OpenMP®

SC'21 Booth Talk Series

SC21

St. Louis, MO | science & beyond.

PyOMP: Parallel Multithreading that is fast AND Pythonic

Tim Mattson, Senior Principal Engineer, Intel Corp.

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

Legal Disclaimer & Optimization Notice

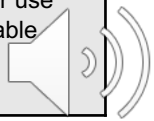
This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.
Notice revision #20110804



Third party names are the property of their owners

Disclaimer

- The views expressed in this talk are those of the speakers and not their employer.

- If we say something “smart” or worthwhile:
  - Credit goes to the many smart people we work with.

- If we say something stupid…
  - It’s our own fault

We work in Intel’s research labs. We don’t build products. Instead, we get to poke into dark corners and think silly thoughts… just to make sure we don’t miss any great ideas.

Hence, our views are by design far “off the roadmap”.

Acknowledgments

- Michel Pelletier (Graphegon):
  - His GraphBLAS binding to python was the inspiration for the design of PyOMP
- Todd Anderson (Intel):
  - A Numba wizard who did the HARD implementation work that made PyOMP possible
- Giorgis Georgakoudis (LLNL) and Johannes Doerfert(ANL):
  - They are working with us to port PyOMP to an OpenMP enabled open-source version of LLVM

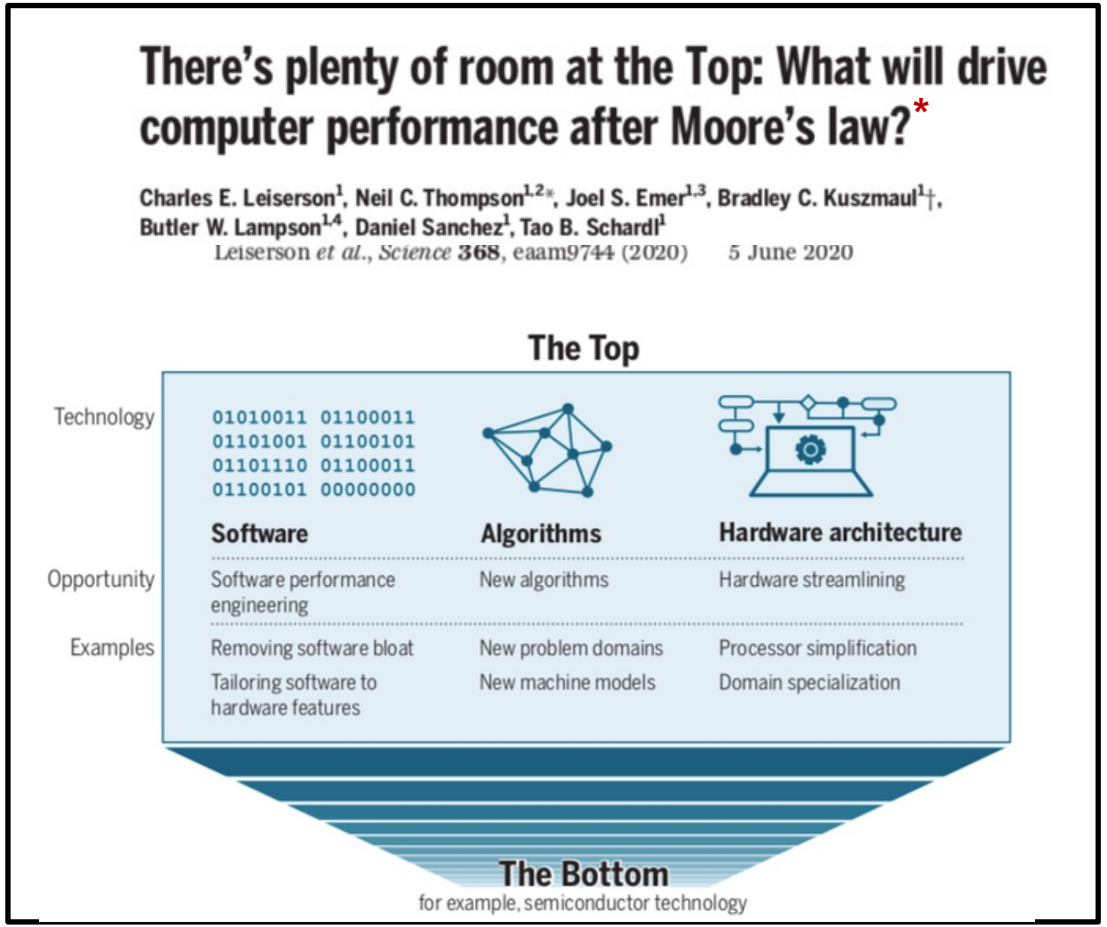
Multithreaded parallel Python through OpenMP support in Numba, Todd Anderson and Tim Mattson, SciPy'2021 http://conference.scipy.org/proceedings/scipy2021/tim_mattson.html

