How to Write Parallel Programs: the Python edition

Tim Mattson*

Parallel Computing Lab, Intel Corp.

*with a lot of help from Todd Anderson and Babu Pillai

Legal Disclaimer & Optimization Notice

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

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

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

Optimization Notice

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

Third party names are the property of their owners

Disclaimer

- The views expressed in this talk are those of the speakers and not their employer.
- If we say something "smart" or worthwhile:
  - Credit goes to the many smart people we work with.
- If we say something stupid…
  - It's our own fault

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

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

Introduction

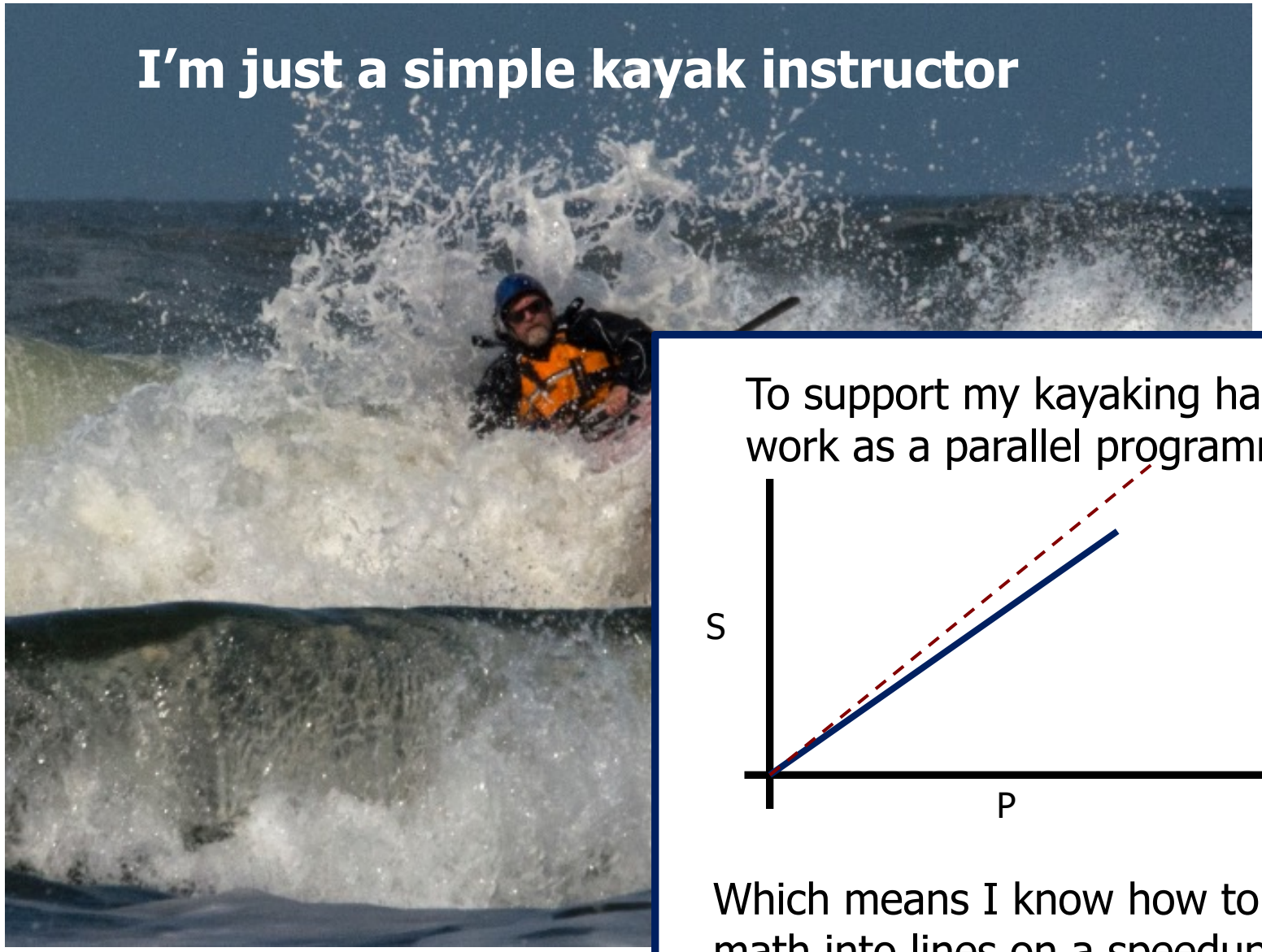


Photo © by Greg Clopton, 2014

To support my kayaking habit, I work as a parallel programmer

Which means I know how to turn math into lines on a speedup plot

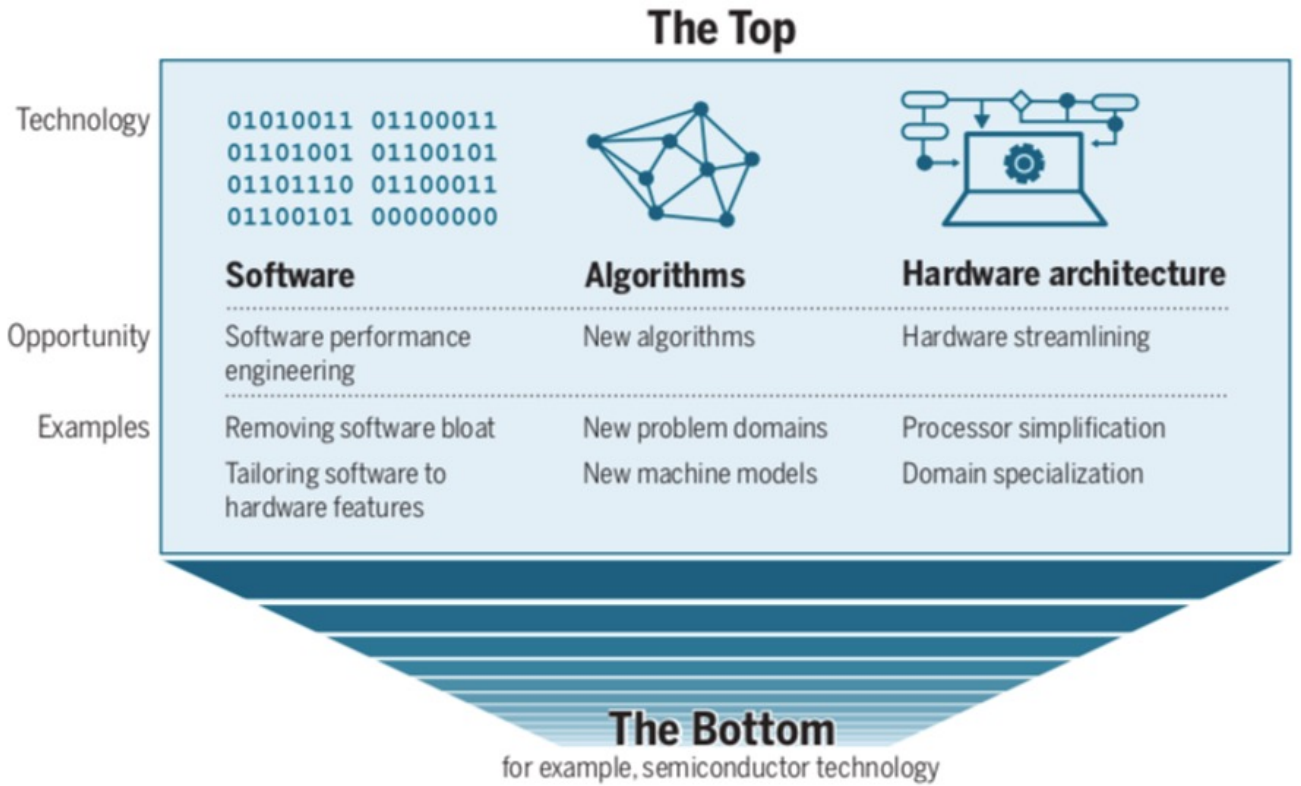
Software vs. Hardware and the nature of Performance

Up until ~2005, performance came from semiconductor technology

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

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

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



Since ~2005 performance comes from "the top"

Better software Tech.
Better algorithms
Better HW architecture#

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

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

The view of Python from an HPC perspective

(from the ”Room at the top” paper).

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

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

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

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

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

The view of Python from an HPC perspective

(from the ”Room at the top” paper).

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

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

This demonstrates a common attitude in the HPC community ….

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

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

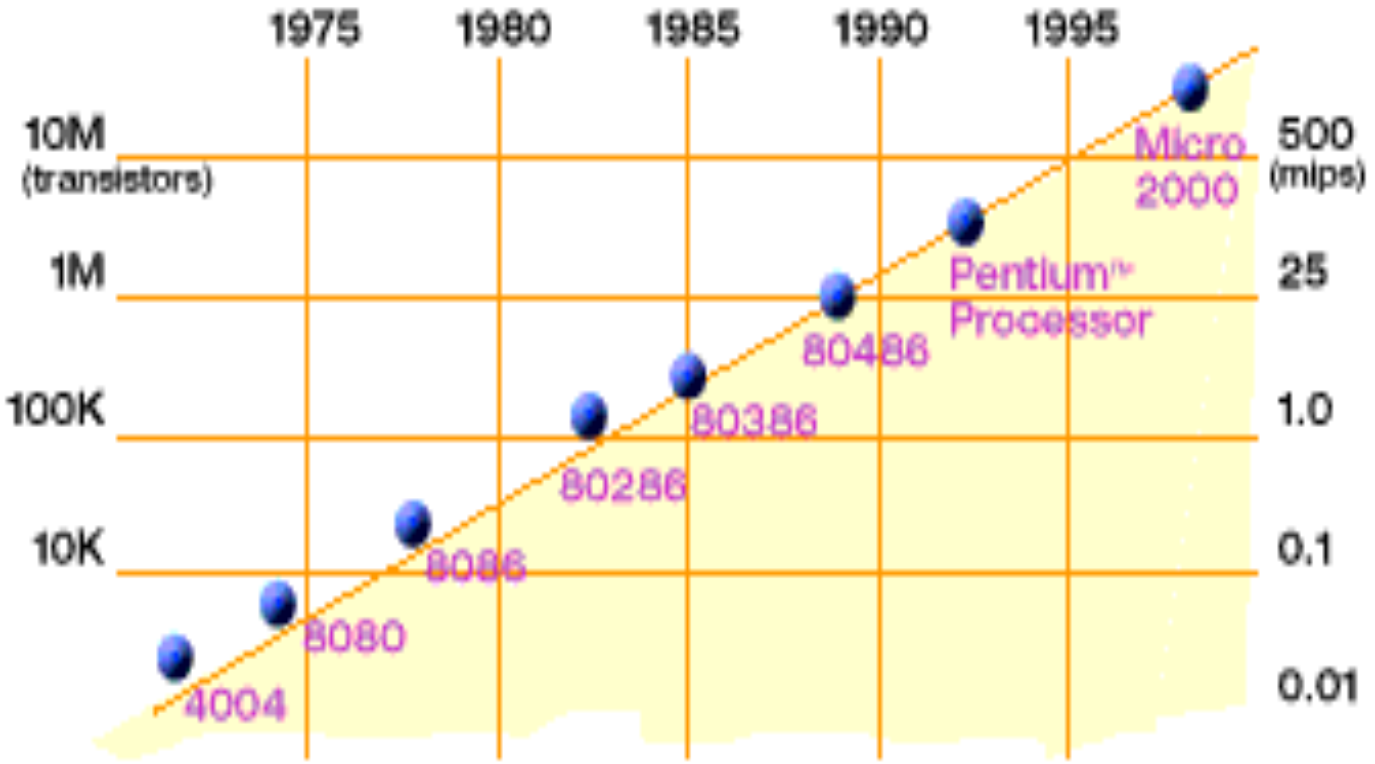
Amazon AWS c4.8xlarge spot instance, Intel® Xeon® E5-2666 v3 CPU, 2.9 Ghz, 18 core, 60 GB RAM

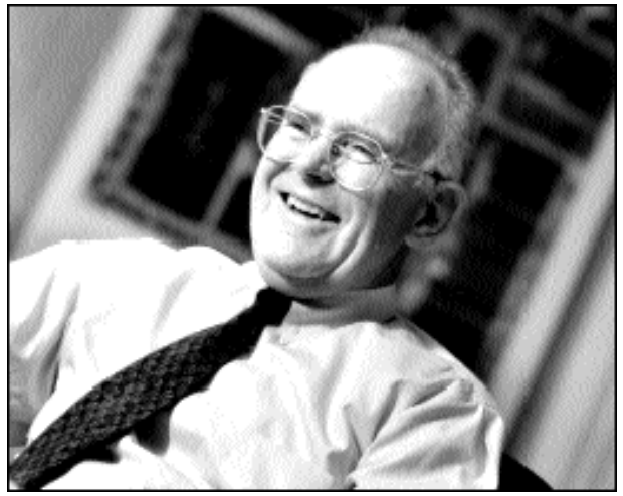
Our goal … to help people “keep their code in Python”

- 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.

Before talking about performance from Python code … we should step back and think about trends in computer performance over time

Moore's Law



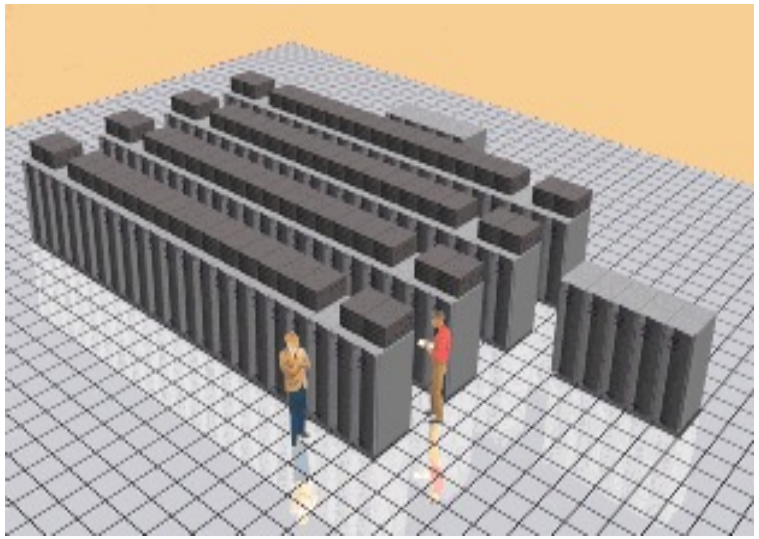


- In 1965, Intel co-founder Gordon Moore predicted (from just 3 data points!) that semiconductor density would double every 18 months.
  - ***He was right!*** Over the last 50 years, transistor densities have increased as he predicted.

Slide source: UCB CS 194 Fall'2010

Moore's Law: A personal perspective

First TeraScale* computer: 1997



Intel's ASCI Option Red

Intel's ASCI Red Supercomputer

9000 CPUs

one megawatt of electricity.

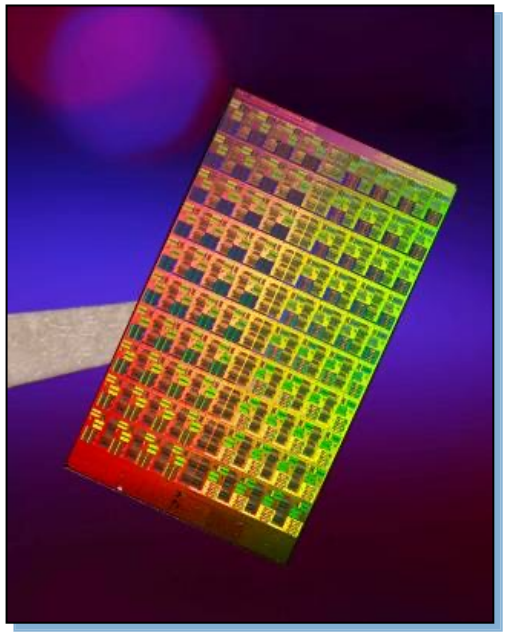
1600 square feet of floor space.

*Double Precision TFLOPS running MP-Linpack

A TeraFLOP in 1996: The ASCI TeraFLOP Supercomputer, Proceedings of the International Parallel Processing Symposium (1996), T.G. Mattson, D. Scott and S. Wheat.

10 years later

First TeraScale% chip: 2007



Intel's 80 core teraScale Chip

1 CPU

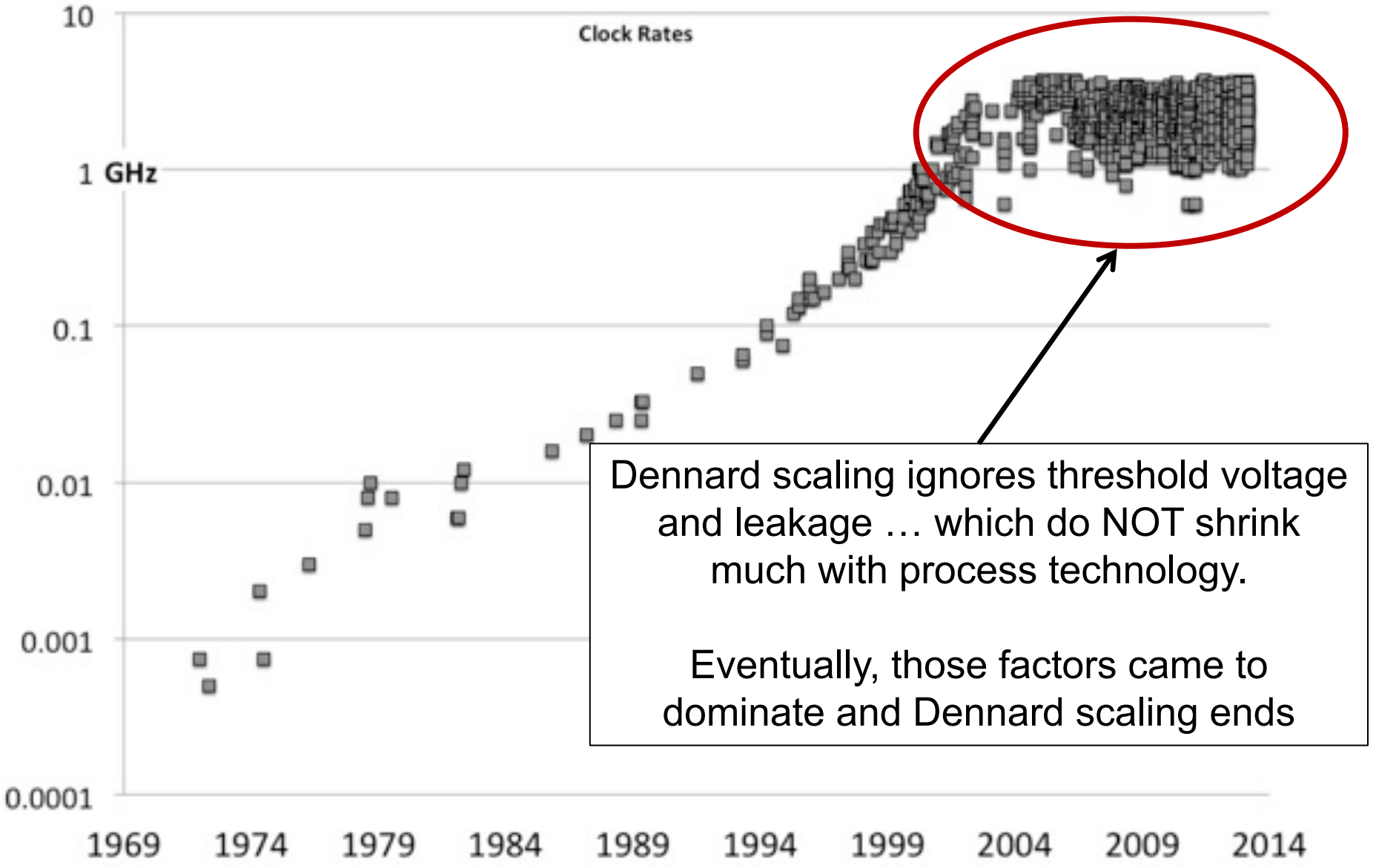
97 watt

275 mm2

%Single Precision TFLOPS running stencil

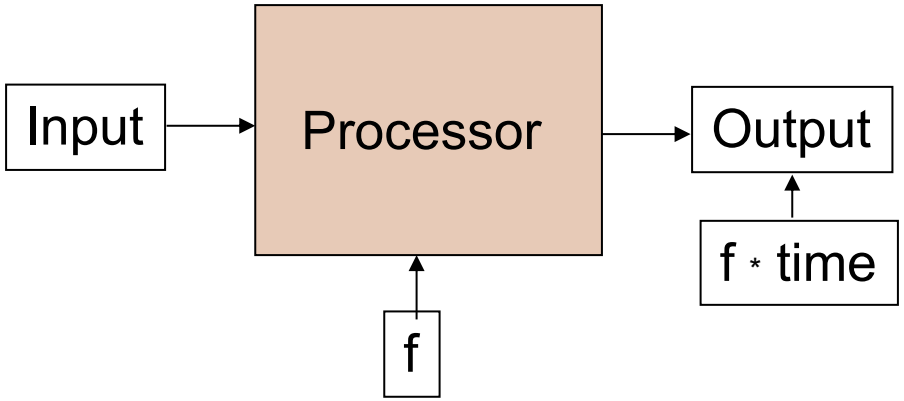
Programming Intel's 80 core terascale processor SC08, Austin Texas, Nov. 2008, Tim Mattson, Rob van der Wijngaart, Michael Frumkin

CPU Frequency (GHz) over time (years)



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

Consider power in a chip …



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:

$V * q = W.$

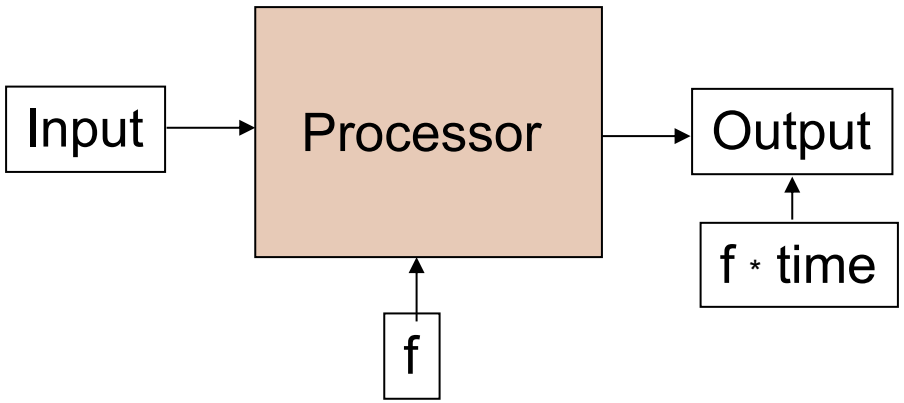
But for a circuit $q = CV$ so

$W = CV^2$

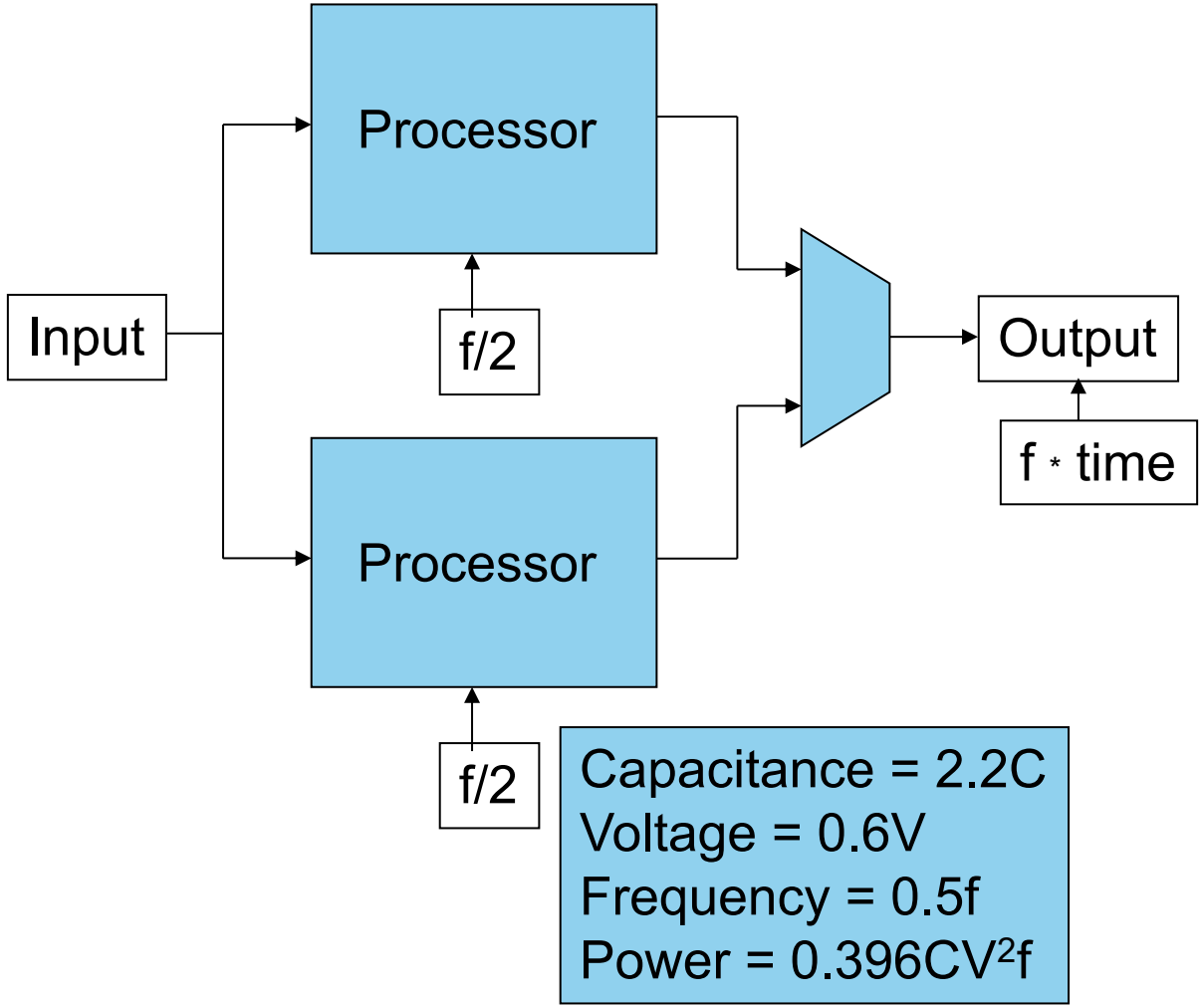
power is work over time … or how many times per second we oscillate the circuit

Power = W* F → 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 = $0.396CV^2f$

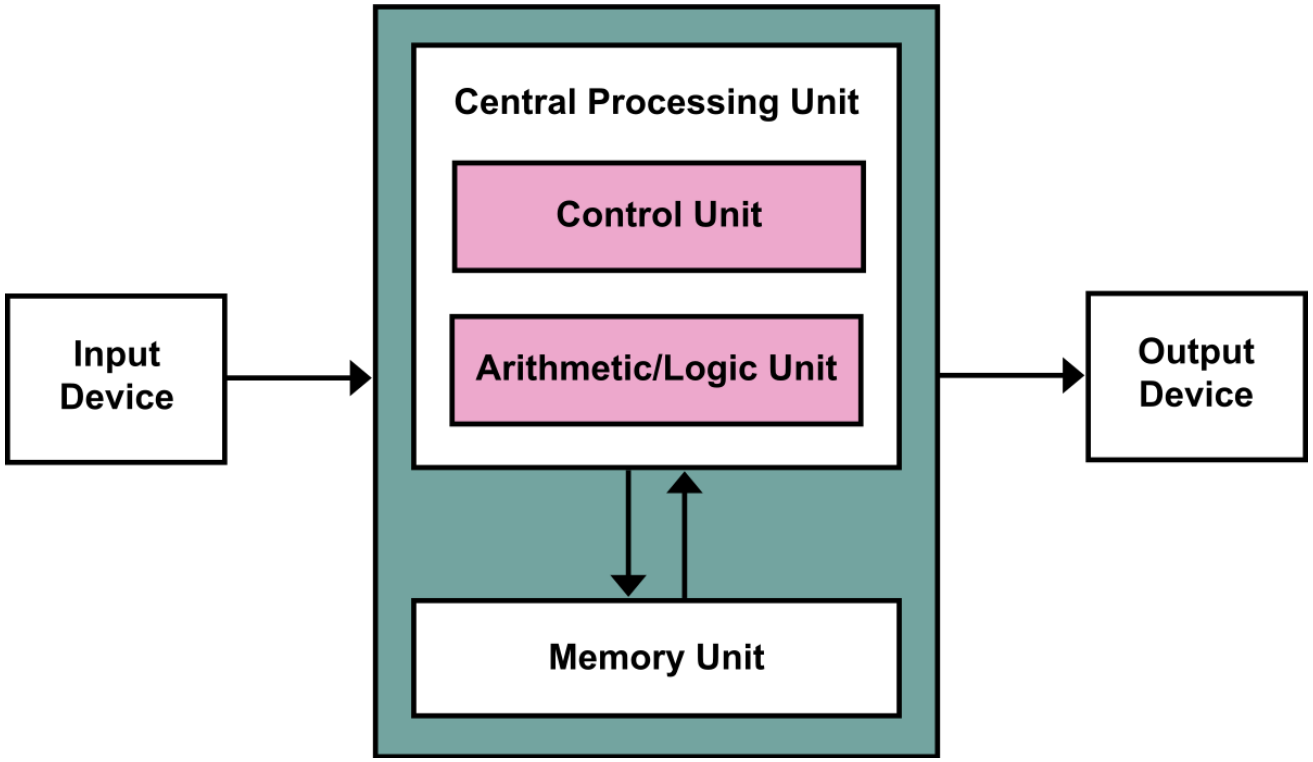
Chandrakasan, A.P.; Potkonjak, M.; Mehra, R.; Rabaey, J.; Brodersen, R.W., "Optimizing power using transformations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,*, vol.14, no.1, pp.12-31, Jan 1995

Source: Vishwani Agrawal

Let’s 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).



