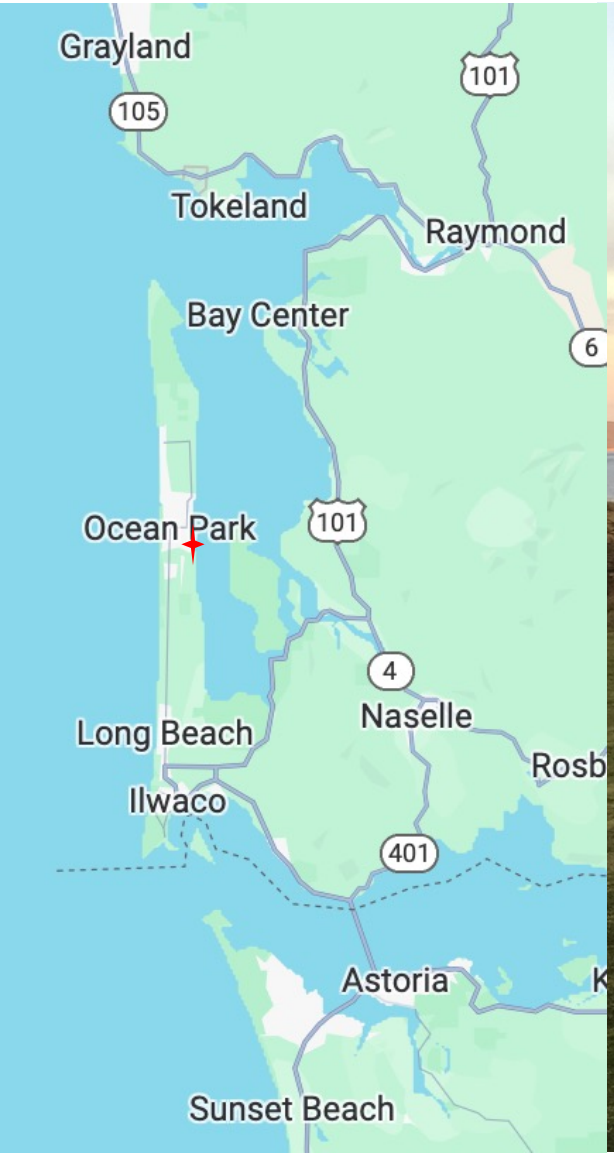
Parallel Programming with Python



Tim Mattson, University of Bristol and Merly.ai

merly.ai

We all love python … but what about performance

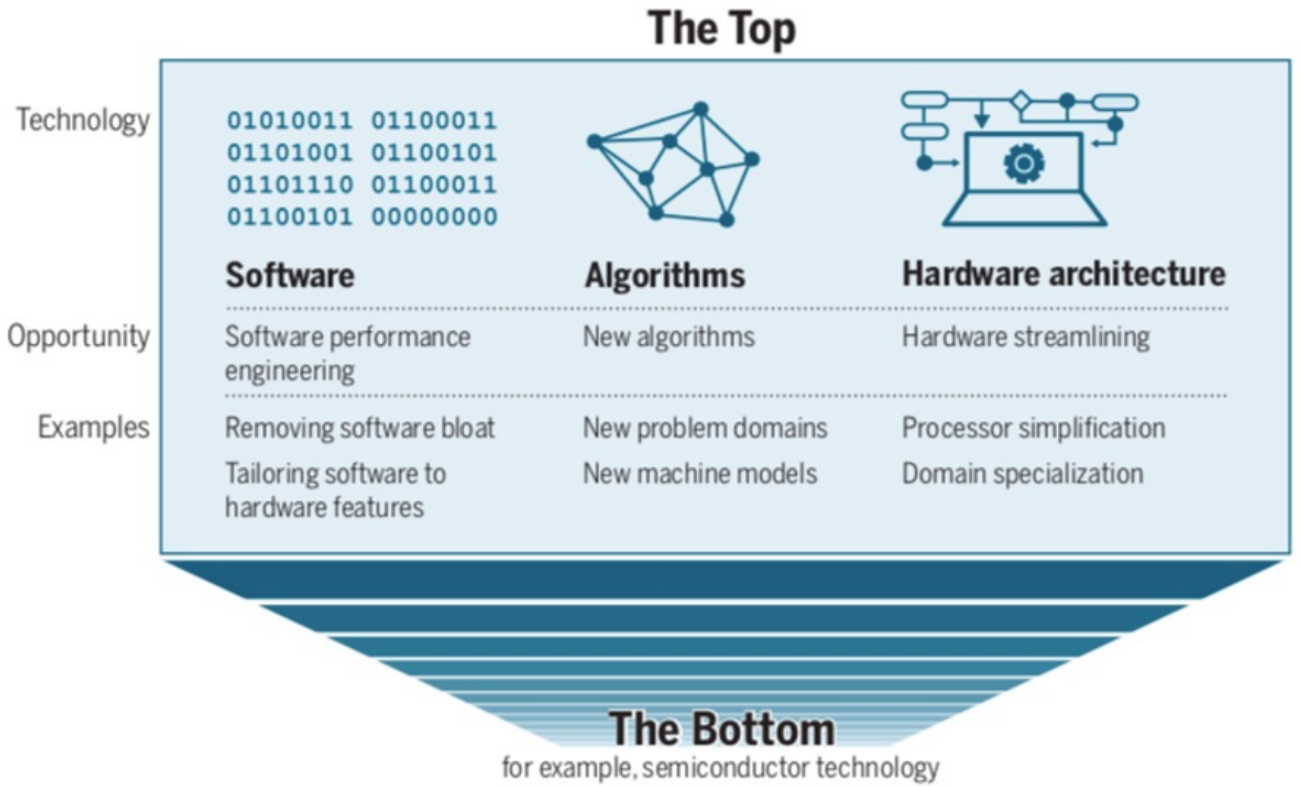
Software vs. Hardware and the nature of Performance