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

LLNL: Lawrence Livermore National Laboratory ANL: Argonne National Laboratory

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

The Top

	Software	Algorithms	Hardware architecture
Technology	01010011 01100011 01101001 01100101 01101110 01100011 01100101 00000000		
Opportunity	Software performance engineering	New algorithms	Hardware streamlining
Examples	Removing software bloat Tailoring software to hardware features	New problem domains New machine models	Processor simplification Domain specialization

The Bottom

for example, semiconductor technology

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 architectu[illegible]tters, but dramatically LE[illegible]S 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	14.95
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	15.95
7	plus AVX intrinsics	0.41	337.812	62,806	2.7	41.45

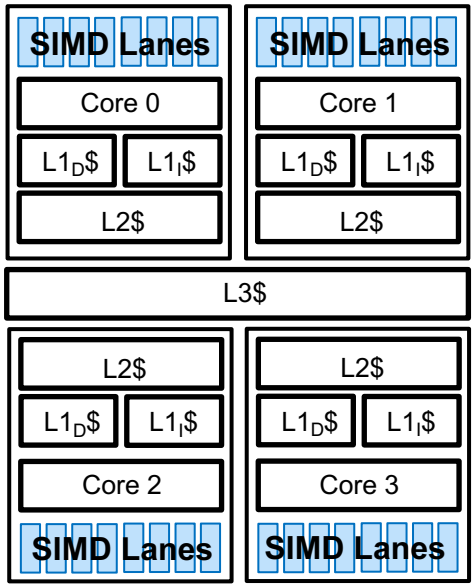
Amazon AWS c4.8xlarge spot instance, Intel® Xeon® E5-2666 v3 CPU, 2.9 Ghz, 18 core, 60 GB RAM

Our goal … to help programmers “keep it in Python”

- Modern technology should be able to map Python onto low-level code (such as C or LLVM) and avoid the “Python performance tax”.
- We’ve* worked on …
  - Numba (2012): JIT Python code into LLVM
  - Parallel accelerator (2017): Find and exploit parallel patterns in Python code.
  - Intel High-Performance Analytics Toolkit and Scalable Dataframe Compiler (2019): Parallel performance from data frames.
  - Intel numba-dppy (2020): Numba ParallelAccelerator regions that run on GPUs via SYCL.

*OK, not “we” … it was mostly Todd

How do you get high performance for a modern CPU?



Three simple principles:

- Lots of threads … at least one per hardware thread (often two hardware threads per core)
- Exploit SIMD lanes form each thread
- Maximize cache utilization

