P

- **P**arallelizing
- **P**ython
- **P**rograms
- **P**owerfully &
- **P**ainlessly

Cliff Kerr

Clinton Collins

May 11th, 2021

Outline

1. Introduction
   a. What's the problem?
   b. What's the solution?
2. The tools
   a. The most basic tool: multiprocess
   b. The simplest tool: Sciris
   c. The coolest tool: Dask
   d. The most flexible tool: Celery
3. Putting it in practice
   a. Jupyter
   b. PyCharm

Image credit: furiousdriving

Code: https://github.com/amath-idm/parallelizing-python

Introduction

What's the problem?

- Many scientific/engineering problems look like:

```
for x in a lot:
    do_thing(x)
```

We're going to focus on this one

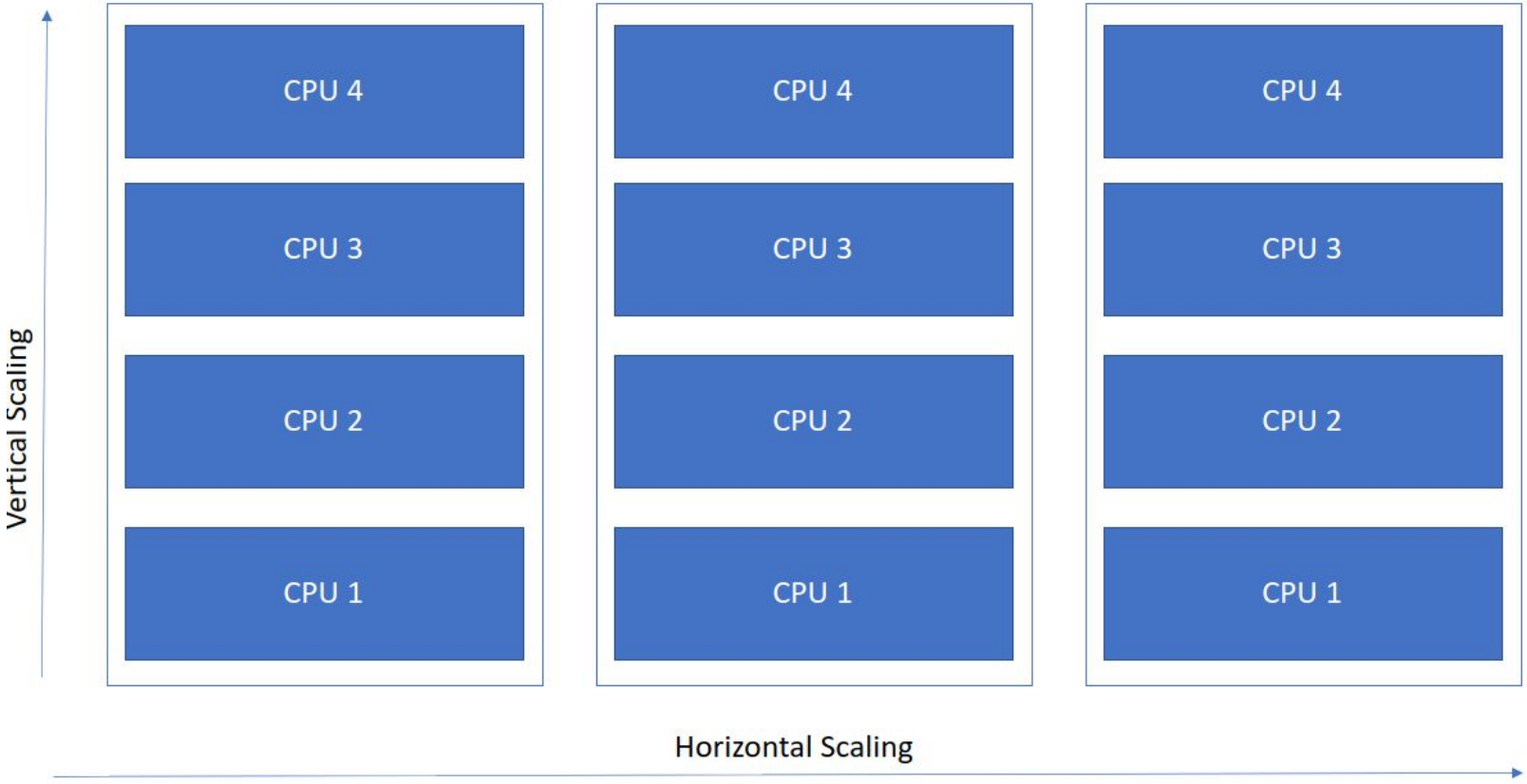
- **Embarrassingly parallel**: every iteration is independent:

```
y = []
for x in [0,1,2,3,4]:
    y[x] = do_arithmetic_on(x)
```