Up until ~2005, performance came from semiconductor technology

There's plenty of room at the Top: What will drive computer performance after Moore's law?*

Charles E. Leiserson[1], Neil C. Thompson[1,2]*, Joel S. Emer[1,3], Bradley C. Kuszmaul[1]†, Butler W. Lampson[1,4], Daniel Sanchez[1], Tao B. Schardl[1]

Leiserson *et al.*, *Science* **368**, eaam9744 (2020) 5 June 2020



Since ~2005 performance comes from "the top"

Better software Tech.
Better algorithms
Better HW architecture#

*It's because of the end of Dennard Scaling … Moore's law has nothing to do with it

#HW architecture matters, but dramatically LESS than software and algorithms

The view of Python from an HPC perspective

(from the ”Room at the top” paper).

```
for I in range(4096):
    for j in range(4096):
        for k in range (4096):
            C[i][j] += A[i][k]*B[k][j]
```

A proxy for computing over nested loops … yes, they know you should use optimized library code for DGEMM

Table 1. Speedups from performance engineering a program that multiplies two 4096-by-4096 matrices. Each version represents a successive refinement of the original Python code. “Running time” is the running time of the version. “GFLOPS” is the billions of 64-bit floating-point operations per second that the version executes. “Absolute speedup” is time relative to Python, and “relative speedup,” which we show with an additional digit of precision, is time relative to the preceding line. “Fraction of peak” is GFLOPS relative to the computer’s peak 835 GFLOPS. See Methods for more details.

Version	Implementation	Running time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak (%)
1	Python	25,552.48	0.005	1	–	0.00
2	Java	2,372.68	0.058	11	10.8	0.01
3	C	542.67	0.253	47	4.4	0.03
4	Parallel loops	69.80	1.969	366	7.8	0.24
5	Parallel divide and conquer	3.80	36.180	6,727	18.4	4.33
6	plus vectorization	1.10	124.914	23,224	3.5	14.96
7	plus AVX intrinsics	0.41	337.812	62,806	2.7	40.45

Amazon AWS c4.8xlarge spot instance, Intel® Xeon® E5-2666 v3 CPU, 2.9 Ghz, 18 core, 60 GB RAM

The view of Python from an HPC perspective

(from the ”Room at the top” paper).

```
for I in range(4096):
    for j in range(4096):
        for k in range (4096):
            C[i][j] += A[i][k]*B[k][j]
```

A proxy for computing over nested loops … yes, they know you should use optimized library code for DGEMM

This demonstrates a common attitude in the HPC community ….

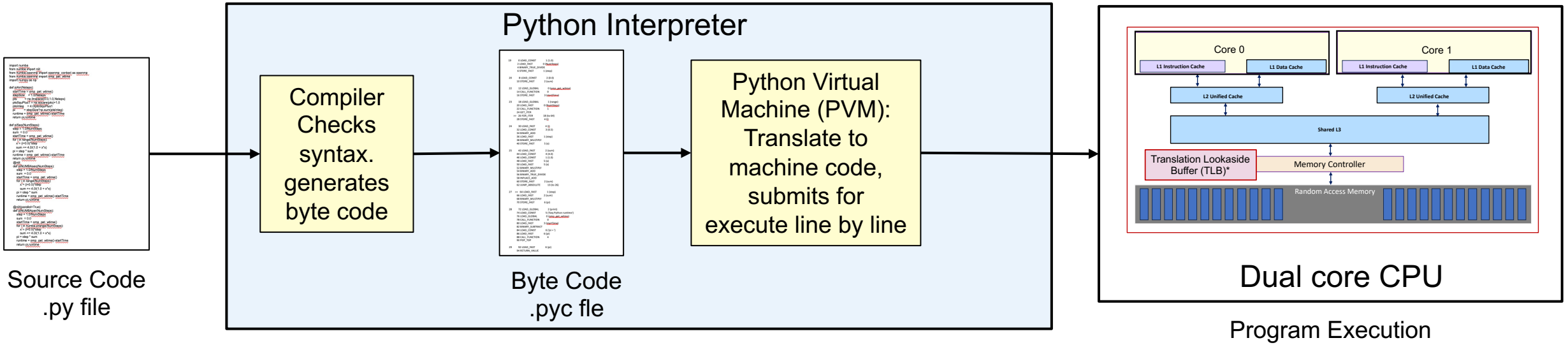
Python is great for productivity, algorithm development, and combining functions from high-level modules in new ways to solve problems. If getting a high fraction of peak performance is a goal … recode in C.

