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?

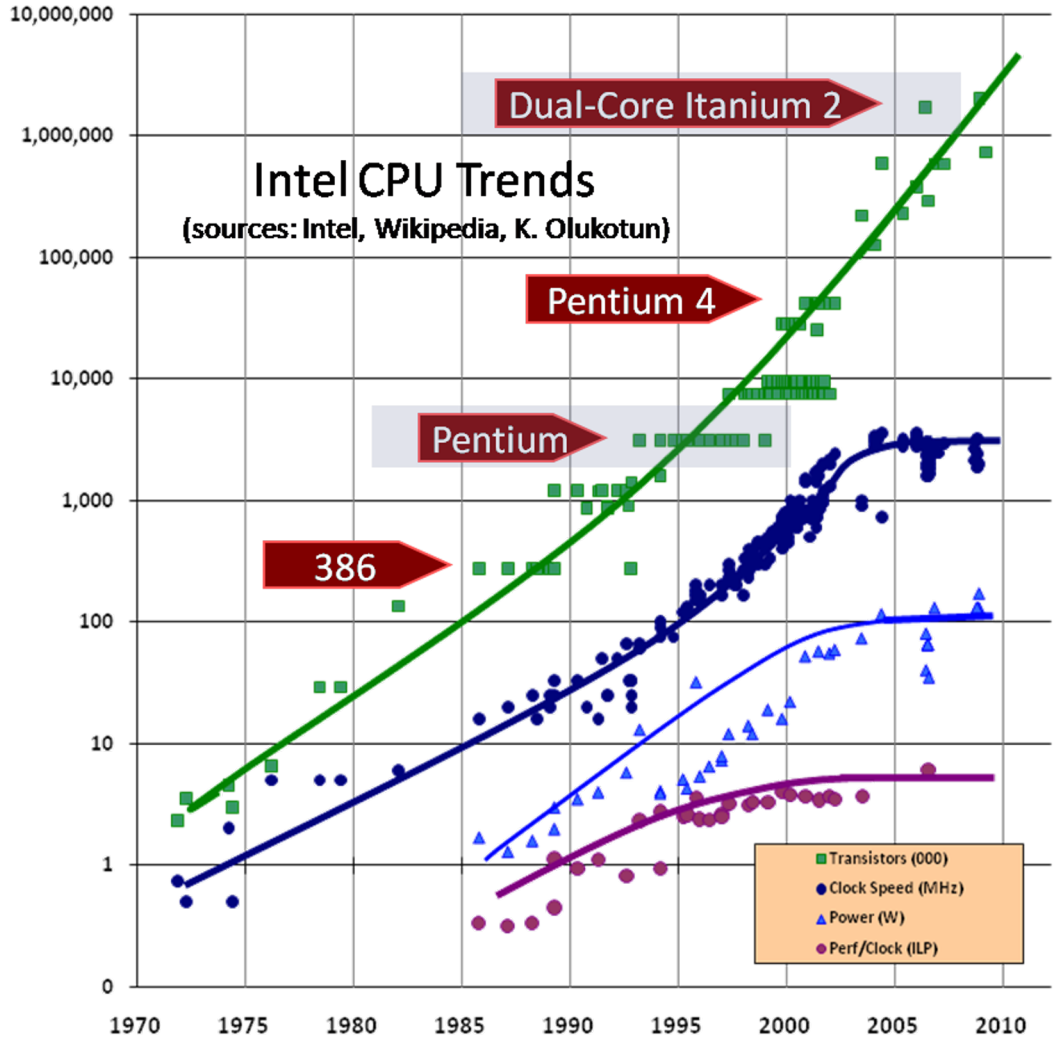
Why parallel programming?

End of the free lunch

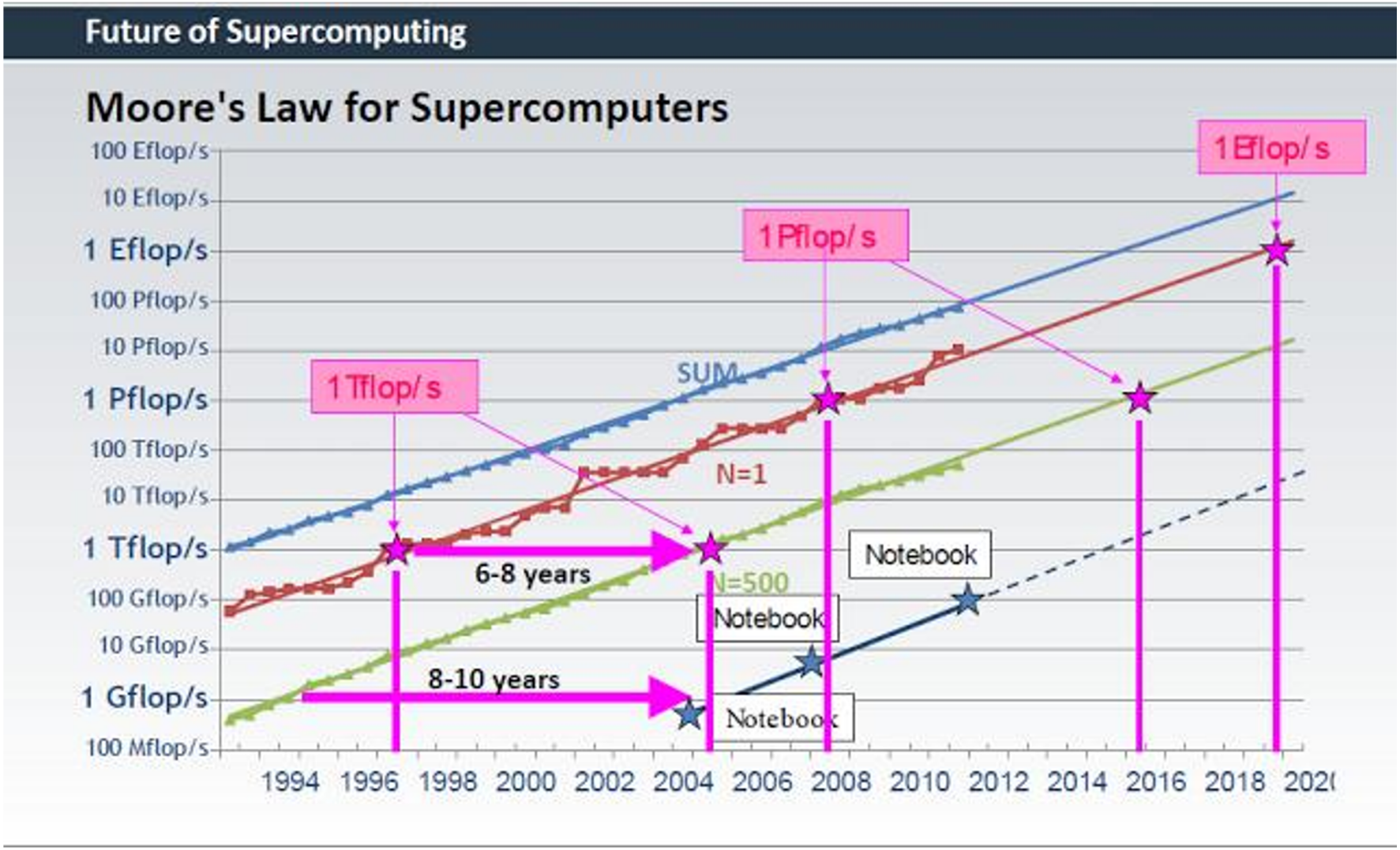
Moore's law means no longer faster processors, only more of them. But beware!

