PYOMP: PARALLEL PROGRAMMING IN PYTHON WITH OPENMP

Tim Mattson (University of Bristol)

Giorgis Georgakoudis (Lawrence Livermore Nat. Lab)

Todd A. Anderson (Intel Labs)

SC24 Atlanta, GA | hpc creates.

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

License

CC BY 4.0

- **You are free to:**
  - **Share** — copy and redistribute the material in any medium or format for any purpose, even commercially.
  - **Adapt** — remix, transform, and build upon the material for any purpose, even commercially.
  - The licensor cannot revoke these freedoms as long as you follow the license terms.
- **Under the following terms:**
  - **Attribution** — You must give appropriate credit , provide a link to the license, and indicate if changes were made . You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
  - **No additional restrictions** — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Disclaimer

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

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

Outline

- Introducing parallel computing and PyOMP
- The PyOMP system
- PyOMP and multithreading (parallelism for the CPU)

Break

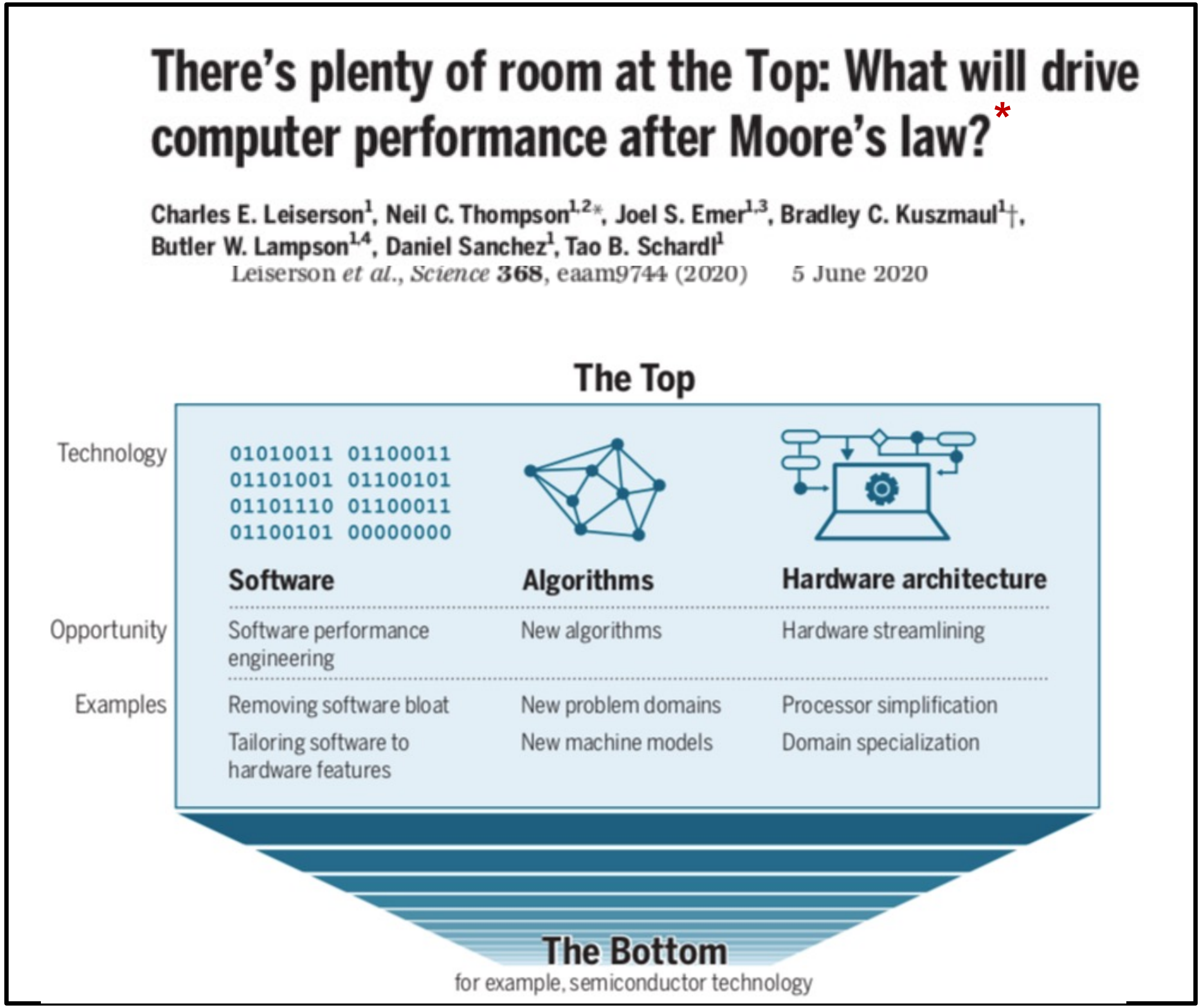
- GPU programming with PyOMP
- Other approaches to parallelism in Python.
- Wrap-up and Q&A

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

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

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 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

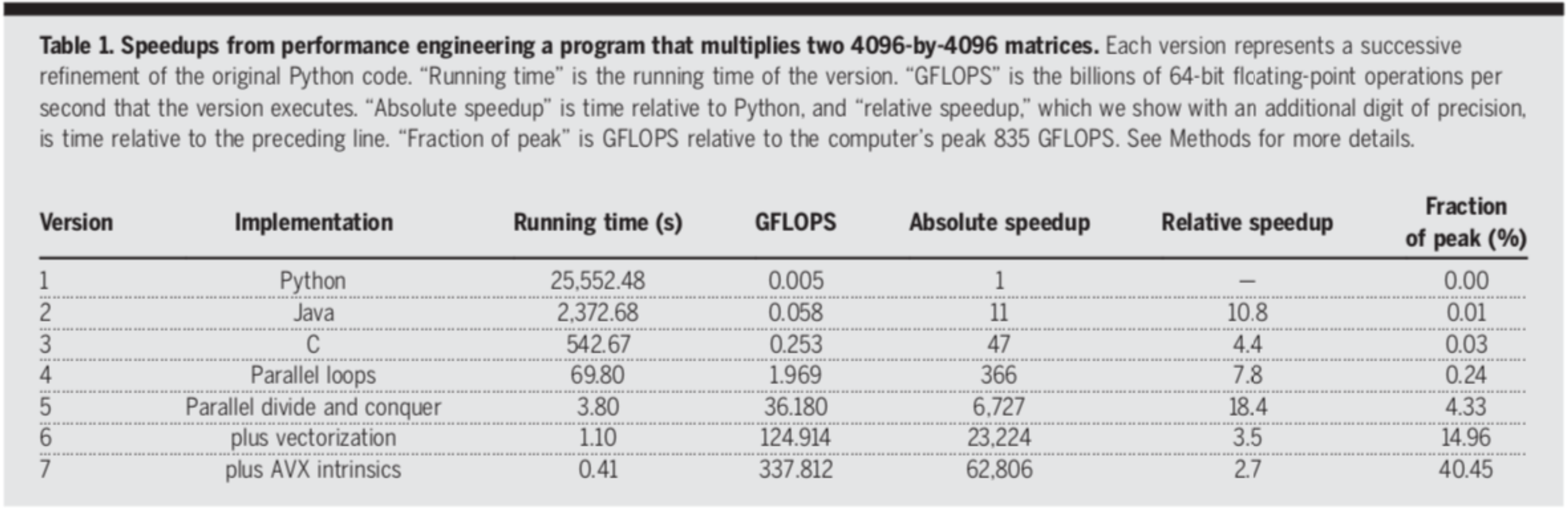


Table 1. Speedups from performance engineering a program that multiplies two 4096-by-4096 matrices. Each version represents a successive refinement of the original Python code. “Running time” is the running time of the version. “GFLOPS” is the billions of 64-bit floating-point operations per second that 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.

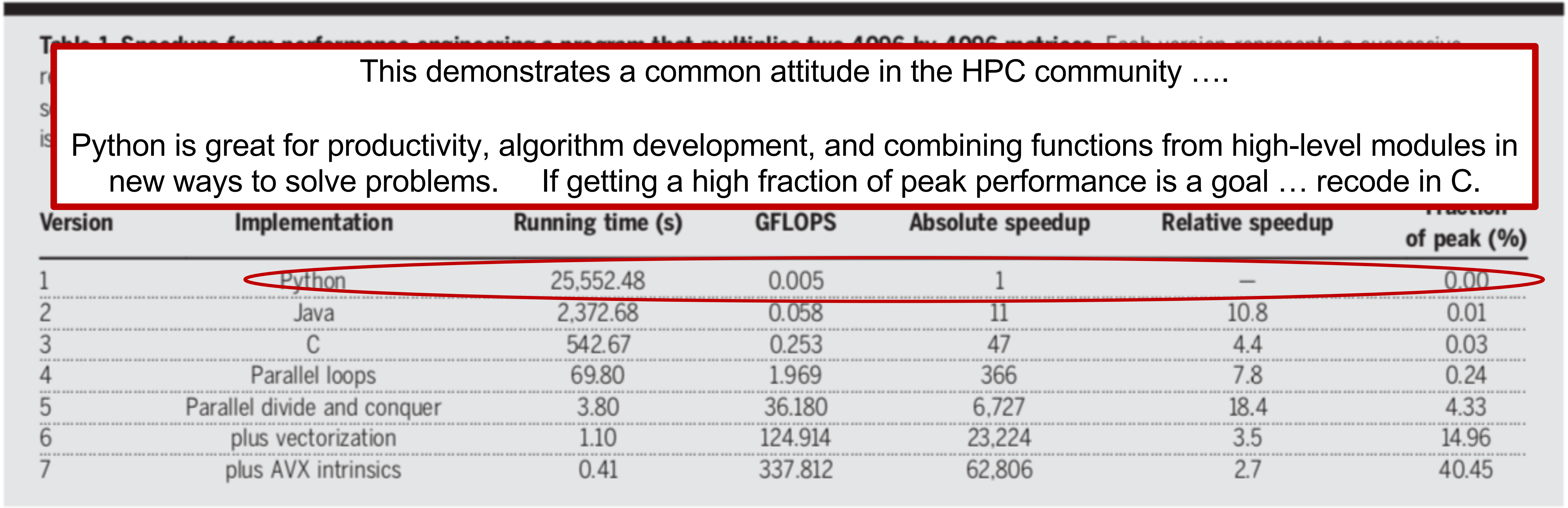


Table 1. Speedups from performance engineering a program that multiplies two 4096-by-4096 matrices. Each version represents a successive

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

Our goal … to help people “keep their code in Python”

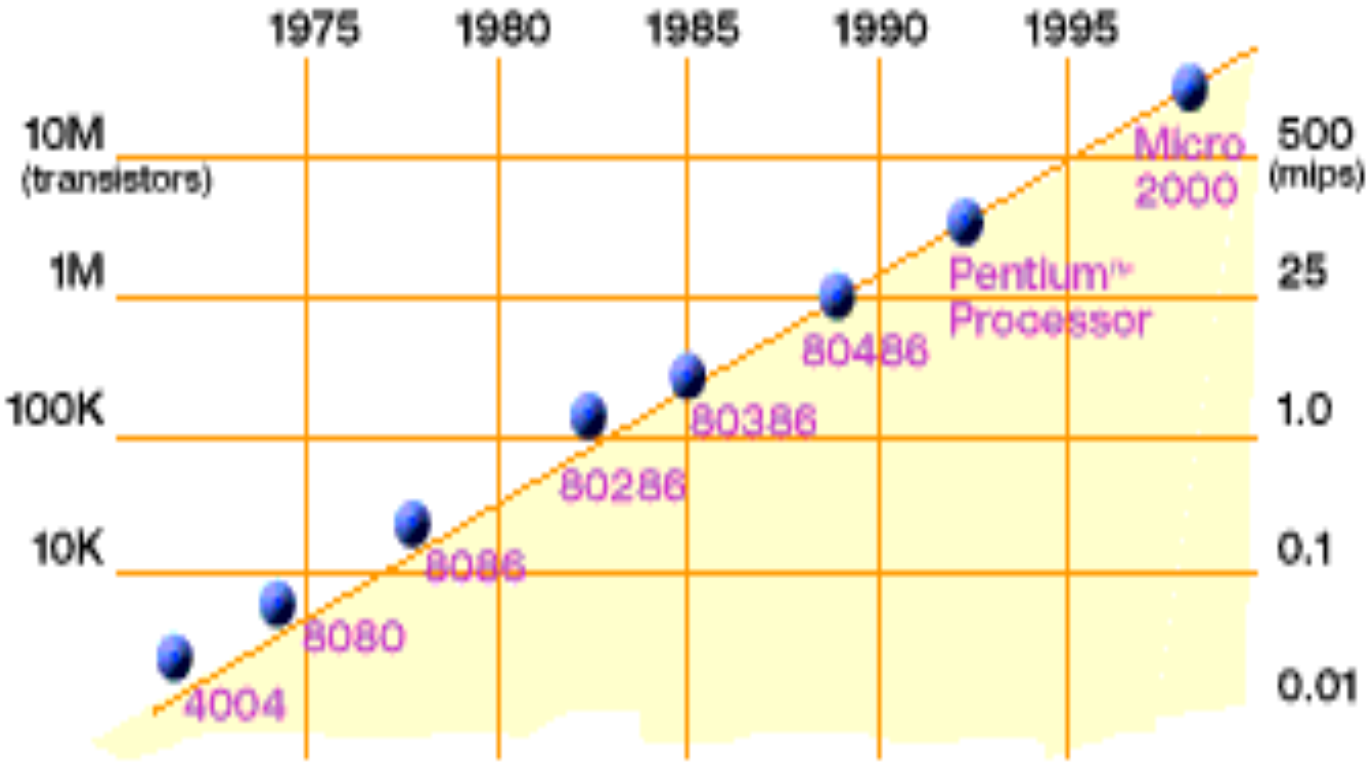
- Modern technology should be able to map Python onto low-level code (such as C or LLVM) and avoid the “Python performance tax”.

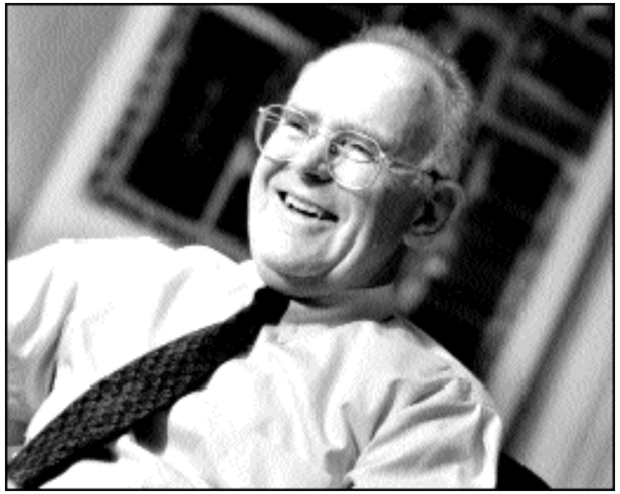
- We’ve worked on …
  - Numba (2012): JIT Python code into LLVM
  - Parallel accelerator (2017): Find and exploit parallel patterns in Python code.
  - Intel High-Performance Analytics Toolkit and Scalable Dataframe Compiler (2019): Parallel performance from data frames.
  - Intel numba-dppy (2020): Numba ParallelAccelerator regions that run on GPUs via SYCL.

If it’s performance you want, then you must go parallel.

It’s in the physics!

Moore's Law

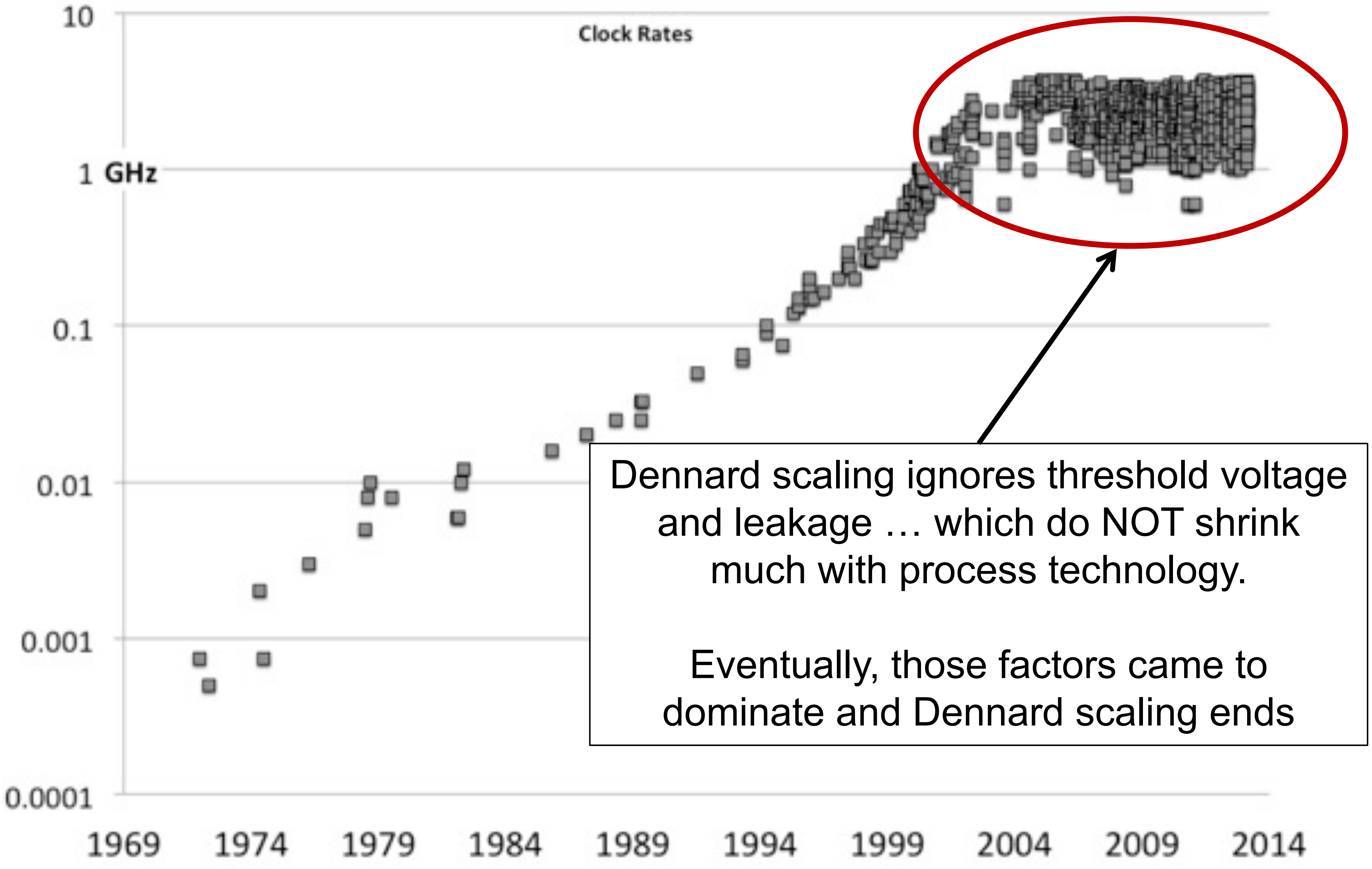




- In 1965, Intel co-founder Gordon Moore predicted (from just 3 data points!) that semiconductor density would double every 18 months.
  - ***He was right!*** Over the last 50 years, transistor densities have increased as he predicted.

Slide source: UCB CS 194 Fall'2010

CPU Frequency (GHz) over time (years)

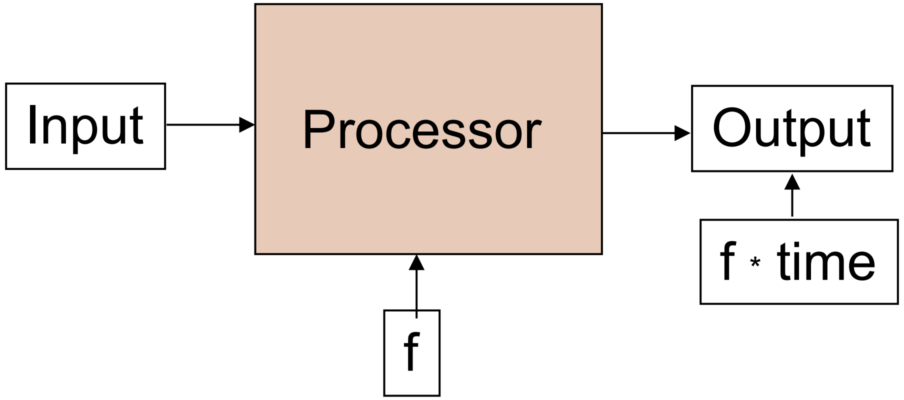


Dennard scaling ignores threshold voltage and leakage … which do NOT shrink much with process technology.

Eventually, those factors came to dominate and Dennard scaling ends

Source: James Reinders (from the book “structured parallel programming”)

Consider power in a chip …



Input

Processor

Output

f * time

f

Capacitance = C
Voltage = V
Frequency = f
Power = CV^2f

C = capacitance … it measures the ability of a circuit to store energy:

$$C = q/V \rightarrow \quad q = CV$$

Work is pushing something (charge or q) across a “distance” … in electrostatic terms pushing q from 0 to V:

$$V * q = W.$$

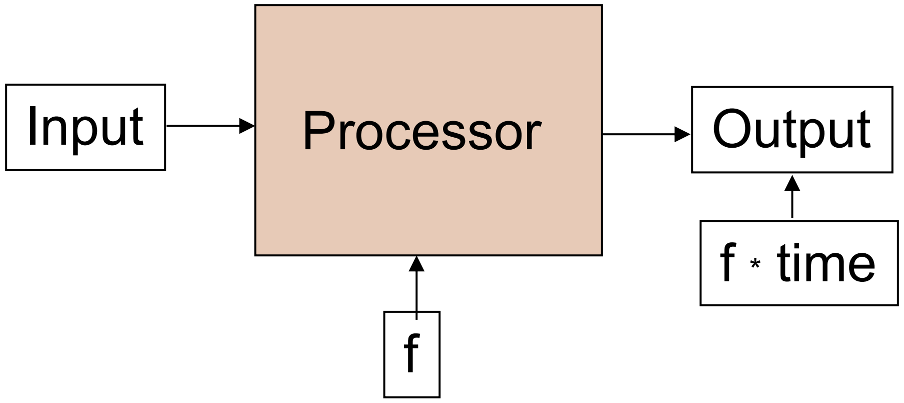
But for a circuit $q = CV$ so

$$W = CV^2$$

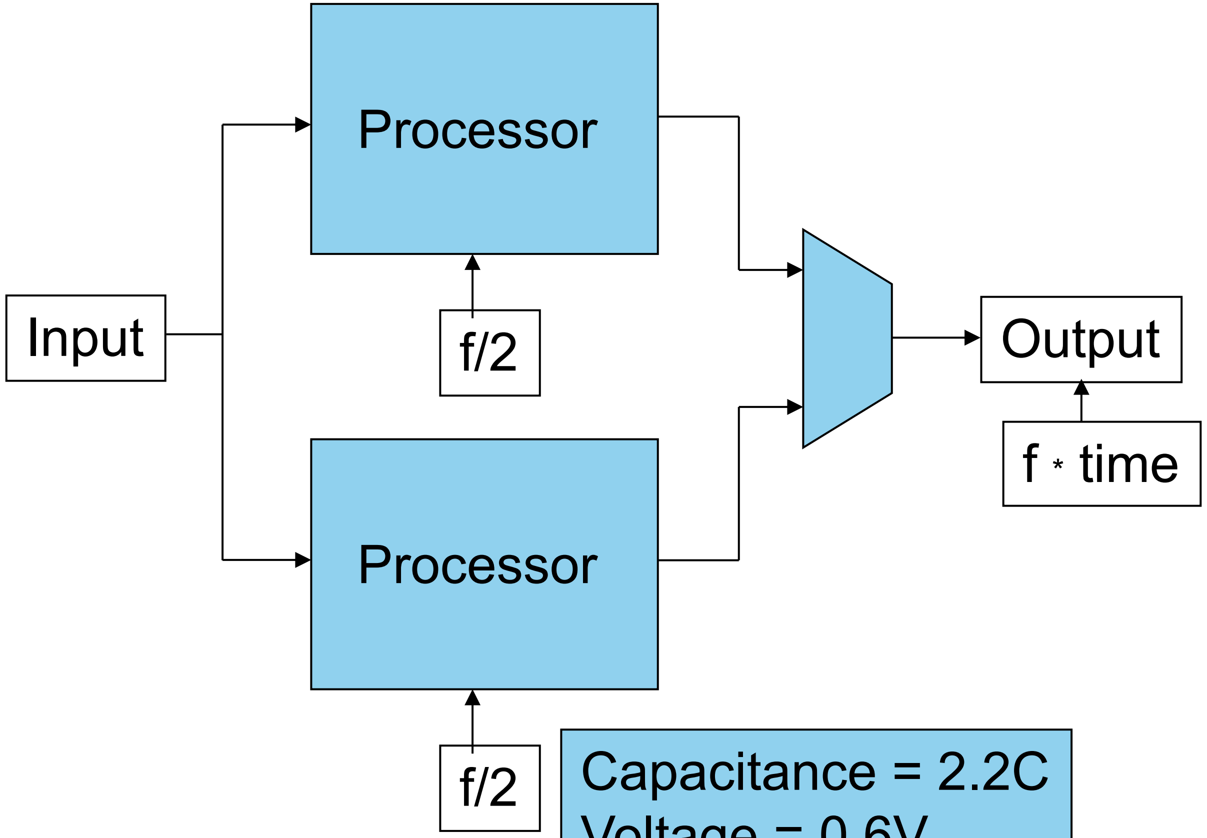
power is work over time … or how many times per second we oscillate the circuit

$$\text{Power} = W* F \quad \rightarrow \quad \text{Power} = CV^2f$$

... Reduce power by adding cores



Capacitance = C
Voltage = V
Frequency = f
Power = CV^2f



Capacitance = 2.2C
Voltage = 0.6V
Frequency = 0.5f
Power = $0.396CV^2f$

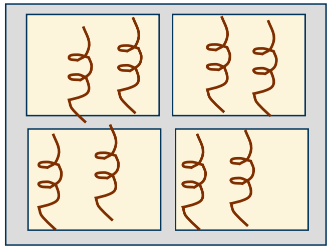
Chandrakasan, A.P.; Potkonjak, M.; Mehra, R.; Rabaey, J.; Brodersen, R.W., "Optimizing power using transformations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,*, vol.14, no.1, pp.12-31, Jan 1995

Source: Vishwani Agrawal

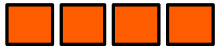
… So now lets talk about parallel hardware

For hardware … parallelism is the path to performance

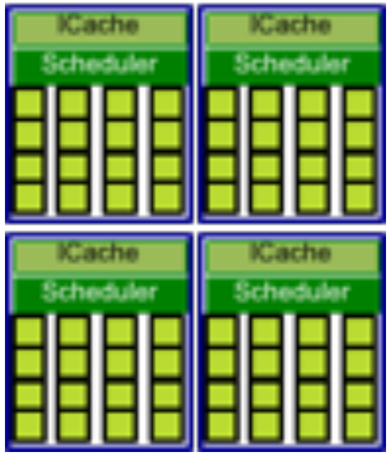
All hardware vendors are in the game … parallelism is ubiquitous so if you care about getting the most from your hardware, you will need to create parallel software.



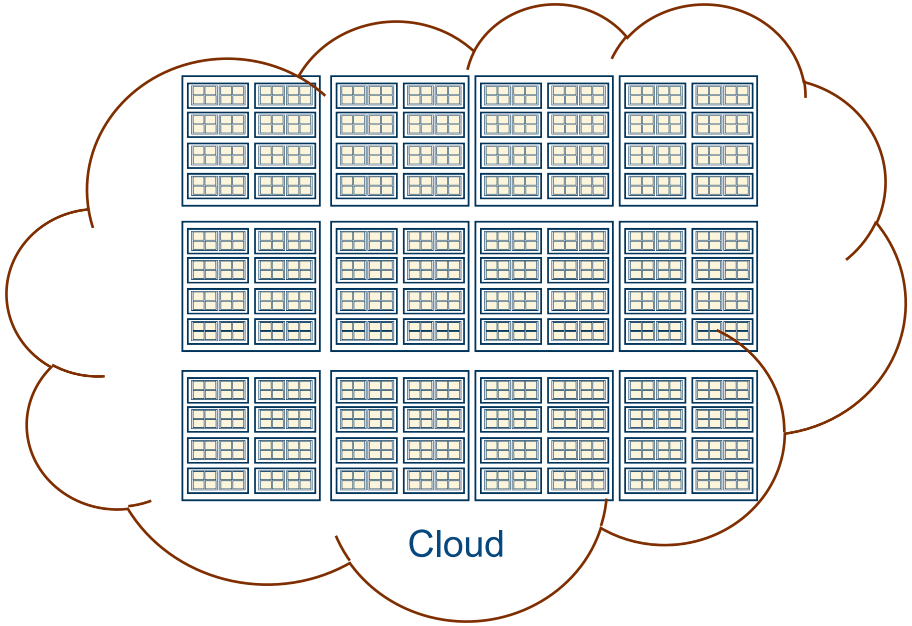
CPU



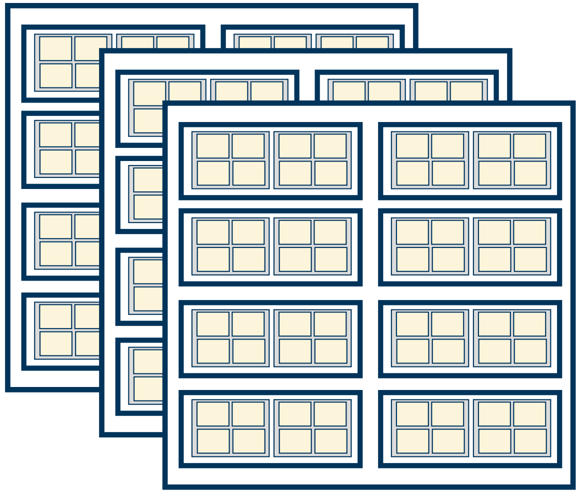
SIMD/Vector



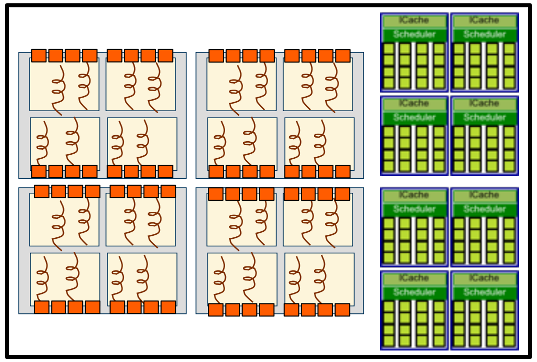
GPU



Cloud



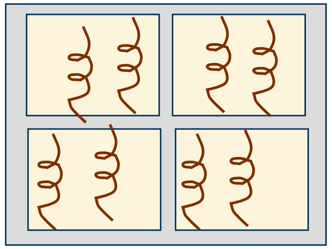
Cluster



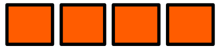
Heterogeneous node

For hardware … parallelism is the path to performance

All hardware vendors are in the game … parallelism is ubiquitous so if you care about getting the most from your hardware, you will need to create parallel software.



