Parallelism in Python

Why Parallel?

- Often our problems will take a while to run. Often that will be longer than we feel like waiting.

- Everything we have talked about so far has assumed the computer does one thing after another. If we can do several things at once, we can get our answer faster.

- If we split up our work across computers, we may even be able to do larger problems with more RAM.

- Goal of today is to see the very basics. Life can get muuuch more complicated.

Types of Parallel

- We'll look at three examples implementing Simpson's rule.
- Multiprocessing. Python starts many threads which might (or might not) work in parallel.
- Numba parallel - split up loops among different workers. (openmp in compiled languages more flexible, may be straight-up easier to use).
- mpi4py - have completely separate python instances. They *pass messages* to each other, must decide how to split up work.
- NB - GPUs can be extremely powerful, but harder to code for. Nvidia GPUs somewhat supported by numba. If you have/have access to GPUs, you should investigate!

Multiprocessing

- Many tasks can be split up with multiprocessing. (multithreading is another option).

- Python starts a new copy of itself (“process”) for every worker.

- Simple example is Pool. Pool creates a bunch of processors, then Pool.map farms out jobs.

Pool Example

```
import numpy as np
from multiprocessing import Pool
import os
import time

def square(x):
    pid=os.getpid()
    print('process ',pid,' is working on ',x)
    time.sleep(3)
    return x*x


if __name__=='__main__':
    nthread=8
    n=4*nthread
    with Pool(nthread) as p:
        vec=p.map(square,range(n+1))
    print('squares are ',vec)
    print('sum of first ',n,' squares is: ',np.sum(vec))
    print('expected: ',n*(n+1)*(2*n+1)//6)
```

square the input, print process ID

Start a pool of nthread processes

Call our function on every entry, save output in a list.

Pool Example Output

```
Jonathans-MBP-2:parallel sievers$ python3 pool_squares.py
process  85576  is working on  0
process  85575  is working on  1
process  85578  is working on  2
process  85577  is working on  3
process  85576  is working on  4
process  85577  is working on  5
process  85575  is working on  6
process  85578  is working on  7
process  85576  is working on  8
squares are  [0, 1, 4, 9, 16, 25, 36, 49, 64]
sum of first  8  squares is:  204
expected:  204
```

- When a process finishes its entry, it gets assigned another one. You do not know which entry will run in which process. Note that 0,4,8 run together, 1,6 run together.

- Pool orders the output for you.

Pool Example Output 2

```
process  85590  is working on  0
process  85590  is working on  1
process  85590  is working on  2
process  85590  is working on  3
process  85590  is working on  4
process  85590  is working on  5
process  85590  is working on  6
process  85590  is working on  7
process  85590  is working on  8
squares are  [0, 1, 4, 9, 16, 25, 36, 49, 64]
sum of first  8  squares is:  204
expected:  204
```

- Without time.sleep(), the first process has run all the tasks before the others even get started.

- Moral - make sure your job is large enough to warrant overhead from parallel.

Integration with Pool

- Let’s write a parallel Simpson’s rule integrator.
- We will parcel out regions of domain to different processes.
- We will have to figure out how to break things up ourselves. What are some possible issues
  (odd # of points, values used may depend on # of threads, how are we going to pass arguments?)
- We will also have to decide if the caller or callee breaks up the domain.

Aside - partial

- We almost always want to pass multiple arguments.

- We've (briefly) seen args,kwargs syntax. Sadly, many python packages will not pass them along for us.

