Optimization & Parallelization

Karl Kosack
CEA Paris-Saclay

ESCAPE School, June 2021

Your code is slow.

Now what?

Topics we will cover

Optimizing your code:

- With Memoization
- With NumPy
- With Numba
- With Cython

Parallelizing your code:

- On a single machine with multiple cores
- On multiple machines

Optimization

ESCAPE School, June 2021

> We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil**

- *Sir Tony Hoare?*
or Donald Knuth?

> We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil**

- *Sir Tony Hoare?*
or Donald Knuth?

From a 1974 article on why GOTO statements are good

Why optimize?

Why optimize?

However… once code is working, you do want it to be efficient!

- want a balance between usability/readability/correctness and speed/memory efficiency
- These are not always both achievable, so err on the side of *usability*

Why optimize?

However… once code is working, you do want it to be efficient!

- want a balance between usability/readability/correctness and speed/memory efficiency
- These are not always both achievable, so err on the side of *usability*

Some things:

- Python is interpreted (though some compilation happens), and can therefore be *slow*
- For-loops in particular are 100 - 1000x slower than C loops…
- There are some nice ways to speed up code, however, and get close to low-level language speed

Slowness of Python

Not an inherent problem with the *language*

- python ≠ CPython!
  - ➤ but CPython does generally get faster each release
- other python implementations exist that are trying to solve the general speed problem:
  - ➤ **pypy** - pypy.org fully JIT-compiled python
  - ➤ **pyston** - optimized CPython from Facebook
  - ➤ other efforts to remove bottlenecks from CPython (no GIL, etc)

Slowness of Python

Not an inherent problem with the *language*

- python ≠ CPython!
  - ➤ but CPython does generally get faster each release
- other python implementations exist that are trying to solve the general speed problem:
  - ➤ **pypy** - pypy.org fully JIT-compiled python
  - ➤ **pyston** - optimized CPython from Facebook
  - ➤ other efforts to remove bottlenecks from CPython (no GIL, etc)

So one option to optimization is:

Do nothing!

Wait for a faster implementation, or a new version of CPython to be released, or swap in a completely different implementation!

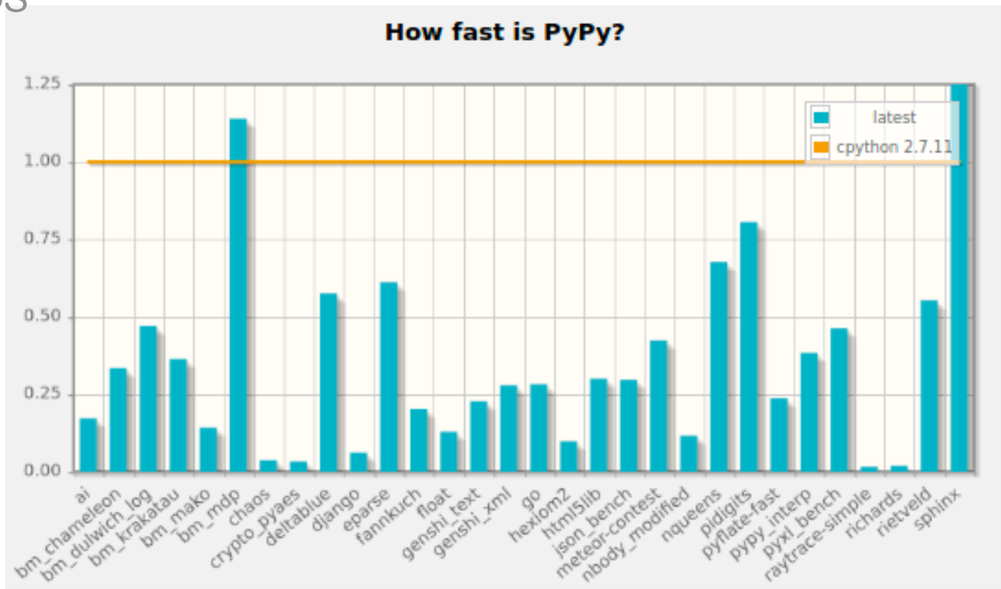
Some notes on PyPy

Just In time → compiled when used, not before

A compiler framework similar to GCC, the default on macOS

Advantages of PyPy:

- all PyPy code is **JIT**-compiled with **LLVM**
- support for most (but not all) of NumPy
- some support for C-extensions, but not all c-code can be run yet
- supports (so far) Python language up to version 3.7.9



Disadvantages:

- Works well speeding-up pure-python code, but scientific code is often a mix of Numpy/scipy/c-code: *it's often slower than CPython!*
- C-extensions not fully supported

But... there is a lot you can do to make your python code faster *now.*

Steps to optimization

1) Make sure code *works correctly* first

- DO NOT optimize code you are writing or debugging!

2) Identify use cases for optimization:

- how often is a function called? Is it useful to optimize it?
- If it is not called often and finishes with reasonable time/memory, stop!

3) *Profile* the code to identify bottlenecks in a more scientific way

- Profile time spent in each function, line, etc
- Profile memory use

4) try to re-write as little as possible to achieve improvement

5) refactor if it is still problematic…

- some times the *design* is what is making the code slow... can it be improved? (e.g.: ***flat better than nested***!)