CPU



SIMD/Vector

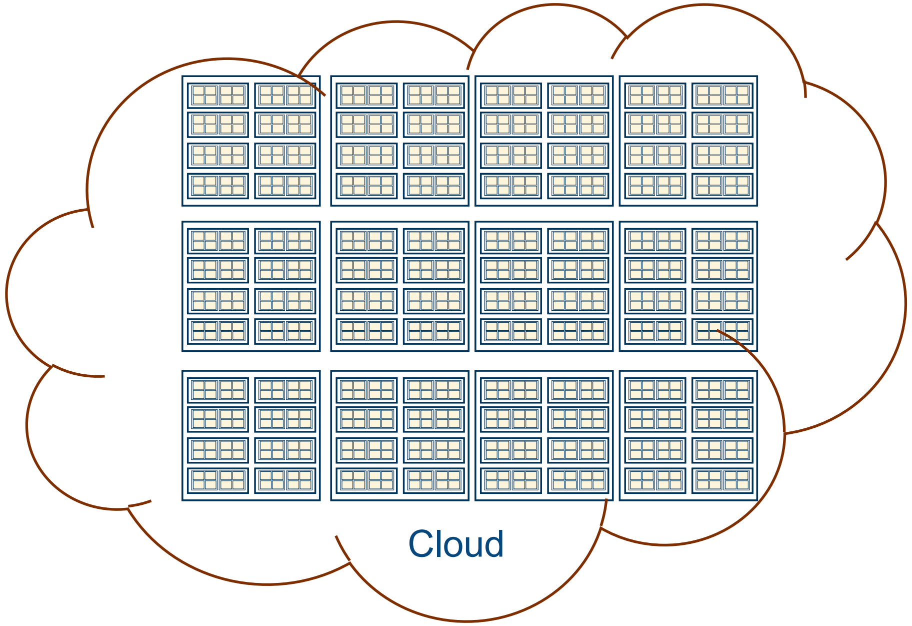
We will let the compiler take care of vectorization for us

ICache
Scheduler

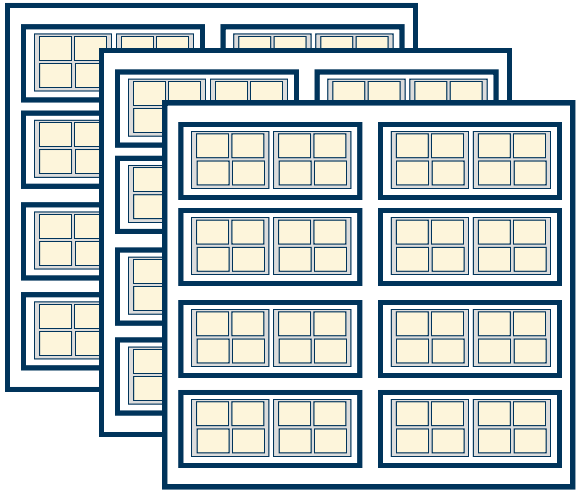
GPU

We will cover the CPU and the GPU

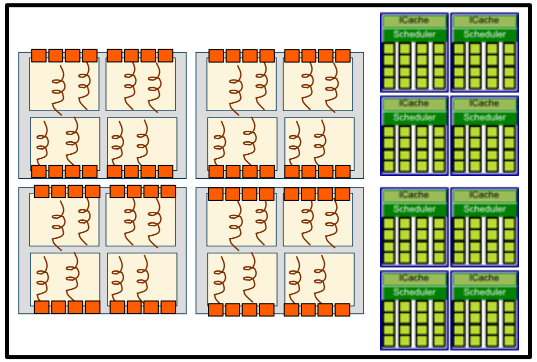
PyOMP works here as well … though we won't discuss this case in this tutorial



Cloud



Cluster

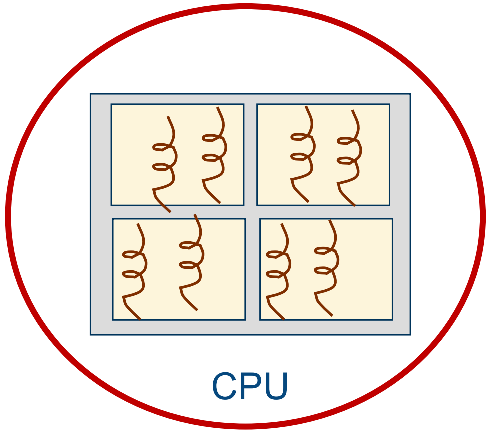


Heterogeneous node

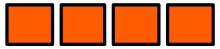
For hardware … parallelism is the path to performance

All hardware vendors are in the game … parallelism is ubiquitous so if you care about getting the most from your hardware, you will need to create parallel software.

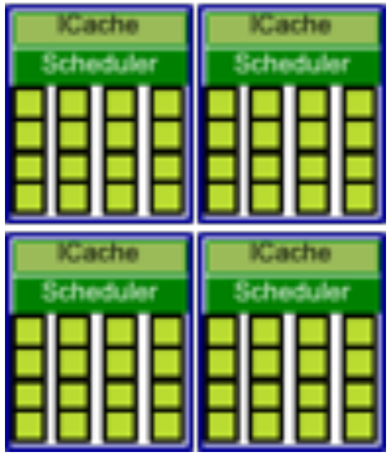
We will start with the CPU



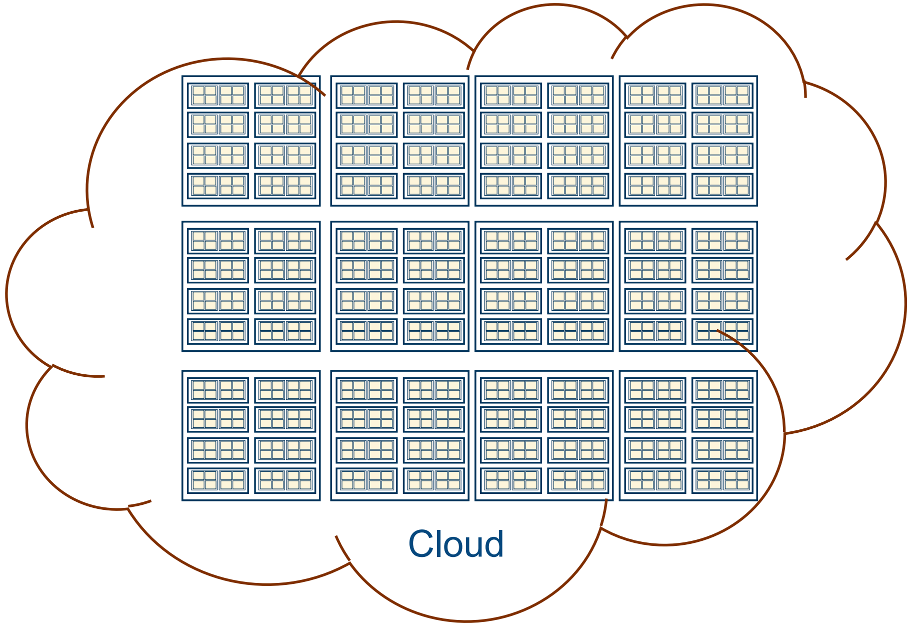
CPU



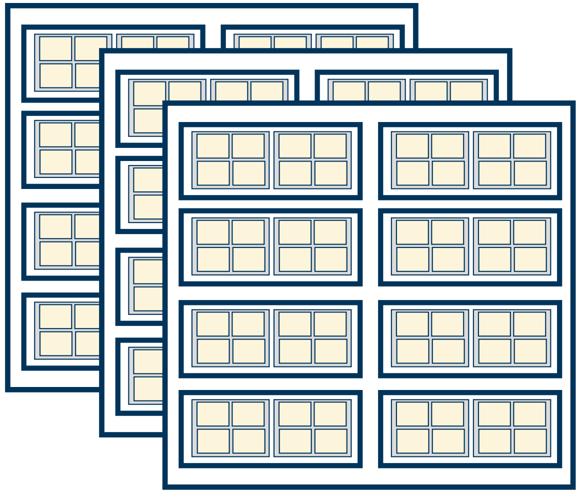
SIMD/Vector



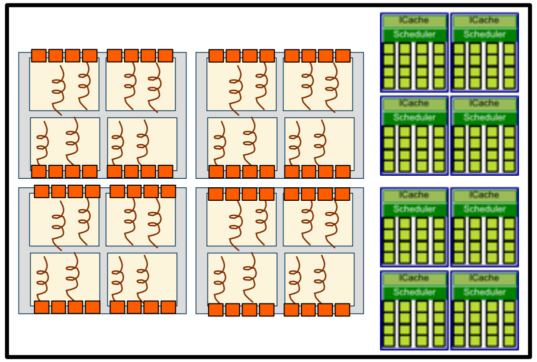
GPU



Cloud



Cluster



Heterogeneous node

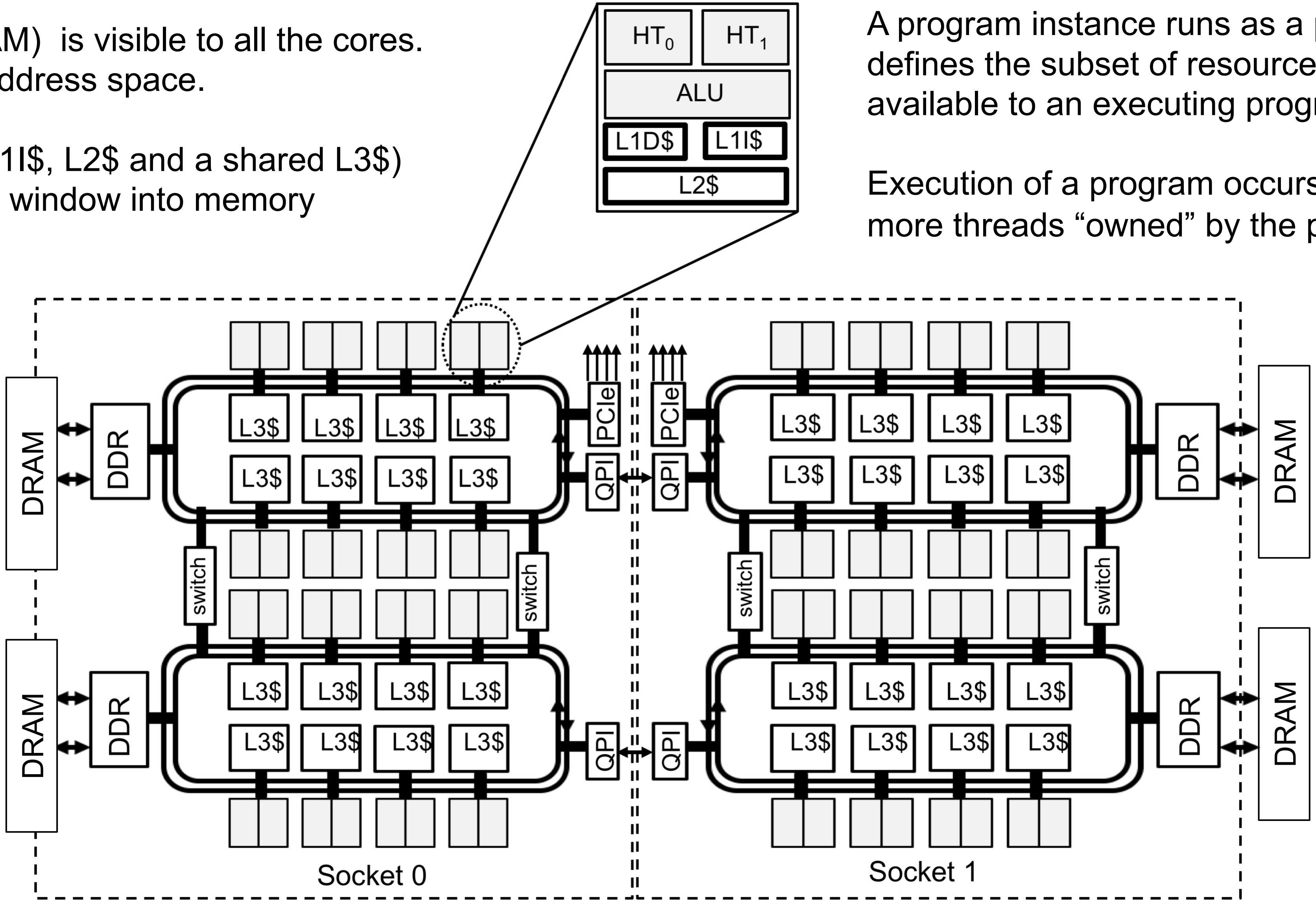
A typical multi-core CPU

All the memory (DRAM) is visible to all the cores. It presents a single address space.

The caches (L1D$, L1I$, L2$ and a shared L3$) provide a high-speed window into memory

A program instance runs as a process. A process defines the subset of resources (such as memory) available to an executing program.

Execution of a program occurs through one or more threads “owned” by the process.



ALU: arithmetic logic unit, HT: hardware thread QPI: quick path interconnect DDR: Dram memory controller DRAM: dynamic random access memory
L!D$: L1 data cache, L1I$: L1 instruction cache L2: a unified (data and instructions) cache

The ubiquitous standard for multithreaded programming on CPUs is OpenMP

OpenMP* Overview

```
C$OMP FLUSH
#pragma omp critical
C$OMP THREADPRIVATE(/ABC/)
CALL OMP_SET_NUM_THREADS(10)
```

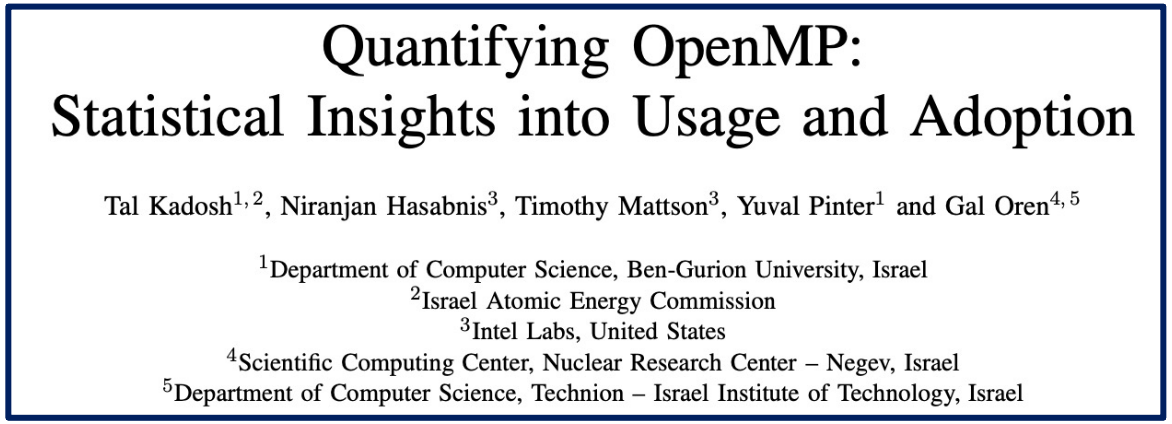
OpenMP: An API for Writing Multithreaded Applications

- A set of compiler directives and library routines for parallel application programmers
- Greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++
- Standardizes established SMP practice + vectorization and heterogeneous device programming

```
C$OMP PARALLEL COPYIN(/blk/)
C$OMP DO lastprivate(XX)
Nthrds = OMP_GET_NUM_PROCS()
omp_set_lock(lck)
```

* The name "OpenMP" is the property of the OpenMP Architecture Review Board.

OpenMP is the most popular Programming model for HPC



Quantifying OpenMP:
Statistical Insights into Usage and Adoption

Tal Kadosh[1, 2], Niranjan Hasabnis[3], Timothy Mattson[3], Yuval Pinter[1] and Gal Oren[4, 5]

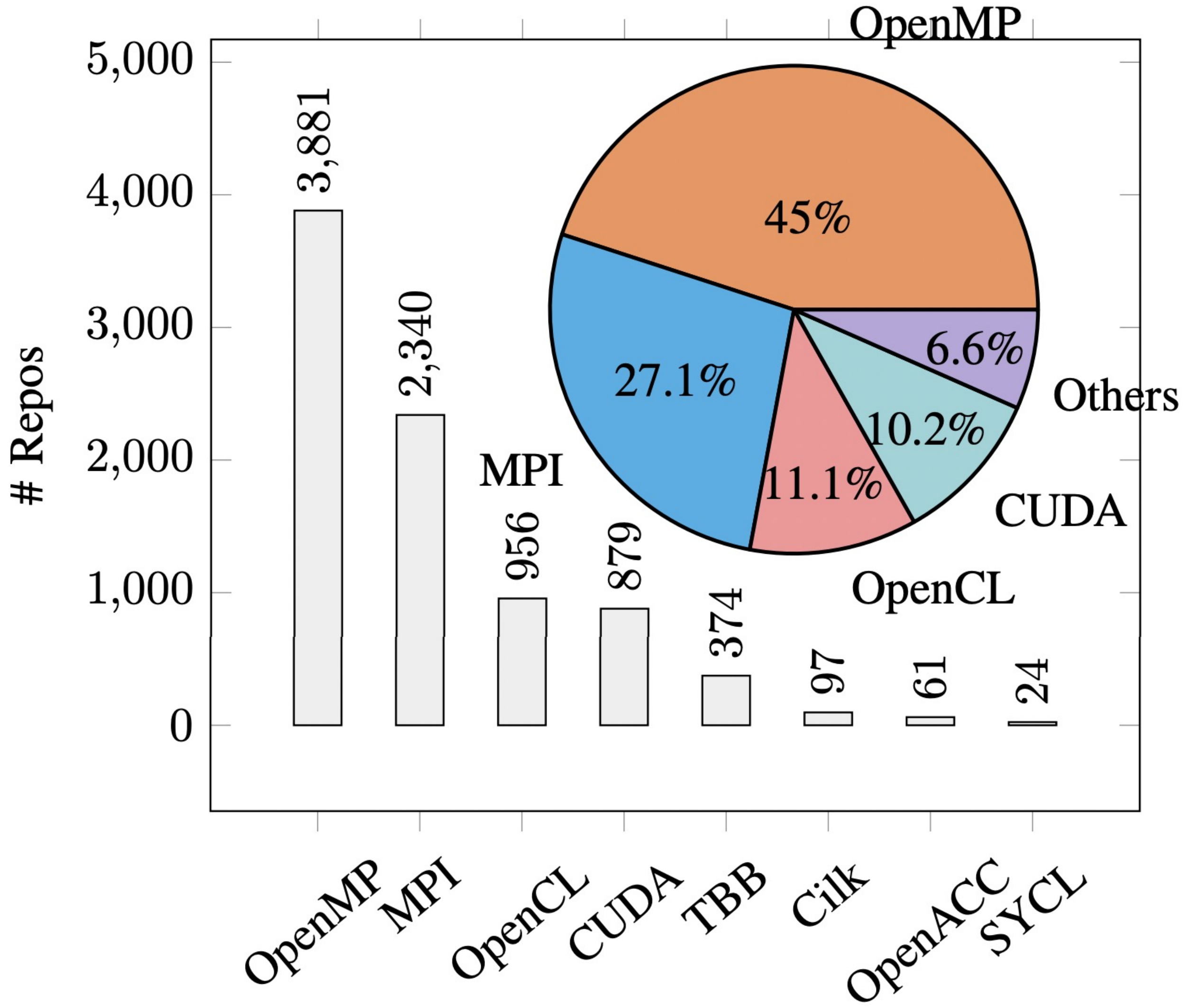
[1]Department of Computer Science, Ben-Gurion University, Israel
[2]Israel Atomic Energy Commission
[3]Intel Labs, United States
[4]Scientific Computing Center, Nuclear Research Center – Negev, Israel
[5]Department of Computer Science, Technion – Israel Institute of Technology, Israel

Download the paper here: https://arxiv.org/abs/2308.08002

We constructed a dataset from all c, c++ and Fortran programs in github for training large language models for parallel code generation.

We analyzed programming model usage across the dataset and found that OpenMP was the most popular of all parallel programming models in github.

Note: we did not collect .cu or .cuf files so we under-counted CUDA usage.



Aggregate numbers over all repositories from 2013 to 2023

PyOMP: OpenMP projected into Python

- A parallel multithreaded “hello world” program with PyOMP

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

@njit
def hello():
    with openmp("parallel"):
         print("hello")
         print("world")

hello()
print("DONE")
```

PyOMP: OpenMP projected into Python

- A parallel multithreaded “hello world” program with PyOMP

Numba Just In Time (JIT) compiler compiles the Python code into LLVM.

Compiled code cached for later use.

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

@njit
def hello():
    with openmp("parallel"):
        print("hello")
        print("world")

hello()
print("DONE")
```

“parallel” creates a team of threads

The code inside the with context manager is packaged into a function and executed by each thread

OpenMP managed through the *with* context manager.

- Numba Just In Time (JIT) compiler compiles the Python code into LLVM thereby bypassing the GIL.   Hence, the threads execute in parallel.

- The string in the with openmp context manager is identical to the constructs in OpenMP. If you know OpenMP for C/C++/Fortran, then you know it for Python

PyOMP: OpenMP projected into Python

- A parallel multithreaded “hello world” program with PyOMP

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

@njit
def hello():
    with openmp("parallel"):
        print("hello")
        print("world")

hello()
print("DONE")
```

When I run this program, here is the output.

```
hello
world
hello
hello
hello
world
hello
world
hello
world
hello
world
world
world
hello
world
DONE
```

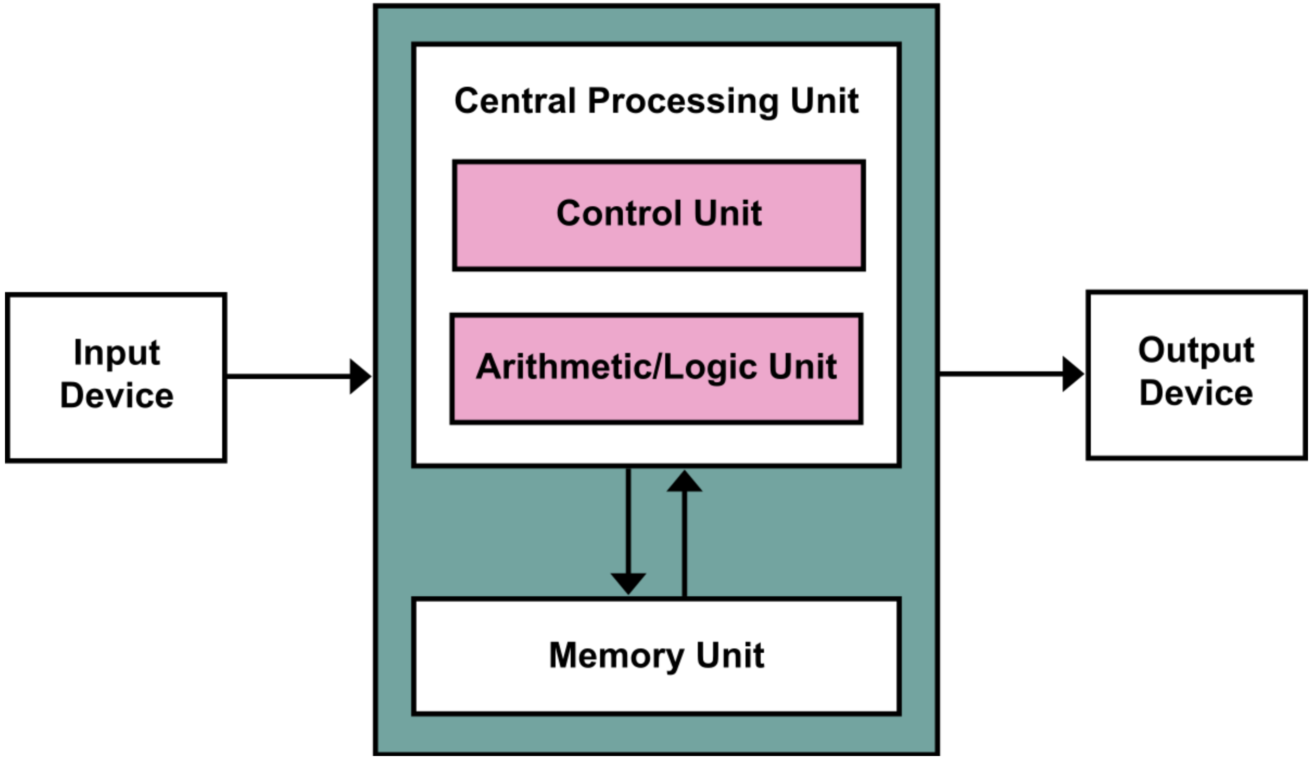
The interleaved print output is different each time I run the program

Why is the output from our hello world program so weird?

To answer that question, we must digress briefly and settle on a few key definitions

Let’s agree on a few definitions:

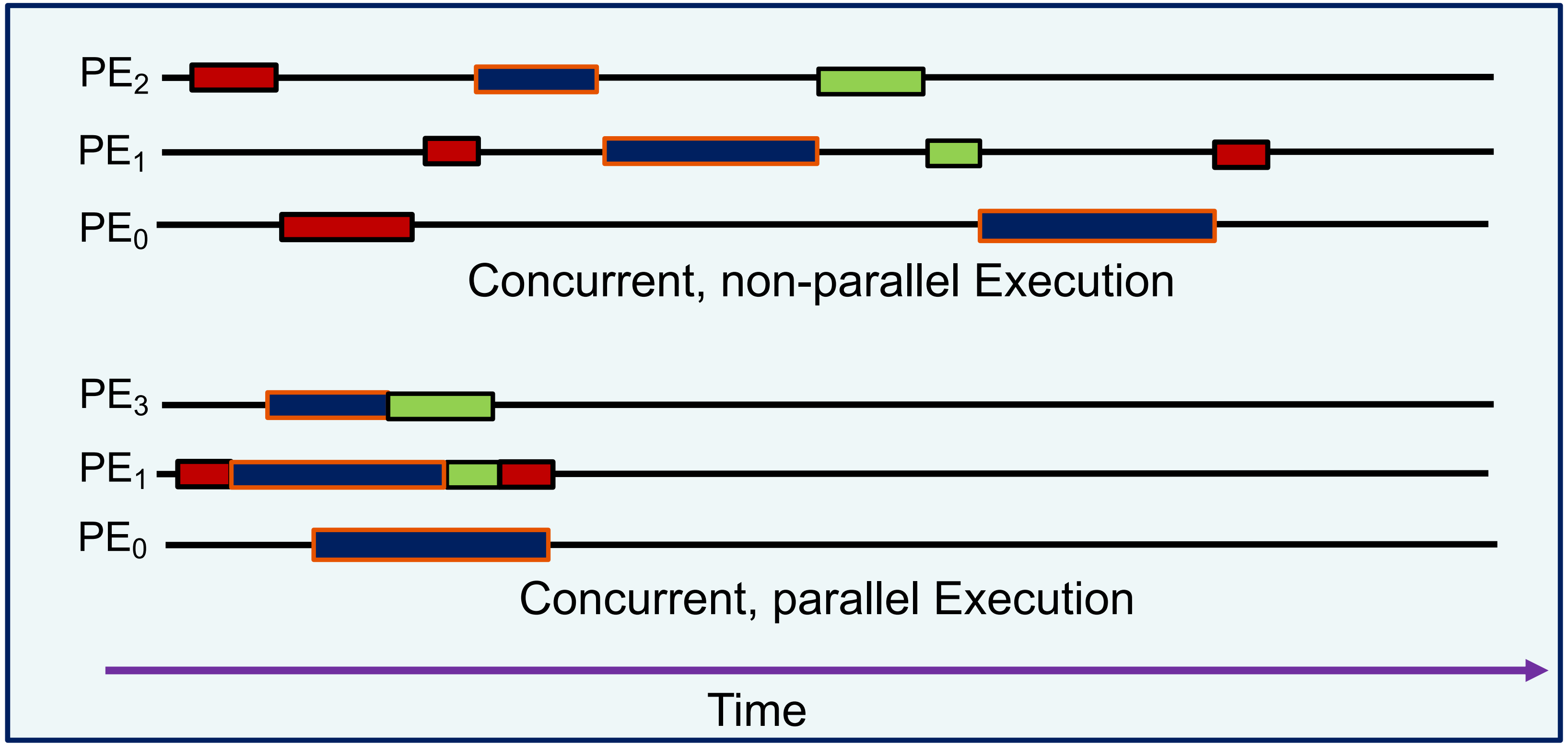
- **Computer:**
  - A machine that transforms *input data* into *output data*.
  - Typically, a computer consists of Control, Arithmetic/Logic, and Memory units.
  - The transformation is defined by a stored **program** (von Neumann architecture).



- **Task:**
  - A specific sequence of instructions plus a data environment. A program is composed of one or more tasks.
- **Active task:**
  - A task that is available to be scheduled for execution. When the task is moving through its sequence of instructions, we say it is making **forward progress**
- **Fair scheduling:**
  - When a scheduler gives each active task an equal *opportunity* for execution.

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.



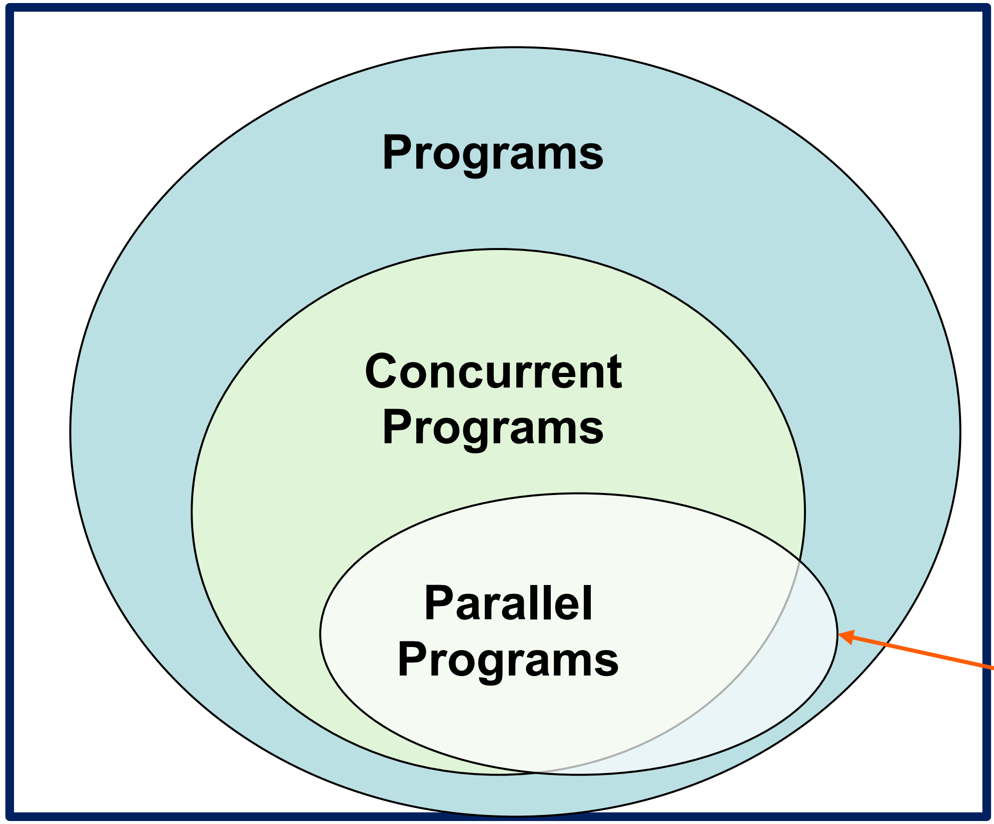
PE_2
PE_1
PE_0
Concurrent, non-parallel Execution
PE_3
PE_1
PE_0
Concurrent, parallel Execution
Time

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.



In most cases, parallel programs exploit concurrency in a problem to run tasks on multiple processing elements

We use Parallelism to:

- Do more work in less time
- Work with larger problems

If tasks execute in “lock step” they are not concurrent, but they are still parallel. Example … a SIMD unit.

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

PyOMP: OpenMP projected into Python

- A parallel multithreaded “hello world” program with PyOMP

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

@njit
def hello():
    with openmp("parallel"):
         print("hello")
         print("world")

hello()
print("DONE")
```

When I run this program, here is the output.

```
hello
world
hello
hello
hello
world
hello
world
hello
world
hello
world
world
world
hello
world
DONE
```

The challenge for programmers writing multithreaded code is to make sure every semantically allowed way statements can interleave results in correct code.

