Task-Based Runtimes and Applications

Elliott Slaughter

Staff Scientist, Computer Science Division, SLAC

CME 213 lecture 2023-06-07

U.S. DEPARTMENT OF ENERGY | Stanford University

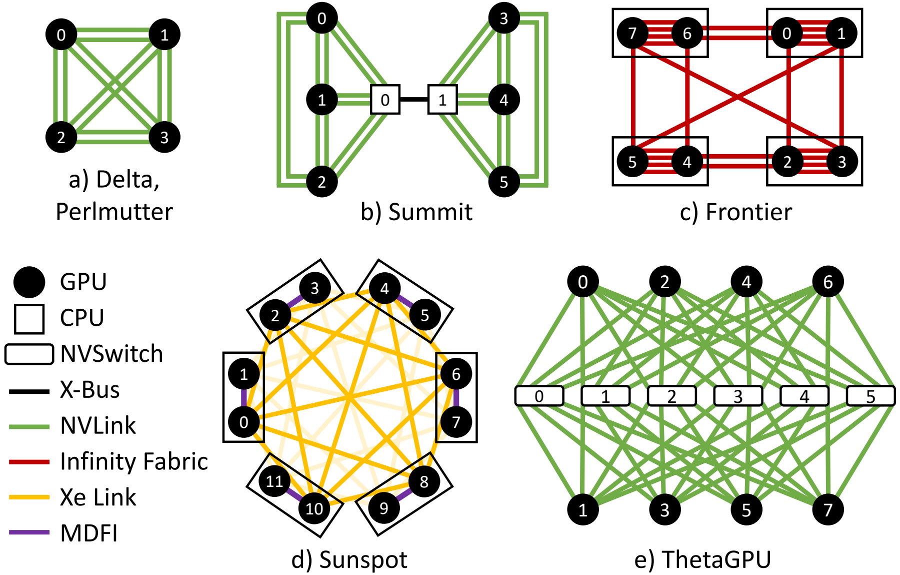
SLAC NATIONAL ACCELERATOR LABORATORY

About Me

- Stanford CS PhD, 2017 (with Alex Aiken)
- SLAC CS research group since 2017

Today’s HPC Landscape

- Power efficiency concerns are driving all next-generation supercomputers to accelerators
- U.S. Department of Energy (DOE) machines:
  - Perlmutter (NERSC): NVIDIA GPUs
  - Frontier (OLCF): AMD GPUs
  - Aurora (ALCF): Intel GPUs
- How to program these machines?



a) Delta, Perlmutter

b) Summit

c) Frontier

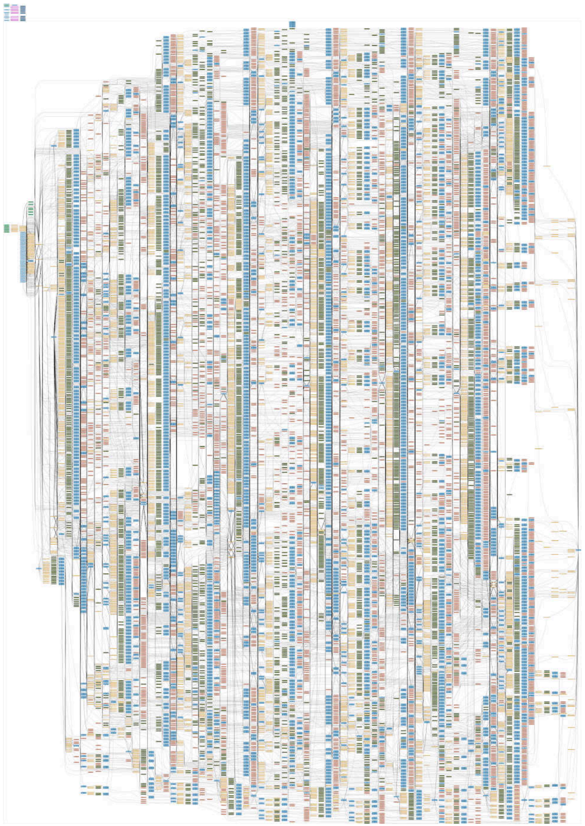
d) Sunspot

e) ThetaGPU

Hidayetoglu, et al. *CommBench: A Communications Tool for Benchmarking Multi-GPU, Multi-NIC Networks with Group-to-Group Patterns*. In submission.