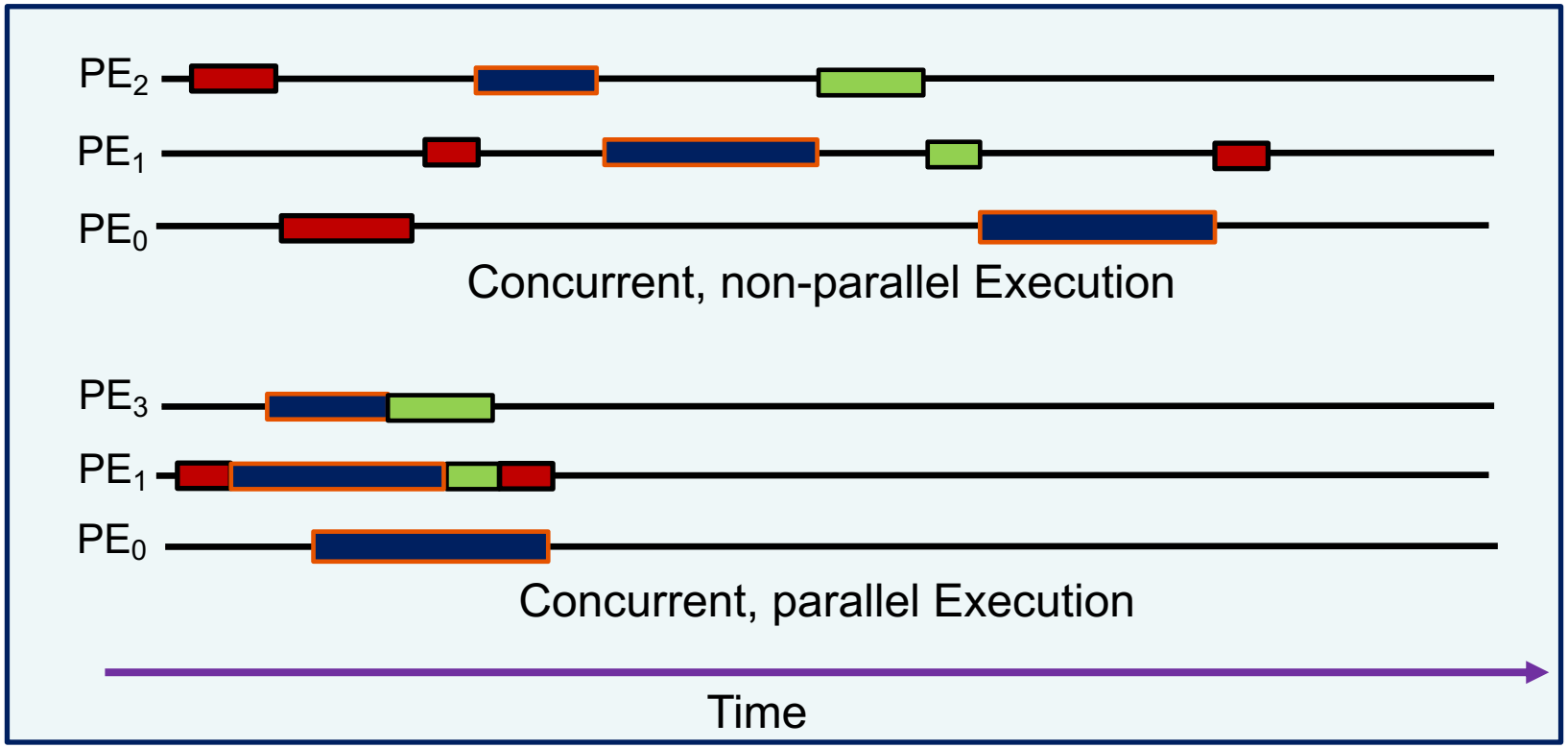
- **Task:**
  - A specific sequence of instructions plus a data environment. A program is composed of one or more tasks.

- **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.

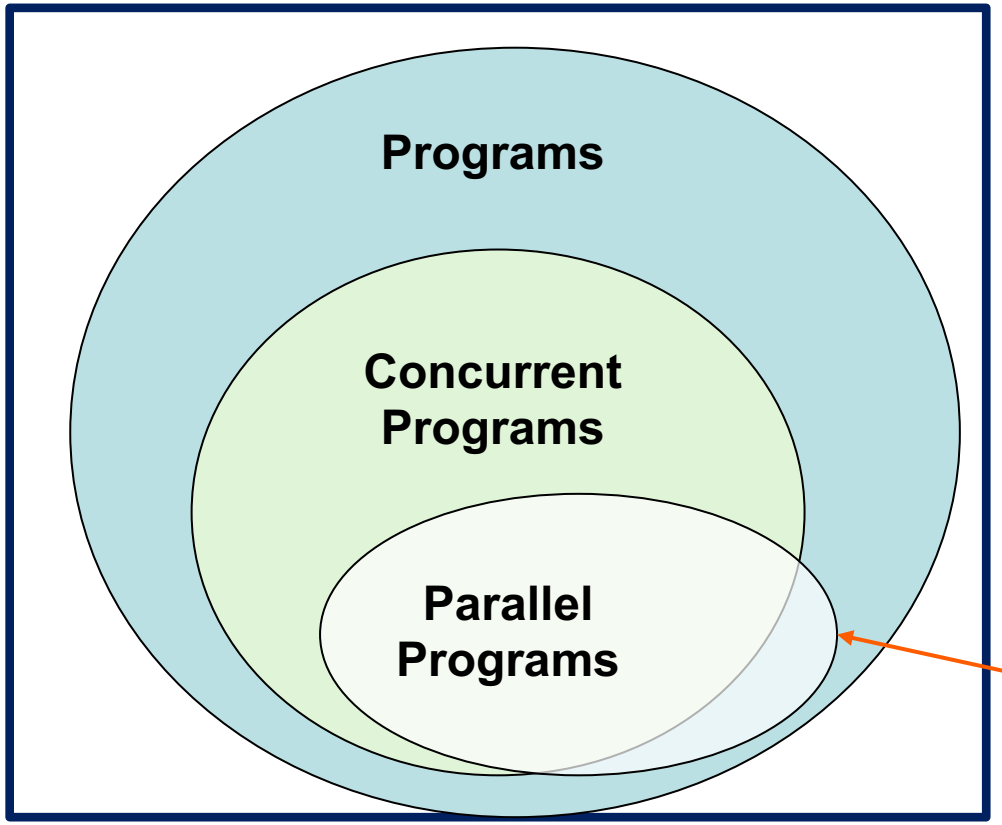


PE = Processing Element

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

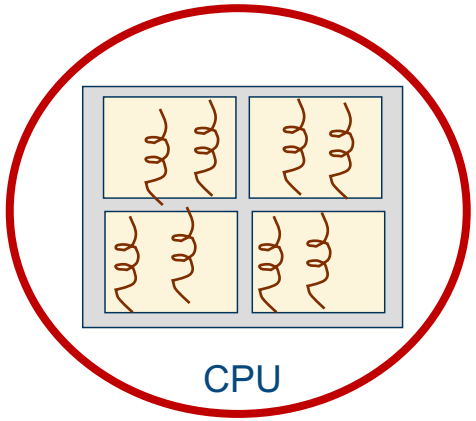
If tasks execute in "lock step" they are not concurrent, but they are still parallel. Example … a SIMD unit.

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

… So now lets talk about parallel hardware

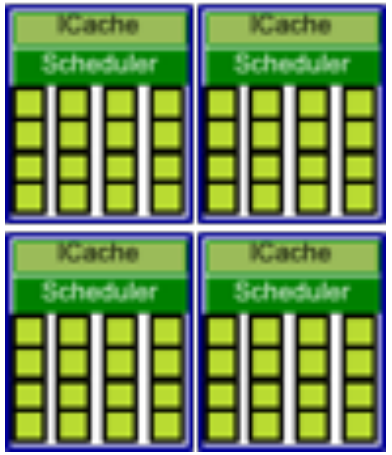
For hardware … parallelism is the path to performance

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.

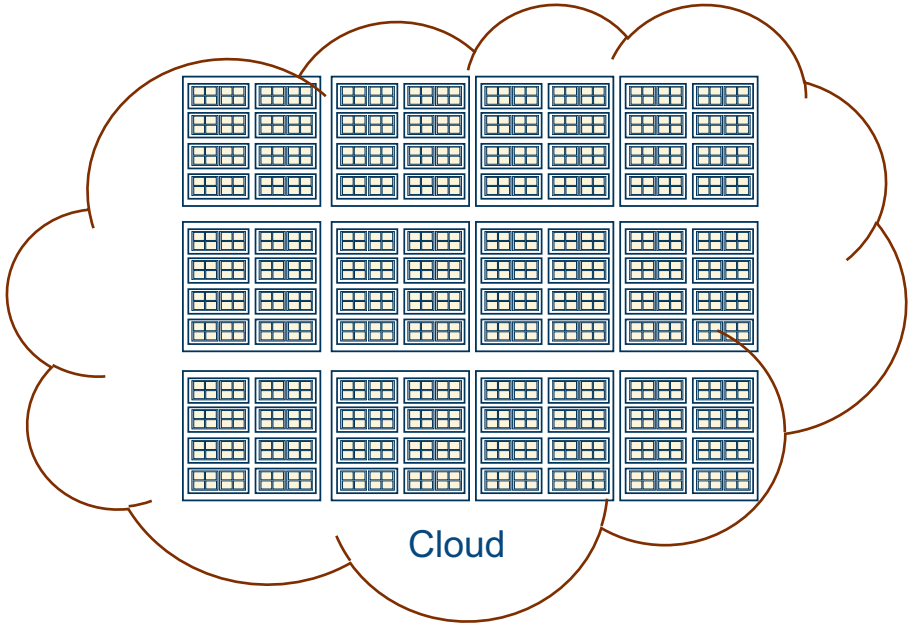


CPU

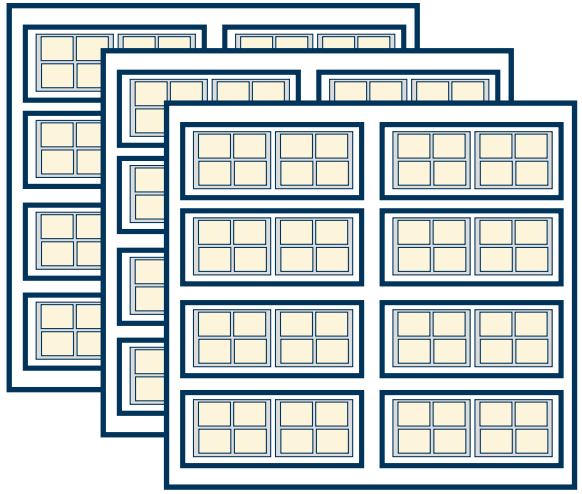
SIMD/Vector



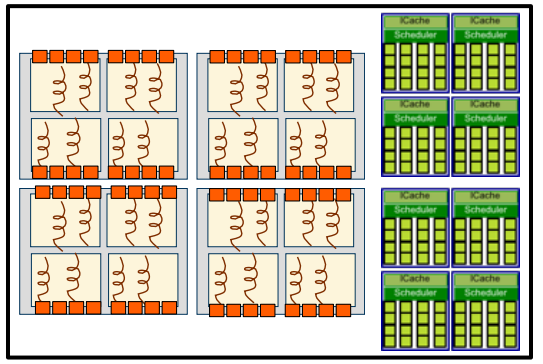
GPU



Cloud



Cluster

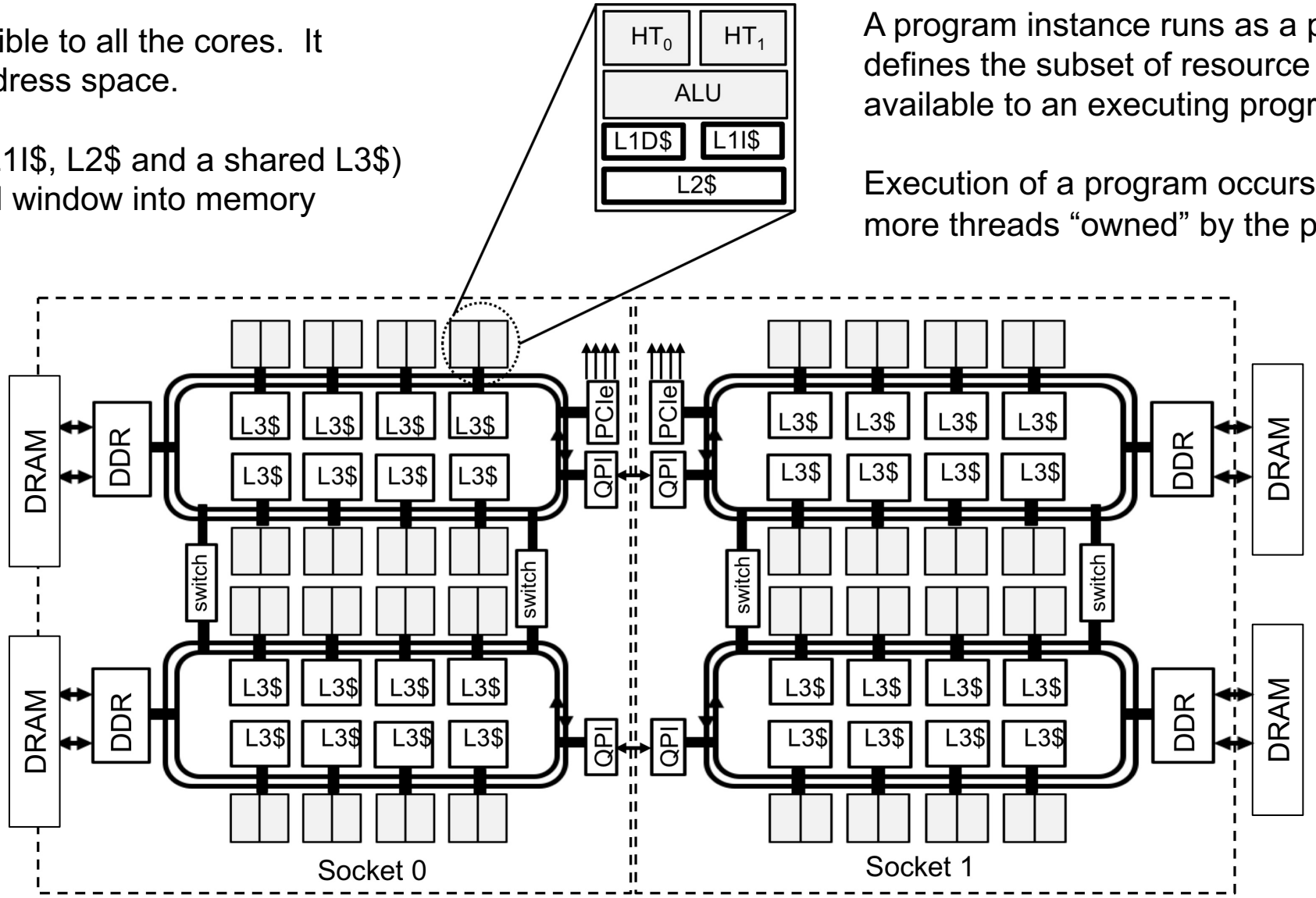


Heterogeneous node

A typical multi-core CPU

All the memory 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



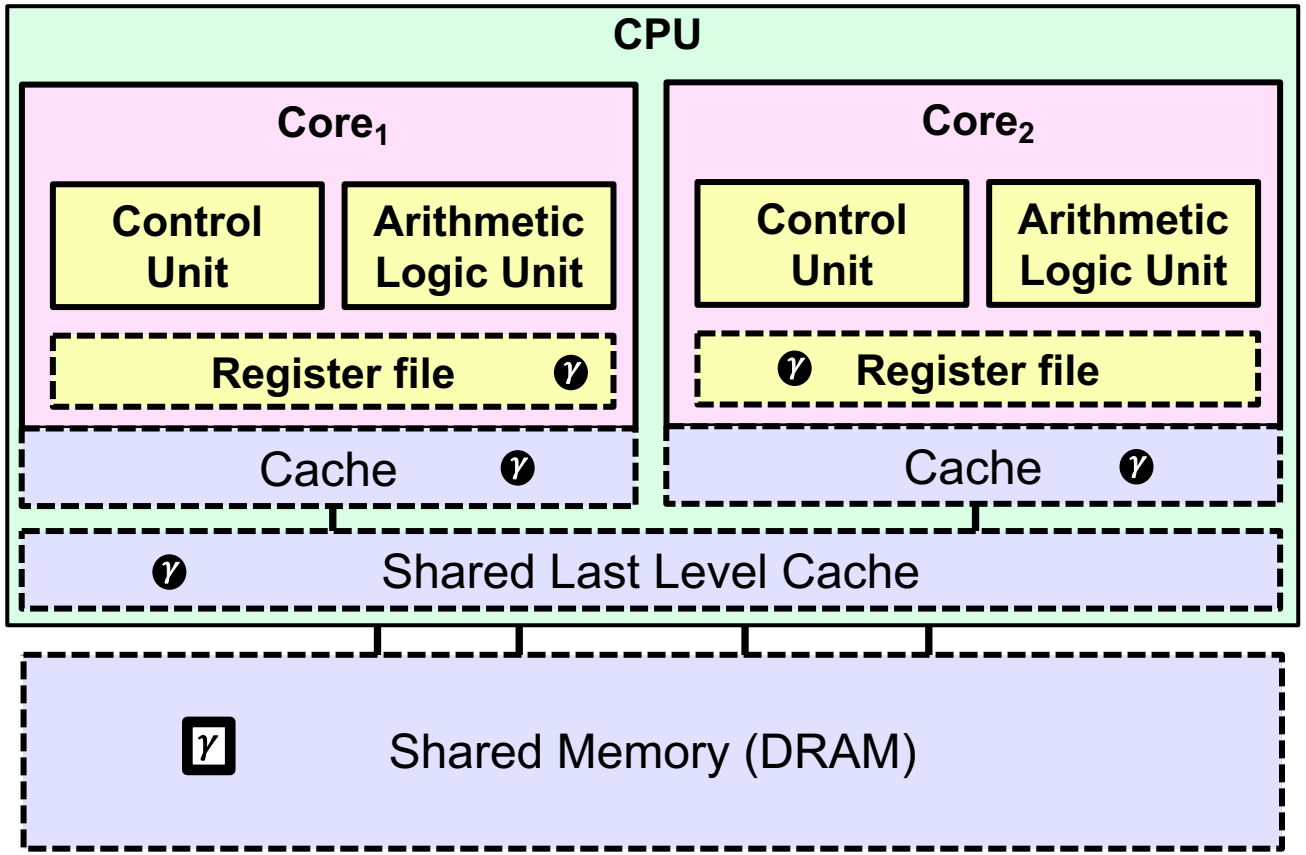
A program instance runs as a process. A process defines the subset of resource (such as memory) available to an executing program.

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

Memory Models …

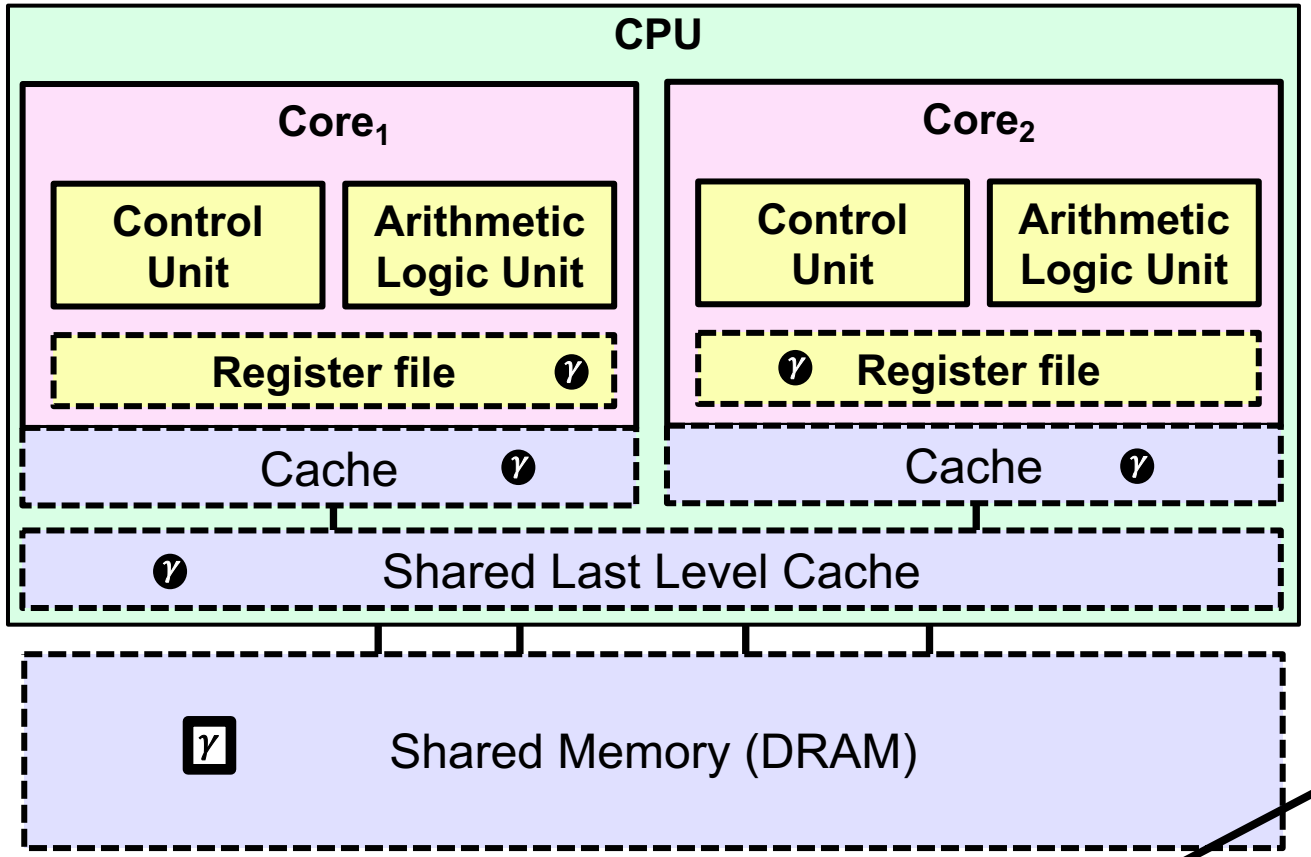
- Programming models for Multithreading support shared memory.
- All threads share an address space … but consider the variable γ



- Multiple copies of a variable (such as γ) may be present at various levels of cache, or in registers and they may ALL have different values.
- So which value of γ is the one a thread should see at any point in a computation?