- functools.partial will create a new function for us

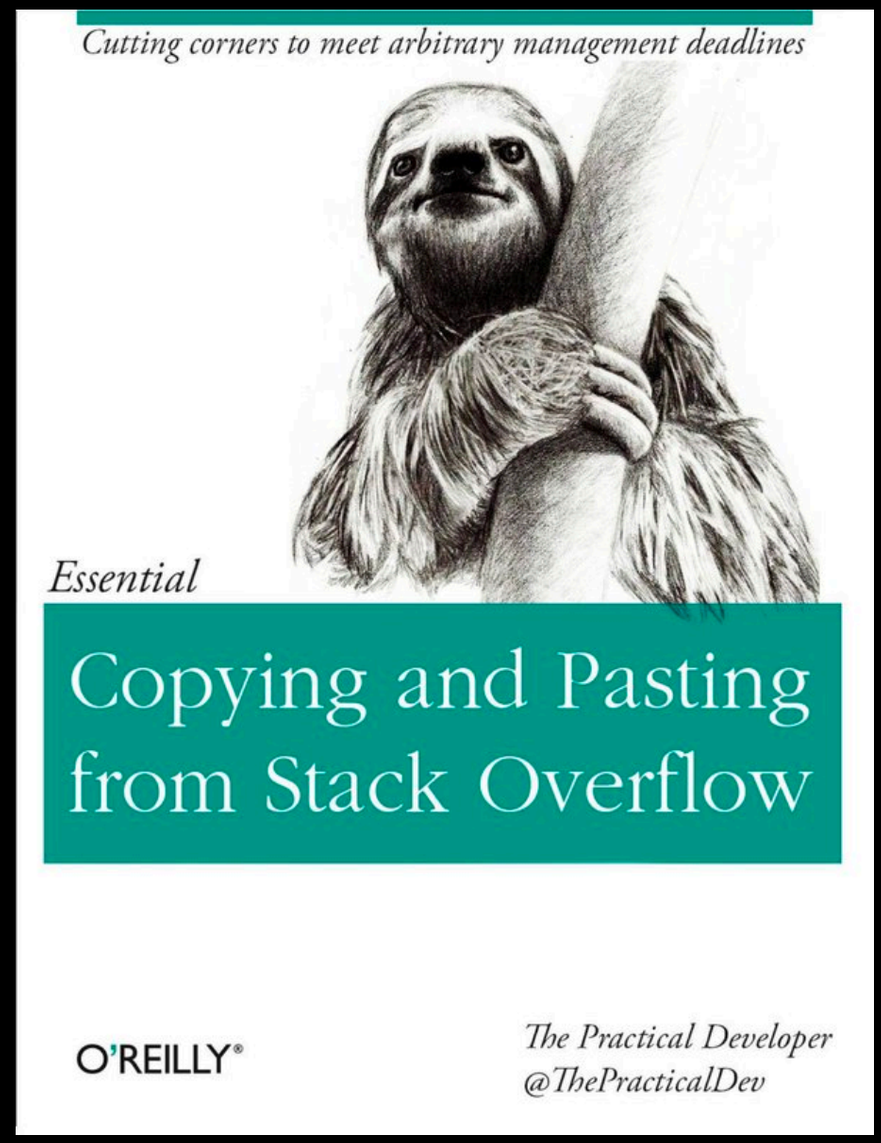
Partial Output

```python
import numpy as np
from functools import partial

def gauss(x,x0,sigma):
    return np.exp(-0.5*(x-x0)**2/sigma**2)


x0=0.5
sigma=2.0
myfun=partial(gauss,x0=x0,sigma=sigma)
x=1.5

print('expected answer is ',gauss(x,x0,sigma))
#note the lack of arguments
print('partial gave us ',myfun(x))
```




```
Jonathans-MBP-2:parallel sievers$ python3 partial_example.py
expected answer is  0.8824969025845955
partial gave us  0.8824969025845955
```