Detailed outline from the tutorial proposal

- Introducing parallel computing and PyOMP

➡
- The PyOMP system

- PyOMP and multithreading (parallelism for the CPU)

– **Break**

- GPU programming with PyOMP

- Other approaches to parallelism in Python.

- Wrap-up and Q&A

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

How did we implement PyOMP?

We used the “magic” of Numba

- PyOMP currently based on a fork of Numba 0.57
- So, PyOMP is a fully functional Numba but with OpenMP support
- At some point in the future it will become a pure extension to Numba

Numba … C-like performance from Python code

- Numba is a JIT compiler. Maps a subset of Python and NumPy API onto LLVM
- Once code is JIT'ed into LLVM, all performance enhancements exposed at the level of LLVM are directly available … result is performance that approaches that from raw C or Fortran
- Source code is pure Python for maximum portability

- Just add the @jit decorator to enable Numba for a function.

```
from numba import jit

@jit
def addit(A,B):
    return (A+B)
```

Numba JIT compiler applied the first time a function is encountered. Numba caches the code so subsequent calls to the function don't run the JIT step.

Numba defines elementwise functions called *ufuncs*

This generates the LLVM code and calls the addition ufunc to do an elementwise add of A and B

- Numerous options in numba … we are barely scratching the surface
  - @jit(nopython=true) do NOT use any Python objects in the generated code. Can be much faster. Equivalent to njit.
  - @jit(parallel=true) invoke parallel accelerator

PyOMP Implementation in Numba

- PyOMP changes to Numba:
  - Adds an OpenMP context manager
  - Provides the ability to call all the OpenMP runtime functions from both Python and Numba JITed code.

- Exception handling disabled in OpenMP regions since Numba exception mechanism breaks OpenMP single-entry/single-exit requirement.

- Variables not listed in a data clauses are SHARED if used before or after OpenMP region, PRIVATE otherwise*.

- Supports most OpenMP 3.5 and much of OpenMP 4.5. Supported directives and clauses can be found at https://pyomp.readthedocs.io/en/latest/.

- Note that one can use @jit(cache=True) Numba decorator to compile the function once and store the result on disk to avoid recompilation each time the program is restarted.

* The scope of shared/private variables exposes subtle issues in how the rules for an OpenMP data environment interacts with how Numba manages the visibility of variables. This is a topic that is still evolving, though in practice it hasn't impacted the usability of PyOMP .

How do you install PyOMP on your own system?

PyOMP installation

- Preferred installation method is through conda.

- We've simplified the installation command to the following
  - conda install -c python-for-hpc -c conda-forge --override-channels 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

Detailed outline from the tutorial proposal

- Introducing parallel computing and PyOMP
- The PyOMP system
- ➡ PyOMP and multithreading (parallelism for the CPU)

Break

- GPU programming with PyOMP
- Other approaches to parallelism in Python.
- Wrap-up and Q&A

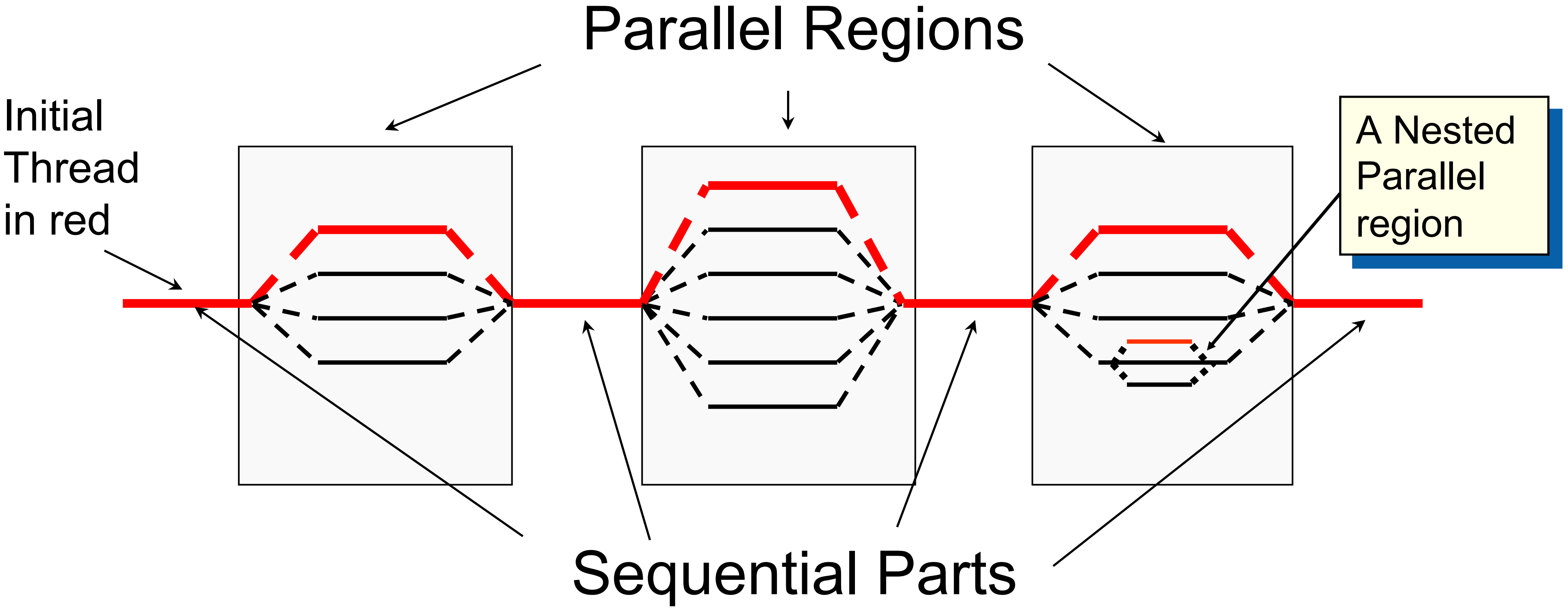
https://github.com/Python-for-HPC/PyOMP

Lets dive into the details of multithreading and how they are most commonly used in an application

OpenMP Execution Model

Fork-Join Parallelism:

- Initial thread **forks** a team of threads as needed.
- They execute in a shared address space … All reads read/write a common set of the variables.
- When the team is finished, the threads **join** together and the initial thread continues
- Parallelism added incrementally until performance goals are met, i.e., the sequential program evolves into a parallel program.



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)

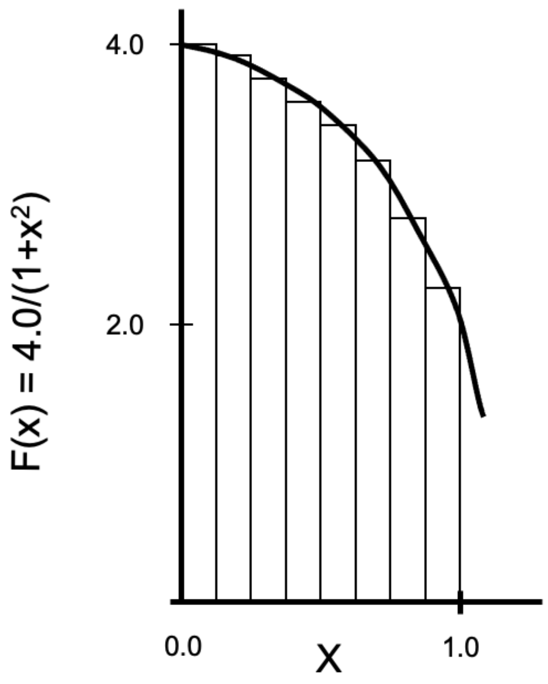
Mathematically, we know that:

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

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 ith interval midpoint.



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

The Loop-level parallelism design pattern

- Parallelism defined in terms of parallel loops … that is, loops where iterations can safely execute when divided between a collection of threads.
- Key elements:
  - identify compute intensive loops in a program
  - Expose concurrency by removing or managing loop carried dependencies
  - Exploit concurrency for parallel execution usually using a parallel loop construct/directive.

The Loop-level parallelism design pattern

- Parallelism defined in terms of parallel loops … that is, loops where iterations can safely execute when divided between a collection of threads.
- Key elements:
  - identify compute intensive loops in a program
  - Expose concurrency by removing or managing loop carried dependencies
  - Exploit concurrency for parallel execution usually using a parallel loop construct/directive.

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

The Loop-level parallelism design pattern

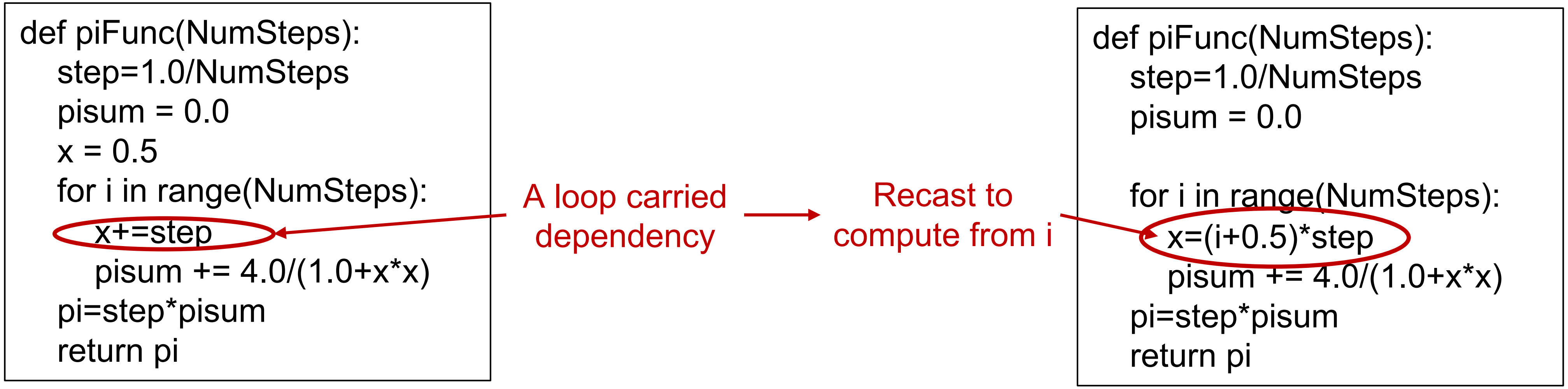
- Parallelism defined in terms of parallel loops … that is, loops where iterations can safely execute when divided between a collection of threads.
- Key elements:
  - identify compute intensive loops in a program
  - Expose concurrency by removing or managing loop carried dependencies
  - Exploit concurrency for parallel execution usually using a parallel loop construct/directive.

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

A loop carried dependency

The Loop-level parallelism design pattern

- Parallelism defined in terms of parallel loops … that is, loops where iterations can safely execute when divided between a collection of threads.
- Key elements:
  - identify compute intensive loops in a program
  - Expose concurrency by removing or managing loop carried dependencies
  - Exploit concurrency for parallel execution usually using a parallel loop construct/directive.



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

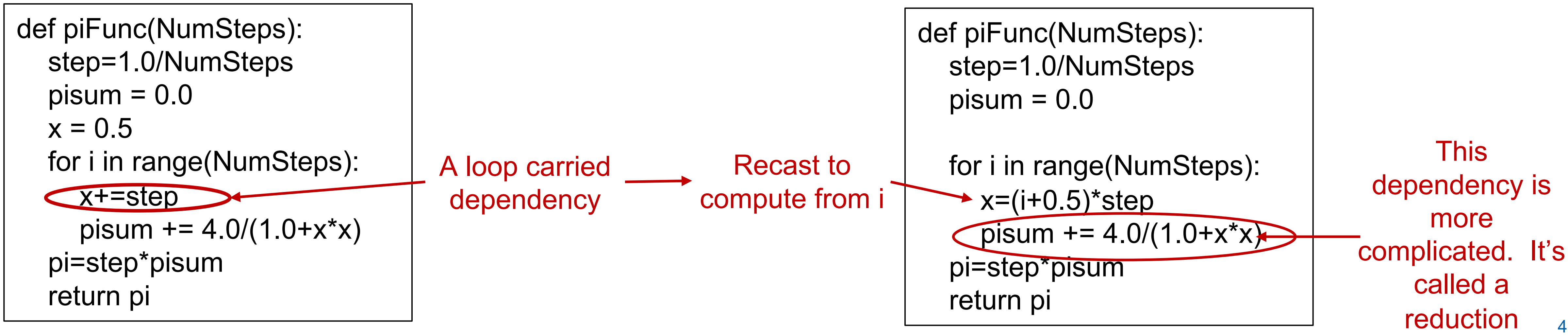
A loop carried dependency

Recast to compute from i

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

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

The Loop-level parallelism design pattern

- Parallelism defined in terms of parallel loops … that is, loops where iterations can safely execute when divided between a collection of threads.
- Key elements:
  - identify compute intensive loops in a program
  - Expose concurrency by removing or managing loop carried dependencies
  - Exploit concurrency for parallel execution usually using a parallel loop construct/directive.



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

A loop carried dependency

Recast to compute from i

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

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

This dependency is more complicated. It's called a reduction

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

Reduction

- OpenMP reduction clause added to a **parallel for**:

  reduction (op : list)

- Inside the **parallel for**:
  - Each thread gets a private copy of each variable in **list** … initialized depending on the "op"
    (e.g., 0 for "+").
  - Updates to the reduction variable from each thread happens to its private copy.
  - The private copies from each thread are combined into a single value … and then combined with the original global value … all using the **op** from the reduction clause.

- The variables in the "list" must be shared in the enclosing parallel region.

```
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)
```

We don't discuss the details here, but you can also add a **reduction** clause to a **parallel** or a **for** construct.

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

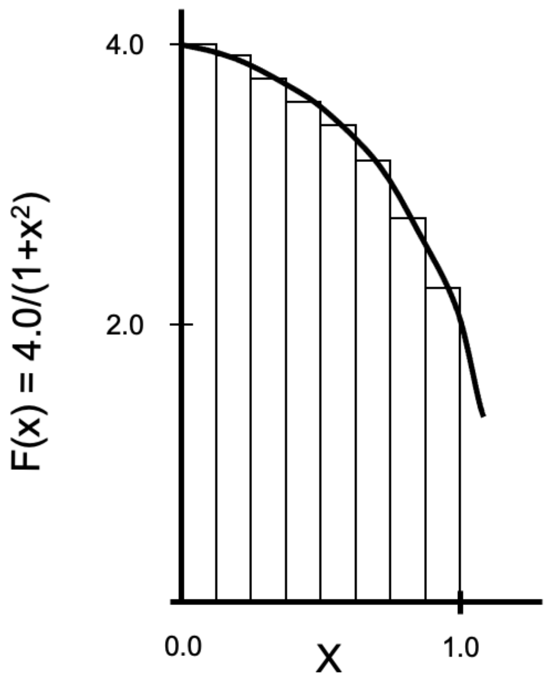
Ran each case 5 times and kept the minimum time. **JIT time is not included** for PyOMP (it was about 1.5 seconds)

Parallel Loop are great … but sometimes you want more control over individual threads

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:

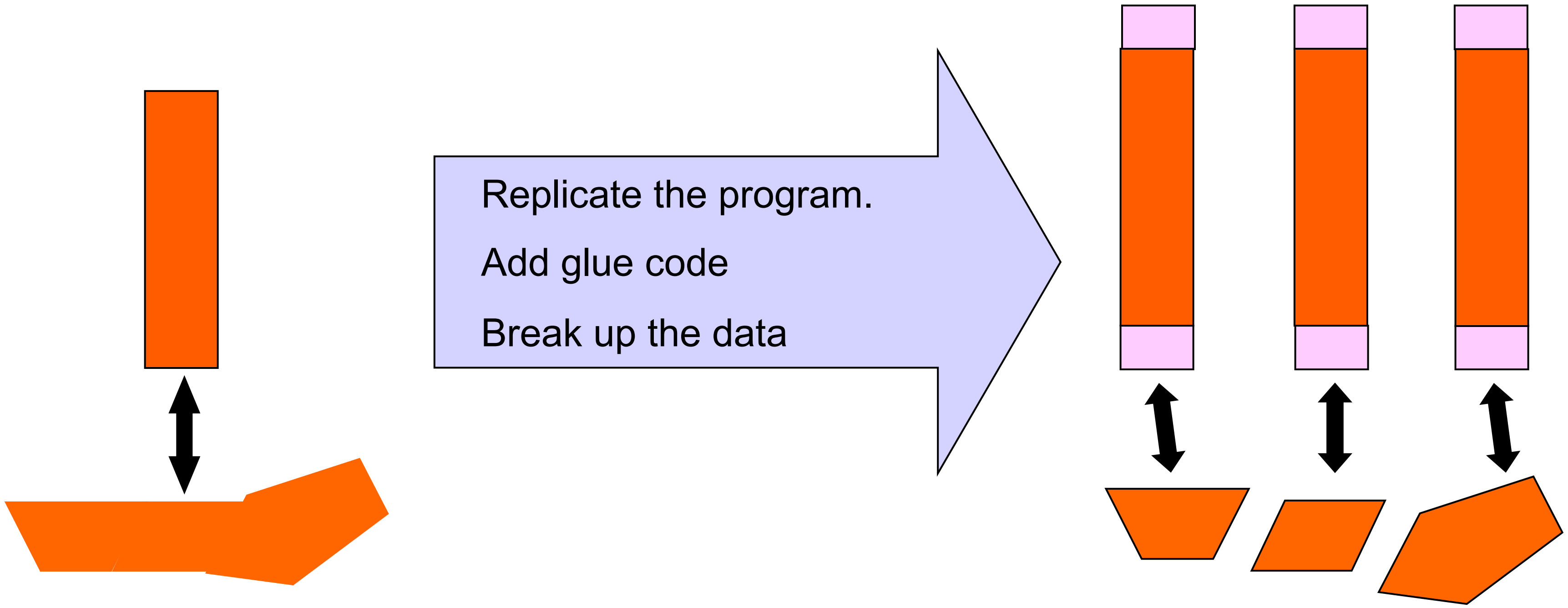
$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

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

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

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.



Replicate the program.

Add glue code

Break up the data

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.

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)

How do we handle problems without such regular structure or with complex load balancing problems?

We do this in OpenMP with explicit tasks

Explicit tasks in PyOMP

- A task is a sequence of statements and an associated data environment. Lots of flexibility in how those tasks are created, so handles irregular parallelism, recursive parallelism, and many other control structures.

- A common pattern … one thread creates explicit tasks and puts them in a queue. All the threads work together to execute them. The single construct causes one thread to execute statements while the other threads wait at a barrier at the end of the single. It’s perfect for task level parallelism.

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

@njit
def irregularPar():
    with openmp("parallel"):
       with openmp("single"):
            StateVal = 1
            while (StateVal > 0):
                 with openmp("task firstprivate(StateVal)"):
                     BigComp(StateVal)
                 StateVal = ExitYet()
    return

irregularPar()
```

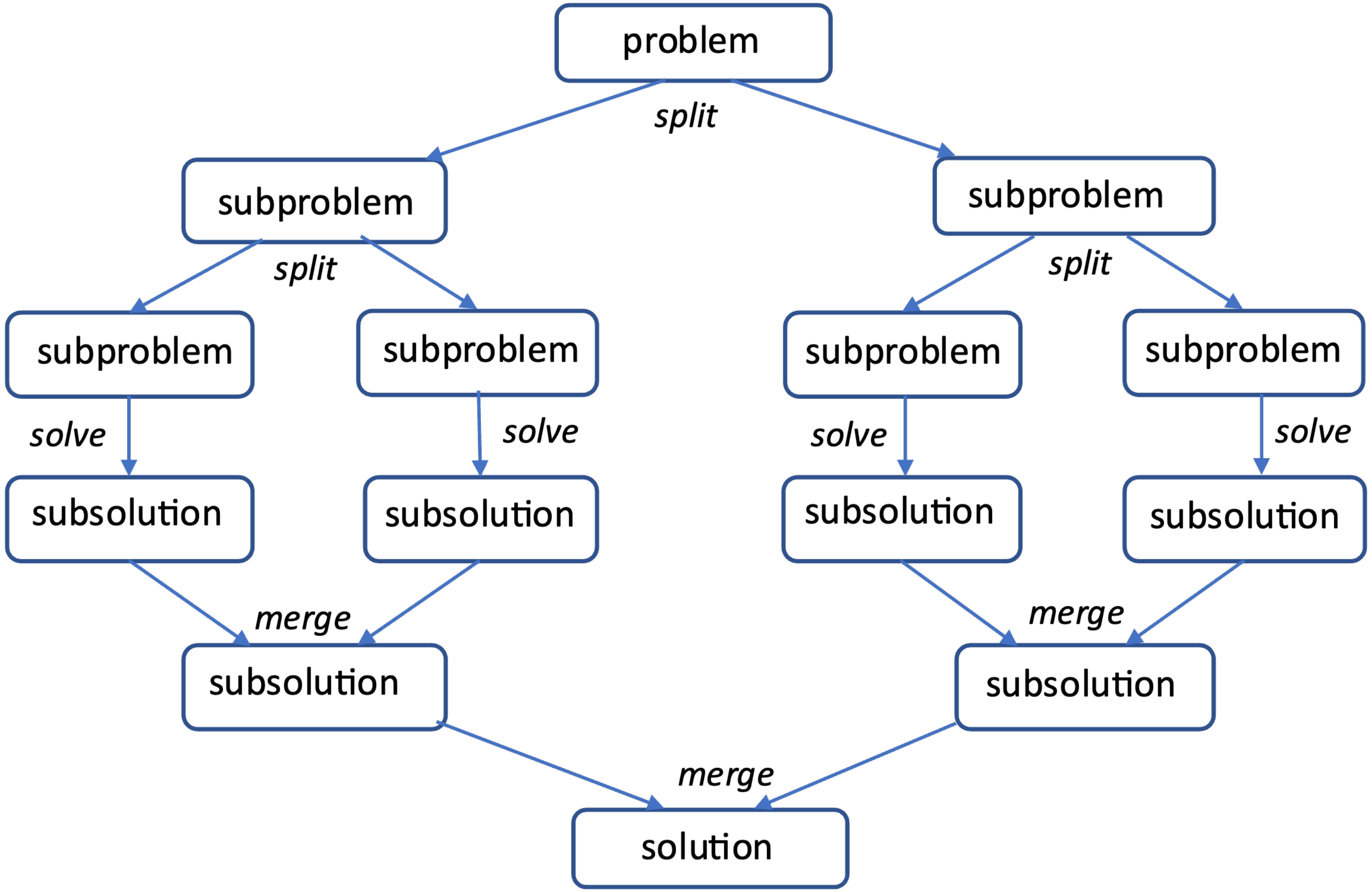
Single: one thread does the work while the other threads wait (and execute tasks) at the barrier implied at the end of single

An explicit task … captures value of the variable `StateVal` and calls BigComp.

Returns a negative value at some point (function not shown)

Divide and conquer design pattern

- Split the problem into smaller sub-problems; continue until the sub-problems can be solved 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):
        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

PyOMP subset of OpenMP for CPU programming

`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

PyOMP subset of OpenMP for CPU programming

`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

Fork threads

Work sharing

Synchronization

Par. Loop support

Data Environment

runtime libraries

Environment

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]
```

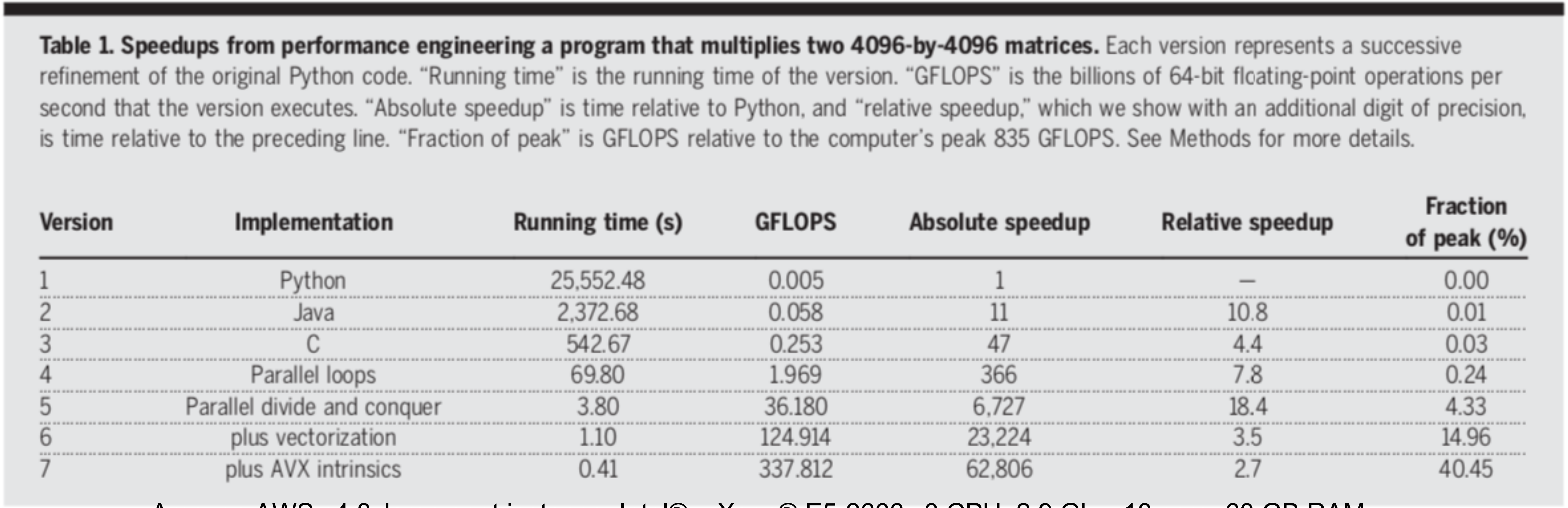


Table 1. Speedups from performance engineering a program that multiplies two 4096-by-4096 matrices. Each version represents a successive refinement of the original Python code. “Running time” is the running time of the version. “GFLOPS” is the billions of 64-bit floating-point operations per second that 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

