Parallel and distributed programming

How-to go parallel

Why?

- You have many independent tasks (easy)

or

- You want to accerelate single complex task (hard)

Recipe:

Turn the single complex task into many independent simple tasks, but how?

How-to go parallel

Why?

- You have many independent tasks (easy)

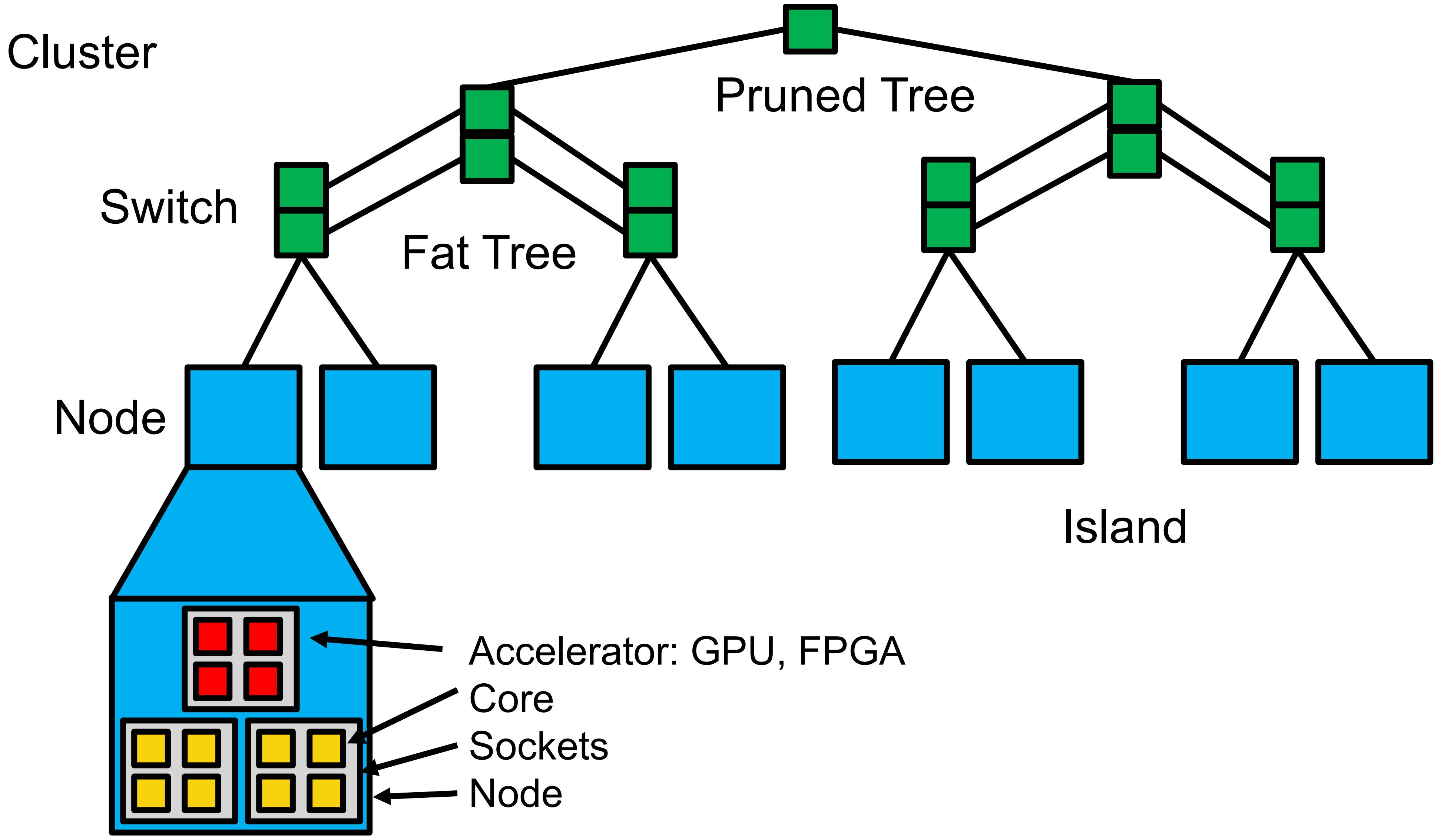
or

- You want to accerelate single complex task (hard)

Recipe:

Turn the single complex task into many independent simple tasks, but how?