Memory Models …

- Programming models for Multithreading support shared memory.
- All threads share an address space … but consider the variable γ

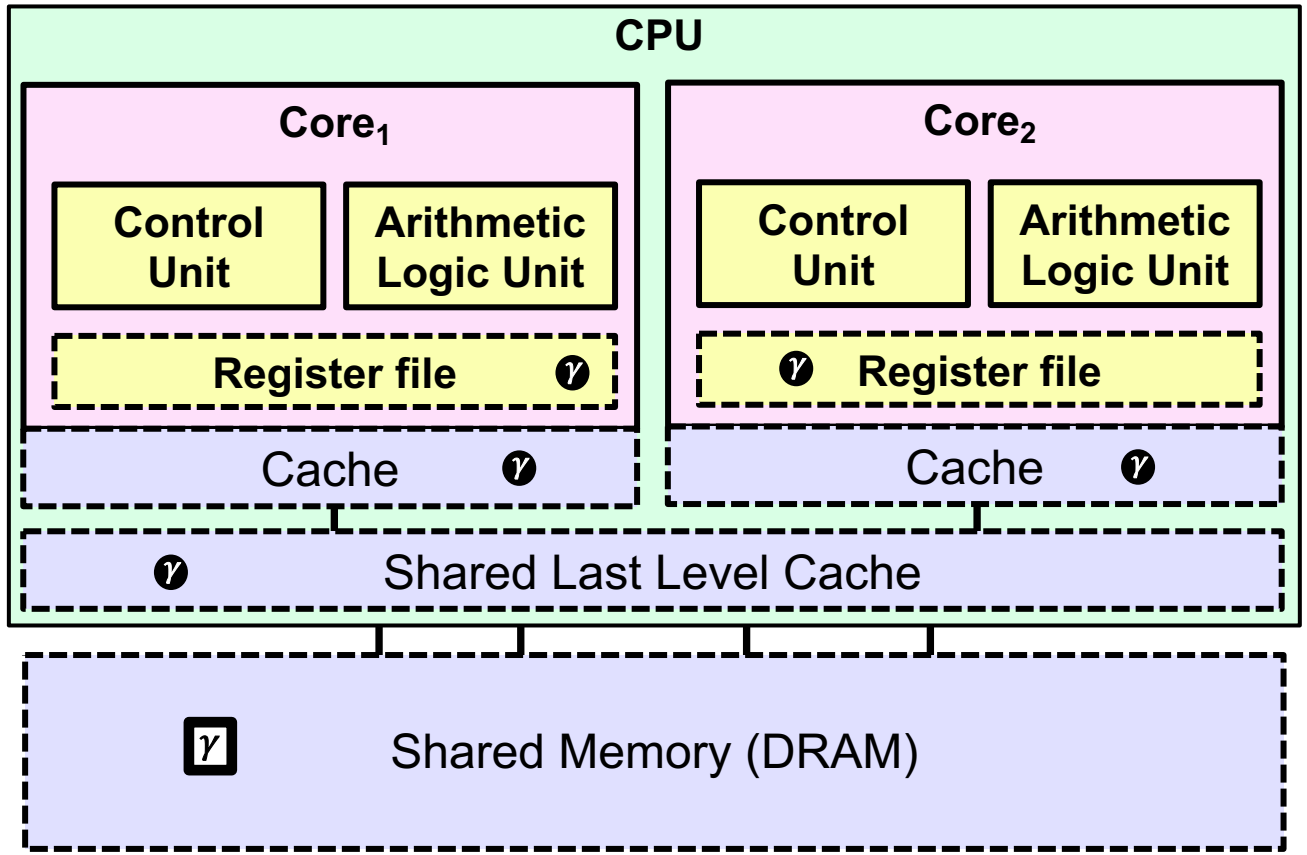


A memory consistency model (or "memory model" for short) provides the rules needed to answer this question.

- Multiple copies of a variable (such as γ) may be present at various levels of cache, or in registers and they may ALL have different values.
- So which value of γ is the one a thread should see at any point in a computation?

Relaxed Consistency memory models

- Most (if not all) multithreading programming models supports a **relaxed-consistency** memory model
  - Threads can maintain a **temporary view** of shared memory that is not consistent with that of other threads
  - These temporary views are made consistent only at certain points in the program
  - The operation that enforces consistency is called a memory-fence or **flush operation***



Flush Example (OpenMP in C)

- Flush forces data to be updated in memory so other threads see the most recent value*

```
double A;

A = compute();

#pragma omp flush(A)

   // flush to memory to make sure other
   //  threads can pick up the right value
```

In OpenMP you use compiler directives (a pragma in C) to tell the compiler what to do for multithreading

Memory consistency is too hard for most people to manage…
so python dodges the issue with the GIL (global interpreter lock)

- Understanding the GIL through a sequence of definitions.

- A **program** instance is called a **process**. The OS uses the process to manage the resources available to an instance of a program including the range of memory addresses, threads, I/O, etc.
- A process forks one or more **threads** to execute the program's instructions
- Threads have their own private memory (a stack) and **share the shared address** space associated with the process (the heap).
- **Threads execute concurrently** … i.e. they are unordered. Order constraints are imposed through a **synchronization** event.
- The most common synchronization event is a **mutex** (a block of code that only one thread at a time can execute … this is called mutual exclusion)
- A **variable** is a name for a location in an address space. If that variable can be read and/or written by multiple threads, it is a **shared variable**.
- A mixture of reads and writes to a shared variable that are unordered is called a **data race**. A program with a data race is undefined.
- A **race condition** occurs when the result of a program depends on the order that threads are scheduled.
- A **thread-safe library** is properly synchronized so it produces the same result if called by multiple concurrent threads or sequentially by a set of threads.
- An **interpreter** is a program that runs other programs. Python, an interpreted language, is run by an interpreter.
- If multiple threads are active and being processed together by a python interpreter updates to shared resources of any kind can lead to a race condition … or worse, a data race.
- Synchronizing at the level of each individual resource is HARD. Python, favoring safety over performance, has a single mutex lock on the interpreter so only one thread at a time can run on the interpreter.
- The mutex used to implement this behavior is called the **Global Interpreter Lock** or the **GIL.**

Memory consistency is too hard for most people to manage…

so python dodges the issue with the GIL (global interpreter lock)

- Understanding the GIL through a sequence of definitions.

– A **program** instance is called a **process**. The OS uses the process to manage the resources available to an instance of a pr… addresses, thre…

– A process forks… program's instru…

– Threads have t… **share the shar**… process (the he…

– **Threads execu**… unordered. Orc… **synchronizatic**…

– The most comn… (a block of code… execute … this…

– A **variable** is a name for a location in an address space. If that variable can be read and/or written by multiple threads, it is a **shared variable**.

– A mixture of reads and writes to a shared variable that are unordered is called a **data race**. A program with a data race is undefined…

…f a program …eduled.

…ized so it …ble concurrent …

…r programs. … an interpreter.

…ocessed together …esources of any …se, a data race.

…al resource is …rmance, has a …single mutex lock on the interpreter so only one thread at a time can run on the interpreter.

– The mutex used to implement this behavior is called the **Global Interpreter Lock** or the **GIL.**

Threads in a python program do not execute in parallel as long as the GIL is engaged.

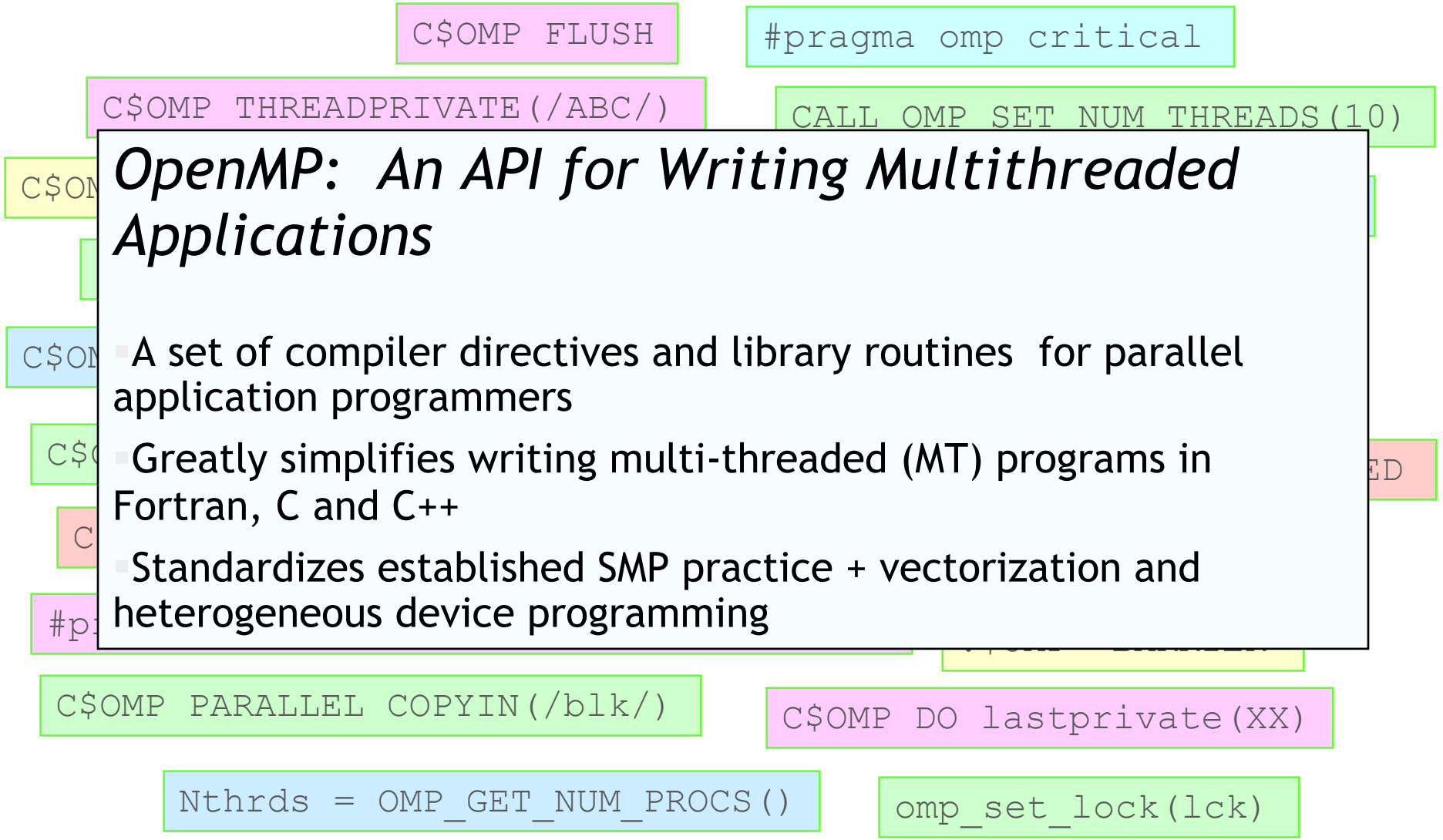
Each process has its own instance of the interpreter and hence, its own GIL.

Two options for parallelism:

- Multiprocessing can be parallel in python but then sharing data between processes is complicated since processes can't see each others address spaces.
- Drop down to a lower layer abstraction beneath the interpreter … and hence no GIL

OpenMP is a particularly simple language for parallel programming … so let's talk about the OpenMP binding to Python and use it to cover the fundamental design patterns of parallel programming

OpenMP* Overview



OpenMP: An API for Writing Multithreaded Applications

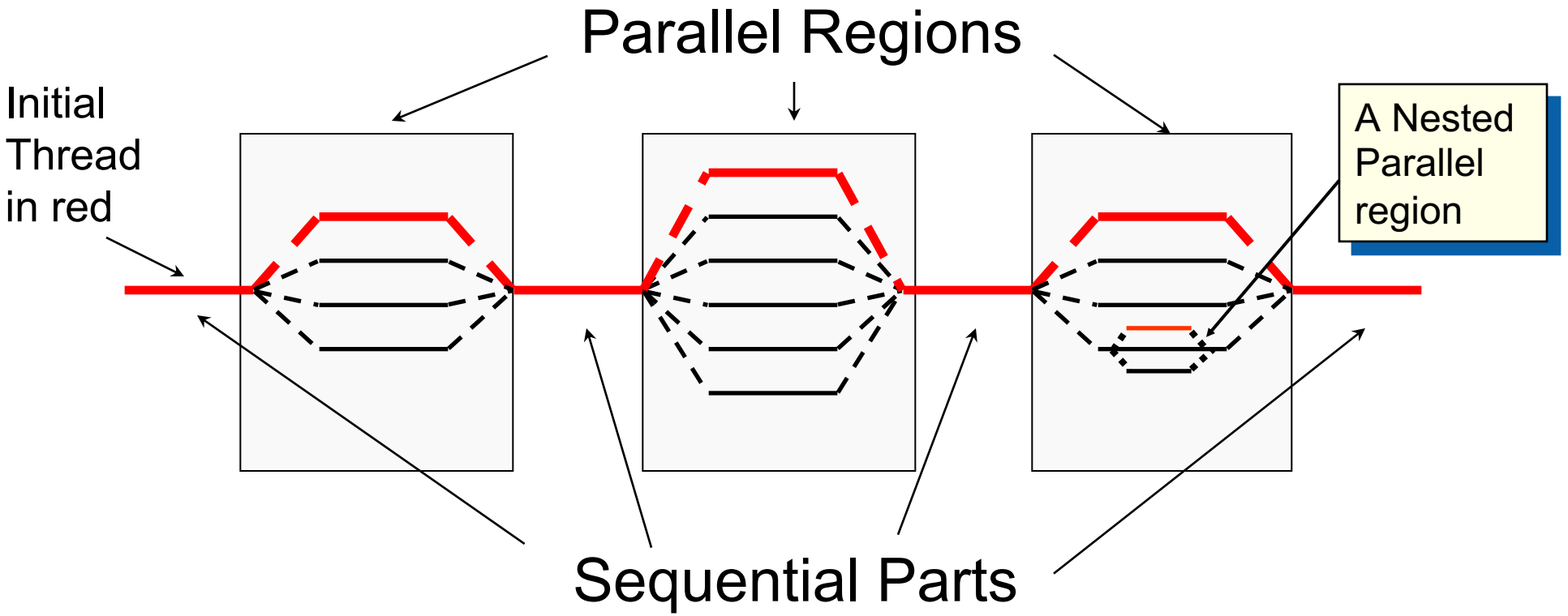
- A set of compiler directives and library routines for parallel application programmers
- Greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++
- Standardizes established SMP practice + vectorization and heterogeneous device programming

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

OpenMP Execution Model

Fork-Join Parallelism:

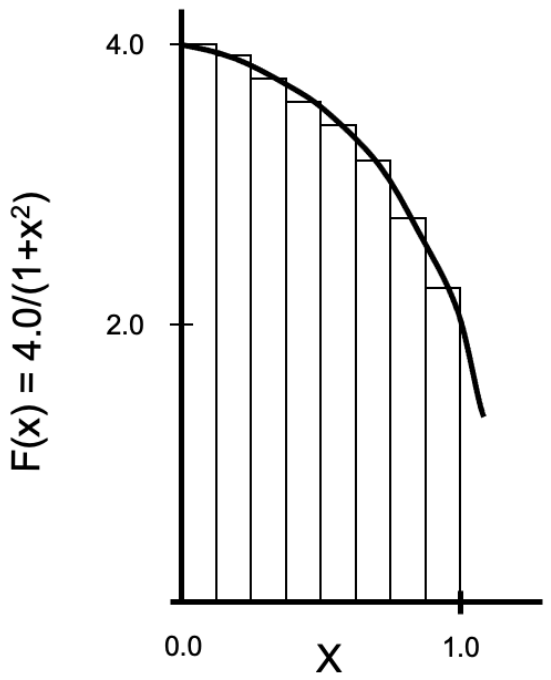
- Initial thread spawns a team of threads as needed.
- Parallelism added incrementally until performance goals are met, i.e., the sequential program evolves into a parallel program.



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 i^{th} interval midpoint.

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

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

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

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

A loop carried dependency

Recast to compute from i

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

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

A loop carried dependency

Recast to compute from i

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

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

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

    pi = step*sum
    return pi

pi = piFunc(100000000)
```

OpenMP constructs managed through the *with* context manager.

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

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

Numerical Integration results in seconds … lower is better

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

10^8 steps

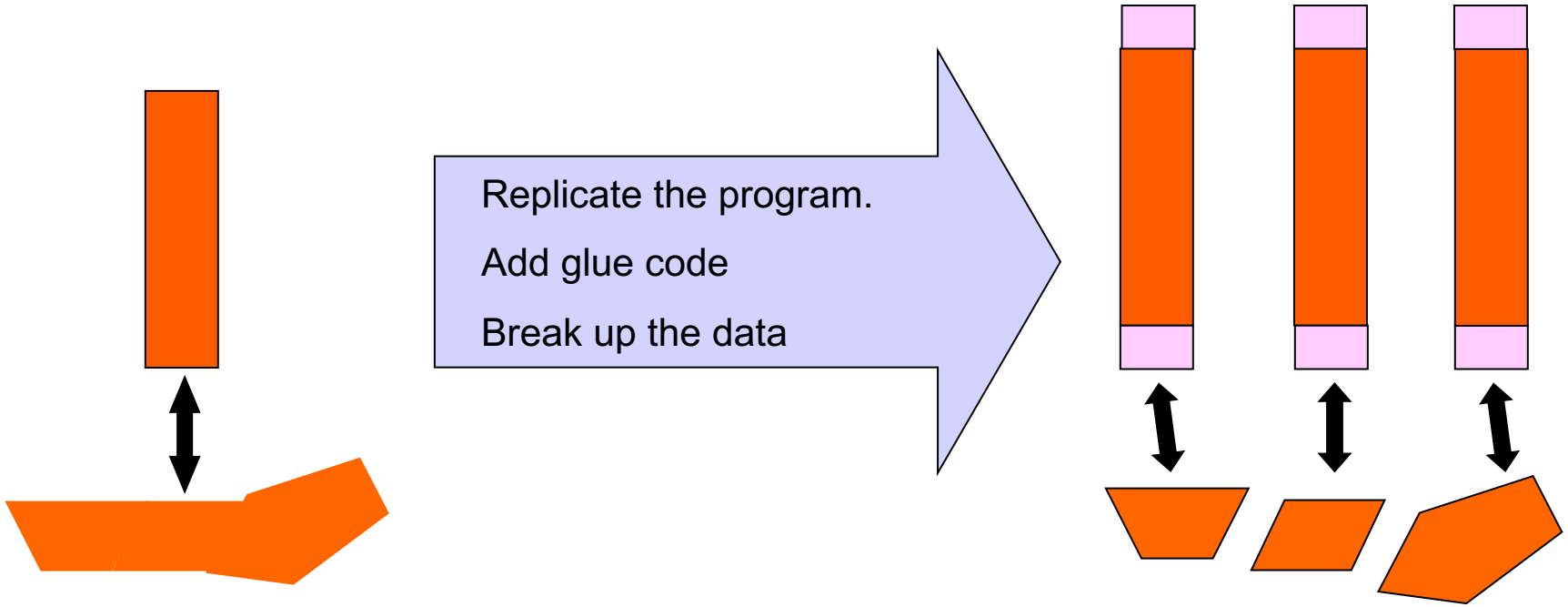
Intel® Xeon® E5-2699 v3 CPU with 18 cores running at 2.30 GHz.

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)

SPMD (Single Program Multiple Data) design pattern

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



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

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

Single Program Multiple Data (SPMD)

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

pi = piFunc(100000000)
```

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

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

Numerical Integration results in seconds … lower is better

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

10^8 steps

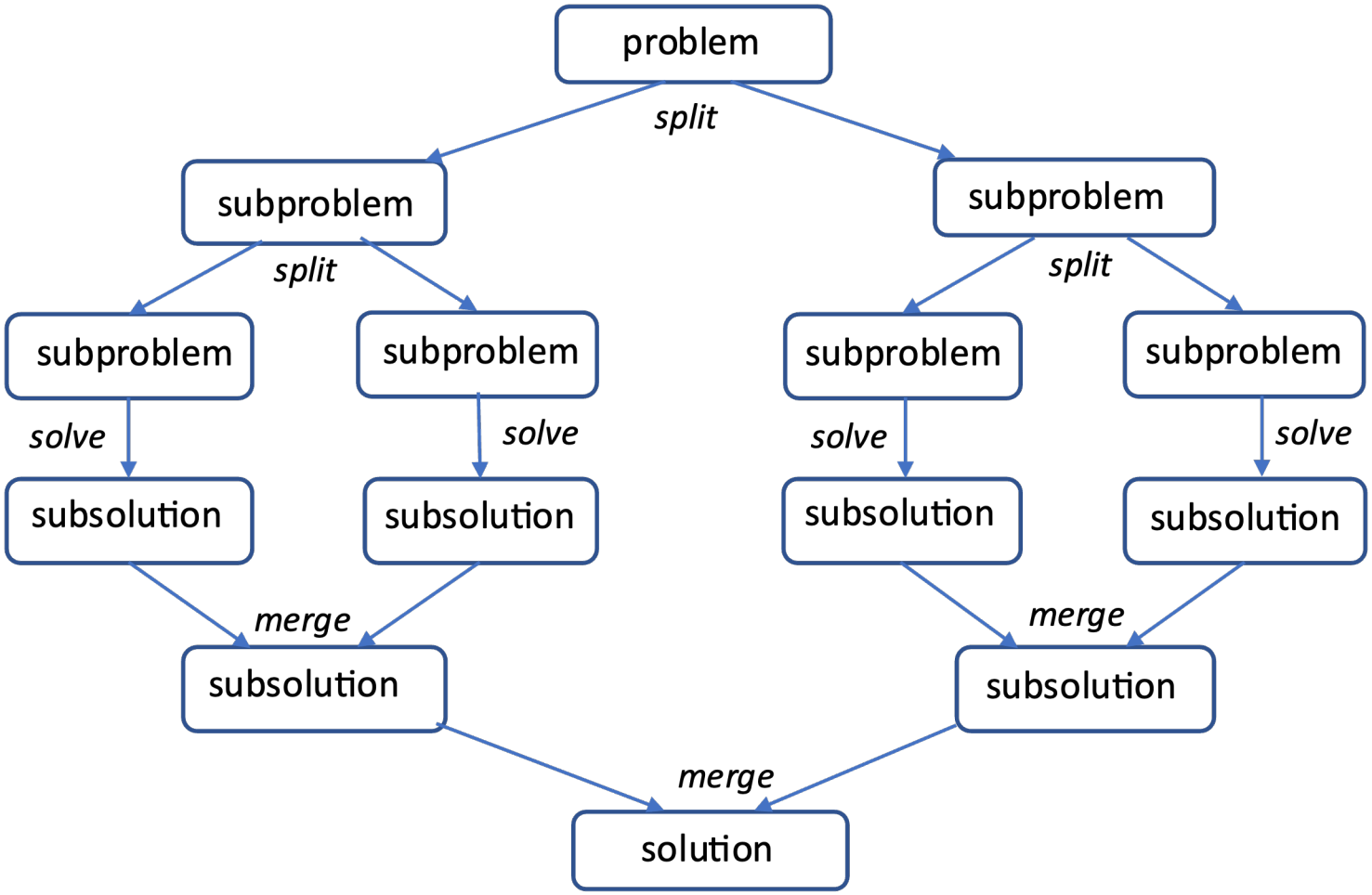
Intel® Xeon® E5-2699 v3 CPU with 18 cores running at 2.30 GHz.

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

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

Divide and conquer design pattern

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



3 Options for parallelism:

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

Divide and conquer (with explicit tasks)

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

Solve

Split

Merge

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

    pi = step*sum
    return step*sum

pi = piFunc(100000000)
```

Fork threads and launch the computation

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

Numerical Integration results in seconds … lower is better

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

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

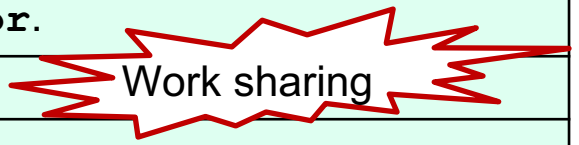
Ran each case 5 times and kept the minimum time. **JIT time is not included** for PyOMP (it was about 1.5 seconds)

OpenMP subset supported in PyOMP

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

OpenMP subset supported in PyOMP

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



How did we implement PyOMP?

We used “the magic” of Numba

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)
```

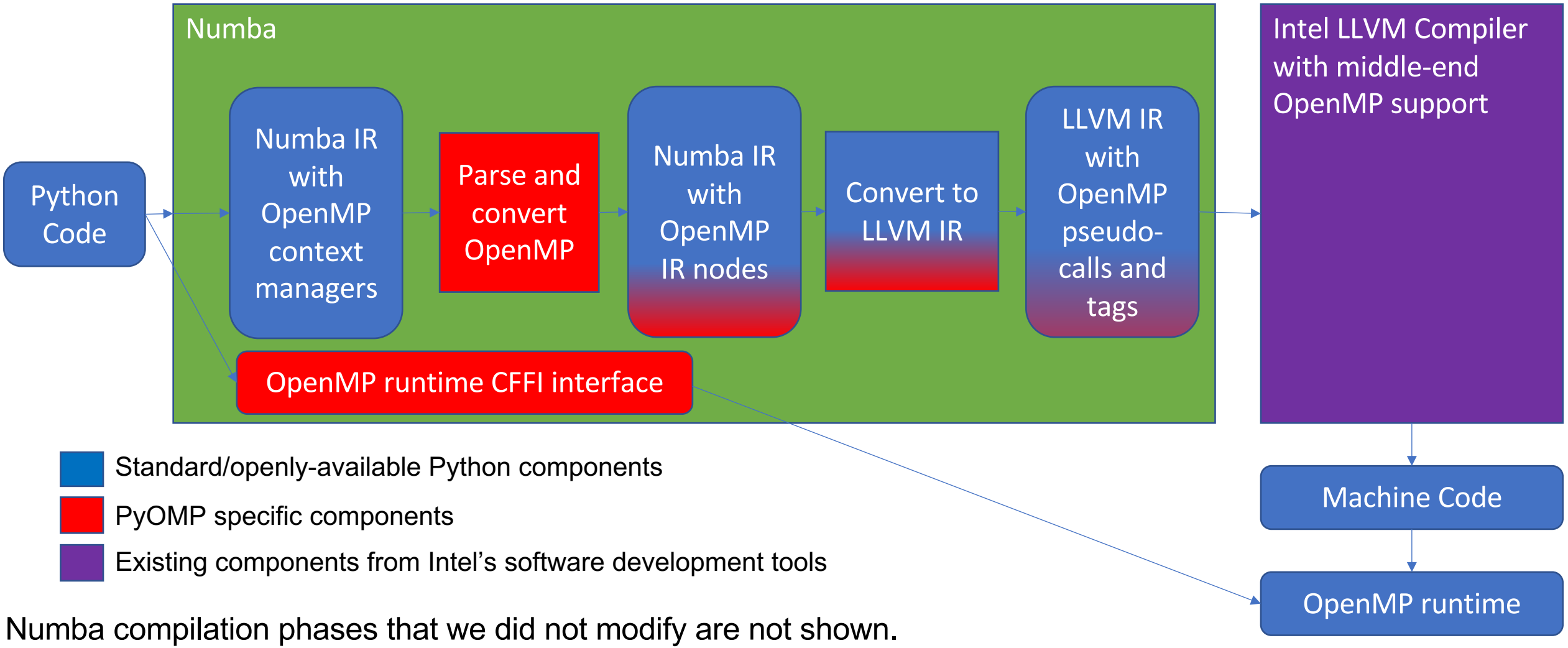
Numba jit compiler applied the first time a function is encountered. Numba caches the code so subsequent calls to the function don't run the jit step.

