INTRODUCTION TO PARALLEL PROGRAMMING

DR. CARL E. FIELDS

(he/him)

RPF Fellow, CCS-2/XCP-2
Los Alamos National Laboratory

Session 15, LSSTC DSFP
CfA, Harvard
July 21, 2022

Los Alamos
NATIONAL LABORATORY

@carlnotsagan

OVERVIEW

Single Processor Computing

- Modern Processors
- Instruction Level Parallelism
- Limits of ILP and modern CPUs

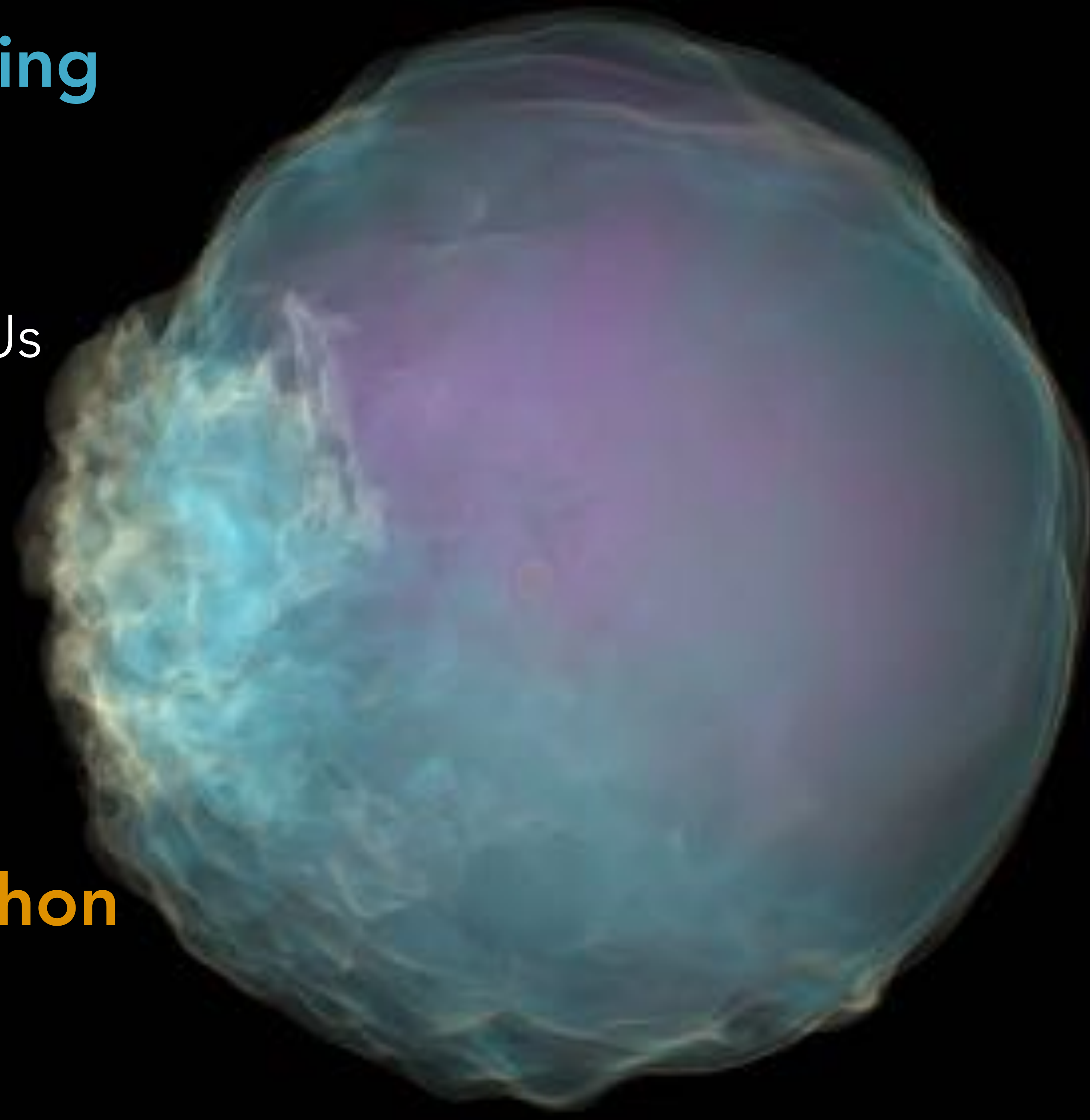
Parallel Computing

- Modern Architectures
- Flynn’s Taxonomy
- Types of parellelism
- Roofline model