- **Majestically parallel** (...?): the iterations are <u>not</u> independent:

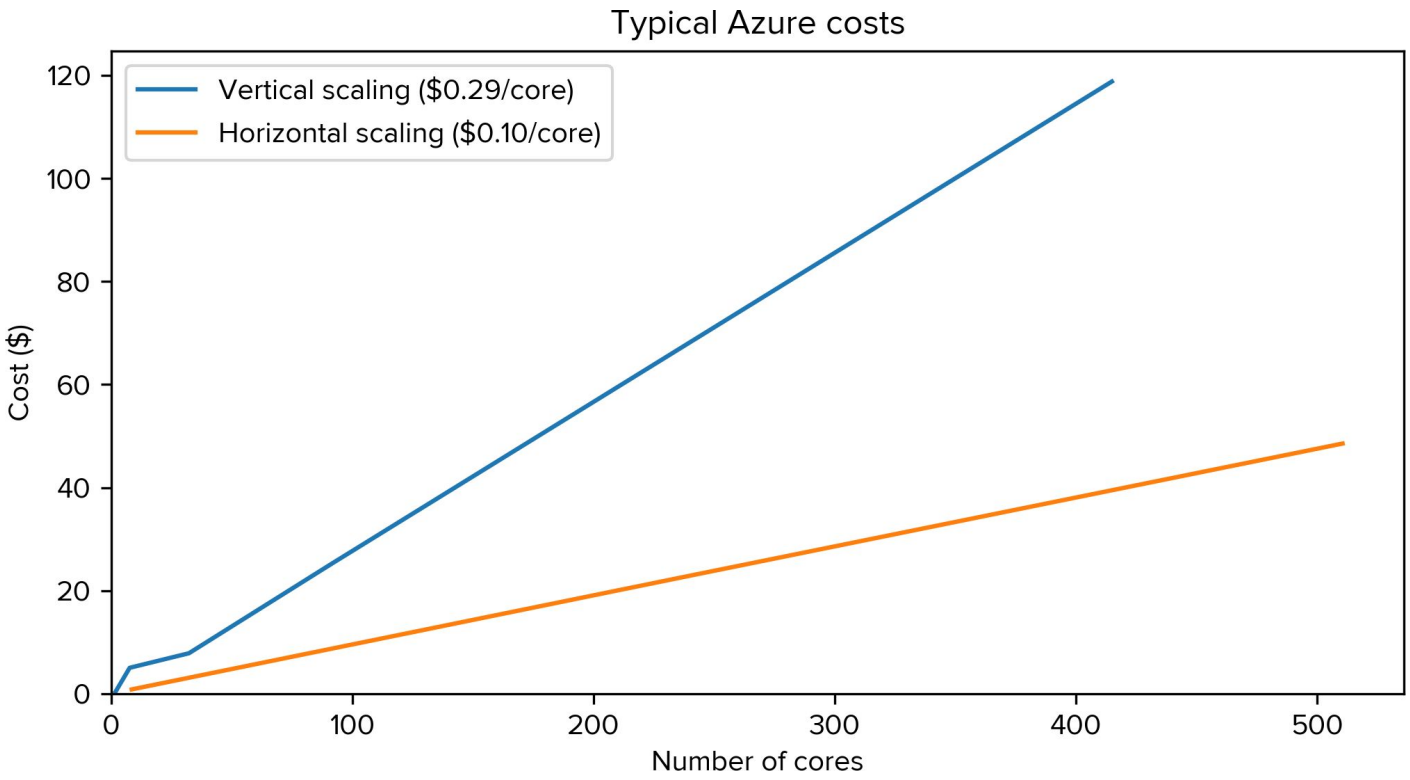
```
y = init()
for t in [0,1,2,3,4]:
    y[t+1] = y[t].update()
```