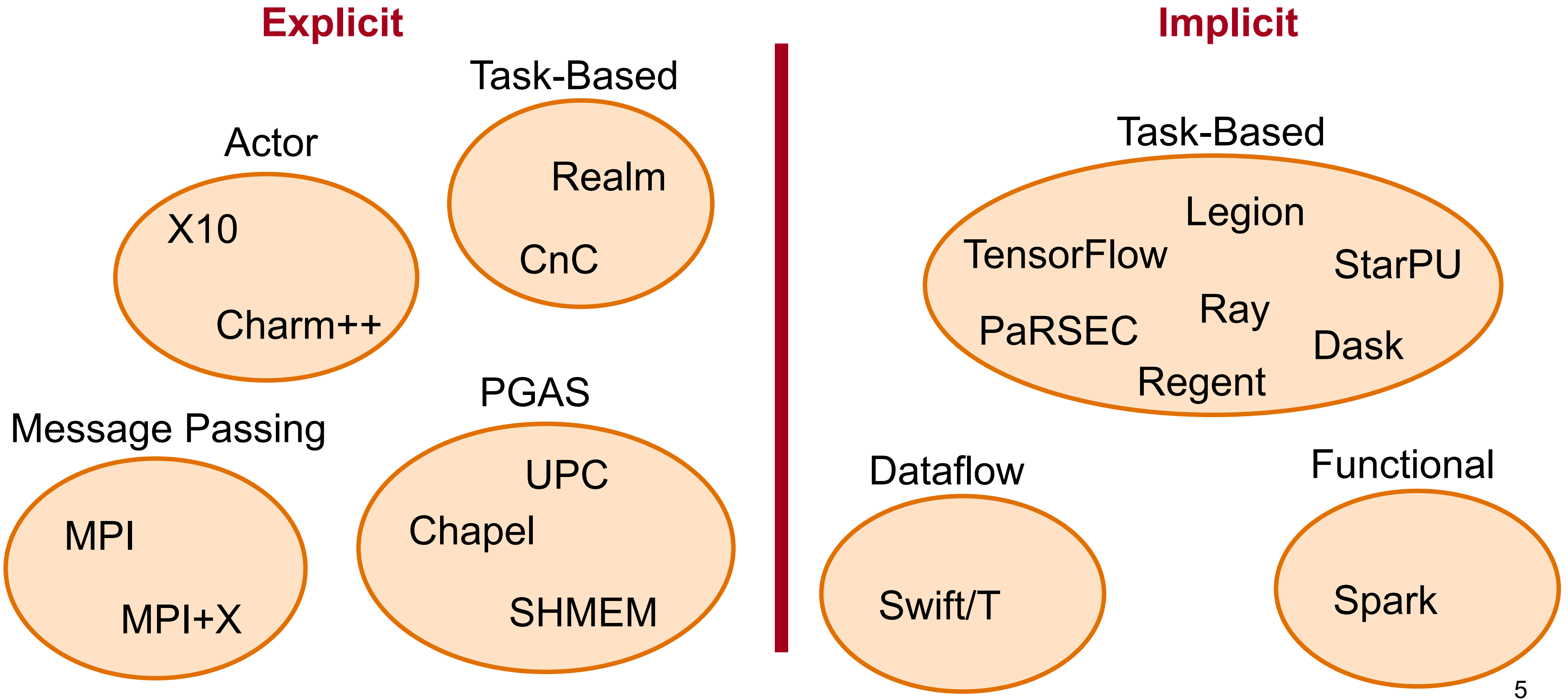
The Good (and Bad) News About Parallelism

- As machines get bigger and more complex, need more parallelism
- Applications already have a large (and growing) amount of untapped parallelism…
- Traditional programming models don't allow us to capture this
- How do we expose it?



At right: dependence graph of S3D, a direct numerical simulation of turbulent combustion

Welcome to the Programming Model Zoo



Explicit

Task-Based

Realm

CnC

Actor

X10

Charm++

PGAS

UPC

Chapel

SHMEM

Message Passing

MPI

MPI+X

Implicit

Task-Based

Legion

TensorFlow

StarPU

Ray

PaRSEC

Dask

Regent

Dataflow

Swift/T

Functional

Spark

This Lecture

- Overview of a task-based system (**Regent**)
- Applications that would not be possible without a task-based system:
  - Zero-effort parallelization of Python NumPy programs (**cuNumeric**)
  - Near zero-effort checkpointing (**Relight**)

Part 1

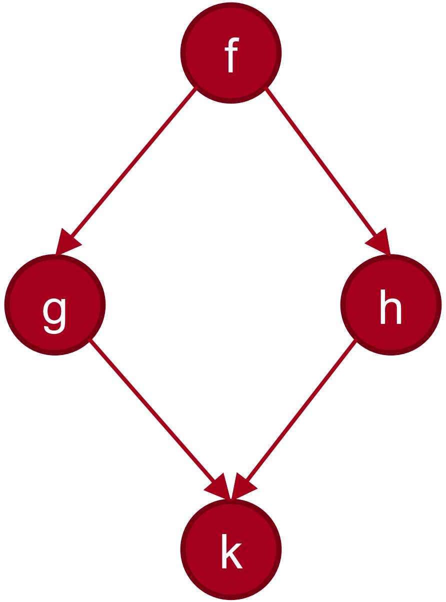
Regent

Task-based programs give sequential semantics to parallel, distributed computation