Parallel Computing in Python

- Numba
- mpi4py

Summary

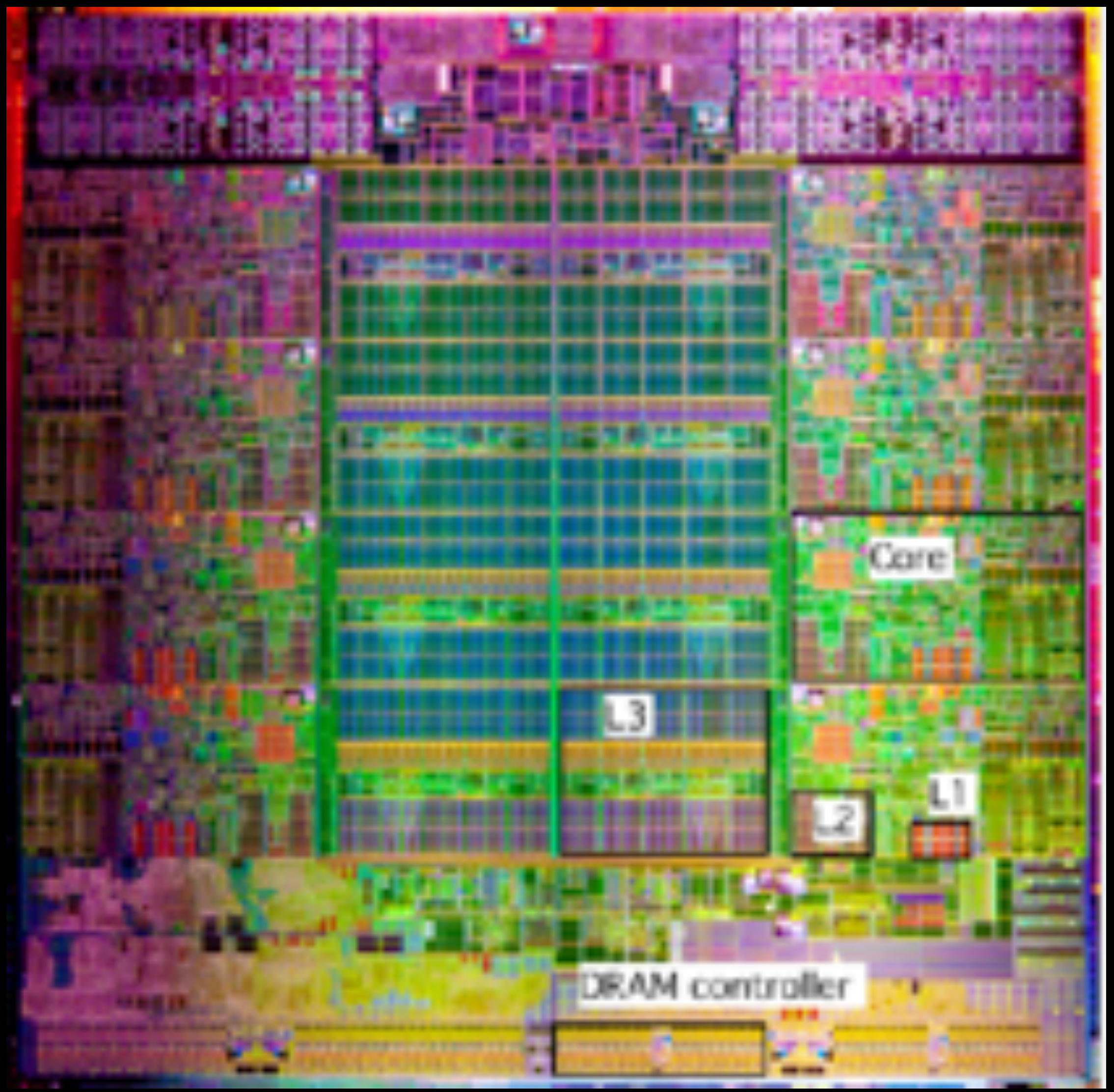


Oxygen shell burning in a 20 solar mass star.

SINGLE PROCESSOR COMPUTING

Modern Central Processing Units (CPUs)

- Multiple compute **cores**
- Hierarchy of memory
- CPU *speed* often measured by the clock rate of each core - 2.4 GHz
- Modern laptops can have 2-12 cores