The solution: horizontal and vertical scaling



The solution: horizontal and vertical scaling

- Vertical scaling: easy
- Horizontal scaling: cheap



Split-apply-combine ("map-reduce")

- **Split** ("map"): decide which data goes to which processor
- **Apply**: perform the (computationally intensive) operation
- **Combine** ("reduce"): gather partial results into global result

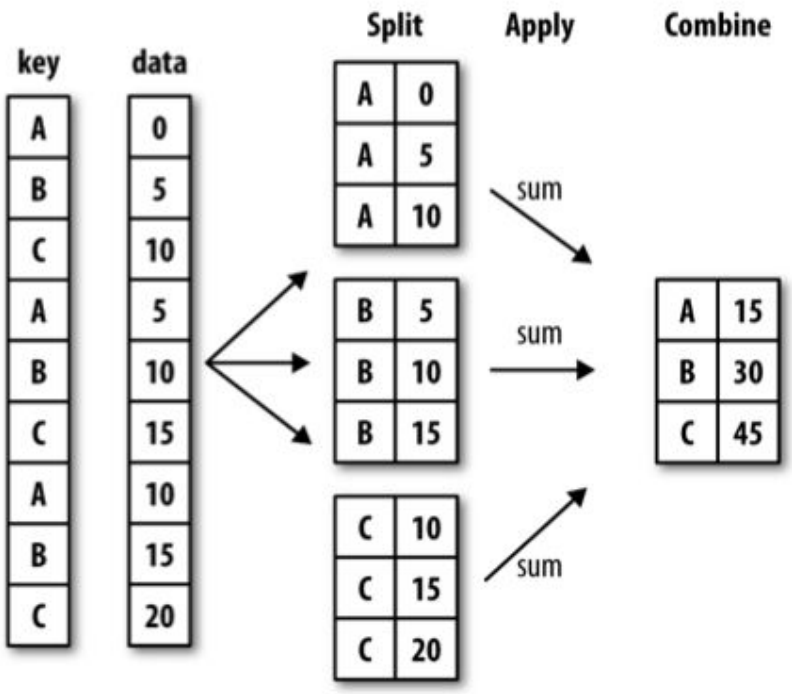


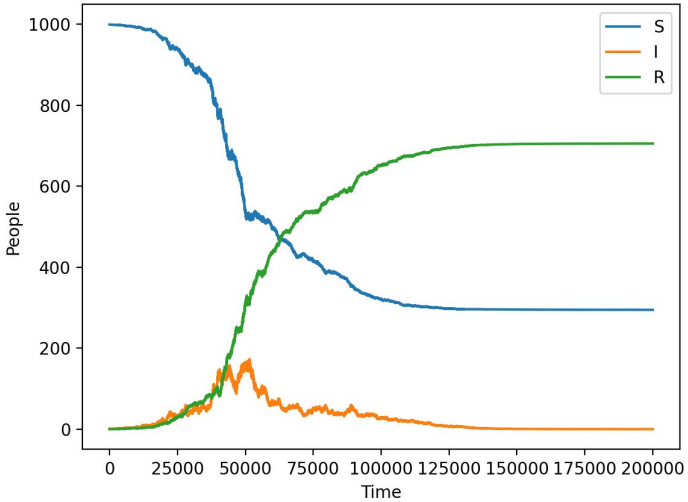
Image credit: Learning Pandas

Tools

The model

(Just stock standard SIR...plus a bit of noise)