2 x 3 GHz < 6 GHz

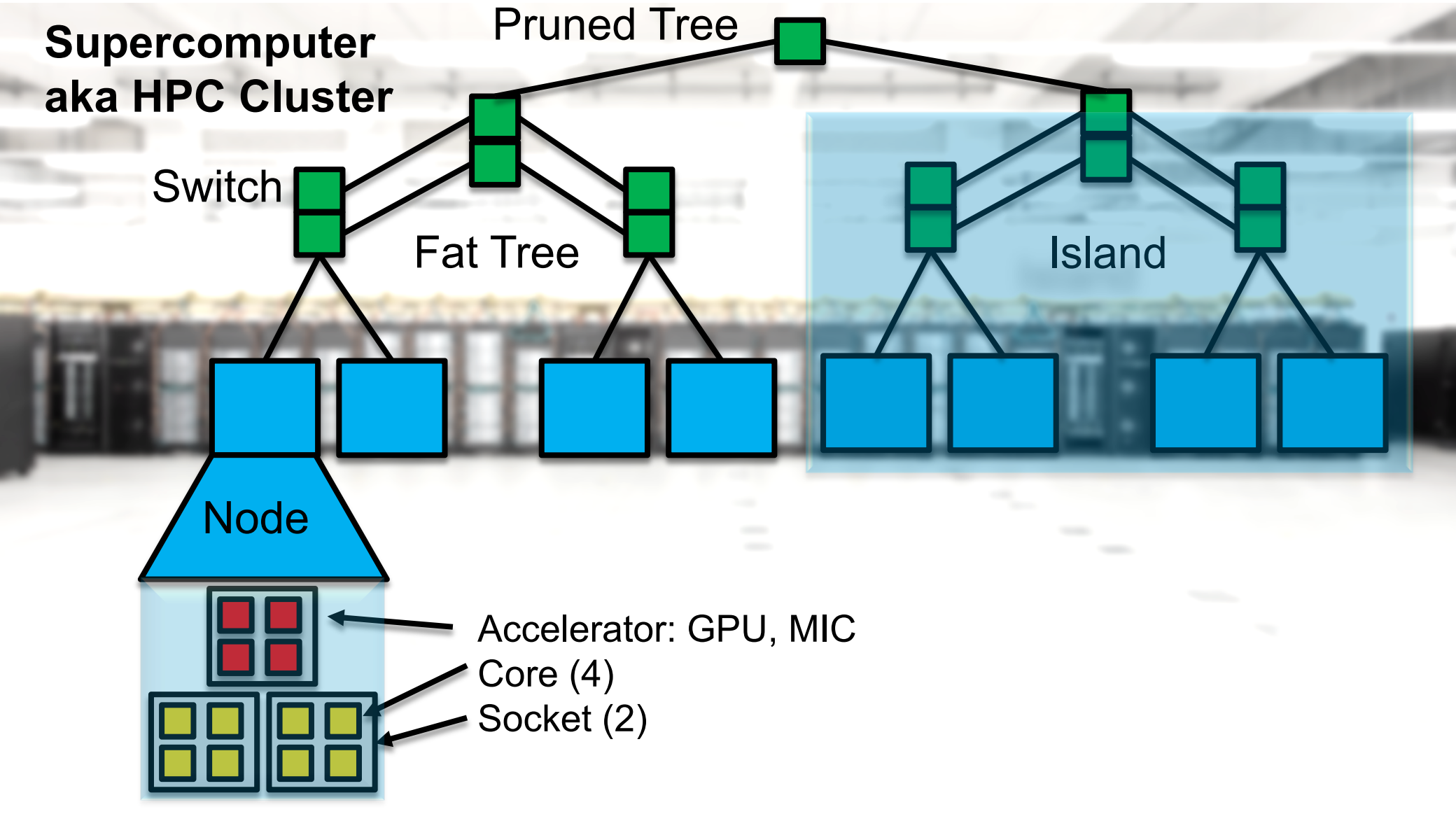
(cache consistency, multi-threading, etc)



Supercomputer scaling



Supercomputer Layout



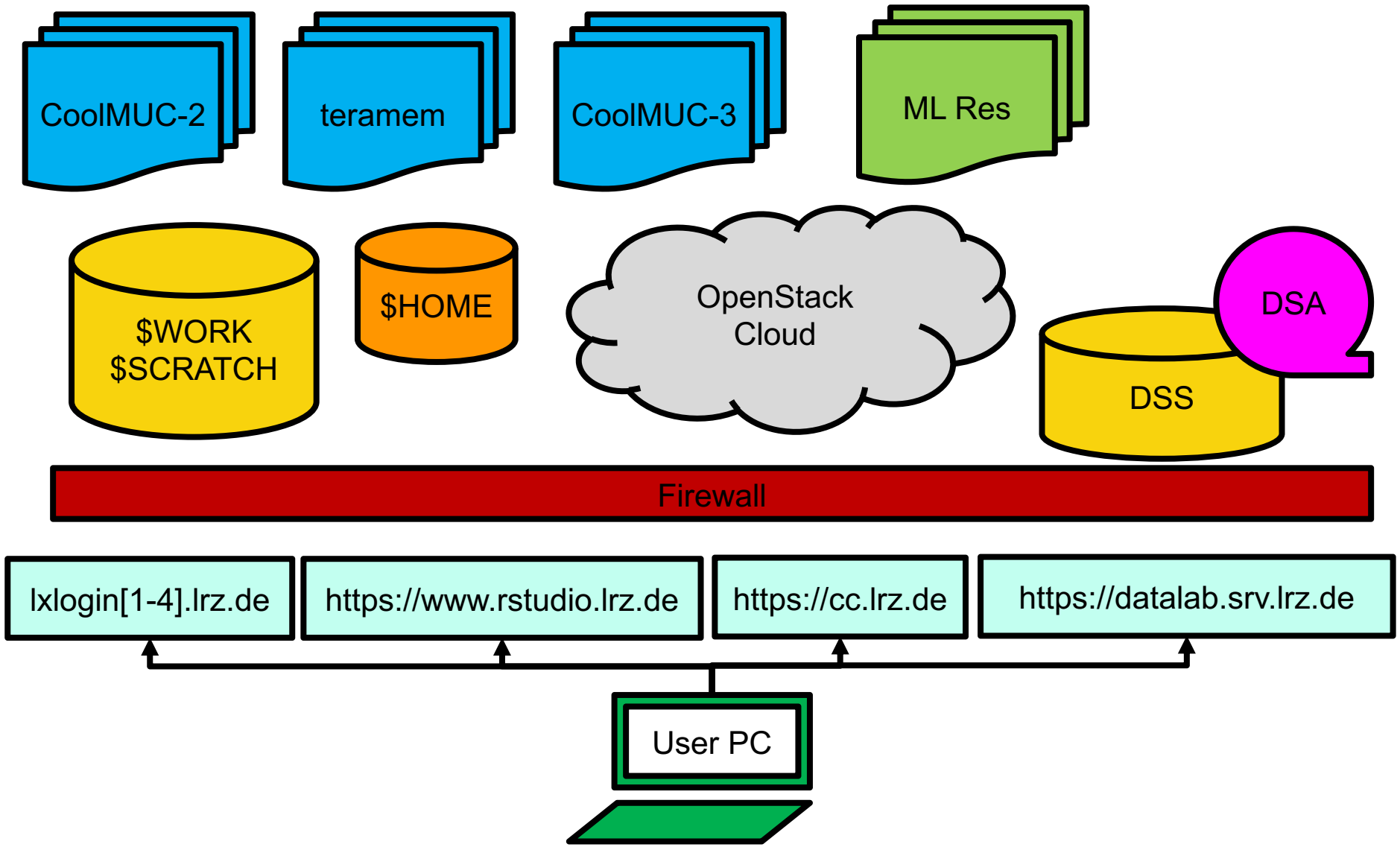
Latency (can kill your program)