Speeding up code 1: Memoization

Basic principle: don't recompute things you computed already!

Instead, compute them once, and just return the pre-computed result when asked. (trade memory for speed)

The hard way:

- keep a dictionary keyed by the input to a function with the output as the value. If the key exists, return the value:

```
RESULTS_CACHE = {}


def memoized_compute(x):
    if x in RESULTS_CACHE:
        return RESULTS_CACHE[x]
    result = do_some_large_computation(x)
    RESULTS_CACHE[x] = result
```

It works, but is ugly and not very *pythonic*...

Also if there are many values of x, you will use a lot of memory

Speeding up code 1: Memoization

The better way: as usual, python already has you covered!

- use **functools.lru_cache**
  → built-in memoization as a decorator
- Specify (roughly) the expected maximum size of the cache
  - ➤ it will still work if you go over it, but just not be as efficient
- It uses (a hash of) all inputs to the function as the key

LRU: Least Recently Used:
Throw away cached items that were not accessed recently, if memory gets slim

(one method for caching, there are many others)

```
RESULTS_CACHE = {}

def memoized_compute(x):
    if x in RESULTS_CACHE:
        return RESULTS_CACHE[x]
    result = do_some_large_computation(x)
    RESULTS_CACHE[x] = result
```

```
from functools import lru_cache

@lru_cache(maxsize=1000)
def do_some_large_computation(x):
    # slow code here
    return result
```

Speeding up code 2: Numpy

For-loops are slow! (in pure python)

Use NumPy vector operations as much as possible → they are optimized already**!**

- don't call a function on many small pieces of data when you can **call it on an array all at once**
- numpy is implemented in C & Fortran *and* it uses fast numerical libraries, optimized for your CPU (e.g. Intel Math Kernel Library MKL, BLAS, LAPACK etc)
- usually just vectorizing your code to avoid some for-loops, will give you great performance.

➤ bad:

```
for ii in range(100):
    x = ii*0.1
    y[ii] = f(x)
```

➤ Good:

```
x = np.linspace(0,10,100)
y = f(x)
```

Speeding up code 2: Numpy

For-loops are slow! (in pure python)

Use NumPy vector operations as much as possible → they are optimized already**!**

- don't call a function on many small pieces of data when you can **call it on an array all at once**
- numpy is implemented in C & Fortran *and* it uses fast numerical libraries, optimized for your CPU (e.g. Intel Math Kernel Library MKL, BLAS, LAPACK etc)
- usually just vectorizing your code to avoid some for-loops, will give you great performance.

➤ bad:

```
for ii in range(100):
    x = ii*0.1
    y[ii] = f(x)
```

➤ Good:

```
x = np.linspace(0,10,100)
y = f(x)
```

This requires practice, and feels very strange at first if you are coming from C programming!

Take some time to look through the *NumPy* and *SciPy* **API documentation** - there are tons of interesting functions to help you!

Speeding up code 3: Numba

Takes python code and *directly* uses introspection to compile it with LLVM

- Pretty **automatic**, *but doesn't always help*! Still need code written in a way that can be optimized (for-loops are actually good here, it can't do much with numpy operations since they are already compiled code)
- Can generate **NumPy "ufuncs"** directly (function that works on scalars but is run on all elements of an array), which are too slow to write in python normally.
- Can even compile to **GPU** code for nVidia *CUDA* and AMD *ROC* GPUs!

```
from numba import jit
from numpy import arange

# jit decorator tells Numba to compile this function.
# The argument types will be inferred by Numba when function is called.
@jit
def sum2d(arr):
    M, N = arr.shape
    result = 0.0
    for i in range(M):
        for j in range(N):
            result += arr[i,j]
    return result

a = arange(9).reshape(3,3)
print(sum2d(a))
```