LRZ from the system perspective



Parallel and Distributed Programming

- multiprocessing
- dask.distributed
- Mpi4py
- Scoop
- Ipython parallel

See also:

https://chryswoods.com/parallel_python/README.html

Global Interpreter Lock (GIL)

- The standard Python interpreter (called CPython) does not support the use of threads well.
- The CPython Python interpreter uses a “Global Interpreter Lock” to ensure that only a single line of a Python script can be interpreted at a time, thereby preventing memory corruption caused by multiple threads trying to read, write or delete memory in parallel.
- Because of the GIL, parallel Python is normally based on running multiple forks of the Python interpreter, each with their own copy of the script and their own GIL.

Embarrassingly parallel

- many independent processes (10 - 100.000)
- no communication between processes
- individual tasklist for each process
- private memory for each process
- results are stored in a large storage medium

Embarrassingly parallel (step-by-step)

- Take as example the following script

***myscript.sh*:**

```
#!/bin/bash
source /etc/profile.d/modules.sh
module load python
source activate py36
cd ~/mydir
python myscript.py
```

You can run it interactively by:

```
$ ./myscript.sh
```

Embarrassingly parallel (step-by-step)

Please do not block the login nodes with production jobs, but run the script in an interactive slurm shell:

```
$ salloc –pmpp2_inter –n1 myscript.sh
```

Change the last line in the script:

```
#!/bin/bash
source /etc/profile.d/modules.sh
module load python
source activate py36
cd ~/mydir
srun python myscript.py
```

Embarrassingly parallel (step-by-step)

Run multiple copies of the the script in an interactive slurm shell:

```
$ salloc –pmpp2_inter –n4 myscript.sh
```

You will get 4 times the output of the same run.

To use different input files you can use the environment variable:

`os.environ['SLURM_PROCID']` (it is set to 0,1,2,3,...)

Use this variable to select your workload.

Example:

```
$ salloc –pmpp2_inter –n2 srun
python –c "import os; os.environ['SLURM_PROCID']"
0
1
```

Embarrassingly parallel (step-by-step)

Run the script as slurm batch job:

```
$ sbatch -pmpp2_inter -n4 myscript.sh
```

You can put the options inside the slurm file:

```
#!/bin/bash
#SBATCH -pmpp2_inter
#SBATCH -n4
source /etc/profile.d/modules.sh
module load python
cd ~/mydir
srun python myscript.py
```

Embarrassingly parallel (step-by-step)

For serial (single node, multithreaded but not MPI) loads use the serial queue and add options for the runtime:

```
#!/bin/bash
#SBATCH --clusters=serial
#SBATCH -n4     # 4 tasks
#SBATCH --time=01:00:00 # 1hour
source /etc/profile.d/modules.sh
module load python
cd ~/mydir
srun python myscript.py

$ sbatch myscript.slurm
```

SLURM Job Arrays

If you want to send a large number of jobs then use Job Arrays.

```
$ sbatch -array=0-31 myscript.slurm
```

The variable SLURM_ARRAY_TASK_ID is set to the array index value. Get it in python via:

```
os.environ['SLURM_ARRAY_TASK_ID']
```

The maximum size of array job is 1000

Important SLURM commands

- List my jobs:

```
$ squeue –Mserial –u <uid>
```

- Cancel my job

```
$ scancel <jobid>
```

- Submit batch job

```
$ sbatch myscript.slurm
```

- Run interactive shell

```
$ salloc -n1 srun --pty bash -i
```

Ipython and ipcluster

The **`ipcluster`** command provides a simple way of starting a controller and engines in the following situations:

