Multithreaded parallel Python through OpenMP support in Numba

Tim Mattson (University of Bristol and Merly.ai … but mostly retired)

tim@timmattson.com

Acknowledgement: The rest of the PyOMP team ….

Giorgis Georgakoudis (LLNL), Todd Anderson (bodo.ai), and Stuart Archibald (anaconda)

merly.ai

This talk in one slide

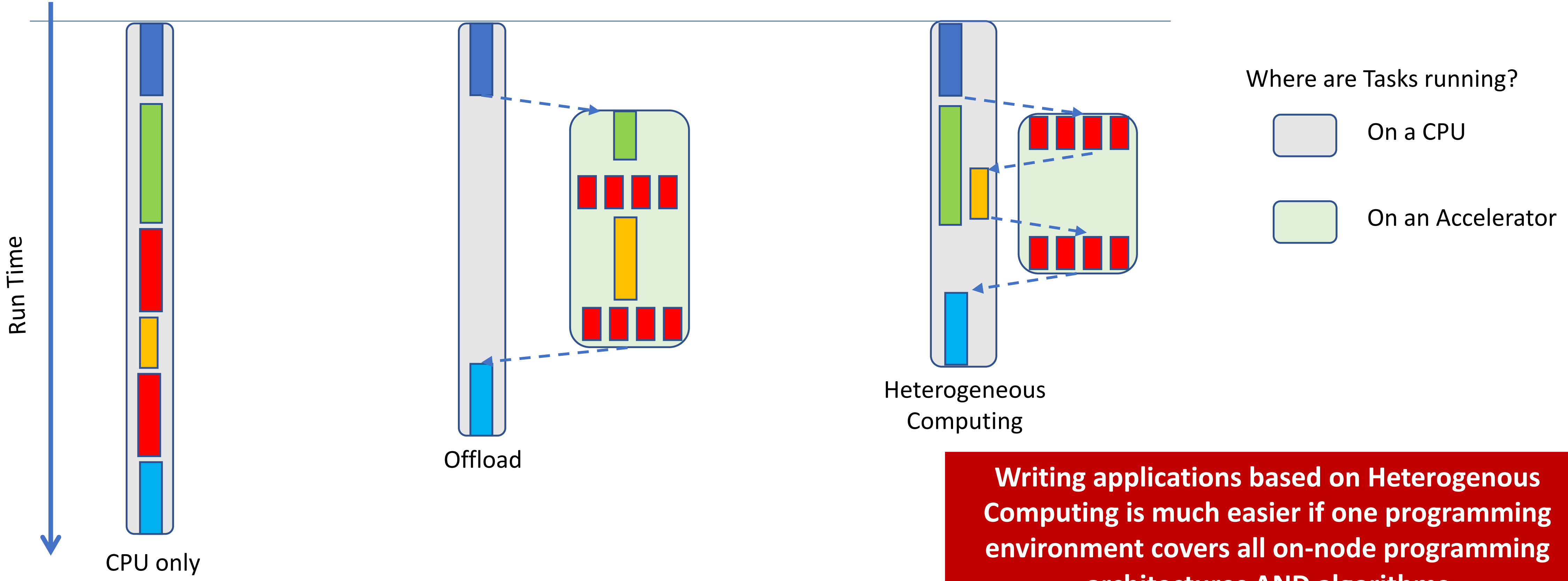
- Given that …
  - **Tasks are <u>NOT</u> best for everything.**
  - **Fragmenting the space of parallel APIs is bad**.
  - **OpenMP is the most popular parallel programing model.**
  - **OpenMP covers the key patterns of task parallel programming**.
  - **One programming model mapping onto multiple programing languages is key**.
  - **PyOMP is cool!**
- Which API should applications developers converge around?

Warning: This talk is part of a panel. To foster discussion, my views are presented aggressively without nuance.

API: Application Programming Interface

No single processor is best at everything

- The idea that you should move everything to the GPU makes no sense
- **Heterogeneous Computing**: Run sub-problems in parallel on the hardware best suited to them.



Writing applications based on Heterogenous Computing is much easier if one programming environment covers all on-node programming architectures AND algorithms

In the early days of parallel computing, we were obsessed with finding the “right” parallel programming environment

ABCPL
ACE
ACT++
Active messages
Adl
Adsmith
ADDAP
AFAPI
ALWAN
AM
AMDC
AppLeS
Amoeba
ARTS
Athapascan-0b
Aurora
Automap
bb_threads
Blaze
BSP
BlockComm
C*.
"C* in C
C**
CarlOS
Cashmere
C4
CC++
Chu
Charlotte
Charm
Charm++
Cid
Cilk
CM-Fortran
Converse
Code
COOL
CORRELATE
CPS
CRL
CSP
Cthreads
CUMULVS
DAGGER
DAPPLE
Data Parallel C
DC++
DCE++
DDD
DICE.
DIPC
DOLIB
DOME
DOSMOS.
DRL
DSM-Threads
Ease .
ECO
Eiffel
Eilean
Emerald
EPL
Excalibur
Express
Falcon
Filaments
FM
FLASH
The FORCE
Fork
Fortran-M
FX
GA
GAMMA
Glenda
GLU
GUARD
HAsL.
Haskell
HPC++
JAVAR.
HORUS
HPC
HPF
IMPACT
ISIS.
JAVAR
JADE
Java RMI
javaPG
JavaSpace
JIDL
Joyce
Khoros
Karma
KOAN/Fortran-S
LAM
Lilac
Linda
JADA
WWWinda
ISETL-Linda
ParLin
Eilean
P4-Linda
Glenda
POSYBL
Objective-Linda
LiPS
Locust
Lparx
Lucid
Maisie
Manifold
Mentat
Legion
Meta Chaos
Midway
Millipede
CparPar
Mirage
MpC
MOSIX
Modula-P
Modula-2*
Multipol
MPI
MPC++
Munin
Nano-Threads
NESL
NetClasses++
Nexus
Nimrod
NOW
Objective Linda
Occam
Omega
OpenMP
Orca
OOF90
P++
P3L
p4-Linda
Pablo
PADE
PADRE
Panda
Papers
AFAPI.
Para++
Paradigm
Parafrase2
Paralation
Parallel-C++
Parallaxis
ParC
ParLib++
ParLin
Parmacs
Parti
pC
pC++
PCN
PCP:
PH
PEACE
PCU
PET
PETSc
PENNY
Phosphorus
POET.
Polaris
POOMA
POOL-T
PRESTO
P-RIO
Prospero
Proteus
QPC++
PVM
PSI
PSDM
Quake
Quark
Quick Threads
Sage++
SCANDAL
SAM
pC++
SCHEDULE
SciTL
POET
SDDA.
SHMEM
SIMPLE
Sina
SISAL.
distributed smalltalk
SMI.
SONiC
Split-C.
SR
Sthreads
Strand.
SUIF.
Synergy
Telegrphos
SuperPascal
TCGMSG.
Threads.h++.
TreadMarks
TRAPPER
uC++
UNITY
UC
V
ViC*
Visifold V-NUS
VPE
Win32 threads
WinPar
WWWinda
XENOOPS
XPC
Zounds
ZPL

Parallel program environments in the 90’s

In the early days of parallel computing, we were obsessed with finding the “right” parallel programming environment

ABCPL
ACE
ACT++
Active messages
Adl
Adsmith
ADDAP
AFAPI
ALWAN
AM
AMDC
AppLeS
Amoeba
ARTS
Athapas
Aurora
Automap
bb_threa
Blaze
BSP
BlockCo
C*.
"C* in C
C**
CarlOS
Cashmer
C4
CC++
Chu
Charlotte
Charm
Charm++
Cid
Cilk
CM-Fortran
Converse
Code
COOL

CORRELATE
CPS
CRL
CSP
Cthreads
CUMULVS
FLASH
The FORCE
Fork
Fortran-M
FX
GA
GAMMA
Glenda

GLU
GUARD
HAsL.
Haskell
HPC++
JAVAR.
Glenda
POSYBL
Objective-Linda
LiPS
Locust
Lparx
Lucid
Maisie
Manifold

Mentat
Legion
Meta Chaos
Midway
Millipede
CparPar
Pablo
PADE
PADRE
Panda
Papers
AFAPI.
Para++
Paradigm

Parafrase2
Paralation
Parallel-C++
Parallaxis
ParC
ParLib++
PSI
PSDM
Quake
Quark
Quick Threads
Sage++
SCANDAL
SAM

pC++
SCHEDULE
SciTL
POET
SDDA.
SHMEM
Visifold V-NUS
VPE
Win32 threads
WinPar
WWWinda
XENOOPS
XPC
Zounds
ZPL

Having such a huge set of options did tremendous damage.

