Scaling NumPy Applications from 1 CPU to Thousands of GPUs

Seshu Yamajala

May 4, 2023

BOLD PEOPLE. VISIONARY SCIENCE. REAL IMPACT.

U.S. DEPARTMENT OF ENERGY Stanford University SLAC NATIONAL ACCELERATOR LABORATORY

About Us

- Computer Science Research Group at SLAC National Accelerator Lab headed by Prof. Alex Aiken at Stanford
- Group’s primary focus is on HPC
- We collaborate with domain scientists on applications of Legion parallel programming framework
- Legion project is a collaboration between:

Stanford

SLAC NATIONAL ACCELERATOR LABORATORY

NVIDIA

Los Alamos NATIONAL LABORATORY

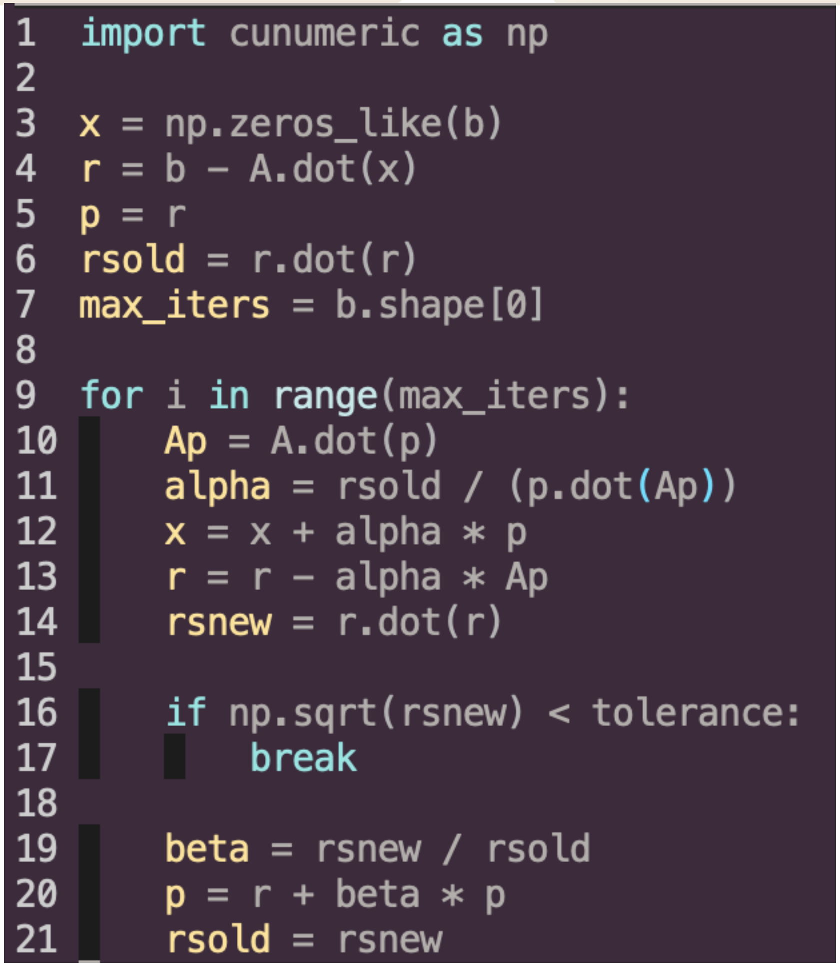
Motivation

- Python has become ubiquitous in all areas of science
- NumPy significantly improves the performance of numerical Python applications
- Together Python and NumPy have lowered barrier for entry for developing complex scientific applications
- Out of the box applications are still limited to:
  - 1 CPU
  - Memory available in a single node
  - No GPUs
- Solutions like Dask, PySpark, CuPy + mpi4py exist but:
  - Not easy to use
  - Require modifications to user code
- What if there was a drop-in replacement for NumPy that could fix these problems? Enter cuNumeric!

What is cuNumeric?

- cuNumeric is:
  - Drop-in replacement for NumPy
  - Distributed
  - GPU accelerated
  - Built on top of Legate and Legion

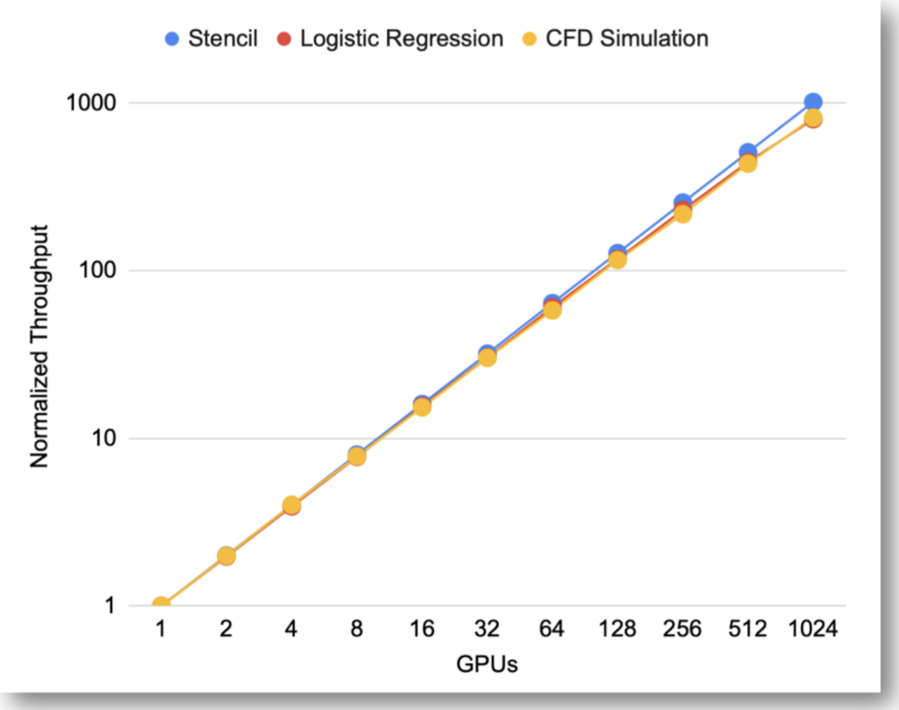
- Conjugate Gradient Solver
  - 1 line change (import cunumeric as np)
  - Scales from 1 CPU to 1024 GPUs and beyond



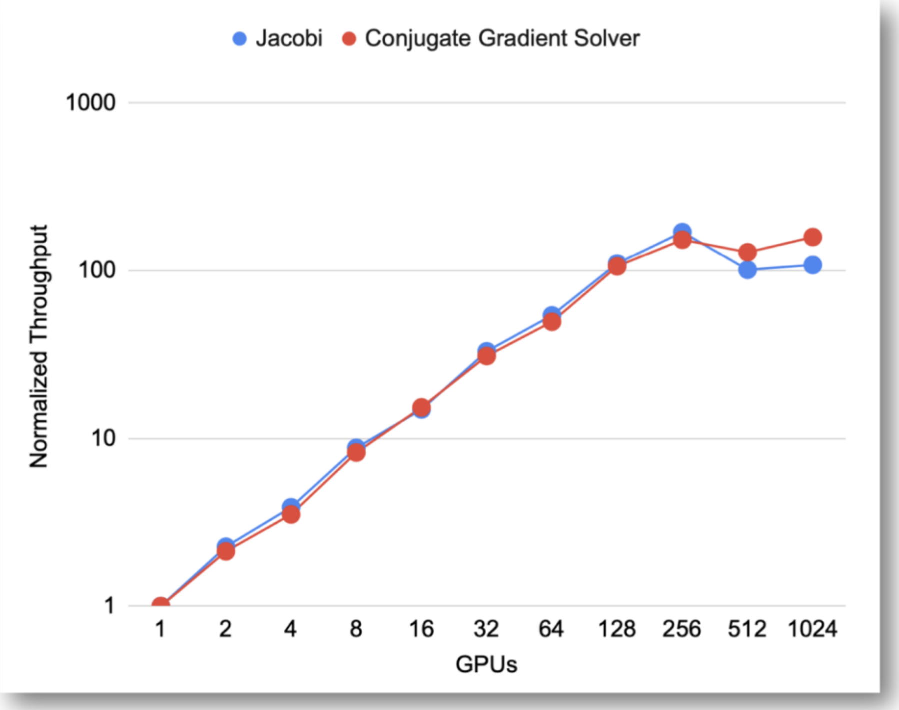
```python
import cunumeric as np

x = np.zeros_like(b)
r = b - A.dot(x)
p = r
rsold = r.dot(r)
max_iters = b.shape[0]

for i in range(max_iters):
    Ap = A.dot(p)
    alpha = rsold / (p.dot(Ap))
    x = x + alpha * p
    r = r - alpha * Ap
    rsnew = r.dot(r)

    if np.sqrt(rsnew) < tolerance:
        break

    beta = rsnew / rsold
    p = r + beta * p
    rsold = rsnew
```