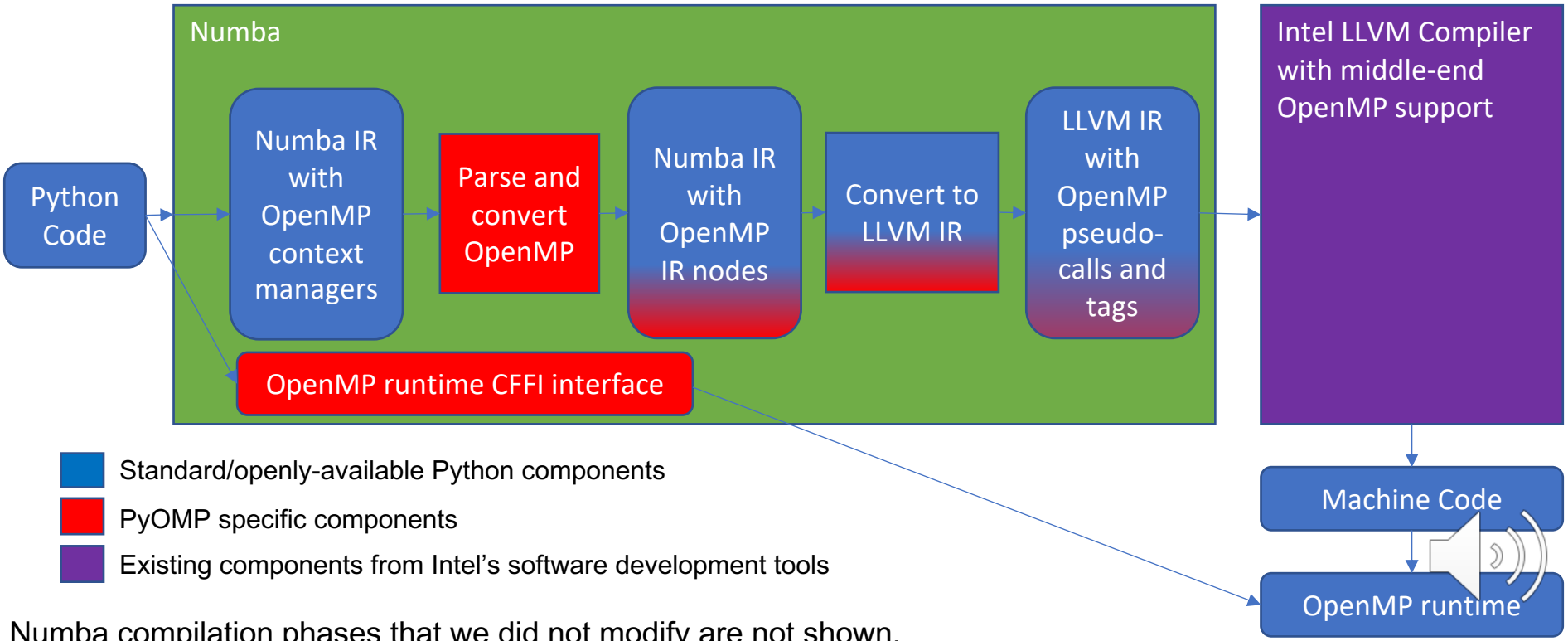
Why not embed parallelism inside Numpy? This works, but it suffers from two problems:

1. Overhead of creating/destroying threads at each operation … increases parallel overhead and limits scalability (due to Amdahl's law)
2. Lost opportunity for parallelism from running multiple Numpy operations in parallel

… We want threads, but the **GIL** (**G**lobal **I**nterpreter **L**ock) prevents multiple threads from making forward progress in parallel. The GIL is great for supporting thread safety and making it hard to write code that contains data races, but it prevents parallel multithreading in Python

What is the most common way in HPC to create multithreaded code? Something called OpenMP

PyOMP Implementation in Numba: Overview

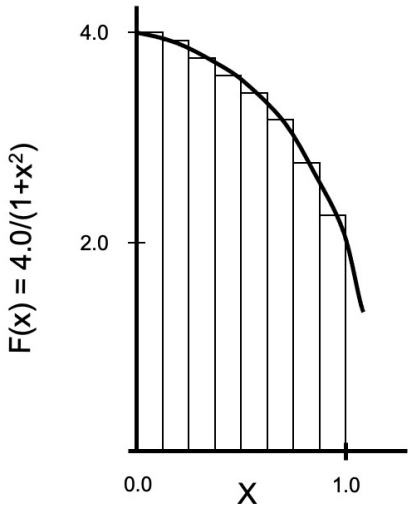


Numba compilation phases that we did not modify are not shown.

Understanding OpenMP

We will explain the key elements of OpenMP as we explore 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 the *with* context manager.