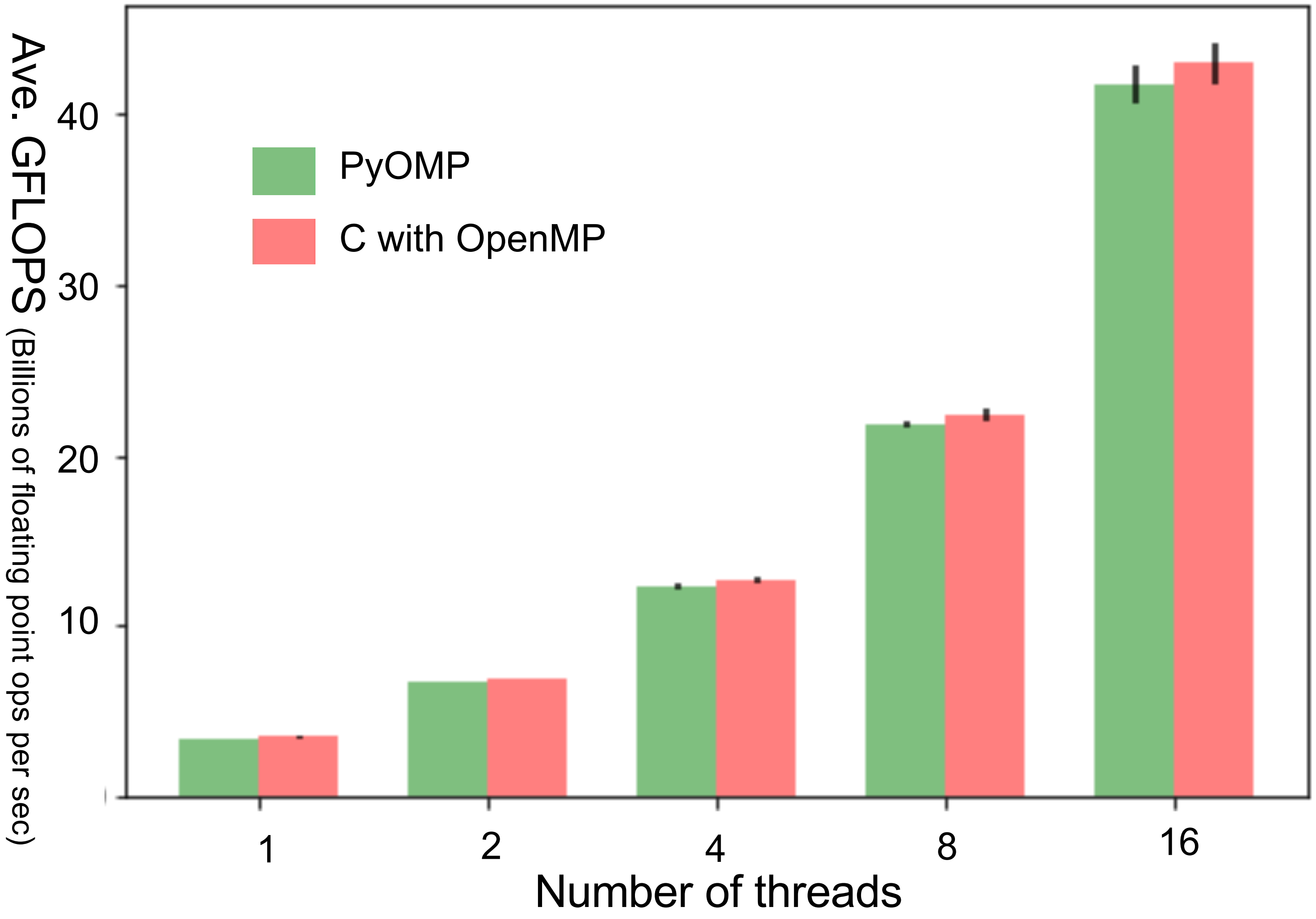
PyOMP DGEMM (Mat-Mul with double precision numbers)

```
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

Matrix Multiplication, double precision, order = 1000, with error bars (std dev)



250 runs for order 1000 matrices

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

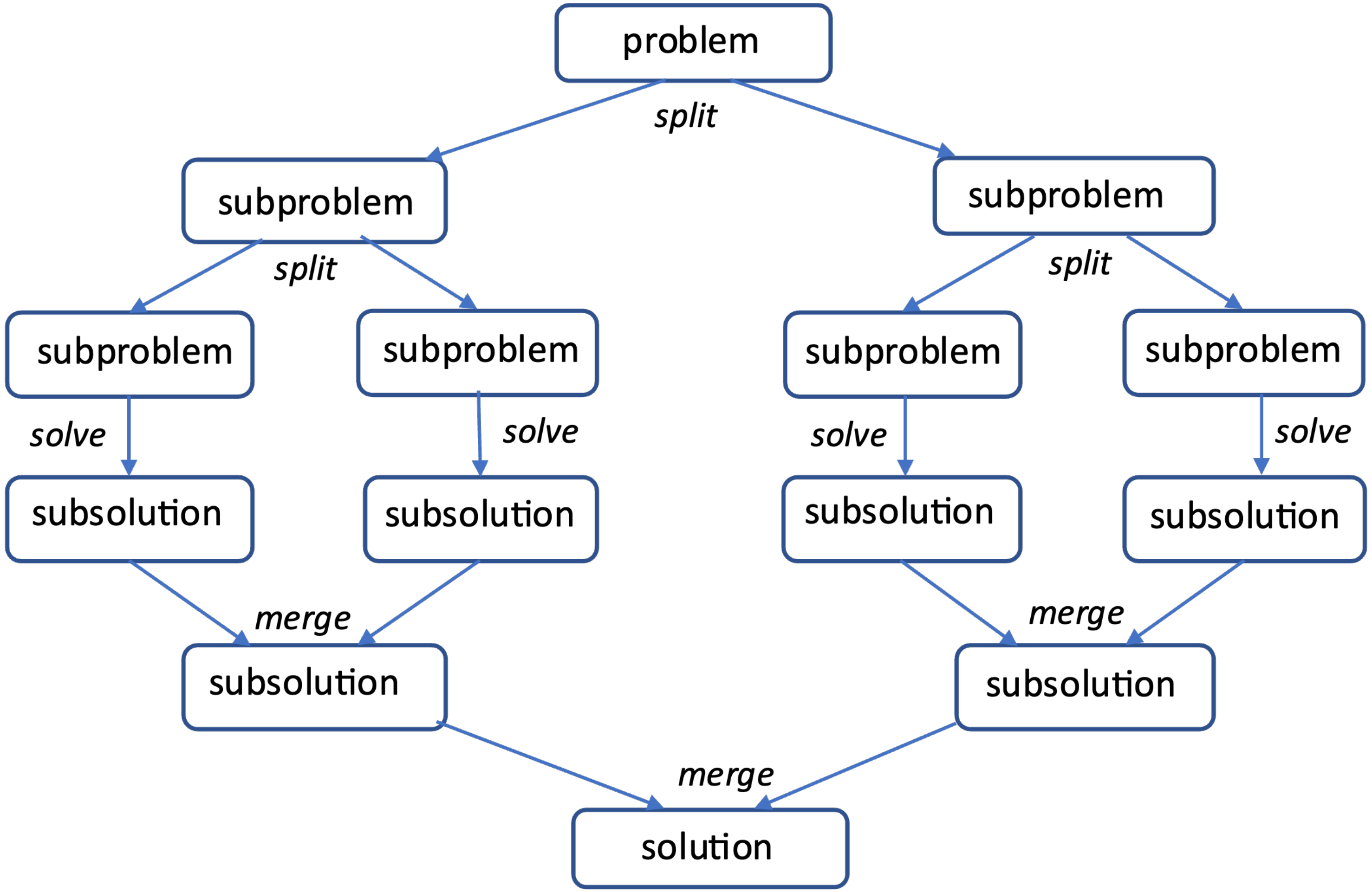
… but remember, the JIT'ed code can be cached for future use. It's straightforward to hide the JIT cost.

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)

… and in talking about PyOMP we have covered three of the key design patterns in parallel programming

Divide and conquer design pattern

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

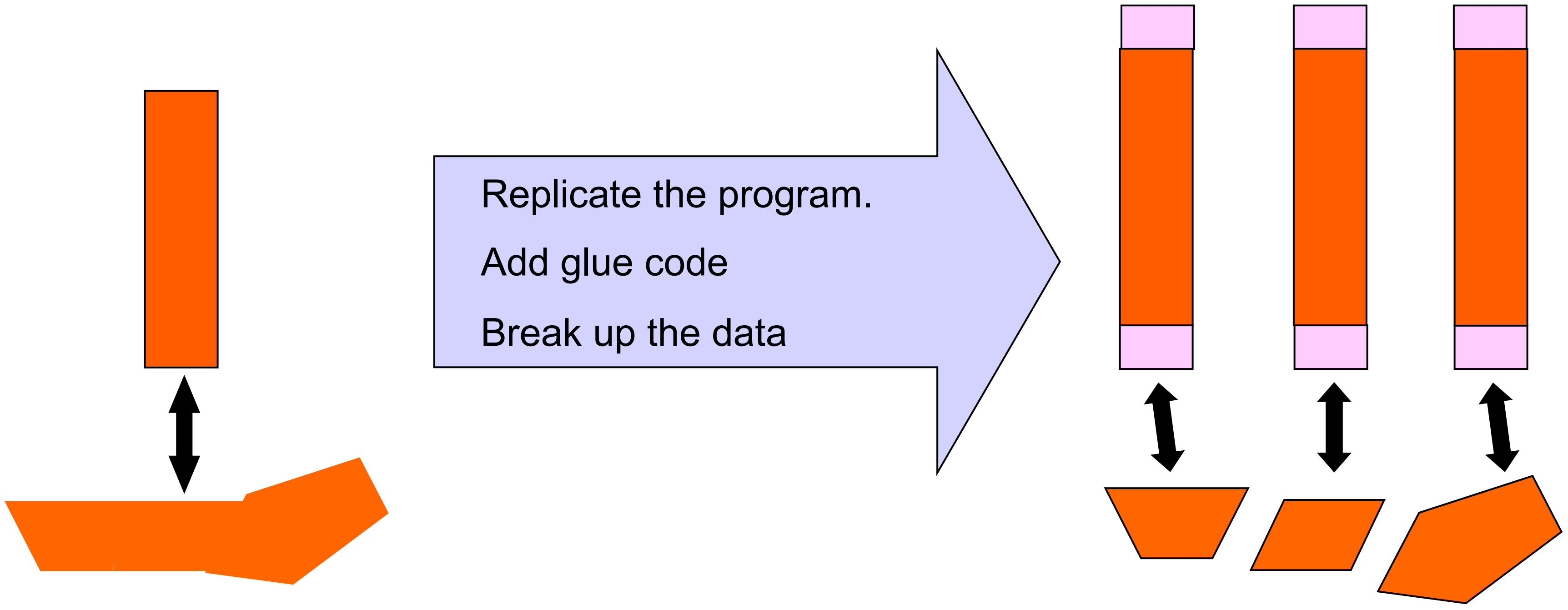


3 Options for parallelism:

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

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.



Replicate the program.

Add glue code

Break up the data

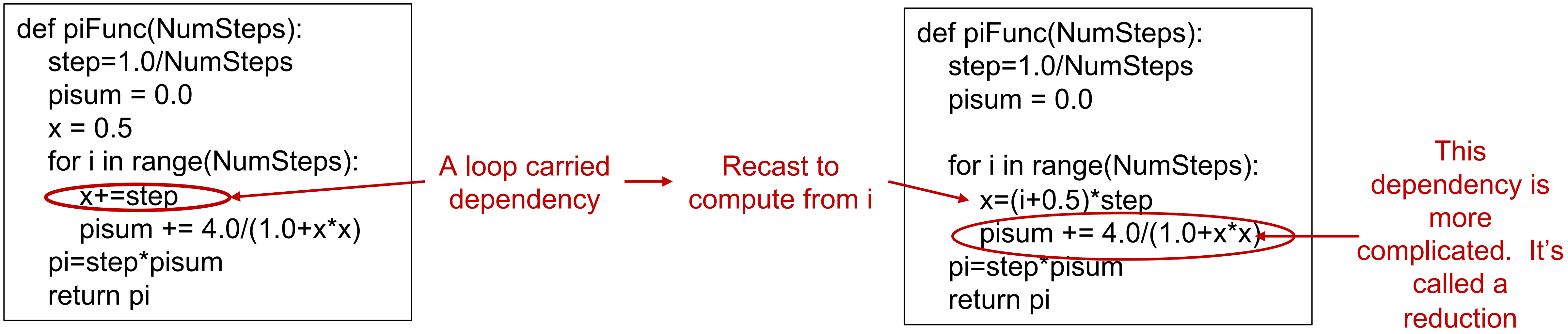
This pattern is very general and has been used to support most (if not all) the algorithm strategy patterns.

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

Third party names are the property of their owners

The Loop-level parallelism design pattern

- Parallelism defined in terms of parallel loops … that is, loops where iterations can safely execute when divided between a collection of threads.
- Key elements:
  - identify compute intensive loops in a program
  - Expose concurrency by removing or managing loop carried dependencies
  - Exploit concurrency for parallel execution usually using a parallel loop construct/directive.



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

A loop carried dependency

Recast to compute from i

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

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

This dependency is more complicated. It's called a reduction

Outline

- Introducing parallel computing and PyOMP
- The PyOMP system
- PyOMP and multithreading (parallelism for the CPU)

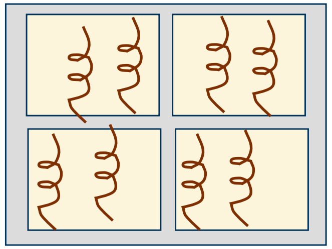
Break

- GPU programming with PyOMP
- Other approaches to parallelism in Python.
- Wrap-up and Q&A

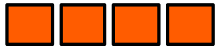
https://github.com/Python-for-HPC/PyOMP

For hardware … parallelism is the path to performance

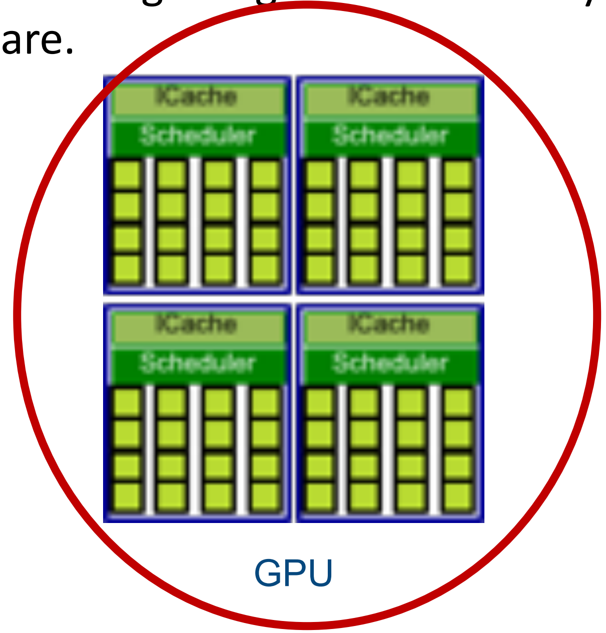
All hardware vendors are in the game … parallelism is ubiquitous so if you care about getting the most from your hardware, you will need to create parallel software.



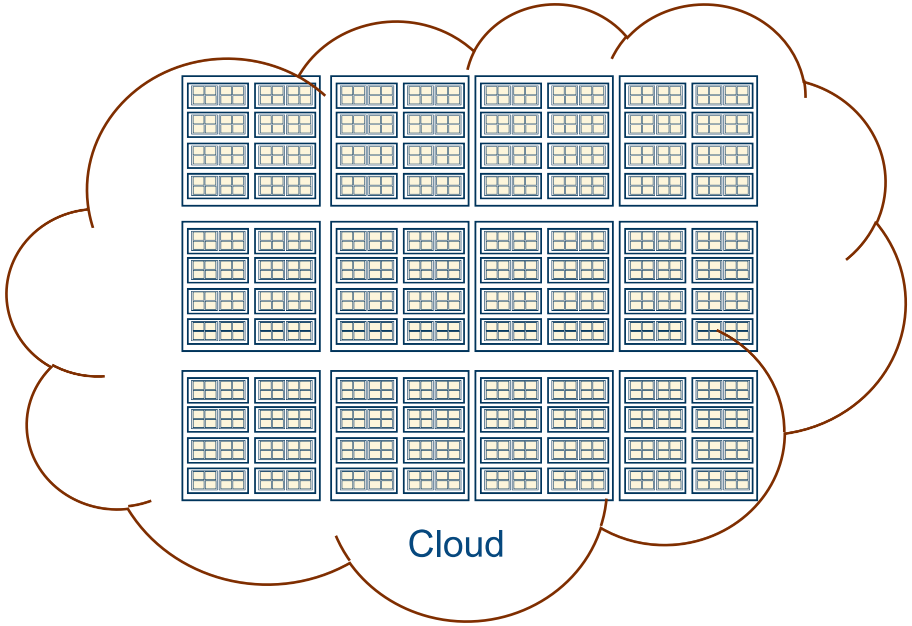
CPU



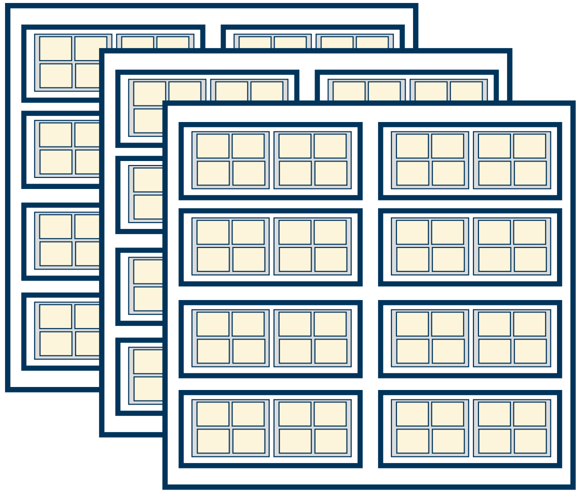
SIMD/Vector



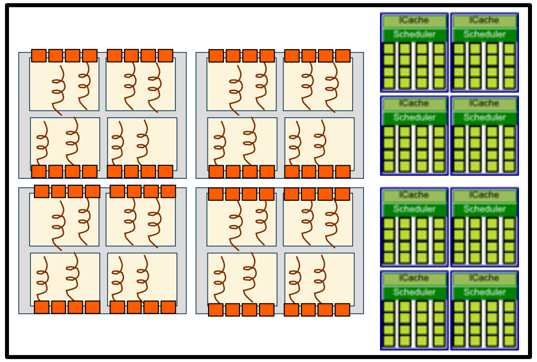
GPU



Cloud



Cluster



Heterogeneous node

Let’s start by understanding GPU programming in general … and then see how it maps onto PyOMP

The “BIG idea” Behind GPU programming

Traditional Loop based vector addition (vadd)

```
int main() {
   int N = . . . ;
   float *a, *b, *c;

   a* =(float *) malloc(N * sizeof(float));

   // ... allocate other arrays (b and c)
   // and fill with data

   for (int i=0;i<N; i++)
      c[i] = a[i] + b[i];

}
```

Data Parallel vadd with CUDA

```
// Compute sum of length-N vectors: C = A + B
void __global__
vecAdd (float* a, float* b, float* c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) c[i] = a[i] + b[i];
}

int main () {
    int N = ... ;
    float *a, *b, *c;
    cudaMalloc (&a,  sizeof(float) * N);
  // ... allocate other arrays (b and c)
  // and fill with data

  // Use thread blocks with 256 threads each
    vecAdd <<< (N+255)/256, 256 >>> (a, b, c, N);
}
```

Assume a GPU with unified shared memory … allocate on host, visible on device too

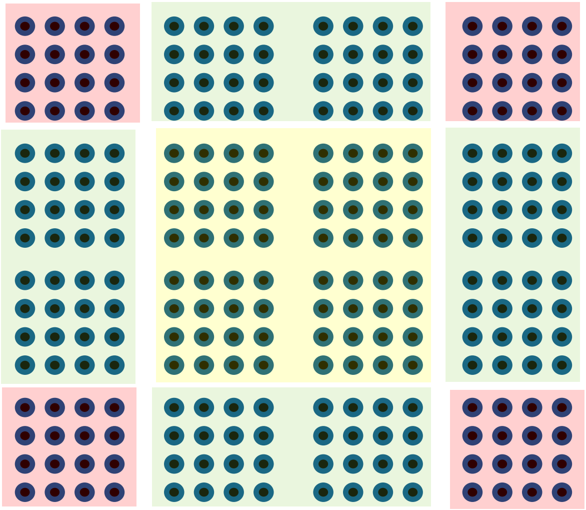
How do we execute code on a GPU: The SIMT model (Single Instruction Multiple Thread)

1. Turn source code into a scalar work-item

```
// Compute sum of length-N vectors: C = A + B
void __global__
vecAdd (float* a, float* b, float* c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) c[i] = a[i] + b[i];
}

int main () {
    int N = ... ;
    float *a, *b, *c;
    cudaMalloc (&a,  sizeof(float) * N);
  // ... allocate other arrays (b and c)
  // and fill with data