Source: Lee, Wonchan. “cuNumeric and Legate: How to Create a Distributed, GPU-Accelerated Library”, 23 March 2023, Nvidia GTC

Weak Scaling Performance



Benchmarks with nearest neighbor communication

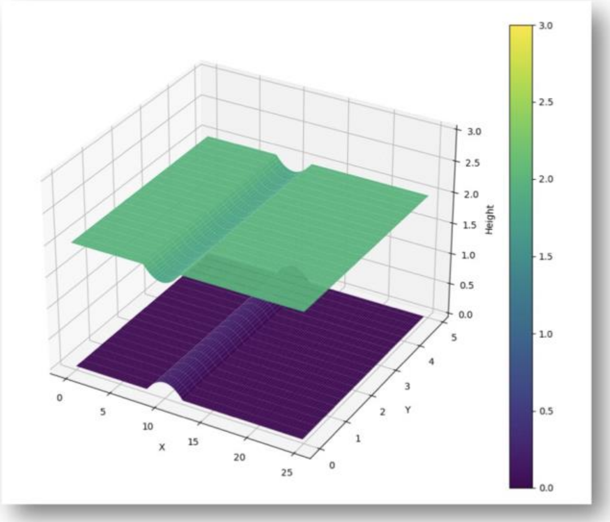


Benchmarks with logarithmic communication complexity

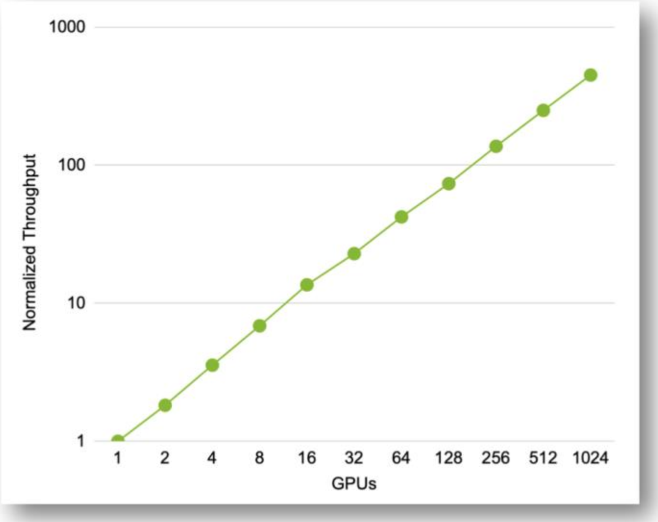
Source: Lee, Wonchan. “cuNumeric and Legate: How to Create a Distributed, GPU-Accelerated Library”, 23 March 2023, Nvidia GTC

TorchSWE Example

- Solver for shallow water equations
- Originally written using CuPy and MPI
- Ported to cuNumeric by removing MPI code



Topography (below) and water level (above)



Weak scaling performance (40M grid points / GPU)

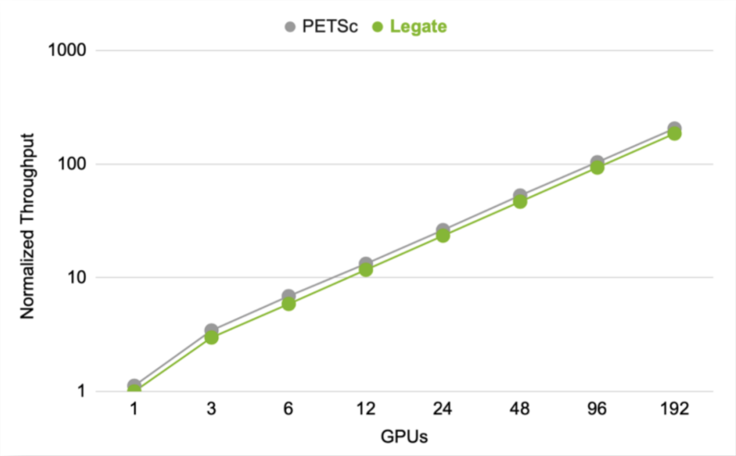
Source: Lee, Wonchan. “cuNumeric and Legate: How to Create a Distributed, GPU-Accelerated Library”, 23 March 2023, Nvidia GTC

Current Status of cuNumeric

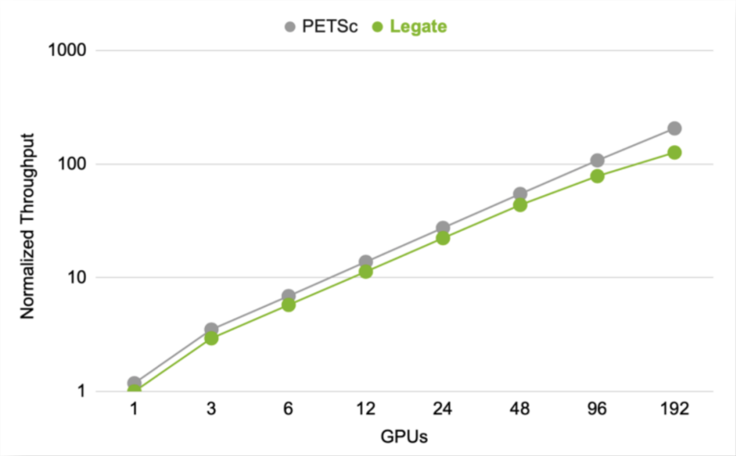
- Beta released by Nvidia in March at GTC’23
- 60% API coverage of NumPy
  - cuNumeric falls back to single-core NumPy for any operations that aren’t implemented in a distributed or GPU-accelerated manner
- Supports Jupyter notebooks but only single node currently
- Will be deployed on Perlmutter any day now
- Whats coming in 2023:
  - np.linalg and np.fft
  - Distributed IO
  - Higher-order operators
  - Performance improvements
- Currently investigating uses of cuNumeric at LCLS-II

What about sparse matrices?

- Legate sparse implements scipy.sparse API
- Currently at 35% coverage for CSR, CSC, COO, and DIA
- Benchmarks competitive with PETSc



SpMV



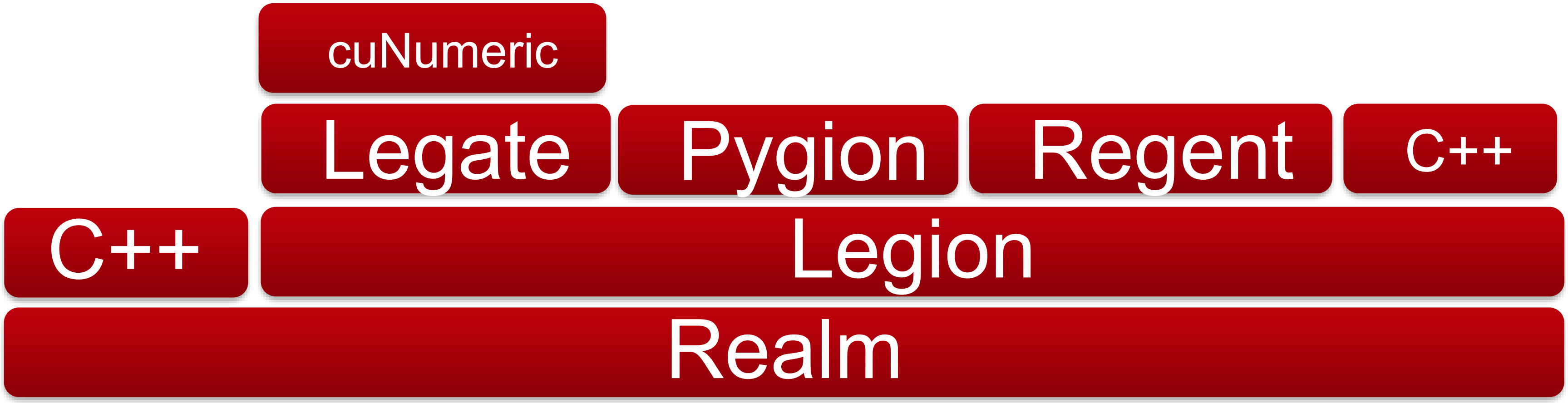
Conjugate Gradient Solver

- What if this still isn't enough?