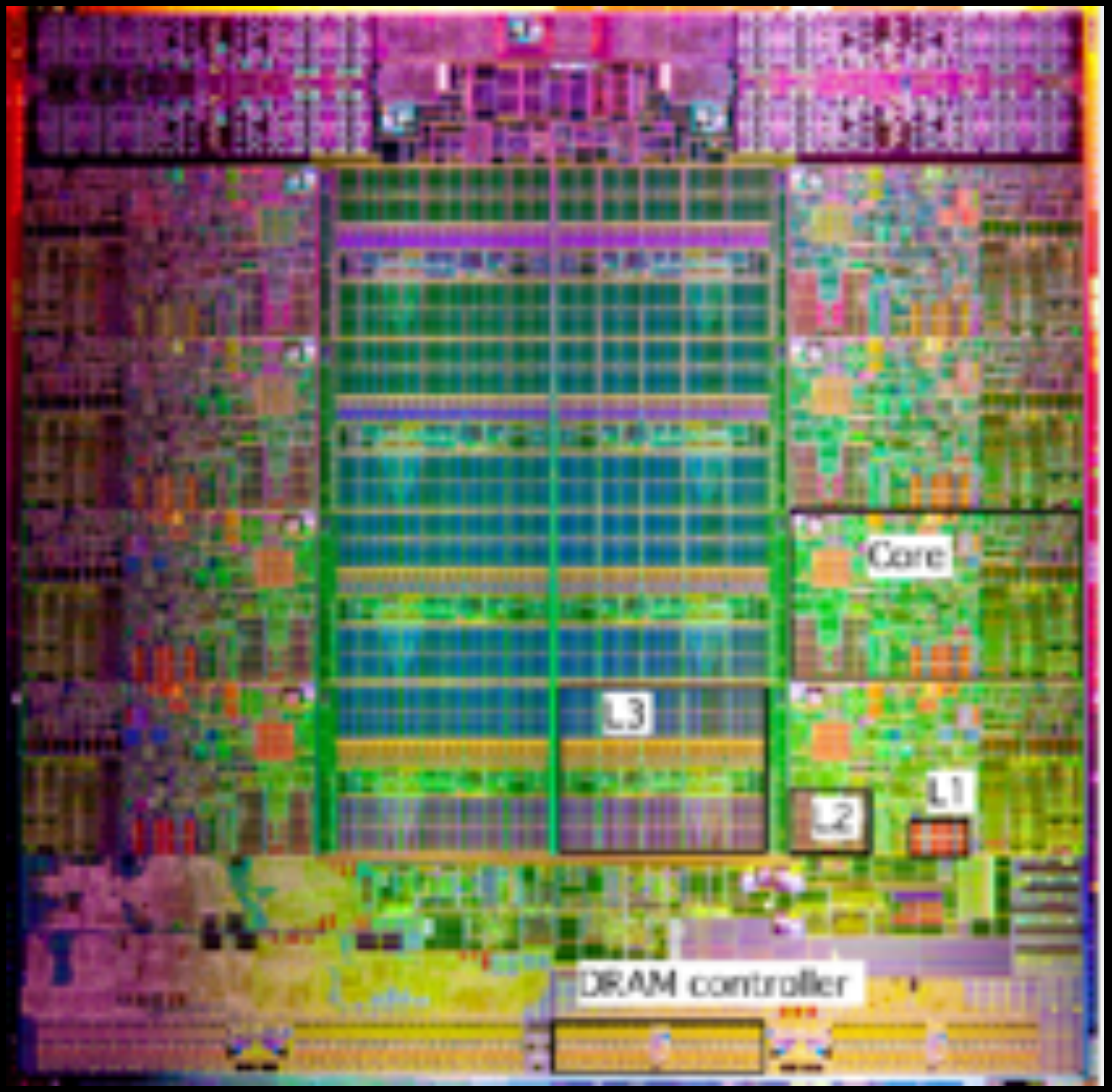
Intel Sandy Bridge Processor showing 8 cores.

SINGLE PROCESSOR COMPUTING

Utilizing processors and their compute cores

Instruction Level Parallelism (ILP)

- Multiple-issue
- Out-of-order execution
- Prefetching of data to determine dependencies
- Pipelining: stream of instructions, maximizing efficiency



Intel Sandy Bridge Processor showing 8 cores.