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

- **parallel**: create a team of threads
- **for**: map loop iterations onto threads
- **private(x)**: each threads gets its own x
- **reduction(+:x)**: combine x from each thread using +

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	

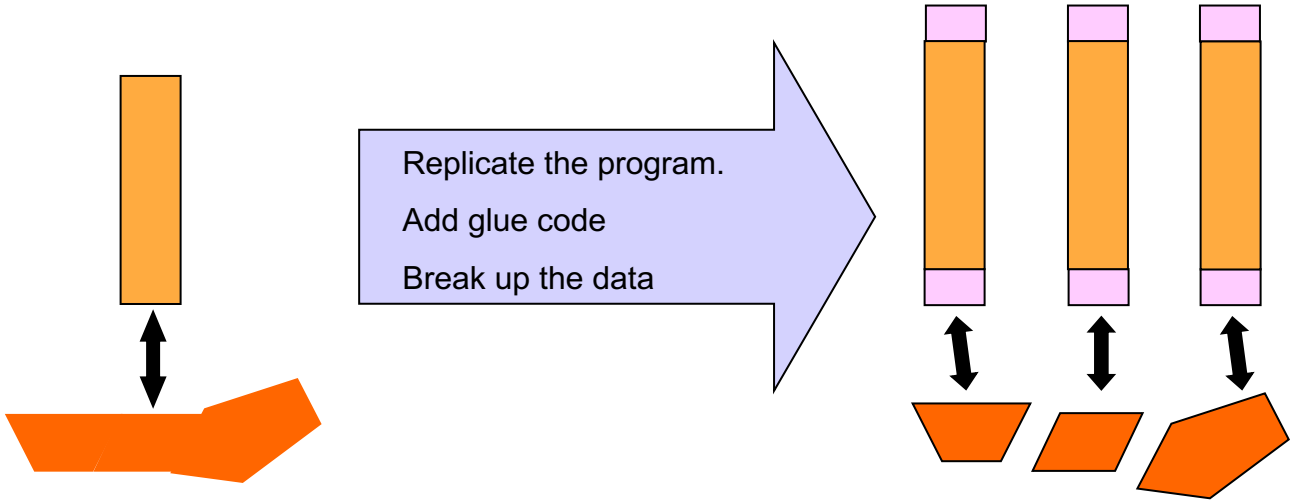
Intel® Xeon® E5-2699 v3 CPU with 18 cores running at 2.30 GHz.

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)

SPMD (Single Program Multiple Data) design pattern

- Run the same program on P processing elements where P can be arbitrarily large.
- Use the rank … an ID ranging from 0 to (P-1) … to select between a set of tasks and to manage any shared data structures.



This pattern is very general and has been used to support most (if not all) the algorithm strategy patterns.

MPI programs almost always use this pattern … it is probably the most commonly used pattern in the history of parallel programming.

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.

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

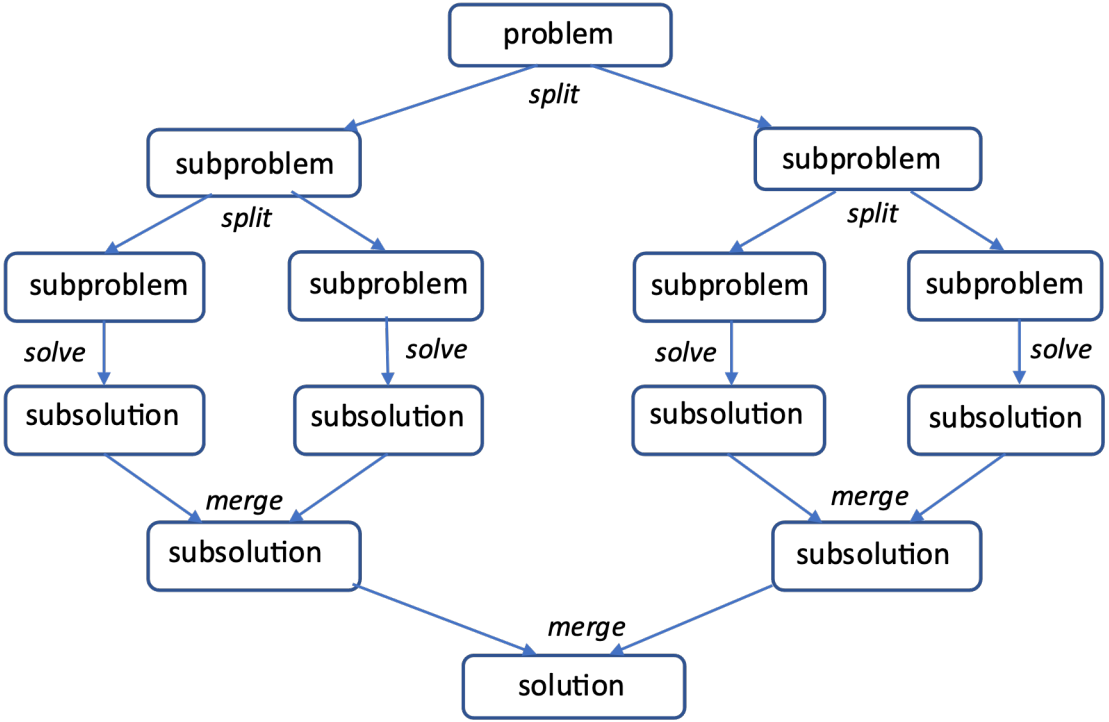
Intel® Xeon® E5-2699 v3 CPU with 18 cores running at 2.30 GHz.