Source: Lee, Wonchan. "cuNumeric and Legate: How to Create a Distributed, GPU-Accelerated Library",
23 March 2023, Nvidia GTC

What if cuNumeric and Legate aren’t enough?

- cuNumeric and Legate are built on top of Legion
- Legion is a:
  - Task-based
  - Data-centric
  - Programming model (supports multiple languages)
- Under the hood cuNumeric and Legate are implemented as a series of Legion tasks, but what does that mean?



cuNumeric

Legate

Pygion

Regent

C++

C++

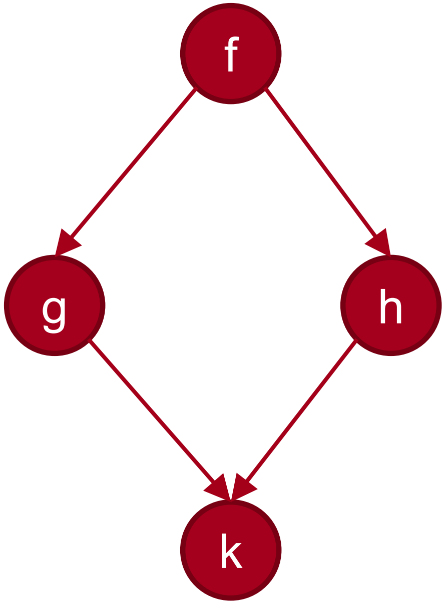
Legion

Realm

Tasks: The Big Idea (1/3)

- Big idea: write sequential code, let the system parallelize it

```
x = f()
y = g(x)
z = h(x)
k(y, z)
```

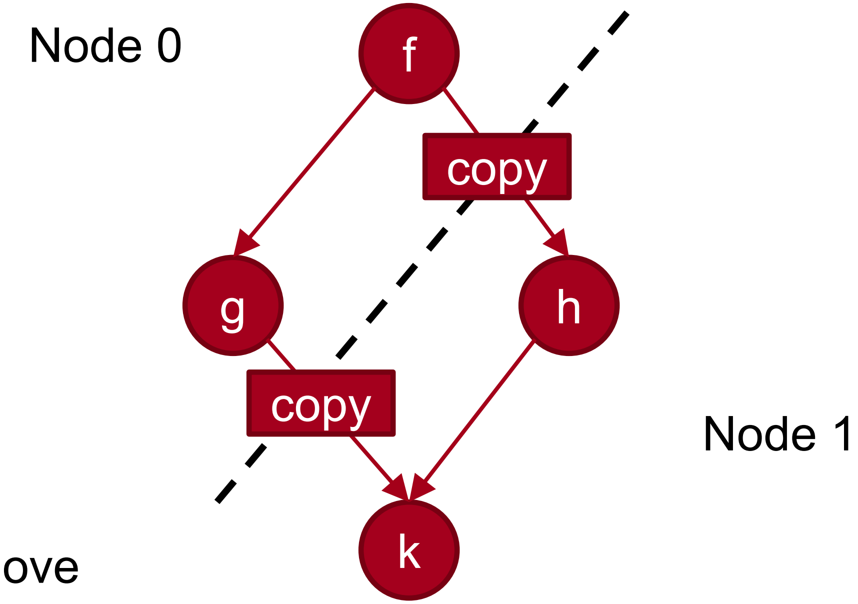


Sequential semantics means no way to get the synchronization wrong!

Tasks: The Big Idea (2/3)

- Big idea: write sequential code, let the system **distribute** it

```
x = f()
y = g(x)
z = h(x)
k(y, z)
```

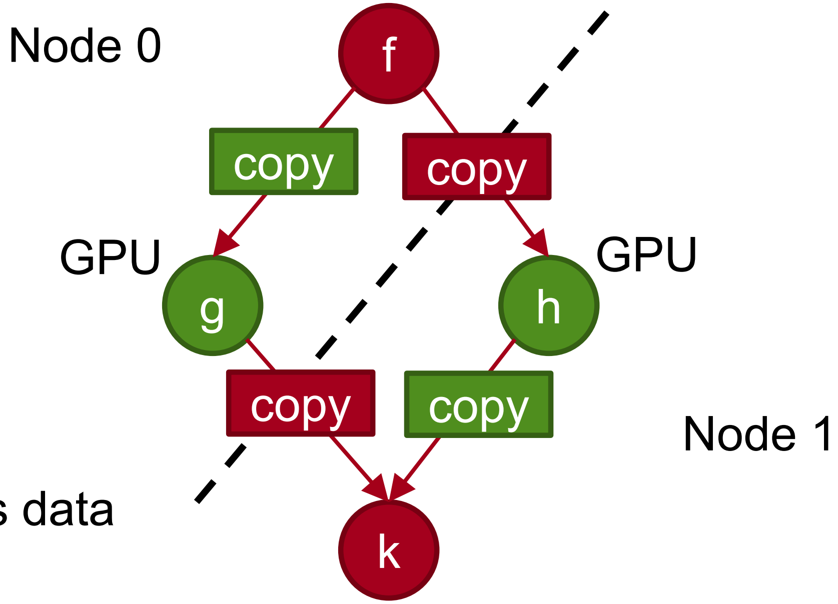


The system determines when messages need to be sent to move data between nodes

Tasks: The Big Idea (3/3)

- Big idea: write sequential code, let the system **accelerate** it

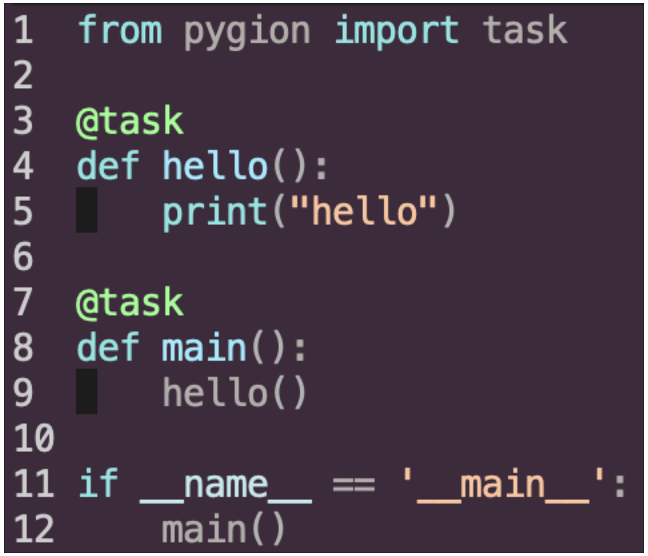
x = f()
y = g(x)
z = h(x)
k(y, z)



The system automatically moves data to/from GPU, no CUDA required

Pygion Basics

- We will describe the Legion programming model using bindings for Python called Pygion
- Concepts apply to C++ and Regent as well



```python
from pygion import task

@task
def hello():
    print("hello")

@task
def main():
    hello()

if __name__ == '__main__':
    main()
```

A task is a function

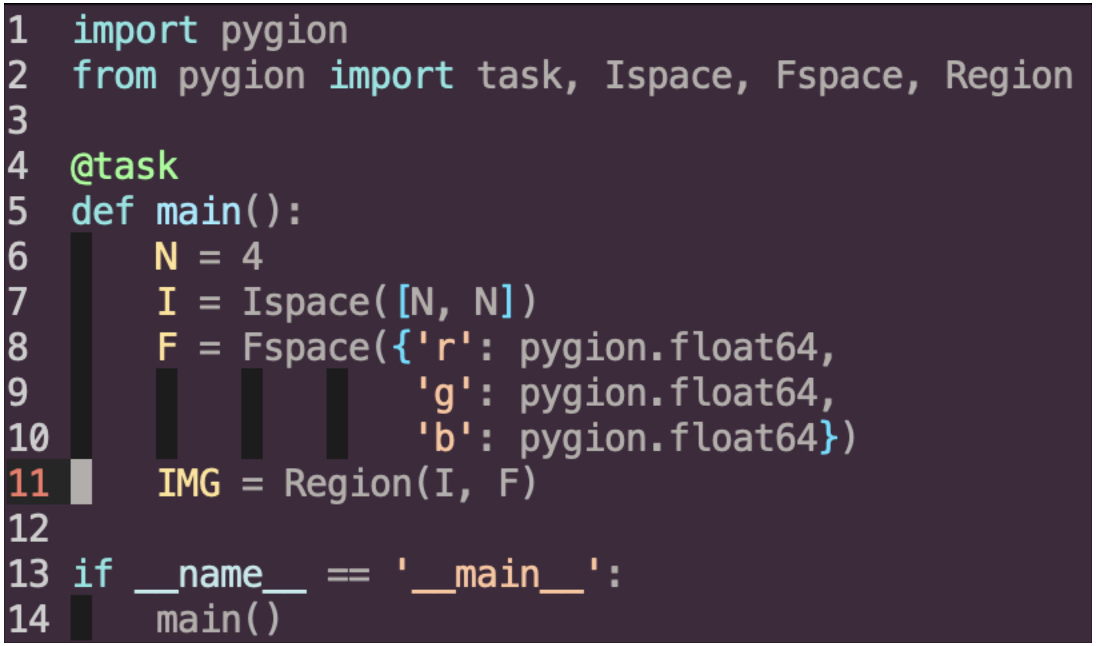
The bodies of tasks execute sequentially