  // Use thread blocks with 256 threads each
    vecAdd <<< (N+255)/256, 256 >>> (a, b, c, N);
}
```

This is CUDA code … the sort of code the OpenMP compiler generates on your behalf

2. Map work-items onto an N dim index space.



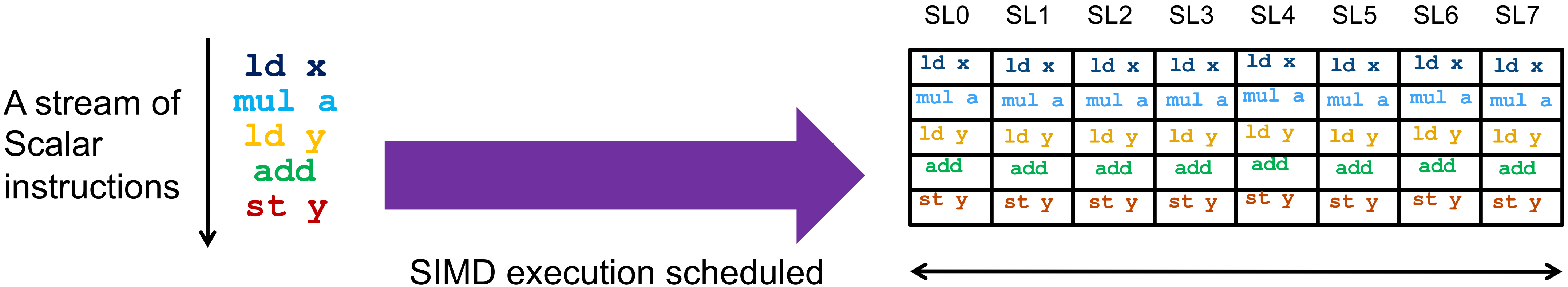
3. Map data structures onto the same index space

4. Run on hardware designed around the same SIMT execution model

Note: The CUDA code defines a 1D grid. I show a 2D grid on this slide to make kernel execution and its relation to data more clear.

SIMT: One instruction stream maps onto many SIMD lanes

- SIMT model: Individual scalar instruction streams are grouped together for SIMD execution on hardware



A stream of Scalar instructions

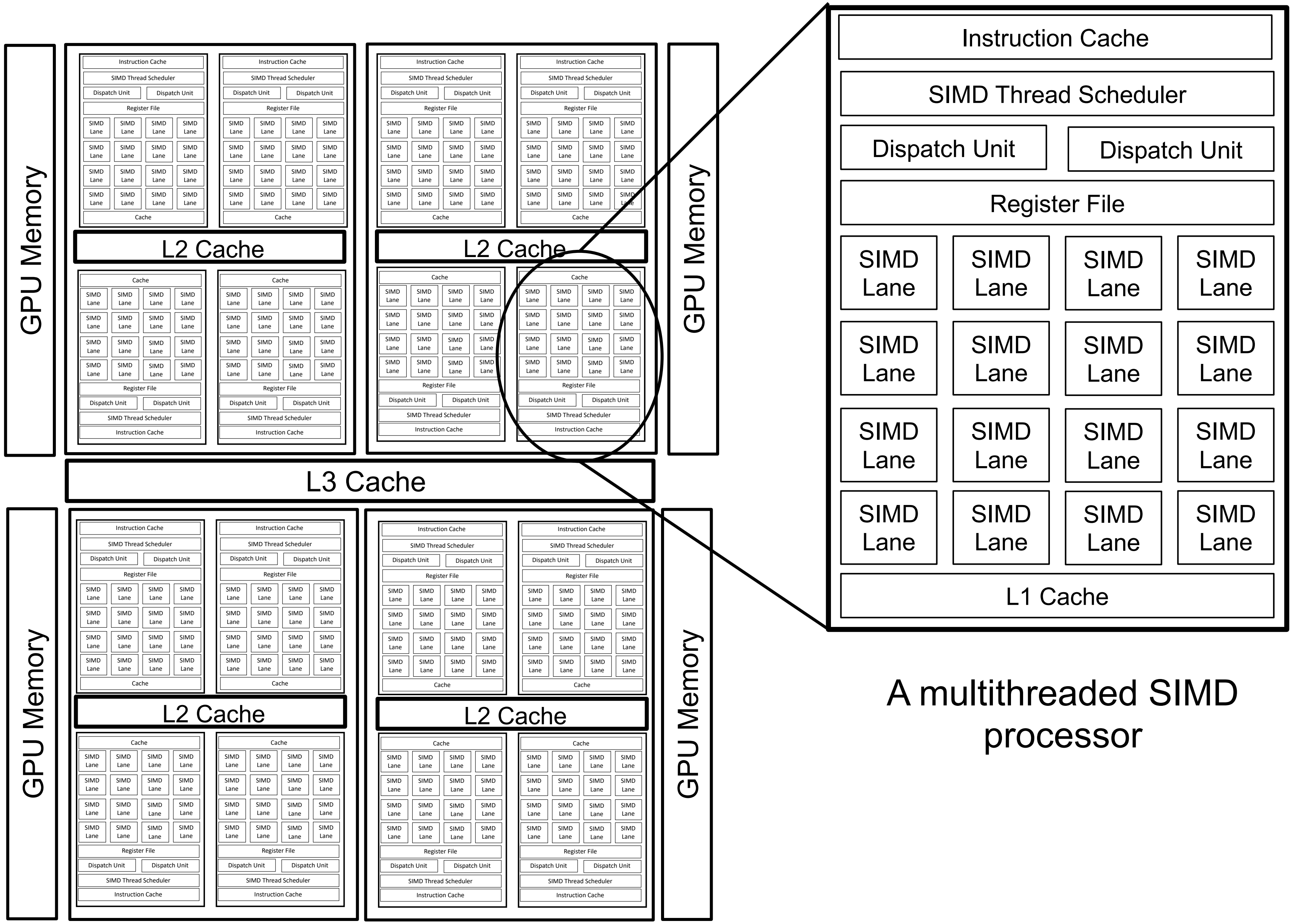
```
ld x
mul a
ld y
add
st y
```

SL0	SL1	SL2	SL3	SL4	SL5	SL6	SL7
ld x	ld x	ld x	ld x	ld x	ld x	ld x	ld x
mul a	mul a	mul a	mul a	mul a	mul a	mul a	mul a
ld y	ld y	ld y	ld y	ld y	ld y	ld y	ld y
add	add	add	add	add	add	add	add
st y	st y	st y	st y	st y	st y	st y	st y

SIMD execution scheduled across a fixed number of SIMD Lanes (SL)

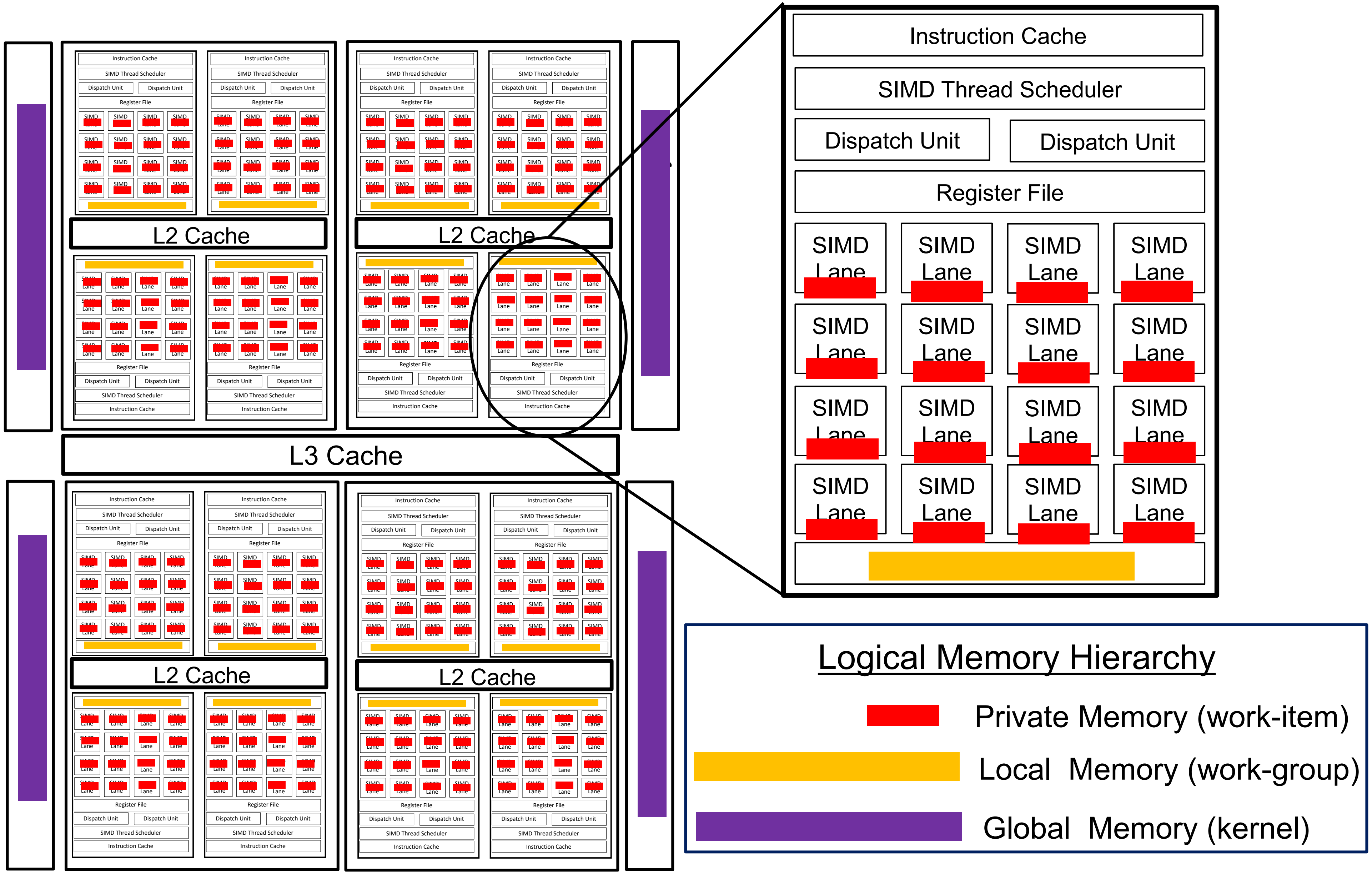
NVIDIA calls this set of work-items a **warp**

A Generic GPU (following Hennessey and Patterson)



A multithreaded SIMD processor

A Generic GPU (following Hennessey and Patterson)



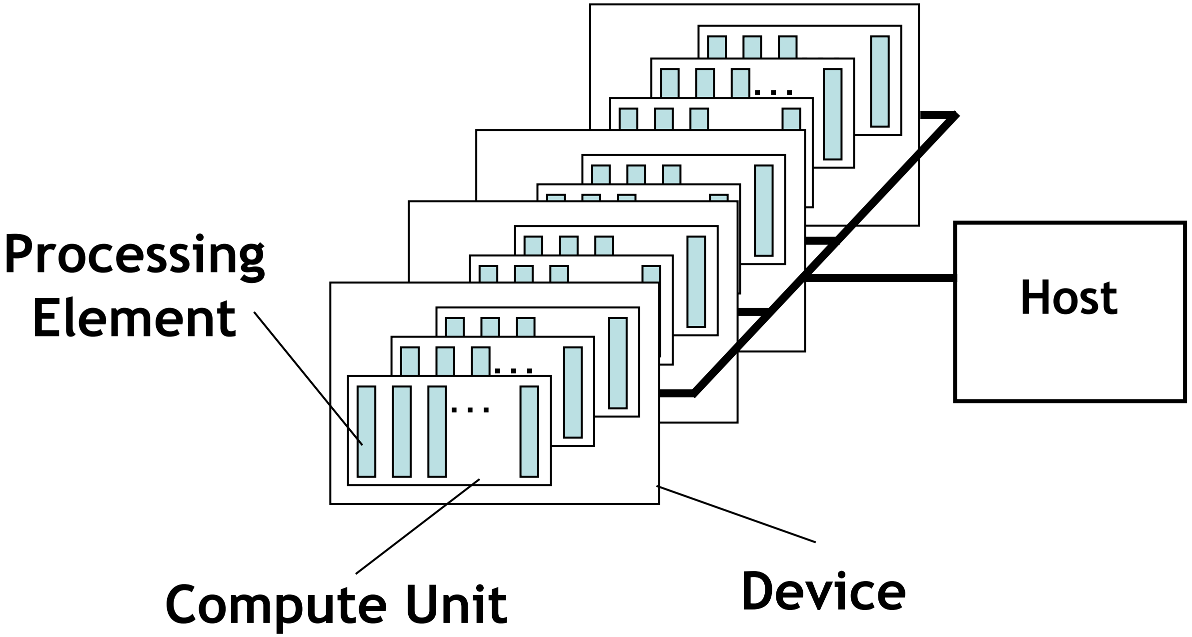
Logical Memory Hierarchy

🟥 Private Memory (work-item)

🟨 Local Memory (work-group)

🟪 Global Memory (kernel)

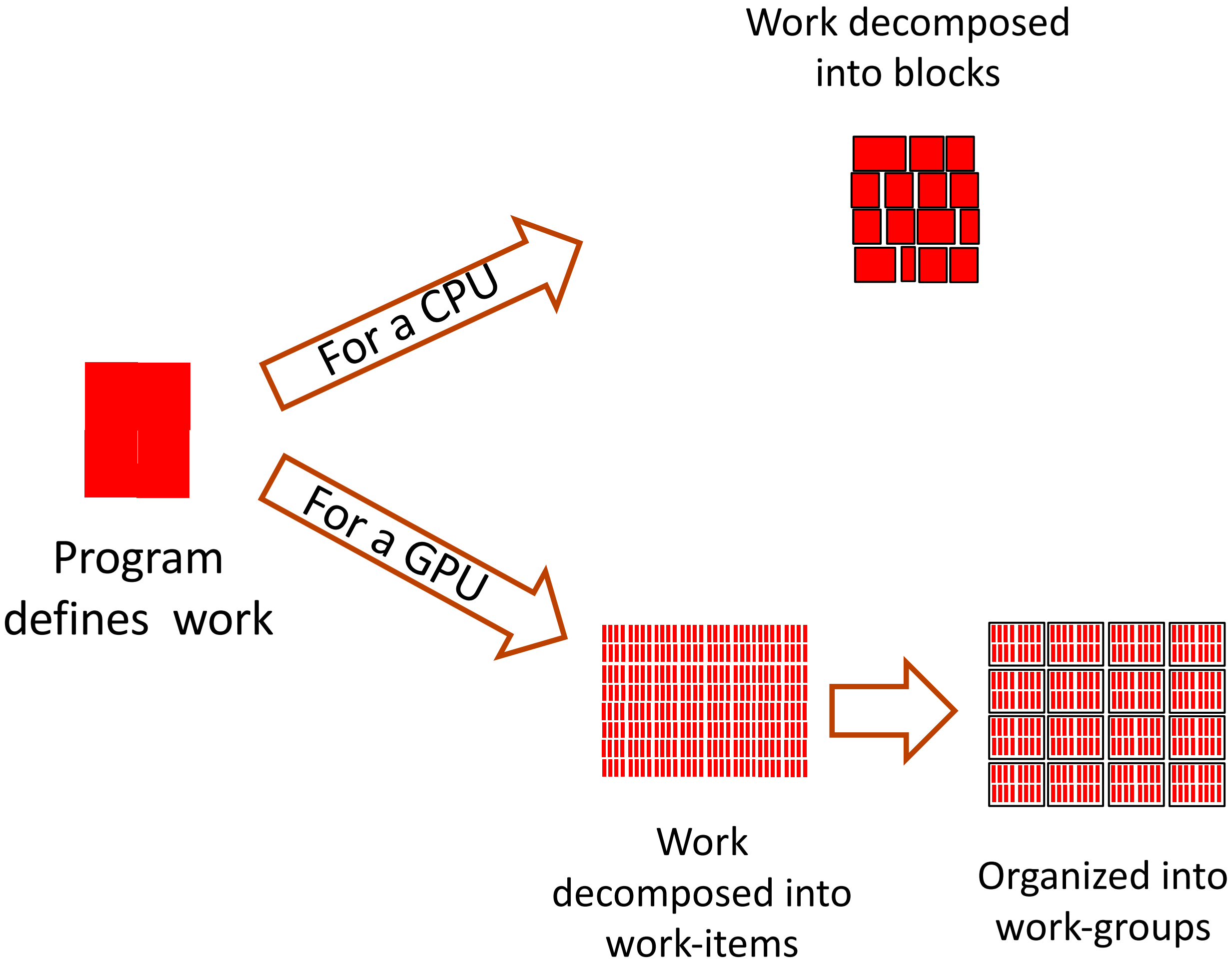
A Generic Host/Device Platform Model



- One ***Host*** and one or more ***Devices***
  - Each Device is composed of one or more Compute Units
  - Each Compute Unit is divided into one or more ***Processing Elements***
- Memory divided into ***host memory*** and ***device memory***

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

Executing a program on CPUs and GPUs



Work decomposed into blocks

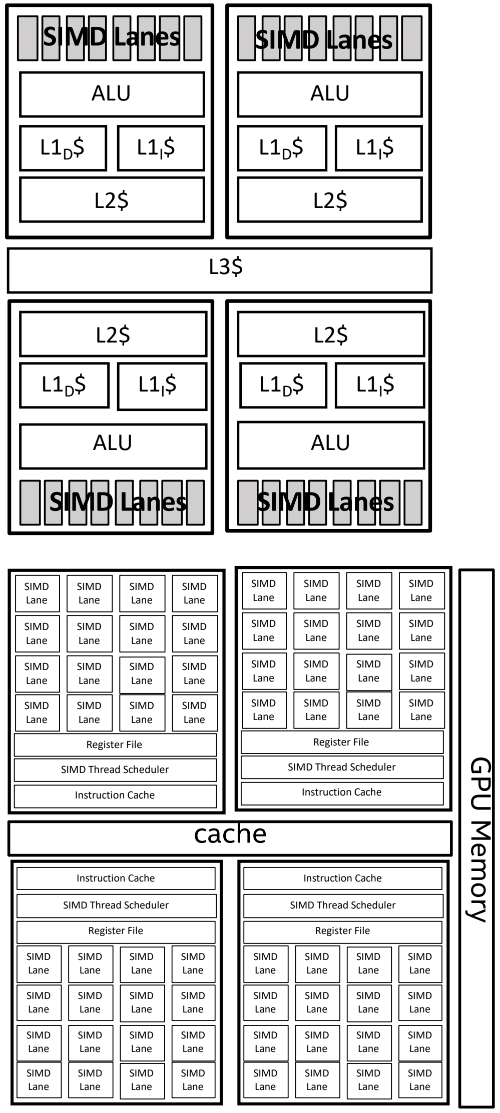
For a CPU

Program defines work

For a GPU

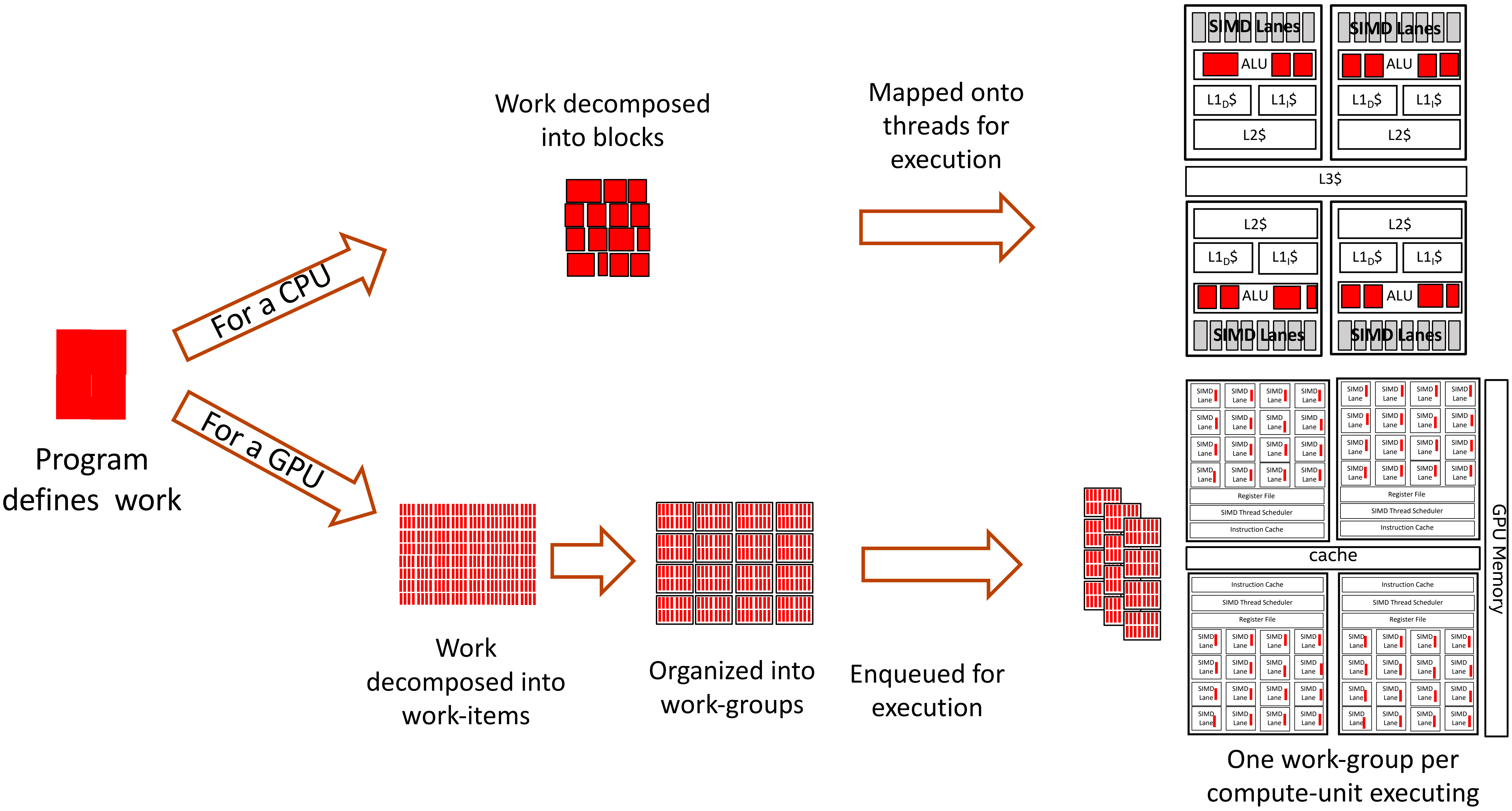
Work decomposed into work-items

Organized into work-groups



One work-group per compute-unit executing

Executing a program on CPUs and GPUs



Program defines work

Work decomposed into blocks

Mapped onto threads for execution

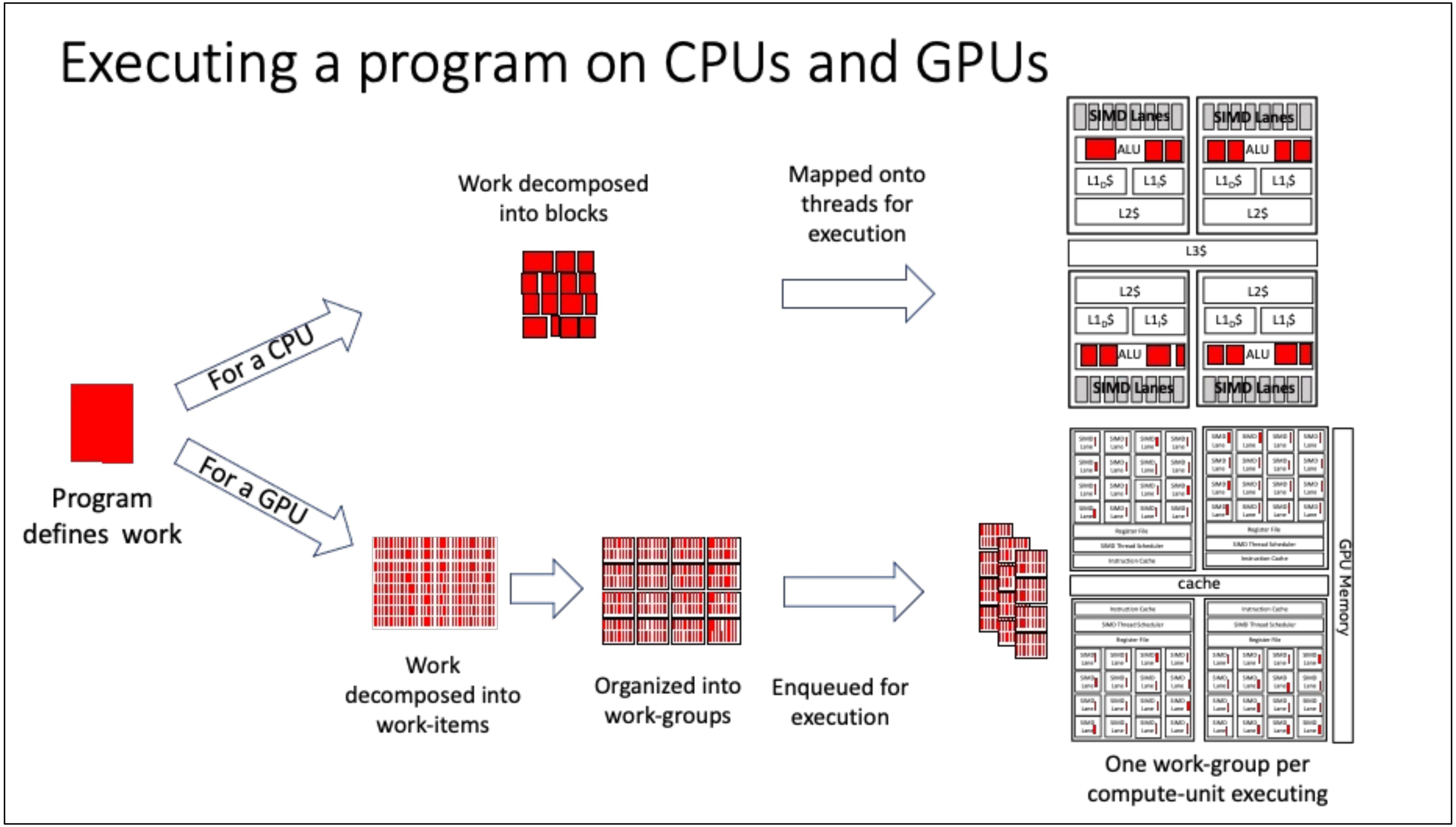
Work decomposed into work-items

Organized into work-groups

Enqueued for execution

One work-group per compute-unit executing

CPU/GPU execution models



Executing a program on CPUs and GPUs

Program defines work

For a CPU

Work decomposed into blocks

Mapped onto threads for execution

For a GPU

Work decomposed into work-items

Organized into work-groups

Enqueued for execution

One work-group per compute-unit executing

For a CPU, the threads are all active and able to make forward progress.

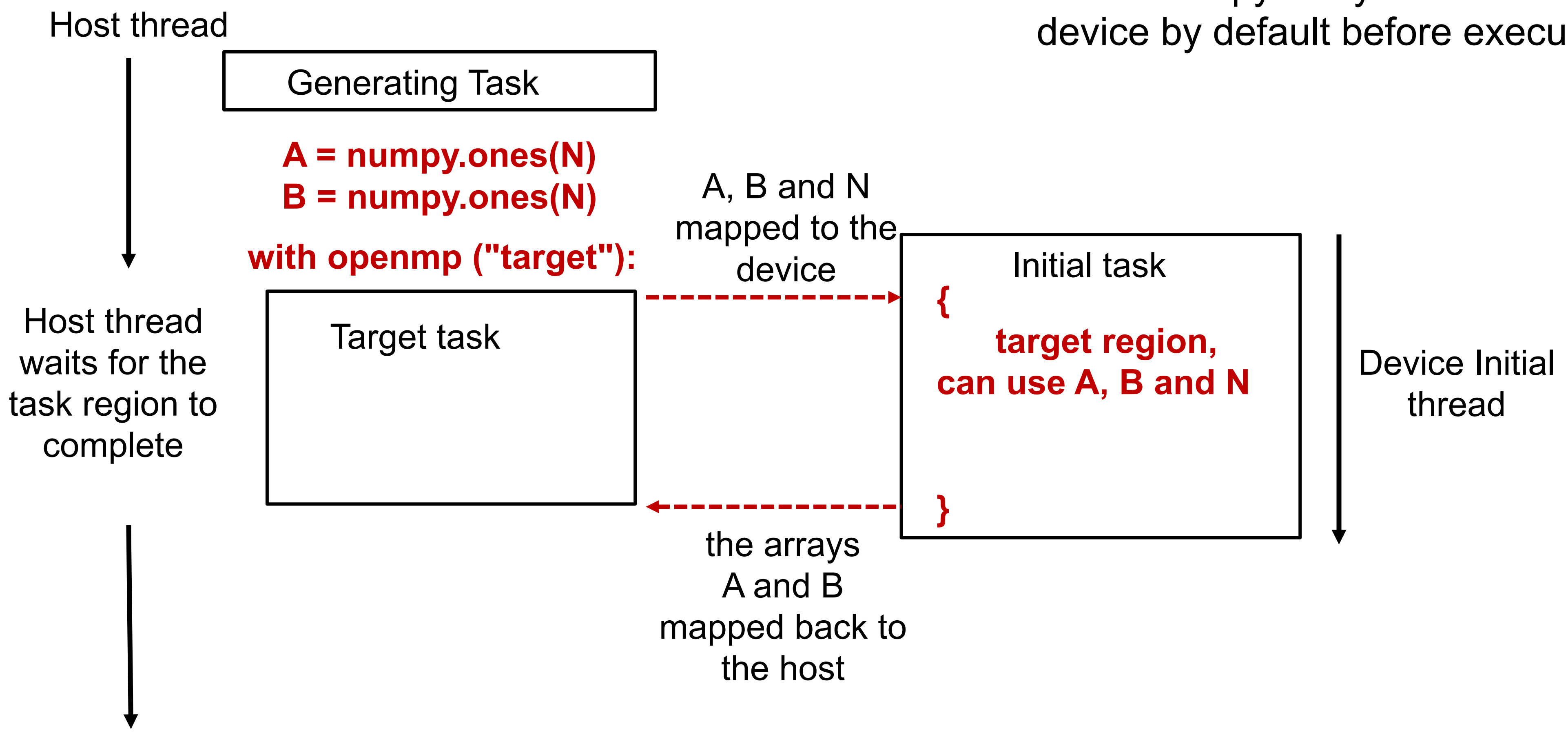
For a GPU, any given work-group might be in the queue waiting to execute.

How do we map a loop onto the GPU execution model in PyOMP?

Step 1: move code and data onto the GPU:

The target construct and default data movement

Scalars and numpy arrays are moved onto the device by default before execution.



Host thread

Generating Task

A = numpy.ones(N)
B = numpy.ones(N)

with openmp ("target"):

Host thread waits for the task region to complete

Target task

A, B and N mapped to the device

Initial task
{
target region,
can use A, B and N
}

the arrays A and B mapped back to the host

Device Initial thread

Only the arrays are moved back to the host after the target region completes

Based on figure 6.4 in **Using OpenMP – The Next Step** by van der Pas, Stotzer and Terboven, MIT Press, 2017

Step 2: Map loop iterations onto the GPU’s SIMD lanes

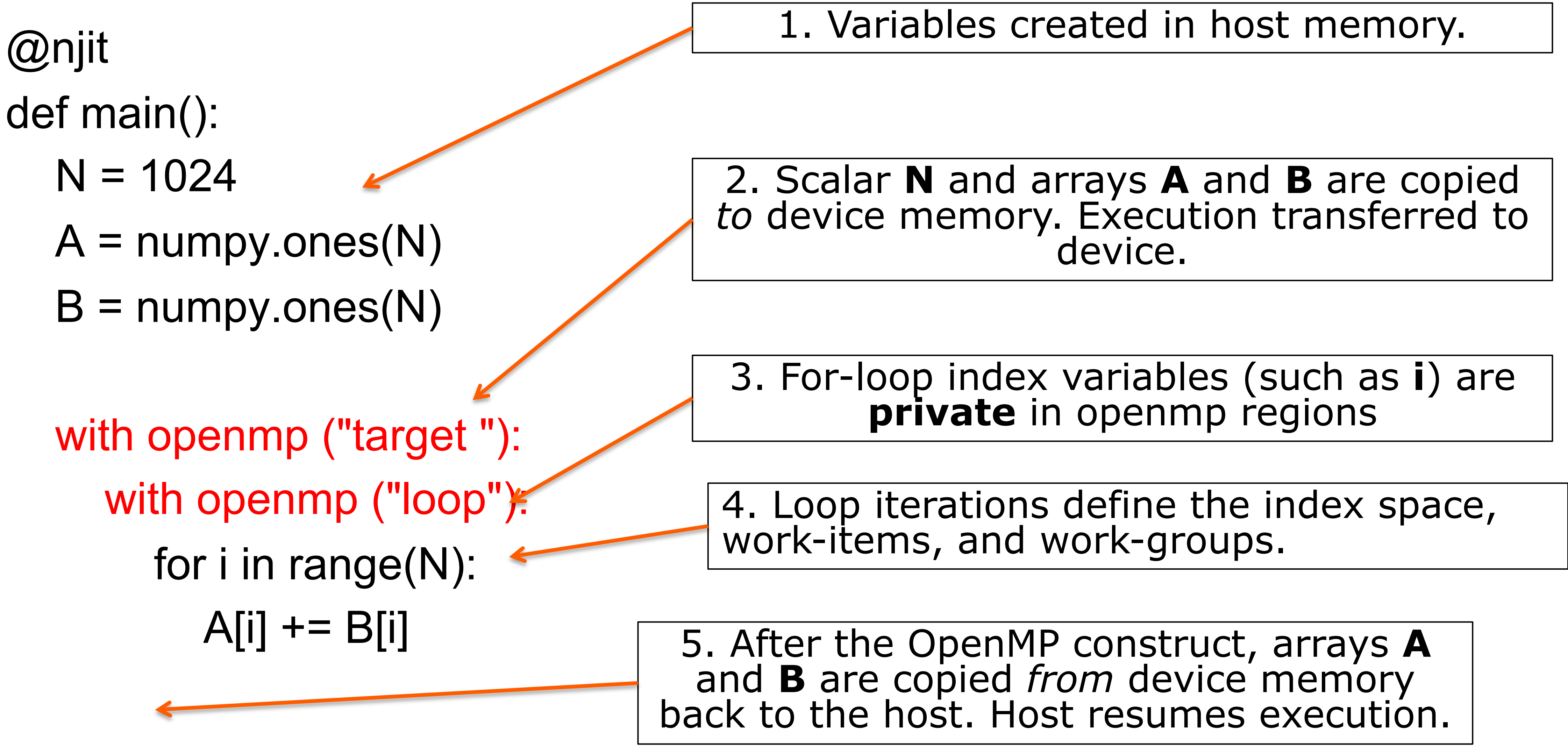
```
@njit
def main():
    N = 1024
    A = numpy.ones(N)
    B = numpy.ones(N)

    with openmp ("target "):
        with openmp ("loop"):
            for i in range(N):
                A[i] += B[i]
```

The loop construct tells the compiler:

> *“this loop will execute correctly if the loop iterations run in any order. You can safely run them concurrently. And the loop-body doesn’t contain any OpenMP constructs. So do whatever you can to make the code run fast”*

The loop construct is a declarative construct. You tell the compiler what you want done but you DO NOT tell it how to “do it”. This is new for OpenMP

Step 2: Map loop iterations onto the GPU’s SIMD lanes



```
@njit
def main():
    N = 1024
    A = numpy.ones(N)
    B = numpy.ones(N)

    with openmp ("target "):
        with openmp ("loop"):
            for i in range(N):
                A[i] += B[i]
```

1. Variables created in host memory.

2. Scalar **N** and arrays **A** and **B** are copied *to* device memory. Execution transferred to device.

3. For-loop index variables (such as **i**) are **private** in openmp regions

4. Loop iterations define the index space, work-items, and work-groups.

5. After the OpenMP construct, arrays **A** and **B** are copied *from* device memory back to the host. Host resumes execution.

Difference from OpenMP/C: PyOMP only has NumPy arrays, which carry size information.   So, PyOMP arrays sent in full by default … as it is with C static-arrays.

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("parallel for private(j,k)"):
        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.

Create a team of threads. Map loop iterations onto them

- **parallel**: creates a team of threads
- **for**: maps loop iterations onto threads.
- **private(j,k)**: each threads gets its own j and k variables
- Loop control index of a parallel for (**i**) is private to each thread.

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.

Implicit data movement covers a small subset of the cases you need in a real program.

To be more general … we need to manage data movement explicitly

Implicit data movement

- Previously, we described the rules for *implicit* data movement … **N**, **A** and **B** moved to the GPU on entry to the target construct. **A** and **B** moved to the CPU on exit from the target construct.

- Notice that in this case, **B** is not changed on the GPU … moving it is a waste of resources

```
@njit
def main():
    N = 1024
    A = numpy.ones(N)
    B = numpy.ones(N)

    with openmp ("target"):
        for i in range(N):
            A[i] += B[i]
```

Controlling data movement with the map clause

```
@njit
def main():
    N = 1024
    A = numpy.ones(N)
    B = numpy.ones(N)

    with openmp ("target map(tofrom: A) map(to: B)"):
        for i in range(N):
            A[i] += B[i]
```

map(tofrom: A) Map data at the start and end of **target** region.

map(to: B) map data at the start of **target** region but NOT at the end.

We use the term “map” since depending on the detailed memory architecture of the CPU and the GPU, data may be in a shared address space so copying may not be needed.

PyOMP array notation

- When mapping data arrays, if you only give the array name then PyOMP transfers the entire array (using the NumPy array metadata to determine the size)

- To transfer less than the full array, the array section syntax can be used
  - **array_name[begin:end]**
  - This follows Python/NumPy slicing syntax where **begin** is inclusive but **end** is exclusive.

    A[N:M]. In set notation implies elements [N:M)
  - Multi-dimensional arrays work as expected when transferred in full. Currently PyOmp doesn't support array-section syntax for multi-dimensional arrays.

C Difference: In C, arrays are usually dynamically allocated and referenced through a pointer. You must use array-section syntax to move data. In C, array-syntax is "(initial-offset: number-of-items)". Fortran uses "begin:end" syntax (as Python does), but the ending index is inclusive (i.e., [begin:end]).

Controlling data movement: the map clause

- **map(to:list)**: On entering the region, variables in the list are initialized on the device using the original values from the host (host to device copy).
- **map(from:list)**: At the end of the target region, the values from variables in the list are copied into the original variables on the host (device to host copy). On entering the region, the initial value of the variables on the device is not initialized.
- **map(tofrom:list)**: the effect of both a map-to and a map-from (host to device copy at start of region, device to host copy at end).
- **map(alloc:list)**: On entering the region, data is allocated and uninitialized on the device.
- **map(list)**: equivalent to **map(tofrom:list)**.

Note: Data movement is defined from the perspective of the host.

```
@njit
def main():
    a = numpy.ones(N)
    b = numpy.ones(N)
    c = numpy.empty(N)
    with openmp ("target teams loop map(to: a,b) map(tofrom: c)"):
        for i in range(N):
                c[i] = a[i] + b[i]
```

When applied to an array, the mapping mode applies only to the array's data. Array metadata is always transferred as **to** and no operations which would change the metadata (e.g., resize) are permitted.

Commonly used clauses on target and loop constructs

- The basic construct* is:

  with openmp ("target *[clause[[,]clause]...]***"):**

    with openmp ("loop *[clause[[,]clause]...]***"):**

      for-loops

- The most commonly used clauses are:
  - **map(to | from | tofrom list)** ← default is tofrom
  - **private(***list***) firstprivate(***list***) lastprivate(***list***) shared(***list***)**
    - behave as data environment clauses in the rest of OpenMP, but note values are only created or copied into the region, not back out “at the end”.
  - **reduction(***reduction-identifier* **:** *list***)**
    - behaves as in the rest of OpenMP
  - **collapse(***n***)**
    - Combines loops before the distribute directive splits up the iterations between teams

Going beyond simple vector addition …

Using OpenMP for GPU application programming … the heat diffusion problem

5-point stencil: the heat program

- The heat equation models changes in temperature over time.

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

- We'll solve this numerically on a computer using an explicit **finite difference** discretisation.
- $u = u(t, x, y)$ is a function of space and time.
- Partial differentials are approximated using diamond difference formulae:

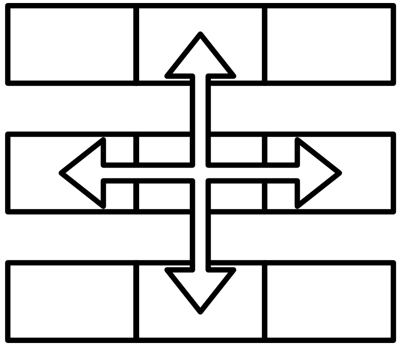
$$\frac{\partial u}{\partial t} \approx \frac{u(t+1, x, y) - u(t, x, y)}{dt}$$

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

  – Forward finite difference in time, central finite difference in space.

5-point stencil: the heat program

- Given an initial value of u, and any boundary conditions, we can calculate the value of u at time t+1 given the value at time t.
- Each update requires values from the north, south, east and west neighbours only:



- Computation is essentially a weighted average of each cell and its neighbouring cells.
- If on a boundary, look up a boundary condition instead.

How do we know the answer is correct?
The Method of Manufactured Solution

- Stencil codes are notoriously difficult to **know** if the answer is “correct”.

- Analytic solutions hard to come by:
  - It’s why you’re using a computer to solve the equation approximately after all!

- Method of Manufactured Solution (MMS) is a way to help determine if the code does the correct thing.

- An approach often used to find errors in CFD codes and check convergence properties.

Method of Manufactured Solution

- **Choose** a function for $u(t, x, y)$, substitute into the equation and work through the algebra.

- Its easier if the differential equation evaluates to zero so we don’t need to consider a right-hand side to the equation.

- $u(0, x, y)$ gives the initial conditions.

- Can evaluate boundary conditions, e.g. bottom boundary $u(0,0, y)$

- Because u is **known** for all timesteps (it was chosen!), the exact solution is **known**.

- Compare the **computed** solution to the known u to compute an error.

- Any differences come from approximations in the method, or a bug in your code.

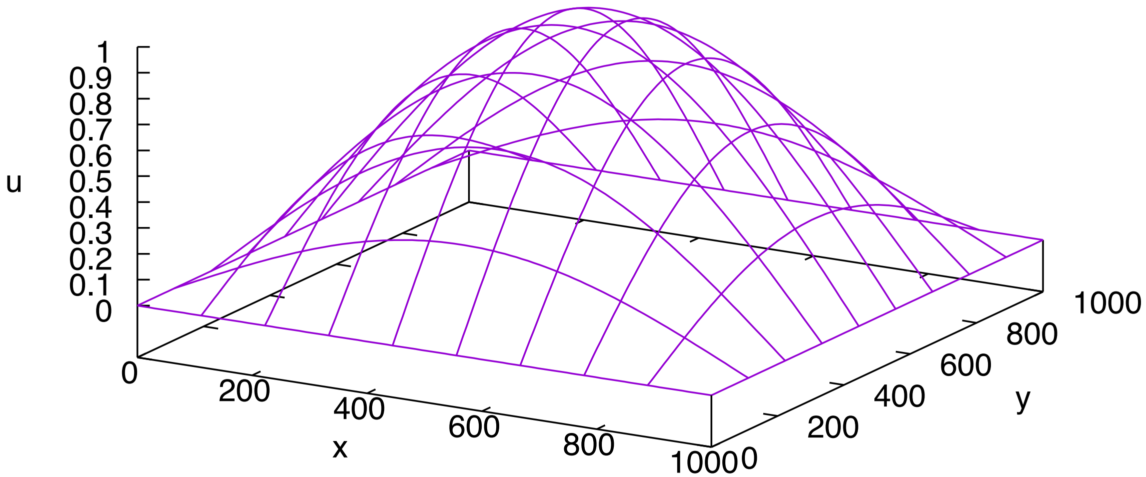
Method of Manufactured Solution

- For the problem of length l, choose u:

$$u(t,x,y) = e^{\frac{-2\alpha\pi^2 t}{l^2}} \sin\frac{\pi x}{l} \sin\frac{\pi y}{l}$$

- Boundary conditions: u is always zero on the boundaries

- Initial value of grid is then $u(0,x,y) = \sin\frac{\pi x}{l} \sin\frac{\pi y}{l}$



Heat diffusion problem …

```
# 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
```

Array-swap on the host works. Why?

u and u_tmp are references to structs that hold NumPy metadata and a data pointer.

The OpenMP runtime creates a device struct at the target enter data construct and maintains a fixed association between host and device struct references.

Hence, as you swap the array variables, the references to the struct addresses in device memory are swapped.

Heat diffusion problem …

```
# 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
```

- Our program takes two optional command line arguments: <ncells> <nsteps>
  - E.g. ./heat 1000 10
  - 1000x1000 cells, 10 timesteps (the default problem size).

- If no command line arguments are provided, it uses a default:
  - These two commands both run the default problem size of 1000x1000 cells, 10 timesteps.
  - ./heat
  - ./heat 1000 10

- A sensible bigger problem is 8000 x 8000 cells and 10 timesteps.

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…

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.

Target data directive

- The **target data** construct creates a target data region … use **map** clauses for explicit data management

Data is mapped onto the device at the beginning of the construct

```
with openmp ("target data map(to: A, B) map(from: C)"):

    with openmp ("target"):
       {do lots of stuff with A, B and C}

    {do something on the host}

    with openmp ("target"):
       {do lots of stuff with A, B and C}
```

one or more **target regions** work within the **target data region**

Data is mapped back to the host at the end of the target data region

Target enter/exit data constructs

- The **target data** construct requires a *structured* block of code.
  - Often inconvenient in real codes.

- Can achieve similar behavior with two standalone directives:
  with openmp ("target enter data map(…"):
  with openmp ("target exit data map(…"):

- The **target enter data** maps variables to the device data environment.
- The **target exit data** unmaps variables from the device data environment.
- Future **target** regions inherit the existing data environment.

Target enter/exit data example

```
@njit
def main():
    N = 1024
    A = numpy.arange(N)

    with openmp ("target enter data map(to: A)"):
        pass

    with openmp ("target teams loop"):
        for i in range(N):
            A[i] = A[i] * A[i]

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

pass is a python keyword indicating an empty block of code.