Supporting programming models is a zero-sum game. Time spent working on a new model means time NOT spent getting established to work well.

This was a headache to application developers who need a small number of models that just work … everywhere and supported by their systems vendors

As we came out of the 1990’s sanity ruled and the HPC applications community converged around two programming environments …. MPI and OpenMP

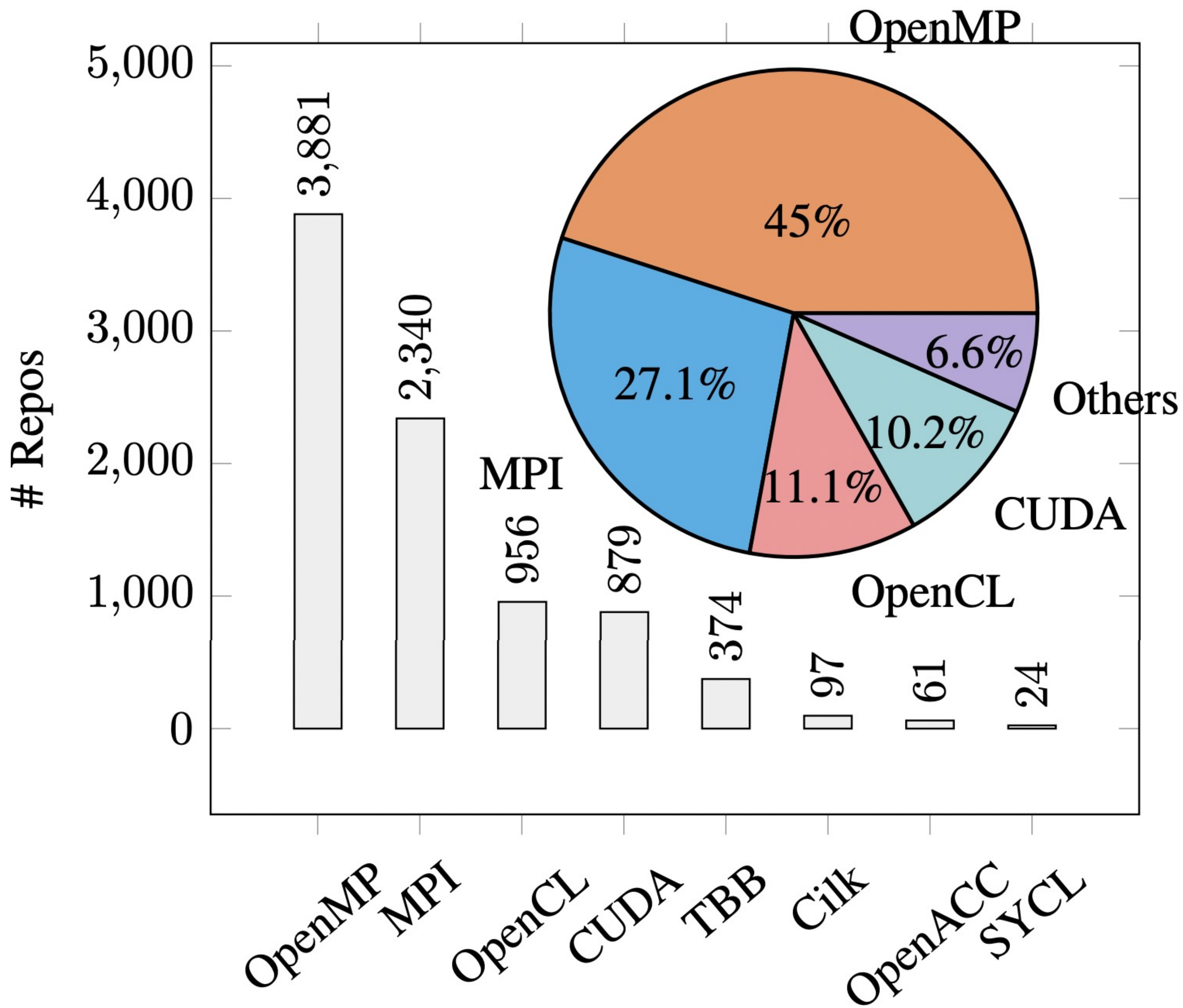
The lives of HPC applications developers improved tremendously!!!!

I fear a new generation of parallel programmers have forgotten this lesson. Can we please NOT screw things up again?

Parallel program environments in the 90’s

OpenMP is the most popular parallel programing model in use today

In a dataset (HPCorpus) of all C/C++/Fortan github repositories from 2013-2023, OpenMP was found to be the most popular parallel programming model



Aggregate numbers over all repositories from 2013 to 2023

Quantifying OpenMP: Statistical insights into usage and adoption, Tal Kadosh, et al., HPEC'2023, https://arxiv.org/abs/2308.08002

Note: since we did not collect files with .cu or .cuf suffices, we undercounted CUDA usage in HPCorpus.

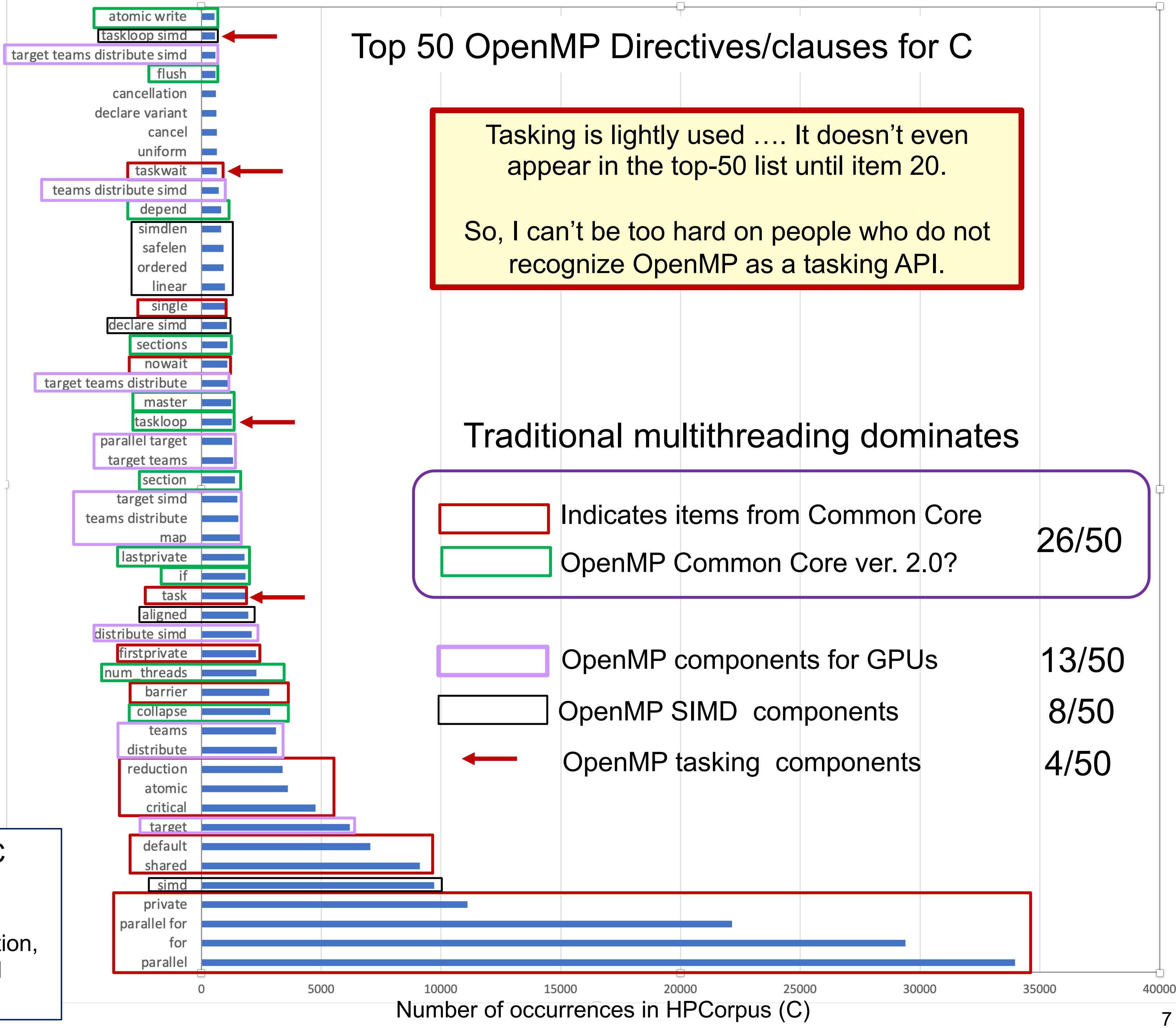
What are people actually using from OpenMP

With the HPCorpus* dataset, we finally have hard-data to analyze what “should” be in the common core.