Tasks call other tasks

Tasks can execute in parallel

Execution begins at main

Pygion: Regions

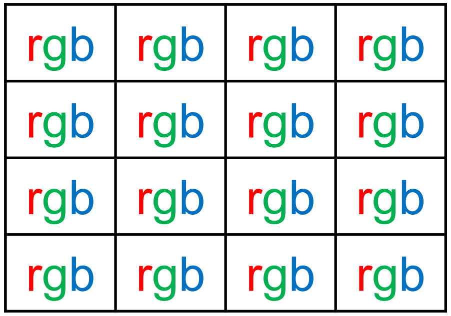


```python
import pygion
from pygion import task, Ispace, Fspace, Region

@task
def main():
    N = 4
    I = Ispace([N, N])
    F = Fspace({'r': pygion.float64,
                'g': pygion.float64,
                'b': pygion.float64})
    IMG = Region(I, F)

if __name__ == '__main__':
    main()
```

Data is stored in **regions**

Regions are like multi-dimensional arrays, have:

- set of indices (**ispace**)
- set of fields (**fspace**)



rgb	rgb	rgb	rgb
rgb	rgb	rgb	rgb
rgb	rgb	rgb	rgb
rgb	rgb	rgb	rgb

Ways Regions are Not Like Arrays

Regions can:

- Move between machines
- Move to CPU or GPU memory
- Have zero or more copies stored
- Have different layouts
- All of the above can change **dynamically**

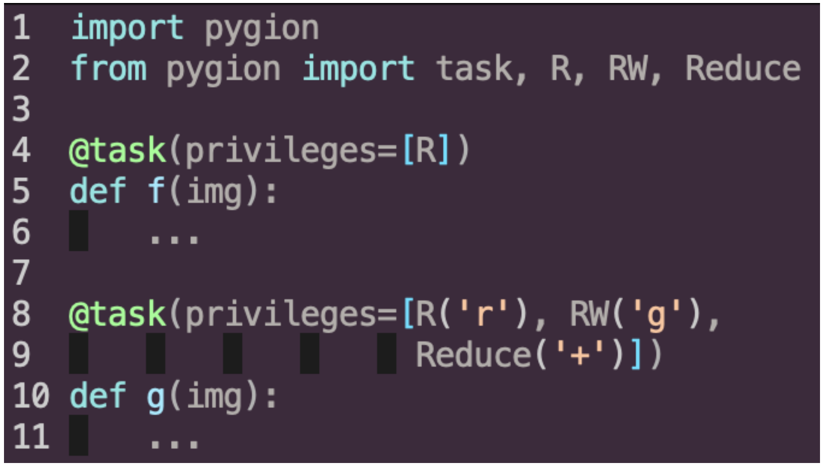
rgb	rgb	rgb	rgb
rgb	rgb	rgb	rgb
rgb	rgb	rgb	rgb
rgb	rgb	rgb	rgb

bgr	bgr	bgr	bgr
bgr	bgr	bgr	bgr
bgr	bgr	bgr	bgr
bgr	bgr	bgr	bgr

r	r	r	r	g	g	g	g	b	b	b	b
r	r	r	r	g	g	g	g	b	b	b	b
r	r	r	r	g	g	g	g	b	b	b	b
r	r	r	r	g	g	g	g	b	b	b	b

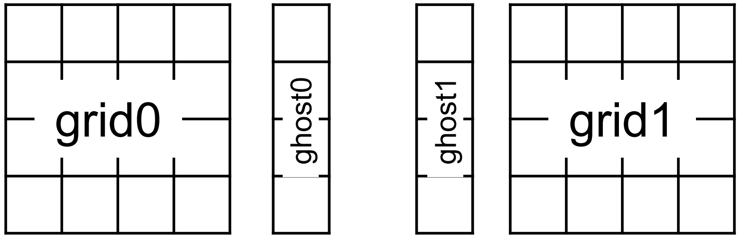
Pygion: Privileges

- Regions are passed to tasks **by reference**
- Must specify privileges used to access data
- Privileges include:
  - Read
  - Write
  - Reduce +, *, min, max, …
- Privileges can specify fields

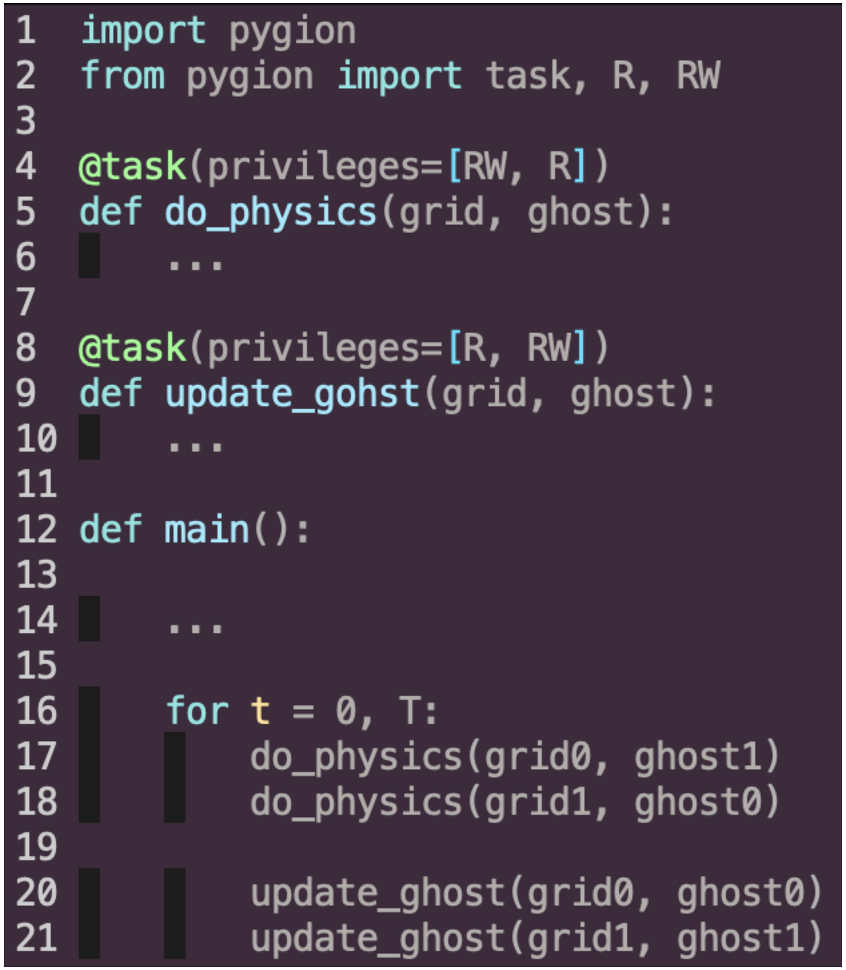


```
import pygion
from pygion import task, R, RW, Reduce

@task(privileges=[R])
def f(img):
    ...

@task(privileges=[R('r'), RW('g'),
                  Reduce('+')])
def g(img):
    ...
```

A Simple Timestep Loop in Pygion?