Version	Implementation	Running time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak (%)
1	Python	25,552.48	0.005	1	—	0.00
2	Java	2,372.68	0.058	11	10.8	0.01
3	C	542.67	0.253	47	4.4	0.03
4	Parallel loops	69.80	1.969	366	7.8	0.24
5	Parallel divide and conquer	3.80	36.180	6,727	18.4	4.33
6	plus vectorization	1.10	124.914	23,224	3.5	14.96
7	plus AVX intrinsics	0.41	337.812	62,806	2.7	40.45

Amazon AWS c4.8xlarge spot instance, Intel® Xeon® E5-2666 v3 CPU, 2.9 Ghz, 18 core, 60 GB RAM

Why is Python so slow?

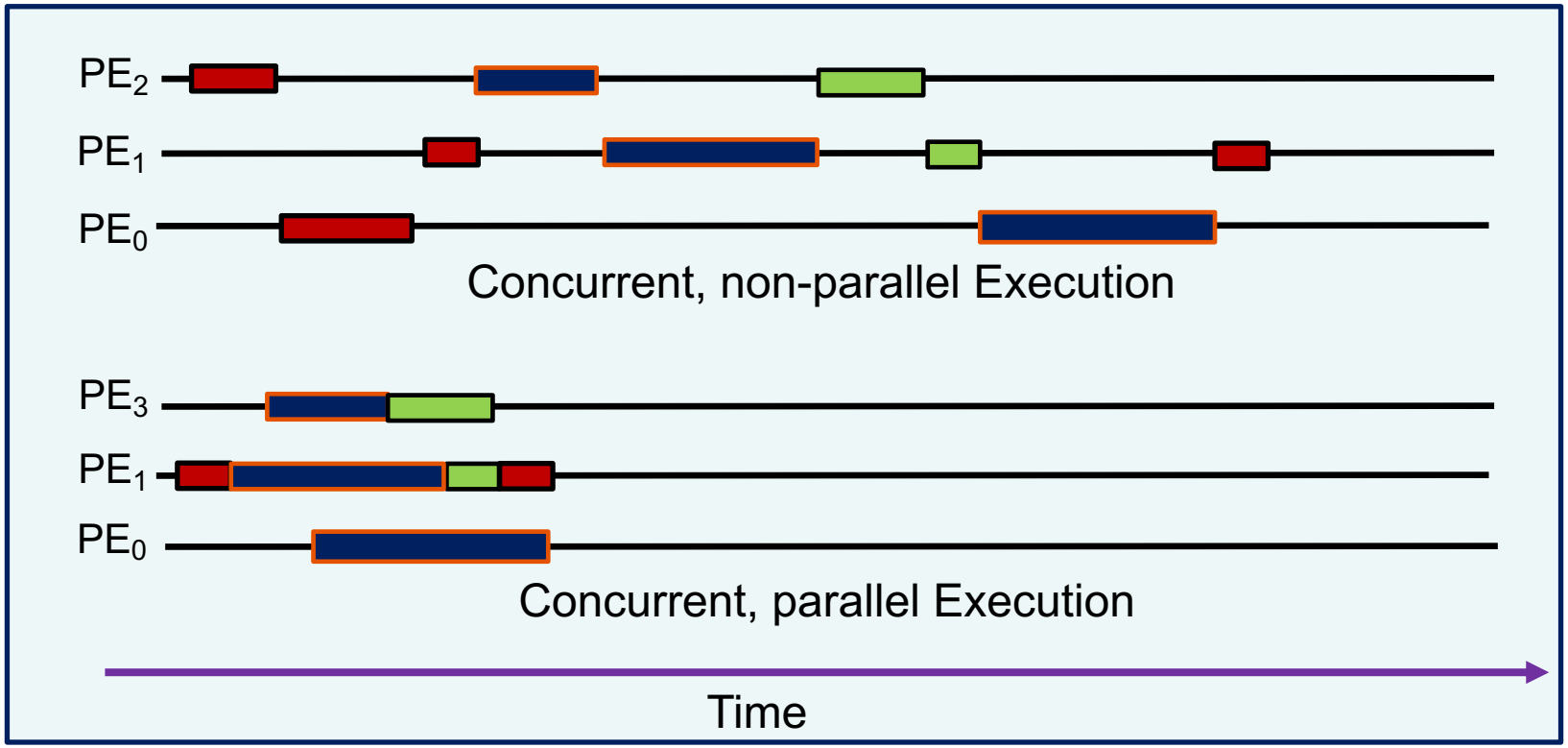
- Python is interpreted … not compiled



- What if I want my Python program to run in parallel. Does that work?
- Not really. Python has a **Global Interpreter lock** (**GIL**). This is a mutex (mutual exclusion lock) so only one thread at a time can make forward progress.