SINGLE PROCESSOR COMPUTING

Methods: Instruction Level Parallelism - Example

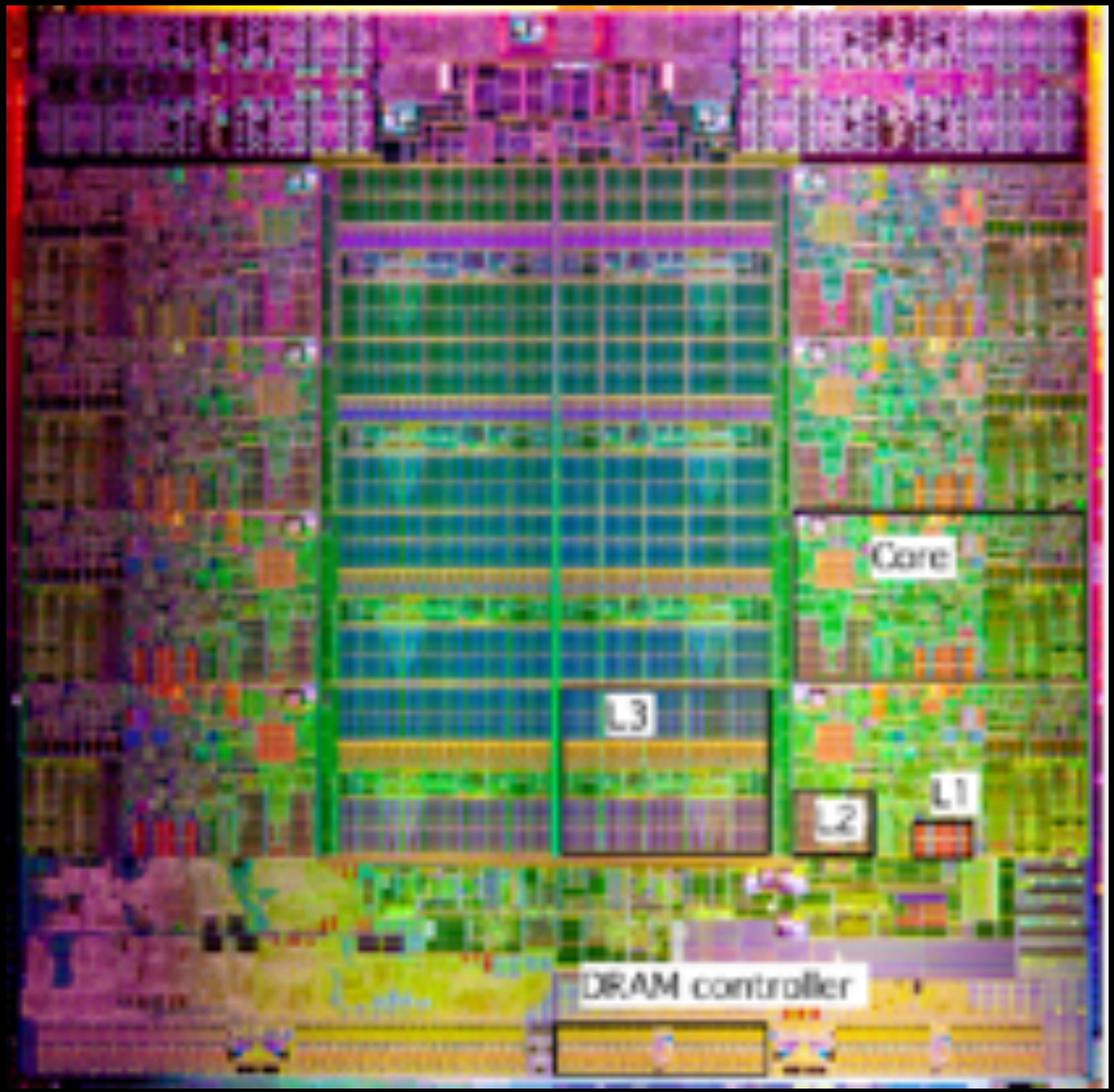
Sequential Execution	Instruction-Level Parallelism
1. a = 10 + 5 2. b = 12 + 7 3. c = a + b Instructions: 3 Cycles: 3	1.A. a = 10 + 5 1.B. b = 12 + 7 2. c = a + b Instructions: 3 Cycles: 2 (-33%)

Credit: Sukaina Xehra

SINGLE PROCESSOR COMPUTING

Approaching the limits of modern CPUs

- Clock speeds limited due to heat production
- Limits of due to intrinsic problem, branch predictions, etc.
- Solution: more compute cores at lower clock speeds.
- Challenge: out of user control, limited by CPU speed