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

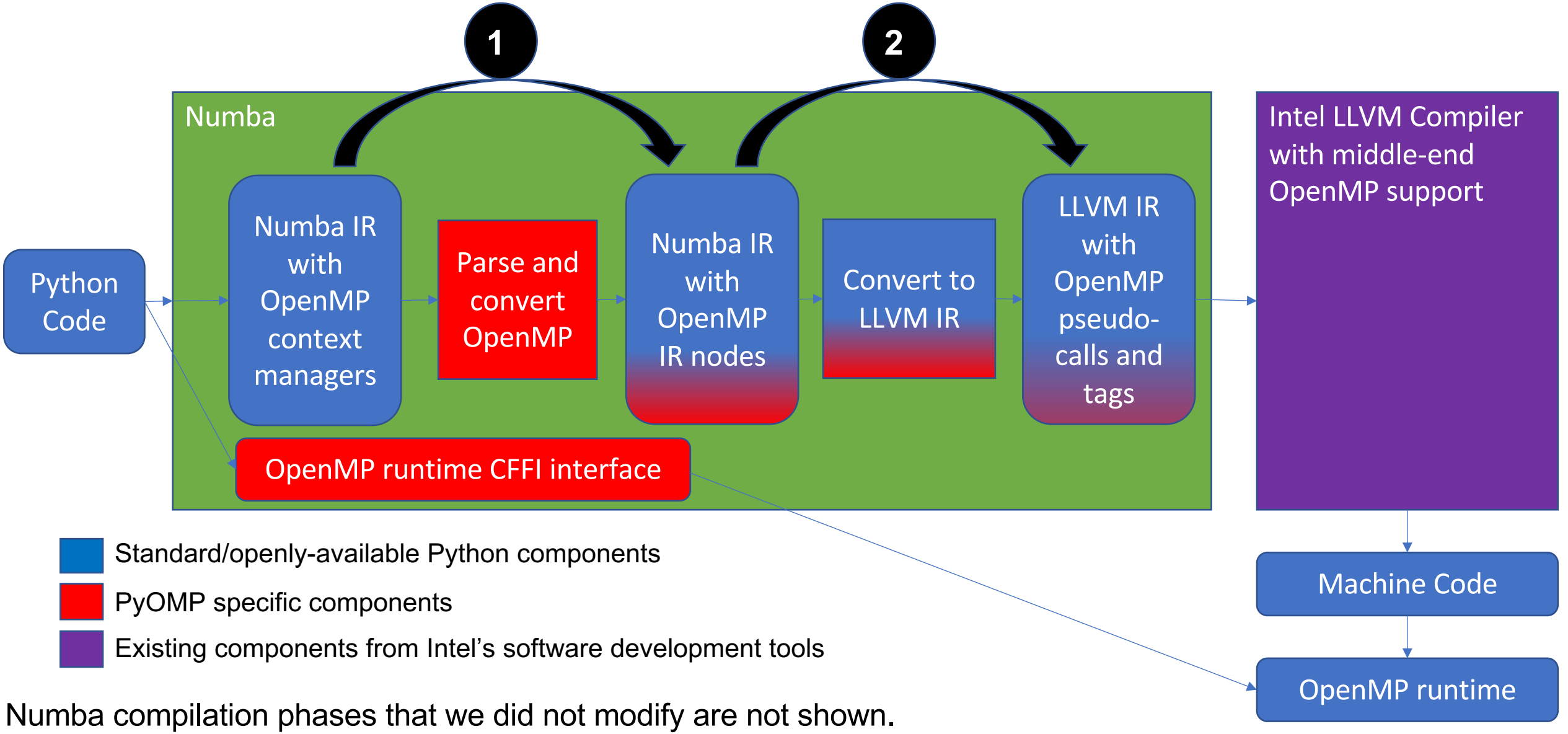
- Numerous options in numba … we are barely scratching the surface
  - @jit (nopython = true) tells the system to NOT use any python objects in the generated code. Can be much faster
  - @jit(parallel = true) invoke parallel accelerator

PyOMP Implementation in Numba: Overview



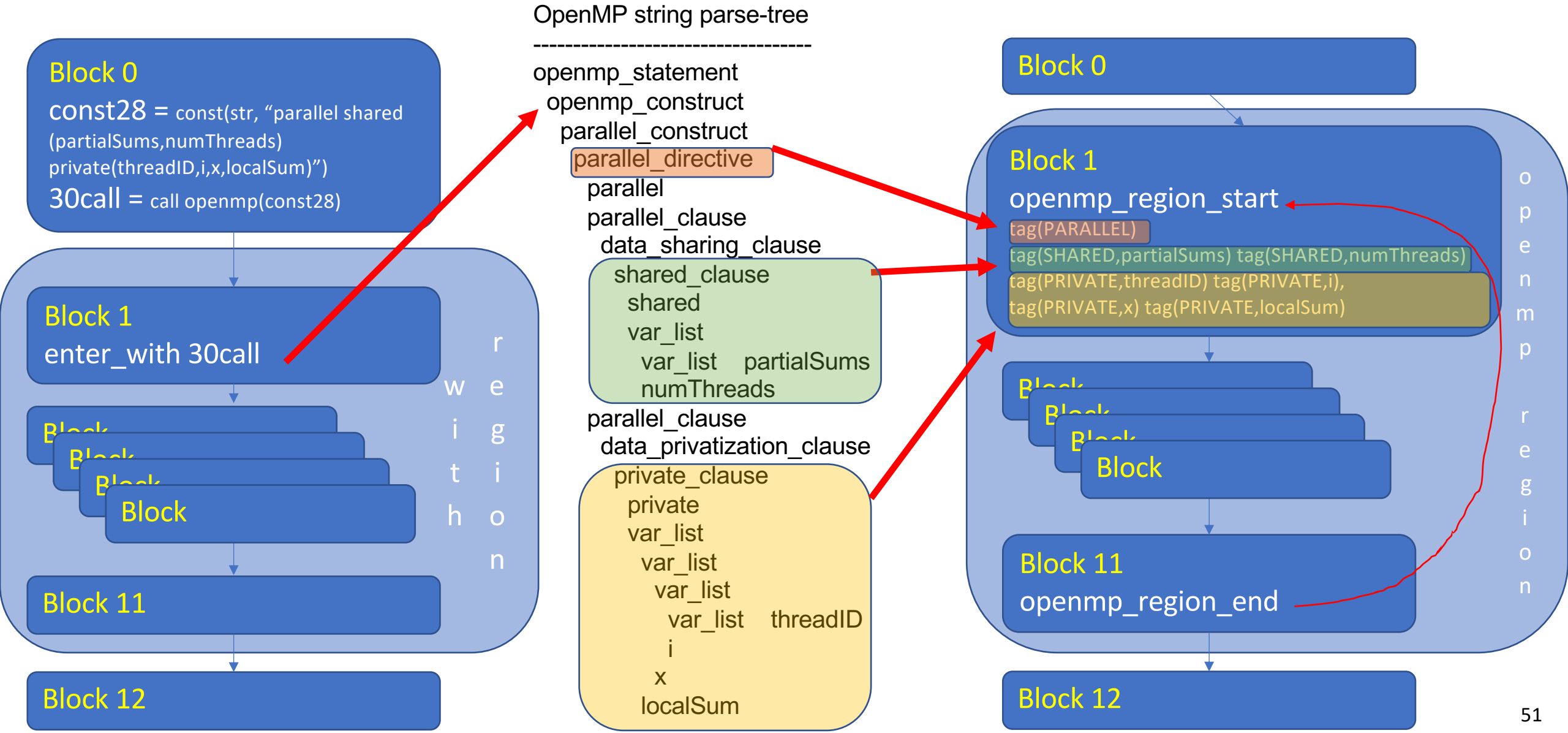
Numba compilation phases that we did not modify are not shown.

PyOMP Implementation in Numba: Overview



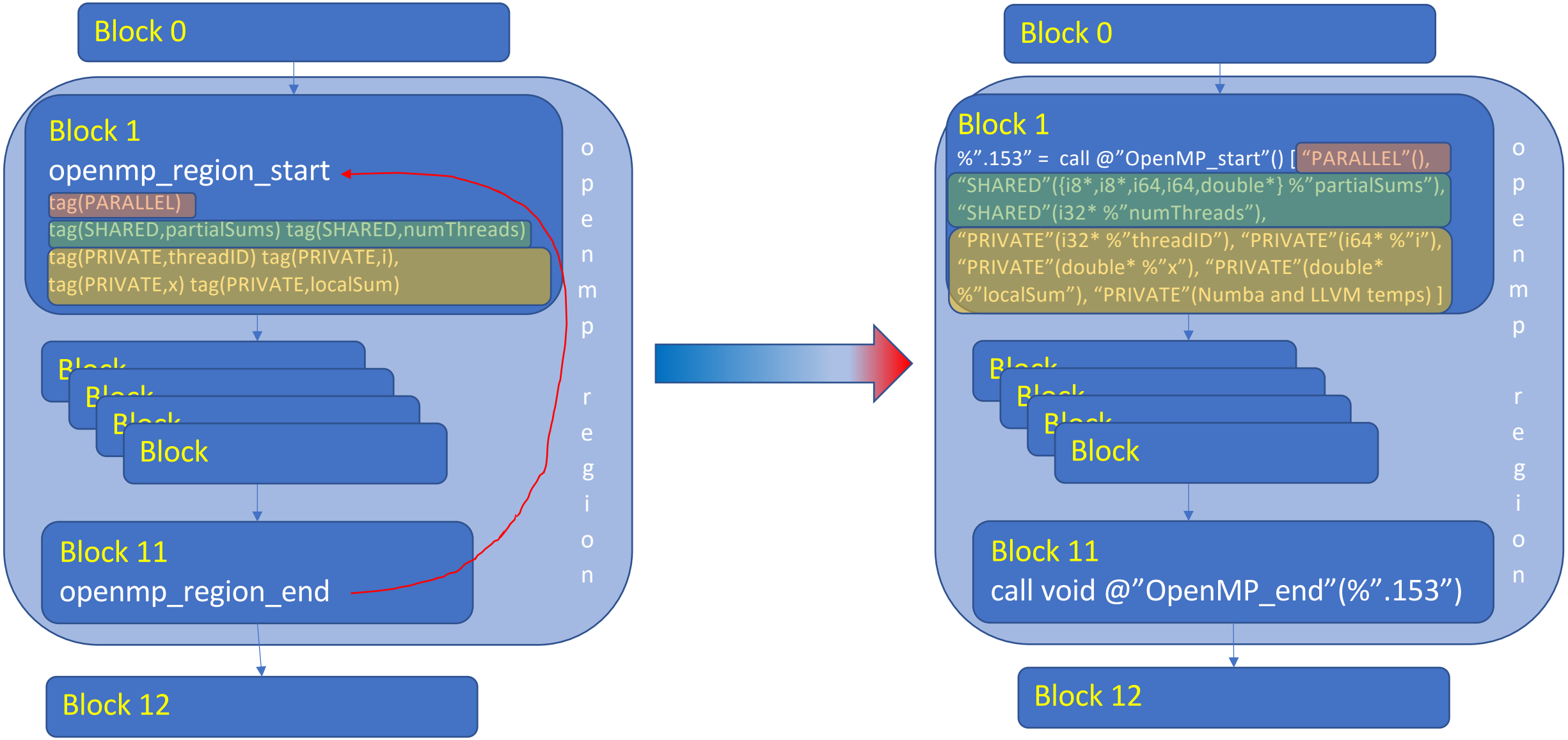
Numba compilation phases that we did not modify are not shown.

1 Numba OpenMP Context Managers to Numba IR Nodes



2

OpenMP Numba IR Nodes to LLVM IR



PyOMP Implementation Details

- 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.
- Loop structure rearranged to match Intel icx LLVM.
- Some OpenMP constructs (e.g., single, barrier, critical) require memory fences. The required memory fence encoded in OpenMP Numba IR node and PyOMP generates LLVM fences instructions for these nodes.

- To download the latest PyOMP

  conda install drtodd13::numba drtodd13::llvmlite -c conda-forge --override-channels

A brief digression to talk about performance issues in parallel programs

Consider performance of parallel programs

Compute N independent tasks on one processor

Load Data | Compute T_1 | … | Compute T_N | Consume Results

$$Time_{seq}(1) = T_{load} + N*T_{task} + T_{consume}$$

Compute N independent tasks with P processors

Load Data

Compute T_1

…

Compute T_N

Consume Results

Ideally Cut runtime by ~1/P

(Note: Parallelism only speeds-up the concurrent part)

$$Time_{par}(P) = T_{load} + (N/P)*T_{task} + T_{consume}$$

Talking about performance

- Speedup: the increased performance from running on P processors.

$$S(P) = \frac{Time_{seq}(1)}{Time_{par}(P)}$$

- Perfect Linear Speedup: happens when no parallel overhead and algorithm is 100% parallel.

$$S(P) = P$$

- Super-linear Speedup: typically due to cache effects … i.e. as P grows, aggregate cache size grows so more of the problem fits in cache

$$S(P) > P$$

Amdahl's Law

- What is the maximum speedup you can expect from a parallel program?

- Approximate the runtime as a part that can be sped up with additional processors and a part that is fundamentally serial.

$$Time_{par}(P) = (serial\_fraction + \frac{parallel\_fraction}{P}) * Time_{seq}$$

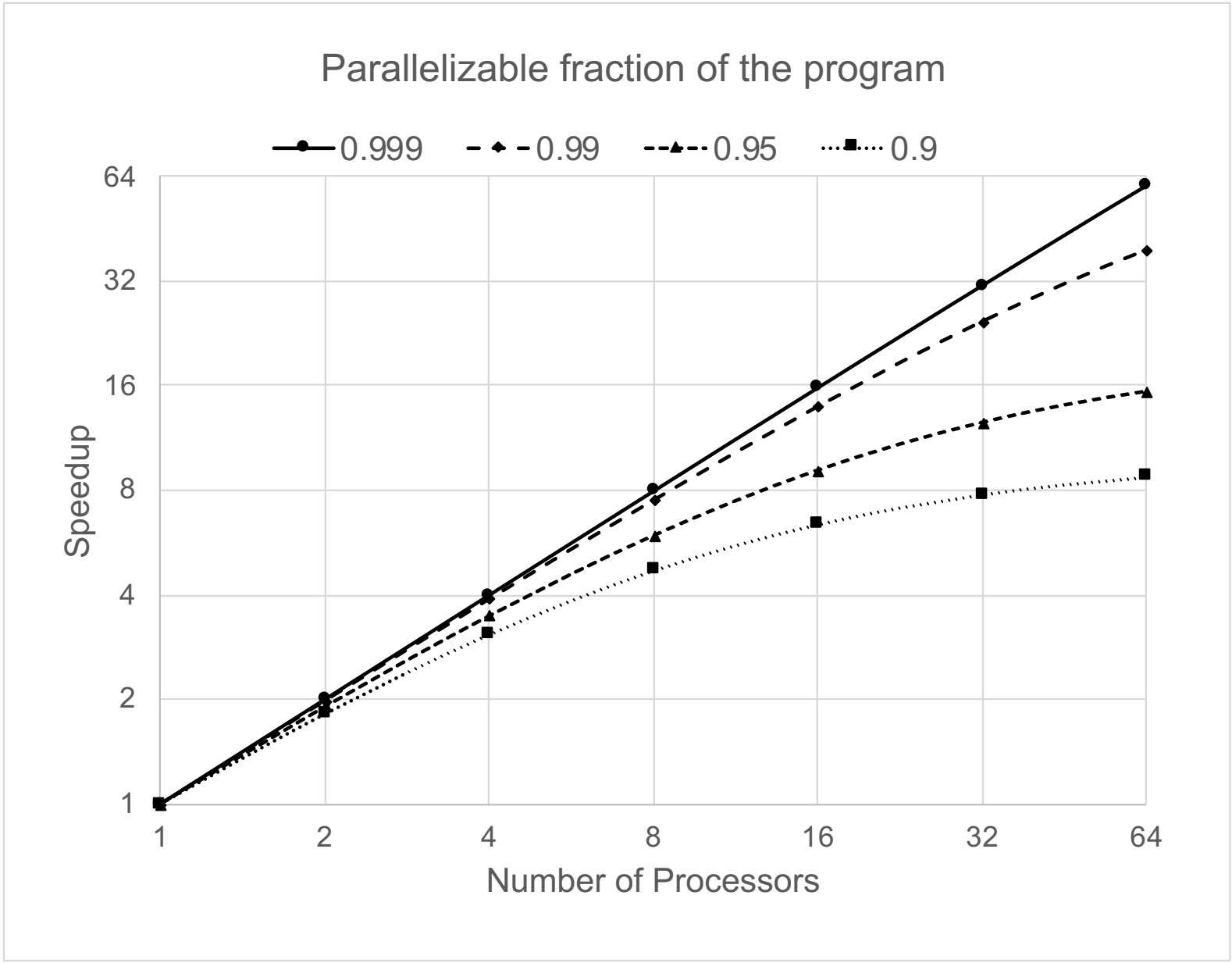
- If the serial fraction is α and the parallel fraction is (1- α) then the speedup is:

$$S(P) = \frac{Time_{seq}}{Time_{par}(P)} = \frac{Time_{seq}}{(\alpha + \frac{1-\alpha}{P}) * Time_{seq}} = \frac{1}{\alpha + \frac{1-\alpha}{P}}$$

- If you had an unlimited number of processors: $P \to \infty$

- The maximum possible speedup is: $S = \frac{1}{\alpha}$ ← Amdahl's Law

Amdahl's Law


Parallelizable fraction of the program
0.999
0.99
0.95
0.9
64
32
16
8
4
2
1
Speedup
1
2
4
8
16
32
64
Number of Processors

So now you should understand my silly introduction slide.

Introduction

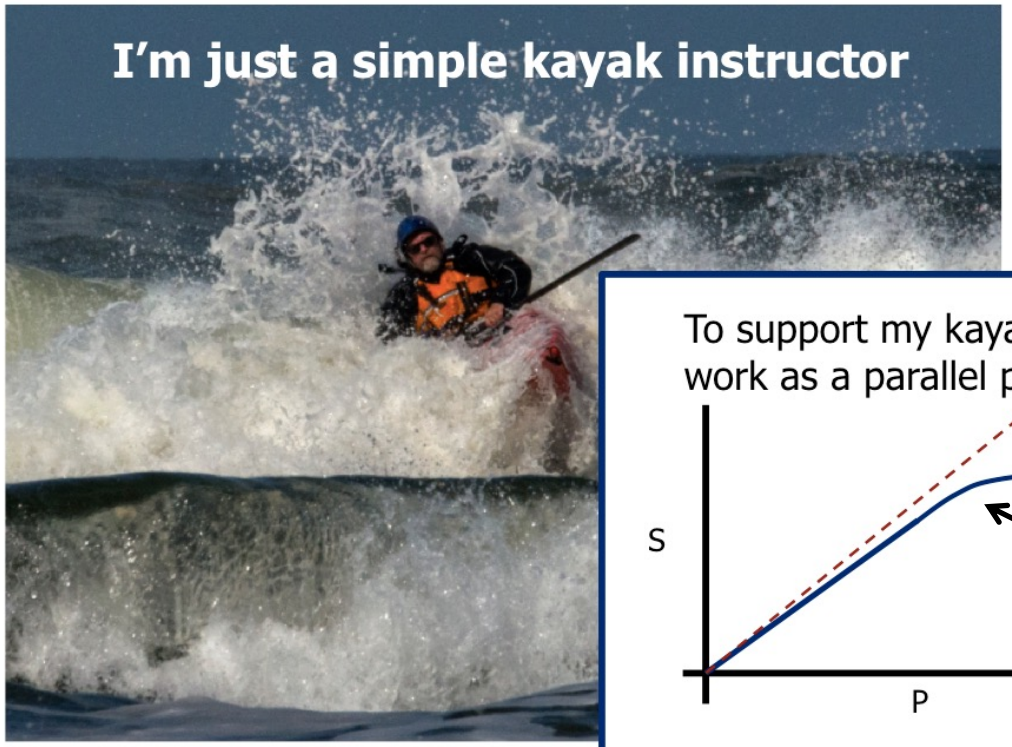


Photo © by Greg Clopton, 2014

To support my kayaking habit I work as a parallel programmer

Which means I know how to turn math into lines on a speedup plot

We measure our success as parallel programmers by how close we come to ideal linear speedup.

A good parallel programmer always figures out when you fall off the linear speedup curve and why that has occurred.

4

… So how good is our PyOMP performance?

The view of Python from an HPC perspective

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

We know better … the IKJ order is more cache friendly

And we picked a smaller problem

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

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

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

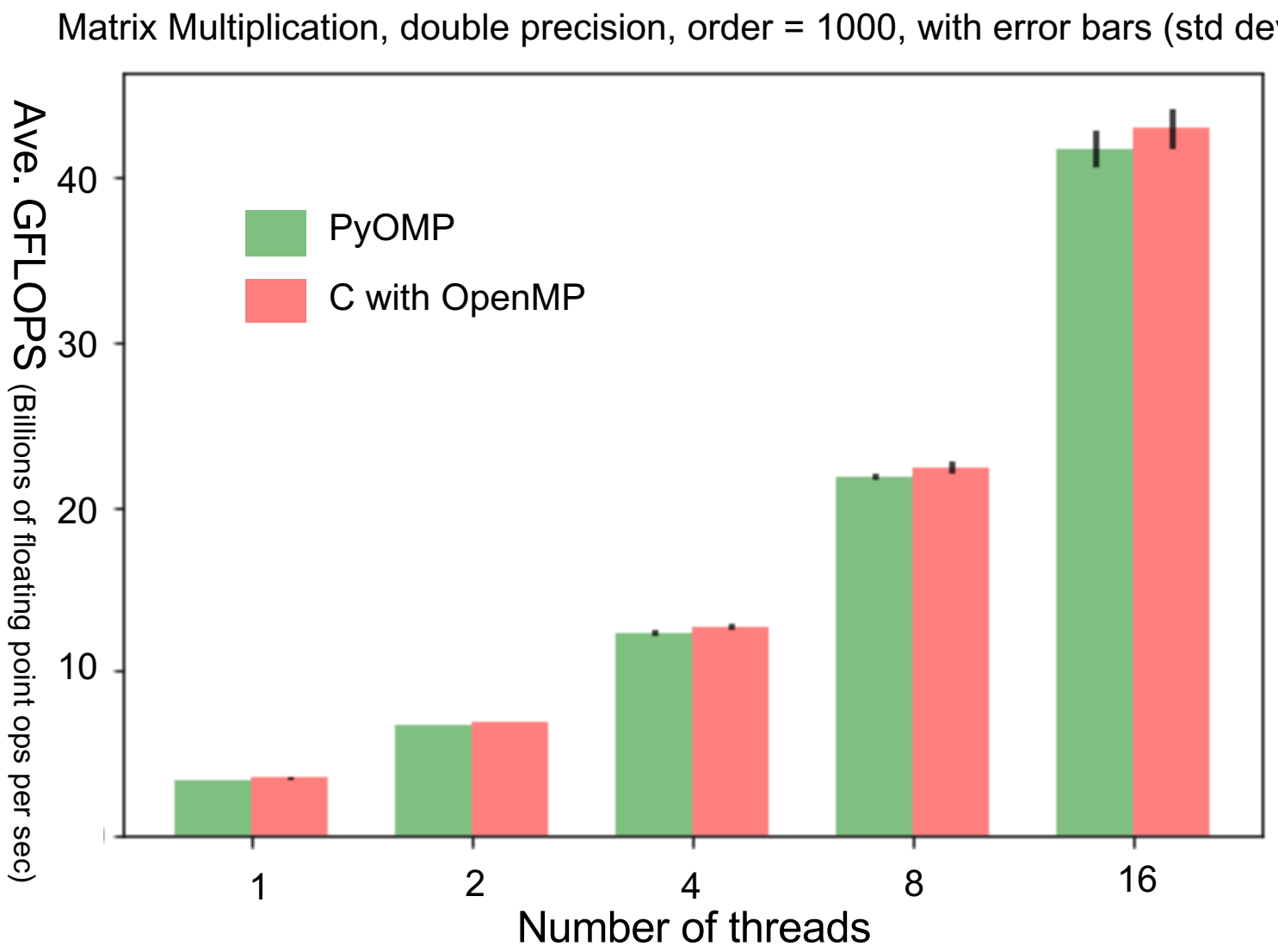
Amazon AWS c4.8xlarge spot instance, Intel® Xeon® E5-2666 v3 CPU, 2.9 Ghz, 18 core, 60 GB RAM

PyOMP DGEMM (Mat-Mul with double precision numbers)