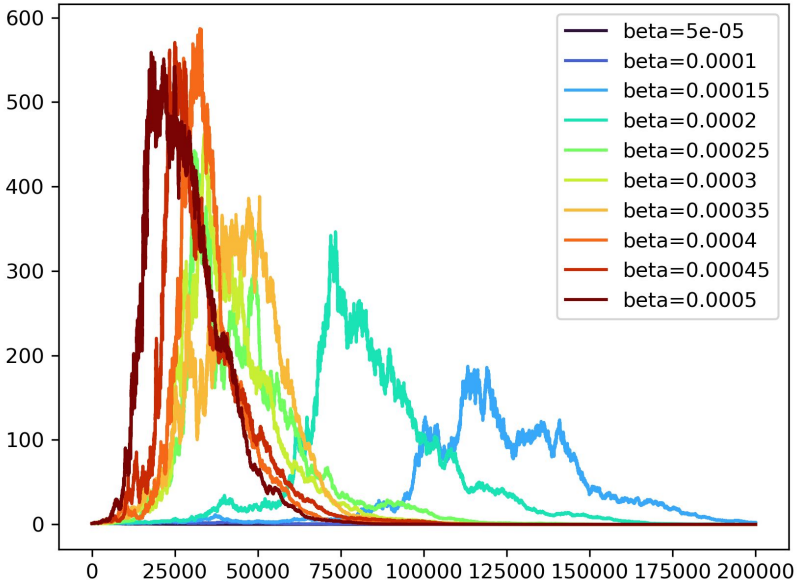
```
>>> run('model.py')
Elapsed time: 0.599 s
<__main__.SIR at 0x7f9359911400>
------------------------------------------------------------
Methods:
  initialize()        plot()              run()
------------------------------------------------------------
    I: [1, 1.0116713953507477, 1.0118260534863504, 1.0241618644003567, 1.037313833 [...]
    N: 1000.0
    R: [0, -0.0017352692409502261, -0.0048905770908480575, -0.011775664935330905,  [...]
    S: [999.0, 998.9900638738902, 998.9930645236045, 998.987613800535, 998.9766982 [...]
 beta: 0.0002
gamma: 0.0001
noise: 30
 npts: 200000.0
 seed: 1
 tvec: array([0.00000e+00, 1.00000e+00, 2.00000e+00, ..., 1.99998e+05, 1.99 [...]
------------------------------------------------------------
```




```python
import numpy as np
import sciris as sc

class SIR(sc.prettyobj):

    def __init__(self, beta=2e-4, gamma=1e-4, npts=2e5, N=1e3, noise=30, seed=1):
        self.beta  = beta
        self.gamma = gamma
        self.npts  = npts
        self.N     = N
        self.noise = noise
        self.seed  = seed
        self.initialize()
        return

    def initialize(self):
        np.random.seed(self.seed)
        self.S = [self.N-1]
        self.I = [1]
        self.R = [0]
        self.tvec = np.arange(self.npts+1)
        return

    def run(self):
        for t in np.arange(self.npts):
            S = self.S[-1]
            I = self.I[-1]
            R = self.R[-1]
            infections = self.beta*S*I/self.N*(1 + self.noise*np.random.randn())
            recoveries = self.gamma*I*(1 + self.noise*np.random.randn())
            S = S - infections
            I = I + infections - recoveries
            R = R + recoveries
            self.S.append(S)
            self.I.append(I)
            self.R.append(R)
        return
```

Tool #0: Running in serial

- Pros: simple
- Cons: slow

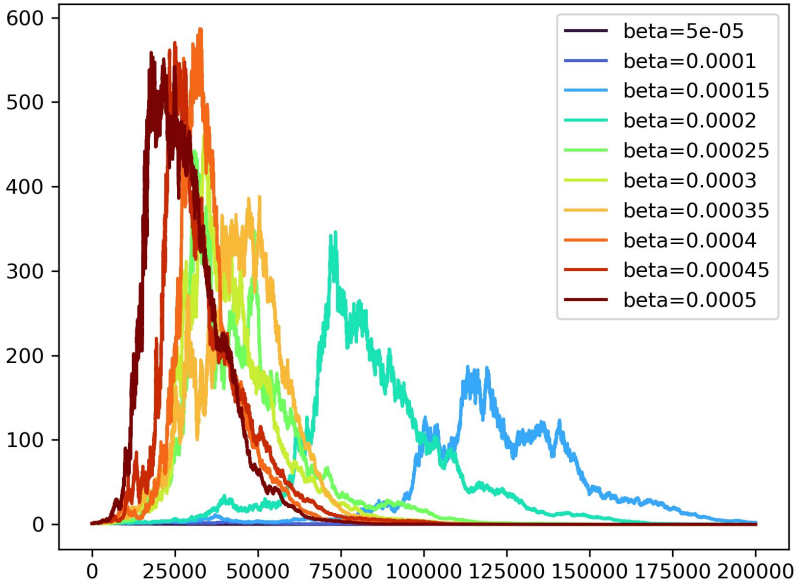



```python
import numpy as np
import sciris as sc
from model import run_sir

# Initialization
n_runs = 10
seeds = np.arange(n_runs)
betas = np.linspace(0.5e-4, 5e-4, n_runs)


if __name__ == '__main__':

    # Run
    sc.tic()
    sirlist = []
    for r in range(n_runs):
        sir = run_sir(seed=seeds[r], beta=betas[r])
        sirlist.append(sir)
    sc.toc()
```

```
Elapsed time: 4.35 s
```