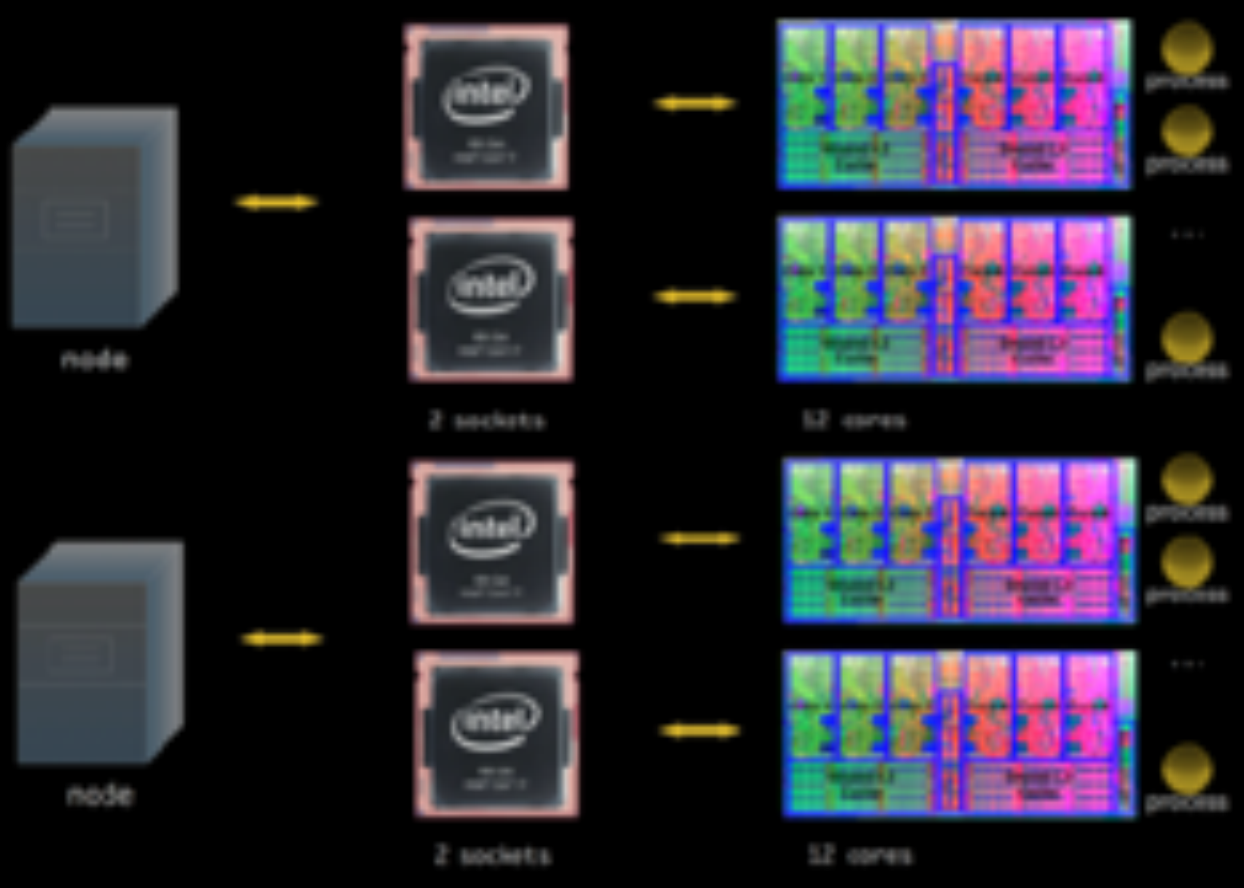


Intel Sandy Bridge Processor showing 8 cores.

PARALLEL COMPUTING

Modern Parallel Computers

- Collection of **nodes** containing multiple processors
- Can be tasked to work on the same problem
- Need to communicate, exchange information, perform calculations
- Rely on explicit parallelism from user beyond ILP.



Cluster example containing nodes with 2 processors (sockets).

PARALLEL COMPUTING

Modern Parallel Computers

- **Hybrid** architectures contain CPUs and Graphics Processing Units (GPUs)
- ***Summit (OLCF)***: 4,608 nodes - 2 CPUs + 6 GPUS each
- ***Stampede2 (TACC)***: 4,200 KNL nodes - 68 cores each

Summit supercomputer at ORNL. Credit: Carlos Jones/ORNL

PARALLEL COMPUTING

Your target machine can determine your approach

PARALLEL COMPUTING

*Determining **how** to characterize your problem*

Flynn's Taxonomy

- Derived from describing the *data* and *control flow* as *shared* or *independent*?
- **Single Program, Multiple Data (SPMD)**: Single program run simultaneously on multiple processors with different pieces of data in order to obtain results faster. SPMD is the most common style of parallel programming.

PARALLEL COMPUTING

*Determining **how** to characterize your problem*

Parallel Computing Methods: SPMD

- **Data/Distributed memory parallelism**: Each processor can run an independent program, and has its own memory without direct access to other processors' memory. Done using **Message Passing Interface (MPI) library**.

- **Task-level/shared memory parallelism**: uses teams of threads, and inside a parallel region the work is distributed over the threads with a work sharing construct. Done using **Open Multi-Processing (OpenMP/OMP) application programming interface.**

PARALLEL COMPUTING

Data/Distributed memory parallelism

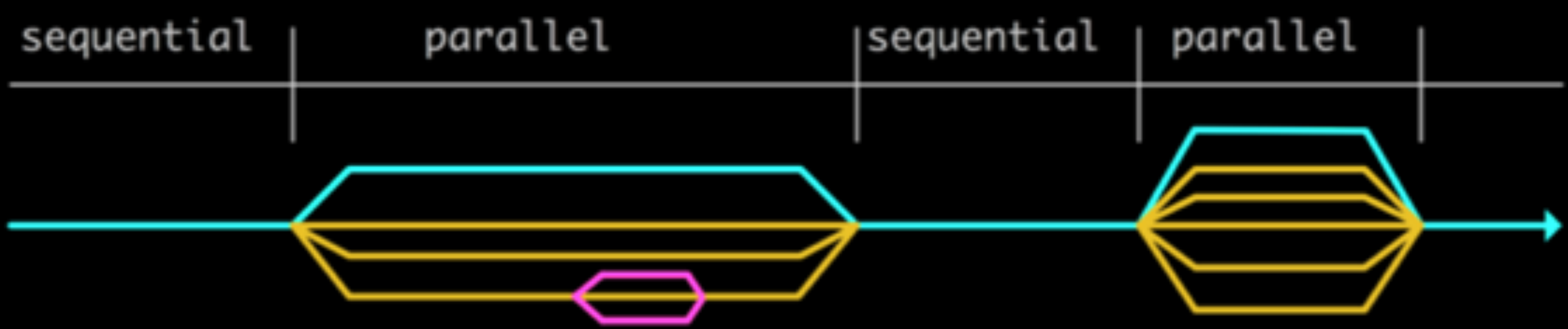
- Employed using **Message Passing Interface (MPI)** library which interfaces with many modern languages.
- Requires explicit management of data and calculations via MPI operations.
- **Rank**: process id used to distinguish processes from each other. Usually from 0 to number of processes.

*Example of **MPI Scatter** Process in the Distributed memory paradigm.*

PARALLEL COMPUTING

Task-level/shared memory parallelism

- Employed using **Open Multi-Processing (OpenMP/OMP)**.
- Parallelism is dynamically activated by a thread spawning a team of threads.
- Typically let your number of **threads** be equal to the number of **cores.**



Thread creation and deletion during parallel execution using OpenMP.