Concurrency vs. Parallelism

- Two important definitions:
  - Concurrency: A condition of a system in which multiple tasks are active and unordered. If **scheduled fairly**, they can be described as ***logically*** making **forward progress** at the same time.
  - Parallelism: A condition of a system in which multiple tasks are **actually** making **forward progress** at the same time.

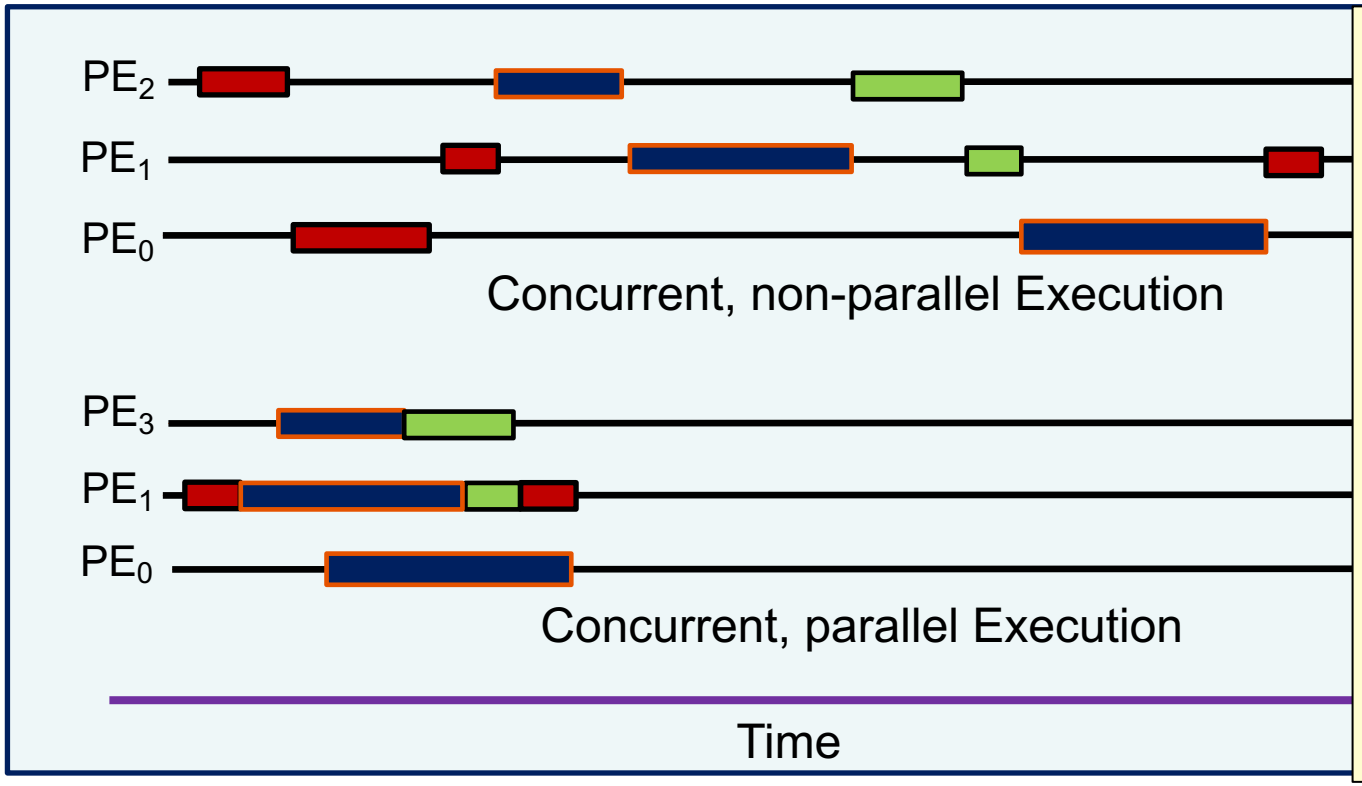


PE = Processing Element

Figure from "An Introduction to Concurrency in Programming Languages" by J. Sottile, Timothy G. Mattson, and Craig E Rasmussen, 2010

Concurrency vs. Parallelism

- Two important definitions:
  - Concurrency: A condition of a system in which multiple tasks are active and unordered. If **scheduled fairly**, they can be described as ***logically*** making **forward progress** at the same time.
  - Parallelism: A condition of a system in which multiple tasks are **actually** making **forward progress** at the same time.



The GIL kills multithreaded parallelism in native python … which seems nuts. But it's actually a good thing. With multithreaded parallelism, threads share an address space and can interfere with updates to locations in memory (a data race). These bugs are so difficult to manage, that the Python creators decided to just sidestep the issue with the GIL

PE = Processing Element

Figure from "An Introduction to Concurrency in Programming Languages" by J. Sottile, Timothy G. Mattson, and Craig E Rasmussen, 2010

How do you install PyOMP on your own system?