Target enter/exit data details

- **with openmp ("target enter data clause[[[,]clause]...]"):**

- Creates a target task to handle data movement between the host and a device.

- clause is one of the following:
  - if(scalar-expression)
  - device(integer-expression)
  - map (map-type: list)

Solution: Reference swapping in action

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

Copy data to device before iteration loop

```
for _ in range(nsteps):

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

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

```
    # 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 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

Target update directive

- You can update data between target regions with the **target update** directive.

Set up the data region ahead of time.

```
with openmp ("target data map(to: A, B) map(from: C)"):

    with openmp ("target"):
        {do lots of stuff with A, B and C}

    with openmp ("target update from(A)"):
       {do something on the host}

    with openmp ("target update to(A)"):
       pass

    with openmp ("target"):
        {do lots of stuff with A, B and C}
```

map A on the device to A on the host.

map A on the host to A on the device. Note: openmp context body cannot be empty so use “pass”

Note: update directive has the transfer direction as the clause: e.g. update to(…)
Compare to map clause with direction inside: map(to: …)

Data movement summary

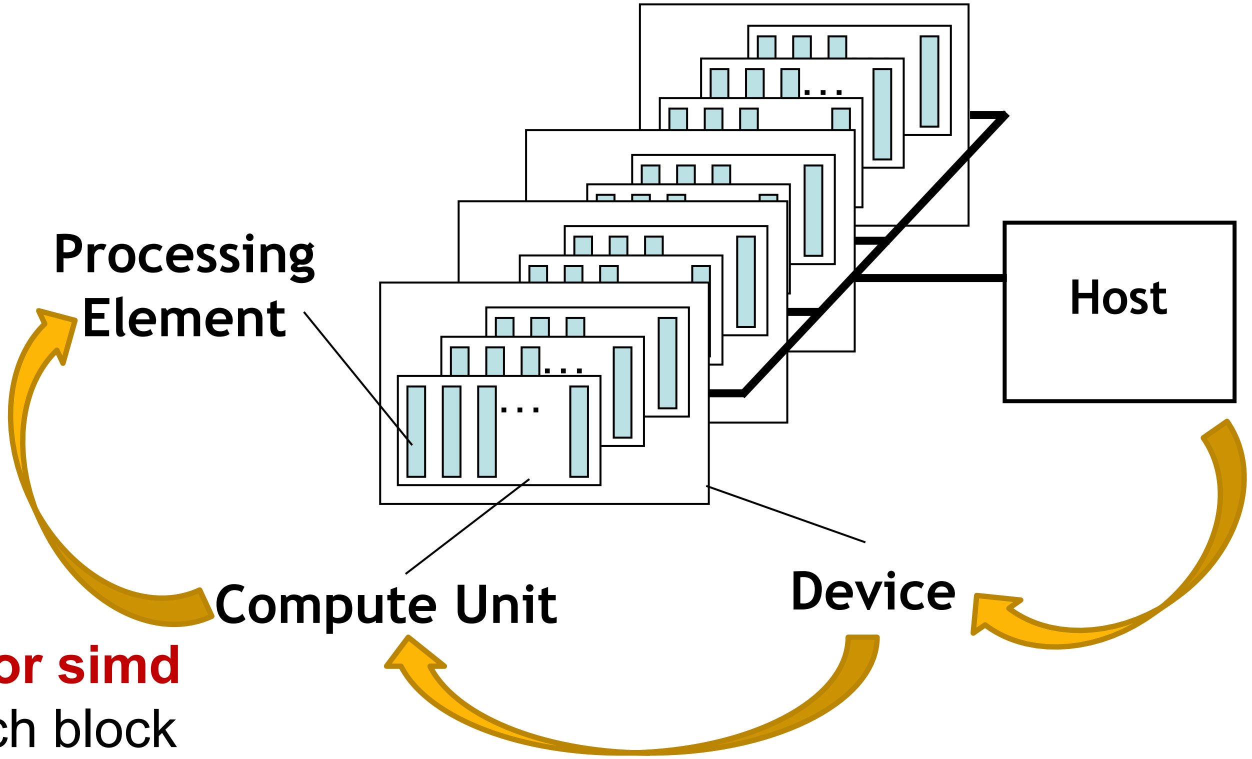
- Data transfers between host/device occur at:
  - Beginning and end of **target** region
  - Beginning and end of **target data** region
  - At the **target enter data** construct
  - At the **target exit data** construct
  - At the **target update** construct

- Can use **target data** and **target enter/exit data** to reduce redundant transfers.

- Use the **target update** construct to transfer data on the fly within a **target data** region or between **target enter/exit data** directives.

The loop construct is great, but sometimes you want more control.

Our host/device Platform Model and OpenMP



Target construct to get onto a device

Parallel for simd to run each block of loop iterations on the processing elements

Teams construct to create a league of teams with one team of threads on each compute unit.

Distribute construct to assign blocks of loop iterations to teams.

teams and distribute constructs

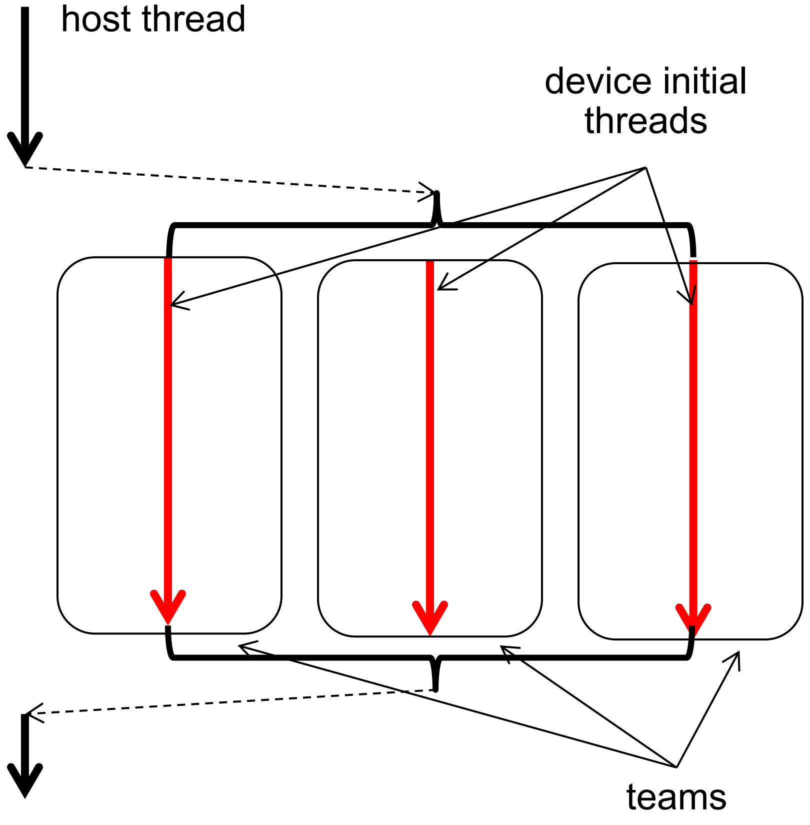
- The **teams** construct
  - Similar to the **parallel** construct
  - It starts a league of thread teams
  - Each team in the league starts as one initial thread – a team of one
  - Threads in different teams cannot synchronize with each other
  - The construct must be “perfectly” nested in a **target** construct

- The **distribute** construct
  - Similar to the **for** construct
  - Loop iterations are workshared across the initial threads in a league
  - No implicit barrier at the end of the construct
  - **dist_schedule(*kind[, chunk_size]*)**
    - If specified, scheduling kind must be static
    - Chunks are distributed in round-robin fashion in chunks of size ***chunk_size***
    - If no chunk size specified, chunks are of (almost) equal size; each team receives at least one chunk

Create a league of teams and distribute a loop among them

- teams construct
- distribute construct

```
with openmp ("target"):
  with openmp ("teams"):
    with openmp ("distribute"):
      for i in range(N):
        …
```

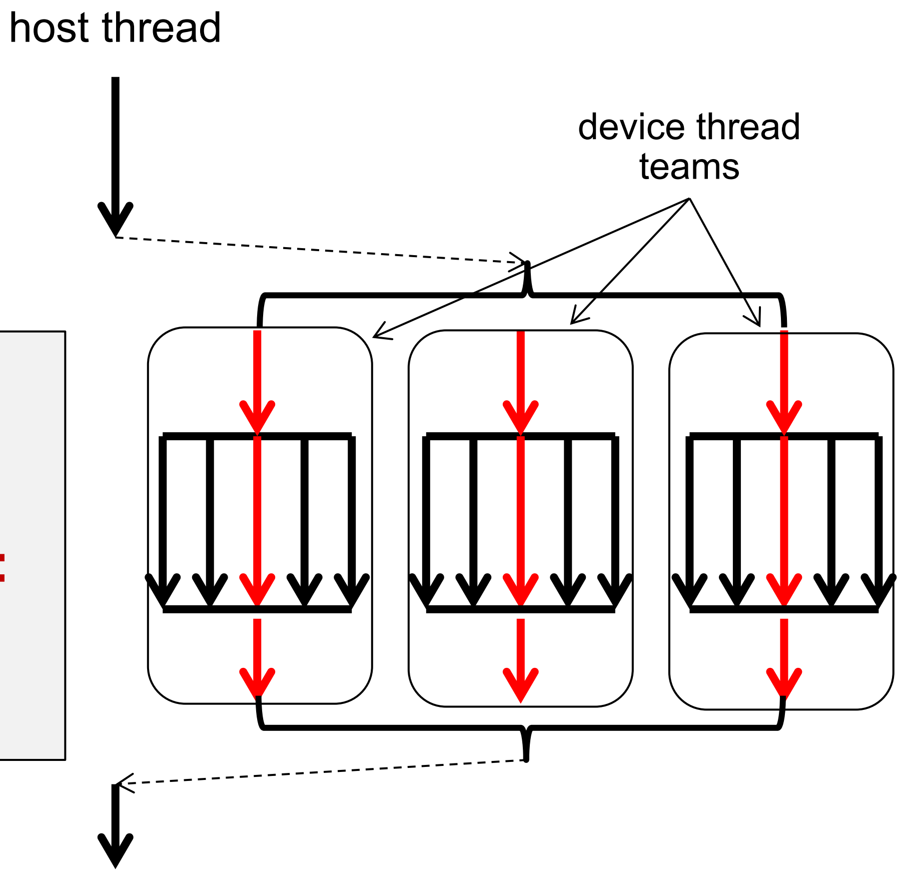


- Transfer execution control to MULTIPLE device initial threads
- Workshare loop iterations across the initial threads.

Create a league of teams and distribute a loop among them and run each team in parallel with its partition of the loop

- teams distribute
- parallel for

```
with openmp ("target"):
  with openmp ("teams"):
    with openmp ("distribute"):
      with openmp ("parallel for"):
        for i in range(N):
          …
```

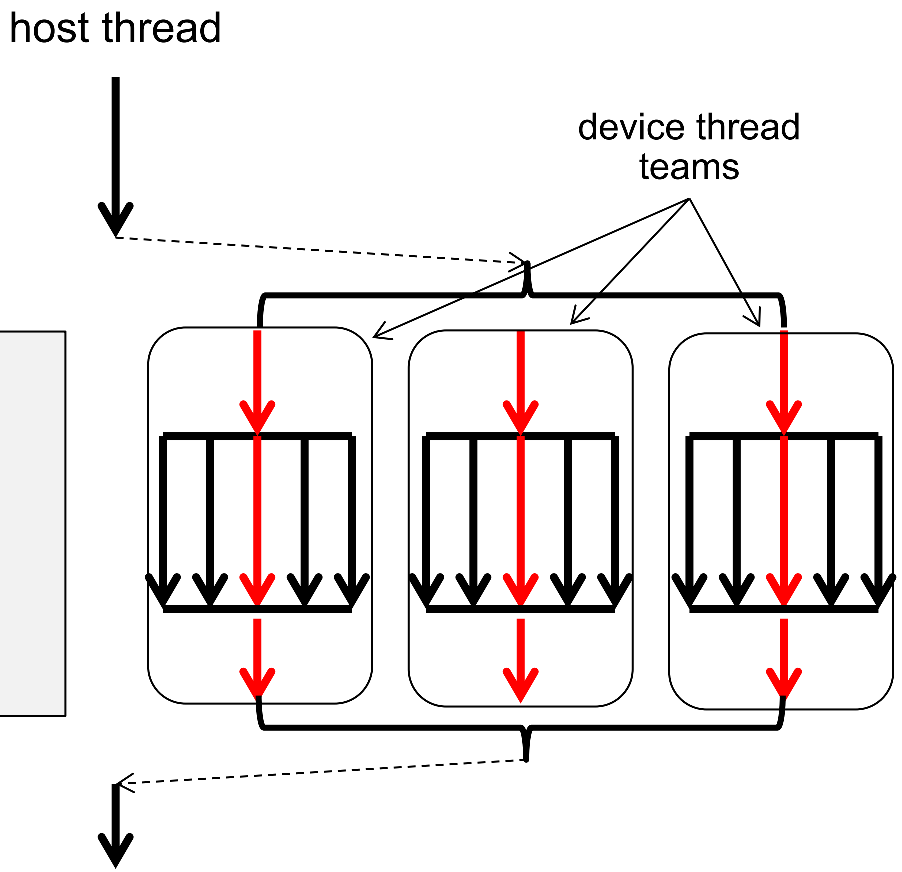


- Transfer execution control to MULTIPLE device initial threads
  - Workshare loop iterations across the initial threads (teams distribute)
- Each initial thread becomes the primary* thread in a thread team
  - Workshare loop iterations across the threads in a team (parallel for)

Create a league of teams and distribute a loop among them and run each team in parallel with its partition of the loop

- loop

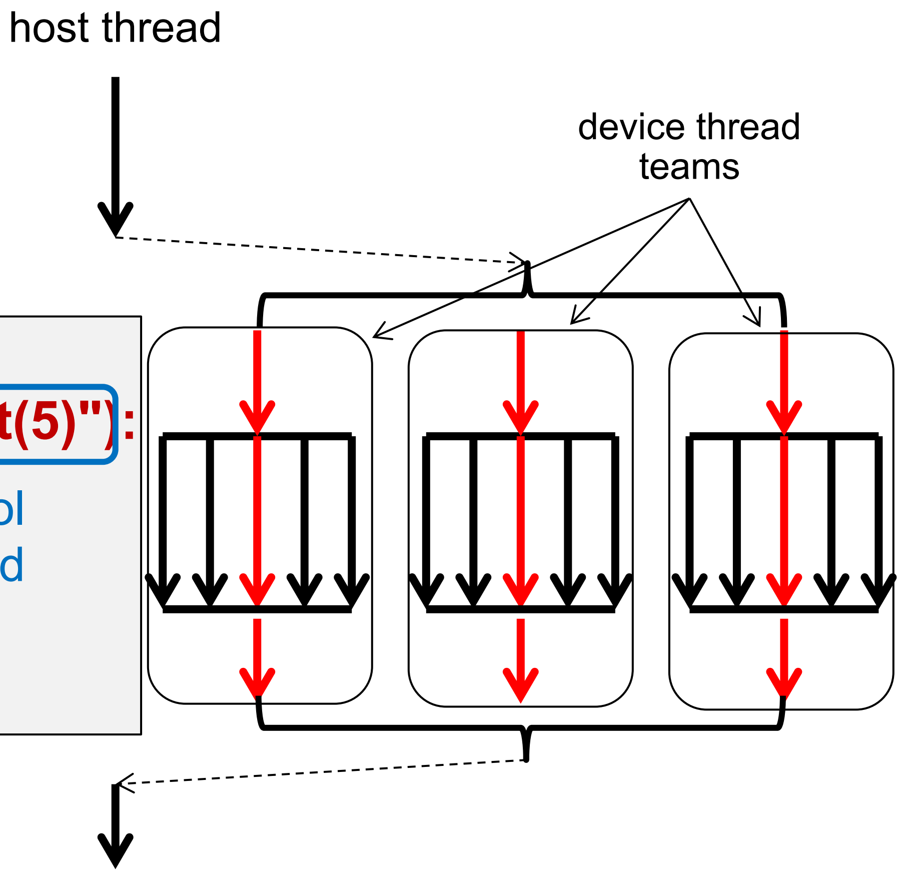
```
with openmp ("target"):
  with openmp ("teams"):
    with openmp ("loop"):
      for i in range(N):
        ...
```



- Transfer execution control to MULTIPLE device initial threads
  - Workshare loop iterations across the initial threads (teams distribute)
- Each initial thread becomes the primary* thread in a thread team
  - Workshare loop iterations across the threads in a team (parallel for)

Create a league of teams and distribute a loop among them and run each team in parallel with its partition of the loop

- teams distribute
- parallel for



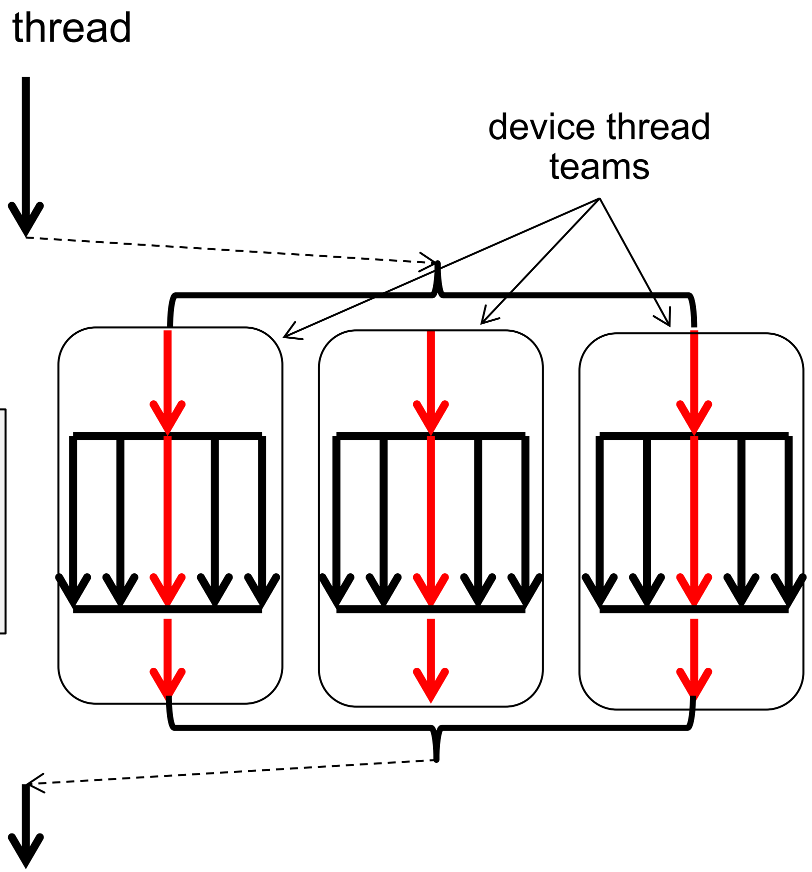
```
with openmp ("target"):
 with openmp ("teams num_teams(3) thread_limit(5)"):
  with openmp ("distribute"):
   with openmp ("parallel for"):
    for i in range(N):
     …
```

Explicit control of number and size of teams

- Transfer execution control to MULTIPLE device initial threads
  - Workshare loop iterations across the initial threads (teams distribute)
- Each initial thread becomes the primary* thread in a thread team
  - Workshare loop iterations across the threads in a team (parallel for)

Create a league of teams and distribute a loop among them and run each team in parallel with its partition of the loop

- Combined construct



```
with openmp ("target teams loop"):
  for i in range(N):
    …
```

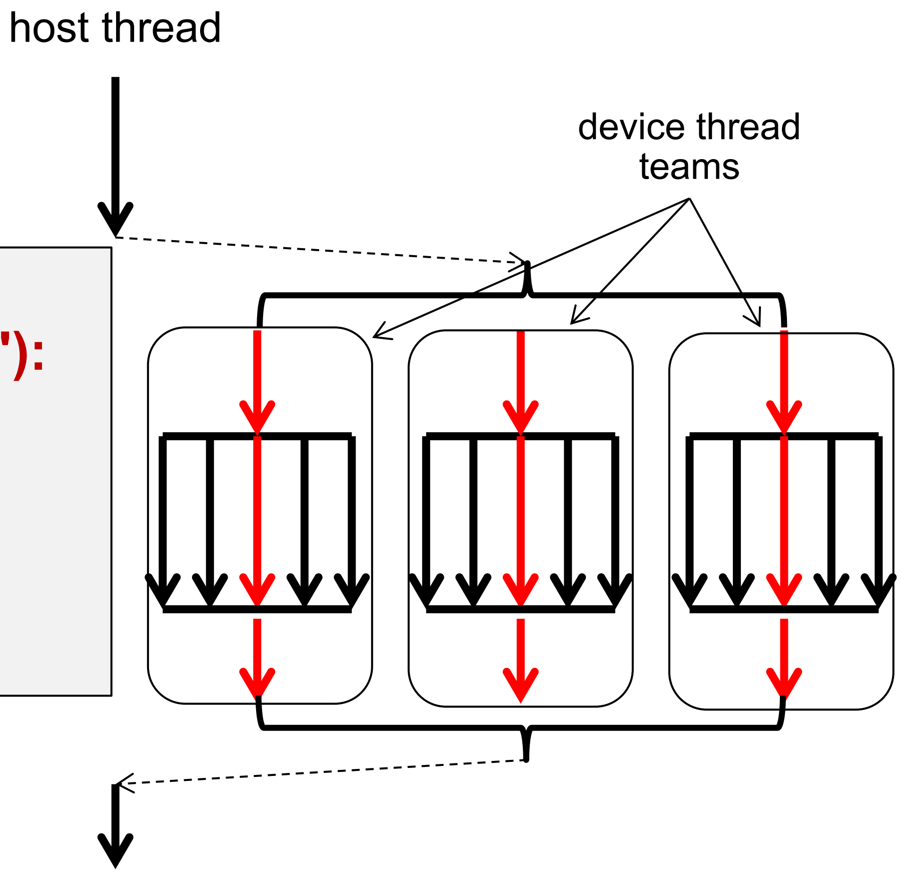
- Transfer execution control to MULTIPLE device initial threads
  - Workshare loop iterations across the initial threads (teams distribute)
- Each initial thread becomes the primary* thread in a thread team
  - Workshare loop iterations across the threads in a team (parallel for)

Create a league of teams and distribute a loop among them and run each team in parallel with its partition of the loop

- teams distribute
- parallel for

Works with nested loops as well

```
with openmp ("target"):
  with openmp ("teams distribute"):
    for i in range(N):
     with openmp ("parallel for"):
       for j in range(M):
        ...
```



- Transfer execution control to MULTIPLE device initial threads
  - Workshare loop iterations across the initial threads (teams distribute)
- Each initial thread becomes the primary* thread in a thread team
  - Workshare loop iterations across the threads in a team (parallel for)

There is MUCH more … beyond what have time to cover

- Do as much as you can with a simple loop construct. It’s portable and as compilers improve over time, it will keep up with compiler driven performance improvements.
- But sometimes you need more:
  - Control over number of teams in a league and the size of the teams
  - Explicit scheduling of loop iterations onto the the teams
  - Management of data movement across the memory hierarchy: global vs. shared vs. private …
  - Calling optimized math libraries
  - Multi-device programming
  - Asynchrony
- Ultimately, you may need to master all those advanced features of GPU programming. But start with loop. Start with how data on the host maps onto the device (i.e. the GPU). Master that level of GPU programming before worrying about the complex stuff.

Outline

- Introducing parallel computing and PyOMP
- The PyOMP system
- PyOMP and multithreading (parallelism for the CPU)

Break

- GPU programming with PyOMP
- ➡ Other approaches to parallelism in Python.
- Wrap-up and Q&A

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

PyOMP is great … but it is a research system still under development.

Let’s talk about parallel programming models and ask the question … what are the key mainstream programming models in Python

But lets first look at programming models from the early days of parallel computing.

Parallel programming environments in the 90's

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
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

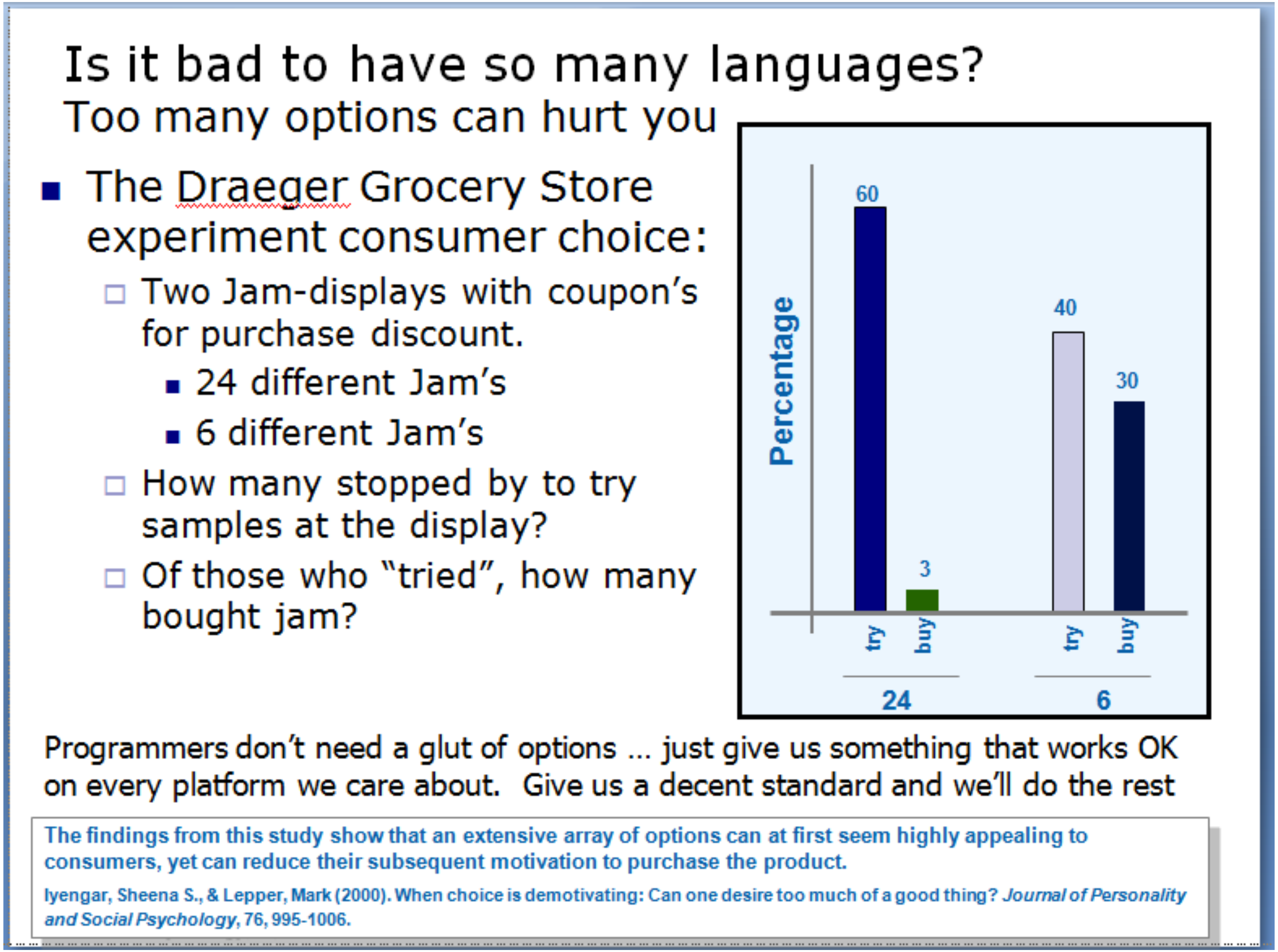
Parallel-C++
Parallaxis
ParC
ParLib++
ParLin
Parmacs
Parti
pC
pC++
PCN
PCP:
PH
PEACE
PCU
PET
PETSc
PENNY
Phosphorus
POET.
Polaris
POOMA
POOL-T
PRESTO
P-RIO
Prospero
Proteus

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

Third party names are the property of their owners.