Tool #1: multiprocessing

- Pros: built-in
- Cons: inflexible, lots of boilerplate

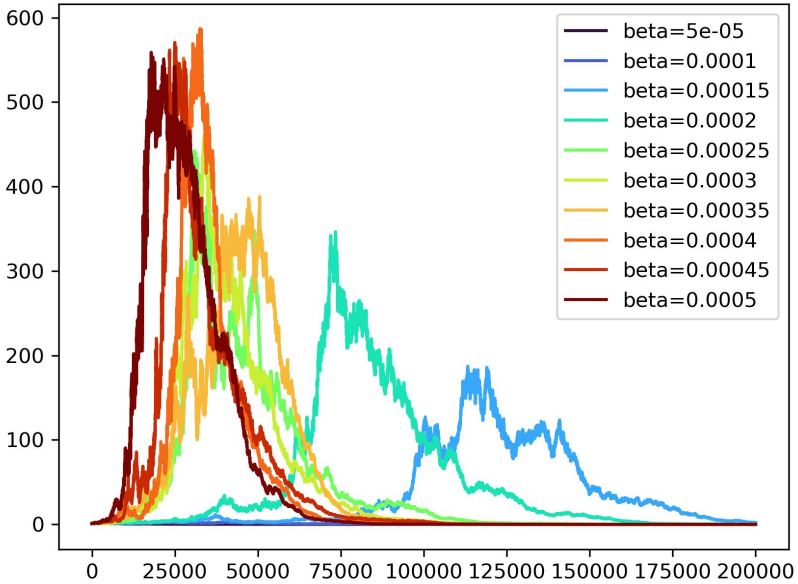



```python
import numpy as np
import sciris as sc
from model import run_sir
import multiprocessing as mp

# Initialization
n_runs = 10
seeds = np.arange(n_runs)
betas = np.linspace(0.5e-4, 5e-4, n_runs)


def run_multiprocessing(args):
    seed, beta = args
    sir = run_sir(seed=seed, beta=beta)
    return sir


if __name__ == '__main__':

    # Run
    sc.tic()
    inputlist = [(seed,beta) for seed,beta in zip(seeds, betas)]
    multipool = mp.Pool(processes=mp.cpu_count())
    sirlist = multipool.map(run_multiprocessing, inputlist)
    multipool.close()
    multipool.join()
    sc.toc()
```

Elapsed time: 1.11 s

Tool #2: Sciris

- Pros: most compact, most flexible input options
- Cons: less configurable run options

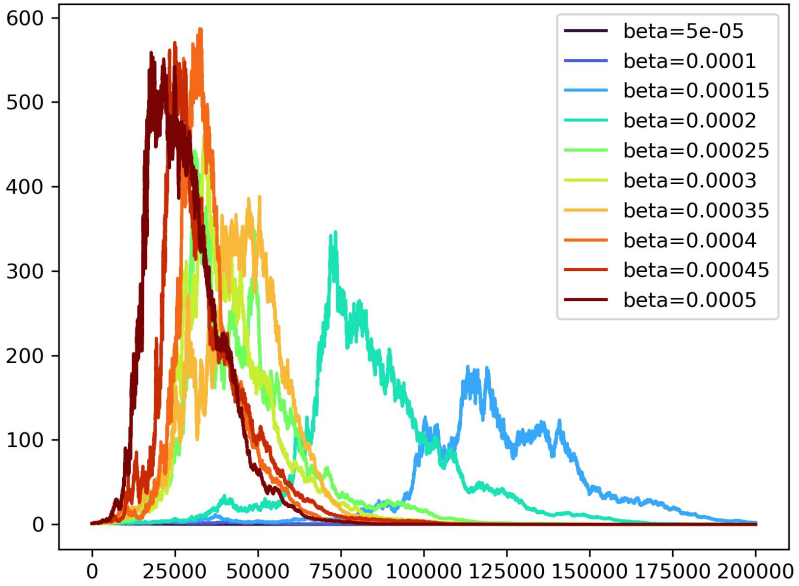



```python
import numpy as np
import sciris as sc
from model import run_sir

# Initialization
n_runs = 10
seeds = np.arange(n_runs)
betas = np.linspace(0.5e-4, 5e-4, n_runs)


if __name__ == '__main__':

    # Run
    sc.tic()
    sirlist = sc.parallelize(run_sir, iterkwargs=dict(seed=seeds, beta=betas))
    sc.toc()
```