PyOMP installation

- Preferred installation method is through conda. Running python from 3.8 to 3.10:
  - conda install -c python-for-hpc -c conda-forge pyomp

- We currently support PyOMP on four systems
  - linux-ppc6le
  - linux-64 (x86_64)
  - osx-arm64 (mac)
  - linux-arm64

- We also have a working (free) JupyterLab under binder for OpenMP CPU at:
  - https://mybinder.org/v2/gh/Python-for-HPC/binder/HEAD

https://github.com/Python-for-HPC/PyOMP

Loop Parallelism code

```
from numba import njit
from numba.openmp import openmp_context as openmp

@njit
def piFunc(NumSteps):
    step = 1.0/NumSteps
    pisum = 0.0

    with openmp ("parallel for private(x) reduction(+:pisum)"):
        for i in range(NumSteps):
            x = (i+0.5)*step
            pisum += 4.0/(1.0 + x*x)

    pi = step*pisum
    return pi

pi = piFunc(100000000)
```

OpenMP managed through the *with* context manager.

Numba Just In Time (JIT) compiler compiles the Python code into LLVM thereby bypassing the GIL. Compiled code cached for later use.

Pass the OpenMP directive into the OpenMP context manager as a string

- **parallel**: creates a team of threads
- **for**: maps loop iterations onto threads.
- **private(x)**: each threads gets its own x
- Loop control index of a parallel for (**i**) is private to each thread.
- **reduction(+:sum)**: combine sum from each thread using +

GIL: Global Interpreter Lock

Numerical Integration results in seconds … lower is better

Threads	PyOMP		C	
	Loop		Loop	
1	0.447		0.444	
2	0.252		0.245	
4	0.160		0.149	
8	0.0890		0.0827	
16	0.0520		0.0451	

10^8 steps

Intel® Xeon® E5-2699 v3 CPU with 18 cores running at 2.30 GHz.

For the C programs we used Intel® icc compiler version 19.1.3.304 as icc -qnextgen -O3 –fopenmp

Ran each case 5 times and kept the minimum time. **JIT time is not included** for PyOMP (it was about 1.5 seconds)

Single Program Multiple Data (SPMD)

```
from numba import njit
import numpy as np
from numba.openmp import openmp_context as openmp
from numba.openmp import omp_get_thread_num, omp_get_num_threads
MaxTHREADS = 32
@njit
def piFunc(NumSteps):
    step = 1.0/NumSteps
    partialSums = np.zeros(MaxTHREADS)
    with openmp("parallel shared(partialSums,numThrds) private(threadID,i,x,localSum)"):
        threadID = omp_get_thread_num()
        with openmp("single"):
            numThrds = omp_get_num_threads()
        localSum = 0.0
        for i in range(threadID, NumSteps, numThrds):
            x = (i+0.5)*step
            localSum = localSum + 4.0/(1.0 + x*x)
        partialSums[threadID] = localSum
    return step*np.sum(partialSums)

pi = piFunc(100000000)
```

- **omp_get_num_threads()**: get N=number of threads
- **omp_get_thread_num()**: thread rank = 0…(N-1)
- **single**: One thread does the work, others wait
- **private(x)**: each threads gets its own x
- **shared(x)**: all threads see the same x

Deal out loop iterations as if a deck of cards (a cyclic distribution) … each threads starts with the Iteration = ID, incremented by the number of threads, until the whole "deck" is dealt out.

The data environment seen by OpenMP threads

- The data environment is the collection of variables visible to the threads in a team.

- Variables can be **shared** or **private**.
  - **Shared variable**: A variable that is visible (i.e. can be read or written) to all threads in a team.
  - **Private variable**: A variable that is only visible to an individual thread.

- All the code associated with an OpenMP directive (such as **parallel** or **for**), including the code in functions called inside that code, is called a **region**. A directive plus code in the immediate block associated with it, is called a **construct**

- Rules for defining a variable as shared or private:
  - A variable is **shared** if it is used before or after an OpenMP construct, otherwise it is **private**.
  - Variables can be made shared or private through clauses included with a directive.

```
from numba import njit
from numba.openmp import openmp_context as openmp

@njit
def piFunc(NumSteps):
    step = 1.0/NumSteps
    pisum = 0.0
    with openmp ("parallel for reduction(+:pisum)"):
        for i in range(NumSteps):
            x = (i+0.5)*step
            pisum += 4.0/(1.0 + x*x)

    pi = step*pisum
    return pi

pi = piFunc(100000000)
```

x first used inside the OpenMP construct … it is private.