This data was constructed by summing up counts for different directives and clauses across time from 2013 to the middle of 2023.

HPCorpus … a data set created by scraping ”all” HPC codes from github written in C, C++ and Fortran.

Quantifying OpenMP: Statistical Insights into Usage and Adoption, Tal Kadosh, Niranjan Hasabnis, Tim Mattson, Yuval Pinter, and Gal Oren, IEEE HPEC 2023



Comparing tasking systems*

OpenMP is a general-purpose task-based programming model.

OpenMP supports the full range of task-based algorithms other than those that depend on

- Distributed memory
- Task resiliency
- Futures

… and with Tech Readiness level 9, it's suitable for serious application work.

	Architectural			Task System				Management				Eng.	
	Communication Model	Distributed Memory	Heterogeneity	Graph Structure	Task Partitioning	Result Handling	Task Cancellation	Worker Management	Resilience Management	Work Mapping	Synchronization	Technological Readiness	Implementation Type
C++ STL	smem	×	×	dag	×	i/e	×	i	×	i/e	e	9	Library
TBB	smem	×	×	tree	×	i	✓	i	×	i	i	8	Library
HPX	gas	i	e	dag	✓	e	✓	i/e	×	i/e	e	6	Library
Legion	gas	i	e	tree	✓	e	×	i	×	i/e	e	4	Library
PaRSEC	msg	e	e	dag	×	e	✓	i	✓	i/e	i	4	Library
OpenMP	smem	×	i/e	dag/tree	×	i	✓	i/e	×	i	i/e	9	Extension
Charm++	gas	i	e	dag	✓	i/e	×	i	✓	i/e	e	6	Extension
OmpSs	smem	×	i	dag	×	i	×	i	✓	i	i/e	5	Extension
AllScale	gas	i	i	dag	✓	i/e	×	i	✓	i	i/e	3	Extension
StarPU	msg	e	e	dag	✓	i	×	i	×	i/e	e	5	Extension
Cilk Plus	smem	×	×	tree	×	i	×	i	×	i	e	8	Lang.
Chapel	gas	i	i	dag	✓	i	×	i	×	i/e	e	5	Lang.
X10	gas	i	i	dag	✓	i	×	i	✓	i/e	e	5	Lang.

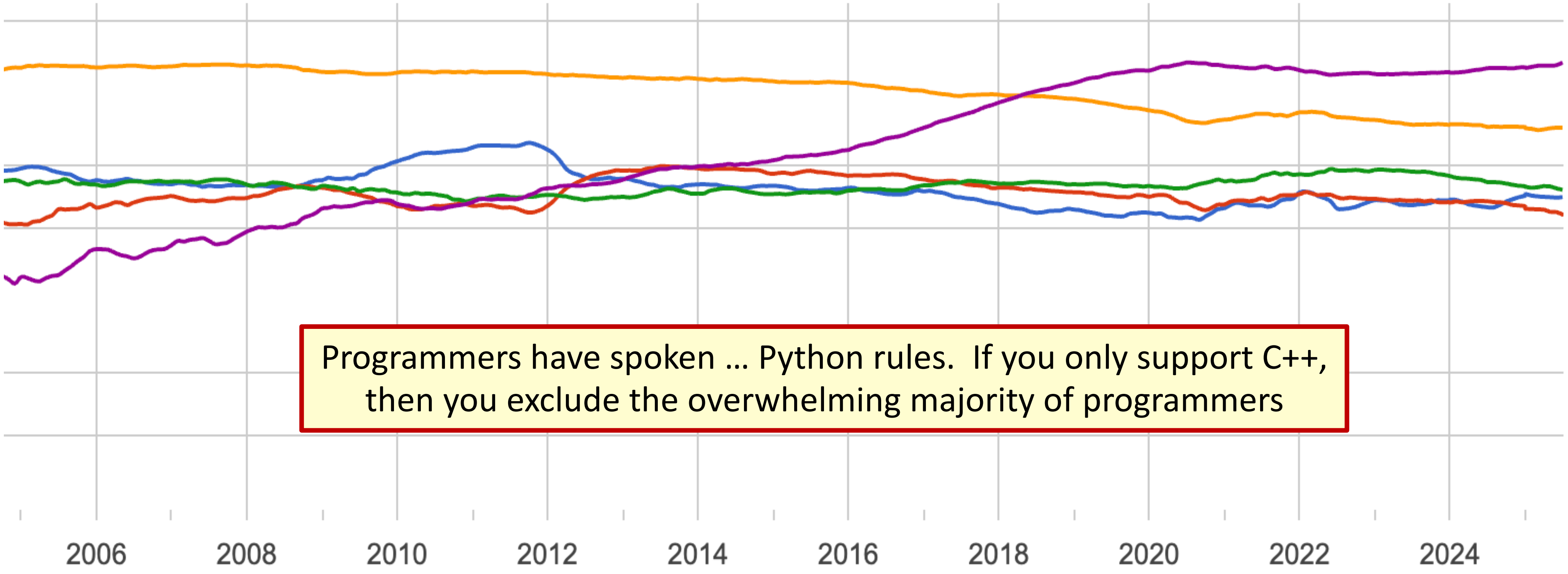
OpenMP notes

- Free agent threads in OpenMP 6.0 supports implicit worker management
- OpenMP 6.0 added reusable static taskgraphs to reduce task management overhead
- Explicit GPU programming in OpenMP is fully integrated with tasks
- Recursive tasks plus taskwait supports trees. The depend clause on task supports DAGs

*A taxonomy of task-based parallel programming technologies for High Performance Computing, Peter Thoman, et. al., J.. Supercomput (2018) https://doi.org/10.1007/s11227-018-2238-4

Python is number One!

Popularity of Programming Languages (PyPl)



Programmers have spoken … Python rules. If you only support C++, then you exclude the overwhelming majority of programmers

Top 5 Languages

Language	Share
Python	31.47 %
Java	15.22%
JavaScript	7.65 %
C/C++	7.05%
C#	5.81%

Taskflow and HPX	C++
OpenCilk	C++, C
OpenMP	C++, C, Fortran, Python

https://pypl.github.io/PYPL.html.

Vertical axis is log(PyPl score)

… So perhaps best way to bring parallel task-based computing to the masses would be to combine OpenMP and Python?

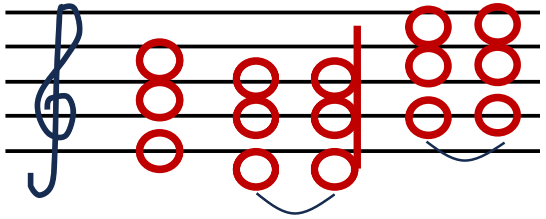
PyOMP

Multithreaded parallel Python through OpenMP support in Numba
Todd Anderson, Timothy G. Mattson, SciPy 2021. http://conference.scipy.org/proceedings/scipy2021/tim_mattson.html

PyOMP: Multithreaded Parallel Programming in Python
Timothy G. Mattson, Todd A. Anderson, Giorgis Georgakoudis, Computing in Science and Engineering, IEEE, November/December 2021

PyOMP: Programming GPUs with OpenMP and Python
Giorgis Georgakouis, Todd A. Anderson, Stuart Archibald, Bronis de Supinski, and Timothy G. Mattson. High Performance Python for Science at Scale workshop at SC24, 2024

Pythonic OpenMP in three-part harmony *



- Incorporated into the numba JIT compiler. The code is JIT'ed into LLVM and therefore avoids the Global Interpreter Lock (GIL) and supports parallel computing with multiple threads.

- Numpy is the standard module used in scientific computing with Python. Hence, PyOMP is optimized to with numpy arrays.

- OpenMP managed through a context manager (that is, a with statement).