- When the controller and engines are all run on localhost. This is useful for testing or running on a multicore computer.
- When engines are started using the **`mpiexec`** command that comes with most MPI implementations
- When engines are started using the SLURM batch system

Using ipcluster

Starting ipcluster:

```
$ ipcluster start -n 4
```

Then start ipython and connect to the cluster:

```
$ ipython
In [1]: from ipyparallel import Client
In [2]: c = Client()
   ...: c.ids
   ...: c[:].apply_sync(lambda: "Hello, world!")
Out[2]: ['Hello, world!', 'Hello, world!', 'Hello,
world!', 'Hello, world!']
```

Ipcluster on SLURM

Create a parallel profile:

```
ipython profile create --parallel --profile=slurm
```

cd into ~/.ipython/profile_slurm/ and add the following:

ipcontroller_config.py:

```
c.HubFactory.ip = u'*'
c.HubFactory.registration_timeout = 600
```

ipengine_config.py:

```
c.IPEngineApp.wait_for_url_file = 300
c.EngineFactory.timeout = 300
```

Cont.

ipcluster_config.py:

```
c.IPClusterStart.controller_launcher_class =
'SlurmControllerLauncher'
c.IPClusterEngines.engine_launcher_class =
'SlurmEngineSetLauncher'
c.SlurmEngineSetLauncher.batch_template = """#!/bin/sh
#SBATCH --ntasks={n}
#SBATCH --clusters=serial
#SBATCH --time=01:00:00
#SBATCH --job-name=ipy-engine-
srun ipengine --profile-dir="{profile_dir}" --cluster-id=""
"""
```

Usage of ipcluster

Start a python shell and import the client function

```
>>> from ipyparallel import Client
```

Connect to the ipcluster

```
>>> c=Client(profile="slurm")
```

Generate a view on the cluster

```
>>> dview=c[:]
```

The view can now be used to perform parallel computations on the cluster

Usage of ipcluster

Run a string containing python code on the ipcluster:

```
>>> dview.execute("import time")
```

Run a single function and wait for the result:

```
>>> dview.apply_sync(time.sleep, 10)
```

Or return immediately:

```
>>> dview.apply_async(time.sleep, 10)
```

Map a function on a list by reusing the nores of the cluster:

```
>>> dview.map_sync(lambda x: x**10, range(32))
```

Defining parallel functions

Define a function that executes in parallel on the ipcluster:

```
In [10]: @dview.remote(block=True)
    ....: def getpid():
    ....:     import os
    ....:     return os.getpid()
    ....:
In [11]: getpid()
Out[11]: [12345, 12346, 12347, 12348]
```

Usage of ipcluster with NumPy

The @parallel decorator parallel functions, that break up an element-wise operations and distribute them, reconstructing the result.

```
In [12]: import numpy as np
In [13]: A = np.random.random((64,48))
In [14]: @dview.parallel(block=True)
    ....: def pmul(A,B):
    ....:     return A*B
```

Loadbalancing

You can create a view of the ipcluster that allows for loadbalancing of the work:

```
>>> lv=c.load_balanced_view()
```

This view can be used with all the above mentioned methods, auch as: execute, apply, map and the decorators.