PARALLEL COMPUTING

Okay, great. But, how do I know which method is best for my problem?

PARALLEL COMPUTING

Some considerations before choosing parallel computing

- Running on multiple processors requires communication.
- If “work” not perfectly distributed, load unbalance can occur.
- Some programs may require many inherently sequential sections.
- Worry about the data. Know your problem!

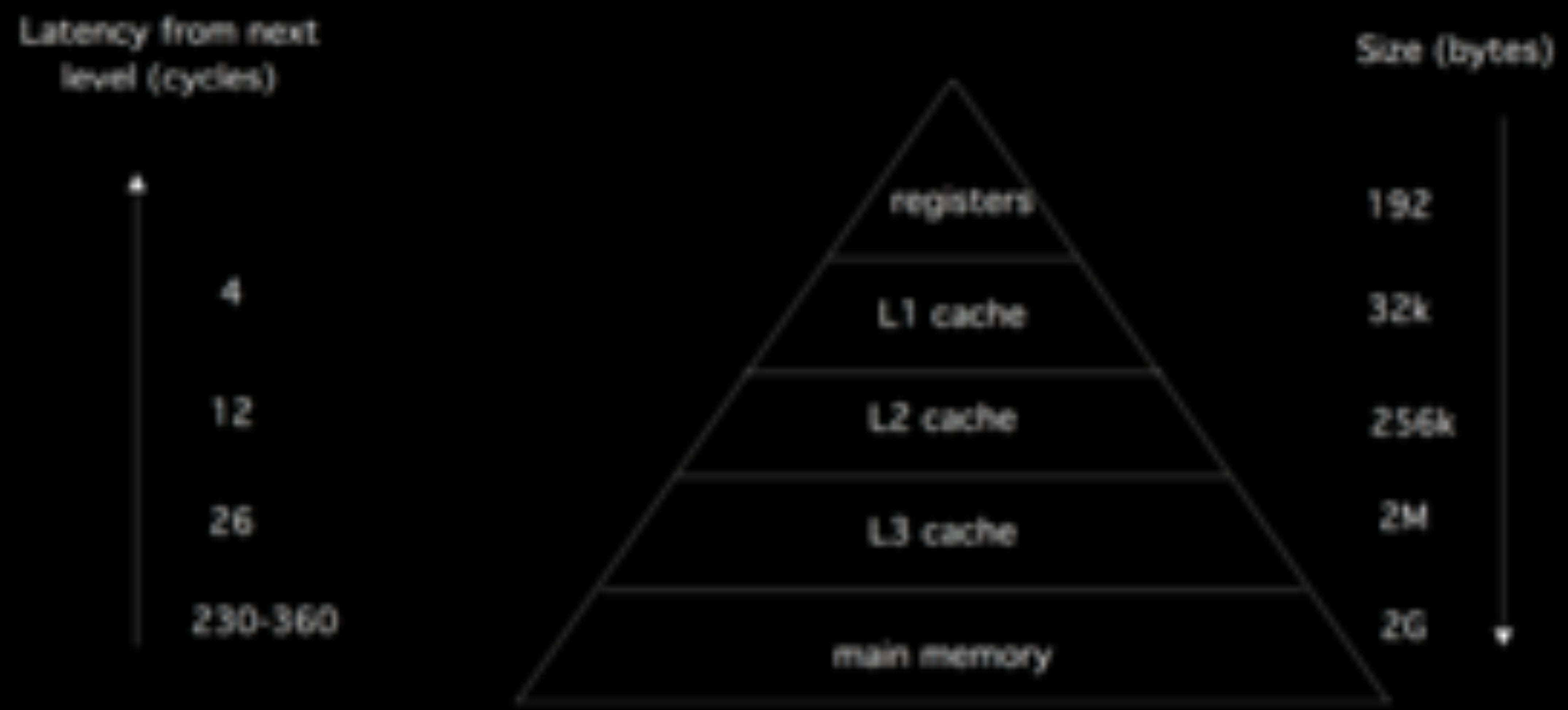
PARALLEL COMPUTING

Some definitions relevant to your problem:

- **Operational Intensity (OI) -** Unique to your problem **-** $\left(\dfrac{operations}{data\ items}\right)$
- **Bandwidth -** An absolute number determined by CPU - is the rate at which data arrives at its destination. $\left(\dfrac{data\ items}{second}\right)$
- **Performance -** $\left(\dfrac{operations}{second}\right)$ **- FLOPS**