Parallel Integrator with Pool V1

Parallel integrator where logic is in integration routine. Function needs to know its ID, and how many processes there are.
Calling routine very simple - we just pass along a range of ints.
Note - each process returns into a list, the total area is the sum of that list.

```
def integrate(id,fun,range,dx,nproc):
    edges=np.linspace(range[0],range[1],nproc+1)
    xmin=edges[id]
    xmax=edges[id+1]
    print('I am ',id,' out of ',nproc,' with range ',xmin,xmax)
    nx=int(np.ceil((xmax-xmin)/dx))
    if (nx%2==0):
        nx=nx+1
    x=np.linspace(xmin,xmax,nx)
    y=fun(x)
    mydx=x[1]-x[0]
    my_sum=y[0]+y[-1]+4*np.sum(y[1::2])+2*np.sum(y[2:-1:2])
    return mydx*my_sum/3
```

```
if __name__=='__main__':
    nproc=4
    dx=1/256
    with Pool(nproc) as p:
        out=p.map(partial(integrate,fun=np.exp,range=[0,1],dx=dx,nproc=nproc),range(nproc)
    print('integral is ',np.sum(out))
```

Parallel Integrator with Pool V2

```
import numpy as np
from multiprocessing import Pool
from functools import partial

def integrate(lims,fun,dx):
    x1=lims[0]
    x2=lims[1]
    print('integrating ',x1,x2)
    n=int(np.ceil((x2-x1)/dx))
    if (n&1)==0:
        n=n+1
    x=np.linspace(x1,x2,n)
    dx=x[1]-x[0]
    tot=fun(x[0])+fun(x[-1])
    tot=tot+4*np.sum(fun(x[1::2]))
    tot=tot+2*np.sum(fun(x[2:-1:2]))
    return dx*tot/3
```

Top: integrator routine with region pre-split. Note that we pass in the limits as a single argument.
Bottom: calling code. Note that we had to decide on number of processes, split up domain, and put start/stop points for each process into a list of lists. Partial handled the rest (see also: Pool.starmap).

Question: would I get identical answer with different # of procs?

```
if __name__=='__main__':
    nthread=8
    x=np.linspace(0,1,nthread+1)
    lims=[None]*nthread
    for i in range(nthread):
        lims[i]=[x[i],x[i+1]]
    dx=0.001
    with Pool(nthread) as p:
        out=p.map(partial(integrate,fun=np.exp,dx=dx),lims)
    print('total is    ',np.sum(out))
    print('analytis is ',np.exp(x[-1])-np.exp(x[0]))
```

Pool Recap

- We can do some parallel computing with multiprocessing. Pool is simplest example, but many other tools exist.
- We have to think about how to break up our region, including which part of code breaks it up.
- Overhead for starting processes is large - small jobs may not actually run faster in parallel.
- NB - multiprocessing (unlike multithreading) gets around global interpreter lock (GIL).

Numba Parallel

- Numba will automatically parallelize loops for us. Add (parallel=True) to decorator, and use nb.prange in place of np.arange.

- Numba will start a number of workers, which (mostly) share common variables.

Race Condition

- Because different threads in numba share variables, if two threads try to write to same location, whoever races there first (or last) wins.

- Output can be random. This is bad! Called a *race condition*, and is one of the classic parallel bugs.

- Always think about who is writing where in parallel programs.

Numba Race Condition

```
import numpy as np
import numba as nb


@nb.njit(parallel=True)
def assign(n):
    val=-1
    for i in nb.prange(n):
        val=i
    print('after loop, value is ',val)

assign(16*64*64)
```

Seral, at the end, val would always equal n-1. When we run in parallel, who knows? Last thread probably wins.

Note - numba did warn us we were being stupid.

```
>>> exec(open("numba_assign.py").read())
/usr/local/lib/python3.8/site-packages/numba/parfors/parfor_lowering.py:974: NumbaParallelSafet
yWarning: Variable val.2 used in parallel loop may be written to simultaneously by multiple wor
kers and may result in non-deterministic or unintended results.

File "<string>", line 8:
<source missing, REPL/exec in use?>

  warnings.warn(NumbaParallelSafetyWarning(msg, loc))
after loop, value is  49151
after loop, value is  61439
after loop, value is  65535
after loop, value is  57343
after loop, value is  53247
after loop, value is  65535
after loop, value is  65535
after loop, value is  65535
after loop, value is  61439
after loop, value is  65535
>>>
```

