YOU MAY NEED PARALLELISM MORE THAN YOU THINK

- embarassingly parallel problems are common
- they may be easier to solve than you think
- but they also be not so trivial ;-)

IMPORTING FILES IN A DATABASE

real life example: CSV data for stock options

code	date	value
000000	2017-07-07	18.64
000001	2017-07-07	12.19
000002	2017-07-07	17.56
000003	2017-07-07	14.25
000004	2017-07-07	18.36

you may think it is dominated by I/O and that there is no benefit in parallelizing since you are writing on a single disk

THINK AGAIN

IMPORTING SEQUENTIALLY

- 2 years of data, 10,000 securities per day, 5+ M of rows, 184 MB of CSV
- notice that I have no SSD and no optimizations of any kind except removing the db indices

```
CREATE TABLE price (code VARCHAR(10), date DATE, value FLOAT);
-- ALTER TABLE price ADD PRIMARY KEY (code, date);
-- ALTER TABLE price DROP CONSTRAINT price_pkey;
```

- how much time do you think it will take?

LET'S TRY

Here is the code

```
import os, time, subprocess

def import_data(path):
    cmd = "COPY price FROM '%s' WITH DELIMITER ','" % path
    subprocess.call(['psql', '-c', cmd])

def main(datadir):
    t0 = time.time()
    n = 0
    fnames = (os.path.join(datadir, f) for f in os.listdir(datadir))
    for _ in map(import_data, fnames):
        n += 1
    dt = time.time() - t0
    print('Imported %d files in %d seconds' % (n, dt))

if __name__ == '__main__':
```

IMPORTING IN PARALLEL

```
import os, time, subprocess
from multiprocessing.dummy import Pool

def import_data(path):
    cmd = "COPY price FROM '%s' WITH DELIMITER ','" % path
    subprocess.call(['psql', '-c', cmd])

def main(datadir):
    t0 = time.time()
    n = 0
    fnames = (os.path.join(datadir, f) for f in os.listdir(datadir))
    for _ in Pool().map(import_data, fnames):
        n += 1
    dt = time.time() - t0
    print('Imported %d files in %d seconds' % (n, dt))
```

EXERCISES

- what happens if you have short csv files?
- what changes if you use psycopg2 instead of psql?
- it is better to use a connection per thread or a cursor per thread?
- what changes if you use processes instead of threads?
- how much the answers depend on the hardware you are using?
- how much the answers depend on the versions of the software you are using?

YOU *NEED* TO ADD INSTRUMENTATION TO YOUR PARALLELIZATION LIBRARY

this is what OpenQuake does

among many other things

Parallel importer using the OpenQuake libraries

```
import os, time,subprocess
from openquake.baselib.performance import Monitor
from openquake.baselib.parallel import Starmap
from openquake.commonlib.datastore import hdf5new

def import_data(path, monitor):
    cmd = "COPY price FROM '%s' WITH DELIMITER ','" % path
    subprocess.call(['psql', '-c', cmd])

def main(datadir):
    t0 = time.time()
    n = 0
    mon = Monitor('import_data', hdf5new().path)
    iterargs = ((os.path.join(datadir, f), mon.new())
                for f in os.listdir(datadir))
    for _ in Starmap(import_data, iterargs):
```

LET'S TRY

A LOT MORE THAN POOL.STARMAP

- it has monitoring and generates an .hdf5 file per calculation
- it has a set of commands for inspecting performance and data transfer
- it scales transparently from thread to processes to celery to ipython to the grid engine (to everything in principle)

IT TAKES CARE OF THE FINE POINTS

- it manages properly the case of failing tasks (clean shutdown and return the traceback even if the task is running on another machine)
- it uses `prctl` to kill properly the pool processes
- it forks before loading data and not after
- it works around a "memory leak" in celery
- tasks have a "suicide" functionality in case you are running out of memory
- ...

OTHER THINGS OPENQUAKE DOES FOR YOU

- the computations are organized in a SQLite database, there are dump and restore facilities, the ability to search calculations, delete calculations...
- implements a serialization protocol Python <-> hdf5 with methods `__toh5__` and `__fromh5__`
- has XML read-write facilities
- AccumDict
- ...

SHOULD YOU DOWNLOAD THE ENGINE?

- the code is on https://github.com/gem/oq-engine and free under the AGPL
- installing it is as as simple as

```
pip install openquake.engine
```

HOWEVER IT IS A FRAMEWORK :-(

- writing frameworks is good, inflicting them on others is not (spoken as an user of Zope, Plone, Pylons, SQLAlchemy, Django and several others frameworks)
- I don't want to enter in the number of the cursed people ;-)

MY TAKE ON FRAMEWORKS

- shipping frameworks is most of the times bad
- shipping libraries is most of the times good
- shipping ideas is always good

you are invited to steal ideas from the engine :-)

LESSONS LEARNED/1

- try significant examples: an approach which is fast for small examples can fail spectacularly for large calculations
- starting from an empty database can be totally different than starting from a big database
- performance on a cluster can be very different from a single machine (think of 4 vs 256 connections)

LESSONS LEARNED/2

- the data transfer is really important
- pickling/unpickling can be an issue
- replace Python objects and dictionaries with numpy.arrays as much as you can
- running out of memory early is a good thing
- a simple and fast *wrong* algorithm may be better than a complex and slow *correct* algorithm!

LESSONS LEARNED/3

- keep the concurrency layer independent from the low level parallelization technology
- all the concurrency in the engine is managed via a *(star)map* and nothing more (shared nothing architecture)
- sometimes it is convenient to allow the workers to read, but do not allow them to write
- profiling is good, but I do that 1% of the time
- instrument the running system instead