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)
```

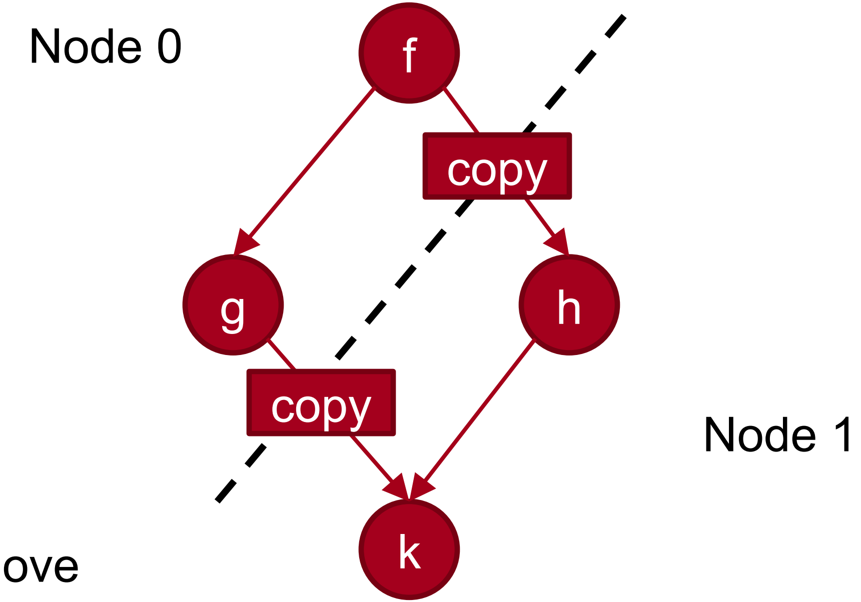


Sequential semantics means no way to get the synchronization wrong!

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)
```

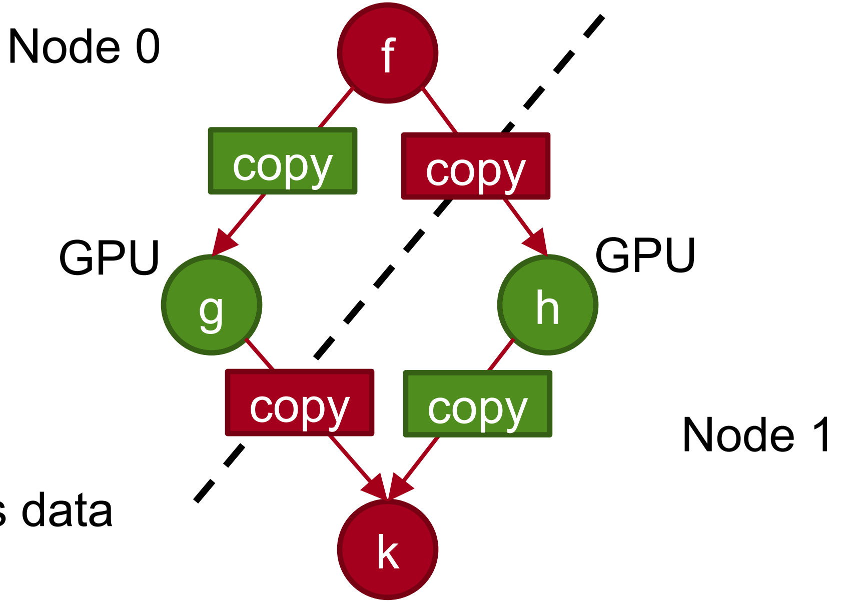


The system determines when messages need to be sent to move data between nodes

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

Implementations of Tasks

In HPC:

- Legion (**Regent**), StarPU, PaRSEC (*covered in this lecture)
- Realm, HPX, OCR, CnC, Uintah, …

Elsewhere:

- TensorFlow, Pytorch
- Dask, Ray
- Spark

Regent Basics

- This lecture will use Regent syntax
- But concepts apply to other task-based systems (PaRSEC, StarPU)

```
task hello()
  println("hello")
end

task main()
  hello()
end
```

A task is a function

The bodies of tasks execute sequentially

Tasks call other tasks

Execution begins at `main`

Regent: Regions

```
fspace rgb {
  r : float, g : float, b : float
}

task main()
  var N = 4
  var grid = ispace(int2d, {N, N})
  var img = region(grid, rgb)
end
```

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

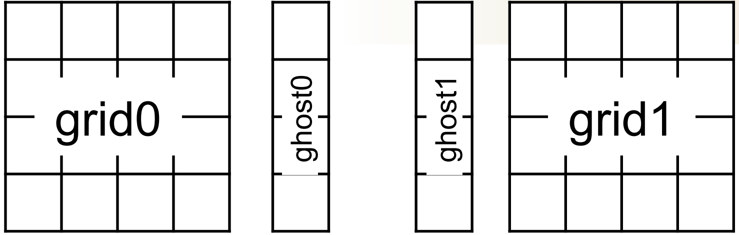
Regent: Privileges

- 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

```
task f(img : region(rgb))
where reads(img)
do ... end
```

```
task g(img : region(rgb))
where reads(img.r),
      writes(img.g),
      reduces max(img.b)
do ... end
```

A Simple Timestep Loop in Regent?