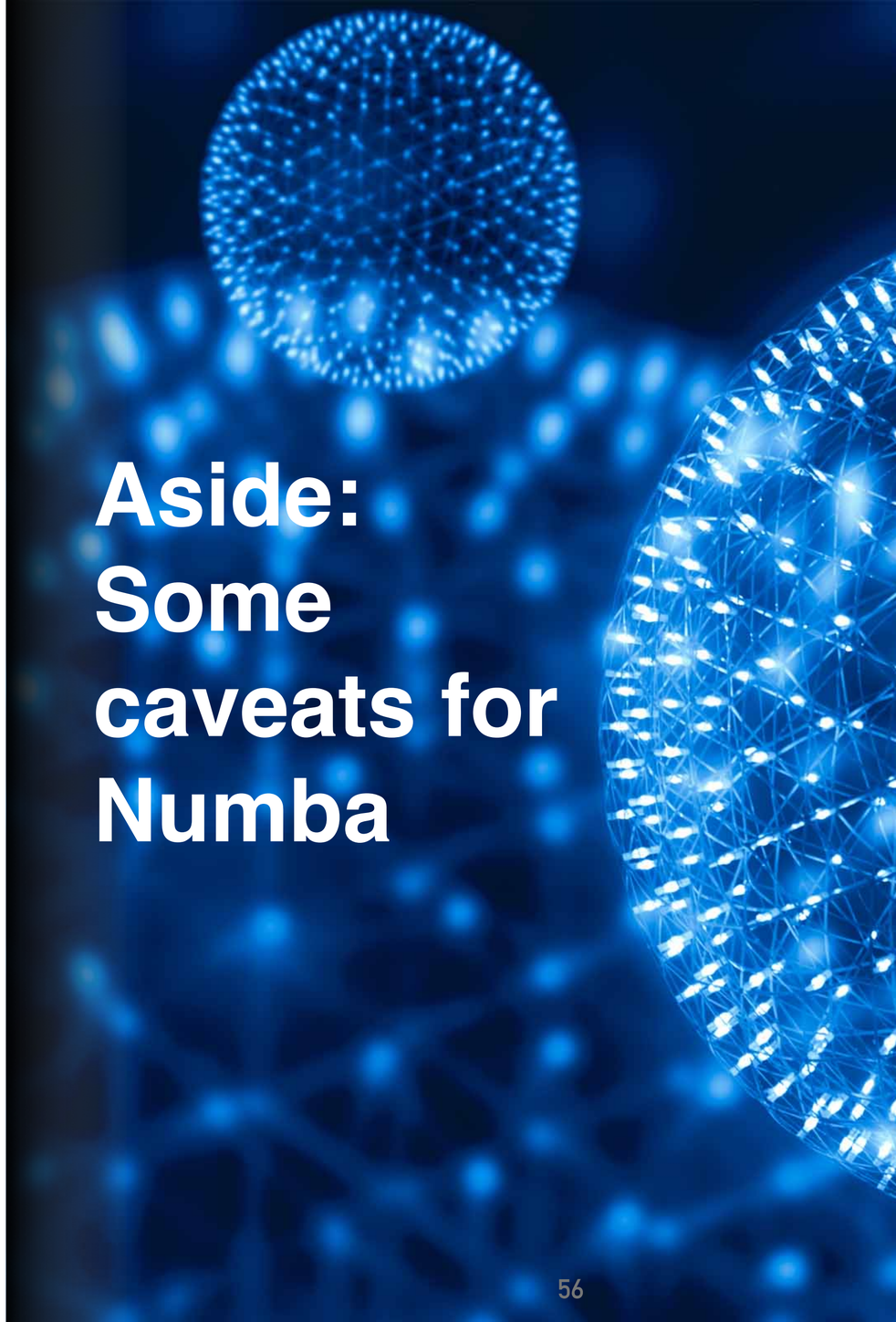
just add this decorator, and it's magic (nearly)

Numba operates in two modes:

- No-Python Mode:
  - gives large performance boost
  - but only supports basic python types and a subset of numpy/scipy operations
- Object Mode
  - fall-back if No Python mode fails
  - supports any python object
  - but gives little or not speed up in most situations

Tip:

- To force it to use No-Python mode
  - set *nopython=True* in the options
  - better: use **@njit**
- **@njit will fail if the code cannot be optimized by numba, and it will tell you why!**
- **There is some discussion that @njit will become the default in the future**



Aside: Some caveats for Numba

More numba caveats:

note that you need to "jit" not only the parent function, but any function that it calls that needs to be sped up. Otherwise, only Object Mode can work!

```python
from timeit import default_timer as timer
from matplotlib.pylab import imshow, jet, show, ion
import numpy as np

from numba import jit


@jit
def mandel(x, y, max_iters):
    """
    Given the real and imaginary parts of a
    complex number,
    determine if it is a candidate for membership
    in the Mandelbrot
    set given a fixed number of iterations.
    """
    i = 0
    c = complex(x,y)
    z = 0.0j
    for i in range(max_iters):
        z = z*z + c
        if (z.real*z.real + z.imag*z.imag) >= 4:
            return i

    return 255
```

```python
@jit
def create_fractal(min_x, max_x, min_y, max_y, image, iters):
    height = image.shape[0]
    width = image.shape[1]

    pixel_size_x = (max_x - min_x) / width
    pixel_size_y = (max_y - min_y) / height
    for x in range(width):
        real = min_x + x * pixel_size_x
        for y in range(height):
            imag = min_y + y * pixel_size_y
            color = mandel(real, imag, iters)
            image[y, x] = color

    return image

image = np.zeros((500 * 2, 750 * 2), dtype=np.uint8)
s = timer()
create_fractal(-2.0, 1.0, -1.0, 1.0, image, 20)
e = timer()
print(e - s)
imshow(image)
```

example from the Numba docs

Numba with NumPy

Numba supports a large number of NumPy functions (and even some scipy)**:**

- It does not actually call NumPy code!
- it *re-implements* it in a way that is compilable with LLVM.

https://numba.pydata.org/numba-doc/dev/reference/numpysupported.html

So what is the point? Isn't NumPy really optimized already?