Numerical Integration results in seconds … lower is better

Threads	PyOMP			C		
	Loop	SPMD		Loop	SPMD	
1	0.447	0.450		0.444	0.448	
2	0.252	0.255		0.245	0.242	
4	0.160	0.164		0.149	0.149	
8	0.0890	0.0890		0.0827	0.0826	
16	0.0520	0.0503		0.0451	0.0451	

10^8 steps

Intel® Xeon® E5-2699 v3 CPU with 18 cores running at 2.30 GHz.

For the C programs we used Intel® icc compiler version 19.1.3.304 as icc -qnextgen -O3 –fiopenmp

Ran each case 5 times and kept the minimum time. **JIT time is not included** for PyOMP (it was about 1.5 seconds)

Divide and conquer (with explicit tasks)

```
from numba import njit
from numba.openmp import openmp_context as openmp
from numba.openmp import omp_get_num_threads, omp_set_num_threads
MIN_BLK = 1024*256
@njit
def piComp(Nstart, Nfinish, step):
    iblk = Nfinish-Nstart
    if(iblk<MIN_BLK):
        pisum = 0.0
        for i in range(Nstart,Nfinish):
            x= (i+0.5)*step
            pisum += 4.0/(1.0 + x*x)
    else:
        sum1 = 0.0
        sum2 = 0.0
        with openmp ("task shared(sum1)"):
            sum1 = piComp(Nstart, Nfinish-iblk/2,step)
        with openmp ("task shared(sum2)"):
            sum2 = piComp(Nfinish-iblk/2,Nfinish,step)
        with openmp ("taskwait"):
            pisum = sum1 + sum2
    return pisum
```

Solve

Split

Merge

```
@njit
def piFunc(NumSteps):
    step = 1.0/NumSteps
    sum = 0.0
    startTime = omp_get_wtime()
    with openmp ("parallel"):
        with openmp ("single"):
            pisum = piComp(0,NumSteps,step)

    pi = step*pisum
    return pi

pi = piFunc(100000000)
```

Fork threads and launch the computation

- **single**: One thread does the work, others wait
- **task**: code block enqueued for execution
- **taskwait**: wait until task in the code block finish

Numerical Integration results in seconds … lower is better

Threads	PyOMP			C		
	Loop	SPMD	Task	Loop	SPMD	Task
1	0.447	0.450	0.453	0.444	0.448	0.445
2	0.252	0.255	0.245	0.245	0.242	0.222
4	0.160	0.164	0.146	0.149	0.149	0.131
8	0.0890	0.0890	0.0898	0.0827	0.0826	0.0720
16	0.0520	0.0503	0.0517	0.0451	0.0451	0.0431

10^8 steps

Intel® Xeon® E5-2699 v3 CPU with 18 cores running at 2.30 GHz.

For the C programs we used Intel® icc compiler version 19.1.3.304 as icc -qnextgen -O3 –fiopenmp

Ran each case 5 times and kept the minimum time. **JIT time is not included** for PyOMP (it was about 1.5 seconds)

There is more …. But this is enough to get you started with CPU programming in PyOMP

So let's wrap up our discussion of CPU programming

The view of Python from an HPC perspective

```
for I in range(4096):
    for j in range(4096):
        for k in range (4096):
            C[i][j] += A[i][k]*B[k][j]
```

We know better … the IKJ order is more cache friendly

And we picked a smaller problem

```
for I in range(1000):
    for k in range(1000):
        for j in range (1000):
            C[i][j] += A[i][k]*B[k][j]
```

Table 1. Speedups from performance engineering a program that multiplies two 4096-by-4096 matrices. Each version represents a successive refinement of the original Python code. "Running time" is the running time of the version. "GFLOPS" is the billions of 64-bit floating-point operations per second that the version executes. "Absolute speedup" is time relative to Python, and "relative speedup," which we show with an additional digit of precision, is time relative to the preceding line. "Fraction of peak" is GFLOPS relative to the computer's peak 835 GFLOPS. See Methods for more details.

Version	Implementation	Running time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak (%)
1	Python	25,552.48	0.005	1	—	0.00
2	Java	2,372.68	0.058	11	10.8	0.01
3	C	542.67	0.253	47	4.4	0.03
4	Parallel loops	69.80	1.969	366	7.8	0.24
5	Parallel divide and conquer	3.80	36.180	6,727	18.4	4.33
6	plus vectorization	1.10	124.914	23,224	3.5	14.96
7	plus AVX intrinsics	0.41	337.812	62,806	2.7	40.45