The load balancer can even have different scheduling strategies like "Least Recently Used", "Plain Random", "Two-Bin Random", "Least Load" and "Weighted"

Example

```
In [3]: view = c[:]
In [4]: view.activate() # enable magics
# run the contents of the file on each engine:
In [5]: view.run('psum.py')
In [6]: view.scatter('a',np.arange(16,dtype='float'))
In [7]: view['a']
Out[7]: [array([ 0.,  1.,  2.,  3.]),
         array([ 4.,  5.,  6.,  7.]),
         array([  8.,   9.,  10.,  11.]),
         array([ 12.,  13.,  14.,  15.])]
In [7]: %px totalsum = psum(a)
Parallel execution on engines: [0,1,2,3]
In [8]: view['totalsum']
Out[8]: [120.0, 120.0, 120.0, 120.0]
```

Shared Memory (your laptop)

- a few threads working closely together (10-100)
- shared memory
- single tasklist (program)
- cache coherent non-uniform memory architecture aka ccNUMA
- results are kept in shared memory

multiprocessing

- Multiprocessing allows your script running multiple copies in parallel, with (normally) one copy per processor core on your computer.
- One is known as the master copy, and is the one that is used to control all of worker copies.
- It is not recommended to run a multiprocessing python script interactively, e.g. via ipython or ipython notebook.
- It forces you to write it in a particular way. All imports should be at the top of the script, followed by all function and class definitions.

multiprocessing

```
# all imports should be at the top of your script
import multiprocessing, sys, os
# all function and class definitions must be next
def sum(x, y):
    return x+y

if __name__ == "__main__":
    # You must now protect the code being run by
    # the master copy of the script by placing it

    a = [1, 2, 3, 4, 5]
    b = [6, 7, 8, 9, 10]

    # Now write your parallel code... etc. etc.
```

Multiprocessing pool

```
from multiprocessing import Pool, current_process

def square(x):
        print("Worker %s calculating square of %d" % (current_process().pid, x))
    return x*x

if __name__ == "__main__":
    nprocs = 2

    # print the number of cores
    print("Number of workers equals %d" % nprocs)

    # create a pool of workers
    pool = Pool(processes=nprocs)

    # create an array of 10 integers, from 1 to 10
    a = range(1,11)

    result = pool.map( square, a )
    total = reduce( lambda x,y: x+y, result )

    print("The sum of the square of the first 10 integers is %d" % total)
```

Multiprocessing futures

- Use futures and a context manager:

```
from concurrent.futures import ThreadPoolExecutor
with ThreadPoolExecutor(max_workers=1) as ex:
    future = ex.submit(pow, 323, 1235)
    print(future.result())
```

scoop

- Scoop is a developing third-party Python module that supports running parallel Python scripts across clouds, distributed compute clusters, HPC machines etc.
- `conda install scoop`
  if you are using anaconda python
- `pip install scoop`
  if you have installed pip
- `easy_install scoop`
  in all other cases (i.e. if the other two commands don’t work)

scoop

```
from scoop import futures

def product(x, y):
    return x*y

def sum(x, y):
    return x+y

if __name__ == "__main__":

    a = range(1,101)
    b = range(101, 201)

    results = futures.map(product, a, b)
    total = reduce(sum, results)

    print("Sum of the products equals %d" % total)
```

Running scoop

- Run this script using the command

```
$ python -m scoop mapreduce.py
```

- You need to use -m scoop so that Scoop has time to set up the distributed cluster before running your script.

```
$ python -m scoop --hostfile hostfile script.py
```

Caveats

Scoop provides a very similar interface as multiprocessing, with the same caveats, requirements and restrictions. For example:

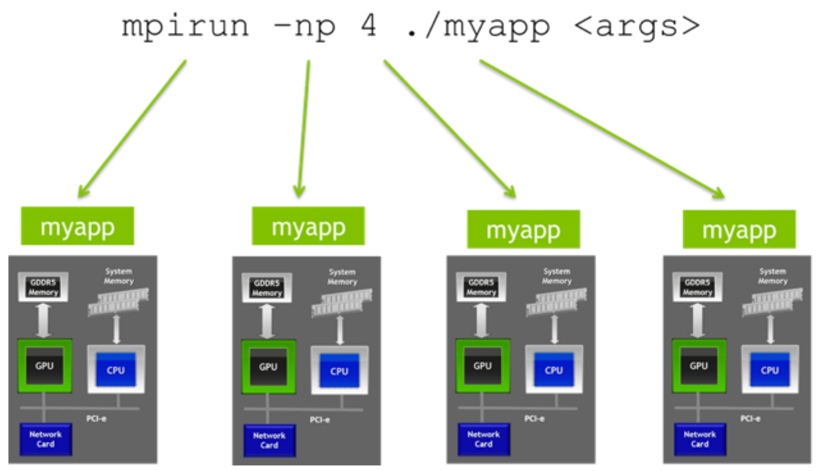
- You must ensure that all use of Scoop is protected within an `if __name__ == "__main__"`
- You must import all modules and declare all functions at the top of your script, before the `if __name__ == "__main__"`
- Scoop does not yet support anonymous (lambda) functions, again because of Python’s poor support for pickling those functions. Hopefully this will change soon.