- Minimize intermediate results!
  - ➤ numpy operations often have to allocate memory for data that is not needed in the end:

```
x = np.arange(1000)
result = A * x**2 + B * x + C
```

in C, you might do this all in one loop, with no extra memory needed:

```
for (i=0; i<x.size; i++) {
    result[i] = A*x[i]*x[i] + B*x[i] + C;
}
```

- More control over parallelization (See next lecture)

Advanced Numba

Numba includes a lot of advanced features and options to *jit* that can help speed things up

- e.g. specify the input and output type mapping, rather than infer it
- Easy NumPy Ufunc generation with *vectorize* and *guvectorize (generalized)*
  - *e.g. let you write code that operates on 1D array, and broadcast it to N-dimensional arrays*
- Options like target='GPU' for producing CUDA code or similar
- Parallelization onto multiple threads with parallel=True (see next lecture)

```
import numpy as np

from numba import guvectorize

@guvectorize(['void(float64[:], intp[:], float64[:])'], '(n),()->(n)')
def move_mean(a, window_arr, out):
    window_width = window_arr[0]
    asum = 0.0
    count = 0
    for i in range(window_width):
        asum += a[i]
        count += 1
        out[i] = asum / count
    for i in range(window_width, len(a)):
        asum += a[i] - a[i - window_width]
        out[i] = asum / count

arr = np.arange(20, dtype=np.float64).reshape(2, 10)
print(arr)
print(move_mean(arr, 3))
```

example from the Numba docs

Summary

Write good clean code first!

Identify bottlenecks in speed and memory with profiling tools

- don't worry so much about things that are not called often!
- try to narrow it down to the most critical parts of code

Use numpy, cython, numba or other technologies to improve the bottleneck

- try not to obfuscate the code to achieve speed! Readability still counts.

Demo

Parallelization

ESCAPE School, June 2021

What is parallelization?

Run non-sequential parts of a computing task *simultaneously*, maximizing resource use.

On a Single Machine

- Multi-threading / processes
  Multiple cores
- Vectorization
  Multiple instructions for one core

On Multiple Machines:

- Batch Queues
- Workflow Systems
- MPI (*Message Passing Interface)*

Multiple Processes vs Multi-Threading

Jobs that run on your computer are called *Processes*

- You can run many at once (your OS handles multi-tasking)
- Each has a process ID (PID) and it's own memory space, and takes some time to start up.
- Processes can start **child** sub-processes (hierarchy)
- Shared memory difficult, socket communication (send/receive messages) usually preferred

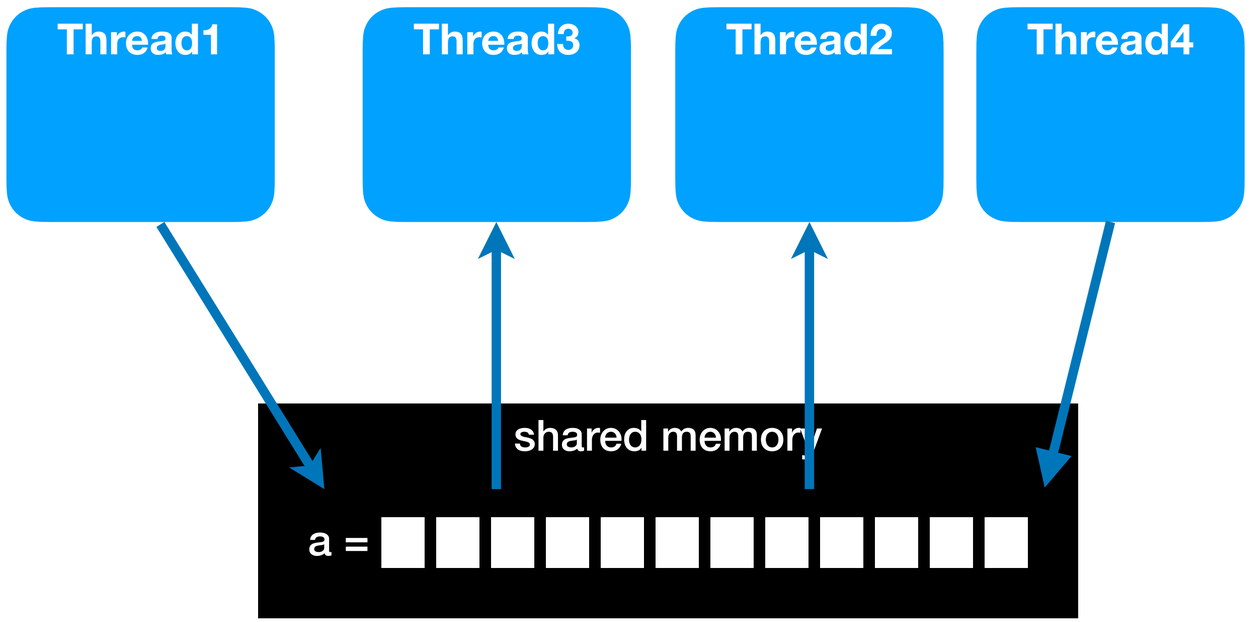
Within a process, you can also start any number of "lightweight sub processes" **called *Threads*.**

- *very* little overhead to start or stop a Thread (start hundreds in a fraction of a second)
- **Memory is shared with parent** - easy! ... but pay attention

Your computer's operating system will automatically schedule Processes or Threads on all available CPU cores

They run in parallel and preemptively (on most systems)

A side note on shared memory



Multiple threads can write to or read from the same object in memory (e.g. an array)

- The **order** in which threads run is **not defined**
- if two threads access the same memory address, but the order in which it happens changes the results, this is called a "**race condition**"
- they are both "racing" to access the memory, the result depends who "wins"