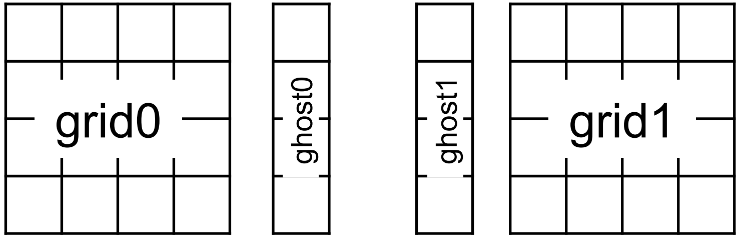
```
for t = 0, T do
  do_physics(grid0, ghost1)
  do_physics(grid1, ghost0)

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

```
task do_physics(
    grid : region(...),
    ghost : region(...))
where reads writes(grid),
      reads(ghost)
do ... end

task update_ghost(
    grid : region(...),
    ghost : region(...))
where reads(grid),
      writes(ghost)
do ... end
```

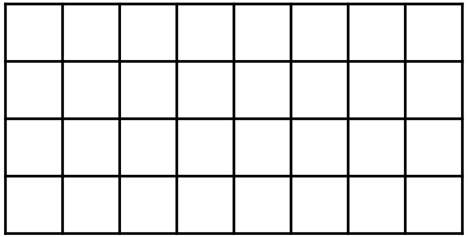
Note: this is idiomatic PaRSEC, StarPU
But **not** Regent

A Key Difference Between the Task-Based Systems

- How do you represent large grids?
  - Can’t fit on a single node
- StarPU, PaRSEC:
  - Create a region for each subgrid
  - And also for each ghost/halo
- Regent, Legion:
  - Create **one** region
  - And **partition** it



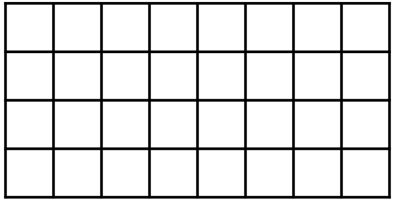
grid (the whole thing)



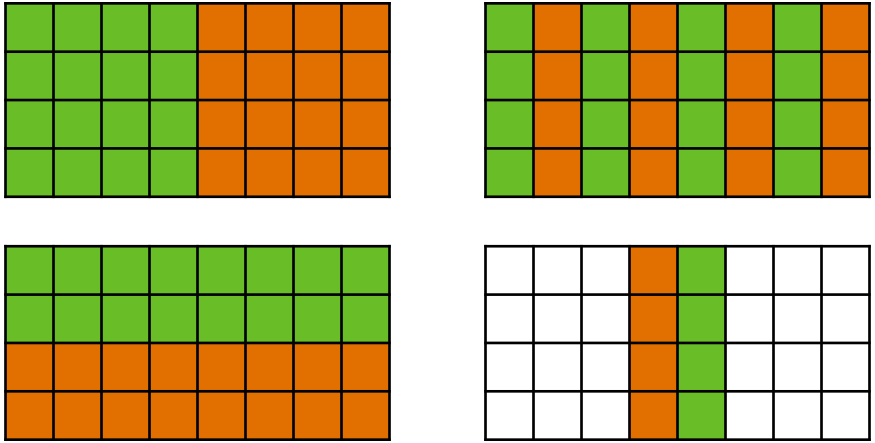
Regent: Partitioning

- 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

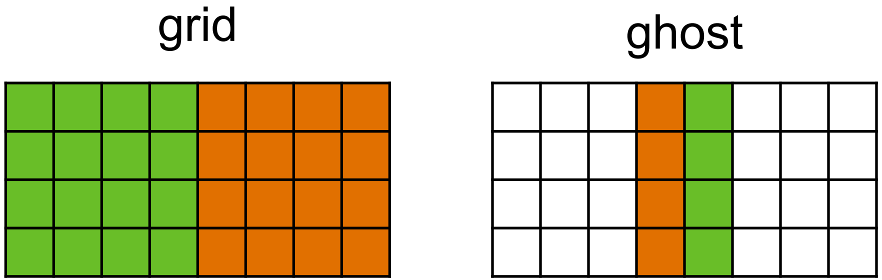
region



sample partitions



A Simple Timestep Loop in Regent (with Partitioning)



These partition the same region

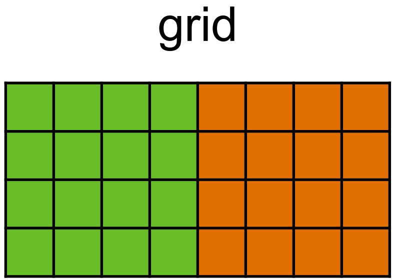
```
for t = 0, T do
  for c = 0, 2 do
    do_physics(grid[c], ghost[c])
  end

  for c = 0, 2 do
    update_ghost(grid[c])
  end
end
```

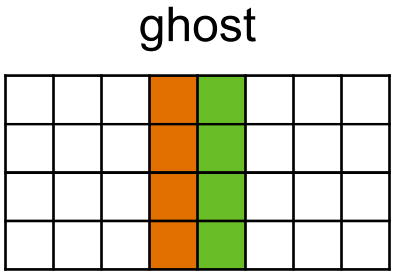
Launch a task per color

No more ghost region argument?

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

A Simple Timestep Loop in Regent (with Partitioning)