PARALLEL COMPUTING

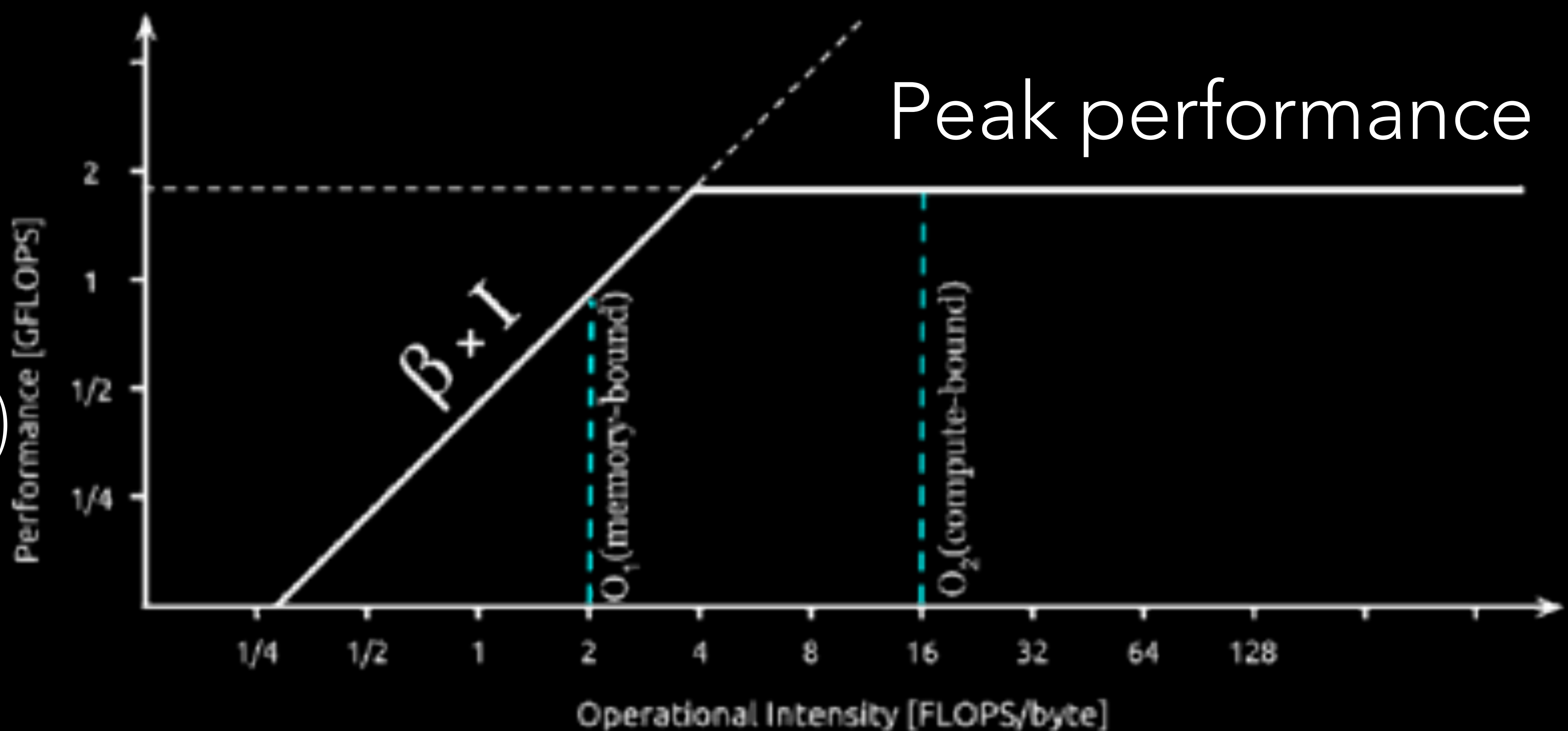
An aside on the memory hierarchy in CPUs



- **Latency:** is the delay between the processor issuing a request for a memory item, and the item actually arriving.
- Data reuse is key! Know your algorithm.

PARALLEL COMPUTING

The roofline model



$$\left(\frac{operations}{second}\right)$$

$$\left(\frac{operations}{data\ items}\right)$$

PARALLEL COMPUTING

The roofline model - where does your problem lie?

- **Memory-Bound -** aspects such as bus speed and cache size become import.
- **Compute-Bound -** the speed of the processor is indeed the most important factor.

PARALLEL COMPUTING

"Perfect examples" often called "embarrassingly parallel" problems

- Little to no startup cost - sequential code. Very little communication. Bandwidth not a factor.
- Consisting of a number of completely independent calculations. Example: Markov Chain Monte Carlo Simulations!
- Obtain close to perfect Speedup/Efficiency.

$$S_p = T_1/T_p$$

PARALLEL COMPUTING

Reality: Most problems will be less than ideal

- Determine where problem lies: compute/memory-bound.
- Minimize communication when possible.
- Identify parallelizable regions.
- Determine dependent / independent calculations.
- Consider target machine / architecture.

PARALLEL COMPUTING

Let's look at a few tools in Python that explore parallel computing.

PARALLEL COMPUTING IN PYTHON

NUMBA: Numba makes Python code fast

How it works:

- Just-in-time (jit) compiler for Python
- Numba reads the Python bytecode for a decorated function.
- Analyzes and optimizes your code.
- Uses the LLVM compiler library to generate a machine code.
- Tailored to your CPU capabilities.

PARALLEL COMPUTING IN PYTHON

NUMBA: Numba makes Python code fast

Ideal use:

- Code that is numerically orientated - high OI
- Uses NumPy a lot
- Lots of loops
- Can target GPUs.