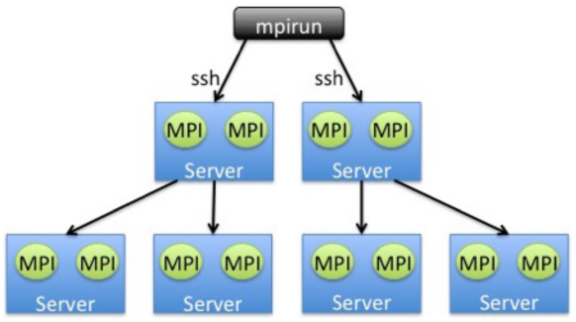
Message Passing

- many independent processes (10 - 100.000)
- one tasklist for all (program)
- everyone can talk to each other (in principle)
- private memory
- needs communication strategy in order to scale out
- very often: nearest neighbor communication
- beware of deadlocks!

mpi4py

```
$ mpiexec -n 4 python myapp.py

from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
if rank == 0:
    data = {'a': 7, 'b': 3.14}
    comm.send(data, dest=1, tag=11)
elif rank == 1:
    data = comm.recv(source=0, tag=11)
```





Worker queue

- many independent processes (10 - 100.000)
- central task scheduler (database)
- private memory for each process
- results are sent back to task scheduler
- rescheduling of failed tasks possible

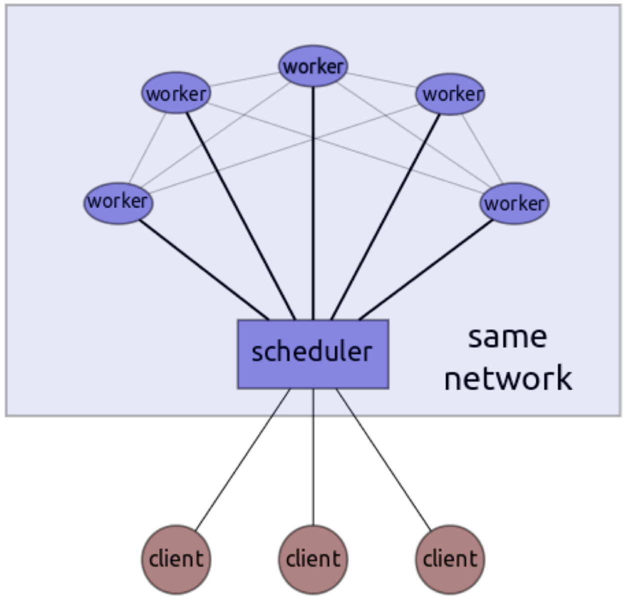
dask.distributed

- Start a scheduler which organizes the computing tasks

```
$ dask-scheduler
```

- dask workers

```
$ dask-worker localhost:8786
$ dask-ssh host.domain
$ mpirun --np 4 dask-mpi
$ dask-ec2
$ dask-kubernetes
$ dask-drmaa
```