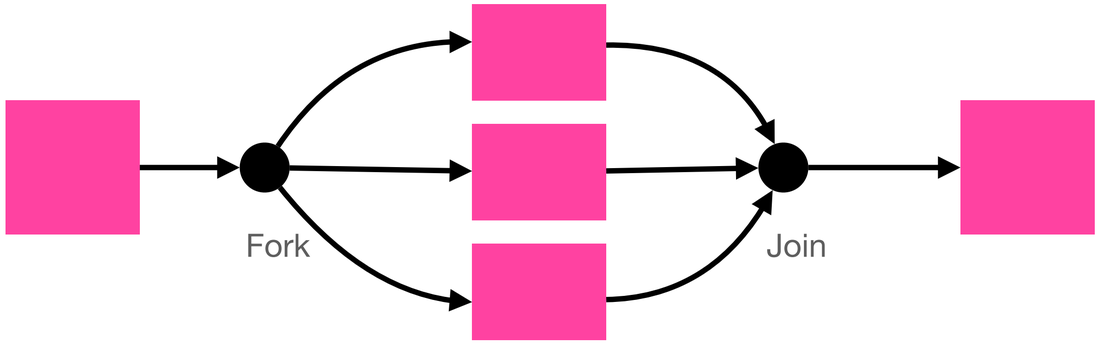
A side note on shared memory

The way to avoid this is by *locking memory* in blocks of code where it must not be changed by another process.

- (implemented via e.g. mutexes, semaphores...)
- It is very easy to make mistakes
- For example **deadlocks** (code hangs because multiple threads ask for a lock in the wrong order)

Preferred method to avoid this: Fork and Join

```
solve(problem):
    if problem is small enough:
        solve problem directly (sequential algorithm)
    else:
        for part in subdivide(problem)
            fork subtask to solve(part)
        join all subtasks spawned in previous loop
        return combined results
```



Python and Multi-threading/processes

Python (CPython) is bad at multi-threading!

- Global Interpreter Lock (GIL) locks shared memory when the interpreter is executing a statement
- Means that in practice **all python threads run on the same core** → parallel but not across cores**!**
- Still useful for things like processing while also waiting for I/O (see also cooperative multi-tasking using async/await however!)

In Python, generally *prefer multiple processes to threads*.

- **CAVEAT**: when python runs **compiled** C-code (or Fortran, or Numba!), multiple threads can be used *inside that code* (but not once it returns to the interpreter)...

Multi-core Multi-threading without thinking...

Many *NumPy* and *SciPy* operations use the underlying libraries like *BLAS*:

- **B**asic **L**inear **A**lgebra **S**ubprograms
- Created in 1979 as standard interface for linear algebra (Fortran)
- Many heavily optimized versions exist today that include **automatic vectorization** and **multi-core parallelization**:
  - ➤ OpenBLAS
  - ➤ Intel *MKL* (Math Kernel Library) - closed source, but included in Anaconda

Functions that use these like *np.dot(X,Y)* **automatically run large operations on all cores in the system**

- you can control how many using environment variables

How to benefit?

Just use NumPy and SciPy more! Avoid loops, use vector-style math.

Don't have to think about parallelization, you some for free (but not perfect)

To Stop NumPy from multi-threading:

```
export MKL_NUM_THREADS=1
export NUMEXPR_NUM_THREADS=1
export OMP_NUM_THREADS=1
```

(e.g. on shared batch systems, you may need to do this)

Parallelism with Numba!

Remember: the GIL only affects python code, not compiled code, where we can use Threads as normal!

Numba makes it really easy to make parallel loops that use threads even knowing about threads!

https://numba.pydata.org/numba-doc/latest/user/parallel.html

- The @njit decorator has a *parallel=True* option
- **It tries to parallelize what it thinks it can** safely (e.g. numpy sums, products, etc)
- You can also say what should be parallelized using *prange* (parallel range) in an explicit loop

```
from numba import njit, prange

@jit(parallel=true)
def my_fast_parallel_func(n):
    for i in prange(10):
        for j in range(1000):
            do_something(i, j, n)
```

This runs in parallel

This doesn't

Numba Parallel Gotchas

```python
@njit(parallel=True)
def prange_wrong_result(x):
    n = x.shape[0]
    y = np.zeros(4)
    for i in prange(n):
        # accumulating into the same element of `y` from different
        # parallel iterations of the loop results in a race condition
        y[:] += x[i]

    return y
```

Numba Parallel Gotchas

Careful: race conditions!
No locking is done, be sure that two the same memory is not written to by two iterations of the loop! Remember the order is not guaranteed.

```
@njit(parallel=True)
def prange_wrong_result(x):
    n = x.shape[0]
    y = np.zeros(4)
    for i in prange(n):
        # accumulating into the same element of `y` from different
        # parallel iterations of the loop results in a race condition
        y[:] += x[i]

    return y
```

Another side note: Multi-threading in C++

OpenMP https://www.openmp.org/

compiler extension to support easier multi-threaded parallelism in C/C++

- easily paralleize loops with little effort
- fork-and-join method made easy
- very similar to what is done in numba parallel! (even more flexible in fact)

```
int main(int argc, char **argv)
{
    int a[100000];

    #pragma omp parallel for
    for (int i = 0; i < 100000; i++) {
        a[i] = 2 * i;
    }

    return 0;
}
```

Demo

(back to the Heat Equation Solver)