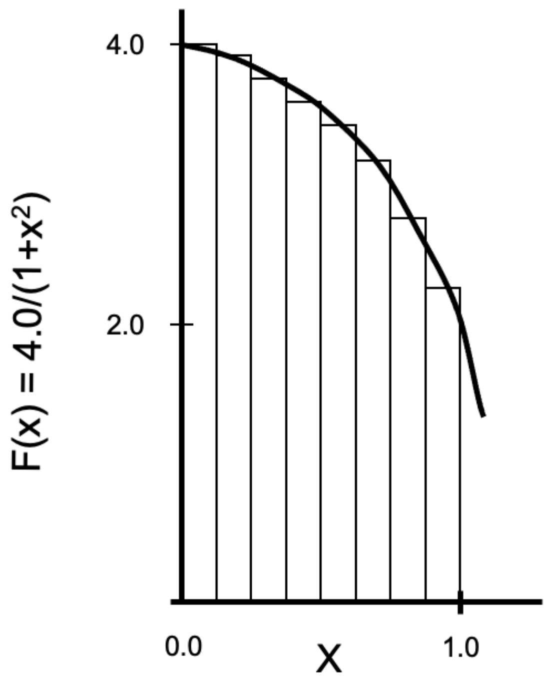
JIT: Just In Time Compilation

*Opening chord progression from the opera *Einstein on the Beach* (Knee Play 1) by Philip Glass

PyOMP by example …

We will understand PyOMP by considering the three fundamental design patterns of OpenMP (**Loop parallelism**, **SPMD**, and **divide and conquer**) applied to the following problem

Numerical Integration (the *hello world* program of parallel computing)



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Each rectangle: width Δx, height $F(x_i)$ at i^{th} interval midpoint.

```
def piFunc(NumSteps):
   step=1.0/NumSteps
   sum = 0.0
   x = 0.5
   for i in range(NumSteps):
      x+=step
      sum += 4.0/(1.0+x*x)
   pi=step*sum
   return pi
```

Loop Parallelism code

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

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

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

    pi = step*sum
    return pi

pi = piFunc(100000000)
```

OpenMP constructs managed through an *openmp* context manager.

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

Python's implicit data management mapped onto OpenMP. Default rules:

- Variables referenced outside the OpenMP construct are shared
- Variables that only appear inside a construct are private
- Python for technical applications typically based on Numpy arrays, so PyOMP focusses on numpy arrays as well.

OpenMP data environment clauses are supported in PyOMP

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)
```

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.

Divide and conquer (with explicit tasks)

```python
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):
        sum = 0.0
        for i in range(Nstart,Nfinish):
            x= (i+0.5)*step
            sum += 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"):
            sum = sum1 + sum2
    return sum
```

Solve

Split

Merge

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

    pi = step*sum
    return step*sum

pi = piFunc(100000000)
```

Fork threads and launch the computation

Numerical Integration results in seconds … lower is better

Threads	PyOMP			C/OpenMP		
	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

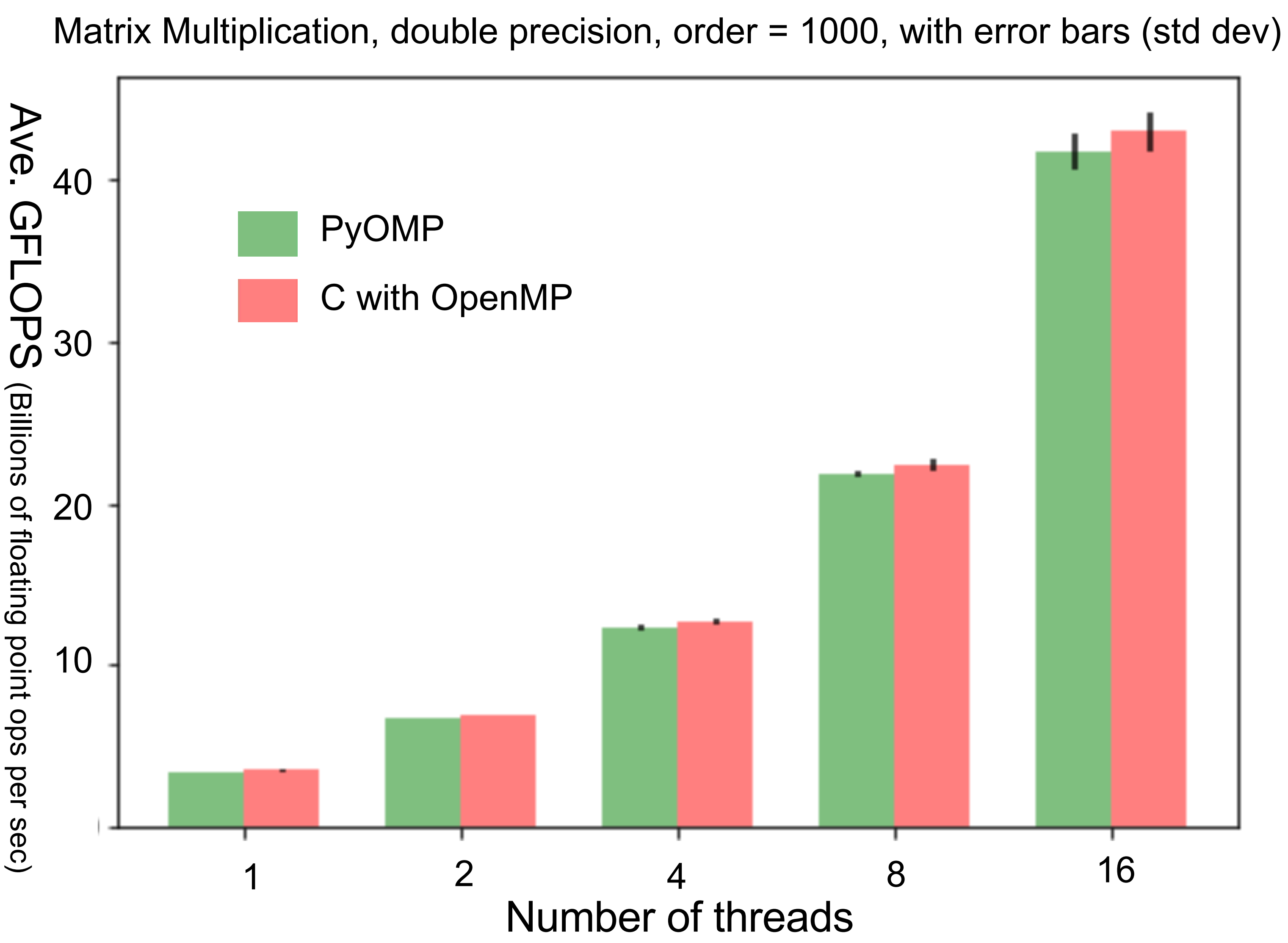
Ran each case 5 times and kept the minimum time. **JIT time is not included** for PyOMP (it was about 1.5 seconds)

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.

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 GPU

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

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

    with openmp ("target teams loop private(x) reduction(+:sum)"):
        for i in range(NumSteps):
            x = (i+0.5)*step
            sum += 4.0/(1.0 + x*x)

    pi = step*sum
    return pi

pi = piFunc(100000000)
```

OpenMP constructs managed through the *with* context manager.

Map the loop onto a 1D index space … the loop body defines the kernel function

PyOMP is easy to install and use

- Conda one-line installation

```
conda install -c python-for-hpc -c conda-forge pyomp
```

- PyPi package installation

```
pip install pyomp
```

- Fast ways to try
  - Binder: https://mybinder.org/v2/gh/Python-for-HPC/binder/HEAD
  - Docker: `docker pull ghcr.io/python-for-hpc/pyomp:latest`

Open Source code on github: https://github.com/Python-for-HPC/PyOMP

This talk in one slide

- Given that …
  - **Tasks are NOT best for everything.** We need a single node-level programming model that does it all … traditional multithreading, GPU-programming, and task level parallelism.
  - **Fragmenting the space of parallel APIs is bad**. HPC is facing an existential challenge in the face of AI. If we make vendors chase multiple parallel programming models, we in the long run damage ourselves.
  - **OpenMP is the most popular parallel programing model.**
  - **OpenMP covers the key patterns of task parallel programming**. We lack futures and distributed computing, but cover the other classic task patterns.
  - **One programming model mapping onto multiple programing languages is key**. C++ is great, but there is a lot of code outside C++ in HPC. Python will become the primary language of HPC!
  - **PyOMP is cool!** The performance you expect from OpenMP but in Python
- Which API should applications developers converge around?
  - OpenMP is the "once and future" choice for task-level parallelism. Taskflow, HPX and Cilk are great research vehicles. But to impact the real world and guide us into a tasky future, OpenMP is the right choice.

Hopefully these points are clear and obvious.