A warning I've been making for the last 20 years



Is it bad to have so many languages?

Too many options can hurt you

- The Draeger Grocery Store experiment consumer choice:
  - □ Two Jam-displays with coupon's for purchase discount.
    - 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?

Programmers don't need a glut of options … just give us something that works OK on every platform we care about. Give us a decent standard and we'll do the rest

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.

Parallel programming environments: post-90s

- The application community (with leadership from the Accelerated Strategic Computing Initiative) pushed for convergence around a small number of programming languages:
  - For clusters and massively parallel computers: MPI
  - For shared memory systems: OpenMP

- With only two languages, vendors could focus on engineering high quality solutions … rather than chasing the latest fad.

- All was good until ~2006 when fully programmable GPUs came along. We are still sorting out what will become the converged solution …
  - Cuda, Sycl, OpenACC, OpenMP ← hopefully the open standard Sycl will win, but its too early to say

How about Parallel programming with Python

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
mi4py
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

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

How about Parallel programming with Python

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
PyLinda
puMPI
pypar
multiprocessing
PyOpenCL

pyPastSet
pypvm
pynpvm
Pyro
Ray
Rthread
ng.MasterSlave
torcpy
StarCluster
dpctl
arkouda
PyOMP
dpnp

We are still early (compared to HPC) in the evolution of parallel programming models for Python.

Hopefully, soon the python application community will come together and help us narrow down to a handful of systems to focus on.

That would allow vendors to carry out HW/SW optimization and focus on quality over ”chasing fads”.

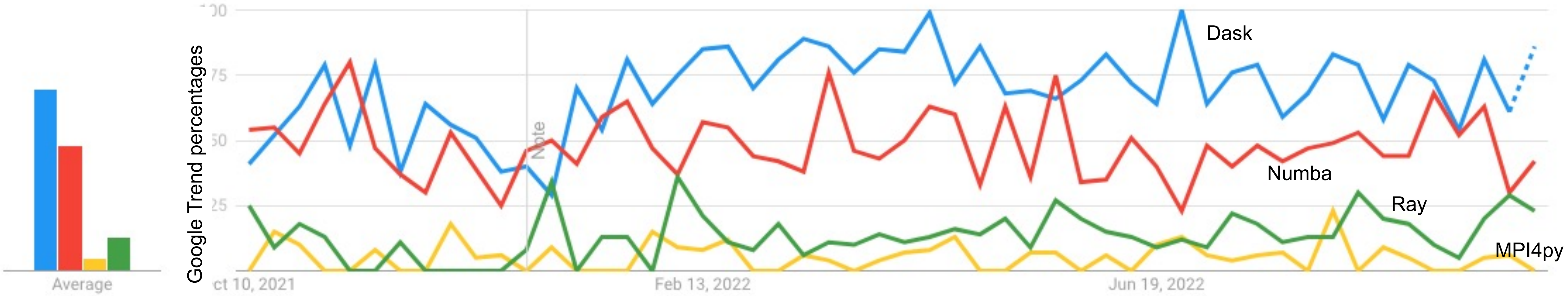
Building on the list at https://wiki.python.org/moin/ParallelProcessing

Popular python parallel Programming models

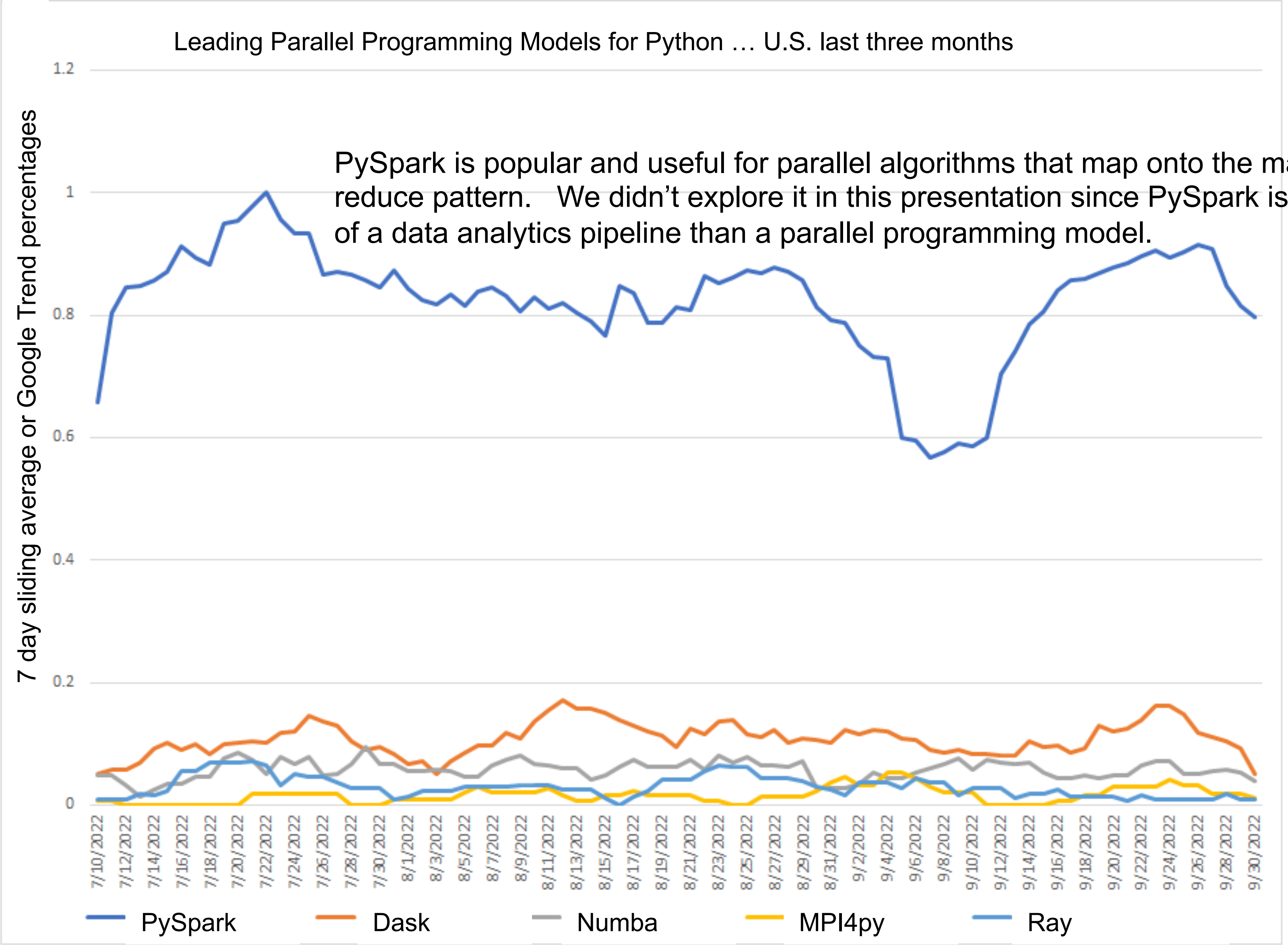
We compared many python parallel programming models with google-trends (which tracks web searches)

These four systems are popular and (in our opinion) are the key systems to consider

Key Parallel Programming Models for Python … U.S. last 12 months



Popular python parallel Programming models



Leading Parallel Programming Models for Python … U.S. last three months

PySpark is popular and useful for parallel algorithms that map onto the map reduce pattern.   We didn't explore it in this presentation since PySpark is more of a data analytics pipeline than a parallel programming model.

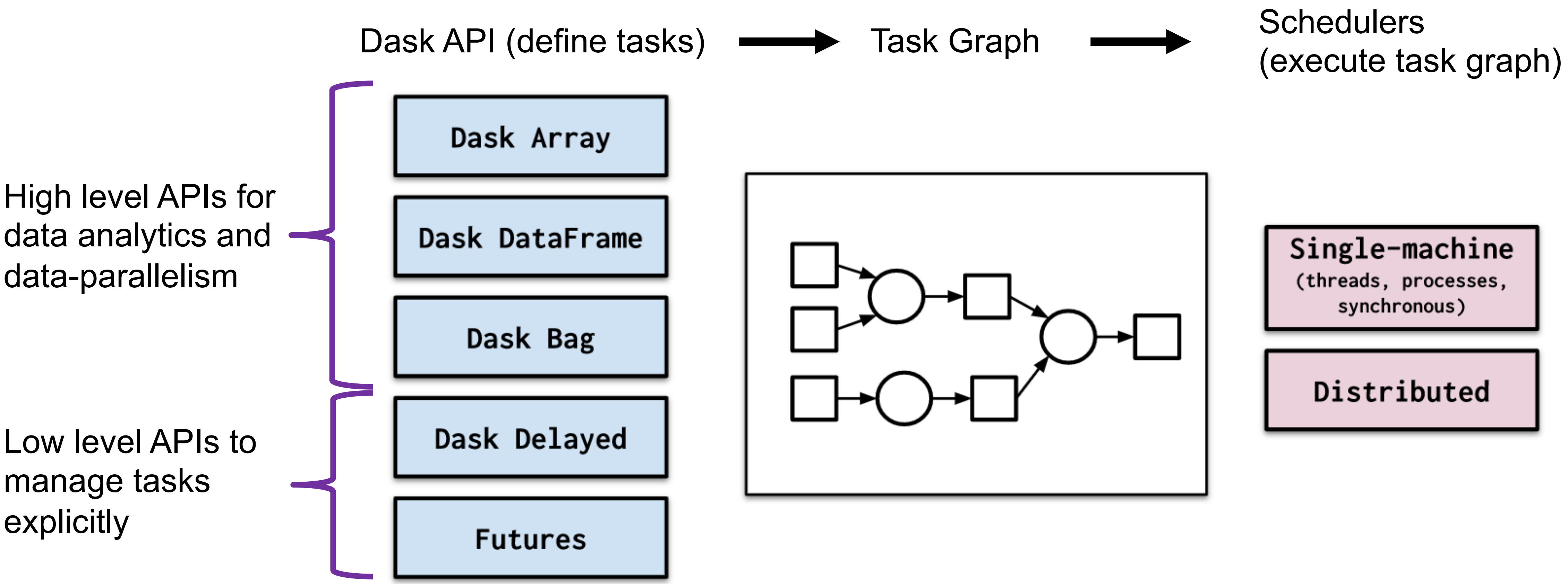
We compared many python parallel programming models with google-trends (which tracks web searches)

Our best guess … these are the top five)

Dask

DASK

- Parallel and distributed computing library for Python
- Client / driver submits tasks to Dask cluster (set of worker processes on one or more physical nodes)



Dask API (define tasks) → Task Graph → Schedulers (execute task graph)

High level APIs for data analytics and data-parallelism

- Dask Array
- Dask DataFrame
- Dask Bag

Low level APIs to manage tasks explicitly

- Dask Delayed
- Futures

Single-machine (threads, processes, synchronous)

Distributed

Source: https://docs.dask.org/en/latest/

Dask Delayed – lazy, remote functions

- Define a remote function:

```
@dask.delayed
def add_one(i):
      time.sleep(1)
      return i+1
```

Decorator turns normal
Python function into Dask
lazy function

- Calling remote function, getting results:

```
futurevalue = add_one(7)
v = futurevalue.compute()
```

Returns immediately after
creating task in task graph

Triggers execution of task graph
Returns value 8 after about
1 second when task completes

Dask – parallel and chaining calls

- Parallel execution:

```
fv = [add_one(i) for i in range(5)]
v = sum(fv)
v = v.compute()
```

Returns immediately with a list of “futures”

Standard Python sum function; Returns immediately with a future

Returns value 15 after about 1 second

- Chained execution:

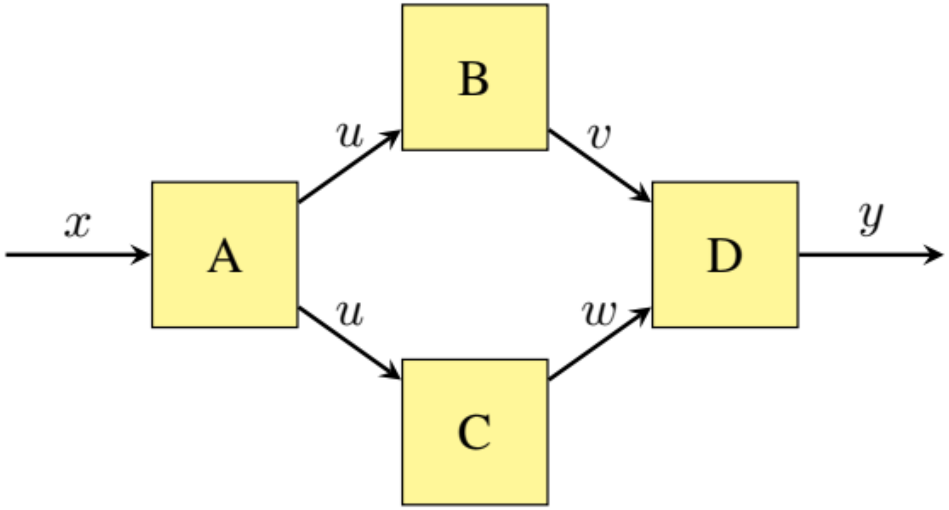
```
v = 2
for x in range(5):
      v = add_one(v)
v = v.compute()
```

These return immediately

Returns value 7 after about 5 seconds

Chaining forms DAGs of Tasks

```
# A, B, C, and D are delayed  functions
u = A(x)
v = B(u)
w = C(u)
y = D(v, w)
y = y.compute()
```



Dask Futures

- Same concept, but eager asynchronous execution, different syntax

```
 def add_one(i):
      time.sleep(1)
      return i+1

future = client.submit(add_one, 3)
result = future.result()
```

Submits add_one(3) for distributed execution

Returns value 4 in about 1 second

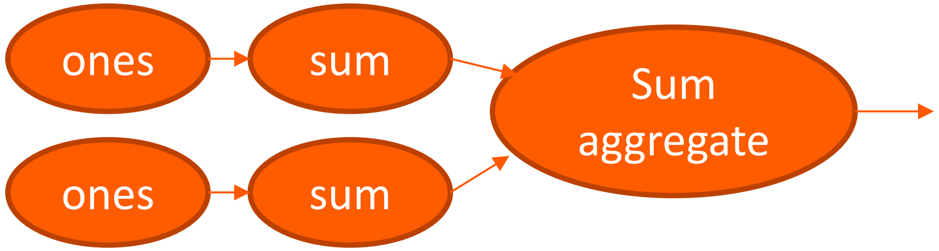
- Note: no decorator, explicit job submission
- Can pass futures as parameters to chain functions/construct DAGs

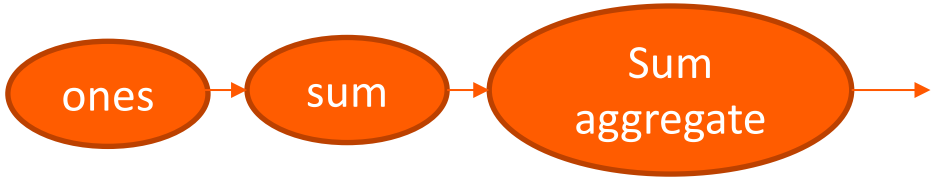
Dask Array

- High-level API provides distributed, Numpy-like array interface
- Arrays partitioned into chunks – serves as unit of storage and computation
- Arrays can be disk-backed, and thus larger than memory
- Array operations are lazy, internally constructing DAG of operations
- Explicit triggering of execution using compute() method
  - Parallel execution of relevant portions of task graph on Dask cluster
  - Computation at chunk granularity
  - Only necessary chunks computed for requested result

Dask Array example

- A = da.ones((1000,1000),chunks=(1000,500))
  - Constructs 1000x1000 array, with two chunks of size 1000x500
- B = da.sum(A, axis=0)
  - Sum along axis 0 → should produce a 1000 element array
- B.compute()
  - Triggers computation of DAG:
  - Parallel execution on chunks
- B[0].compute()
  - Only compute chunks needed for B[0]
- Typically, Dask will not materialize a derived array
  - Keeps the DAG that describes how to compute it
  - May need to recompute (but may cache results as well)
  - Optimized for computations on disk-based data that won't fit in memory
- Persist() method to force computation, materialization of an array





Multi-tasking, Pi program with Dask

```
import numpy as np
import dask

@dask.delayed
def calc_pi(nstart, nstop, step):
    start = (nstart+0.5)*step
    stop = (nstop-0.5)*step
    nsteps = nstop-nstart
    X = np.linspace(start, stop, num=nsteps)
    Y = 4.0 / (1.0 + X*X)
    return np.sum(Y)

def piFunc(NumSteps, NumTasks):
    step = 1.0/NumSteps
    s = 0
    for i in range(NumTasks):
        nstart = (i*NumSteps)//NumTasks
        nstop = ((i+1)*NumSteps)//NumTasks
        s = s + calc_pi(nstart, nstop, step)
    s = s.compute()
    return step*s

if __name__=="__main__":
    from dask.distributed import Client
    client = Client()
    pi = piFunc(100000000, 100)
```

Calculate over part of the range;
Written in Numpy vector style
Faster than Python loops, but use
memory for the arrays X, Y, temps

Start NumTasks tasks,
construct DAG of operations
computing sum

Trigger execution, wait for
completion, get result

Initialize dask “cluster” on local
machine; can provide address
to connect to remote cluster

Numba with ParallelAccelerator

Numba … C-like performance from Python code

- Numba is a JIT compiler. Maps a subset of python with numpy arrays onto LLVM
- Once code is JIT'ed into LLVM, all performance enhancements exposed at the level of LLVM are directly available … result is performance that approaches that from raw C or Fortran
- Source code is pure python for maximum portability

- Just add the @jit decorator to enable numba for a function.

```
from numba import jit

@jit
def addit(A,B):
    return (A+B)
```

Numba jit comiler applied the first time a function is encountered. Caches the code so subsequent calls to the function don't run the jit step.

Numba defines elementwise functions called *ufuncs*

This generates the LLVM code and calls the addition ufunc to do an elementwise add of A and B

- Numerous options in numba … we are barely scratching the surface
  - @jit (nopython = true) tells the system to NOT use any python objects in the generated code. Can be much faster
  - @jit(parallel = true) invoke parallel accelerator

Numba with ParallelAccelerator

Numba
ParallelAccelerator

- ParallelAccelerator has been Available in Numba since 2017.
- Let's users parallelize their code with a one-line change, namely annotating their Numba "jit" decorator with "parallel=True"
- Identifies operations in the code with concurrent semantics and executes them in parallel, making full use of modern multi-core CPUs.
- Allows operations to be fused together and to eliminate temporaries which results in improved cache utilization.
- Works for vector-style codes as well as explicitly parallel loops annotated with the prange keyword.

ParallelAccelerator

- Accelerates execution of Python applications by auto-parallelizing and optimizing numeric operations
- Brings performance without rewriting code in “performance languages”

```
@numba.jit(nopython=True, parallel=True)
def logistic_regression(Y, X, w, iter):
    for i in range(iter):
        w -= np.dot(((1.0 / (1.0 + np.exp(-Y * np.dot(X, w))) - 1.0) * Y), X)
    return w
```

1 line change
6x better
performance

Y, X, and w are numpy arrays. Elementwise operations and dot products are transparently mapped onto threads for parallel execution.

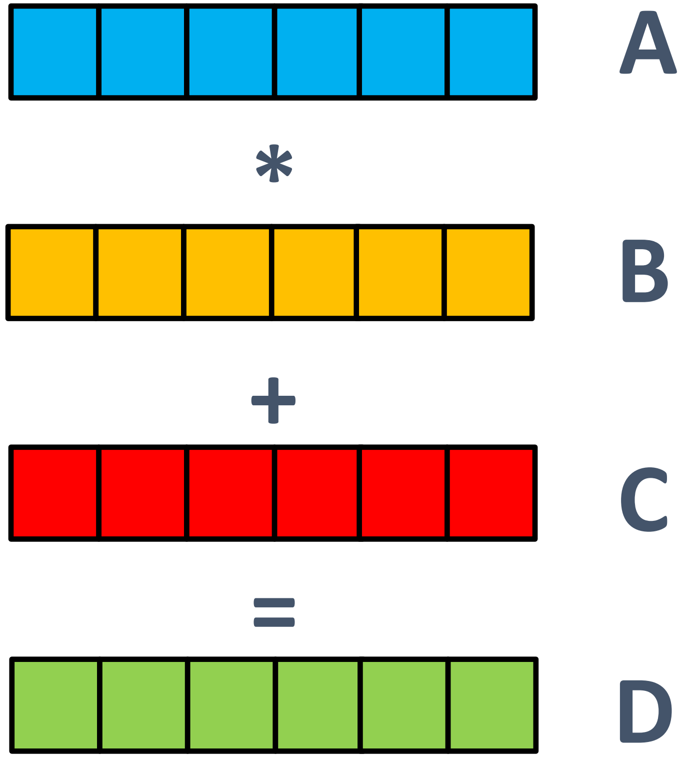
The Data Parallelism design pattern … the parallelism is expressed through the data .. Typically as functions applied independently to the elements of data structures combined with collective ops (such as dot products).

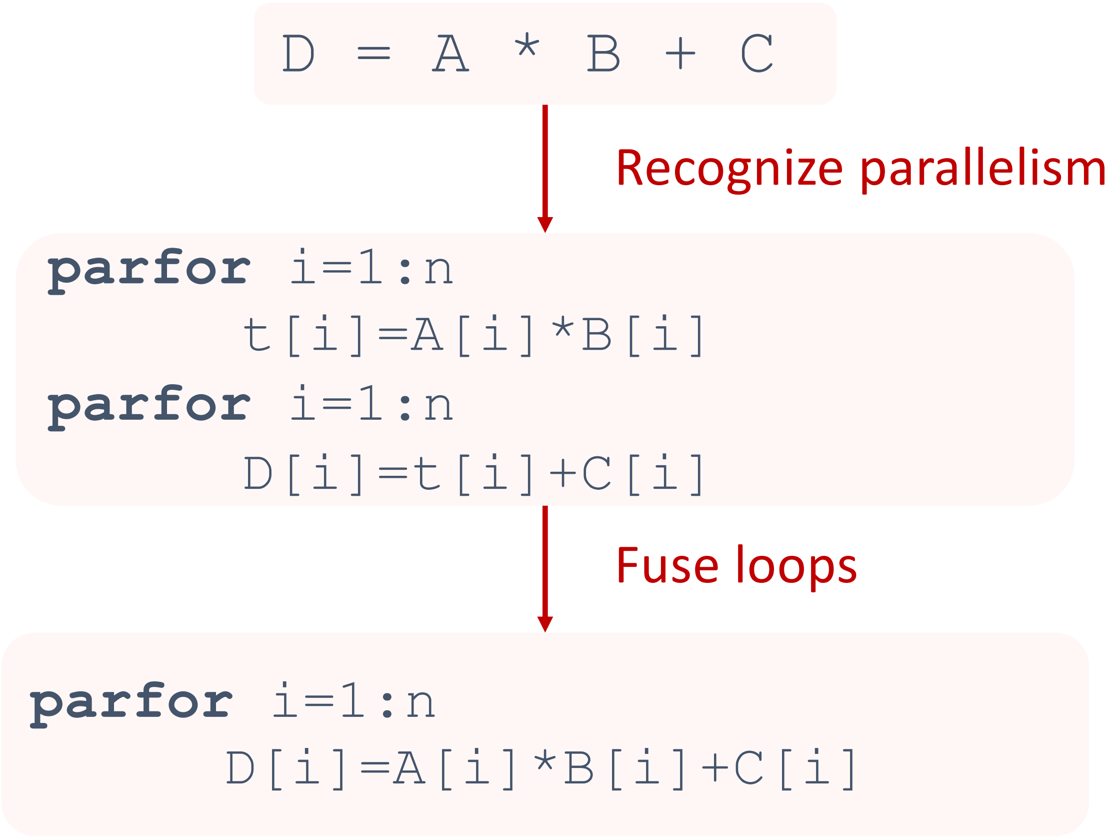
Parallel Accelerator

Works with numba to JIT code that executes in parallel. It does the following:

1. Recognize parallelism.
   - Pattern recognition of operations with concurrent semantics.
2. Represent parallelism.
   - Numba’s parfor node – represents a strictly nested set of for loops known to have no cross-iteration dependencies.
3. Optimizations.
   - Fusion – combine compatible parfors together. Eliminates unnecessary temporary arrays and traverses arrays only once for better cache utilization.
4. Run in parallel.
   - Improves performance by leveraging multiple cores and vector instructions.

Transformation carried out for array-based data parallelism





```
D = A * B + C
```

Recognize parallelism

```
parfor i=1:n
      t[i]=A[i]*B[i]
parfor i=1:n
      D[i]=t[i]+C[i]
```

Fuse loops

```
parfor i=1:n
      D[i]=A[i]*B[i]+C[i]
```

ParallelAccelerator – Softmax program

```
import numba

@numba.njit(parallel=True)
def sigArr(A):
    Amax = np.max(A)
    Ashift = A - Amax
    expAshift = np.exp(Ashift)
    Normalization = np.sum(expAshift)
    reciNorm = 1/Normalization
    sigma = expAshift*reciNorm
    return sigma
```

- Same as the NumPy version.
- np.max executed in one parallel region.
- Subtraction, exp, and sum fused into one parallel region.
- Ashift temporary eliminated.
- expAshift * reciNorm the final parallel region.

ParallelAccelerator: loop level parallelism

The Pi program

```
import numba

@numba.njit(parallel=True)
def pi():
    num_steps = 1000000
    step = 1.0 / num_steps
    the_sum = 0.0
    for i in numba.prange(num_steps):
        x = (0.5 + i) * step
        the_sum += 4.0 / (1.0 + x * x)
    pi = step * the_sum
    return pi

print(pi())
```

- ParallelAccelerator includes parallel loops for loop-level parallelism
- The **prange** construct causes equal portions of the iteration space from 0 to num_steps distributed to each core.
- The reduction (the_sum += …) recognized and implemented safely and efficiently in parallel.

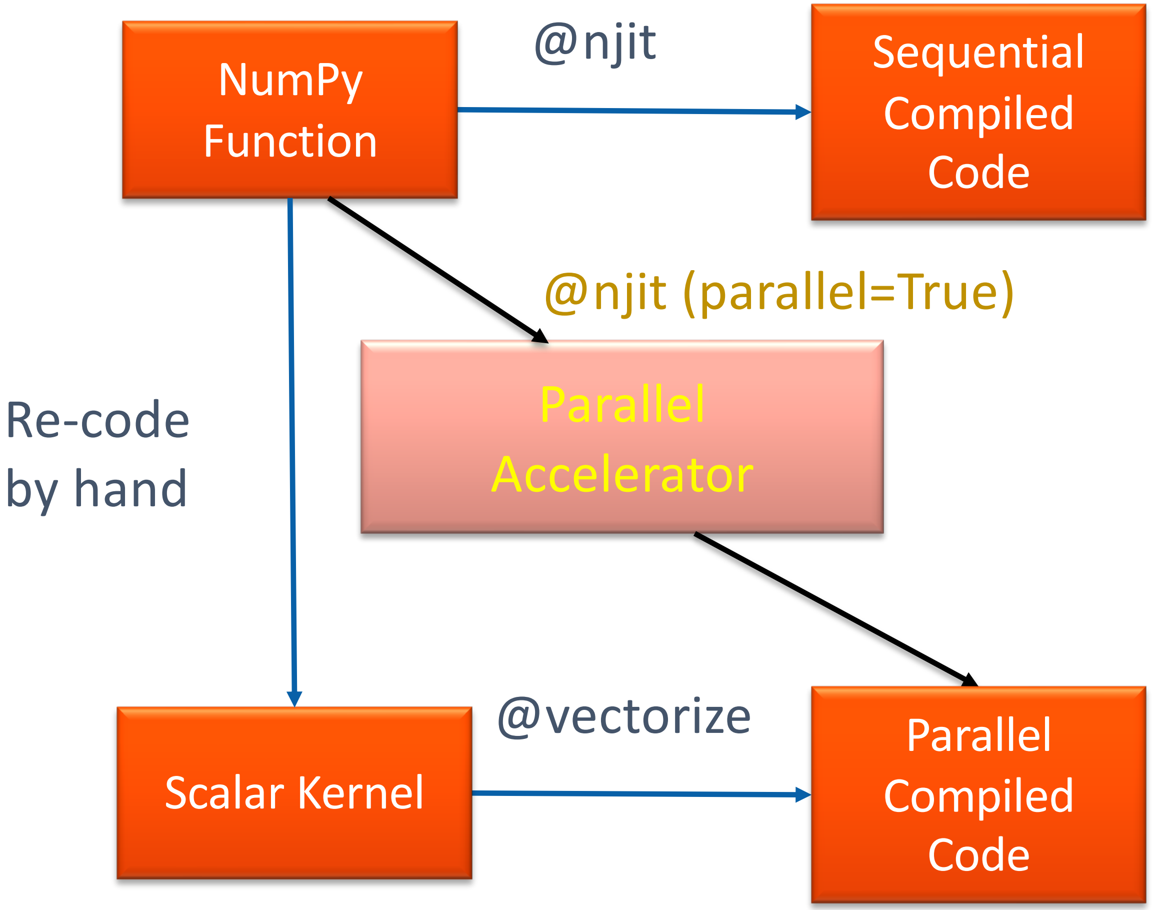
Running Parfors in Parallel

- Generate a Numba function (i.e., a generated ufunc or *gufunc*) with a loop nest corresponding to the parfor’s loop nest.
  - Adds a schedule argument that specifies which threads do which iterations.
- Add the body of the parfor inside the loop nest.
- Allocate a reduction array for each reduction (warner: scalers NOT in a reduction lead to data races).
- Initialize each thread’s reduction value from this array and write back to the array just before the end of the parallel region.
- Generate code to perform final reduction across these arrays after parallel region.
- Execute gufunc using Numba’s existing parallel execution infrastructure.
- Scheduling:
  - The default scheduler is equivalent to OpenMP static and divides multi-dimensional iteration space up into approximately equal-sized hyperrectangles, one for each available core.
  - Programmers may optionally specify a chunksize, which results in the equivalent of OpenMP dynamic scheduling behavior.

