N
```

In [10]:
```
calc_pi_dask(10**6)
```

Out[10]:
```
3.142852
```

In [11]:
```
time_dask = %timeit -o calc_pi_dask(10**6)
```

```
22.1 ms ± 142 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

In [12]:
```
speedup = time_forloop.average / time_dask.average
print(f"Using dask is {speedup:.2f} times faster than using the for loop")
```

```
Using dask is 25.32 times faster than using the for loop
```

Compare memory and performance

In [13]:
```
from memory_profiler import memory_usage
import matplotlib.pyplot as plt
```

In [14]:
```
# Little wrappers to make life easy
N = 10**8
def pi_with_numpy():
    return calc_pi_numpy(N)

def pi_with_dask():
    return calc_pi_dask(N)
```

In [15]:
```
memory_numpy = memory_usage(pi_with_numpy, interval=0.01)
time_numpy = %timeit -o pi_with_numpy()
```

```
1.72 s ± 32.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

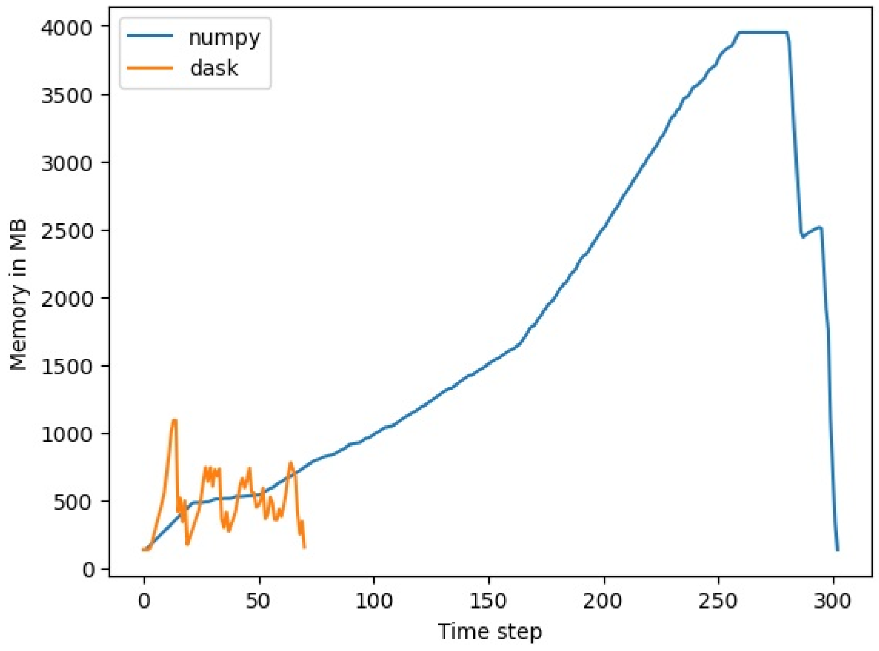
In [16]:
```
memory_dask = memory_usage(pi_with_dask, interval=0.01)
time_dask = %timeit -o pi_with_dask()
```

```
673 ms ± 4.52 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

In [17]:
```
speedup = time_numpy.average / time_dask.average
print(f"Using dask is {speedup:.2f} times faster than using numpy")
```

```
Using dask is 2.56 times faster than using numpy
```

In [18]:
```
plt.plot(memory_numpy, label="numpy")
plt.plot(memory_dask, label="dask")
plt.xlabel("Time step")
plt.ylabel("Memory in MB")
plt.legend()
plt.show()
```



Using numba to accelerate python code

numba makes it easy for us to build accelerated functions. We can call the decorator `numba.jit`. What does it do? -- It sends the function, to which we apply the decorator, to machine code when we execute the cell

In [19]:
```
import numba

@numba.jit
def sum_range_numba(a):
    "Compute the sum of the numbers in the range [0, a)"
    x = 0
    for i in range(a):
        x += 1
    return x
```

Let's time three different versions of this.

In [20]:
```
## Naive python iterators
time_naive = %timeit -o sum(range(10**7))
```

```
120 ms ± 3.69 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

In [21]:
```
# numpy
time_numpy = %timeit -o np.arange(10**7).sum()
```

```
13.5 ms ± 8.98 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In [22]:
```
# numba
time_numba = %timeit -o sum_range_numba(10**7)
# sometimes a warning appears "[could be that ..] intermediate result is being cached".
# re-running usually helps
```

```
126 ns ± 2.44 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

In [23]:
```
speedup = time_numpy.average / time_numba.average
print(f"Using numba is {speedup:.2f} times faster than using numpy")
```

```
Using numba is 107262.71 times faster than using numpy
```

So, numba is 100k times faster (?)

(14.7.5 * 10e-3) / (128 * 10e-9) = (17.5 * 10e-3) / (12.8 * 10e-8) = 10e-5

`numba` JIT does not work on every python or numpy feature, but a function is a good candidate if written with a python for-loop over a large range of values, as with `sum_range_numba()`.

Note on just-in-time compilation

- there may be little or no speed-up when function is called the first time
- similarly, when using `timeit` you may get the following message
    - `The slowest run took 14.83 times longer than the fastest. This could mean that an intermediate result is being cached.`
- reason: on the first call, the JIT compiler needs to compile the function. on subsequent runs, the function is reused
- **the same function can only be reused if it is called with the same argument types (int, float, etc)**

In [24]:
```
%time sum_range_numba(10**7)
```

```
CPU times: user 4 µs, sys: 1 µs, total: 5 µs
Wall time: 6.2 µs
```

Out[24]:
```
10000000
```

In [25]:
```
%time sum_range_numba(10.**7)
```

```
CPU times: user 31.3 ms, sys: 0 ns, total: 31.3 ms
Wall time: 31.2 ms
```

Out[25]:
```
10000000
```

In [26]:
```
%time sum_range_numba(10.**7)
```

```
CPU times: user 4 µs, sys: 1 µs, total: 5 µs
Wall time: 6.44 µs
```

Out[26]:
```
10000000
```

Challenge: numbify `calc_pi`

Create a numba version of `calc_pi`. Time it.

In [27]:
```
# note we're moving back to a function that looks similar to the original function
@numba.jit
def calc_pi_numba(N):
    "Compute pi using N samples with numba"
    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 < 1.0: # don't need sqrt b/c 1**2 = 1
            M += 1

    return 4*M/N
```

In [28]:
```
calc_pi_numba(10**7)
```

Out[28]:
```
3.1414012
```

In [29]:
```
time_numba = %timeit -o calc_pi_numba(10**6)
```

```
6.27 ms ± 5.46 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In [30]:
```
speedup = time_dask.average / time_numba.average
print(f"Using numba is {speedup:.2f} times faster than using dask")
```

```
Using numba is 107.30 times faster than using dask
```

- also: random number generator changes

Conclusion

- measuring - knowing: always profile your code to see which parallelization method works best
- numba often outperforms other methods, but it is not always possible to rewrite code so that one can use numba with it