And if not, I look forward to our discussions

API: Application Programming Interface

Backup Content

- GPU Programming with PyOMP
- How is PyOMP Implemented?
- Python and the future of HPC
- Programming ecosystem fragmentation and choice overload

Loop Parallelism code naturally maps onto the GPU

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

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

    with openmp (”target teams loop private(x) reduction(+:sum)"):
        for i in range(NumSteps):
            x = (i+0.5)*step
            sum += 4.0/(1.0 + x*x)

    pi = step*sum
    return pi

pi = piFunc(100000000)
```

OpenMP constructs managed through the *with* context manager.

Map the loop onto a 1D index space … the loop body defines the kernel function

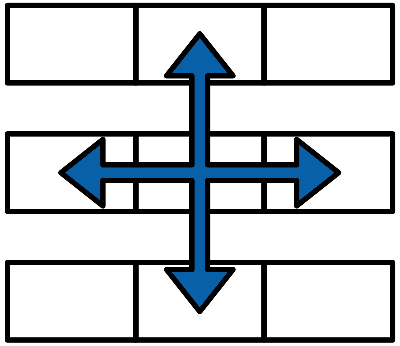
5-point stencil: Heat diffusion problem

$$\frac{\partial u}{\partial t} - \alpha \nabla^2 u = 0$$

```
# 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
```

$$\frac{\partial u}{\partial t} \approx \frac{u(t+1,x,y) - u(t,x,y)}{dt}$$

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u(t,x+1,y) - 2u(t,x,y) + u(t,x-1,y)}{dx^2}$$



5-point stencil: solve kernel

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

```
@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))
```

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…

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))
```

There can be many time steps …

For each step, $(2*N^2)$*sizeof(TYPE) bytes move between the host and the device

- 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: Explicitly manage the device data environment

```
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

For PowerPoint clarity, we packed the finite difference code into the function solve()

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

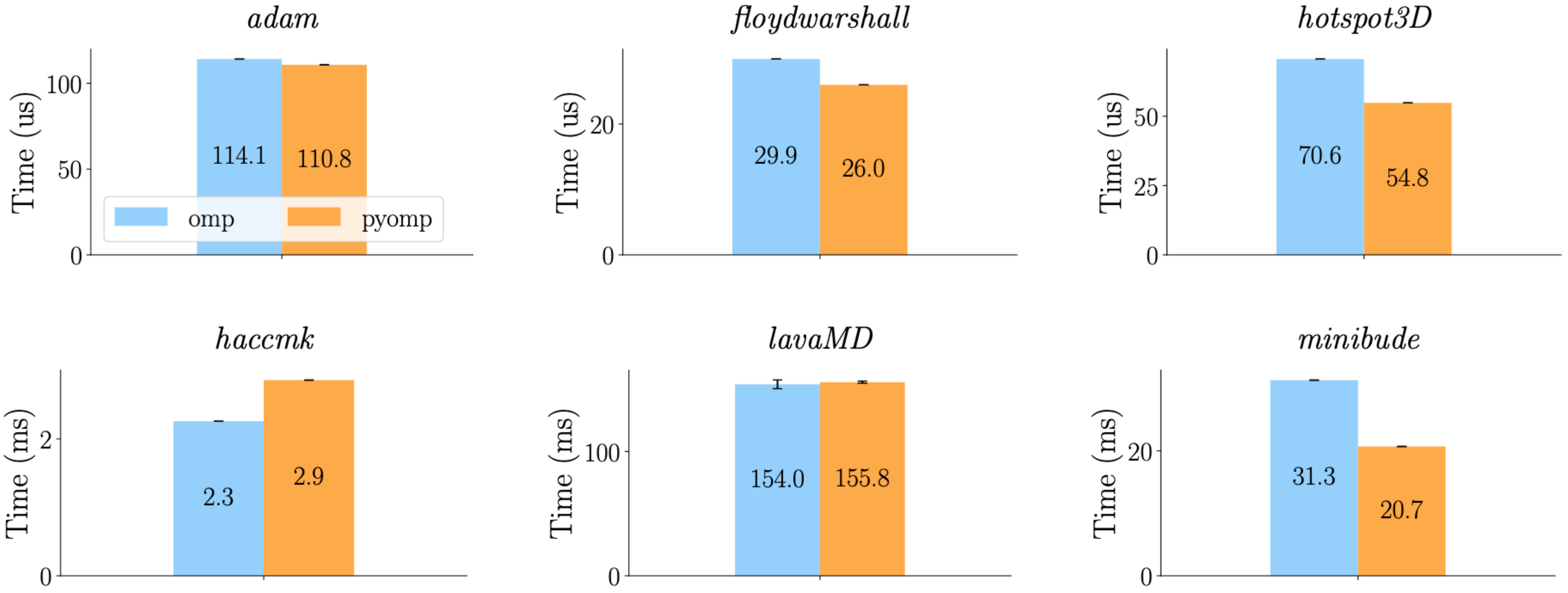
Copy data from device after iteration loop

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

PyOMP HECBench GPU results

Program	Description	Input
adam	Adaptive moment estimation stochastic optimization for machine learning	10000 200 100
floydwarshall	Floyd-Warshall path finding algorithm	1024 100 16
haccmk	HACC cosmology code micro-kernel	1000
hotspot3D	Thermal modeling for 3D integrated circuits stencil computation	512 8 5000 power_512x8 temp_512x8
lavaMD	Molecular dynamics	-boxes1d 30
miniBUDE	Ligand-protein docking	–deck data/bm1 –wgsize 256 –iterations 100



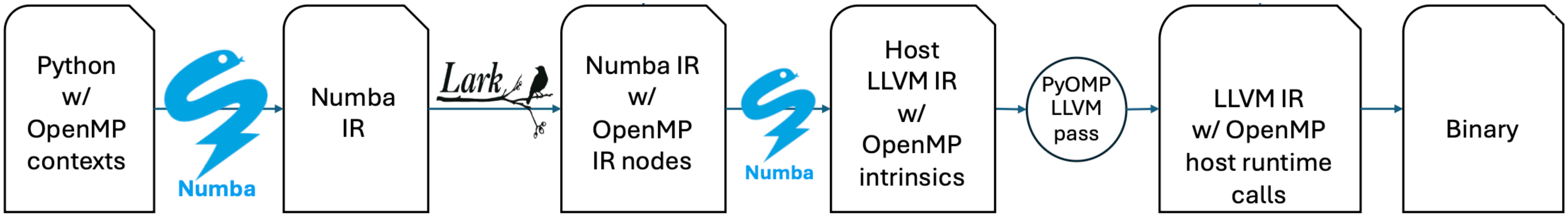
Details in an IWOMP’25 paper submission

AMD EPYC 7763 CPU with an NVIDIA A100 GPU with 80 GB or memory.
Python 3.9.18, Numba 0.57, llvm-lite 0.40, CUDA 12.2 with driver version 525.105.17

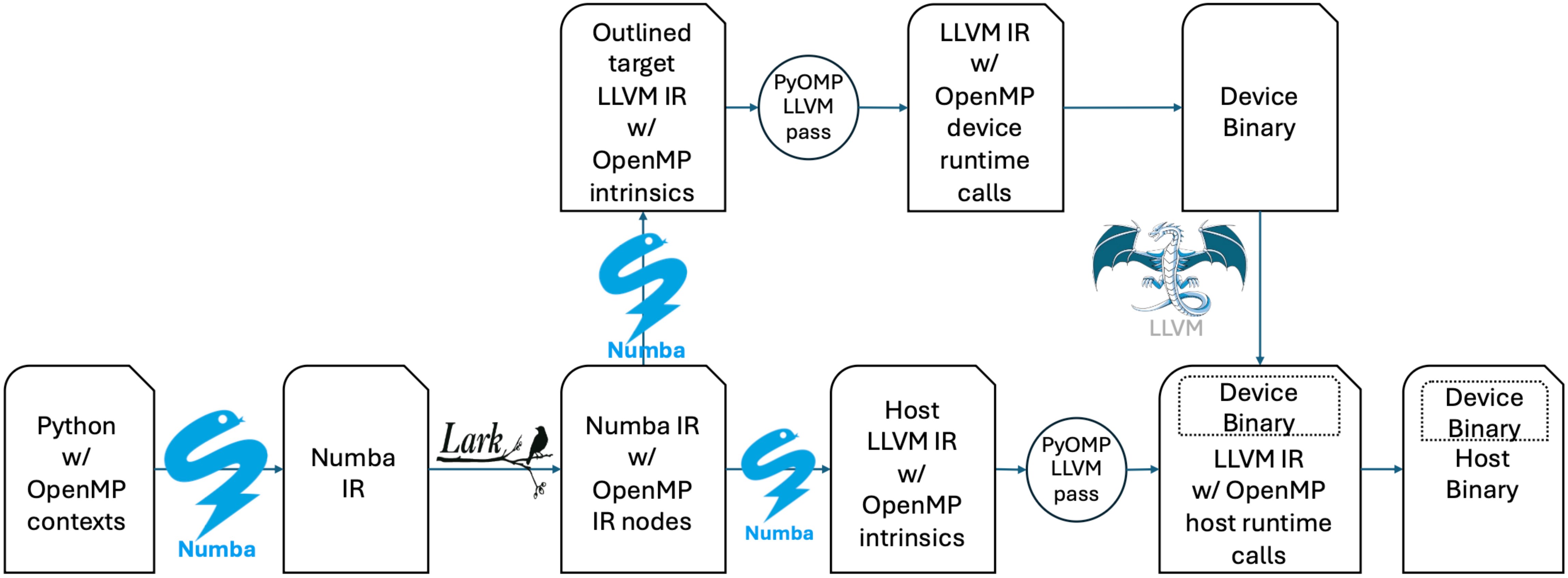
Backup Content

- GPU Programming with PyOMP
- How is PyOMP Implemented?
- Python and the future of HPC
- Programming ecosystem fragmentation and choice overload

PyOMP implementation: CPU



PyOMP implementation: CPU + GPU



PyOMP: a Numba extension for upgradeability and maintainability

- Depends on Numba as a compiler toolkit
  - Similar to numba-cuda, numba-hip

- Uses Numba's LLVM dependencies
  - llvmlite: provides python bindings for the LLVM API (Currently supports LLVM 14.x – We may need to patch PyOMP when Numba moves to LLVM 18/19)

- Tested with Numba 0.57.x, 0.58.x
  - Architectures: linux-64 (x86_64), osx-arm64 (mac), linux-arm64, linux-ppc64le

PyOMP piggybacks on the off-the-shelf Numba ecosystem.

We don't need to do any extra work to adapt as new versions of Numba are released

Backup Content

- GPU Programming with PyOMP
- How is PyOMP Implemented?
- Python and the future of HPC
- Programming ecosystem fragmentation and choice overload

What HPC old-timers think of Python?

(from the paper, There's plenty of room at the top. Leiserson et. al. Science vol. 368, June 2020).

They used matrix multiplication to explore the connection between software and performance