translates to

Getting data from:	
CPU register	1 ns
L2 cache	10 ns
memory	80 ns
network(IB)	200 ns
GPU(PCIe)	50.000 ns
harddisk	500.000 ns

Getting food from the...	
fridge	10s
microwave	100s ~ 2min
pizza service	800s ~ 15min
city mall	2000s ~ 0.5h
mum sends cake	500.000s ~ 1 week
grown in garden	5.000.000s ~ 2 months

LRZ from the user perspective



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 ~/miniconda39/bin/activate py39
cd ~/mydir
python3 myscript.py
```

You can make it run interactively by:

```
$ chmod +x ./myscript.sh
```

then

```
$ ./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 -pcm2_inter --ntasks=1 myscript.sh
```

Change the last line in the script:

```
#!/bin/bash
source ~/miniconda39/bin/activate py39
cd ~/mydir
srun python3 myscript.py
```

Embarrassingly parallel (step-by-step)

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

```
$ salloc –pcm2_inter –-ntasks=4 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 –pcm2_inter –ntasks=2 srun python –c "import os;
print(os.environ['SLURM_PROCID'])"
0
1
```

Embarrassingly parallel (step-by-step)

Run the script as slurm batch job:

```
$ sbatch -pcm2_inter --ntasks=4 myscript.sh
```

You can put the options inside the slurm file:

```
#!/bin/bash
#SBATCH -pcm2_inter
#SBATCH --ntasks=4
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
```

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

multithreading

- 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.

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.

-> advice: don't use it, it is a pain to debug

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.
```

Warning! No interactive usage (shell, jupyter, IDLE,...)

Note: Functionality within this package requires that the `__main__` module be importable by the children. This is covered in Programming guidelines however it is worth pointing out here. This means that some examples, such as the `multiprocessing.pool.Pool` examples will not work in the interactive interpreter. For example:

```
>>> from multiprocessing import Pool
>>> p = Pool(5)
>>> def f(x):
...     return x*x
...
>>> with p:
...   p.map(f, [1,2,3])
Process PoolWorker-1:
Process PoolWorker-2:
Process PoolWorker-3:
Traceback (most recent call last):
AttributeError: 'module' object has no attribute 'f'
AttributeError: 'module' object has no attribute 'f'
AttributeError: 'module' object has no attribute 'f'
```

(If you try this it will actually output three full tracebacks interleaved in a semi-random fashion, and then you may have to stop the parent process somehow.)

ipcluster

Starting ipcluster with 3 workers:

```
$ ipcluster start -n 3
```

Then start ipython and connect to the cluster:

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

lrz

Usage of ipcluster

Run a string containing python code on the ipcluster:

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

Run a single function and wait for the result:

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

Or return immediately:

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

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

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

Defining parallel functions

Define a function that executes in parallel on the ipcluster:

```
In [10]: @c.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 defines 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]: @c.parallel(block=True)
    ....: def pmul(A,B):
    ....:     return A*B
```

Message Passing

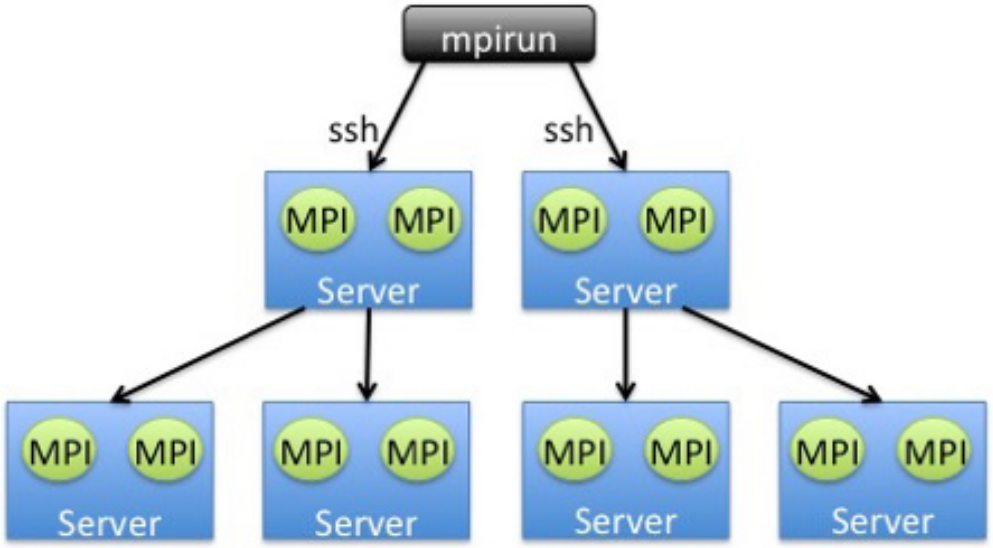
- many independent processes (10 - 100.000)
- one tasklist for all (program)
- everyone can talk to each other (in principle)
- private memory
- needs communication strategy in order to scale out
- very often: nearest neighbor communication
- beware of deadlocks!