Debugging (understanding) Numba parallel

You can inspect what numba has done!

```
@jit(parallel=True)
def some_function():
        ... code here
```

```
some_function.parallel_diagnostics()
```

Basic multi-core processing

Python's standard library provides all that you need to use all your cores via the concurrent.futures*

- concurrent.futures.**ThreadPoolExecutor**:
  - ➤ sends work (function calls) to multiple worker *threads*
  - ➤ see previous slide, however → will not really give you multi-core performance
- concurrent.futures.**ProcessPoolExecutor**:
  - ➤ launches multiple worker *processes* and sends work to each
  - ➤ There is some overhead in creating the workers and for sending data between them, so small jobs may be slower than non-parallel!
- and more via external packages... (see MPIPoolExecutor later)

* There is also the older but more complex *multiprocessing* module that has similar functionality

Example

```python
from concurrent.futures import ProcessPoolExecutor
from time import sleep


def work(x):
    sleep(5)
    return x**2

if __name__ == "__main__":

    with ProcessPoolExecutor() as pool:
        future = pool.submit(work, 10)

        # non-blocking calls to done()
        for ii in range(10):
            print("Is it done?", future.done())
            sleep(1)

        # blocking call to result()
        print("Result:",  future.result())
```

Tip: you need this guard line! It will fail otherwise

```
Is it done? False
Is it done? False
Is it done? False
Is it done? False
Is it done? False
Is it done? False
Is it done? True
Is it done? True
Is it done? True
Is it done? True
Result: 100
```

Example with map() interface

```python
from concurrent.futures import ProcessPoolExecutor
from time import sleep
import random


def work(x):
    sleep(random.uniform(1,5))
    print(f"Computing {x}")
    return x**2

if __name__ == "__main__":

    values = [1,2,3,10,20,50,100,200]

    with ProcessPoolExecutor() as pool:
       print("Input ", values)
       output = pool.map(work, values)
       print("Output ", output) # non-blocking! (generator object)
       print("Output ", list(output)) #blocking!
```

```
Input  [1, 2, 3, 10, 20, 50, 100, 200]
Output  <generator object _chain_from_iterable_of_lists at 0x107f875f0>
Computing 2
Computing 100
Computing 10
Computing 20
Computing 200
Computing 1
Computing 50
Computing 3
Output  [1, 4, 9, 100, 400, 2500, 10000, 40000]
```

Basic multi-core processing (2)

Python's standard library also provides an older module called multiprocessing*

- multiprocessing.**Pool**
  - ➤ creates a set of worker processes
  - ➤ provides an interface to loop over jobs in an iterable that works exactly like the python built-in map() command:

```
results  = map(function, sequence)
```

which is equivalent to :

```
results  = (function(x) for x in sequence)
```

With *multiprocessing* this becomes:

```
pool = multiprocessing.Pool()
results = pool.map(function, sequence)
```

* There is also a newer (but less feature-full) *concurrent.futures* module that has similar functionality

Basic multi-core processing (2)

Python's standard library also provides an older module called multiprocessing*

- multiprocessing.**Pool**
  - ➤ creates a set of worker processes
  - ➤ provides an interface to loop over jobs in an iterable that works exactly like the python built-in map() command:

```
results  = map(function, sequence)
```

which is equivalent to :

```
results  = (function(x) for x in sequence)
```

With *multiprocessing* this becomes:

```
pool = multiprocessing.Pool()
results = pool.map(function, sequence)
```

* There is also a newer (but less feature-full) *concurrent.futures* module that has similar functionality

***IMPORTANT NOTE*:**

The way both **multiprocessing** and **concurrent.futures** work prevents them from running in a notebook!

The functions that use them must be in a **module**/script.

Differences in the two implementations

Multiprocessing:

- has a easier to understand (for new users) interface
- allows inter-process communication with Pipes and Queues
- provides map()-like interface, with array return

Concurrent.futures:

- Provides a unified interface for many parallelization backends
- Provides map()-like interface returned as generator (lazy
- Much simpler API (good and bad - you have less flexibility)

Both:

- Do not allow direct shared memory → require you to use a mechanism to communicate with other proceses

to the notebook!

How to parallelize using multiprocessing

Now, if there is time, lets talk a bit about multi-machine parallelism...