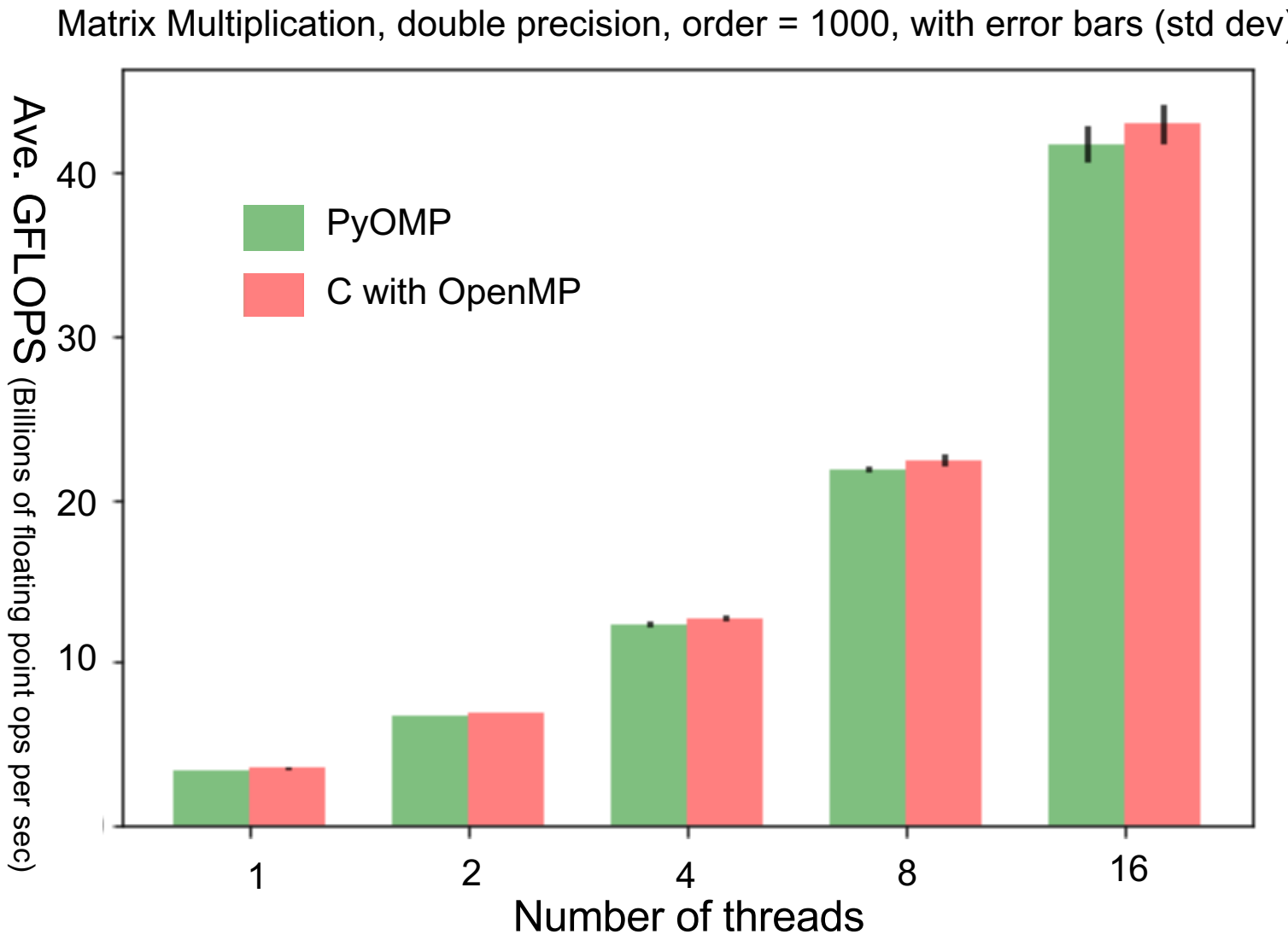
Amazon AWS c4.8xlarge spot instance, Intel® Xeon® E5-2666 v3 CPU, 2.9 Ghz, 18 core, 60 GB RAM

PyOMP DGEMM (Mat-Mul with double precision numbers)

```
from numba import njit
import numpy as np
from numba.openmp import openmp_context as openmp
from numba.openmp import omp_get_wtime

@njit(fastmath=True)
def dgemm(iterations,order):

    # allocate and initialize arrays
    A = np.zeros((order,order))
    B = np.zeros((order,order))
    C = np.zeros((order,order))

    # Assign values to A and B such that
    # the product matrix has a known value.
    for i in range(order):
        A[:,i] = float(i)
        B[:,i] = float(i)
```

```
tInit = omp_get_wtime()
with openmp("parallel for private(j,k)"):
      for i in range(order):
          for k in range(order):
              for j in range(order):
                  C[i][j] += A[i][k] * B[k][j]

dgemmTime = omp_get_wtime() - tInit

# Check result
checksum = 0.0;
for i in range(order):
    for j in range(order):
        checksum += C[i][j]
ref_checksum = order*order*order
ref_checksum *= 0.25*(order-1.0)*(order-1.0)
eps=1.e-8
if abs((checksum - ref_checksum)/ref_checksum) < eps:
    print('Solution validates')
    nflops = 2.0*order*order*order
    print('Rate (MF/s): ',1.e-6*nflops/dgemmTime)
```

DGEMM PyOMP vs C-OpenMP



250 runs for order 1000 matrices

PyOMP times **DO NOT** include the one-time JIT cost of ~2 seconds.

… but remember, the JIT'ed code can be cached for future use. It's straightforward to hide the JIT cost.