MPI

- cluster of workers
- message passing interface MPI
- **mpirun** starts the same program on all workers

```
$ mpirun -n workers myapp.exe
```

communication via
mpi protocol
(send/receive)



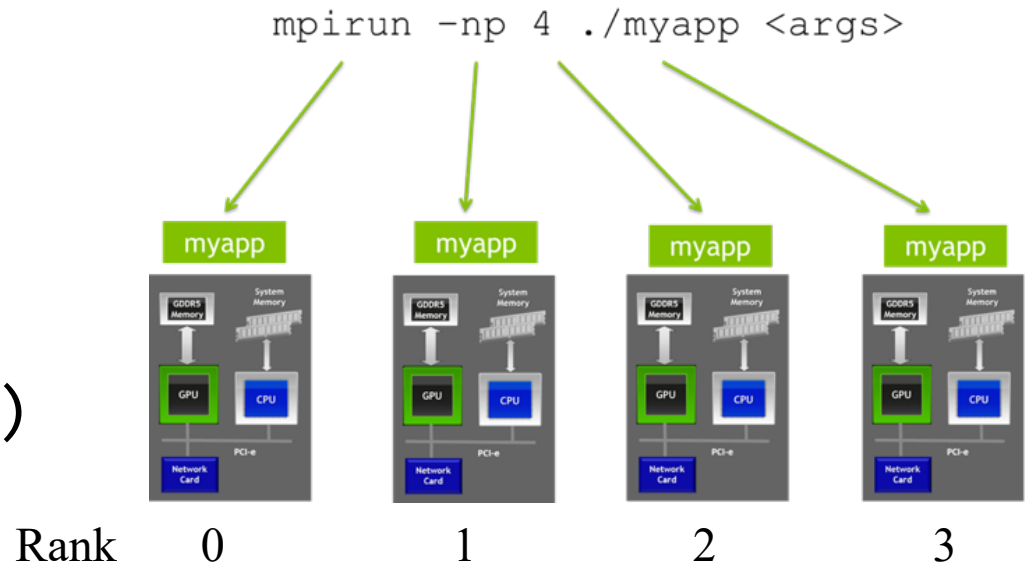
mpi4py - Communication

Get communicator:

```
>>> comm = MPI.COMM_WORLD
```

Get rank of worker:

```
>>> rank = comm.Get_rank()
```



Send Data (blocking):

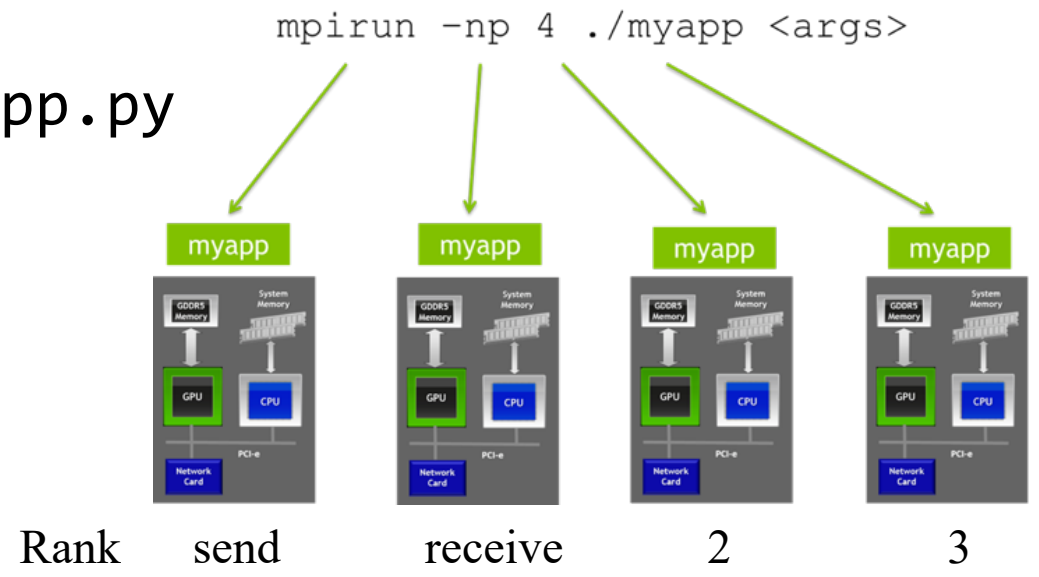
```
>>> comm.send(data, dest=1)
```

Receive Data (blocking):

```
>>> data = comm.recv(source=0)
```

mpi4py

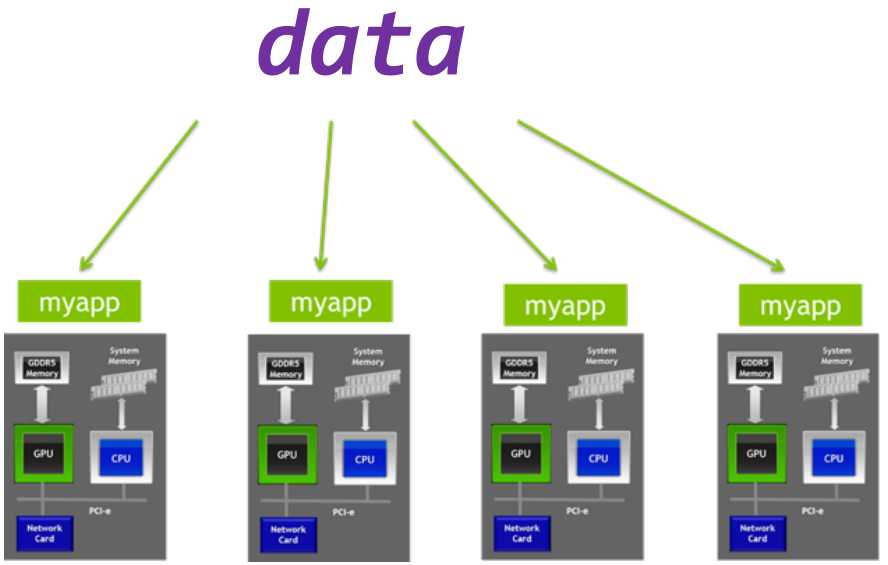
```
$ mpiexec -n 2 python myapp.py
```



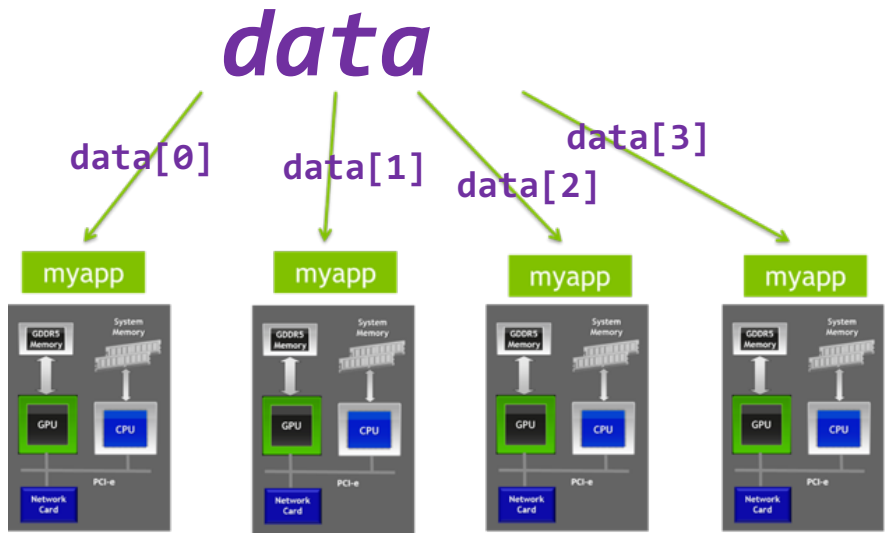

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = [2,4,7]
    comm.send(data, dest=1, tag=11)
elif rank == 1:
    data = comm.recv(source=0, tag=11)
    print(data)
```

mpi4py - Broadcast

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = [2,4,7]
else:
    data = None

data = comm.bcast(data, root=0)
```