```
for t = 0, T do
  for c = 0, 2 do
    do_physics(grid[c], ghost[c])
  end

  for c = 0, 2 do
    update_ghost(grid[c])
  end
end
```

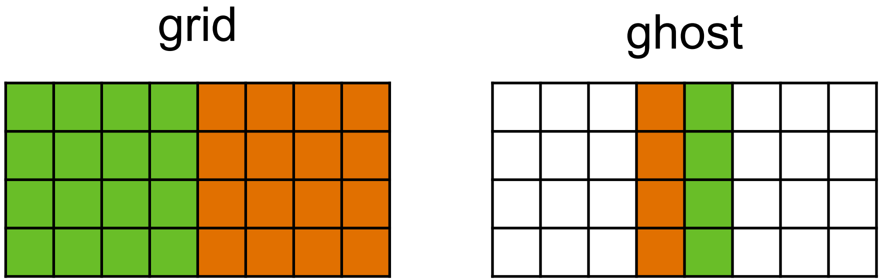
Privileges are updated to include fields

```
task do_physics(
    grid : region(...),
    ghost : region(...))
where writes(grid.x),
      reads(grid.y, ghost.y)
do ... end
```

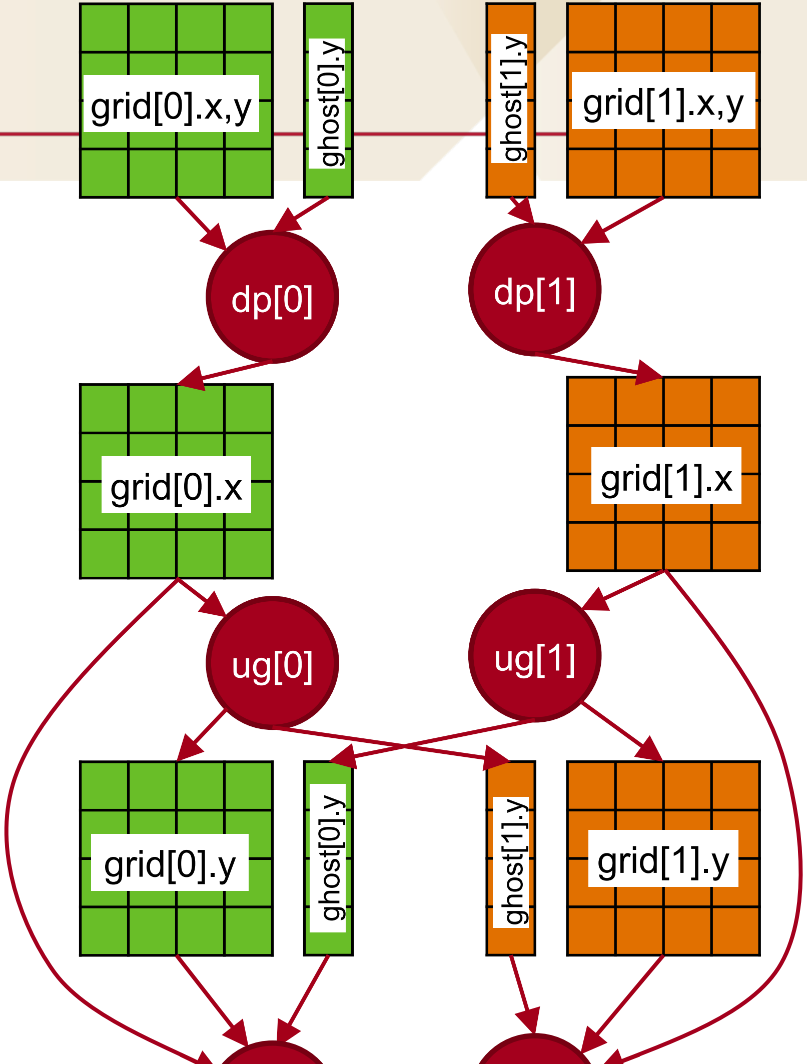
Important: use different fields, otherwise tasks cannot run in parallel!

```
task update_ghost(
    grid : region(...))
where reads(grid.x),
      writes(grid.y)
do ... end
```

Timestep Loop: Execution



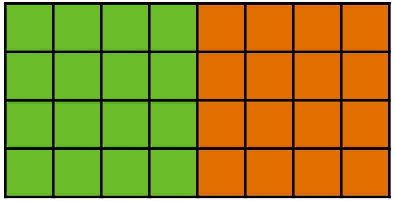
```
for t = 0, T do
  for c = 0, 2 do
    do_physics(grid[c],      ghost[c])
    --         W(x) R(y),    R(y)
  end

  for c = 0, 2 do
    update_ghost(grid[c]) -- W(y), R(x)
  end
end
```



More on Partitioning

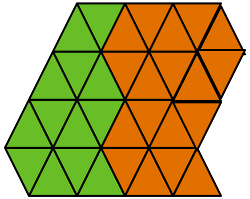
Equal partitioning

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

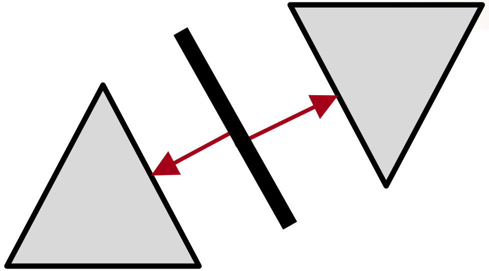


Partition by field (e.g., METIS)

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

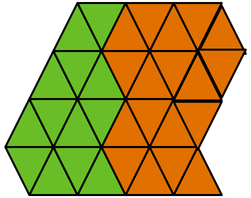


Dependent Partitioning



Partition by field (METIS)