Elapsed time: 1.16 s

Tool #3: Dask

- Pros: full data science workflow, extremely flexible and powerful
- Cons: steeper learning curve

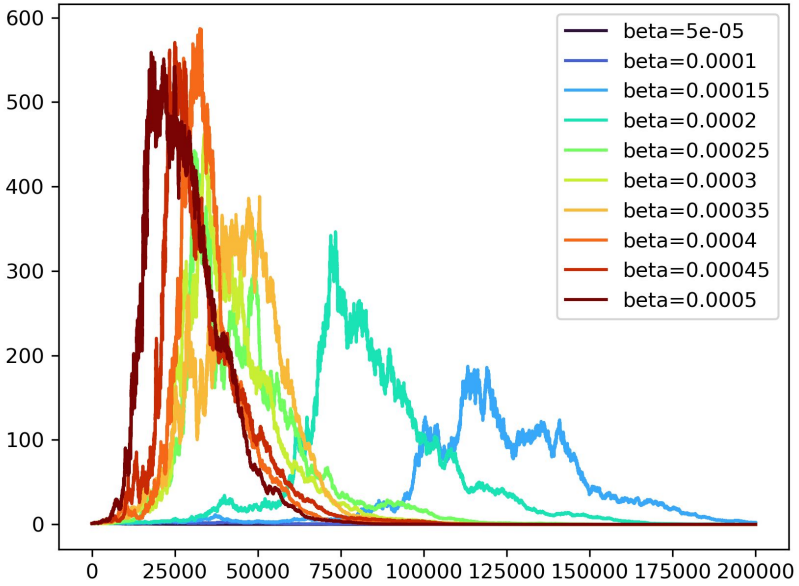



```python
import numpy as np
import sciris as sc
from model import run_sir
import dask
from dask.distributed import Client

# Initialization
n_runs = 10
seeds = np.arange(n_runs)
betas = np.linspace(0.5e-4, 5e-4, n_runs)


def run_dask(seed, beta):
    sir = run_sir(seed=seed, beta=beta)
    return sir


if __name__ == '__main__':

    # Run
    sc.tic()
    client = Client(n_workers=n_runs)
    queued = []
    for r in range(n_runs):
        run = dask.delayed(run_dask)(seeds[r], betas[r])
        queued.append(run)
    sirlist = list(dask.compute(*queued))
    sc.toc()
```

Elapsed time: 2.76 s

Tool #4: Celery

- Pros: only tool designed to handle horizontal scaling
- Cons: *very* steep learning curve




```python
import numpy as np
import sciris as sc
from model import run_sir
from celery import Celery
import subprocess

# Initialization
n_runs = 10
seeds = np.arange(n_runs)
betas = np.linspace(0.5e-4, 5e-4, n_runs)

app = Celery('t4_celery', backend='rpc://', broker='pyamqp://guest@localhost//')
app.conf['task_serializer'] = 'pickle'
app.conf['result_serializer'] = 'pickle'
app.conf['accept_content'] = ['json', 'pickle']

@app.task
def run_celery(args):
    seed, beta = args
    sir = run_sir(seed=seed, beta=beta)
    return sir

if __name__ == '__main__':

    # Run
    sc.tic()
    celery_cmdline = 'celery -A t4_celery worker --loglevel=INFO'.split(' ')
    subprocess.Popen(celery_cmdline)
    inputlist = [(seed,beta) for seed,beta in zip(seeds, betas)]
    jobs = []
    for arg in inputlist:
        jobs.append(run_celery.delay(arg))
    sirlist = []
    for job in jobs:
        while not job.ready():
            sc.timedsleep(1)
        else:
            sirlist.append(job.get())
    sc.toc()
```

```
Elapsed time: 2.31 s
```