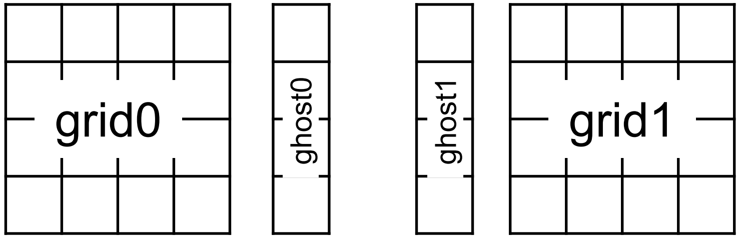
Note: this is **not** idiomatic Pygion



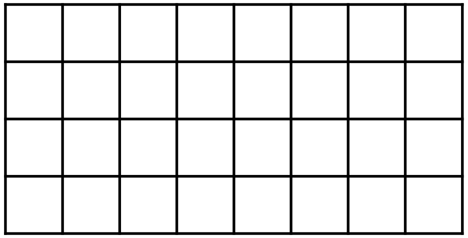
```
import pygion
from pygion import task, R, RW

@task(privileges=[RW, R])
def do_physics(grid, ghost):
    ...

@task(privileges=[R, RW])
def update_gohst(grid, ghost):
    ...

def main():

    ...

    for t = 0, T:
        do_physics(grid0, ghost1)
        do_physics(grid1, ghost0)

        update_ghost(grid0, ghost0)
        update_ghost(grid1, ghost1)
```

A Key Difference Between the Task-Based Systems

- How do you represent large grids?
  - Can't fit on a single node
- Other task-based systems (Dask):
  - Create a region for each subgrid
  - And also for each ghost/halo
- Pygion, Legion:
  - Create **one** region
  - And **partition** it

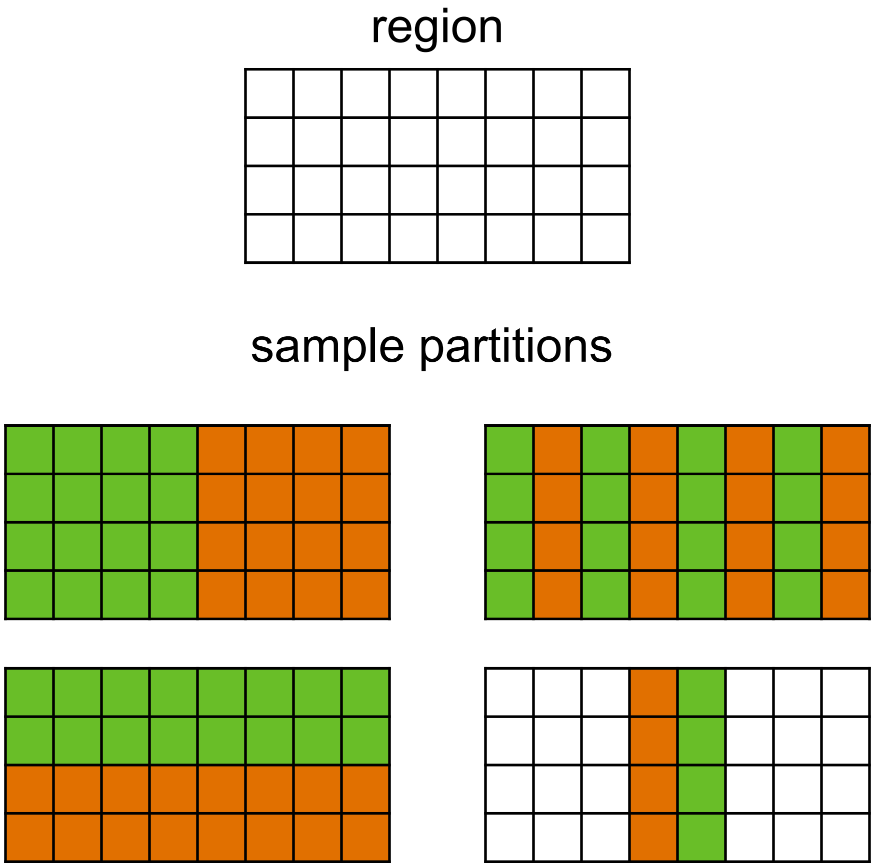


grid (the whole thing)



Pygion: Partitioning

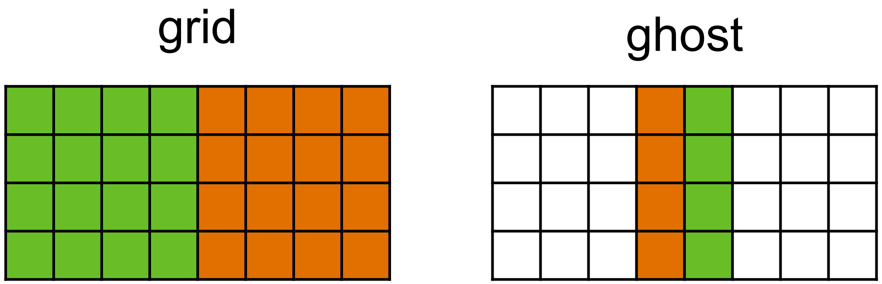
- Partitions divide regions into **subregions**
- Conceptually, a **coloring** on the region
- Important: subregions are **views**, not **copies**
  - As if there is only one copy of the region in memory



region

sample partitions

A Simple Timestep Loop in Pygion (with Partitioning)



These partition the same region

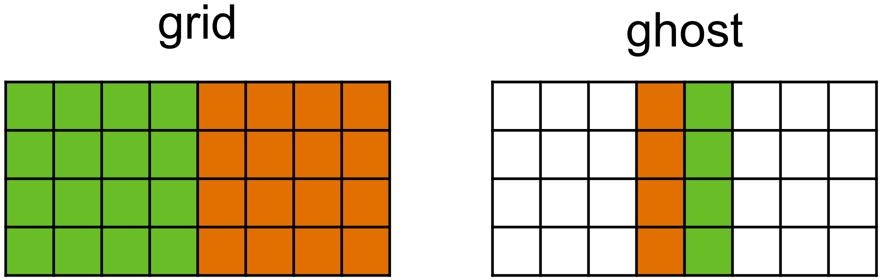
```
for t = 0, T do
  for c = 0, 1 do
    do_physics(grid[c], ghost[c])
  end

  for c = 0, 1 do
    update_ghost(grid[c])
  end
end
```

Launch a task per color

No more ghost region argument?

Because it refers to the same data, ghost is now updated automatically

A Simple Timestep Loop in Pygion (with Partitioning)



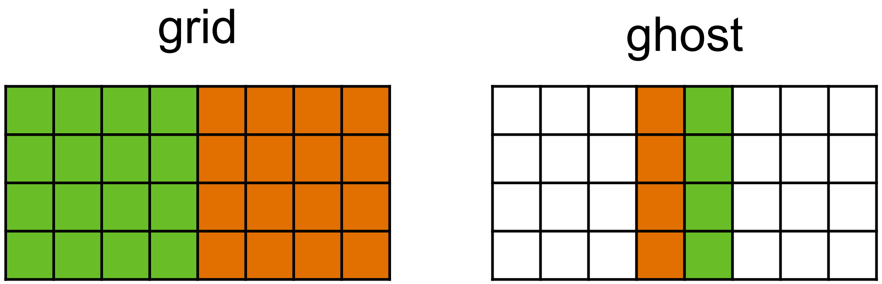
```
for t = 0, T do
  for c = 0, 1 do
    do_physics(grid[c], ghost[c])
  end

  for c = 0, 1 do
    update_ghost(grid[c])
  end
end
```