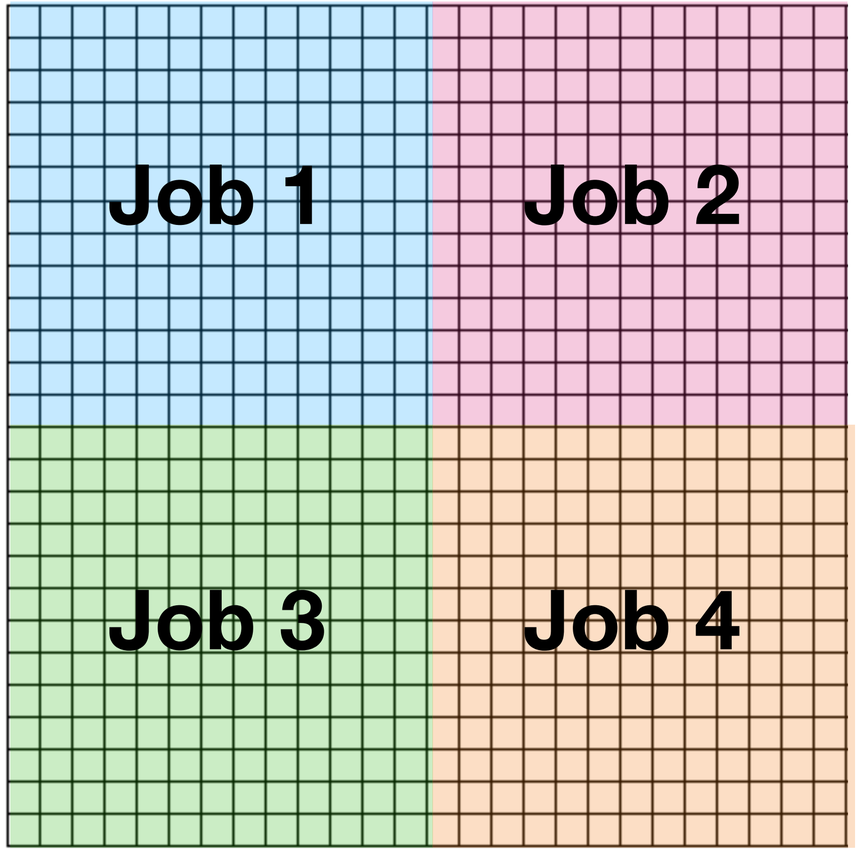
Workflows with inter-job communication

Much of theoretical science has problems that operate on large N-dimensional grids of numbers :

- magneto-hydrodynamic simulations (e.g. of a supernova)
- computational particle physics (e.g. lattice QCD)
- weather prediction

These cannot be "embarassingly parallelized"

- require **block parallelism**
- No shared memory (as in Threads), so have to exchange information a edges of blocks
- A good solution: MPI via the MPI4py library



https://mpi4py.readthedocs.io/en/stable/

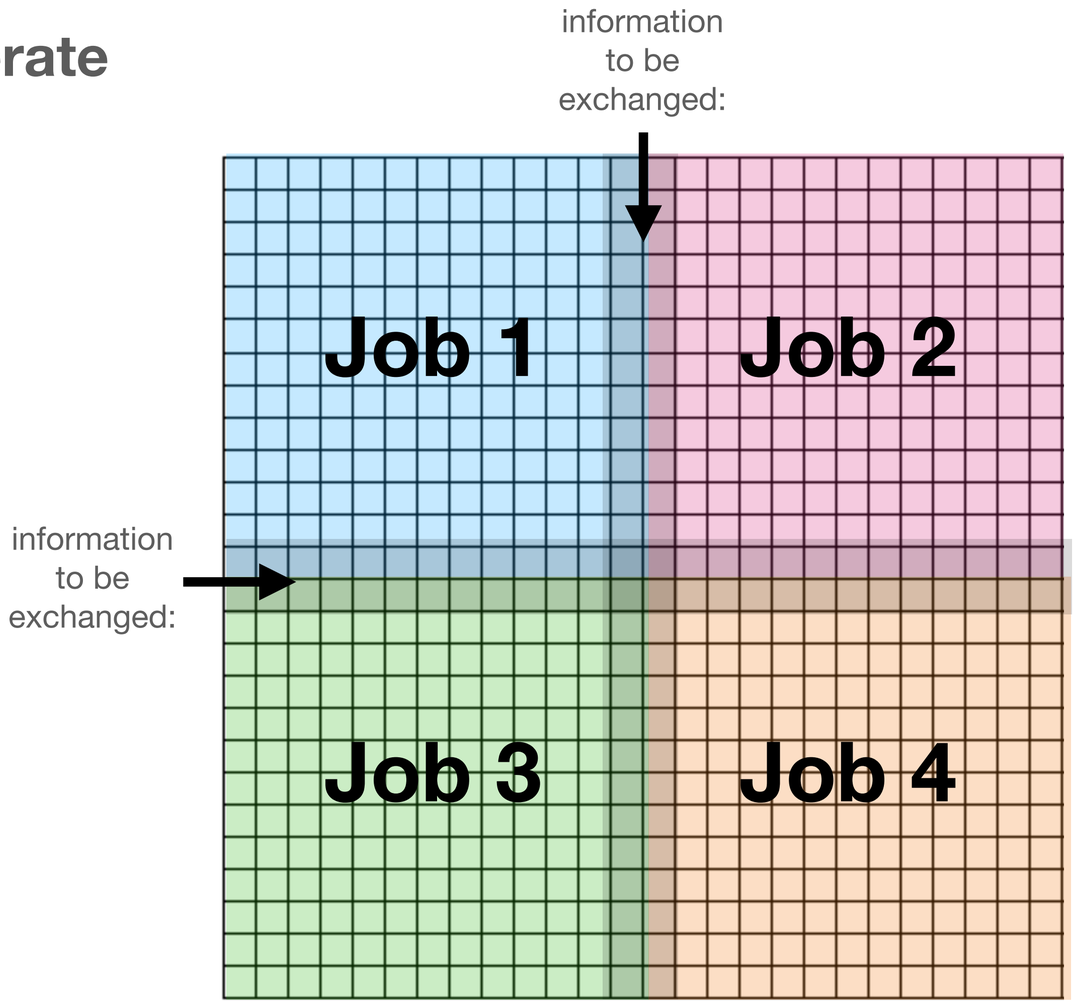
Workflows with inter-job communication

Much of theoretical science has problems that operate on large N-dimensional grids of numbers :