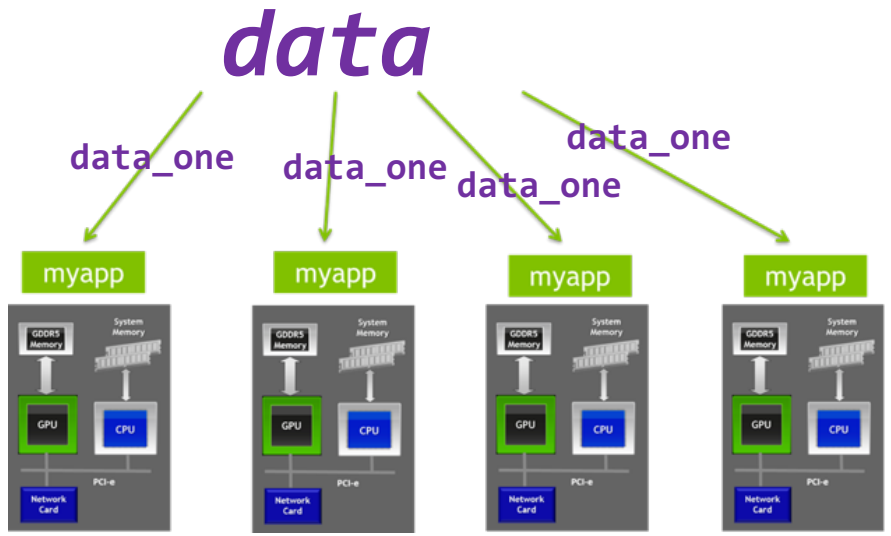
mpi4py - Scatter




```
from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

if rank == 0:
    data = [(i+1)**2 for i in range(size)]
else:
    data = None

data = comm.scatter(data, root=0)
```

mpi4py - Gather

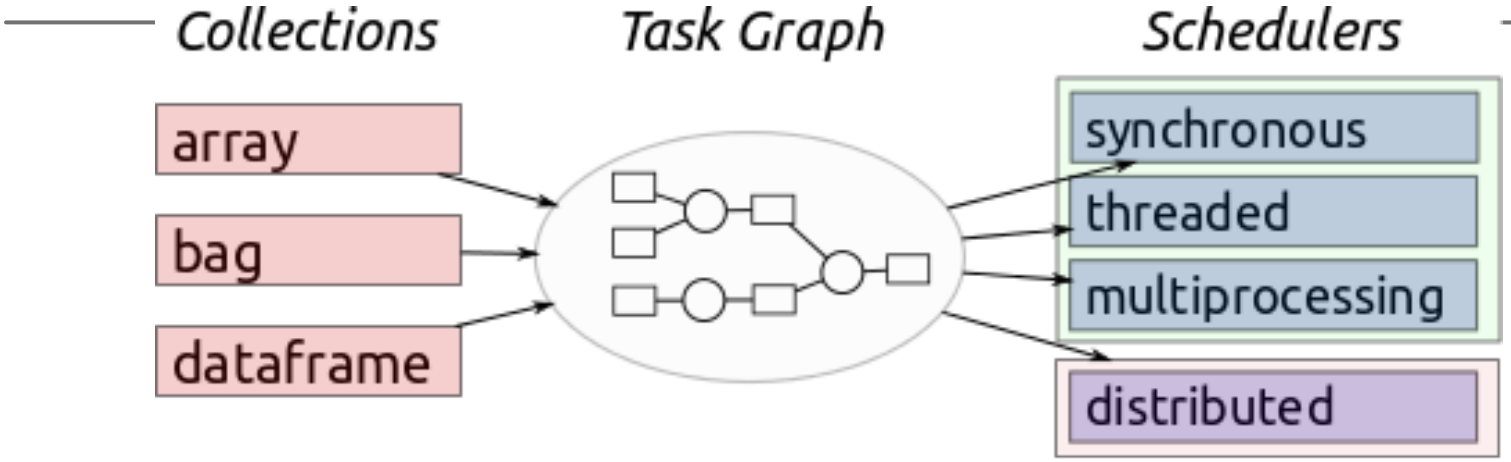
```
from mpi4py import MPI


comm = MPI.COMM_WORLD
rank = comm.Get_rank()

data_one = (rank+1)**2

data = comm.gather(data_one, root=0)
```



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

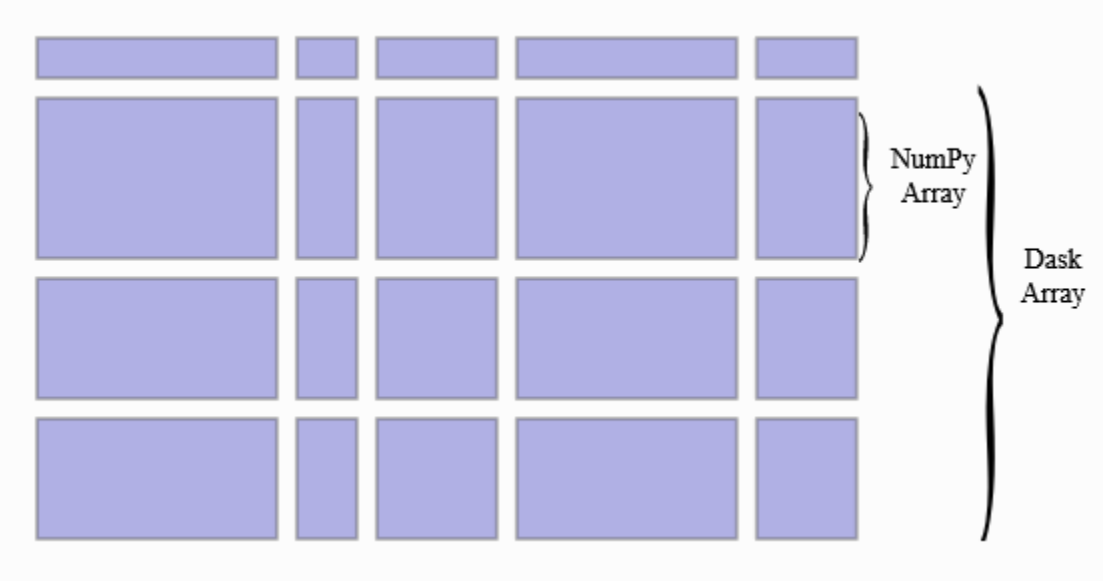
dask



Familiar: Provides parallelized NumPy array and Pandas DataFrame objects
Flexible: Provides a task scheduling interface for more custom workloads and integration with other projects.
Native: Enables distributed computing in Pure Python with access to the PyData stack.
Fast: Operates with low overhead, low latency, and minimal serialization necessary for fast numerical algorithms
Scales up: Runs resiliently on clusters with 1000s of cores
Scales down: Trivial to set up and run on a laptop in a single process, even on a smartphone running android
Responsive: Designed with interactive computing in mind it provides rapid feedback and diagnostics to aid humans

dask.array

- dask arrays are composed of numpy arrays.
- the subarrays can live in the same process or in another process on a different node
- dask has a scheduler which distributes the work on a whole cluster if needed