Privileges are updated to include fields

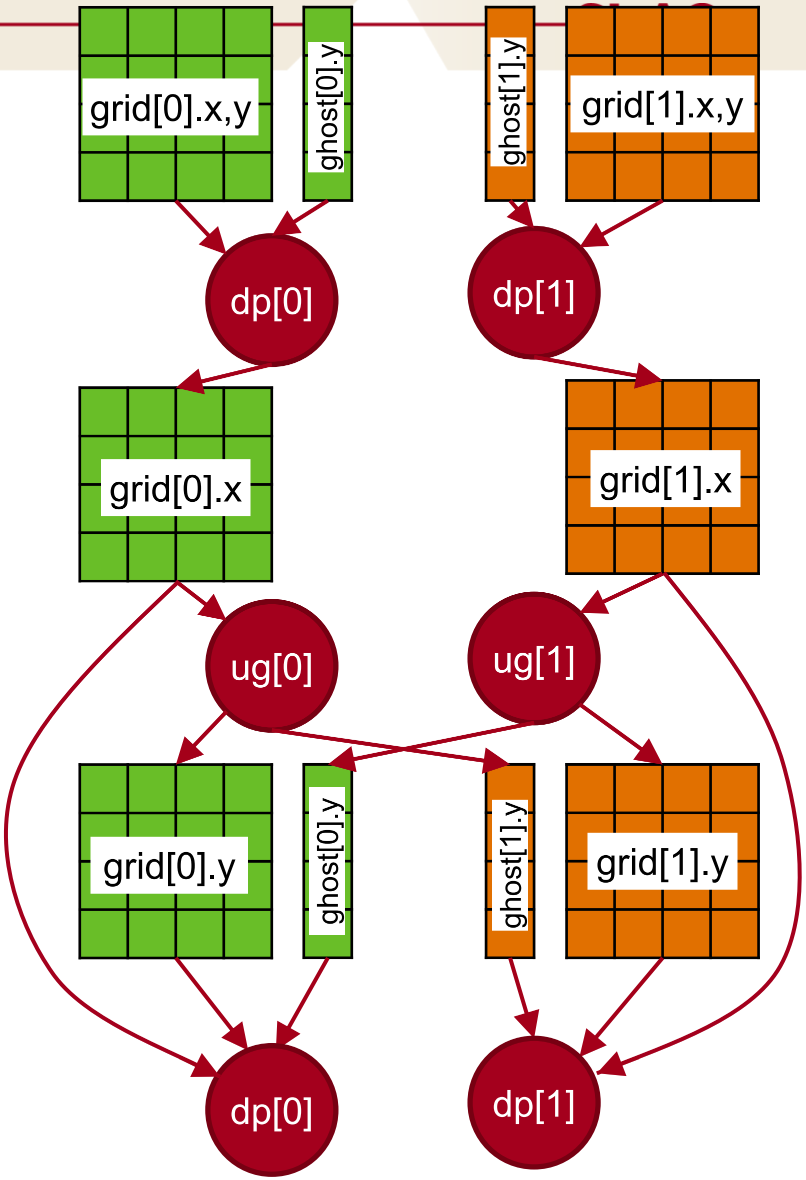
```
@task(privileges([RW('x'), R('y')])
def do_physics(grid, ghost):
…

@task(privileges([R('x'), RW('y')])
def update_ghost(grid):
…
```

Important: use different fields, otherwise tasks cannot run in parallel!

Timestep Loop: Execution



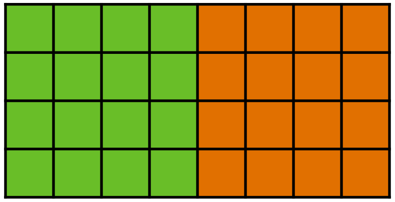
```
for t = 0, T do
  for c = 0, 1 do
    do_physics(grid[c],      ghost[c])
    --         W(x) R(y),    R(y)
  end

  for c = 0, 1 do
    update_ghost(grid[c]) -- W(y), R(x)
  end
end
```



More on Partitioning

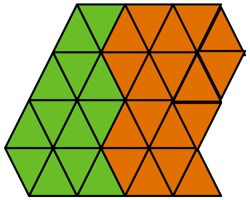
Equal partitioning

```
partition(equal, r,
    ispace(int2d, {2,1}))
```

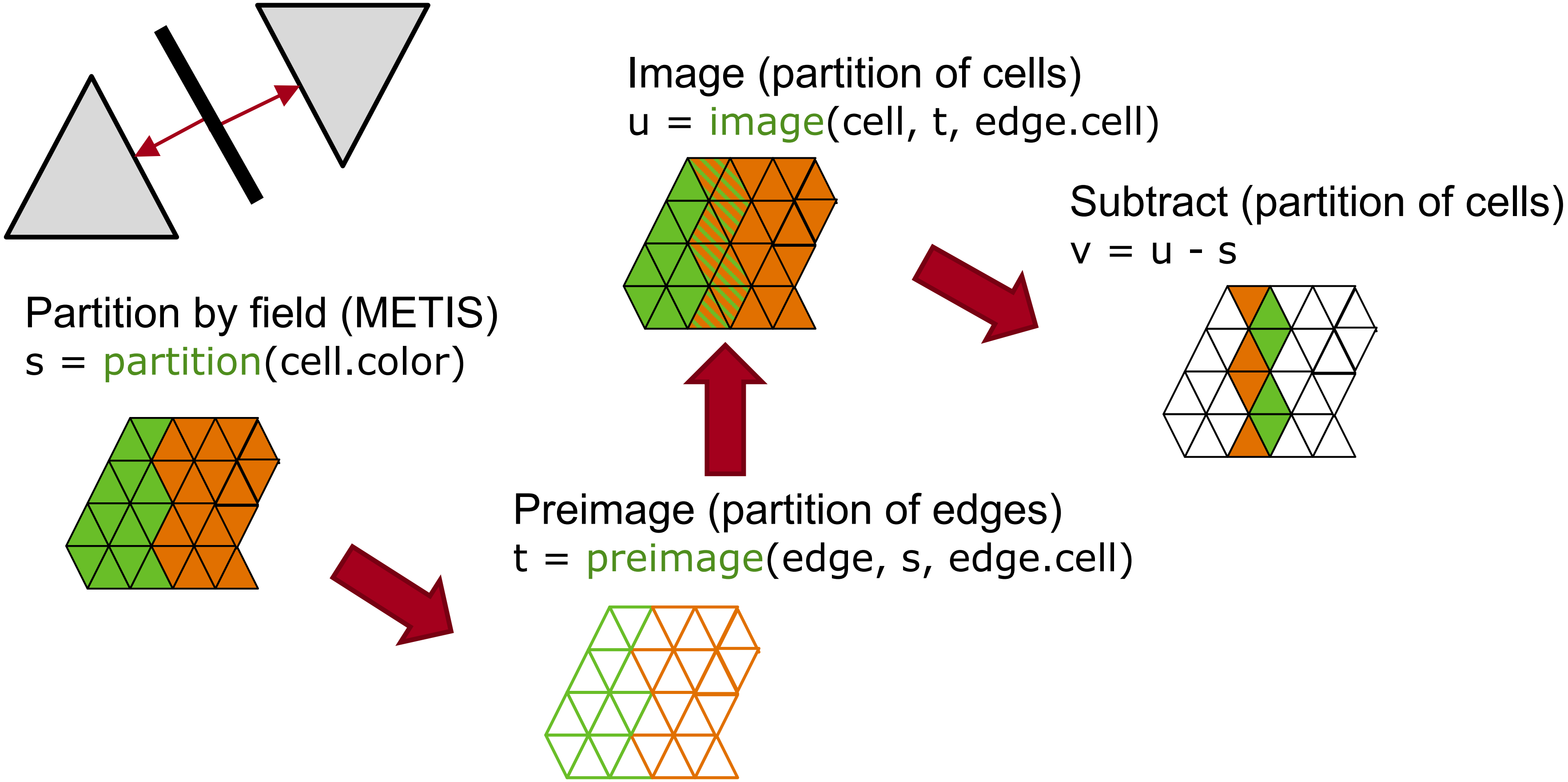


Partition by field (e.g., METIS)

```
run_metis(r) -- W(color)
partition(r.color,
    ispace(int1d, 2))
```



Dependent Partitioning



Partition by field (METIS)
s = partition(cell.color)

Preimage (partition of edges)
t = preimage(edge, s, edge.cell)

Image (partition of cells)
u = image(cell, t, edge.cell)