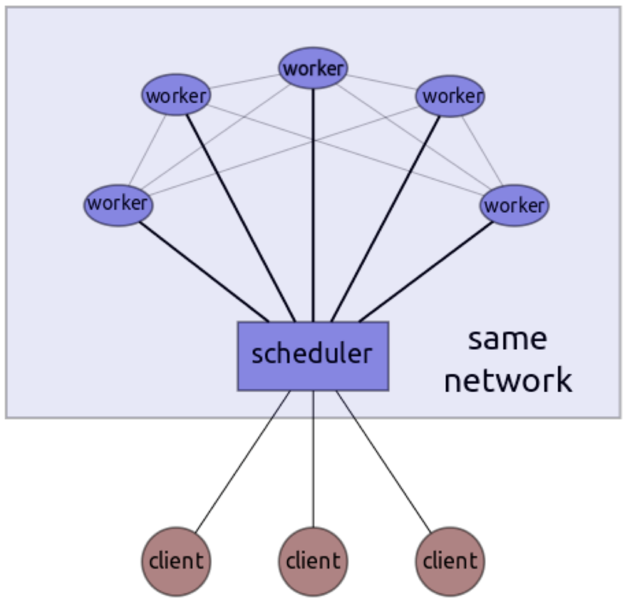


dask.distributed

- Start a client

```
>>> from distributed import Client
>>> client = Client('localhost:8786')
```

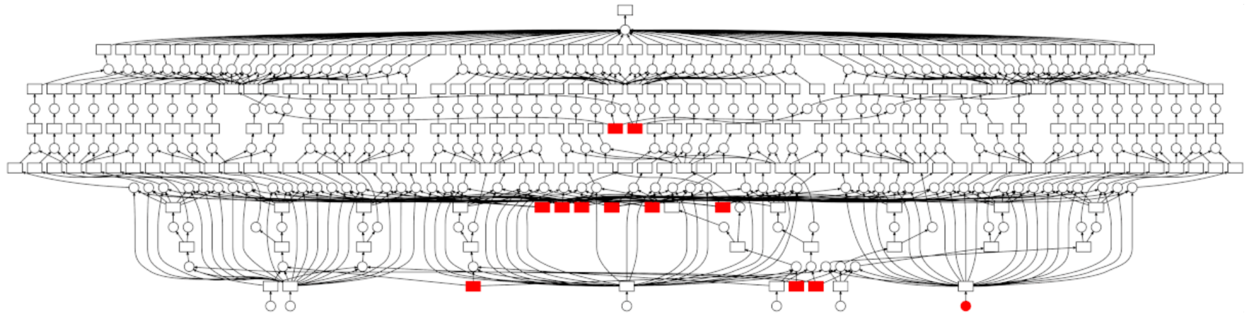
now all dask operations will be distributed to the scheduler which distributes them to the cluster