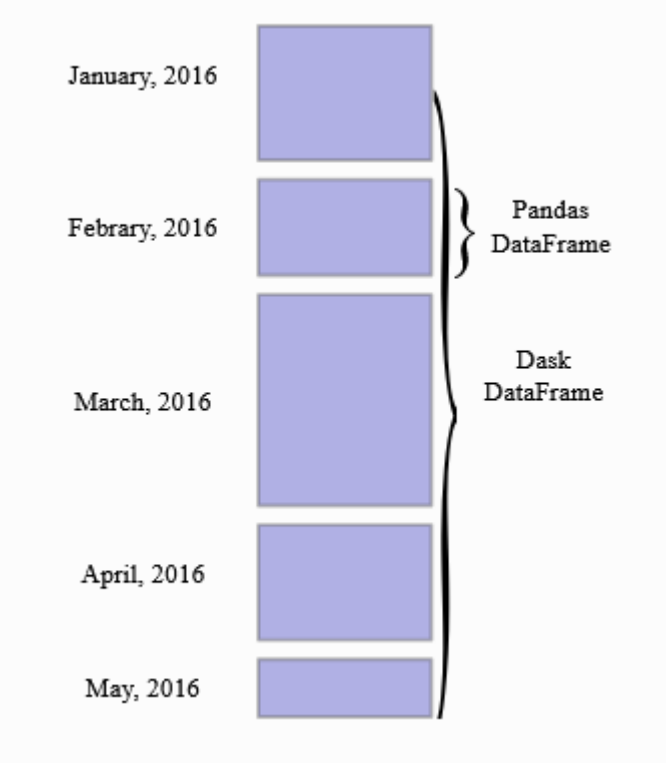

```
>>> import dask.array as da
>>> a=da.random.uniform(size=1000, chunks=100)
```

https://docs.dask.org/en/latest/array-api.html

dask.dataframe

- like dask.arrays uses numpy arrays, dask.dataframe uses pandas
- dask.dataframes can be distributed over a cluster of nodes and operations on them are scheduled by the dask scheduler

```
>>> import dask.dataframe as dd
>>> df=dd.read_csv('2014-*.csv')
```



dask vs numpy and pandas

Dask can be used like Numpy (often)

```
import numpy as np
f = h5py.File('myfile.hdf5')
x = np.array(f['/small-data'])

x - x.mean(axis=1)
```

```
import dask.array as da
f = h5py.File('myfile.hdf5')
x = da.from_array(f['/big-data'],
                  chunks=(1000, 1000))
x - x.mean(axis=1).compute()
```

Dask can be used like Pandas (often)

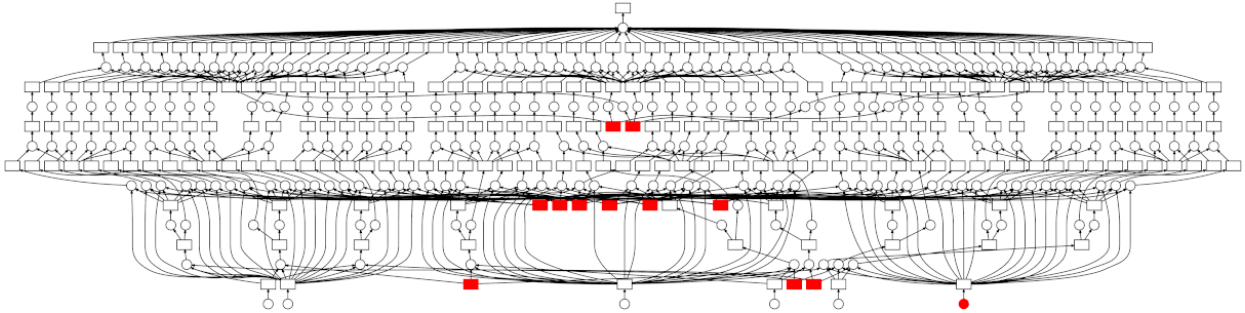
```
import pandas as pd
df = pd.read_csv('2015-01-01.csv')
df.groupby(df.user_id).value.mean()
```

```
import dask.dataframe as dd
df = dd.read_csv('2015-*-*.csv')
df.groupby(df.user_id).value.mean().compute()
```

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.