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

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

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

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

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

dgemmTime = omp_get_wtime() - tInit

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

DGEMM PyOMP vs C-OpenMP



250 runs for order 1000 matrices

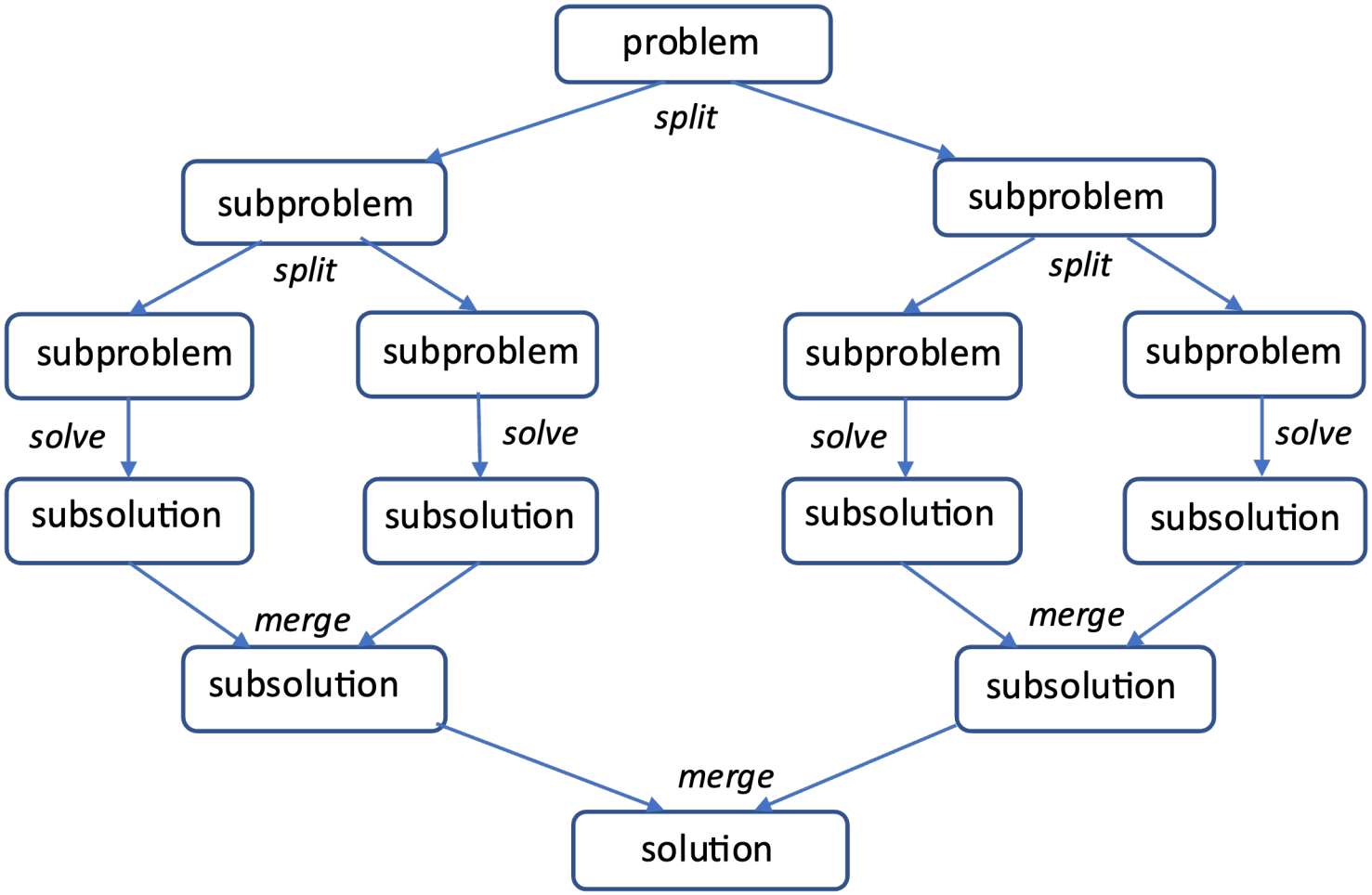
PyOMP times **DO NOT** include the one-time JIT cost of ~2 seconds.

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

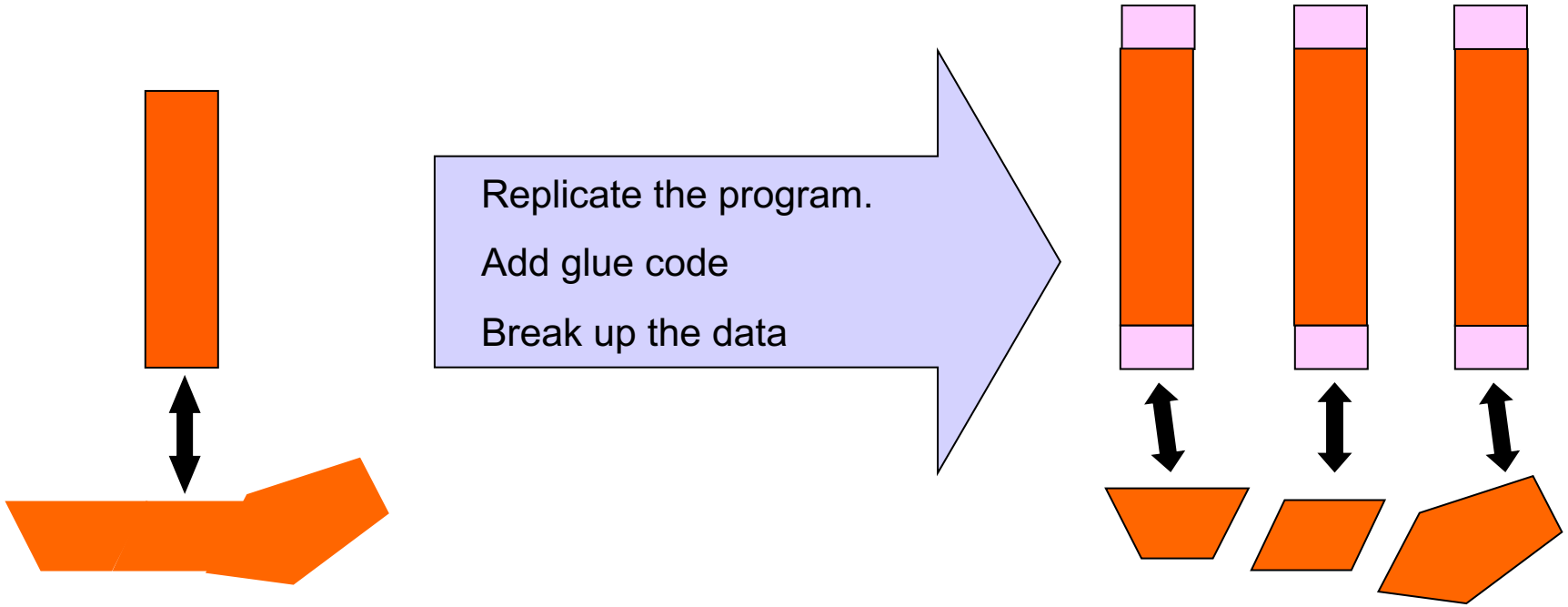


3 Options for parallelism:

- 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.



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.

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

A loop carried dependency

Recast to compute from i

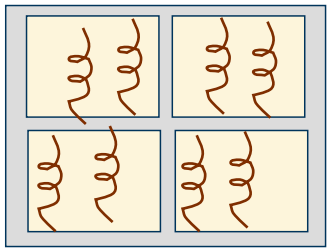
```
def piFunc(NumSteps):
    step=1.0/NumSteps
    sum = 0.0

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

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

Let’s return to our conversation about parallel hardware

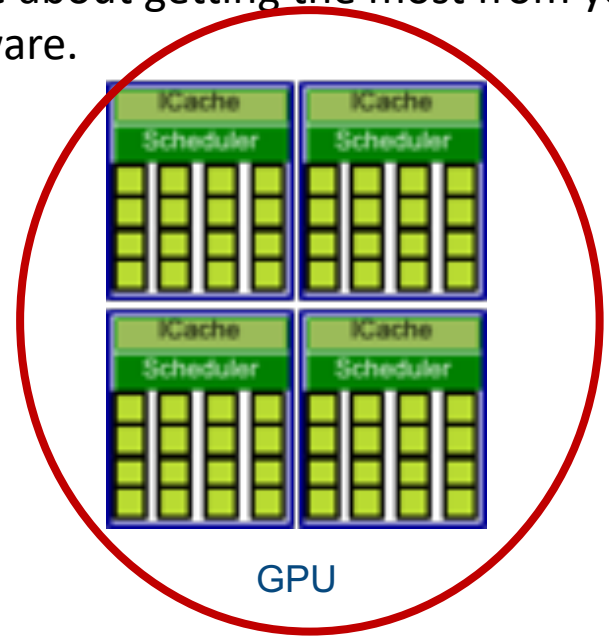
For hardware … parallelism is the path to performance

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.

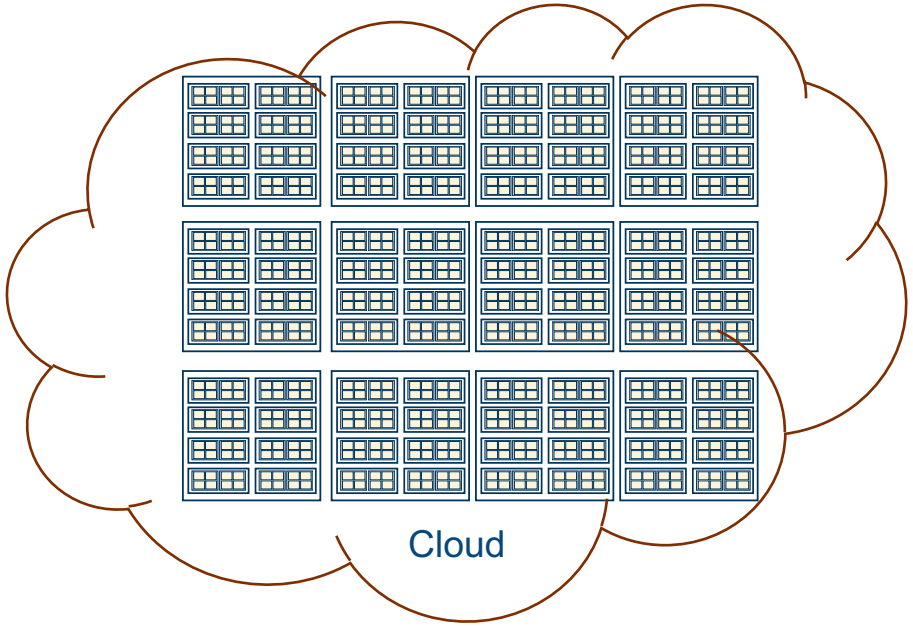


CPU

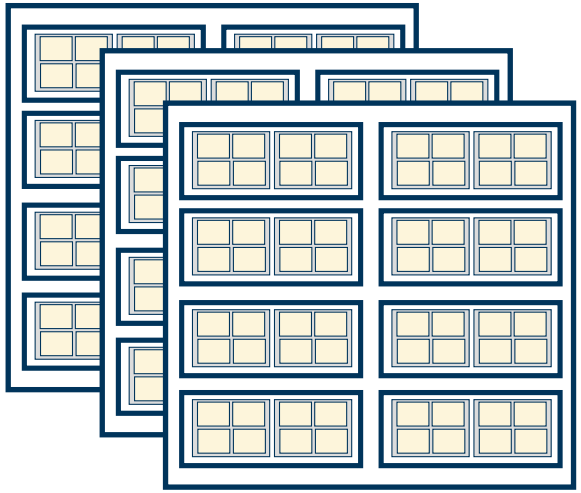
SIMD/Vector



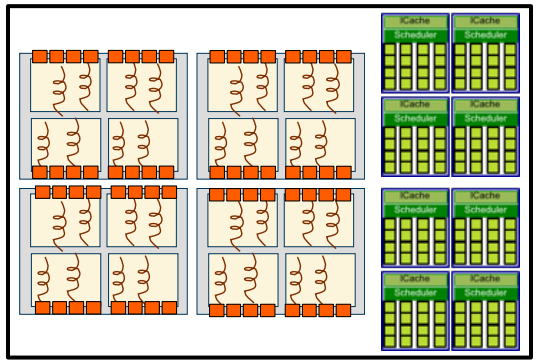
GPU



Cloud

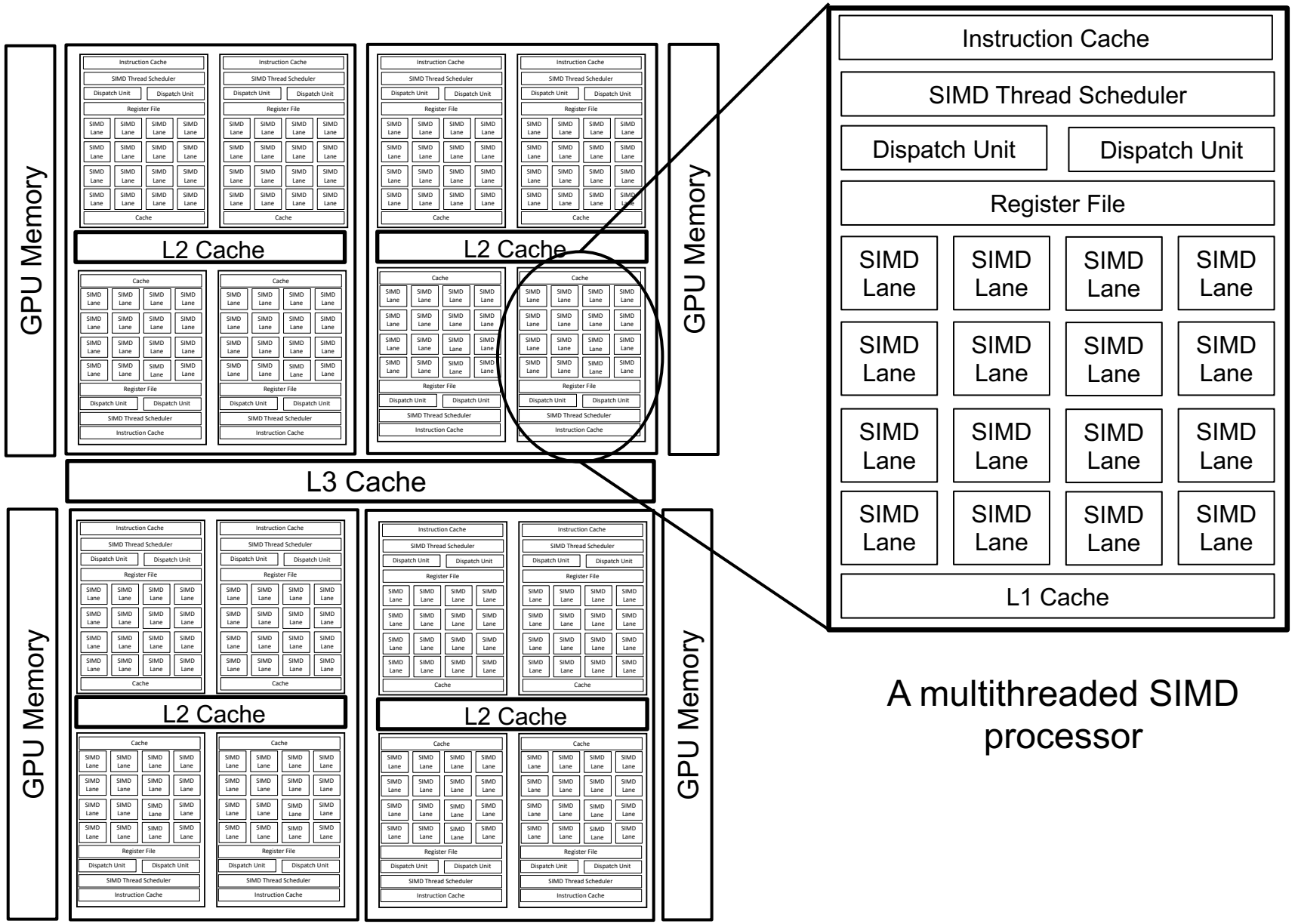


Cluster



Heterogeneous node

A Generic GPU (following Hennessey and Patterson)



A multithreaded SIMD processor

The "BIG idea" of GPU programming

Traditional loops

```
void
trad_mul(int n,
         const float *a,
         const float *b,
         float *c)
{
  int i;
  for (i=0; i<n; i++)
    c[i] = a[i] * b[i];
}
```

Data Parallel OpenCL

```
kernel void
dp_mul(global const float *a,
       global const float *b,
       global float *c)
{
  int id = get_global_id(0);

  c[id] = a[id] * b[id];

} // execute over "n" work-items
```

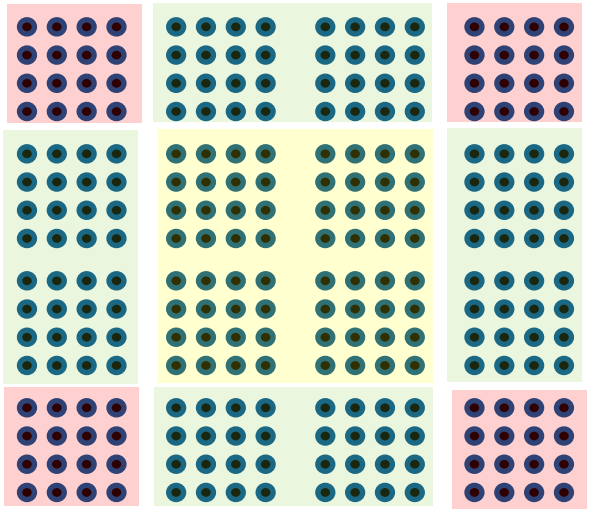
How do we execute code on a GPU: The SIMT model (Single Instruction Multiple Thread)

1. Turn source code into a scalar work-item

```
extern void reduce(   __local  float*,   __global float*);

__kernel void pi(  const int niters, float  step_size,
       __local  float*  l_sums,  __global float*  p_sums)
{
  int n_wrk_items  = get_local_size(0);
  int loc_id       = get_local_id(0);
  int grp_id   = get_group_id(0);
  float x, accum = 0.0f;    int i,istart,iend;

  istart =   (grp_id * n_wrk_items   + loc_id) * niters;
  iend   = istart+niters;

  for(i= istart; i<iend; i++){
     x = (i+0.5f)*step_size;    accum += 4.0f/(1.0f+x*x); }

  l_sums[local_id] = accum;
  barrier(CLK_LOCAL_MEM_FENCE);
  reduce(l_sums, p_sums);
}
```

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

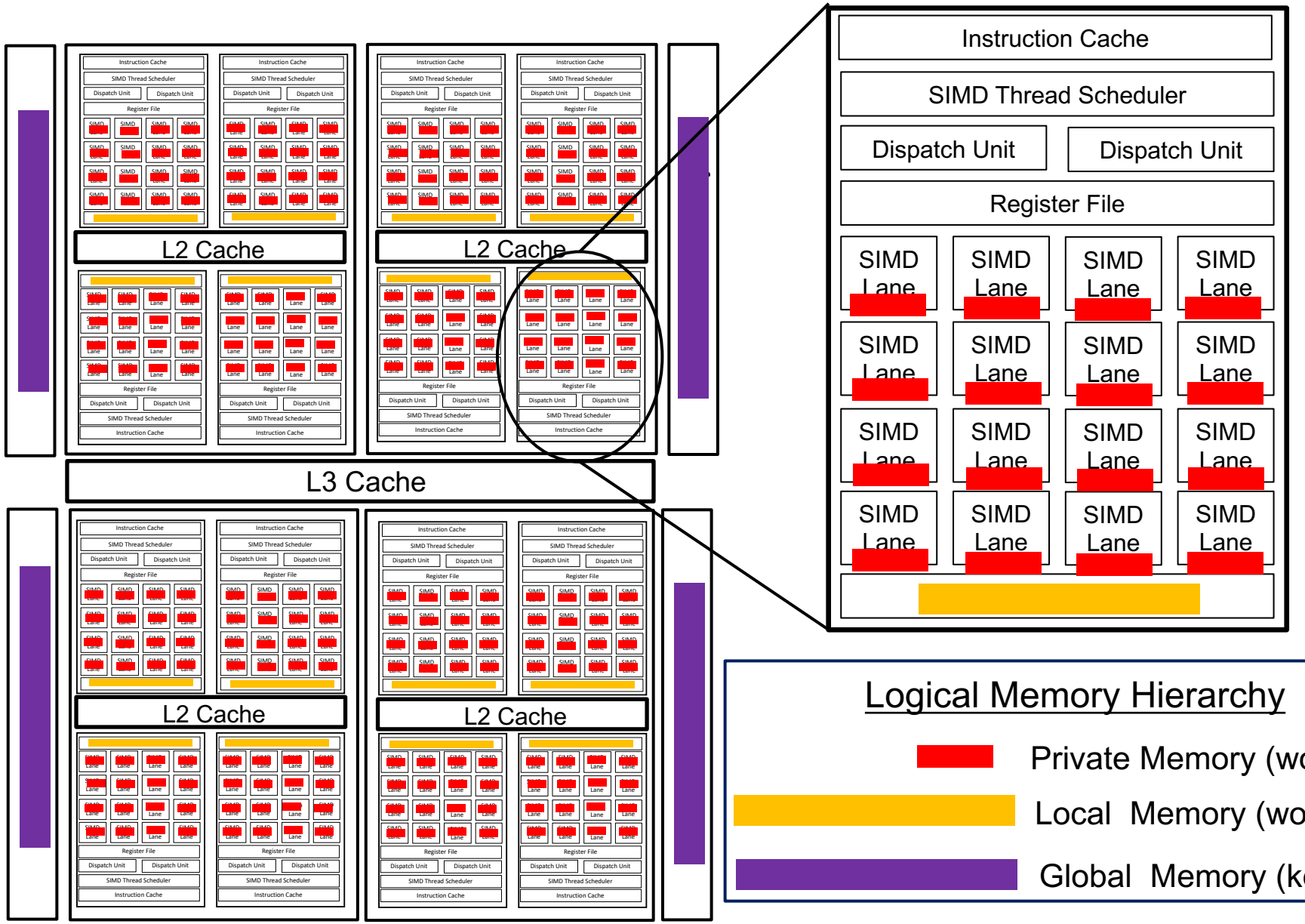
2. Map work-items onto an N dim index space.



3. Map data structures onto the same index space

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

A Generic GPU (following Hennessey and Patterson)



GPU terminology is Broken (sorry about that)

Hennessy and Patterson	CUDA	OpenCL
Multithreaded SIMD Processor	Streaming multiprocessor	Compute Unit
SIMD Thead Scheduler	Warp Scheduler	Work-group scheduler
SIMD Lane	CUDA Core	Processing Element
GPU Memory	Global Memory	Global Memory
Private Memory	Local Memory	Private Memory
Local Memory	Shared Memory	Local Memory
Vectorizable Loop	Grid	NDRange
Sequence of SIMD Lane operations	CUDA Thread	work-item
A thread of SIMD instructions	Warp	sub-group

How do we execute code on a GPU: OpenCL and CUDA nomenclature

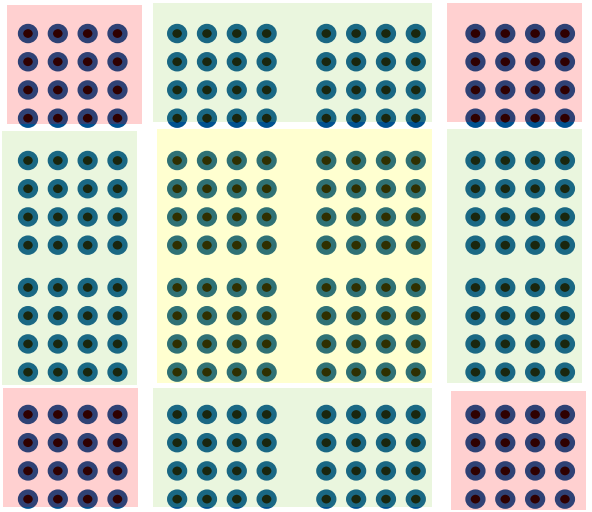
Turn source code into a scalar **work-item** (a CUDA **thread**)

```
extern void reduce(   __local  float*,   __global float*);

__kernel void pi(  const int niters, float  step_size,
       __local  float*  l_sums,  __global float*  p_sums)
{
  int n_wrk_items  = get_local_size(0);
  int loc_id       = get_local_id(0);
  int grp_id   = get_group_id(0);
  float x, accum = 0.0f;    int i,istart,iend;

  istart =   (grp_id * n_wrk_items   + loc_id) * niters;
  iend   = istart+niters;

  for(i= istart; i<iend; i++){
     x = (i+0.5f)*step_size;    accum += 4.0f/(1.0f+x*x); }

  l_sums[local_id] = accum;
  barrier(CLK_LOCAL_MEM_FENCE);
  reduce(l_sums, p_sums);
}
```

This code defines a **kernel**

Submit a kernel to an OpenCL **command queue** or a CUDA **stream**

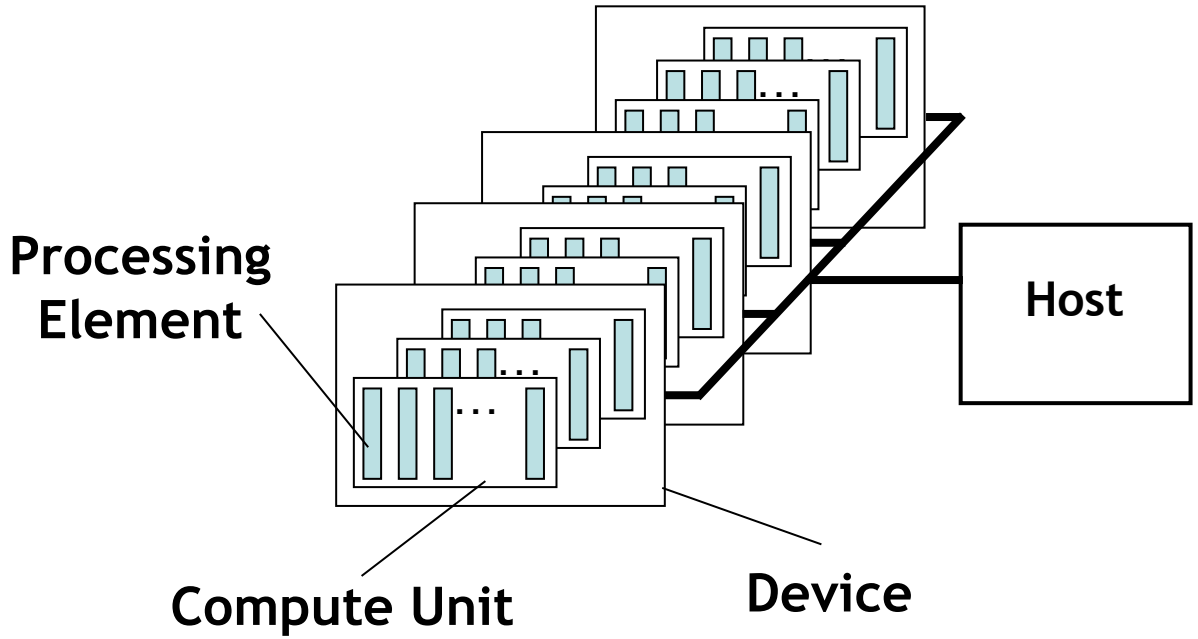
Organize work-items into **work-groups** and map onto an an N dim index space. CUDA calls a work-group a **thread-block**



OpenCL index space is called an **NDRange.** CUDA calls this a **Grid**

It's called SIMT, but GPUs are really vector-architectures with a block of work-items executing together (a **subgroup** in OpenCL or a **warp** with CUDA)

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***

Loop Parallelism code naturally maps onto the GPU

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

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

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

    pi = step*sum
    return pi

pi = piFunc(100000000)
```

OpenMP constructs managed through the *with* context manager.