```
s = partition(cell.color)
```



Preimage (partition of edges)

```
t = preimage(edge, s, edge.cell)
```

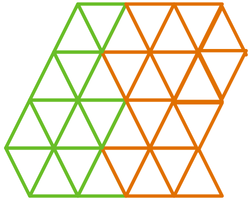
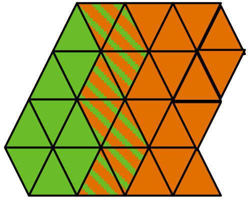


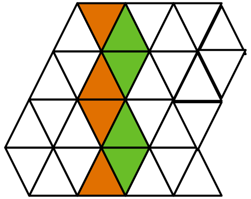
Image (partition of cells)

```
u = image(cell, t, edge.cell)
```



Subtract (partition of cells)

```
v = u - s
```



Regent Optimization: Index Launches

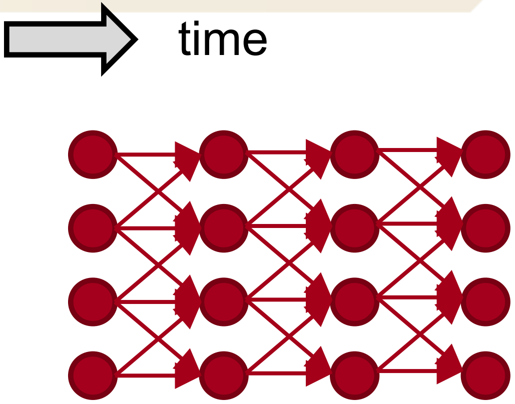
```
for t = 0, T do
  for c = 0, 4 do -- index launch
    do_physics(grid[c], ghost[c])
  end

  for c = 0, 4 do -- index launch
    update_ghost(grid[c])
  end
end
```

These loops are index launches

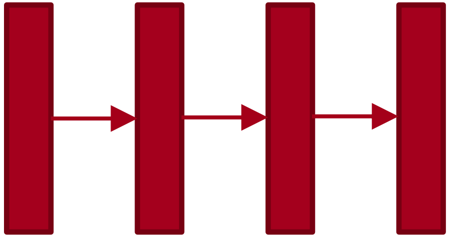
This is an automatic optimization, no input required by the user

Regent Optimization: Index Launches

```
for t = 0, T do
  for c = 0, 4 do -- index launch
    do_physics(grid[c], ghost[c])
  end