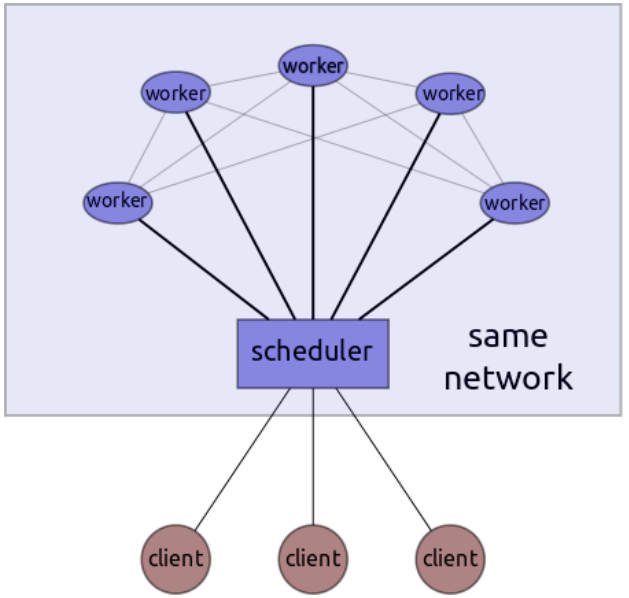
dask.distributed

- 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
```

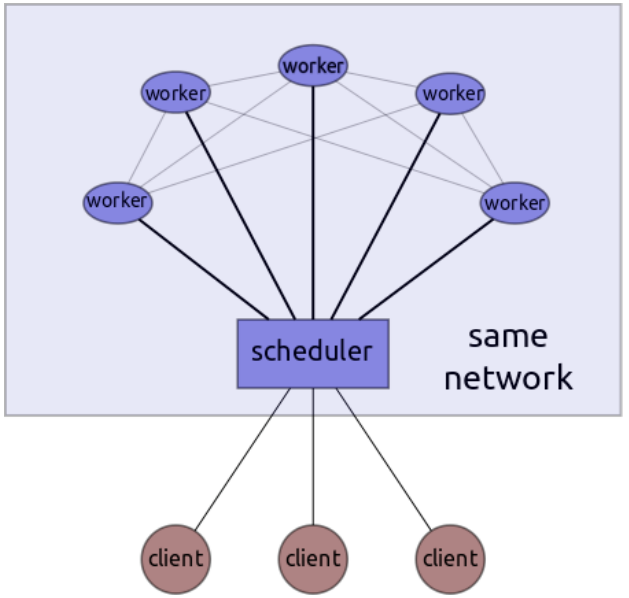


dask.distributed

- Start a client

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

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



When to use dask

DASK

Dask DataFrame is used in situations where Pandas or Numpy is commonly needed, usually when they fail due to data size or computation speed:

- Manipulating large datasets, even when those datasets don’t fit in memory
- Accelerating long computations by using many cores
- Distributed computing on large datasets with standard Pandas operations like groupby, join, and time series computations

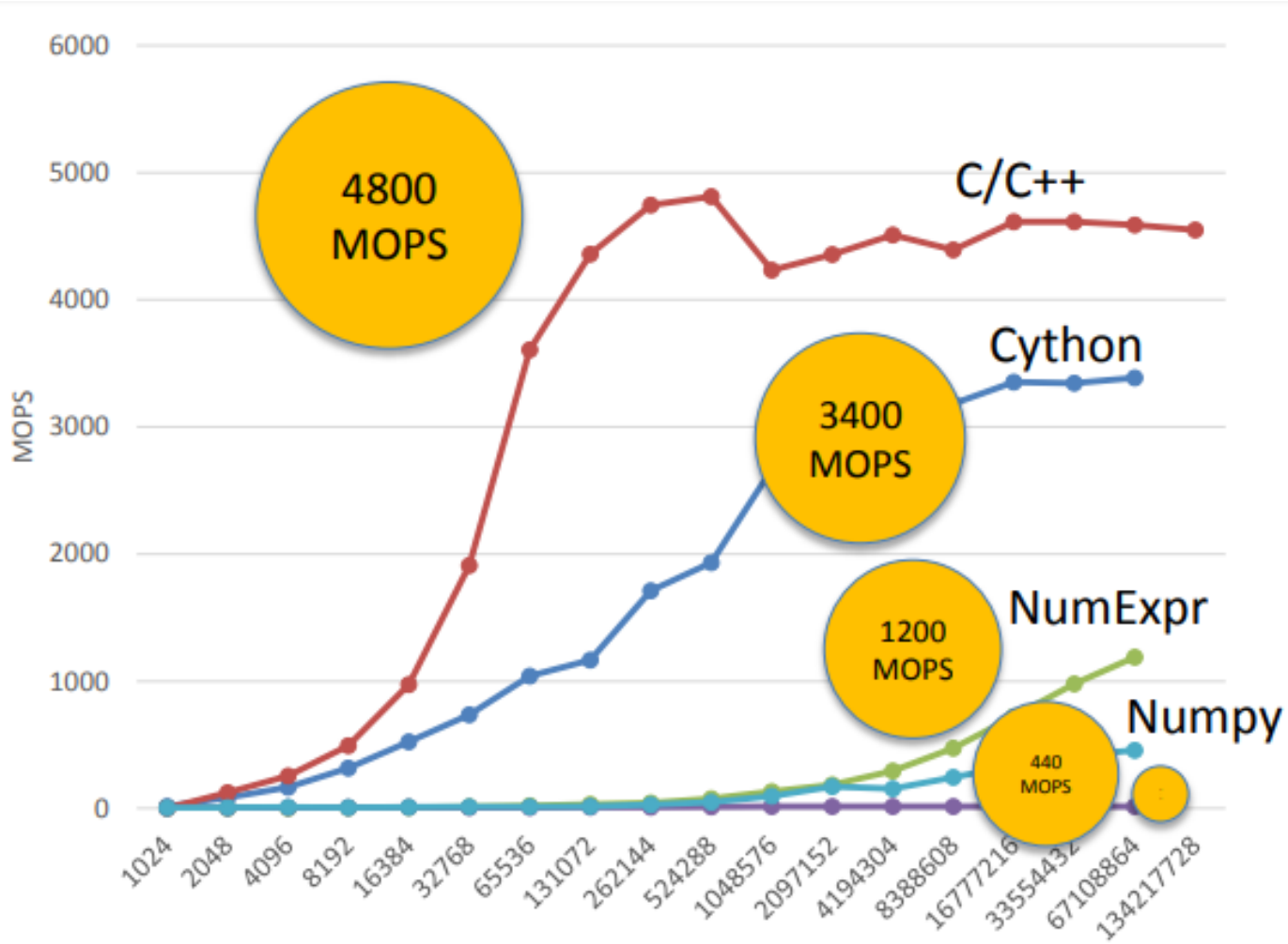
When NOT to use dask

DASK

Dask DataFrame may not be the best choice in the following situations:

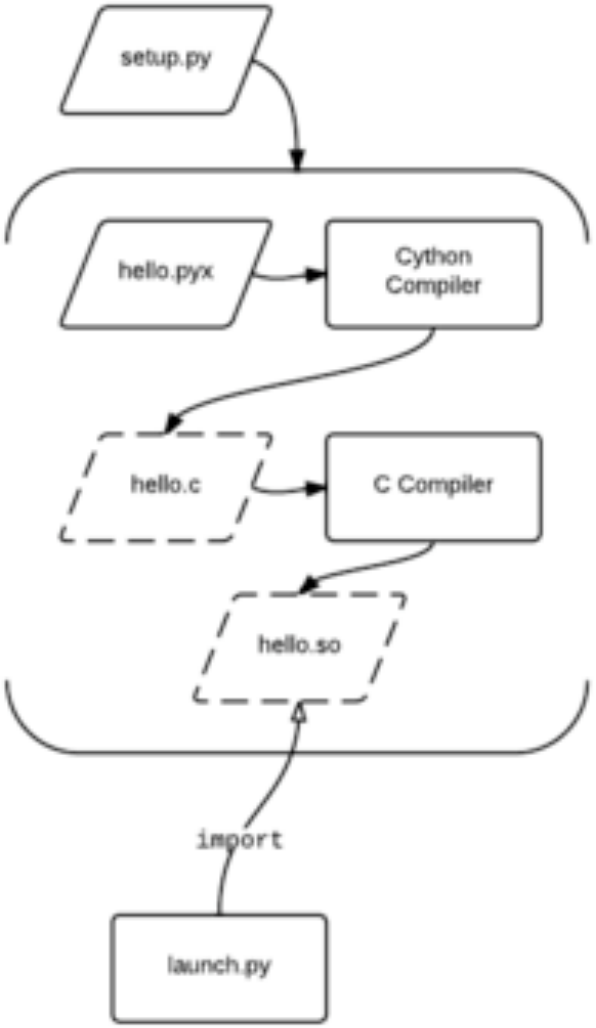
- If your dataset fits comfortably into RAM on your laptop, then you may be better off just using Pandas. There may be simpler ways to improve performance than through parallelism
- If your dataset doesn't fit neatly into the Pandas tabular model, then you might find more use in dask.bag or dask.array
- If you need functions that are not implemented in Dask DataFrame, then you might want to look at dask.delayed which offers more flexibility
- If you need a proper database with all that databases offer you might prefer something like Postgres

Python numerical libraries



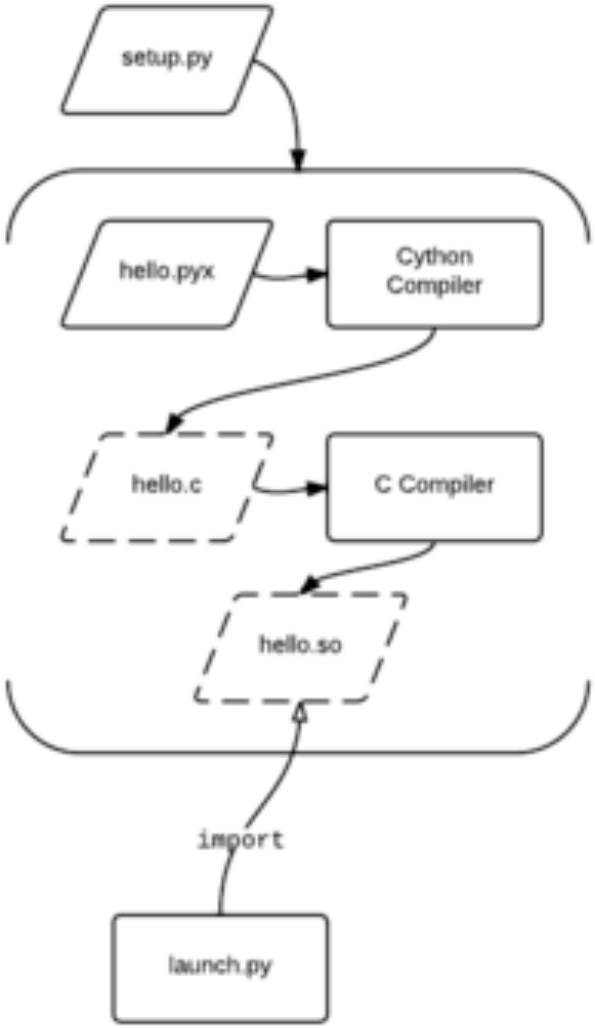
cython

- superset of the Python programming language
- designed to give C-like performance
- code is mostly written in Python
- compiled language that generates CPython extension modules
- extension modules can then be loaded and used by regular Python code using the import statement
- Cython files have a .pyx extension



cython

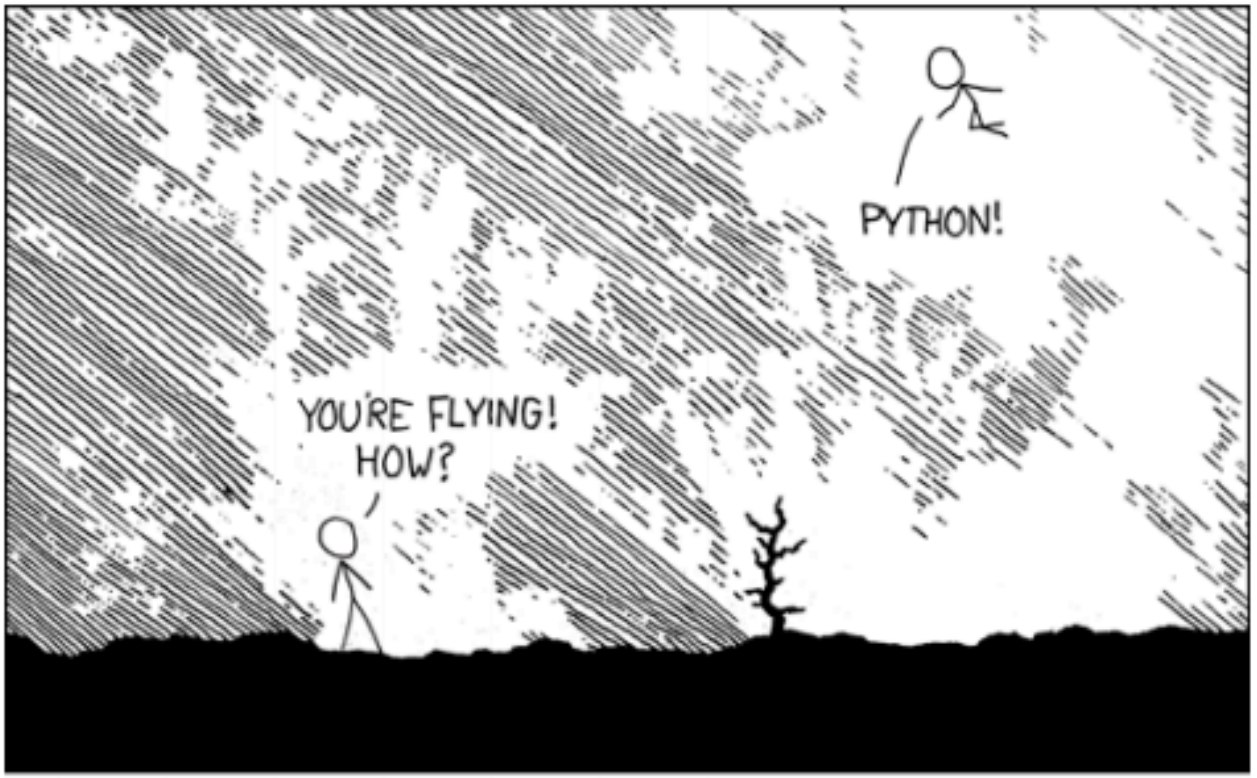
```
hello.pyx:
def say_hello():
    print "Hello World!"