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

- **target**: map execution from the host onto the device
- **teams loop**: Map kernel instances onto PEs inside the compute units
- **private(x)**: each threads gets its own x
- **reduction(+:x)**: combine x from each work-item using +

Note: We are still implementing the target constructs in PyOMP. Should be done by beginning of 2023

PE: processing element

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

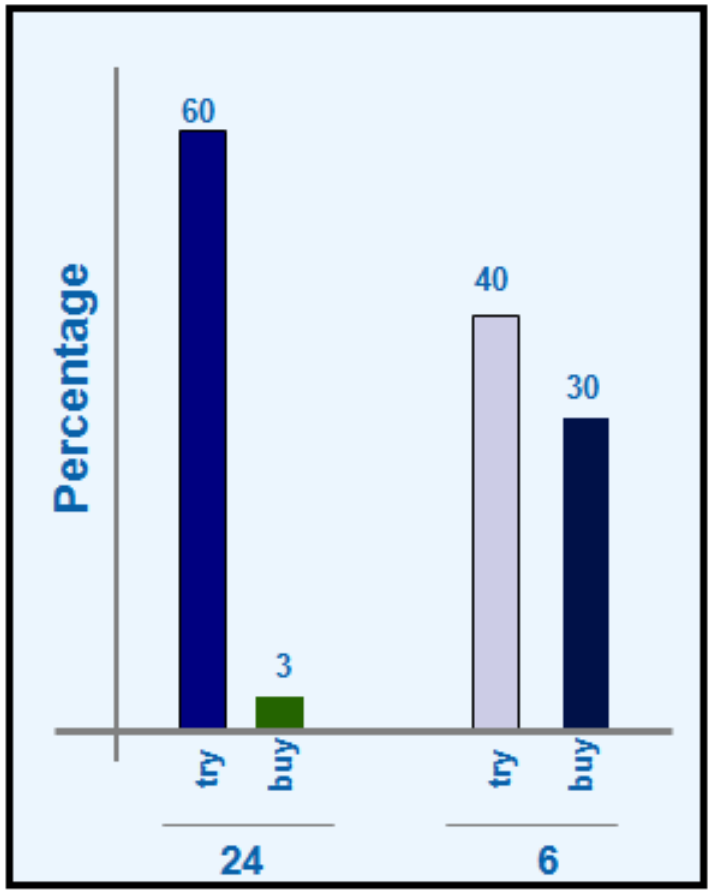
ABCPL
ACE
ACT++
Active messages
Adl
Adsmith
ADDAP
AFAPI
ALWAN
AM
AMDC
AppLeS
Amoeba
ARTS
Athapascan-0b
Aurora
Automap
bb_threads
Blaze
BSP
BlockComm
C*.
"C* in C
C**
CarlOS
Cashmere
C4
CC++
Chu
Charlotte
Charm
Charm++
Cid
Cilk
CM-Fortran
Converse
Code
COOL
CORRELATE
CPS
CRL
CSP
Cthreads
CUMULVS
DAGGER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

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
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.

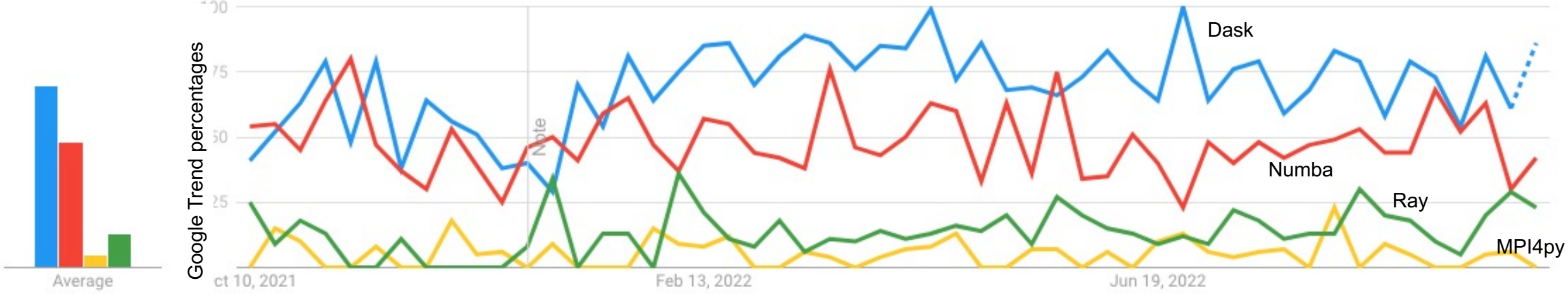
That would allow vendors to carry out HW/SW optimization and focus on quality over ”chasing fads”.

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

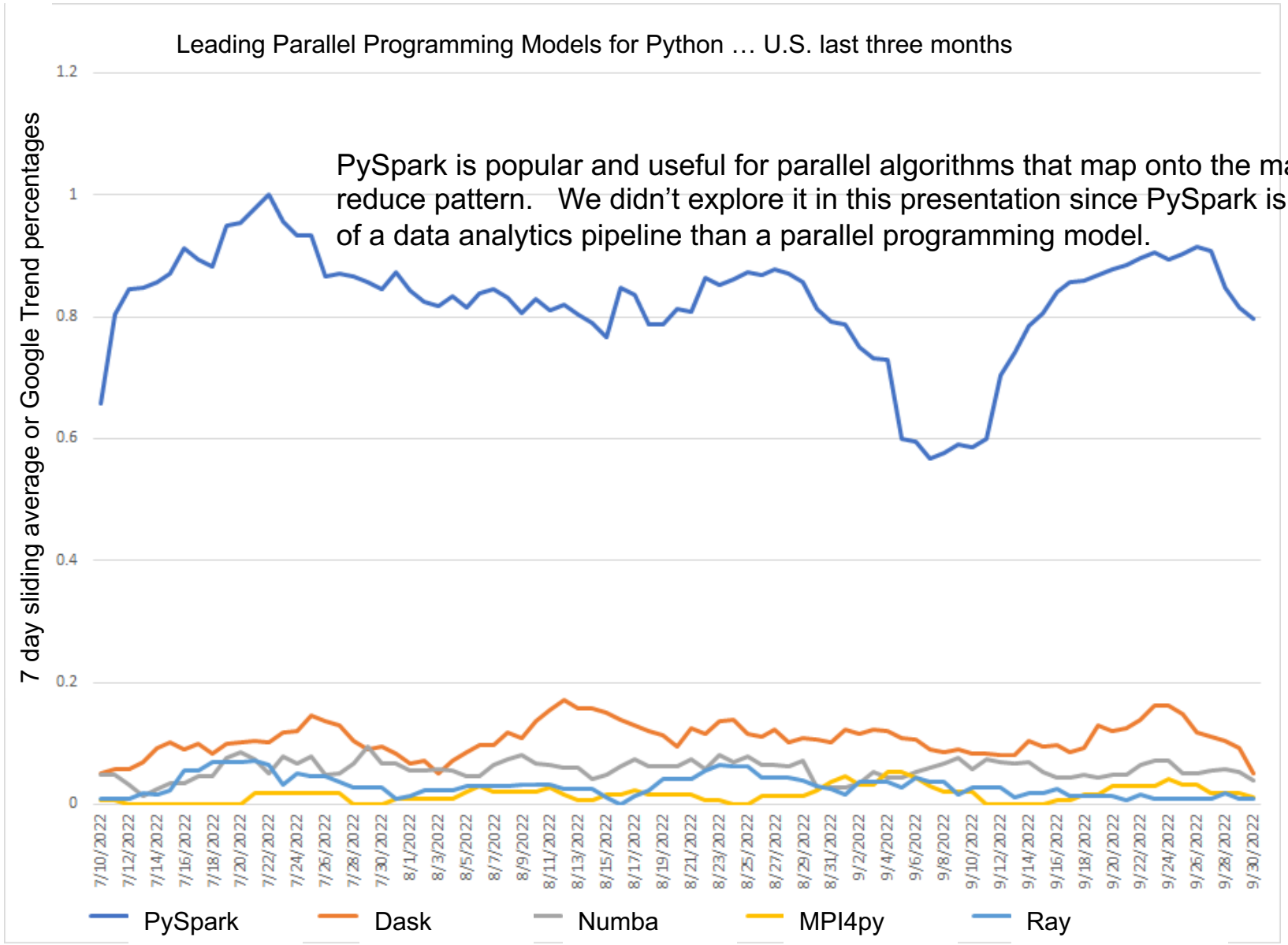
Popular python parallel Programming models

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



Popular python parallel Programming models



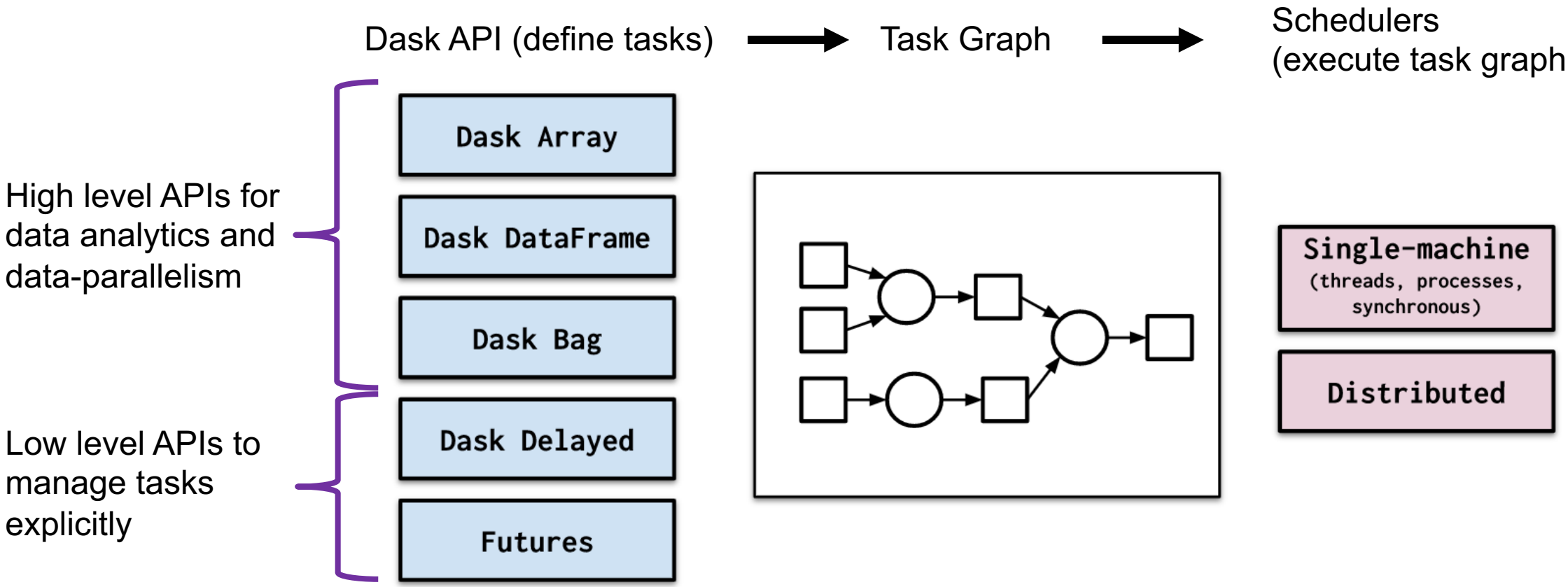
We compared many python parallel programming models with google-trends (which tracks web searches)

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)



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:

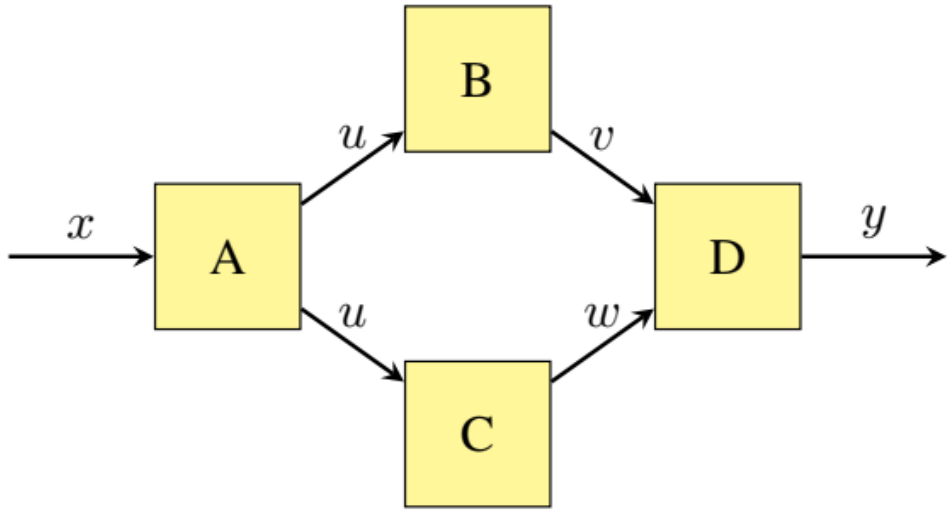
```
v = 2
for x in range(5):
    v = add_one(v)
v = v.compute()
```

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

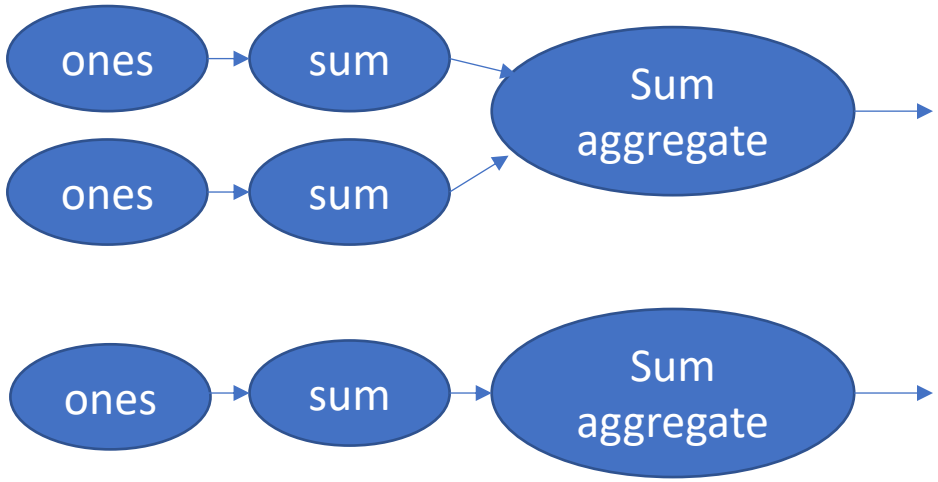
- Note: no decorator, explicit job submission
- Can pass futures as parameters to chain functions/construct DAGs

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

- 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.

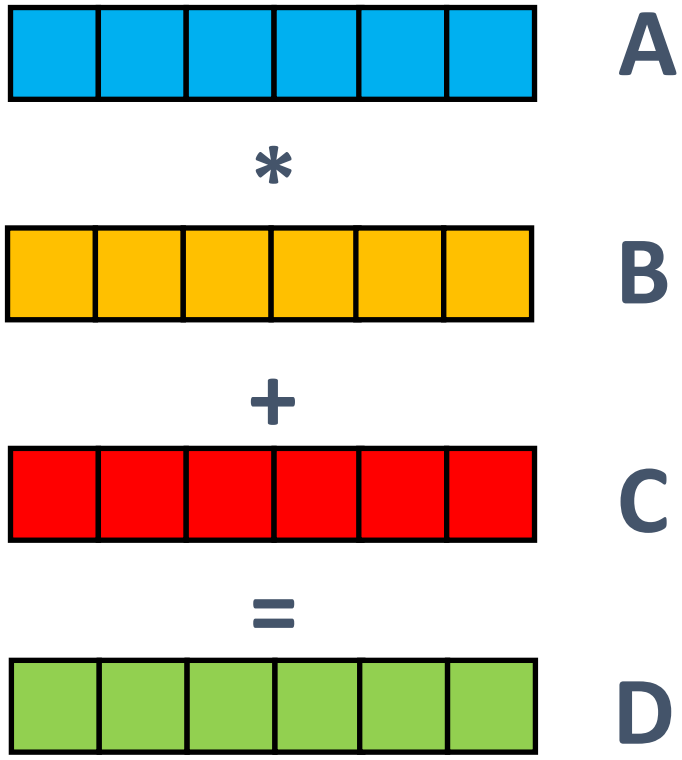
The Data Parallelism design pattern … the parallelism is expressed through the data .. Typically as functions applied independently to the elements of data structures combined with collective ops (such as dot products).

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.

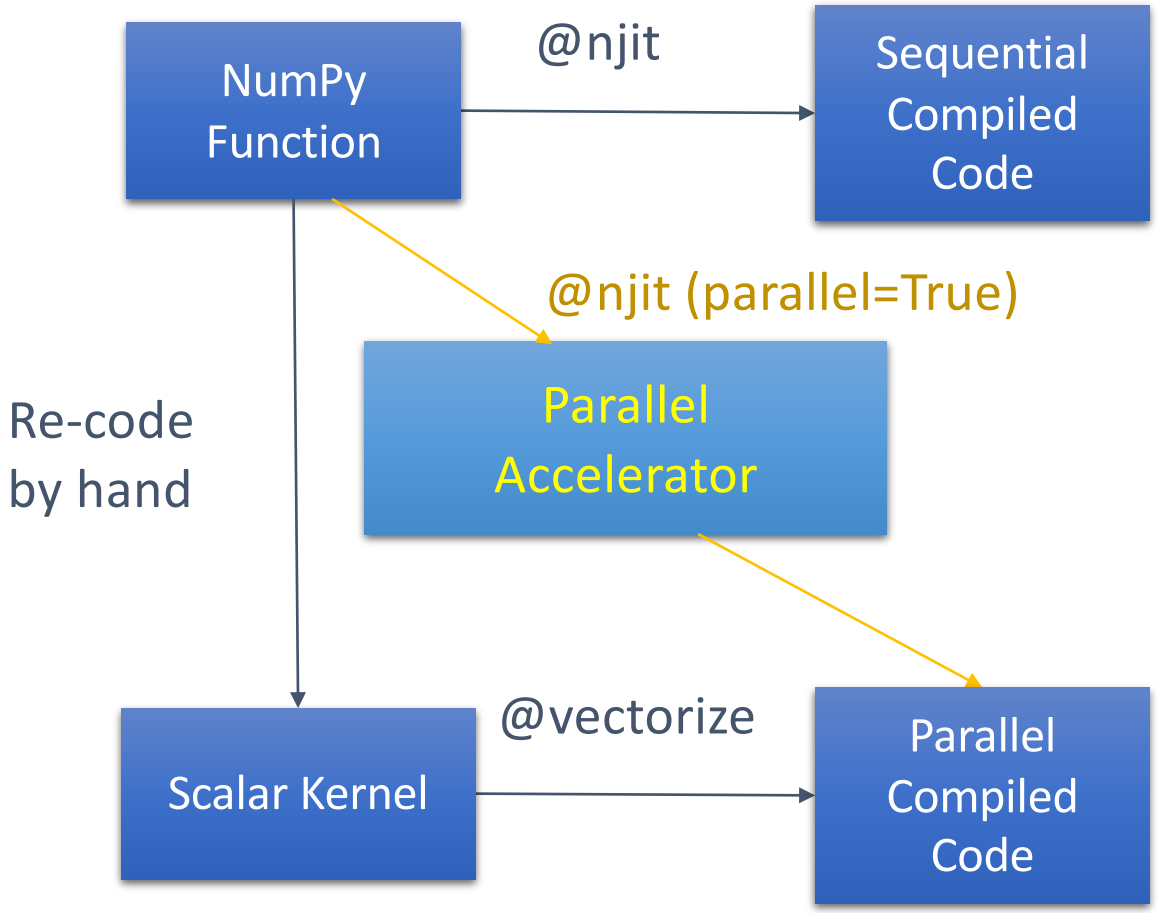
Running Parfors in Parallel

- 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

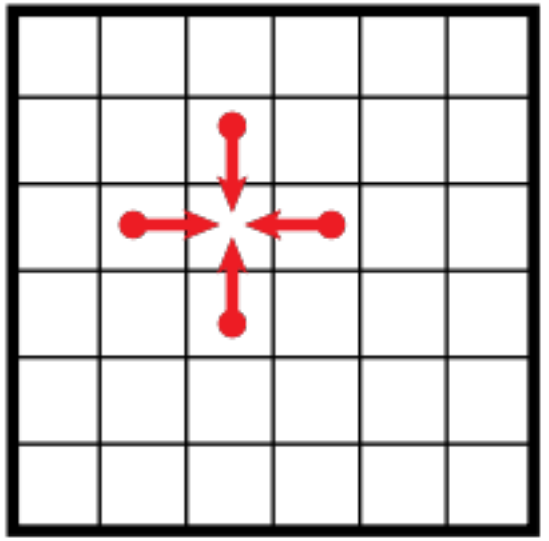
The following patterns are recognized by ParallelAccelerator for parallel execution:

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

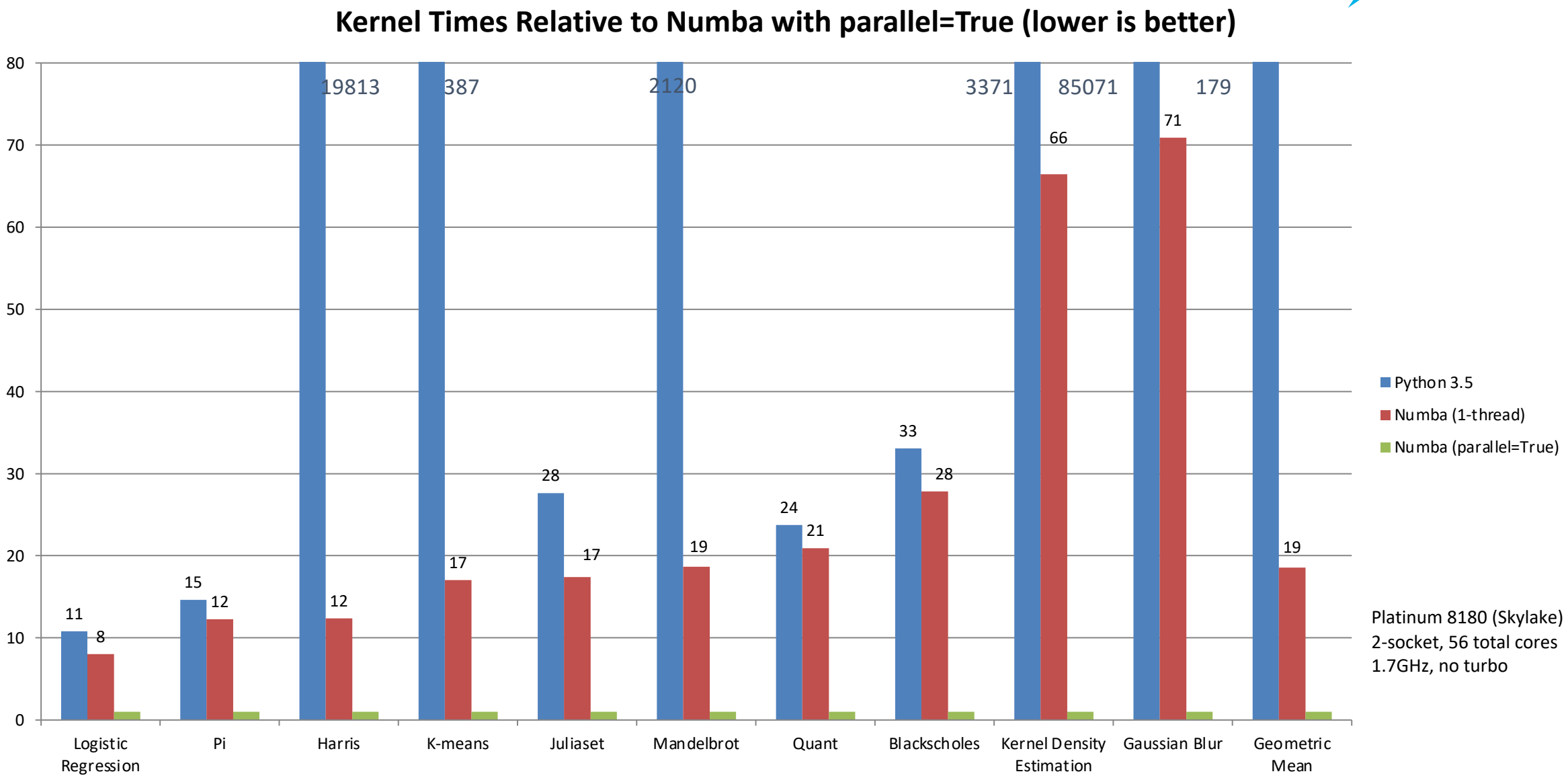
Other ParallelAccelerator Technology

- 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



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.