dask execution graph

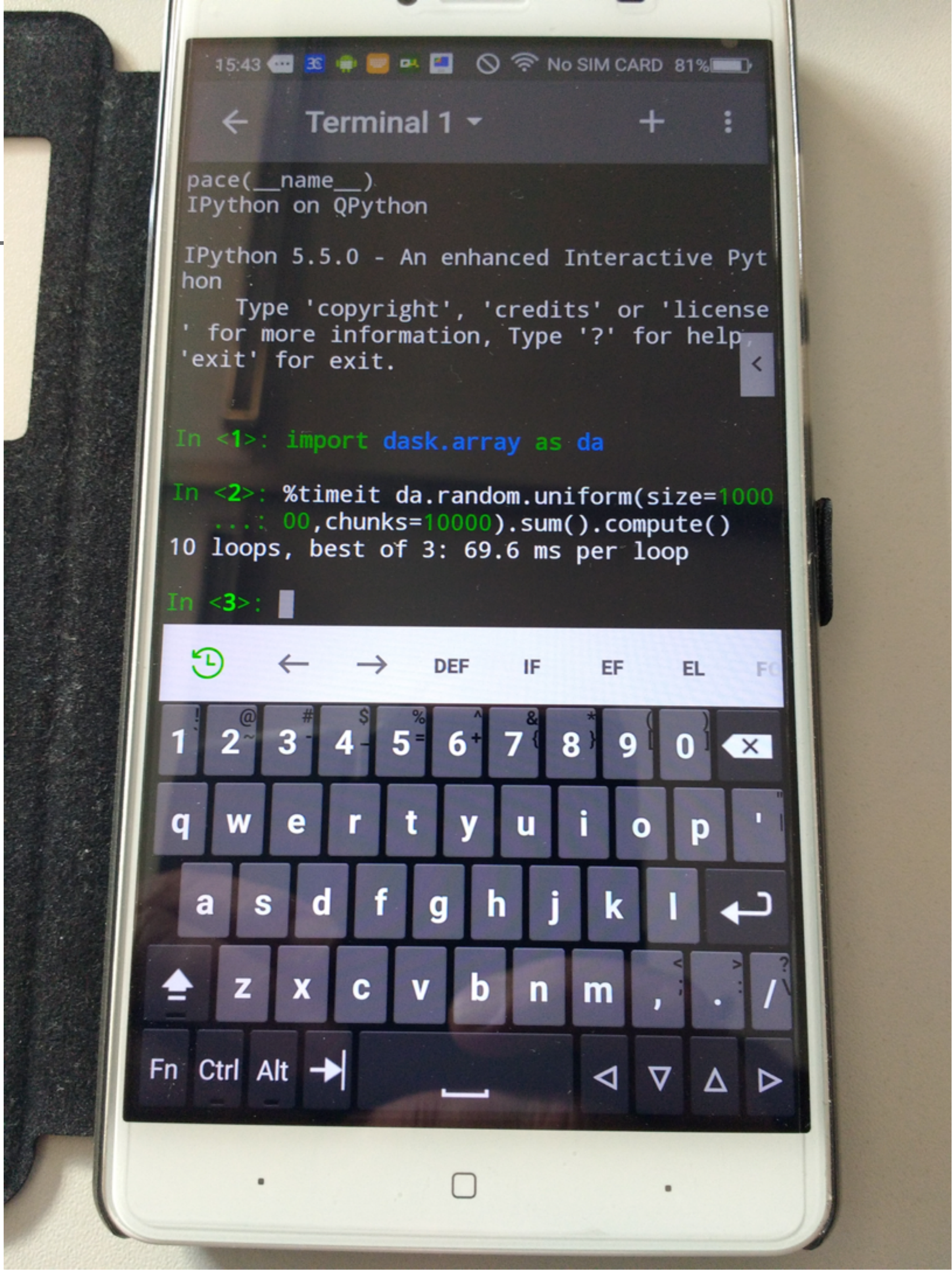
```
>>> a=da.random.uniform(size=1000,chunks=100)
>>> b=a.sum()
>>> c=a.mean()*a.size
>>> d=b-c
>>> d.compute()
```

the computation starts at the last command. If you have a dask cluster then all computations can be distributed to the cluster.