Comparison

Serial

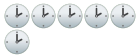
```python
sirlist = []
for r in range(n_runs):
    sir = run_sir(
        seed=seeds[r],
        beta=betas[r]
    )
    sirlist.append(sir)
```

4.35 seconds

Multiprocessing

```python
import multiprocessing as mp

def run_multiprocessing(args):
    seed, beta = args
    sir = run_sir(seed=seed, beta=beta)
    return sir

inputlist = [
    (seed,beta) for
    seed,beta in
    zip(seeds, betas)
]
multipool = mp.Pool(processes=mp.cpu_count)
sirlist = multipool.map(
    run_multiprocessing,
    inputlist
)
multipool.close()
multipool.join()
```

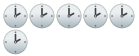
1.11 seconds



Sciris

```python
import sciris as sc

sirlist = sc.parallelize(run_sir,
    iterkwargs=dict(seed=seeds, beta=betas)
)
```

1.16 seconds



Dask

```python
import dask
from dask.distributed import Client

def run_dask(seed, beta):
    sir = run_sir(seed=seed, beta=beta)
    return sir

client = Client(n_workers=n_runs)
queued = []
for r in range(n_runs):
    run = dask.delayed(run_dask)(
        seeds[r],
        betas[r]
    )
    queued.append(run)
sirlist = list(dask.compute(*queued))
```

2.76 seconds

Celery

```python
from celery import Celery
import subprocess

app = Celery('t4_celery', backend='rpc://',
broker='amqp://localhost//')
app.conf['task_serializer'] = 'pickle'
app.conf['result_serializer'] = 'pickle'
app.conf['accept_content'] = ['json', 'pickle']

@app.task
def run_celery(args):
    seed, beta = args
    sir = run_sir(seed=seed, beta=beta)
    return sir

celery_cmdline = 'celery -A t4_celery worker'.split(' ')
subprocess.Popen(celery_cmdline)
inputlist = [
    (seed,beta) for
    seed,beta in
    zip(seeds, betas)
]
jobs = []
for arg in inputlist:
    jobs.append(run_celery.delay(arg))
sirlist = []
for job in jobs:
    while not job.ready():
        sc.timedsleep(1)
    else:
        sirlist.append(job.get())
```

2.31 seconds

🕑 = 0.2 seconds

Shameless promotion of `sc.parallelize()`

Simple shortcut to multiprocessing.map():

```
def f(x):
    return x*x

results = sc.parallelize(f, [1,2,3])
```

Iterate by list of tuples, dict-of-lists, or list-of-dicts:

```
def f(x,y):
    return x*y

results1 = sc.parallelize(f, iterarg=[(1,2),(2,3),(3,4)])
results2 = sc.parallelize(f, iterkwargs={'x':[1,2,3], 'y':[2,3,4]})
results3 = sc.parallelize(f, iterkwargs=[{'x':1, 'y':2}, {'x':2, 'y':3}, {'x':3, 'y':4}])
```

Can't-get-more-embarrassingly parallel:

```
def rnd():
    np.random.seed()
    return np.random.random()

results = sc.parallelize(rnd, 10, ncpus=4)
```

Supply non-iterated function arguments and load balancing:

```
def myfunc(i, x, y):
    np.random.seed()
    xy = [x+i*np.random.randn(100), y+i*np.random.randn(100)]
    return xy

xy1 = sc.parallelize(myfunc, x=5, y=10, iterarg=[0,1,2])
xy2 = sc.parallelize(myfunc, kwargs={'x':3, 'y':8}, iterarg=range(5), maxload=0.8, interval=0.2)
```