Reduction

```
@nb.njit(parallel=True)
def add(n):
    tot=0
    val=-1
    for i in nb.prange(n):
        tot=tot+i
    return tot
for i in range(10):
    print('sum is ',add(16*64*64))
```

```
>>> exec(open("numba_add.py").read())
sum is  2147450880
sum is  2147450880
sum is  2147450880
sum is  2147450880
sum is  2147450880
sum is  2147450880
sum is  2147450880
sum is  2147450880
sum is  2147450880
sum is  2147450880
```

- Adding results across threads is called *reduction*, and is a very common parallel operation.
- Numba knows about this, and supports some cases that otherwise look like they should be wrong.
- Be *very* careful about this.

Bad Reduction

```
import numpy as np
import numba as nb


@nb.njit(parallel=True)
def add(n):
    tot=np.zeros(1)
    val=-1
    for i in nb.prange(n):
        tot[0]=tot[0]+i
    return tot[0]
for i in range(10):
    print('sum is ',add(16*64*64))
```

```
>>> exec(open("numba_add_v2.py").read())
sum is  679467008.0
sum is  905957376.0
sum is  662689792.0
sum is  645912576.0
sum is  746575872.0
sum is  679467008.0
sum is  864016384.0
sum is  385867776.0
sum is  796907520.0
sum is  536862720.0
```

- Identical to previous, except tot is 1-element array.
- Now numba does not do the right thing, and it doest warn us.
- Always listen to errors, but correctness is your job!

Numba Integrator

```
@nb.njit (parallel=True)
def integrate_exp(x0,x1,dx):
    n=int( (x1-x0)/dx)
    #n needs to be odd.  below code is a very fast way
    #of making sure that's true.  n&1 does bit-wise and
    #and so is zero if n is even, 1 if n odd.  ^ is xor
    #so (n&1)^1 is zero if n is odd, 1 if n even.  equivalent to
    #if iseven(n), n=n+1, but no if statements so very fast.
    n=n+((n&1)^1)
    dx=(x1-x0)/(n-1)
    tot=0.0
    for i in nb.prange(1,n-1):
        #now that I broke you in with the bit operators,
        #2+2*(n&1) will be 4 if n is odd, 2 if n is even
        tot=tot+(2+2*(i&1))*np.exp(x0+i*dx)
    tot=tot+np.exp(x0)+np.exp(x1)
    return dx*tot/3
```

```
dx=0.000001
ans=integrate_exp(0,1,dx)
t1=time.time();ans=integrate_exp(0,1,dx);t2=time.time();dt=t2-t1;
print('we got ',ans,' expected ',np.exp(1)-np.exp(0),' in ',dt)
t1=time.time();ans=simpson(np.exp,0,1,dx);t2=time.time();dt2=t2-t1
print('vectorized answer is ',ans,' in ',dt2)
print('speed up is ',dt2/dt)
```

```
>>> exec(open("numba_integrate.py").read())
we got  1.7182818284590449  expected  1.718281828459045  in  0.0013182163238525391
vectorized answer is  1.7182818284590449  in  0.009827852249145508
speed up is  7.4554168927473325
```

Numba Recap

- Numba can parallelize loops for us. Unlike multiprocessing, we did not have to think about splitting the domain.

- We did open ourselves to race conditions. Under the hood, numba made a copy of tot *including* its starting value, accumulated into private copies, then added them together. If tot!=0, you could get unexpected answer.

- Our numba version was ~1.5x faster than vectorized, but ~8x faster when run in parallel. We won!

- Why was 8 core version not 8x faster than 1-core numba version?