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 design pattern

- Split the problem into smaller sub-problems; continue until the sub-problems can be solve directly



3 Options for parallelism:

- Do work as you split into sub-problems
- Do work at the leaves
- Do work as you recombine

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):
        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

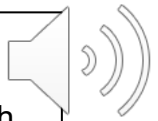
Split

Merge

```
@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

- **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

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

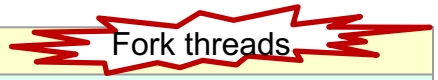
Ran each case 5 times and kept the minimum time. **JIT time is not included** for PyOMP (it was about 1.5 seconds)

OpenMP subset supported in PyOMP

`with openmp("parallel"):`	Create a team of threads. Execute a parallel region
`with openmp("for"):`	Use inside a parallel region. Split up a loop across the team.
`with openmp("parallel for"):`	A combined construct. Same a **`parallel`** followed by a **`for`**.
`with openmp ("single"):`	One thread does the work. Others wait for it to finish
`with openmp("task"):`	Create an explicit task for work within the construct.
`with openmp("taskwait"):`	Wait for all tasks in the current task to complete.
`with openmp("barrier"):`	All threads arrive at a barrier before any proceed.
`with openmp("critical"):`	Mutual exclusion. One thread at a time executes code
`schedule(static [,chunk])`	Map blocks of loop iterations across the team. Use with **`for`**.
`reduction(op:list)`	Combine values with op across the team. Used with **`for`**
`private(list)`	Make a local copy of variables for each thread. Use with **`parallel`**, **`for`** or **`task`**.
`firstprivate(list)`	**`private`**, but initialize with original value. Use with **`parallel`**, **`for`** or **`task`**
`shared(list)`	Variables shared between threads. Use with **`parallel`**, **`for`** or **`task`**.
`default(none)`	Force definition of variables as **`private`** or **`shared`**.
`omp_get_num_threads()`	Return the number of threads in a team
`omp_get_thread_num()`	Return an ID from 0 to the number of threads minus one
`omp_set_num_threads(int)`	Set the number of threads to request for parallel regions
`omp_get_wtime()`	Return a snapshot of the wall clock time.
`OMP_NUM_THREADS=N`	Environment variable to set the default number of threads

OpenMP subset supported in PyOMP