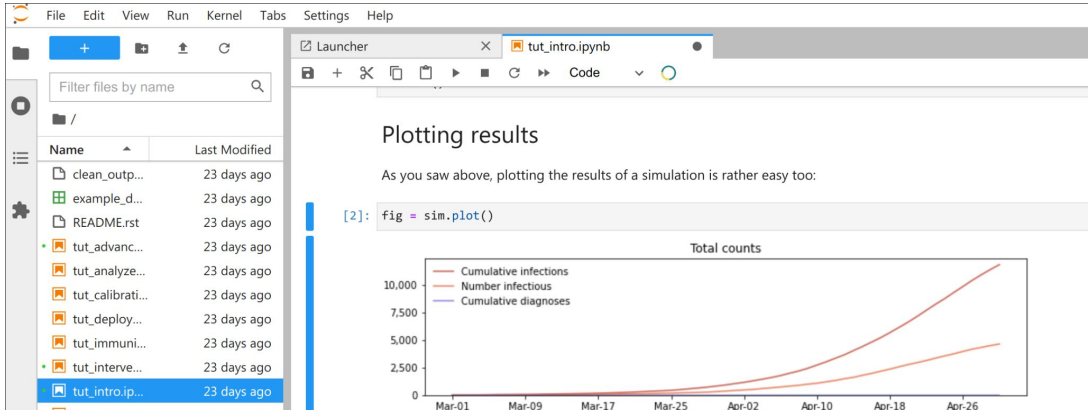
Putting it in practice

Connecting Jupyter to an HPC

1. Launch Jupyter Lab on the remote HPC:
   `remote:~> jupyter lab --no-browser --port=9464`

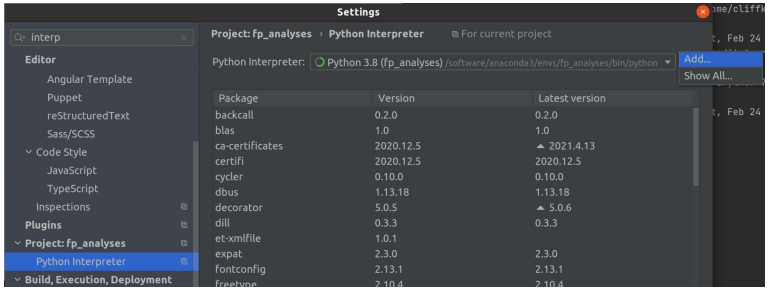
2. Create an SSH tunnel from your local computer to the remote HPC:
   `local:~> ssh -L 9464:remote.org:9464 user@remote.org -N -v -v`

3. Go to `localhost:9464` in your browser – edit local, run remote!

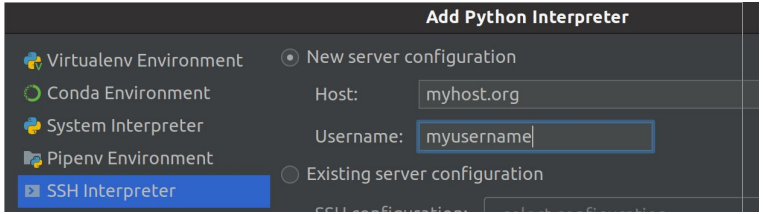


Connecting PyCharm to an HPC

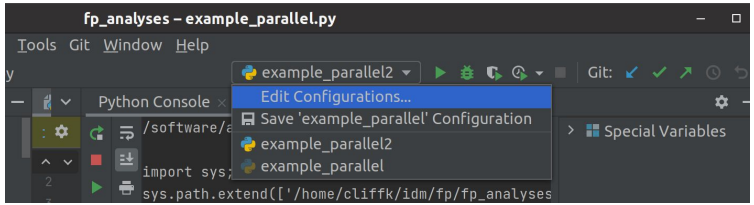
1. In PyCharm, go to File > Settings > Project > Python Interpreter > Add:



2. Create a new SSH interpreter:



3. Select this interpreter from "Edit configurations":



Conclusions

- Parallelizing doesn't have to be hard – can be literally one line, within your IDE

- Things that can always be parallelized:
  - Parameter sweeps
  - Sensitivity analyses
- Things that can't be (easily) parallelized:
  - Sequential timesteps
  - Interacting agents

Image credit: hoppip

- If you can use vertical scaling, Sciris will probably do everything you need
- If you need horizontal scaling, try Dask
- If all else fails, use Celery