  for c = 0, 4 do -- index launch
    update_ghost(grid[c])
  end
end
```



Without optimization

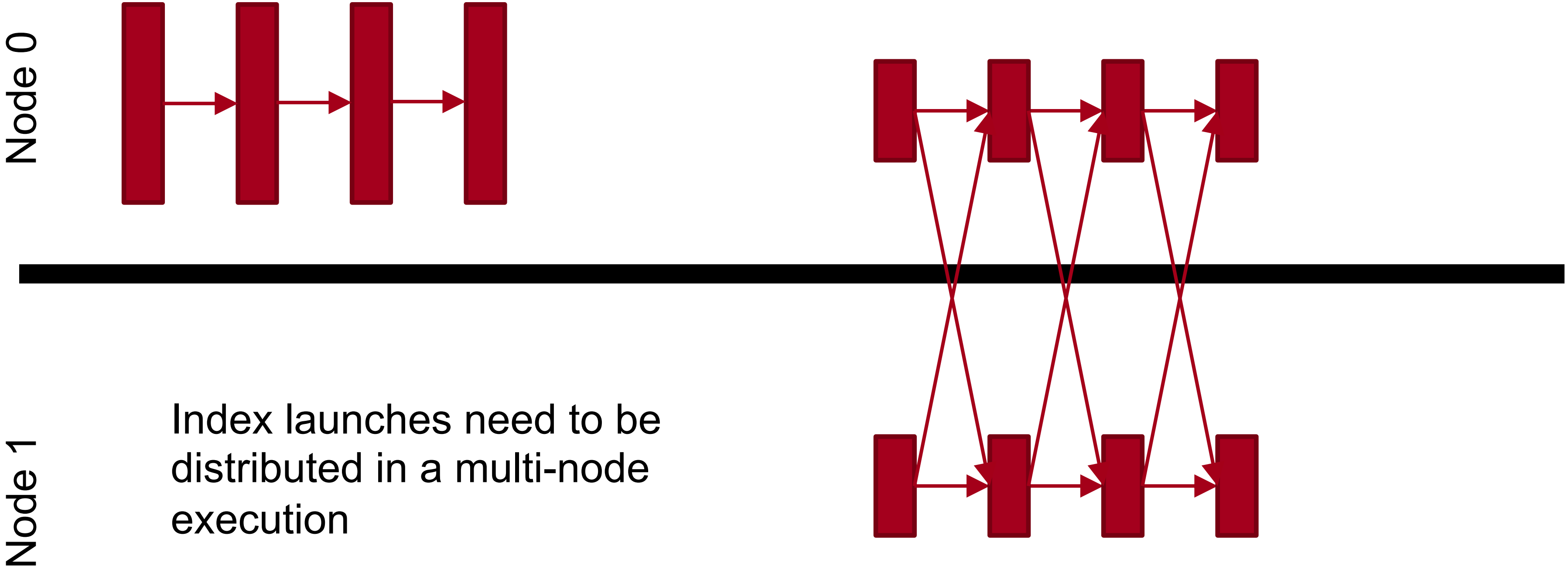


With optimization

Index launches reduce overhead in the runtime

Regent Optimization: Control Replication

(Without optimization)

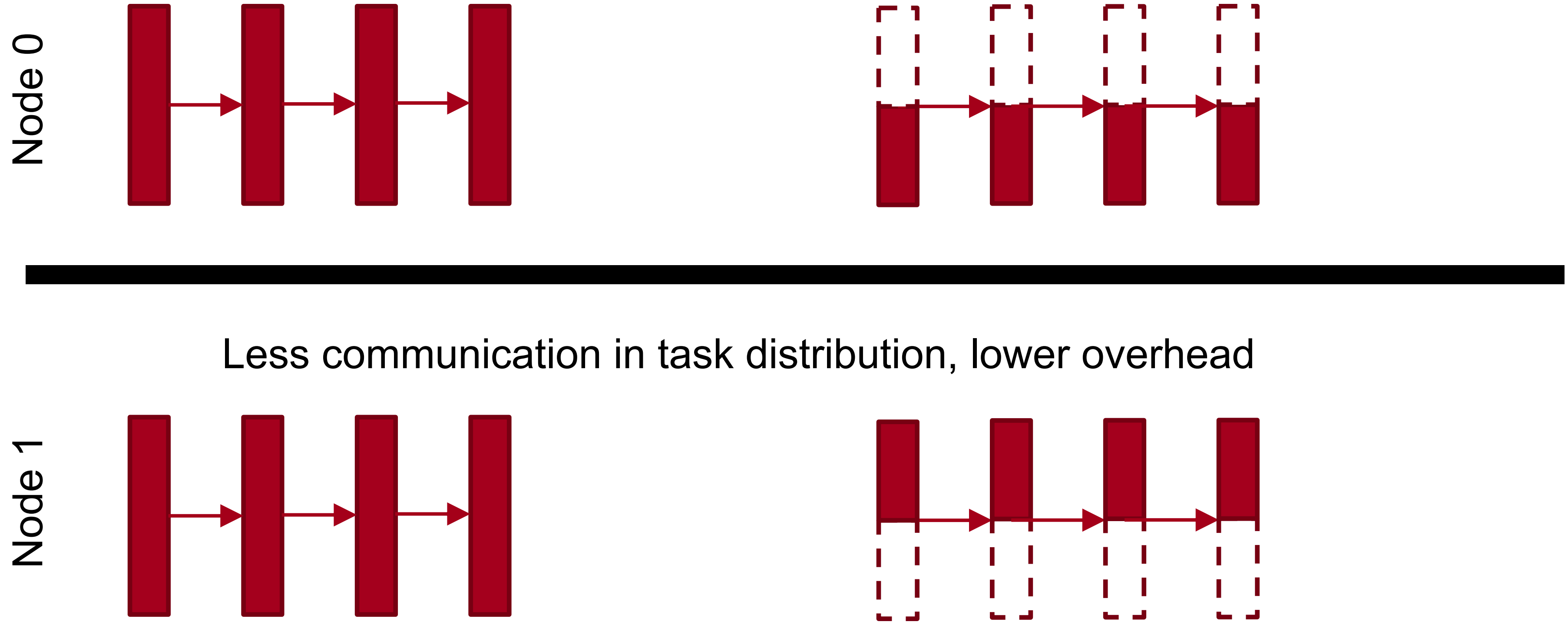


Index launches need to be distributed in a multi-node execution

This can be inefficient

Regent Optimization: Control Replication

(With optimization)



Less communication in task distribution, lower overhead

(Nearly) automatic optimization in Regent programs

Control Replication in StarPU, PaRSEC

- No control replication optimization in StarPU, PaRSEC
- Why?
  - No partitions: no way to reason about global data distribution
  - No index launches: no way to reason about global task distribution

```
for t = 0, T do
  if rank == 0 then
    do_physics(grid0, ghost1)
  end
  if rank == 1 then
    do_physics(grid1, ghost0)
  end
...
```

StarPU, PaRSEC programs need to manually filter tasks for efficient execution

Regent/Legion avoid this via partitioning and optimizations (index launches, control replication)

Using the GPU: Regent Code Generation