Subtract (partition of cells)
v = u - s

Pygion Examples: Stencil

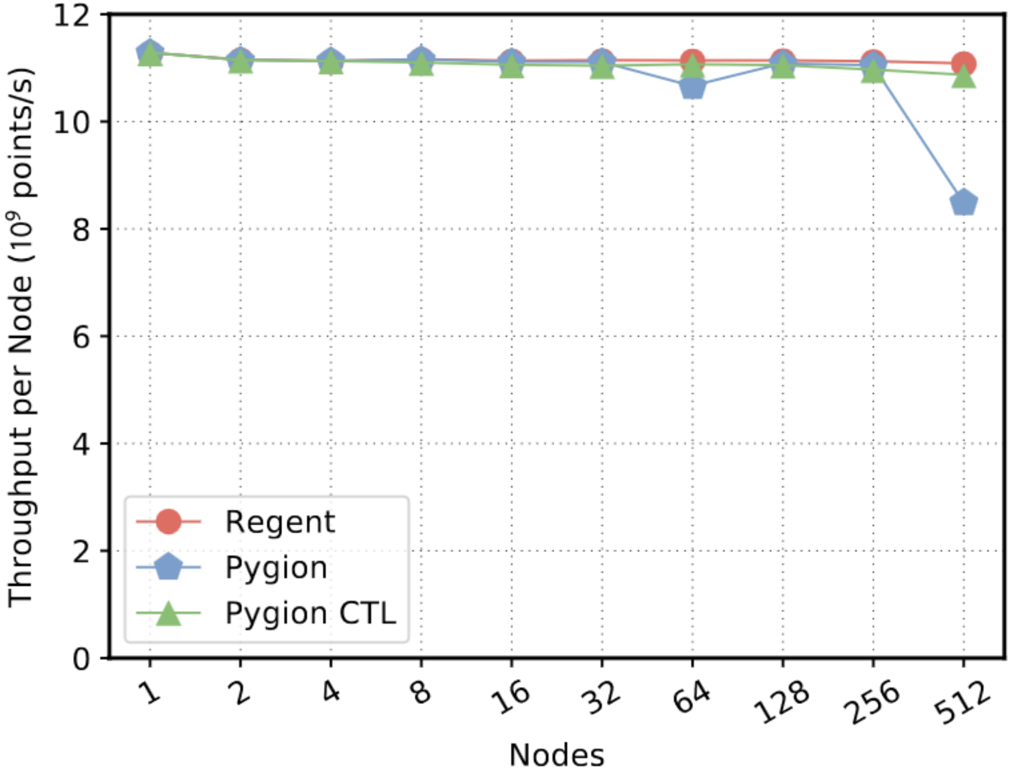


Fig. 1. Stencil weak scaling, 9×10^8 points/node.

Weak scaling Stencil on Piz Daint – main task ported to Pygion, leaf tasks in Regent

Source: Slaughter, Elliott. “Pygion: Flexible, Scalable Task-Based Parallelism with Python”, PAW-ATM 2019

Pygion Examples: Circuit

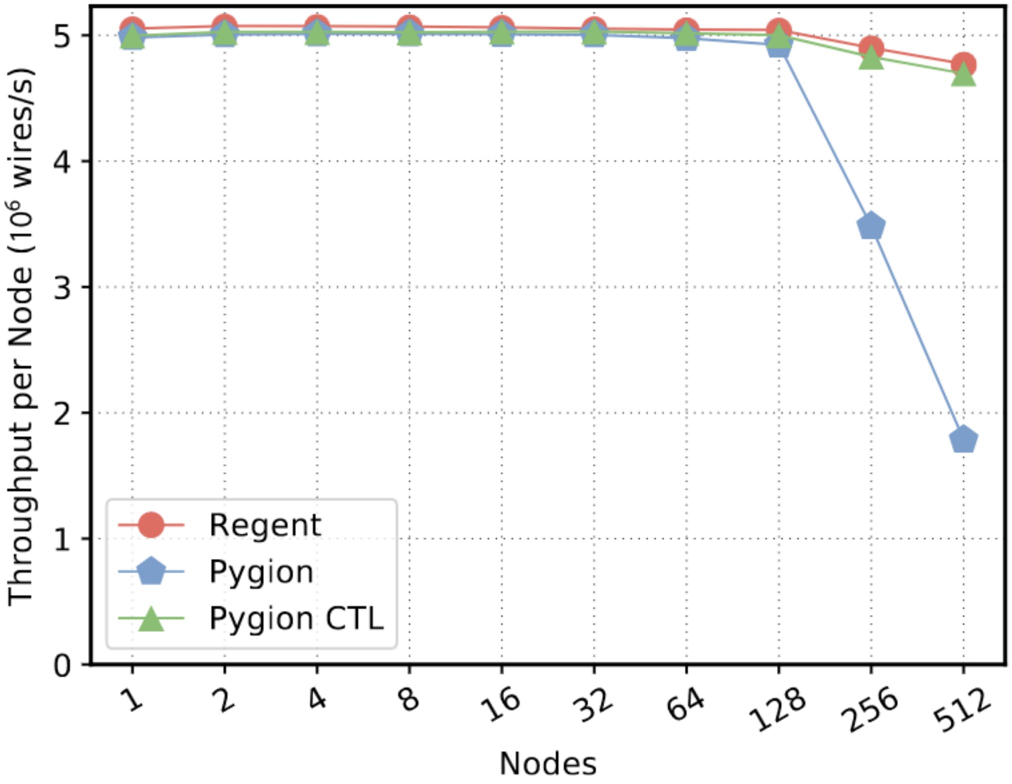


Fig. 2. Circuit weak scaling, 2×10^5 wires/node.

Weak scaling Circuit on Piz Daint – circuit simulation on unstructured graph – main task in Pygion, leaf tasks in Regent

Source: Slaughter, Elliott. “Pygion: Flexible, Scalable Task-Based Parallelism with Python”, PAW-ATM 2019

Pygion Examples: Pennant

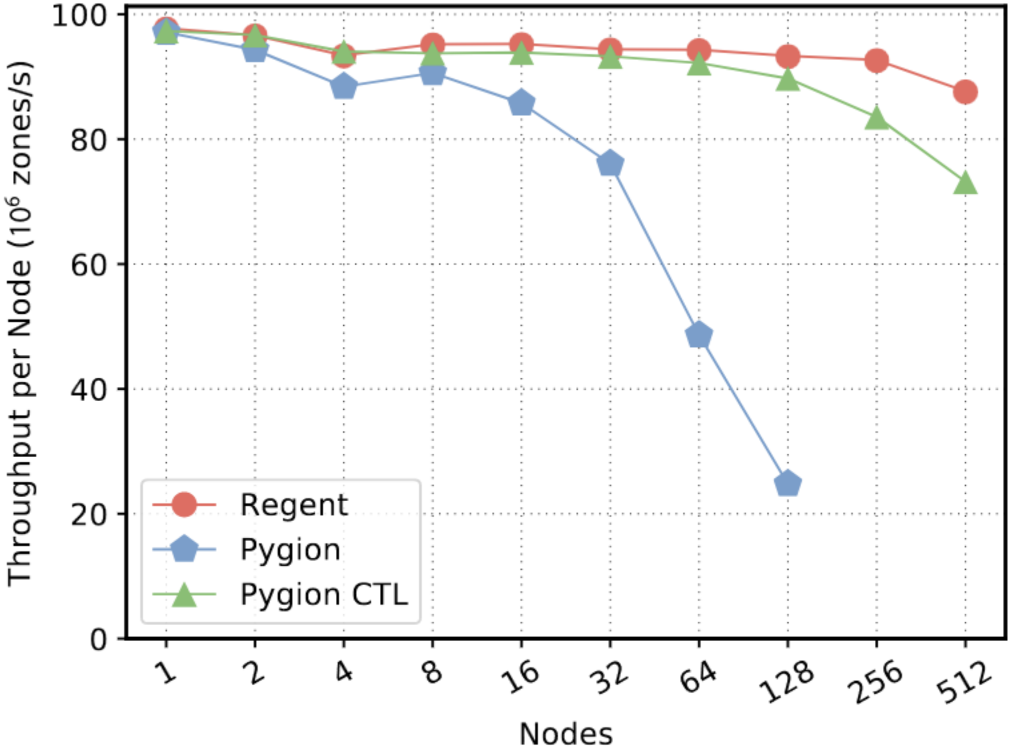


Fig. 3. Pennant weak scaling, 7.4×10^6 zones/node.