```
for I in range(4096):
  for j in range(4096):
    for k in range (4096):
      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

What HPC old-timers think of Python

(from the paper, There's plenty of room at the top. Leiserson et. al. Science vol. 368, June 2020).

They used matrix multiplication to explore the connection between software and performance

```
for I in range(4096):
   for j in range(4096):
      for k in range (4096):
         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.

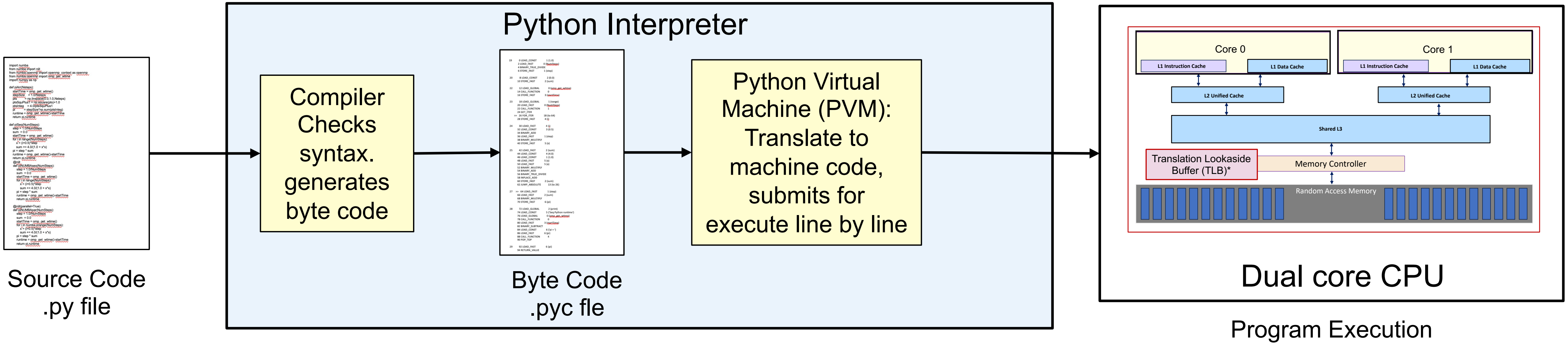
Version	Implementation	Running time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak (%)
1	Python	25,552.48	0.005	1	—	0.00
2	Java	2,372.68	0.058	11	10.8	0.01
3	C	542.67	0.253	[illegible]	[illegible]	0.03
4	Parallel loops	69.80	1.969	[illegible]	[illegible]	0.24
5	Parallel divide and conquer	3.80	36.180	[illegible]	[illegible]	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

Python performance is a joke. No serious HPC programmer would **EVER** use Python

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 … dynamically compiled

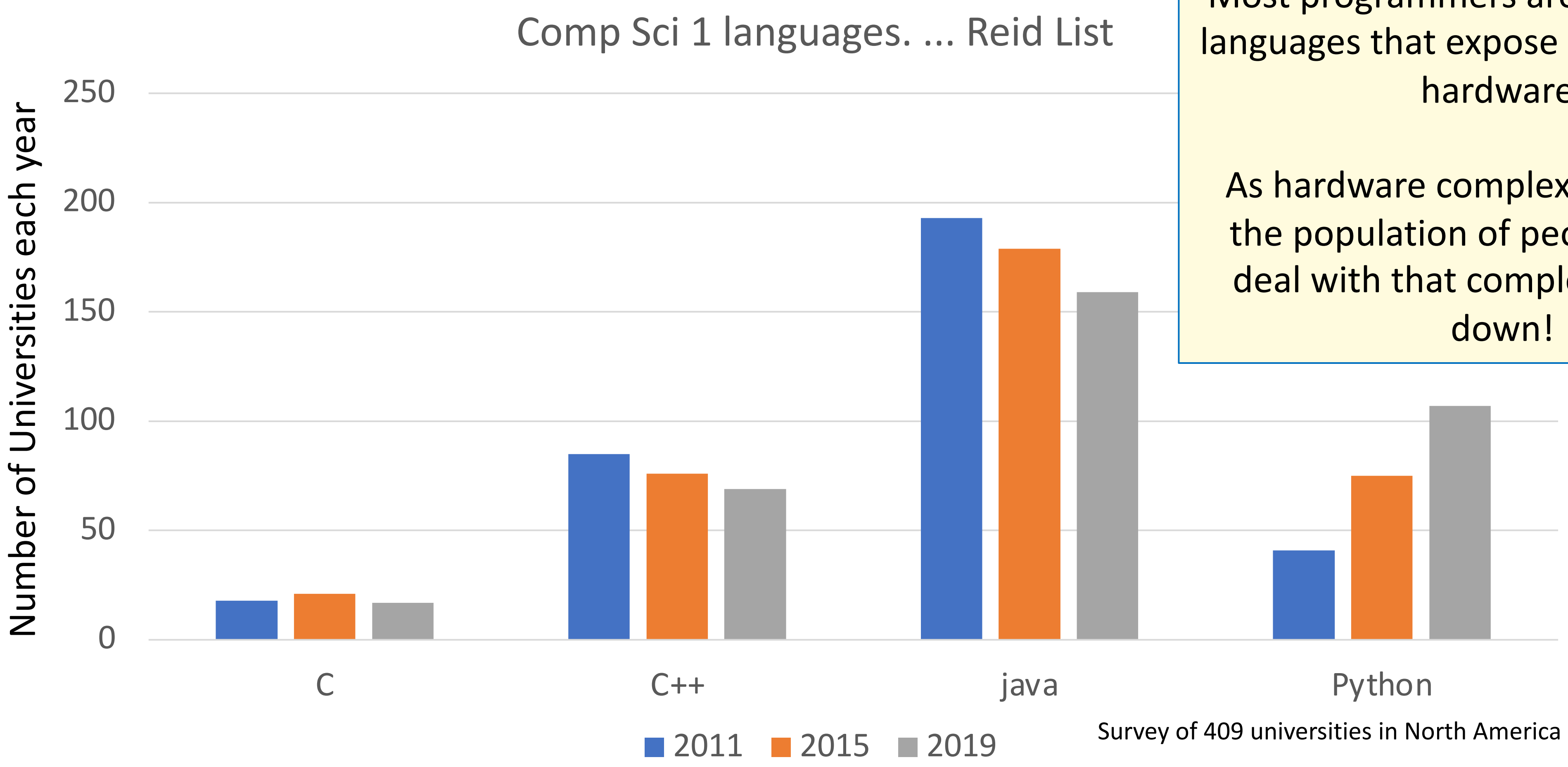


- What if I want my Python program to run in parallel. Does that work?
- Not really. Python has a **Global Interpreter lock** (**GIL**). This is a mutex (mutual exclusion lock) to allow only one thread at a time can make forward progress.

Primary Language used in first year, Computer Science Courses

Most programmers are NOT learning languages that expose features of the hardware.

As hardware complexity increases, the population of people who can deal with that complexity is going down!



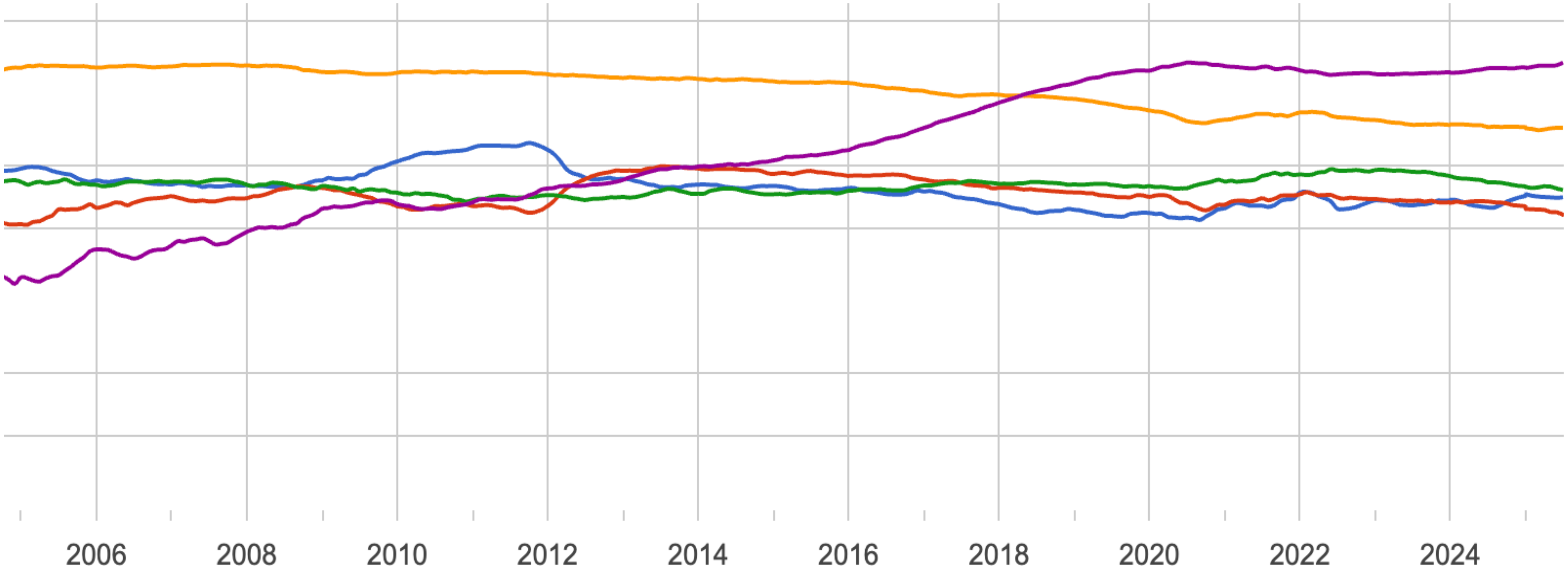
Survey of 409 universities in North America

The Reid List tracks a large sample of North American Universities and the languages they use in teaching.

The Reid List was started by Richard Reid in the 1990s. He has retired but others are carrying on the tradition. The above data comes from Trends Of Commonly Used Programming Languages in CS1 And CS2 Learning, Robert M. Siegfried, Katherine G. Herbert-Berger, Kees Leune, Jason P. Siegfried, The 16th International Conference on Computer Science & Education (ICCSE 2021) August 18-20, 2021.

Python is number One!

Popularity of Programming Languages (PyPl)



Top 5 Languages

Language	Share
Python	31.47 %
Java	15.22%
JavaScript	7.65 %
C/C++	7.05%
C#	5.81%

Programmers have spoken … Python rules. Old-timers (like me) need to stop being such arrogant snobs and help make Python a first class HPC language

https://pypl.github.io/PYPL.html. Vertical axis is log(PyPl score)

Backup Content

- GPU Programming with PyOMP
- How is PyOMP Implemented?
- Python and the future of HPC
- Programming ecosystem fragmentation and choice overload

In the early days of parallel computing, we were obsessed with finding the “right” parallel programming environment

ABCPL
ACE
ACT++
Active messages
Adl
Adsmith
ADDAP
AFAPI
ALWAN
AM
AMDC
AppLeS
Amoeba
ARTS
Athapascan-0b
Aurora
Automap
bb_threads
Blaze
BSP
BlockComm
C*.
"C* in C
C**
CarlOS
Cashmere
C4
CC++
Chu
Charlotte
Charm
Charm++
Cid
Cilk
CM-Fortran
Converse
Code
COOL

CORRELATE
CPS
CRL
CSP
Cthreads
CUMULVS
DAGGER
DAPPLE
Data Parallel C
DC++
DCE++
DDD
DICE.
DIPC
DOLIB
DOME
DOSMOS.
DRL
DSM-Threads
Ease .
ECO
Eiffel
Eilean
Emerald
EPL
Excalibur
Express
Falcon
Filaments
FM
FLASH
The FORCE
Fork
Fortran-M
FX
GA
GAMMA
Glenda

GLU
GUARD
HAsL.
Haskell
HPC++
JAVAR.
HORUS
HPC
HPF
IMPACT
ISIS.
JAVAR
JADE
Java RMI
javaPG
JavaSpace
JIDL
Joyce
Khoros
Karma
KOAN/Fortran-S
LAM
Lilac
Linda
JADA
WWWinda
ISETL-Linda
ParLin
Eilean
P4-Linda
Glenda
POSYBL