launch.py:
import hello
hello.say_hello()
```



cython in ipython/jupyter notebooks

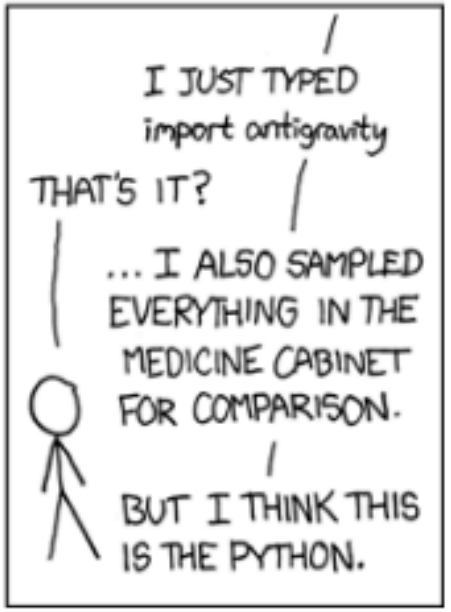
```
In [1]: %load_ext Cython

In [2]: %%cython
   ...: def f(n):
   ...:     a = 0
   ...:     for i in range(n):
   ...:         a += i
   ...:     return a
   ...:
   ...: cpdef g(int n):
   ...:     cdef int a = 0, i
   ...:     for i in range(n):
   ...:         a += i
   ...:     return a
   ...:

In [3]: %timeit f(1000000)
42.7 ms ± 783 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [4]: %timeit g(1000000)
74 µs ± 16.6 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

The End: XKCD


YOU'RE FLYING! HOW?
PYTHON!


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/693973
and rate this course!

Your feedback is highly appreciated!
Thank you!