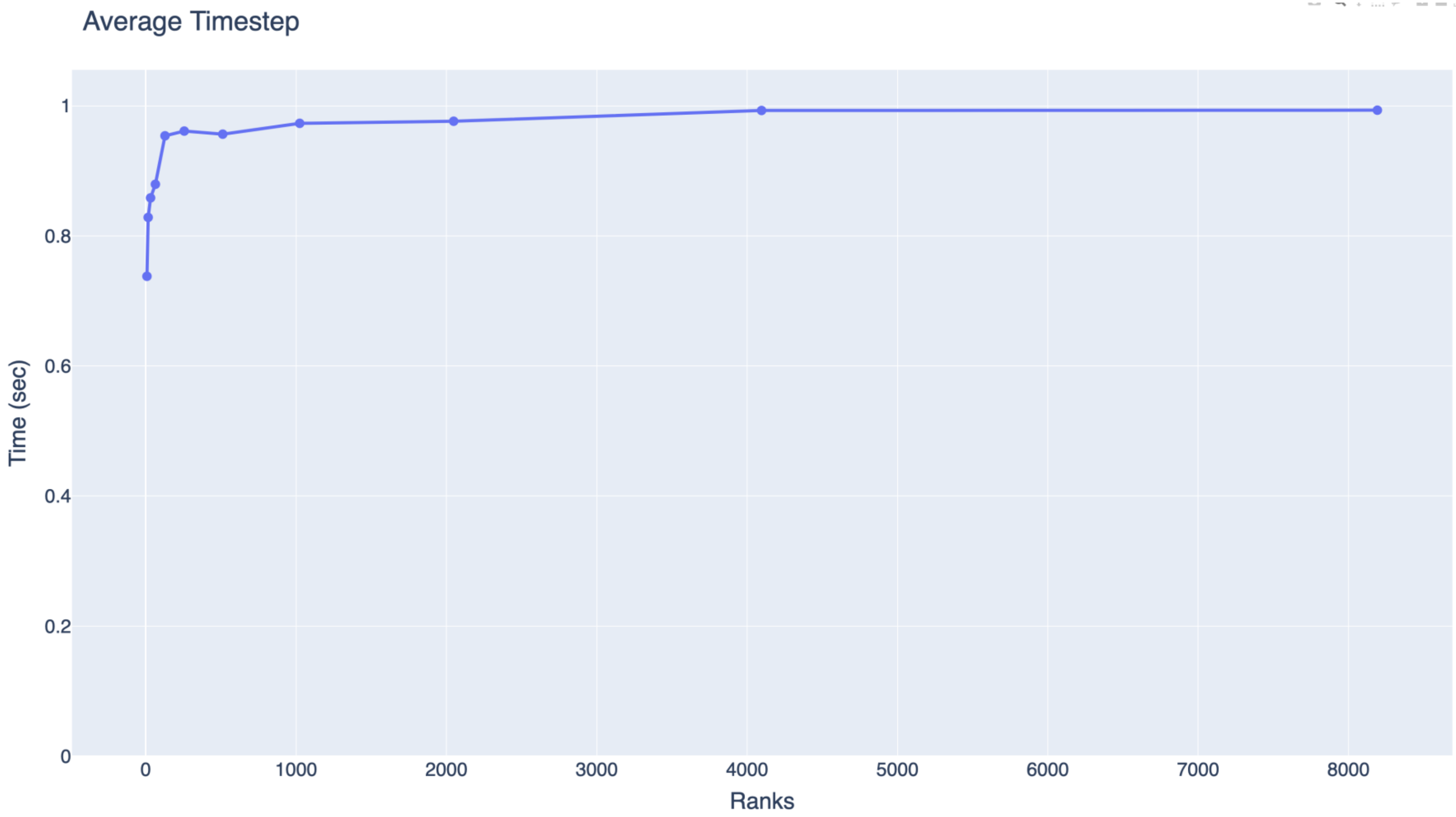
Weak scaling Pennant on Piz Daint – Lagrangian hydrodynamics simulation on 2D unstructured mesh – main task in Pygion, leaf tasks in Regent

Source: Slaughter, Elliott. “Pygion: Flexible, Scalable Task-Based Parallelism with Python”, PAW-ATM 2019

Pygion GPU support

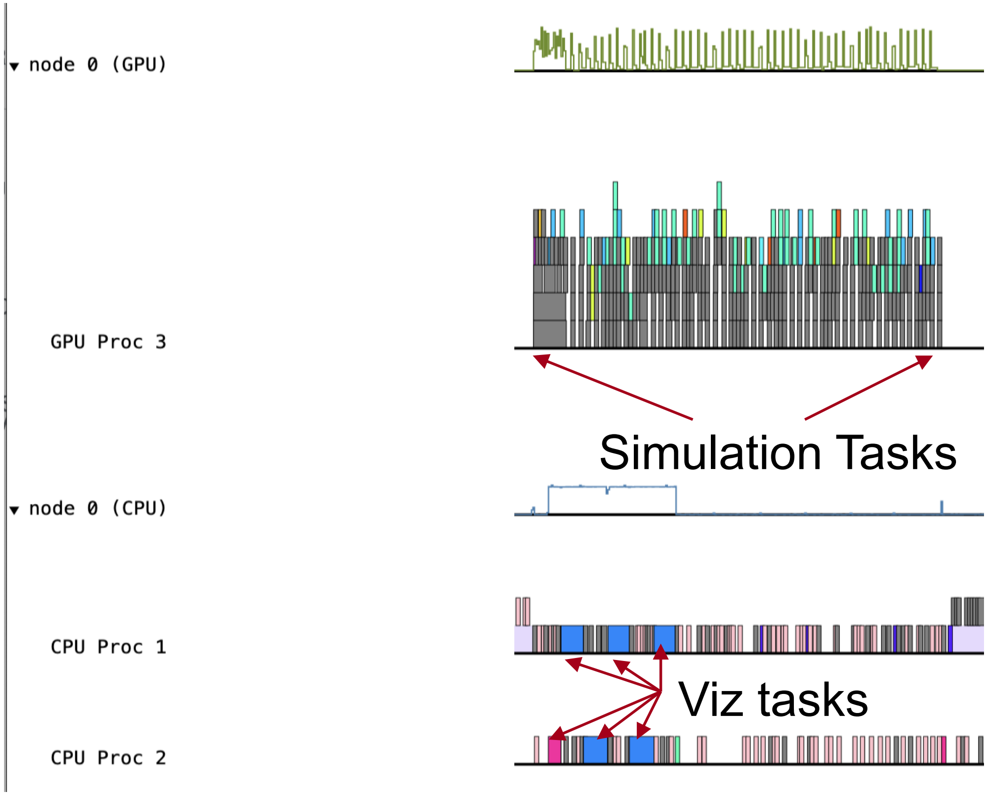
- Previous three weak-scaling examples used GPUs with Python
- Regent is a compiler built on top of Lua and Terra
- First class support for Legion programming model
- Regent can generate code for AMD and Nvidia GPUs with Intel support coming soon
- Regent tasks can be called from Pygion
- Underlying Legion runtime is the same regardless of language user space application is written in
  - Pygion and Regent have similar scaling properties

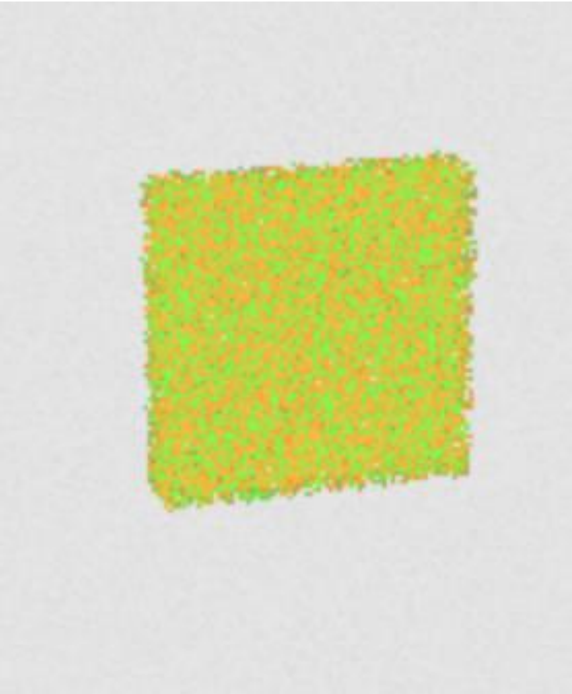
Regent Example: S3D



Weak scaling S3D on Frontier - Direct Numerical Simulation of Turbulent Combustion

S3D In-situ Visualization





- Legion runtime allows for efficient use of resources
- Viz tasks on CPU perfectly overlapped with simulation tasks on GPU

How does visualization have no impact on simulation?

- Tasks can register different variants CPU, GPU
- Mapper API allows users to select:
  - Which processor a task executes on, CPU or GPU
  - Make copies of data (known as instances in Legion terms)
  - Select which memories instances live in sys mem, framebuffer, zero-copy, …
  - Select what layout instances have
  - And more!
- Mapper allows for portability between machines

Summary

- Legion is a task-based data-centric parallel programming model
- Legion ecosystem provides different options for writing portable and scalable HPC applications:
  - cuNumeric – drop-in, distributed, GPU-accelerated replacement for NumPy
  - Pygion – Python Bindings for Legion
  - Regent – Language with first class support for Legion and GPU code generation
  - C++ - Can always drop back to this if needed

SLAC