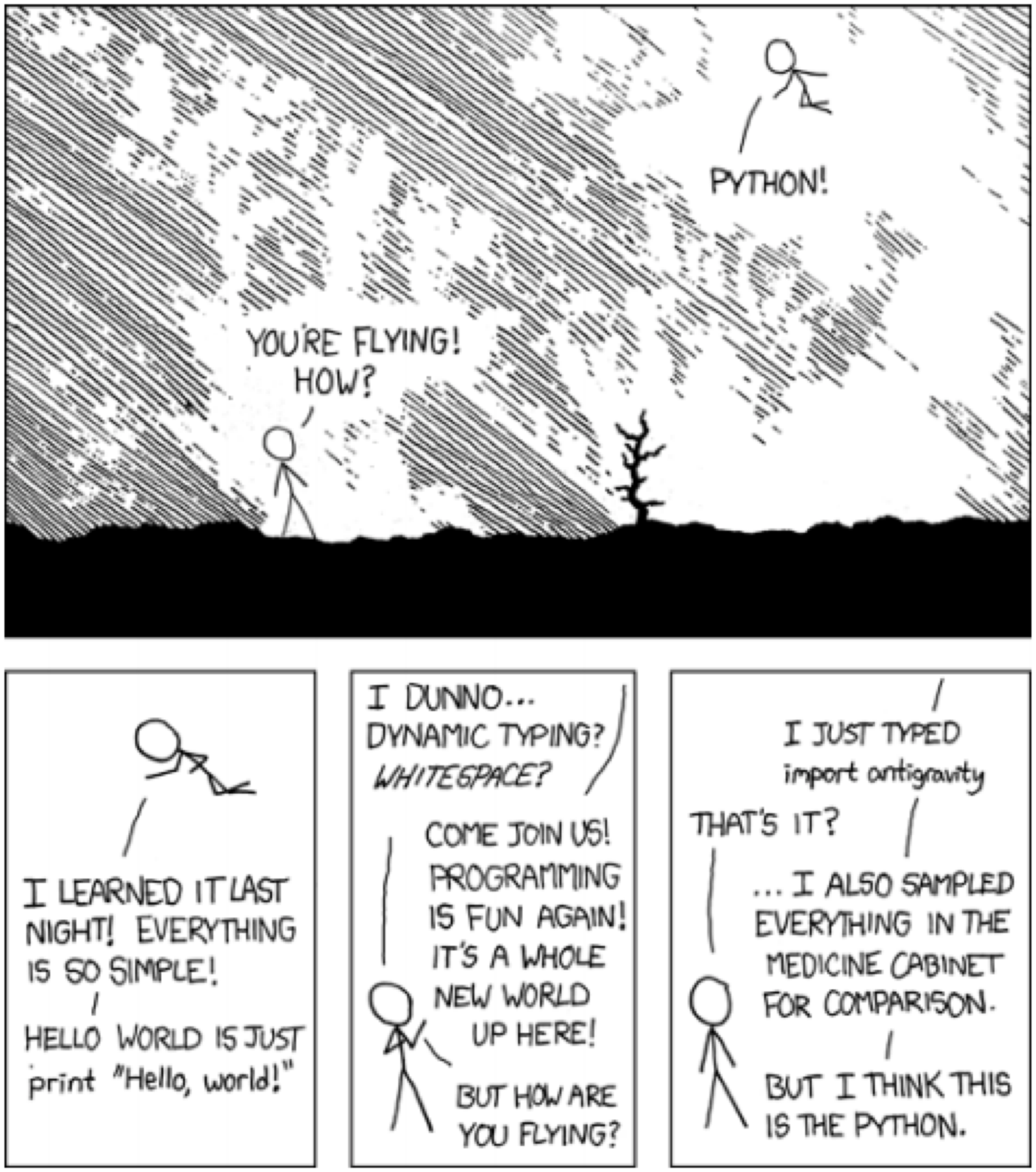
DASK mobile

- install qpython
- open pip console
- install dask
- install toolz
- install ipython



```
pace(__name__)
IPython on QPython

IPython 5.5.0 - An enhanced Interactive Pyt
hon
    Type 'copyright', 'credits' or 'license
' for more information, Type '?' for help,
'exit' for exit.

In <1>: import dask.array as da

In <2>: %timeit da.random.uniform(size=1000
   ...: 00,chunks=10000).sum().compute()
10 loops, best of 3: 69.6 ms per loop

In <3>:
```

The End: XKCD


PYTHON!
YOU'RE FLYING! HOW?
I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "Hello, world!"
I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN! IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?
I JUST TYPED import antigravity
THAT'S IT?
... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.

Course Evaluation

Please visit
https://survey.lrz.de/index.php/6939
73
and rate this course!

Your feedback is highly appreciated!
Thank you!