Parfor optimizations

- Array analysis
  - Called the “secret sauce” by Numba’s lead developer.
  - Tracks integers and arrays to determine when two or more arrays must have a common dimension length.
- Fusion
  - Parfors with equivalent nested loops are merged (under certain conditions).
  - Equivalence determined by array analysis.
  - Reduces looping overhead, minimizes passes over arrays (cache friendly), eliminates temporaries.
- Loop invariant code motion
  - Operations not recursively dependent on loop indices moved before the loop.
- Allocation hoisting
  - Allows allocation of space for arrays of the same size created by the loop body to be moved before the loop.
- Threads compute reductions locally and combined after the parallel region to get the final value.

How ParallelAccelerator fits into Numba



- Most of ParallelAccelerator could be done manually using Numba’s @vectorize or @guvectorize but those APIs are very difficult to use, are error prone, and time-consuming.
- ParallelAccelerator achieves this performance with a one or two line code change.

Recognizing Parallelism

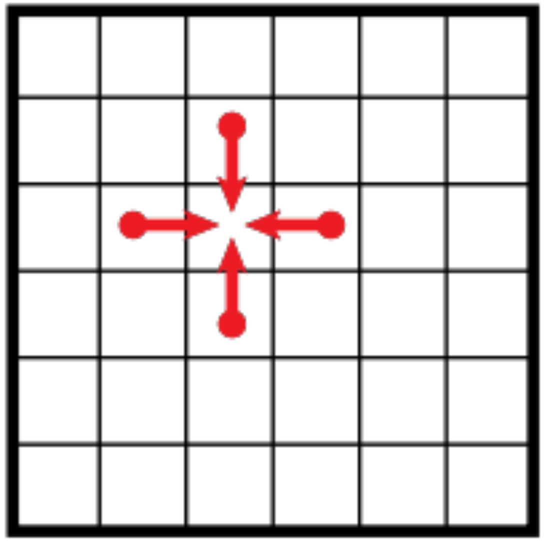
The following patterns are recognized by ParallelAccelerator for parallel execution:

1. Implicit
   - Element-wise operations: unary(+,-,~), binary(+,-,*/,//?,%,|,>>,^,&,**,//), comparison(==,!=,<,<=,>,>=), NumPy ufuncs, user-defined DUFunc.
   - NumPy reductions: sum, prod, min, max, argmin, argmax, mean, var, std.
   - Array creation: zeros, ones, arrange, linspace, and random array create for all available distributions.
   - NumPy dot: matrix/vector or vector/vector.
   - Array assignment.
   - Functools.reduce.
   - Stencil decorator.
2. Explicit
   - prange, pndindex

Other ParallelAccelerator Technology

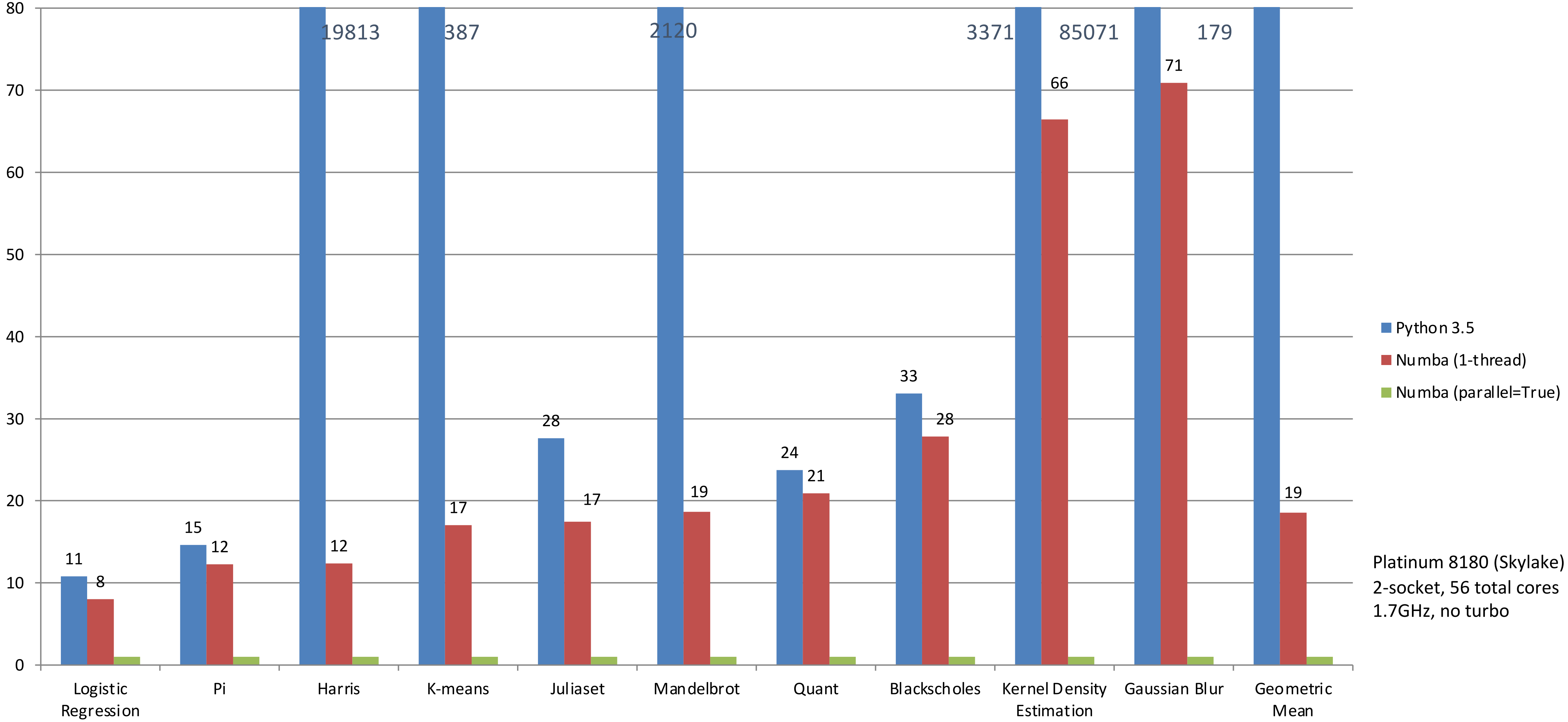
- Stencils are very common in scientific computing.
- ParallelAccelerator provides a productive stencil abstraction with automatic parallelization.

```
@stencil
def jacobi_kernel(a):
   return 0.25 * (a[0,1] + a[0,-1] + a[-1,0] + a[1,0])

@numba.njit(parallel=True)
def run_jacobi(a):
   return jacobi_kernel(a)
```



Performance

Kernel Times Relative to Numba with parallel=True (lower is better)



Platinum 8180 (Skylake)
2-socket, 56 total cores
1.7GHz, no turbo

ParallelAccelerator – Next steps

- Gradually add support for new NumPy functions or variants of existing NumPy functions supported by Numba.

- Continues to add additional code recognition patterns that enable it to infer the size of arrays which in turn enable additional fusion opportunities.

- Long term, MLIR dialects are being developed that express tensor operations with concurrent semantics. These dialects will then be lowered to existing MLIR dialects that also have support for not only the kind of fusion currently supported by ParallelAccelerator but also polyhedral fusion. The MLIR pipeline also includes functionality to lower these operations with concurrent semantics not only to multi-core CPUs but also various types of accelerators including GPUs.

- From the user perspective, nothing will change but we hope to incorporate this new MLIR-based compilation pipeline into Numba which will provide a superset of the existing parallelization opportunities as well as providing better backend code generation.

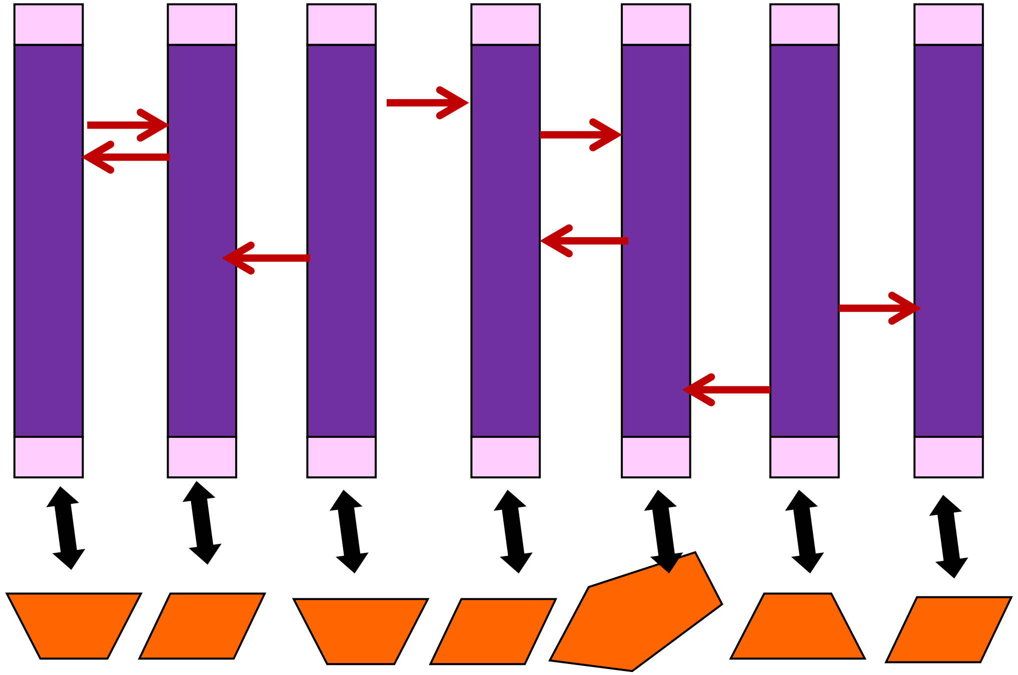
MPI4py

Execution Model: Communicating Sequential Processes (CSP)

- A collection of processes are launched when the program begins to execute.

- The processes interact through explicit communication events. All aspects of coordinating the processes (i.e. synchronization) are expressed in terms of communication events. →

- The CSP model does not interact with any concurrency issues inside a process … to the CSP model, they processes appear to be sequential.



- CSP is very general, but in practice, it is paired with the SPMD pattern

- Message passing systems are the class of APIs used to express CSP execution models.

- MPI is the dominant message passing library … has been since the mid 1990's.

- It has been extended to go well beyond CSP, but frankly few applications developers use those features.

MPI4py

- MPI4py: python binding to MPI

    - An MPI instance is initialized on import

```
from mpi4py import MPI

print(“Hello World!”)
```

- An MPI instance is finalized when all python processes in the program execution complete

- To launch a single mpi program on multiple nodes of a system (distributed memory) use the program **mpirun** where the flag **–np** is used to select how many copies of the program to run

```
> mpirun –np 3 python helloMPI.py

  Hello World!
  Hello World!
  Hello World!
```

MPI4py: Communicators, ranks and number of processes

- MPI in practice is all about the SPMD pattern … i.e., run the same program on each node and use the rank (ID) and number of processes to split up the work.

  - A **communicator** is used to organize MPI operations … it is a communication context and a process group.
  - If Np is the number of processes (the **size** of the process group), the **rank** is a unique number ranging from 0 to (Np-1). We use the rank as an ID for processes.

```
from mpi4py import MPI

comm =  MPI.COMM_WORLD
Np = comm.Get_size()
ID  = comm.Get_rank()

print(“Hello World from {0} or {1} \n”.format(ID, Np))
```

```
> mpirun –np 3 python helloMPI.py

  Hello World from 1 of 3
  Hello World from 0 of 3
  Hello World from 2 of 3
```

MPI4py: passing messages

- Processes coordinate their execution by passing messages … communication and synchronization combined through message passing function.

  - MPI4py supports two types of communication: one for **generic objects**, and another for **buffers** in contiguous memory (such as numpy arrays).
    - **Lower case function names**: Generic objects
    - **Uppercase function names**: Buffer objects

  - Buffer objects are much more efficient so if you are working with numpy arrays, use the Buffer object interface.

```
from mpi4py import MPI
comm =  MPI.COMM_WORLD
 Np = comm.Get_size()
 ID  = comm.Get_rank()

if (myrank == 0):
        a = [“I”,”love”,”MPI4py”]
        comm.send(a, dest = 1, tag=42)

else
        a_recv = comm.recv(source=0, tag=42)
        print(“ I am proc {0} and {0}\n”.format(a_recv))
```

```
> mpirun –np 2 python helloMPI.py

 I am proc 1 and ['I', 'love', 'MPI4py']
```

MPI Communication

- Blocking Communication
  - Python objects
    - comm.send(sendobj, dest=1, tag=0)
    - recvobj = comm.recv(None, src=0, tag=0)
  - Numpy buffer
    - comm.Send([sendarray, count, datatype], dest=1, tag=0)
    - comm.Recv([recvarray, count, datatype], src=0, tag=0)
- Nonblocking Communication
  - Python objects
    - reqs = comm.isend(obj, dest=1, tag=0)
    - reqr = comm.irecv(src=0, tag=0)
    - reqs.wait()
    - data = reqr.wait()
  - Numpy buffer
    - reqs = comm.Isend([sendarray, count, datatype], dest=1, tag=0)
    - reqr = comm.Irecv([recvarray, count, datatype], src=0, tag=0)
    - MPI.Request.Waitall([reqs, reqr])

We show these message passing routines for the case of node 0 sending a message to node 1

The parameter **tag** is used to prevent confusion between similar messages sent between pairs of node. It can take any integer type you wish … in this case 0

The parameter **datatype** is the MPI datatype which includes **MPI.INT**, **MPI.FLOAT**, **MPI.DOUBLE**, **MPI.CHAR** and others

count is the number of items of type datatype in the buffer

You can use type discovery in Python and write the triple [array, count, type] as just the array … so this becomes:
Reqr = comm.Irecv(recvarray, src=0, tag=0)

MPI4py: Reductions

- MPI includes all the usual collective communication routines (gather, scatter, broadcast, and more). The most commonly used is **reduction**.

  - Program sums area under the curve to compute an integral that ideally is equal to pi
  - We use a cyclic distribution of the loop to spread out the work among the processes
  - Reduction to compute the final answer

```
> mpirun –np 4 python piMPI.py

  pi is 3.1415926535899388
```

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
id   = comm.Get_rank()
numb   = comm.Get_size()
nsteps = 1000000

print(' Rank: ',id, ' numb: ',numb)

step = 1.0/nsteps
sum = np.array(0.0,'d')
pi = np.array(0.0,'d')
for i in range (id,nsteps,numb):
    x = step*(i+0.5)
    sum = sum + 4.0/(1.0 + x*x)

comm.Reduce(sum, pi, op=MPI.SUM, root=0)

if (id == 0):
   pi = pi * step
   print(' pi is :', pi)
```

Python multiprocessing

Python Multiprocessing

- Fork multiple processes from Python
- Useful to overcome GIL limitation, utilize multi-core machines
- Forked child processes run target function, with a set of arguments
- Multiple communication, coordination options:
  - Pipes, Queues
  - Shared memory arrays
  - Semaphores, mutexes
- Common patterns: fork-join, pipelines

Multiprocessing code

```
import numpy as np
import multiprocessing as mp

def calc_pi(nstart, nstop, step, i, outArr):
    out = np.frombuffer(outArr, dtype=np.float64)
    start = (nstart+0.5)*step
    stop = (nstop-0.5)*step
    nsteps = nstop-nstart
    X = np.linspace(start, stop, num=nsteps)
    Y = 4.0 / (1.0 + X*X)
    out[i] = np.sum(Y)

def piFunc(NumSteps, NumProcs):
    step = 1.0/NumSteps
    outArr = mp.Array('d',NumProcs,lock=False)
    out = np.frombuffer(outArr, dtype=np.float64)
    procs = []
    for i in range(NumProcs):
        nstart = (i*NumSteps)//NumProcs
        nstop = ((i+1)*NumSteps)//NumProcs
        procs.append( mp.Process( target=calc_pi,
            args=(nstart, nstop, step, i, outArr)) )
    for p in procs: p.start()
    for p in procs: p.join()
    return step * sum(out)

pi = piFunc(100000000,50)
```

Wrap shared memory buffer as numpy array object

Compute over part of range; written in numpy vector style; could use Python loops (slower, less memory), or Numba
Store result in position i of output array

Construct shared memory array,
Wrap as numpy array object

Construct processes to perform computation over parts of total range
Fork, Join pattern
Final reduction on shared memory array

Pi program

- Single dual-socket server
  - 2x Intel® Xeon® E5-2699v3 @ 2.3Ghz (36 cores, 72 hypercores, total)
  - 128GB RAM
- Mean, stddev of 10 runs (unless stated otherwise), after 1 warmup (in seconds)
- For multithreaded runs, we used the default number of threads.

Num steps	1e6	1e7	1e8	1e9	1e10
Python loops	0.09 (0.0006)	0.92 (0.006)			
Numpy		0.135 (0.005)	1.45 (0.0015)		
Numba		0.039 (0.001)	0.39 (0.001)	3.92 (0.003)	
Parallel Accelerator			0.019 (0.003)	0.141 (0.002)	1.48 (0.077)
Multiprocessing			0.229 (0.002)	1.54 (0.016)	
Dask		0.133 (0.008)	0.75 (0.04)	6.9 (0.46)	
PyOMP (loop)	0.051 (0.004) 5 runs	0.041 (0.005) 5 runs	0.073 (0.005) 5 runs	0.282 (0.02) 5 runs	1.56 (0.02) 5 runs

} Single threaded

} Compiled

← Compiled

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

OpenMP Organizations

- OpenMP Architecture Review Board (ARB) URL, the “owner” of the OpenMP specification:

  www.openmp.org

- OpenMP User’s Group (cOMPunity) URL:

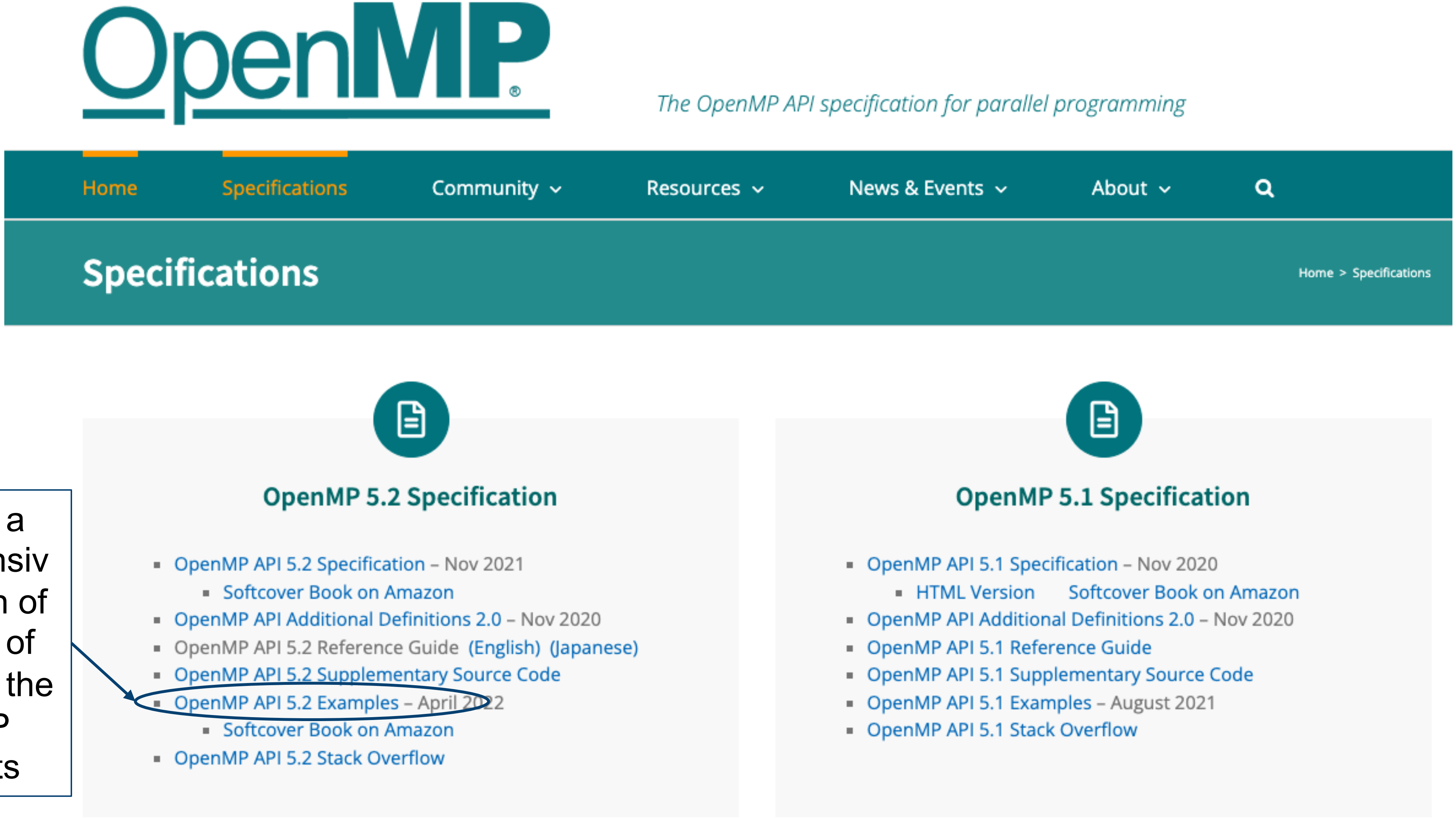
  www.compunity.org

Get involved, join the ARB and cOMPunity.

Help define the future of OpenMP

Resources

- www.openmp.org has a wealth of helpful resources



Specifications

OpenMP 5.2 Specification

- OpenMP API 5.2 Specification – Nov 2021
    - Softcover Book on Amazon
- OpenMP API Additional Definitions 2.0 – Nov 2020
- OpenMP API 5.2 Reference Guide (English) (Japanese)
- OpenMP API 5.2 Supplementary Source Code
- OpenMP API 5.2 Examples – April 2022
    - Softcover Book on Amazon
- OpenMP API 5.2 Stack Overflow

OpenMP 5.1 Specification

- OpenMP API 5.1 Specification – Nov 2020
    - HTML Version Softcover Book on Amazon
- OpenMP API Additional Definitions 2.0 – Nov 2020
- OpenMP API 5.1 Reference Guide
- OpenMP API 5.1 Supplementary Source Code
- OpenMP API 5.1 Examples – August 2021
- OpenMP API 5.1 Stack Overflow

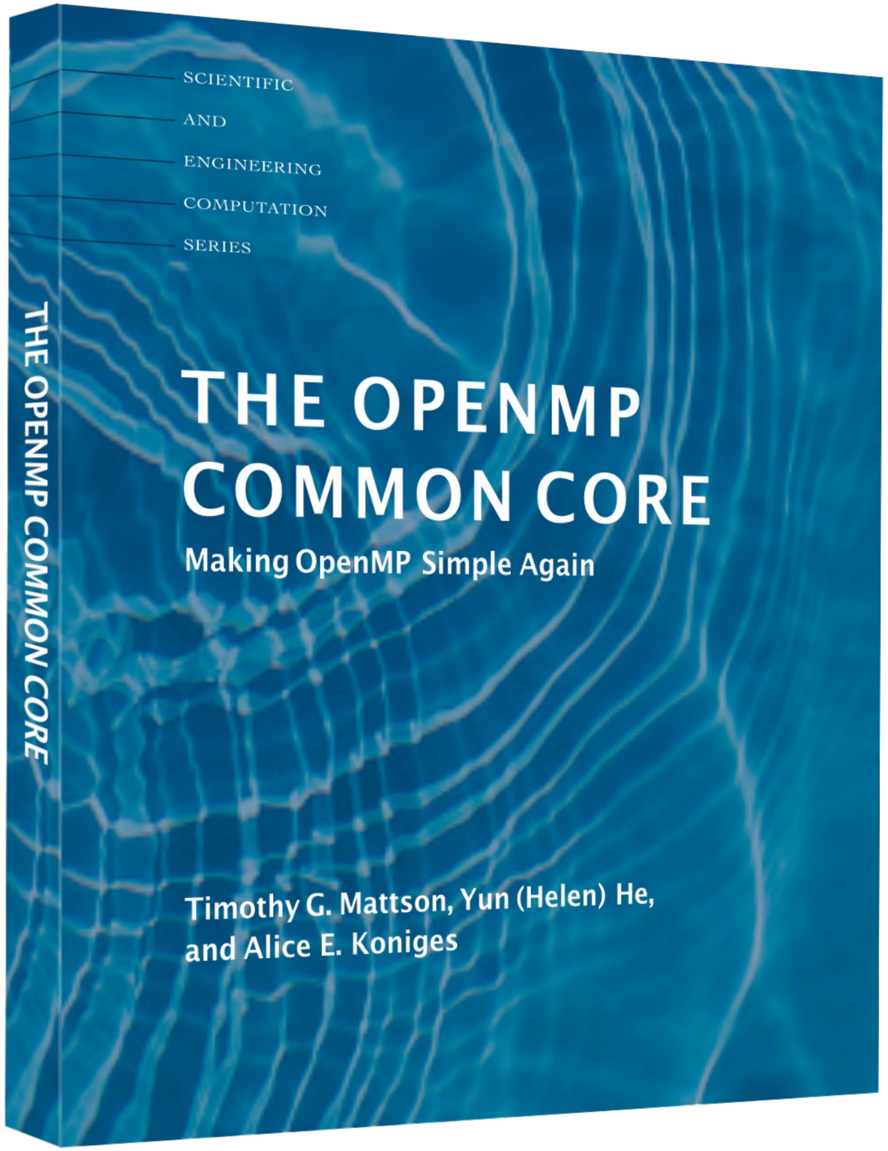
Including a comprehensive collection of examples of code using the OpenMP constructs

To learn OpenMP:

- An exciting new book that Covers the Common Core of OpenMP plus a few key features beyond the common core that people frequently use

- It’s geared towards people learning OpenMP, but as one commentator put it … **everyone at any skill level should read the memory model chapters**.

- Available from MIT Press



www.ompcore.com for code samples and the Fortran supplement

Books about OpenMP

A great book that covers OpenMP features beyond OpenMP 2.5

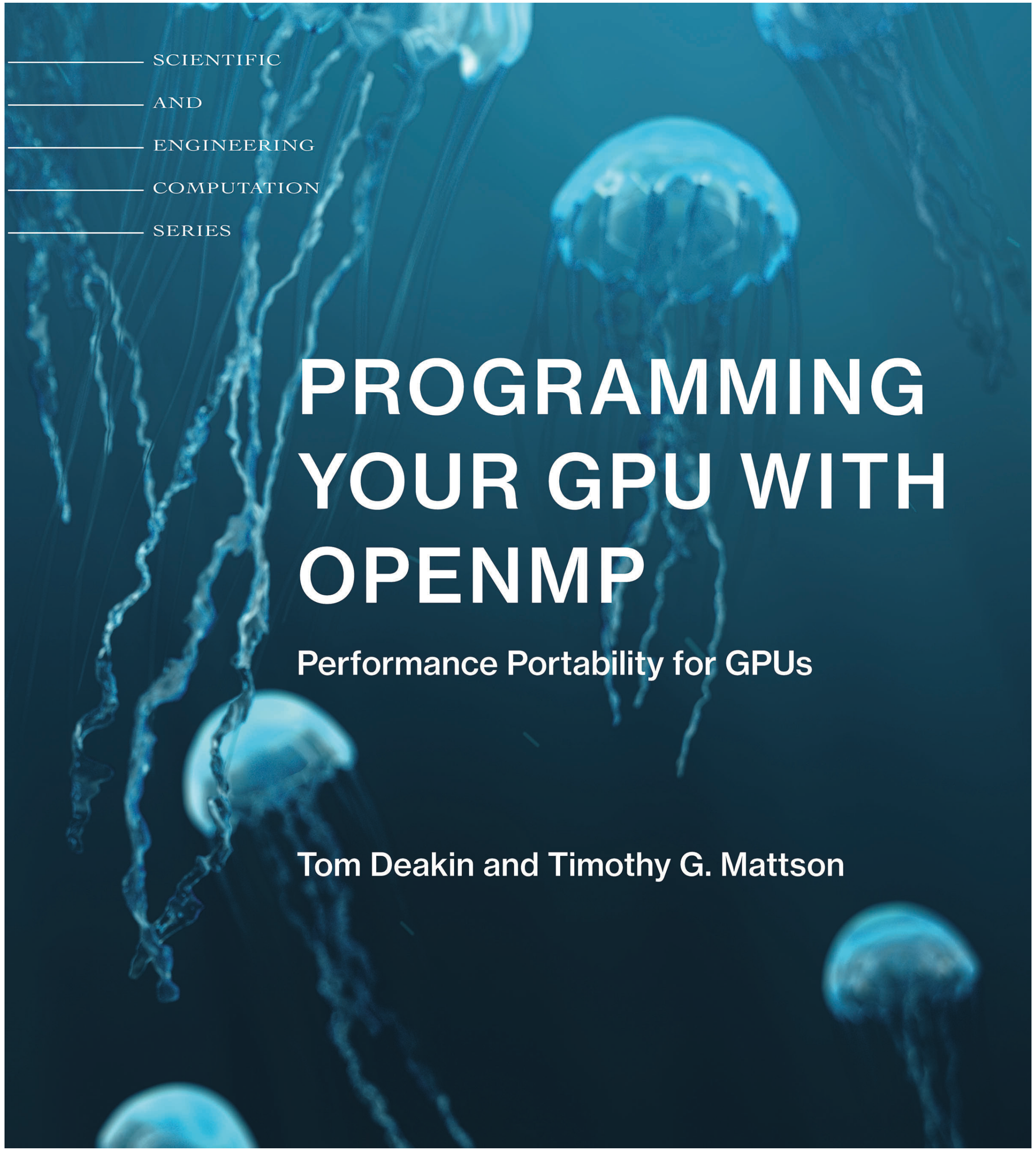
SCIENTIFIC
AND
ENGINEERING
COMPUTATION
SERIES
USING OPENMP – THE NEXT STEP
Affinity, Accelerators, Tasking, and SIMD
Ruud van der Pas, Eric Stotzer, and Christian Terboven
USING OPENMP – THE NEXT STEP

Books about OpenMP

The latest book on OpenMP …

Now available at amazon.com and MIT press.

A book about how to use OpenMP to program a GPU.



SCIENTIFIC
AND
ENGINEERING
COMPUTATION
SERIES
PROGRAMMING YOUR GPU WITH OPENMP
Performance Portability for GPUs
Tom Deakin and Timothy G. Mattson