- magneto-hydrodynamic simulations (e.g. of a supernova)
- computational particle physics (e.g. lattice QCD)
- weather prediction

These cannot be "embarassingly parallelized"

- require **block parallelism**
- No shared memory (as in Threads), so have to exchange information a edges of blocks
- A good solution: MPI via the MPI4py library



https://mpi4py.readthedocs.io/en/stable/

Demo

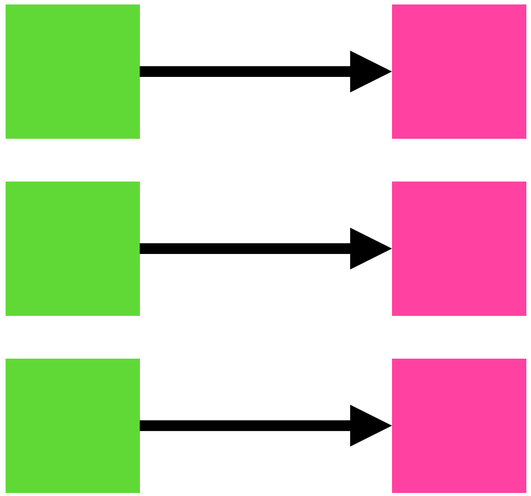
Simple Parallelization on Many Machines

Data oriented science is often a example of an "embarrassingly parallel problem"

- I/O and processing times can take a long time
- But the work can trivially be split into many "jobs", with no communication between them needed
- Typical solution: use a **batch queuing system**!

Example:

I have 100 *raw data files* and I want to process them to *reduced data files*

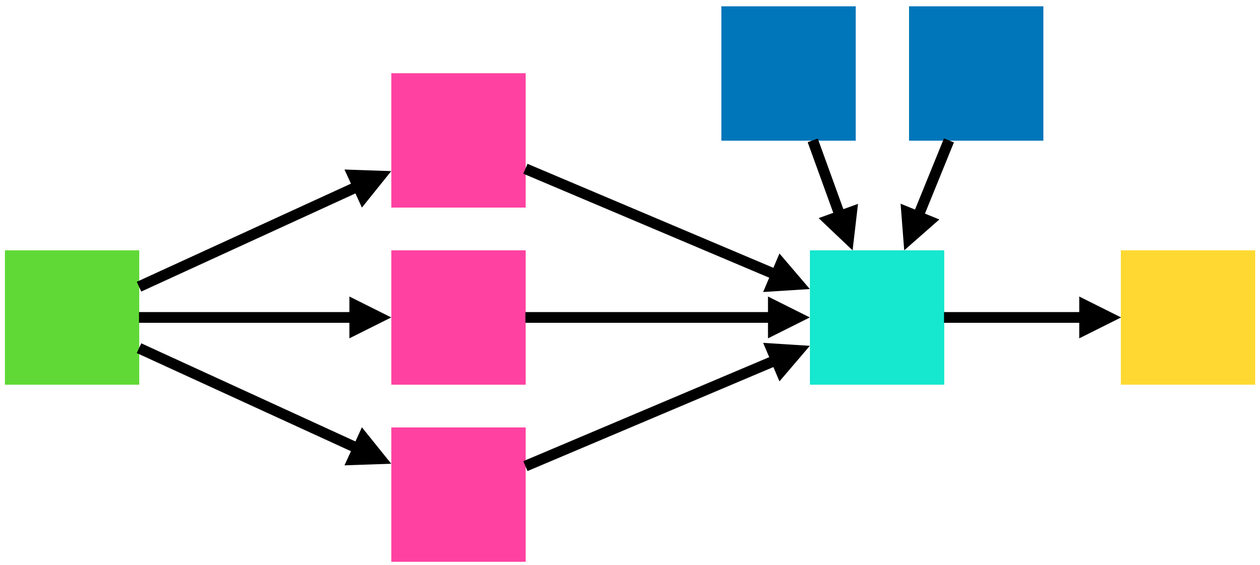


General procedure:

```
for job in list_of_work:
    batch_queue.submit(work)
```

Wait for all jobs to
Download the results.

More complicated parallel workflows



Imaging your workflow is a *directed acyclic graph* (DAG)

- Similar to a *Makefile* (which is a language to define DAGs!)
- A simple batch queue will require you to do a lot of work manually
- fortunately there are systems to help!

Workflow Management Systems

Example: *Apache Airflow* airflow.apache.org

- fully python based workflow management system
- somewhat complex to set up and get running (compared to single-machine systems)
- define your DAG as steps in python

```
mamba install -c conda-forge apache-airflow
```

→ but probably needs it's own env due to lots of dependencies

Many many others...

- DIRAC (used by CMS, CTA)

The problem with workflow management

No one language is used... so for each system, you usually have to re-write your workflow

Common Workflow Language www.commonwl.org

- started by biologists working in bioinformatics
- defines a common way to define a DAG in **YAML** text, where each step can be an executable with inputs and outputs. Steps run in Docker Containers for easy reproducibility.
- many systems already support this language, so DAGs can run on local machine, or on an **Airflow** cluster, or others