Objective-Linda
LiPS
Locust
Lparx
Lucid
Maisie
Manifold

Mentat
Legion
Meta Chaos
Midway
Millipede
CparPar
Mirage
MpC
MOSIX
Modula-P
Modula-2*
Multipol
MPI
MPC++
Munin
Nano-Threads
NESL
NetClasses++
Nexus
Nimrod
NOW
Objective Linda
Occam
Omega
OpenMP
Orca
OOF90
P++
P3L
p4-Linda
Pablo
PADE
PADRE
Panda
Papers
AFAPI.
Para++
Paradigm

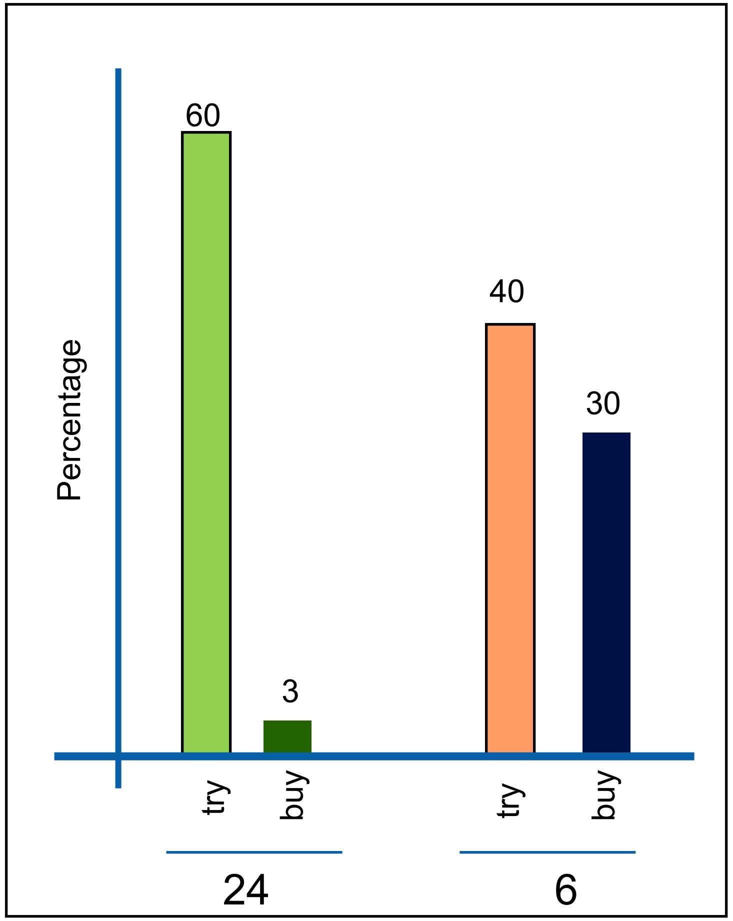
Parafrase2
Paralation
Parallel-C++
Parallaxis
ParC
ParLib++
ParLin
Parmacs
Parti
pC
pC++
PCN
PCP:
PH
PEACE
PCU
PET
PETSc
PENNY
Phosphorus
POET.
Polaris
POOMA
POOL-T
PRESTO
P-RIO
Prospero
Proteus
QPC++
PVM
PSI
PSDM
Quake
Quark
Quick Threads
Sage++
SCANDAL
SAM

pC++
SCHEDULE
SciTL
POET
SDDA.
SHMEM
SIMPLE
Sina
SISAL.
distributed smalltalk
SMI.
SONiC
Split-C.
SR
Sthreads
Strand.
SUIF.
Synergy
Telegrphos
SuperPascal
TCGMSG.
Threads.h++.
TreadMarks
TRAPPER
uC++
UNITY
UC
V
ViC*
Visifold V-NUS
VPE
Win32 threads
WinPar
WWWinda
XENOOPS
XPC
Zounds
ZPL

Parallel program environments in the 90’s

Language obsessions: More isn't always better

- The Draeger Grocery Store experiment and consumer choice:
  - Two Jam-displays with coupons for a discount on purchase.
    - 24 different Jam's
    - 6 different Jam's
  - How many stopped by to try samples at the display?
  - Of those who "tried", how many bought jam?



The findings from this study show that an extensive array of options can at first seem highly appealing to consumers, yet can reduce their subsequent motivation to purchase the product.

Iyengar, Sheena S., & Lepper, Mark (2000). When choice is demotivating: Can one desire too much of a good thing? *Journal of Personality and Social Psychology*, 76, 995-1006.

In the early days of parallel computing, we were obsessed with finding the "right" parallel programming environment

ABCPL
ACE
ACT++
Active messages
Adl
Adsmith
ADDAP
AFAPI
ALWAN
AM
AMDC
AppLeS
Amoeba
ARTS
Athapascan-0b
Aurora
Automap
bb_threads
Blaze
BSP
BlockComm
C*.
"C* in C
C**
CarlOS
Cashmere
C4
CC++
Chu
Charlotte
Charm
Charm++
Cid
Cilk
CM-Fortran
Converse
Code
COOL

CORRELATE
CPS
CRL
CSP
Cthreads
CUMULVS
DAGGER
DAPPLE
Data Parallel C
DC++
DCE++
DDD
DICE.
DIPC
DOLIB
DOME
DOSMOS.
DRL
DSM-Threads
Ease .
Excalibur
Express
Falcon
Filaments
FM
FLASH
The FORCE
Fork
Fortran-M
FX
GA
GAMMA
Glenda

GLU
GUARD
HAsL.
Haskell
HPC++
JAVAR.
HORUS
HPC
HPF
IMPACT
ISIS.
JAVAR
JADE
Java RMI
javaPG
JavaSpace
JIDL
Joyce
Khoros
Karma
WWWinda
ISETL-Linda
ParLin
Eilean
P4-Linda
Glenda
POSYBL
Objective-Linda
LiPS
Locust
Lparx
Lucid
Maisie
Manifold

Mentat
Legion
Meta Chaos
Midway
Millipede
CparPar
Mirage
MpC
MOSIX
Modula-P
Modula-2*
Multipol
MPI
MPC++
Munin
Nano-Threads
NESL
NetClasses++
Nexus
Nimrod
Orca
OOF90
P++
P3L
p4-Linda
Pablo
PADE
PADRE
Panda
Papers
AFAPI.
Para++
Paradigm

Parafrase2
Paralation
Parallel-C++
Parallaxis
ParC
ParLib++
ParLin
Parmacs
Parti
pC
pC++
PCN
PCP:
PH
PEACE
PCU
PET
PETSc
PENNY
Phosphorus
P-RIO
Prospero
Proteus
QPC++
PVM
PSI
PSDM
Quake
Quark
Quick Threads
Sage++
SCANDAL
SAM

pC++
SCHEDULE
SciTL
POET
SDDA.
SHMEM
SIMPLE
Sina
SISAL.
distributed smalltalk
SMI.
SONiC
Split-C.
SR
Sthreads
Strand.
SUIF.
Synergy
Telegrphos
SuperPascal
CGMSG.
hreads.h++.
readMarks
RAPPER
uC++
UNITY
UC
V
ViC*
Visifold V-NUS
VPE
Win32 threads
WinPar
WWWinda
XENOOPS
XPC
Zounds
ZPL

With Choice overload in mind … what did we accomplish with all these different options for parallel programming?

Parallel program environments in the 90's

In the early days of parallel computing, we were obsessed with finding the “right” parallel programming environment

ABCPL
ACE
ACT++
Active messages
Adl
Adsmith
ADDAP
AFAPI
ALWAN
AM
AMDC
AppLeS
Amoeba
ARTS
Athapascan-0b
Aurora