Ray

Intro to Ray

- Provides simple API to use multiple processes
  - Get around GIL limits, utilize multi-core, multiple nodes
- Remote function abstraction
- Remote stateful objects (actors model)
- Executes as tasks dispatched to Ray “cluster” (set of Ray worker processes on one or more nodes)
- Enables construction of large computation graphs (function composition)

Ray API – remote functions

- Define a remote function:

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

Decorator turns normal Python function into Ray remote function

- Calling remote function, getting results:

```
futurevalue = add_one.remote(7)
v = ray.get(futurevalue)
```

Returns immediately after creating task to run our function

Returns value 8 after about 1 second when task completes

Ray – parallel and chaining calls

- Parallel execution:

```
fv = [add_one.remote(i) for i in range(5)]
v = ray.get(fv)
```

Returns immediately

Returns list [1,2,3,4,5]
after about 1 second

- Chained execution:

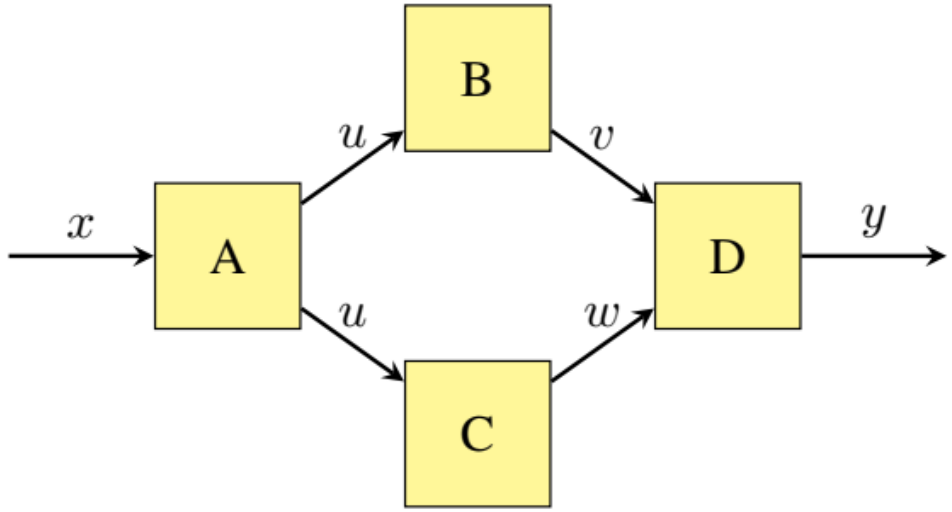
```
v = 2
for x in range(5):
    v = add_one.remote(v)
v = ray.get(v)
```

These return immediately

Returns value 7 after
about 5 seconds

Chaining forms DAGs of Tasks

```
# A, B, C, and D are remote
functions
fut_u = A.remote(x)
fut_v = B.remote(fut_u)
fut_w = C.remote(fut_u)
fut_y = D.remote(fut_v, fut_w)
y = ray.get(fut_y)
```



Ray API – “actor model”

- Stateful remote class instances:

```
@ray.remote
class foo:
        def __init__(self, v):
                self.val = v
        def calc(self, x):
                time.sleep(1)
                self.val, x = x, self.val+x
                return x
```

- Instantiation, calls:

```
robj = foo.remote(3)
fv = robj.calc.remote(7)
v = ray.get(fv)
```

Creates remote instance of class
robj is a reference – can only make remote calls

Call, get return value just as with remote functions

Ray Stateful Objects, cont'd.

- Remote class instances allowed to make remote calls to functions and other class instances
- Deterministic execution of an instance's methods
  - FIFO scheduling
  - Non-reentrant at class instance level

```
fv =[robj.calc.remote(x) for x in range(5)]
v = ray.get(fv)
```

Will execute sequentially not in parallel

Returns after 5 seconds

  - Normal recursion not possible
  - However, can queue up execution of methods to run after current method returns (e.g., tail recursion)

Loop parallelism with Tasks on Ray


```
import numpy as np
import ray
ray.init()

@ray.remote
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
    vals = []
    for i in range(NumTasks):
        nstart = (i*NumSteps)//NumTasks
        nstop = ((i+1)*NumSteps)//NumTasks
        vals.append(
            calc_pi.remote(nstart, nstop, step) )
    return step * sum(ray.get(vals))

pi = piFunc(100000000, 100)
```

Initialize ray workers on local machine; can provide options for connecting to cluster

Calculate over part of the range;
This is in Numpy vector-style code, which is 6-7x faster than Python loops, but incur space overheads for arrays;
Could call Numba function from here, too

Start NumTasks tasks

Wait for and get results of all of the tasks

Divide and conquer code on Ray

```
import numpy as np
import ray
ray.init()

MIN_BLK = 1024*256

@ray.remote
def calc_pi(nstart, nstop, step):
    iblk = nstop-nstart
    if iblk<MIN_BLK:
        start = (nstart+0.5)*step
        stop = (nstop-0.5)*step
        X = np.linspace(start, stop, num=iblk)
        Y = 4.0 / (1.0 + X*X)
        s = np.sum(Y)
    else:
        s1 = calc_pi.remote(nstart, nstart+iblk//2, step)
        s2 = calc_pi.remote(nstart+iblk//2, nstop, step)
        s = ray.get(s1)+ray.get(s2)
    return s

def piFunc(NumSteps):
    step = 1.0/NumSteps
    val = calc_pi.remote(0, NumSteps, step)
    return step*ray.get(val)

pi = piFunc(100000000)
```

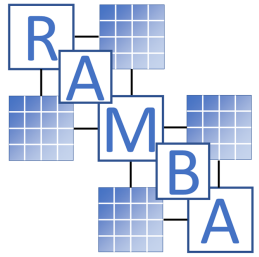
If range small enough, just do the computation

Otherwise, start 2 tasks, each computing over half the range;
Starting tasks from other tasks is fine

Waiting for results from within task temporarily relinquishes resources, so will not deadlock on cluster resources

ramba

Ramba



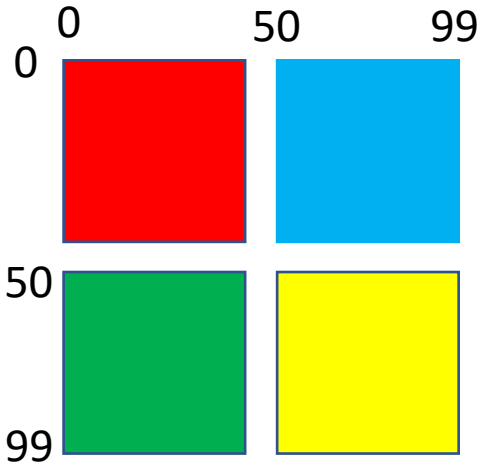
- Key idea: numpy, but compiled and distributed (running on Ray or MPI)
- Goal: write NumPy vector-style code, but execute in parallel (threads, processes, and distributed)

- Introduces a distributed array data structure
  - Looks like a NumPy array, but is partitioned across processes/nodes
- Preserve NumPy-like operations, API:
  - Basic per-element arithmetic operations
  - Simple reductions
  - Array slicing / views
- Extended API provides “skeletons” that represent common programming patterns (e.g., map)

The main use case is as a fast, drop-in replacement for NumPy, allowing a Python programmer to make use of multiple threads, processes, and nodes with little to no code changes.

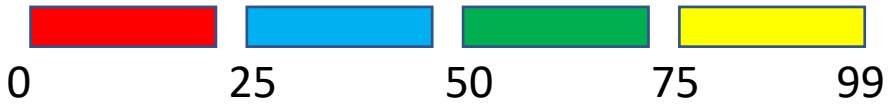
Ramba array examples (4 workers)

- A=ramba.zeros((100,100))



Create 50x50 array on each worker; init to 0

- B=ramba.ones(100)

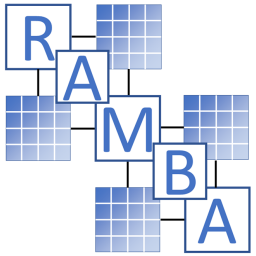


Create array size 25 on each worker; init to 1

- B[20:60] += 4

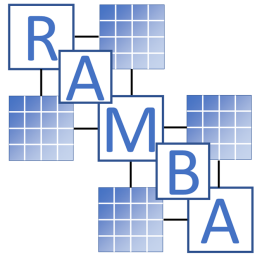
Temporary ndarray provides mutable view mapped to the arrays on a subset of workers; execute in-place add of 4

Deferred Operations



- Ndarray operations do not immediately trigger execution on remote workers
- Instead, the operations are queued up, forming a DAG of operations
- Program continues, adding more operations
- Operations that return values (rather than ndarrays) and explicit sync() trigger execution
  - DAG traversed to determine set of operations to execute
- Operations are fused into a single function / loop if possible
  - Compatible operations have same amount of work on each worker
  - Write-after-write and read-after-write conflicts handled by not fusing, adding temporary arrays
  - Numba-JIT-compiled code is generated for each fused set, use parallel when possible

Deferred Ops Example



- A += B + 5*C
- Ramba deferred ops:
  - All are fused into single loop
  - Temp arrays not materialized
  - Single call to workers, single traversal of arrays

- Standard Numpy execution:
  - Do 5*C, store in tmp1
  - Do B+tmp1, store in tmp2
  - Do A+=tmp2
  - Uses extra storage for temporaries
  - Traverses arrays multiple times

Generated code

```
@numba.njit(parallel=True)
def ramba_deferred_ops_func_48400739313330594710(global_start, ramba_tmp_var_00002,
                    ramba_tmp_var_00004, ramba_tmp_var_00005, ramba_tmp_var_00001):
    itershape = ramba_tmp_var_00002.shape
    for index in numba.pndindex(itershape):
        ramba_tmp_var_00000 = ramba_tmp_var_00001 * ramba_tmp_var_00002[index]
        ramba_tmp_var_00003 = ramba_tmp_var_00004[index] + ramba_tmp_var_00000
        ramba_tmp_var_00005[index] += ramba_tmp_var_00003
```

Array parallel code on Ramba

```
import ramba as np

def calc_pi(nsteps):
    step = 1.0/nsteps
    X = np.linspace( step*0.5, 1.0-step*0.5,
                                    num=nsteps )
    Y = 4.0*step / (1.0 + X*X)
    return np.sum(Y, asarray=True)

def piFunc(nsteps):
    pi_arr = calc_pi(nsteps)
    return pi_arr[0]

pi = piFunc(100000000)
```

Calculation written in Numpy "vector-style" code

asarray flag ensures return value from sum is an array (with one element)
This allows the sum and computation of Y to be fused, and to defer execution

Actual execution is triggered here
At this point X and Y are out of scope, so no array is materialized
Execution can occur over multiple threads and processes, on multiple nodes

Avoiding materialization of temporaries

```
A = ramba.arange(1000)
B = A*0.05
result = B + 25
ramba.sync()
```

This results in construction of 3 arrays: A, B, result

```
def myFunc()
    A = ramba.arange(1000)
    B = A*0.05
    return B + 25
result = myFunc()
ramba.sync()
```

By the time execution happens, A and B are no longer in scope; Ramba detects this, and avoids materialization of these arrays

Drop-in Numpy replacement

- E.g.: stencil to smooth out data

```
import numpy as np
def smooth(A):
    return 0.25*A[:-2]+
        0.5*A[1:-1] +
        0.25*A[2:]
```

- Returns smoothed version of input
- Each element is a weighted sum of the preceding and following one

- Ramba version:

```
import ramba as np
def smooth(A):
    return 0.25*A[:-2]+
        0.5*A[1:-1] +
        0.25*A[2:]
```

- Just 1 line change (for import)
- Works even in distributed context
- Automtically communicates between workers to get data along borders of partitions

Ramba – Softmax program

```
import ramba as np

def softmax(A):
    maxval = np.max(A)
    out = (A-maxval).exp()
    normfactor = 1.0/np.sum(out)
    out *= normfactor
    return out
```

- This is essentially the same as a Numpy version
- This can work for any Ramba source array A
- Leverages threading and distribution (assuming A is distributed)
- Output distributed like source array
- Note the use of in-place multiply to avoid creation of an extra array

Ramba – Manual matmul

```
import ramba as np

def matmul(A,B):
    a,b = A.shape
    b2,c = B.shape
    assert b==b2
    Ax = np.broadcast_to(A.T,
            (c,b,a)).T
    Bx = np.broadcast_to(B,
            (a,b,c))
    Cx = Ax*Bx
    C = np.sum(Cx, axis=1)
    return C
```

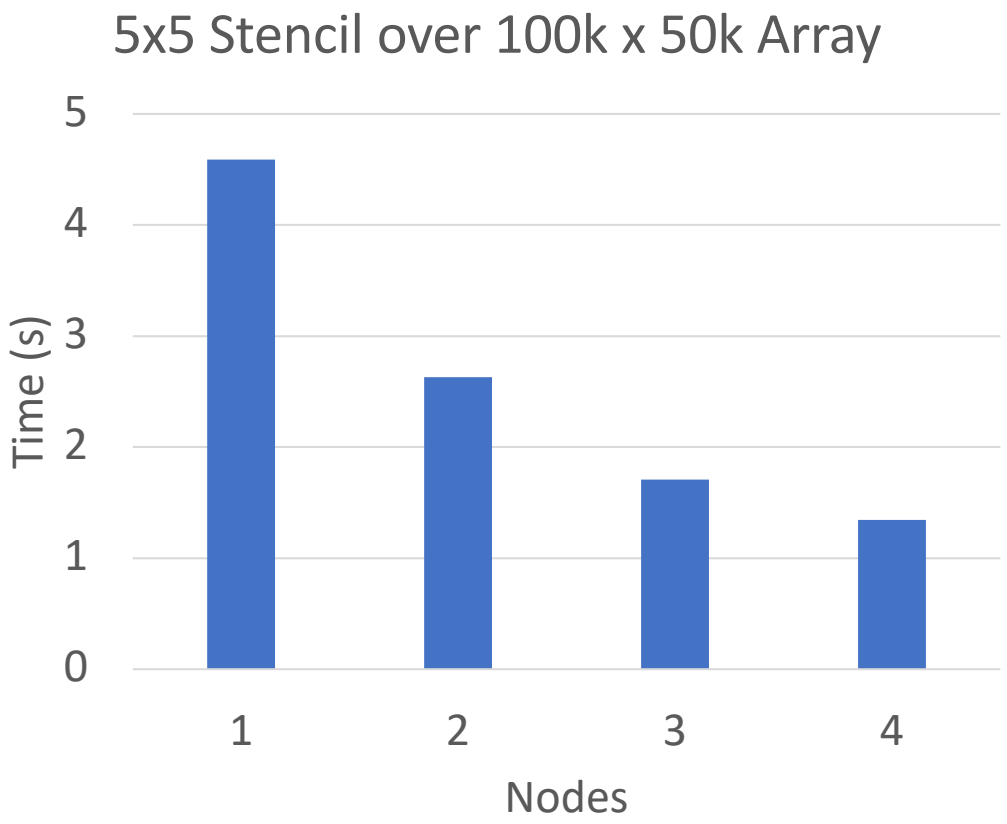
- Ax and Bx are “broadcasted” 3D views of A and B, adding an extra last and first axis, respectively
- The main computation is effectively a triply nested loop running on the 3D views
- The reduction along axis 1 is fused into the multiply loop, so Cx is not materialized
- This leverages multiple threads as well as multiple processes / nodes
- Note: this code can run in Numpy

Ramba – Cool program: General Stencil

```
import ramba as np

def stencil(A,W):
    a,b = A.shape
    c,d = W.shape
    w = a-c+1
    h = b-d+1
    assert a>=c and b>=d
    out = np.zeros((w,h))
    for i in range(c):
        for j in range(d):
            out += (W[i,j] *
                    A[i:w+i,j:h+j])
    return out
```

- Applies rectangular stencil to 2D array
- A is a ramba array, W is a numpy array of stencil weights
- The main computation is effectively a quadruply-nested loop
- The implicit 2D loop over A/out is promoted to the outside, and the explicit loop over the weights is unrolled
- This works with threads and distribution, ***automatically*** communicating data between processes/nodes as needed
- Note: this code can run in Numpy

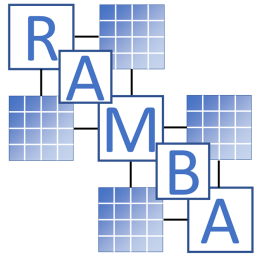
Ramba – General Stencil results



- Measured on 4 node cluster, each node:
  - 2x Intel® Xeon® E5-2699 v3 @ 2.3 GHz
  - 128 GB RAM
- 100k x 50k float64 main array
- 5x5 float64 stencil
- 1 warmup run, 10 measured runs
- 2 processes per node

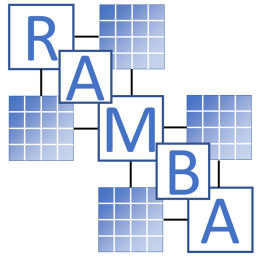
Nodes	Time	Speedup	Efficiency
1	4.59	1.00	1.00
2	2.63	1.75	0.87
3	1.71	2.69	0.90
4	1.35	3.41	0.85

Key Numpy features supported



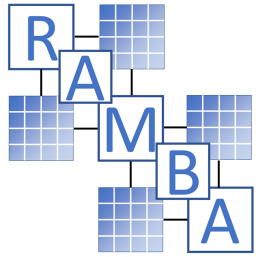
- Most core array APIs supported
  - Constructors, e.g., ones, fill_like, etc.
  - Ranges, e.g., arange, linspace
  - Arithmetic, trig operations
  - Reductions
  - Matrix multiply (2D)
- Mutable views
  - Slice indexing extracts mutable subarray that shares elements with source array, as in numpy
  - Many other systems construct copies, often immutable
- Axis manipulations, “broadcasting”

Numpy features not supported



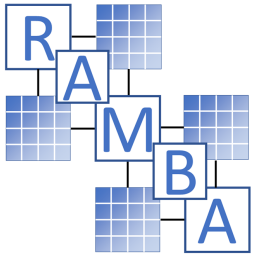
- Reshape
  - Reshape in numpy is cheap, and typically does not involve copying data
  - most reshape operations require extensive copying in a distributed context
  - Provide reshape_copy to clearly indicate
- Splitting, tiling arrays
- Insert / remove / rearrange elements
- “fancy” indexing
- Numpy non-core APIs, e.g., numpy.linalg, numpy.FFT

API extensions beyond Numpy



- `sync()` to force all deferred operations to execute
- `asarray()` method converts distributed array to numpy array
- `asarray` flag for reductions; keeps result as an array
- Options to select how arrays are partitioned when constructed
- Skeletons for common code patterns
  - Map
  - Reduce
  - Stencil
  - Cumulative sum

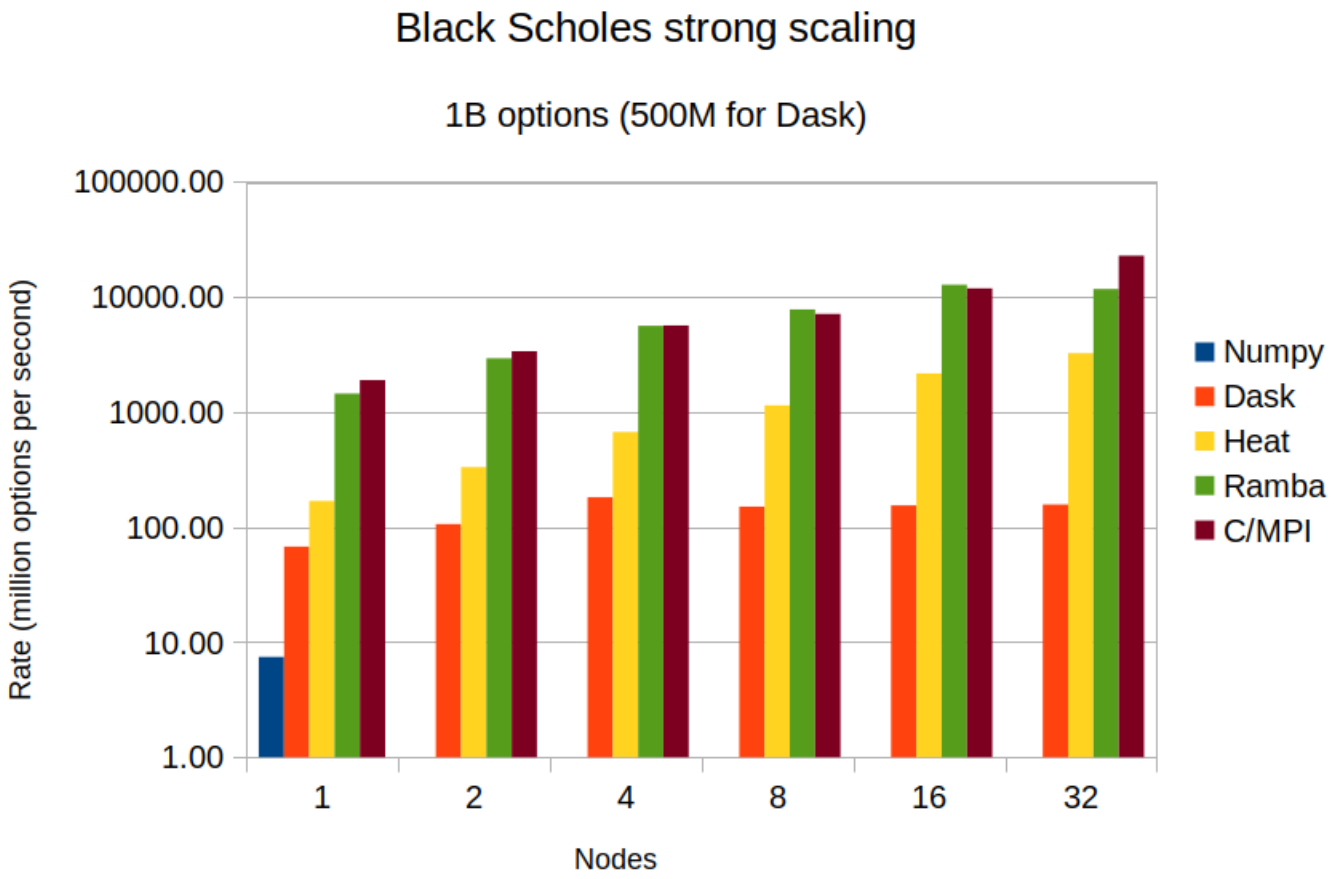
Internals: Controller-Worker vs. SPMD



- Ramba can use Ray or MPI as distribution layer
- On Ray, uses a Controller-Worker model
  - Python program executes as the controller
  - Each ndarray operation queues work, data allocations for workers
  - Fan-out, Fan-in on each set of fused operations
- With MPI, defaults to Single Program, Multiple Data (SPMD) model
  - Python program executes on each rank
  - Array operations work on local parts on each rank, may trigger communication
  - Each runs independently until explicit synchronization or communications
  - Potentially more scalable than Controller-Worker
- Non-array operations execute once (on controller) with Ray, but run on every rank with MPI (unless conditioned on MPI rank)

Black Scholes computation on Ramba

- We demonstrate high single-node speed and good scalability with largely unmodified Numpy code running (only changes are to import Ramba, and to get correct timings with asynchronous / deferred computation).
- Ramba performs on par with a C/MPI+OpenMP implementation, though strong scaling drops off at 32 nodes.
- In contrast, other frameworks for distributed Python arrays are far behind – by an order of magnitude or more. Dask in particular is disappointing – slow on a single node, scales poorly, and uses more memory than the other systems.



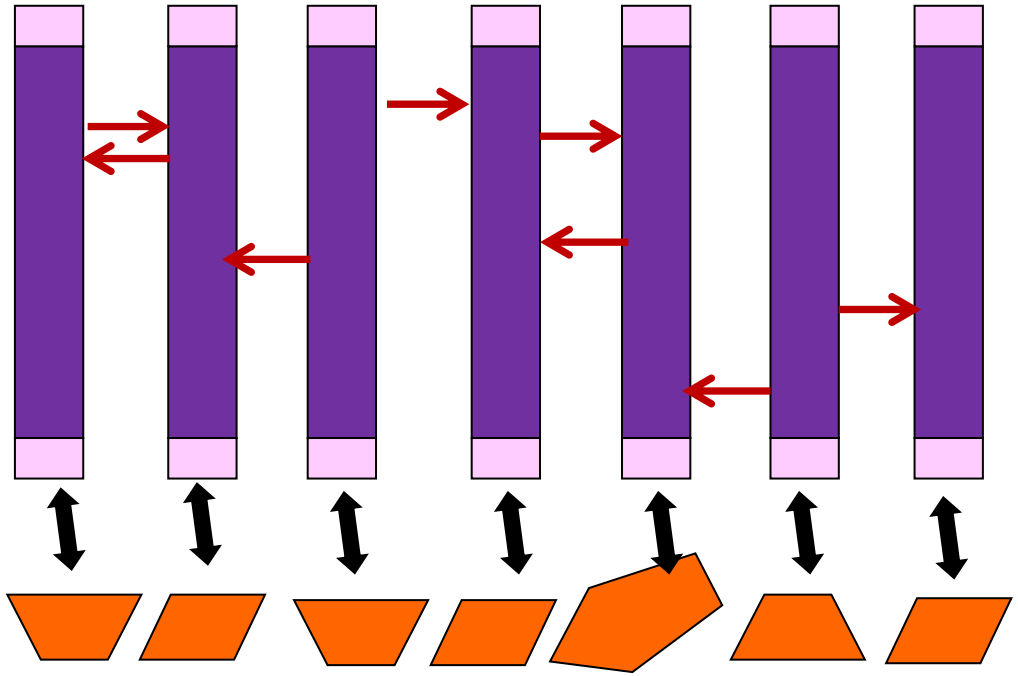
Ramba – Next steps

- Ramba is in active development by our team, with performance and feature additions on a regular basis. Available on public GitHub repo.
- We are looking for partners / customers to try using Ramba to accelerate existing Numpy programs and scale them out to distributed settings.
- We are also actively looking for landing zones for the Ramba technology. This may be other existing distributed array frameworks in Python, or wrappers such as Xarray.