Intel® Xeon® E5-2699 v3 CPU, 18 cores, 2.30 GHz, threads mapped to a single CPU, one thread/per core, first 16 physical cores.
Intel® icc compiler ver 19.1.3.304 (icc –std=c11 –pthread –O3 xHOST –qopenmp)

Loop Parallelism code naturally maps onto the CPU

```
from numba import njit
import numpy as np
from numba.openmp import openmp_context as openmp

@njit(fastmath=True)
def dgemm(iterations,N):

    # allocate and initialize numpy arrays
    # A, B and C of size N by N.   <<< code not shown>>>

    with openmp("target teams loop collapse(2) private(j)"):
        for i in range(N):
            for k in range(N):
                for j in range(N):
                    C[i][j] += A[i][k] * B[k][j]
```

OpenMP constructs managed through the *with* context manager.

Map the loop onto a 2D index space … the loop body defines the kernel function

- **target**: map execution from the host onto the device
- **teams loop**: Map kernel instances onto PEs inside the compute units
- **collapse(2)**: combine following two loops into a single iteration space.
- **private(j)**: each threads gets its own j variable
- Indices of parallelized loops (**i,k**) are private to each thread.

PE: processing element. The finest-grained processing element inside a GPU. Also known as a SiMD lane or CUDA-core.

5-point stencil: solve kernel

```
@njit
def solve(n, alpha, dx, dt, u, u_tmp):
    # Finite difference constant multiplier
    r = alpha * dt / (dx ** 2)
    r2 = 1 - 4 * r
    # Loop over the nxn grid
        for i in range(n):
            for j in range(n):
                # Update the 5-point stencil.
                # Using boundary conditions on the edges of the domain.
                # Boundaries are zero because the MMS solution is zero there.
                u_tmp[j, i] = (r2 * u[j, i] +
                               (u[j, i+1] if i < n-1 else 0.0) +
                               (u[j, i-1] if i > 0   else 0.0) +
                               (u[j+1, i] if j < n-1 else 0.0) +
                               (u[j-1, i] if j > 0 else 0.0))
```

25,000x25,000 grid for 10 time steps
* Xeon Platinum 8480+: 67.6 secs

Solution: parallel stencil (heat)

25,000x25,00 grid for 10 time steps
- Xeon Platinum 8480+: 67.6 secs
- Nvidia V100: 22.6 secs

```
@njit
def solve(n, alpha, dx, dt, u, u_tmp):
    """Compute the next timestep, given the current timestep"""

    # Finite difference constant multiplier
    r = alpha * dt / (dx ** 2)
    r2 = 1 - 4 * r
    with openmp ("target loop collapse(2) map(tofrom: u, u_tmp)"):
        # Loop over the nxn grid
        for i in range(n):
            for j in range(n):
                u_tmp[j, i] = (r2 * u[j, i] +
                               (u[j, i+1] if i < n-1 else 0.0) +
                               (u[j, i-1] if i > 0   else 0.0) +
                               (u[j+1, i] if j < n-1 else 0.0) +
                               (u[j-1, i] if j > 0 else 0.0))
```

Data Movement dominates…

25,000x25,00 grid for 10 time steps
- Xeon Platinum 8480+: 67.6 secs
- Nvidia V100: 22.6 secs

```
# Loop over time steps
for _ in range(nsteps):
    # solve over spatial domain for step t
    solve(n, alpha, dx, dt, u, u_tmp)

    # Array swap to get ready for next step
    u, u_tmp = u_tmp, u
```

Typically, many time steps!

solve() function uses this context:
with openmp ("target loop collapse(2) map(tofrom: u, u_tmp)"):

For each iteration, copy from device $(2*N^2)*$sizeof(TYPE) bytes

- We need to keep data resident on the device *between* target regions
- We need a way to manage the device data environment across iterations.

Solution: Reference swapping in action

```
with openmp ("target enter data map(to: u, u_tmp)"):
    pass

for _ in range(nsteps):

    solve(n, alpha, dx, dt, u, u_tmp);

    # Array swap to get ready for next step
    u, u_tmp = u_tmp, u

with openmp ("target exit data map(from: u)"):
    pass
```

Copy data to device before iteration loop

Change solve() routine to remove map clauses:
`with openmp ("target loop collapse(2)”)`

Copy data from device after iteration loop

25,000x25,00 grid for 10 time steps

- Xeon Platinum 8480+ default data movement: 67.6 secs
- Nvidia V100 default data movement: 22.6 secs
- Nvidia V100 target enter/exit: 1.2 secs

Summary

- Parallel programming is here to stay. If you don't need it today, you will eventually. Fortunately, it's really fun.
- Software outlives hardware. Do not let a vendor lock you in to their platform. Portability must be non-negotiable.
- There are too many parallel programming models for python. Focus on the core principles and fundamental design patterns. Don't wear yourself out chasing the latest fad.

My Greenlandic skin-on-frame kayak in the middle of Budd Inlet during a negative tide