`with openmp("parallel"):`	Create a team of threads. Execute a parallel region
`with openmp("for"):`	Use inside a parallel region. Split up a loop across the team.
`with openmp("parallel for"):`	A combined construct. Same a **`parallel`** followed by a **`for`**.
`with openmp ("single"):`	One thread does the work. Others wait for it to finish
`with openmp("task"):`	Create an explicit task for work within the construct.
`with openmp("taskwait"):`	Wait for all tasks in the current task to complete.
`with openmp("barrier"):`	All threads arrive at a barrier before any proceed.
`with openmp("critical"):`	Mutual exclusion. One thread at a time executes code
`schedule(static [,chunk])`	Map blocks of loop iterations across the team. Use with **`for`**.
`reduction(op:list)`	Combine values with op across the team. Used with **`for`**
`private(list)`	Make a local copy of variables for each thread. Use with **`parallel`**, **`for`** or **`task`**.
`firstprivate(list)`	**`private`**, but initialize with original value. Use with **`parallel`**, **`for`** or **`task`**
`shared(list)`	Variables shared between threads. Use with **`parallel`**, **`for`** or **`task`**
`default(none)`	Force definition of variables as **`private`** or **`shared`**.
`omp_get_num_threads()`	Return the number of threads in a team
`omp_get_thread_num()`	Return an ID from 0 to the number of threads minus one
`omp_set_num_threads(int)`	Set the number of threads to request for parallel regions
`omp_get_wtime()`	Return a snapshot of the wall clock time.
`OMP_NUM_THREADS=N`	Environment variable to set the default number of threads



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	14.95
7	plus AVX intrinsics	0.41	337.812	62,806	2.7	40.45

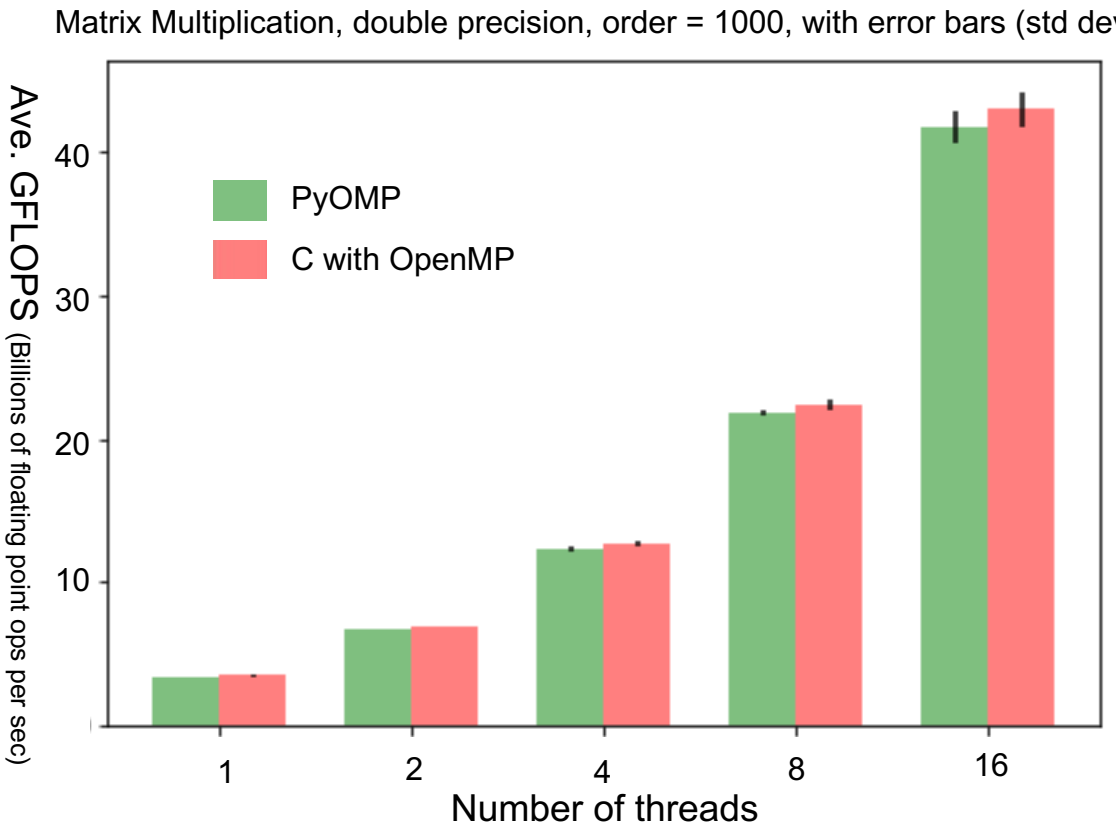
Amazon AWS c4.8xlarge spot instance, Intel® Xeon® E5-2666 v3 CPU, 2.9 Ghz, 18 core, 60 GB RAM

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.

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)

Summary

- We've created a research prototype OpenMP interface in Python called PyOMP.
  - It is based on Numba and an OpenMP enabled LLVM

- Next steps:
  - We need to carry out detailed benchmarking (DASK, Ray, MPI4py)
  - We need to map PyOMP onto an open source, publicly available LLVM
    - Work ongoing in partnership between Intel, ANL, and LLNL.
    - Track our progress at:

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

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