Aside - OpenMP version

```
double integrate (double x0, double x1, double dx)
{
  int n=(x1-x0)/dx;
  if ((n&1)==0)
    n++;
  printf("n is %d\n",n);
  dx=(x1-x0)/(n-1);
  double tot=exp(x0)+exp(x1);
  //start a parallel region, every thread shares copies of variables
#pragma omp parallel shared(x0,x1,n,tot,dx) default(none)
  {
    //anything declared here is now going to be thread-private
    //we'll use mytot to sum up our private copy
    double mytot=0;
    //handy-dandy utility that splits up a loop amongst threads
#pragma omp for
    for (int i=1;i<n-1;i++) {
      int fac=2+2*(i&1);
      mytot=mytot+fac*exp(x0+i*dx);
    }
    //only let one thread at a time in here, avoids race condition
    //when reducing
#pragma omp critical
    tot+=mytot;
  }
  return tot*dx/3;
}
```

MPI4PY

- We can also break up our job into completely separate processes. Possibly running on different machines.
- Each process has a unique rank in [0,nproc), which we can use to e.g. break up domains.
- Processes send messages (MPI==Message Passing Interface)
- For integration, we will use standard Simpson’s routine - exactly what we used in Pool v2 example.

MPI Body

```
comm=MPI.COMM_WORLD
myrank = comm.Get_rank()
nproc=comm.Get_size()
print('I am ',myrank,' out of ',nproc)
x0=0.0
x1=1.0
dx=0.001

#break up the domain and decide which part
#this process owns
x=np.linspace(x0,x1,nproc+1)
myx0=x[myrank]
myx1=x[myrank+1]
myans=integrate([myx0,myx1],np.exp,dx)
```

MPI Body Comments

- First three lines are generic. MPI_COMM_WORLD is a way for all processes to speak to each other. Get_size() reports # of processes, Get_rank() reports which one you are.

- Set global range/dx to cover for problem. You *probably* want them all to agree - but they don't have to.

- Break up global domain, and decide which piece we own, and carry out our piece of integral. Now to assemble…

Reduction

```
#at this point each process knows its piece of
#the answer.  Now accumulate onto process 0
if myrank==0:
    ans=myans
    for i in range(1,nproc):
        ans=ans+comm.recv(source=i)
    print('on process ',myrank,' integral is ',ans)
else:
    comm.send(myans,0)
```

- We will assemble our answer on process 0. All other processes send their answer to 0 (comm.send). Process zero loops through all other processes and gets their answer.

- Only process 0 gets the final answer.

Running/Output

```
Jonathans-MBP-2:parallel sievers$ mpirun -np 4 python3 mpi_integrate_manual_reduce.py
I am  1  out of  4
I am  2  out of  4
I am  3  out of  4
I am  0  out of  4
integrating  0.25 0.5
integrating  0.5 0.75
integrating  0.0 0.25
integrating  0.75 1.0
on process  0  integral is  1.7182818284590557
```

- We need to start multiple copies of python. mpirun (sometimes mpiexec) will do this, and on a cluster will start on different computers.

- We see each process works on a piece, and process 0 reports correct answer.

Allreduce

```
ans=comm.allreduce(myans)
print('on process ',myrank,' integral is ',ans)
```

```
on process  0  integral is  1.718281828459056
on process  2  integral is  1.718281828459056
on process  1  integral is  1.718281828459056
on process  3  integral is  1.718281828459056
```

- Reduction is common enough that MPI supports it. reduce will put the answer on one node, allreduce will put the answer on all nodes.

- Allreduce is not only simpler to use - if implemented well, it is faster as well! Use the right tool for the right job.

Recap

- We saw how to use multiprocessing.Pool, numba, and mpi4py to break up Simpson's rule.

- Pool/MPI require you to figure out how to break up your problem.

- Numba will break it up for you, but watch for race conditions.

- Stay tuned on numba - it is still missing many features supported by compilers (see OpenMP) that are getting added in.

- Parallel computing can get (very, very) tricky, but many cases ae not too bad. My hope is you can at least get started now with google/stack overflow.