PARALLEL COMPUTING IN PYTHON

NUMBA: Numba makes Python code fast

Works well on:

```python
from numba import jit
import numpy as np

x = np.arange(100).reshape(10, 10)

@jit(nopython=True) # Set "nopython" mode for best performance, equivalent to @njit
def go_fast(a): # Function is compiled to machine code when called the first time
    trace = 0.0
    for i in range(a.shape[0]):   # Numba likes loops
        trace += np.tanh(a[i, i]) # Numba likes NumPy functions
    return a + trace              # Numba likes NumPy broadcasting

print(go_fast(x))
```

PARALLEL COMPUTING IN PYTHON

NUMBA: Numba makes Python code fast

Would ***not*** *work well on:*

```
from numba import jit
import pandas as pd

x = {'a': [1, 2, 3], 'b': [20, 30, 40]}

@jit
def use_pandas(a): # Function will not benefit from Numba jit
    df = pd.DataFrame.from_dict(a) # Numba doesn't know about pd.DataFrame
    df += 1                        # Numba doesn't understand what this is
    return df.cov()                # or this!

print(use_pandas(x))
```

PARALLEL COMPUTING IN PYTHON

mpi4py: MPI for Python package

How it works:

- Based on MPI-2 C++ bindings.
- Translates standard MPI-2 bindings for C++ to Python.
- Supports communication of generic Python object as well as fast, near C-speed, direct array data communication of buffer-provider objects (e.g., NumPy arrays).

```
$ mpiexec -n 4 python script.py
```

Check out Victor Eijkhout's (TACC) book on HPC.

PARALLEL COMPUTING IN PYTHON

mpi4py: MPI for Python package

Point-to-Point Communication (Example):

```python
from mpi4py import MPI
import numpy

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# passing MPI datatypes explicitly
if rank == 0:
    data = numpy.arange(1000, dtype='i')
    comm.Send([data, MPI.INT], dest=1, tag=77)
elif rank == 1:
    data = numpy.empty(1000, dtype='i')
    comm.Recv([data, MPI.INT], source=0, tag=77)
```

1. Imports
2. Get WORLD information. Including current rank (process).
3. Rank 0 creates array and uses Send.
4. Rank 1 creates *empty* array to be filled as Recv'd from Rank 0.

- *Your problem* will determine the needed communication.

PARALLEL COMPUTING IN PYTHON

mpi4py: MPI for Python package

Scattering Python Objects (Example):

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

if rank == 0:
    data = [(i+1)**2 for i in range(size)]
else:
    data = None
data = comm.scatter(data, root=0)
assert data == (rank+1)**2
```

1. Imports
2. Get WORLD information. Including current rank (process) and total number of ranks (size).
3. Rank 0 creates array and distributed using scatter.
4. *Local* data is determined by rank.

- Want to ***verify*** that individual rank has desired local data.

PARALLEL COMPUTING IN PYTHON

Considerations for choosing the best tool(s) for the job

Memory:

- Shared? OpenMP/Numba
- Distributed? mpi4py

Operational Intensity:

- Compute-bound/memory-bound?

Access to resources:

- What is the composition of the compute nodes?

Profiling can help choose path forward

- Tools for measure program speed, efficiency!

PARALLEL COMPUTING IN PYTHON

One more comment on Performance Portability

Definition:

- Ability of computer programs and applications to operate effectively across different platforms.

In practice:

- Write generic routines for most HPC platforms - CPUs/GPUs
- Leverage Performance Portability frameworks like **Kokkos**!

PARALLEL COMPUTING IN PYTHON

Kokkos: Core Libraries

Abstraction Layers in Programming

- Kokkos Core implements a programming model in C++ for writing performance portable applications targeting all major HPC platforms.
- Supports CUDA, HIP, SYCL, HPX, OpenMP and C++ threads as backend programming models.
- For Python too - **PyKokkos**

PARALLEL COMPUTING IN PYTHON

PyKokkos: *a framework for writing performance portable kernels in Python.*

PyKokkos: Example

```python
import pykokkos as pk

@pk.workunit
def hello(i: int):
    pk.printf("Hello, World! from i = %d\n", i)
```

```python
pk.parallel_for(10, hello)
```

1. Imports
2. Define workunit.
3. Call work unit passing number of threads - not unique to an architecture. Determined by Kokkos.

- Provides more portability than Numba, less limited than Cython.

SUMMARY

Single processor computing

- Characterized modern CPU and components.
- Discussed ILP and non-user controlled parallelization.
- Limitations of ILP and modern CPU design.

Parallel Computing

- Modern parallel computing architectures
- Flynn's Taxonomy and the SPMD Model
- Distributed (MPI) and Shared Memory (OpenMP) Parallel Approaches
- Roofline Model: where does my problem lie?

Parallel Computing in Python

- Numba and j-i-t compilation examples and limitations
- mpi4py examples
- Performance Portability considerations and PyKokkos

THANK YOU

Worry about the data (and operational intensity)!

Web: ***carlnotsagan.com***
Email: ***carlnotsagan@lanl.gov***

Los Alamos
NATIONAL LABORATORY

@carlnotsagan