Automap
bb_threads
Blaze
BSP
BlockComm
C*.
"C* in C
C**
CarlOS
Cashmere
C4
CC++
Chu
Charlotte
Charm
Charm++
Cid
Cilk
CM-Fortran
Converse
Code
COOL

CORRELATE
CPS
CRL
CSP
Cthreads
CUMULVS
DAGGER
DAPPLE
Data Parallel C
DC++
DCE++
DDD
DICE.
Ease .
ECO
Eiffel
Eilean
Emerald
EPL
Excalibur
Express
Falcon
Filaments
FM
FLASH
The FORCE
Fork
Fortran-M
FX
GA
GAMMA
Glenda

GLU
GUARD
HAsL.
Haskell
HPC++
JAVAR.
HORUS
HPC
HPF
IMPACT
ISIS.
JAVAR
JADE
Karma
KOAN/Fortran-S
LAM
Lilac
Linda
JADA
WWWinda
ISETL-Linda
ParLin
Eilean
P4-Linda
Glenda
POSYBL
Objective-Linda
LiPS
Locust
Lparx
Lucid
Maisie
Manifold

Mentat
Legion
Meta Chaos
Midway
Millipede
CparPar
Mirage
MpC
MOSIX
Modula-P
Modula-2*
Multipol
MPI
Nimrod
NOW
Objective Linda
Occam
Omega
OpenMP
Orca
OOF90
P++
P3L
p4-Linda
Pablo
PADE
PADRE
Panda
Papers
AFAPI.
Para++
Paradigm

Parafrase2
Paralation
Parallel-C++
Parallaxis
ParC
ParLib++
ParLin
Parmacs
Parti
pC
pC++
PCN
PCP:
Phosphorus
POET.
Polaris
POOMA
POOL-T
PRESTO
P-RIO
Prospero
Proteus
QPC++
PVM
PSI
PSDM
Quake
Quark
Quick Threads
Sage++
SCANDAL
SAM

pC++
SCHEDULE
SciTL
POET
SDDA.
SHMEM
SIMPLE
Sina
SISAL.
distributed smalltalk
SMI.
SONiC
Split-C.
Telegrphos
SuperPascal
TCGMSG.
Threads.h++.
TreadMarks
TRAPPER
uC++
UNITY
UC
V
ViC*
Visifold V-NUS
VPE
Win32 threads
WinPar
WWWinda
XENOOPS
XPC
Zounds
ZPL

> Furthermore, engineering is a zero-sum game … time spent chasing the next great programming model is time NOT spent making the models we have actually work

Parallel program environments in the 90’s

The end of the crisis

- In the early 90's, the HPC community was fed up with message passing chaos. Driven largely by application developers, we created MPI (version 1.0 released in 1994).

- In the late 90's, the HPC community working in the Accelerated Strategic Computing Initiative (ASCI) used their influence over which HPC systems were purchased to "force" vendor's hands to support a standard for programming shared memory systems. The result was OpenMP (version 1.0 released in 1997).

Portable parallel programming is important for the people who create HPC applications. It took their direct involvement and dedication to create open standards and end parallel programming chaos.

The major parallel Programming systems in 2024 …
well at least we have our act together in two cases. ☹

- In HPC, 3 programming environments dominate … covering the major classes of hardware.
  - **MPI**: distributed memory systems … though it works nicely on shared memory computers.
  - **OpenMP**: Shared memory systems … more recently, GPGPU too.
  - **CUDA**, **OpenCL**, **Sycl**, **OpenACC**, **OpenMP** … : GPU programming (use CUDA if you don't mind locking yourself to a single vendor … it is a really nice programming model)

Parallel programming with Python is terribly fragmented

dispy
Delegate
forkmap
forkfun
Jobibppmap
POSH
pp
pprocess
processing
PyCSP
PyMP
Ray
remoteD
torcp
VecPy
batchlib
Celery
Charm4py
PyCUDA
Ramba

Dask
Deap
disco
dispy
DistributedPYthon
exec_proxy
execnet
iPython
job_stream jug
mpi4py
NetWorkSpaces
PaPy
papyrus
PyCOMPSs
PyLinda
pyMPI
pypar
multiprocessing
PyOpenCL

pyPastSet
pypvm
pynpvm
Pyro
Ray
Rthread
ScientificPython.BSP
Scientific.DistrubedComputing.MasterSlave
Scientific.MPI
SCOOP
seppo
PySpark
Star-P
superrpy
torcpy
StarCluster
dpctl
arkouda
PyOMP
dpnp

Python programmers are locked into the same dystopic world of HPC in the 90's.

History suggests that this won't get better until the python applications community demands (and dedicates themselves) to a minimal set of open, standard solutions

Building on the list at https://wiki.python.org/moin/ParallelProcessing