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
  - 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

```
from mpi4py import MPI

print("Hello World!")
```

```
> 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

What about the GPU?

Data Parallel Essentials For Python

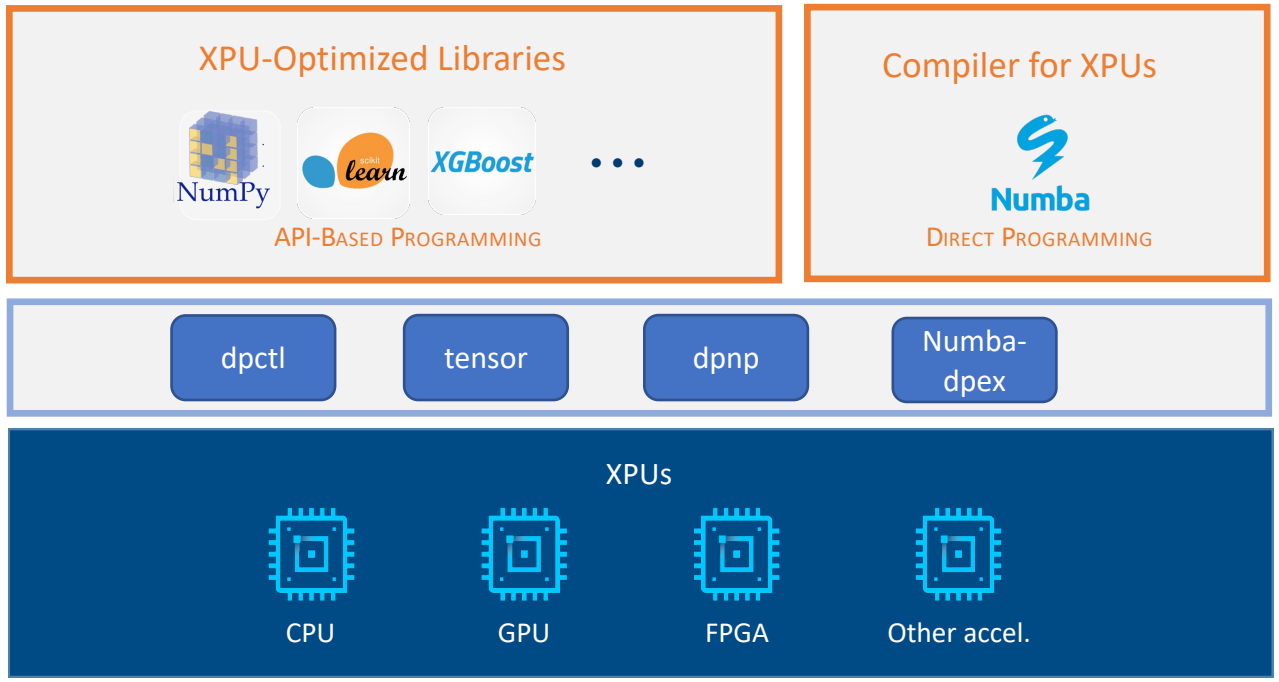
Interfacing oneAPI and Python

Diptorup Deb
Oleksandr Pavlyk
Intel Corp

March 08, 2022

Data Parallel Extensions for Python

Fostering a oneAPI/SYCL-based ecosystem for PyDATA



SYCL is an open industry standard for programming GPUs, CPUs and other accelerators. We are part of a growing community that is trying to make SYCL the ubiquitous standard for GPU programming.

Core Goals

- Prescribe a Pythonic offload model and interoperability API
  - offload model (compute follows data)
  - data interchange and interoperation specification
- Building blocks to foster a SYCL-based ecosystem in Python
  - SYCL USM-based Python array library (Array API standard)
  - Compilation of Python bytecode to SPIR-V (a standard IR for GPU programming) for SYCL
  - SYCL-based drop-in replacement for NumPy
- Ease Python native extension development for oneAPI and SYCL libraries

Programming Model Goals

Offload Model

- Pythonic offload model following array API spec (https://data-apis.org/array-api/latest/)
- Offload happens where data currently resides (“compute follows data”)

```
X = dp.array([1,2,3])
Y = X * 4
```

executed on default device

```
X = dp.array([1,2,3], device=“gpu:0”)
Y = X * 4
```

executed on “gpu:0” device

```
X = dp.array([1,2,3], device=“gpu:0”)
Y = dp.array([1,2,3], device=“gpu:1”)
Z = X + Y
```

Error! Arrays are on different devices

Interoperability

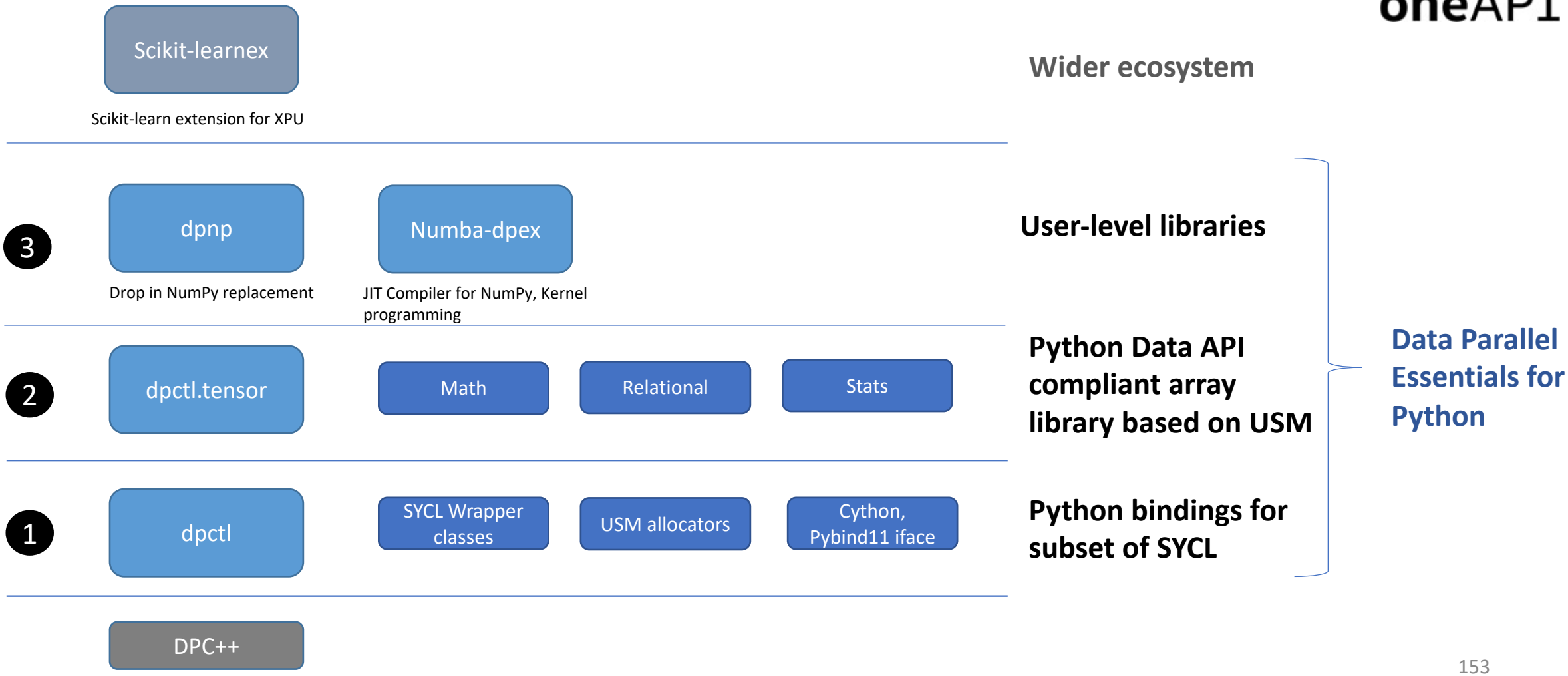
Native extensions

- Extend the dlpack standard (https://github.com/dmlc/dlpack)

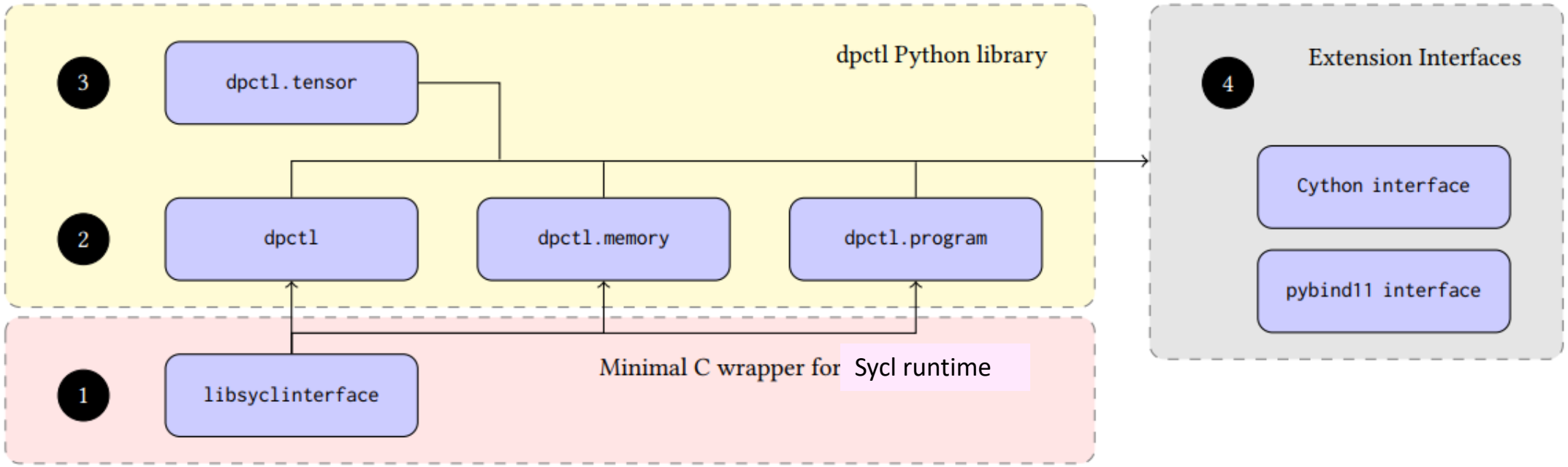
Pure Python modules

- Define a protocol like NumPy’s __array_interface__ and CuPy’s __cuda_array_interface__

Current Ecosystem



dpctl – Data parallel control



1 Library providing a minimal C API for the main Sycl runtime classes

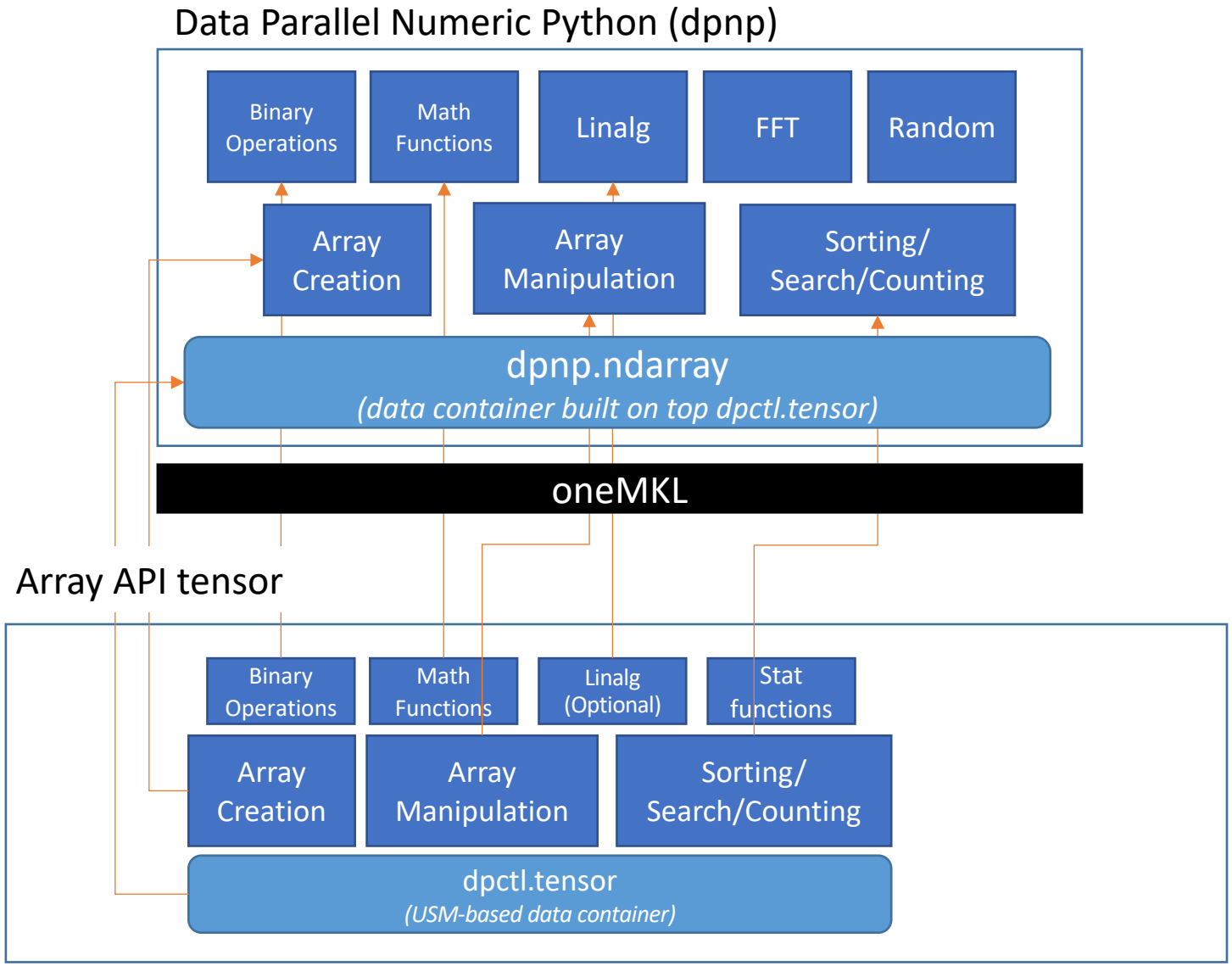
2 Python modules exposing SYCL runtime classes, USM allocators, and kernel bundle

3 A data API standard complaint array library supporting USM allocated memory

4 Native API to use dpctl objects in Cython and pybind11 extensions modules

USM: Unified shared memory … an address space visible to the host and the GPU

Array Libraries



Original CPU script

```
import numpy as np

x = np.array([[1, 1], [1, 1]])
y = np.array([[1, 1], [1, 1]])

res = np.matmul(x, y)
```

Modified XPU script

```
import dpnp as dp

x = dp.array([[1, 1], [1, 1]], device="gpu")
y = dp.array([[1, 1], [1, 1]], device="gpu")

res = dp.matmul(x, y) # res resides on gpu
```

Extension Interfaces


```cpp
#include "dpctl4pybind11.hpp"
#include <CL/sycl.hpp>
#include <oneapi/mkl.hpp>
#include <pybind11/pybind11.h>
#include <pybind11/stl.h>

void gemv_blocking(sycl::queue q,
                   dpt::usm_ndarray m,
                   dpt::usm_ndarray v,
                   dpt::usm_ndarray r,
                   const std::vector<sycl::event> &deps = {})
{
    auto n = m.get_shape(0);
    auto m = m.get_shape(1);
    int mat_typenum = m.get_typenum();
    /* various legality checks omitted */
    sycl::event res_ev;

    if (mat_typenum == UAR_DOUBLE) {
        auto *mat_ptr = m.get_data<double>());
        auto *v_ptr = v.get_data<double>());
        auto *r_ptr = r.get_data<double>();
        res_ev = oneapi::mkl::blas::row_major::gemv(
            q, oneapi::mkl::transpose::nontrans, n, m, 1,
            mat_ptr, m, v_ptr, 1, 0, r_ptr, 1, depends);
    }
    else
        throw std::runtime_error("unsupported");

    res_ev.wait();
}

PYBIND11_MODULE(_onemkl, m)
{
    // Import the dpctl extensions
    import_dpctl();
    m.def("gemv_blocking", &gemv_blocking, "oneMKL gemv wrapper");
}
```

- **Create a Python ext. to call onemkl::gemv in < 40 loc (fits on a slide)**
- **Invoke it seamless from Python using dpctl, dpctl.tensor**

```python
import dpctl;
import numpy as np
import dpctl.tensor as dpt
import onemkl4py

# Programmatically select a device
d = select_device()
# Create an execution queue for the selected device
q = dpctl.SyclQueue(d)
# Allocate matrices and vectors objects using NumPy
Mnp, vnp = np.random.randn(5, 3), np.random.randn(3)
# Copy data to a USM allocation
M = convert_numpy_to_tensor(Mnp, q)
v = convert_numpy_to_tensor(vnp, q)
r = dpt.empty((5,), dtype="d", sycl_queue=q)
# Invoke a binding for the oneMKL gemv kernel.
onemkl4py.gemv_blocking(M.sycl_queue, M, v, r, [])
```

Numba-dpex

Array-style programming

```
@njit(parallel=True)
def l2_distance(a, b, c)
    return np.sum((a-b)**2)
```

NumPy (array) style programming. Requires minimum code changes to compile existing Numpy code for XPU.
Nvidia cuPy offers this programming model with JIT fusion capabilities via cupy.fuse()

Explicit prange (parfor) loops

```
@njit(parallel=True)
def l2_distance(a, b, c)
    s = 0.0
    for i in prange(len(a))
        s += (a[i]-b[i])**2
    return s
```

Parfor-style programming. Preferred by some users when iteration space requires complex indexing.
Unique for CPU. Intel extends to XPU via numba-dpex. No CUDA alternatives to date

OpenCl-style kernel programming

```
@kernel(access_type={"read_only": ["a", "b"], write_only:["c"]})
def l2_distance(a, b, c)
    i = numba_dpex.get_global_id(0)
    j = numba_dpex.get_global_id(1)
    sub = a[i,j] - b[i,j]
    sq = sub ** 2
    atomic.add(c, 0, sq)
```

Most advanced programming model.
Recommended to get highest performance on XPU yet avoiding DPC++.
Nvidia @cuda.jit offers this programming model in Numba

Current Status

- Included in oneAPI Basekit and Intel Distribution for Python* (IDP)
- Open-source development on github.com/IntelPython
- Packages available from Anaconda cloud and PyPi

Programming Model

Compute Follows Data

- Pythonic offload model following array API spec
- Explicit control over execution based on data placement

Interoperability

- __sycl_usm_array_interface__ modeled after NumPy's __array_interface__
- Dlpack for exchanging data across native extensions

```
import dpnp as dp
    # Case 1
    #  Allocate X on the default device
    X = dp.array([1,2,3])
    #  scaling of X executed on device of X, result
        placed into Y
    Y = X * 4
    # Case 2
    #  Allocate X on "gpu:1"
    X = dp.array([1,2,3], device="gpu:1")
    # Executed on "gpu:1"
    Y = X * 4
    # Case 3
    X1 = dp.array([1,2,3], device="gpu:1")
    X2 = dp.array([1,2,3], device="gpu:0")
    #  error!
    Y = X1 + X2

    # Arrays can be associated with another device
    # (copy is performed if needed)
    X1a = X1.to_divice(device=dev)
```

Wrap-up

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)

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)
Ramba				0.274 (0.02)	1.56 (0.02)
Multiprocessing			0.229 (0.002)	1.54 (0.016)	
Ray-tasks			0.156 (0.004)	1.52 (0.02)	
Ray-divide-conquer		0.098 (0.04)	0.36 (0.06)	Crash	
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

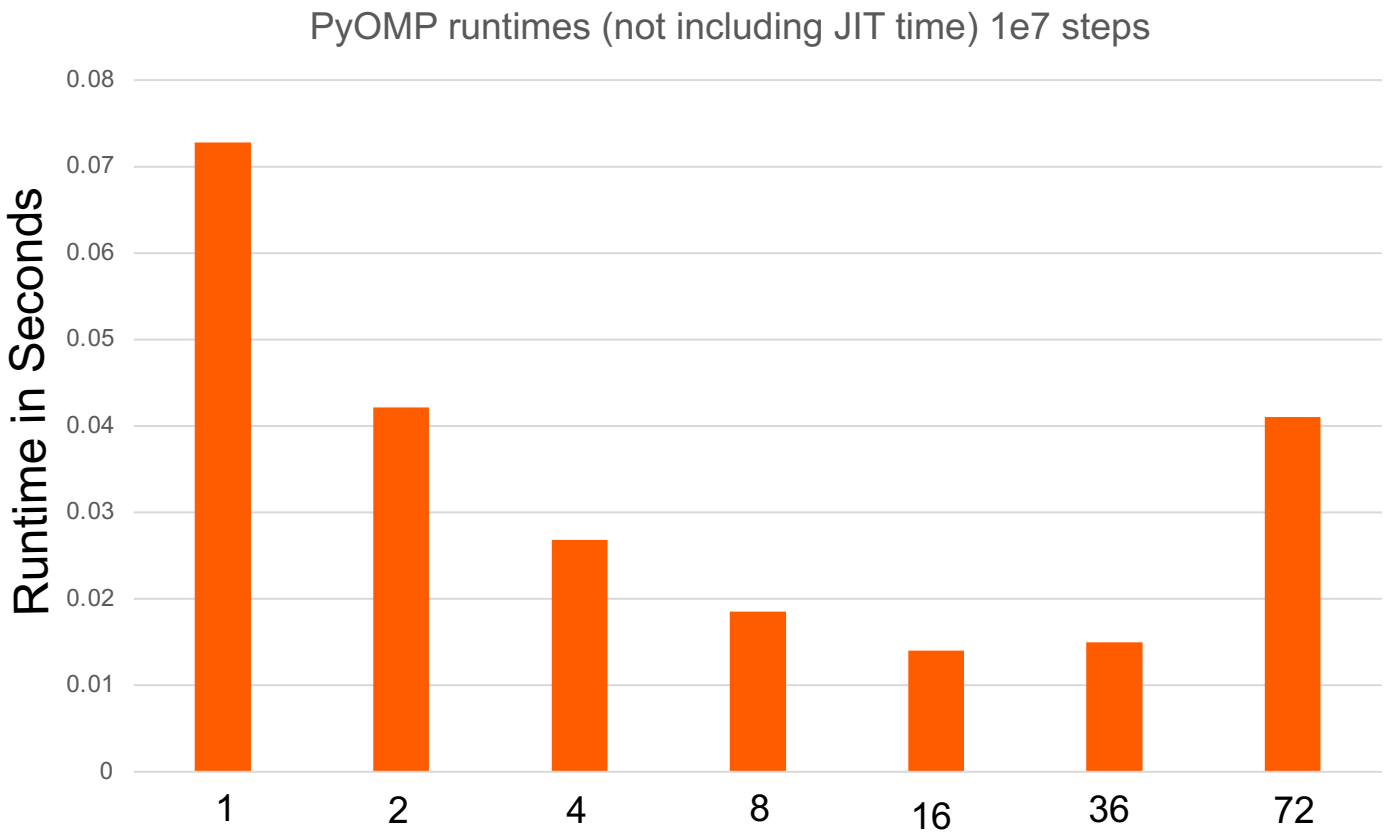
Single threaded

Compiled

← Compiled

Don’t use more threads than the problem demands

- You should always check how the runtime improves with the number of threads.



Using the PyOMP program for the system on the previous slide … 18 cores per socket and two sockets for 36 physical cores. Hyperthreading doubles the number of hardware threads to 72.

Adding threads adds overhead … especially when crossing between sockets.

Hyperthreading costs performance if the arithmetic logic units are fully saturated by a single thread.

Choosing a parallel programming model

- There is no single programming model to rule them all. You need to look at a number of factors and decide what is best for you.
  - Remember … in the long run, proliferation of parallel programming models hurts us all. Unless you are researching programming environments for python, pick one that looks like it will become a “leader of the pack”
  - Understand your applications and the design patterns you’ll be using. Make sure the model you choose works for those patterns.
  - What type of hardware will you be using over the lifetime of the application. Most of us use CPUs but will you need a cluster as the problem size grows? Are GPUs important to you?
  - Workloads are rarely just a single app. Think about all the components that make up your workflow and make sure your programming model covers all the hardware elements those components need.
  - The most important criteria (in order) are portability, productivity and then performance.

Fundamental design patterns of parallel programming

- Application patterns
  - Loop level parallelism
  - SPMD
  - Divide and conquer
  - Data Parallelism
  - Geometric decomposition
- Execution Patterns
  - Multiprocessing
  - Multithreading/fork-join
  - Actors
  - Map reduce
  - Task graph
  - Kernel/SIMT/vector (as with all things GPU, the nomenclature is messed up)

Parallel programming can be overwhelming … but when you look across parallel programs people write, most use a modest number of different design patterns.

There really aren't that many patterns you need to learn

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

How to write parallel programs: the Python Edition

A half day tutorial (lecture style) **75% introductory, 25% intermediate** **Prerequisite knowledge: none**

- **Abstract**

  Everyone wants to write parallel programs, right? Well, of course not. Parallel programming is hard. We'd all rather write code using whatever languages and APIs we know best and leave parallelism to others. Unfortunately, parallelism is here to stay. Most of working in scientific computing will need to use parallel systems.

  This tutorial will demystify parallel computing. We'll say a bit about hardware and why parallel computing (including with heterogeneous processors) is something you just can't avoid. Then we will review the key concepts and fundamental design patterns of parallel computing. With that conceptual basis in place, we'll review some of the common approaches to expressing parallelism in Python programs. Then with an eye to the future, we'll look at some research systems under development that should make parallel programming in python even better. The goal is a conceptual foundation for parallel programming, not mastery of any one approach. Basically, you need to know a bit about parallel programming before making an informed decision on which API to use for parallel programming. This tutorial will give you that information.

- **Motivation**
  - We go to quickly to DASK or mpi4py without spending time on the fundamentals. If someone really wants to be a productive parallel programming, it makes sense to spend some time understanding the core ideas that cut across various APIs. That is the purpose of this tutorial.
- **Outline**:
  - What every scientific programmer should know about parallel hardware
  - Understanding performance in parallel programs
  - The fundamental design patterns of parallel programming
  - The most common APIs for parallelism in Python using the patterns we just learned
  - Some interesting research systems for parallelism in Python
  - Closing thoughts and reflections on how to choose a parallel API for use in your own projects