```
__demand(__cuda)
task do_physics(grid : region(...))
where ... do
  for cell in grid do
    cell.x = ...
  end
end
```

One line to get automatic GPU code generation in Regent, no CUDA required

Summary: Regent

Pros:

- Write sequential code, run in parallel
  - And distributed
  - And GPU
- No synchronization bugs
- Automatically asynchronous, automatic data movement

Cons:

- More explicit about data partitioning, tasks
  - For the system to help you, you need to tell it more about what you're doing

Part 2

cuNumeric

Write NumPy, get GPU + distributed for free

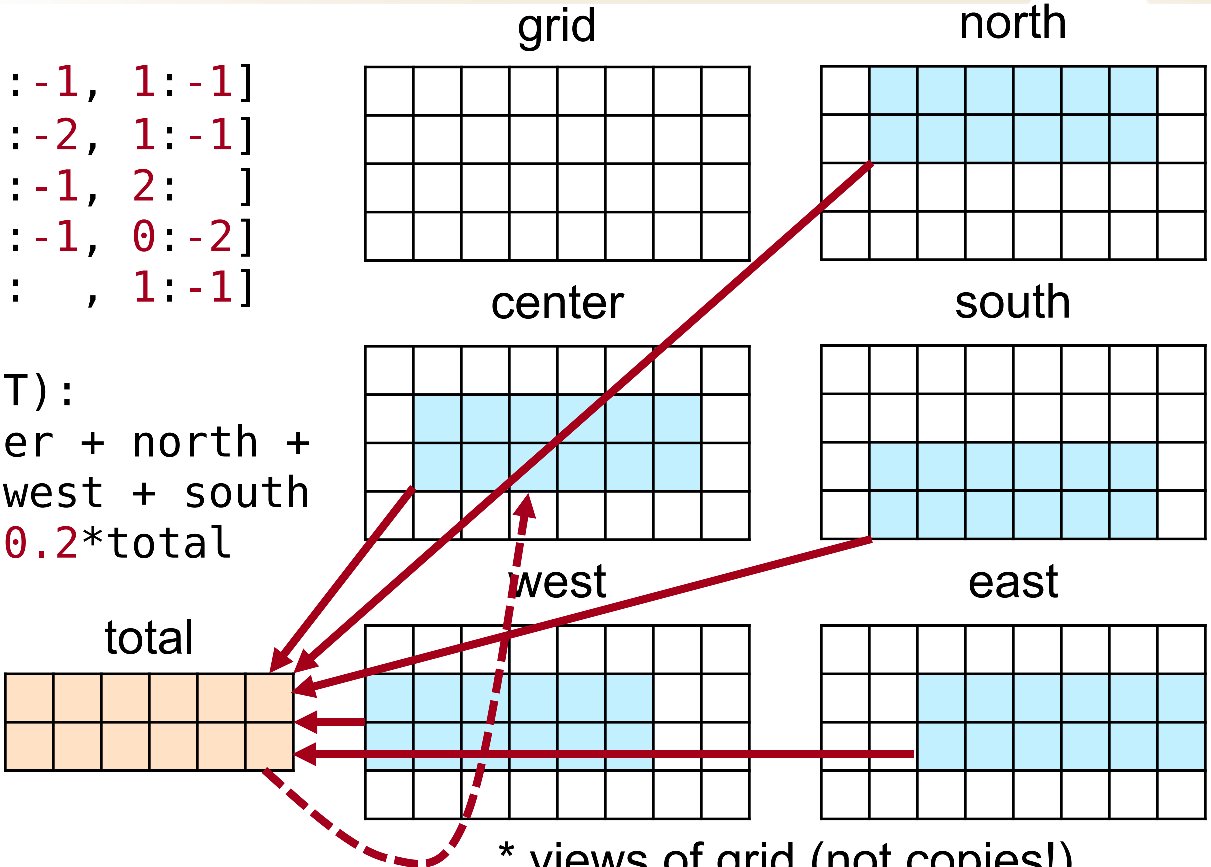
Why Python? Why NumPy?

- Most domain scientists are not experts in distributed programming
  - They didn’t take CME 213!
- Choices:
  - Write in the language you know (e.g., Python)
  - Learn MPI + CUDA + …
  - Not everyone has that much time to invest
- Python is slow
  - Enter NumPy: library functions to make Python faster for array computations

```
center = grid[1:-1, 1:-1]
north  = grid[0:-2, 1:-1]
east   = grid[1:-1, 2:  ]
west   = grid[1:-1, 0:-2]
south  = grid[2:  , 1:-1]

for t in range(T):
    total = center + north +
        east + west + south
    center[:] = 0.2*total
```

NumPy Stencil Example

```
center = grid[1:-1, 1:-1]
north  = grid[0:-2, 1:-1]
east   = grid[1:-1, 2:  ]
west   = grid[1:-1, 0:-2]
south  = grid[2:  , 1:-1]

for t in range(T):
    total = center + north +
            east + west + south
    center[:] = 0.2*total
```



* views of grid (not copies!)

NumPy and cuNumeric

NumPy Pros:

- Pure Python
- Rich API: many operators, rich indexing

NumPy Cons:

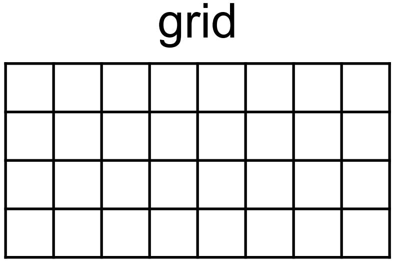
- CPU-only (often single-threaded)
- GPU requires a separate library
- Not distributed

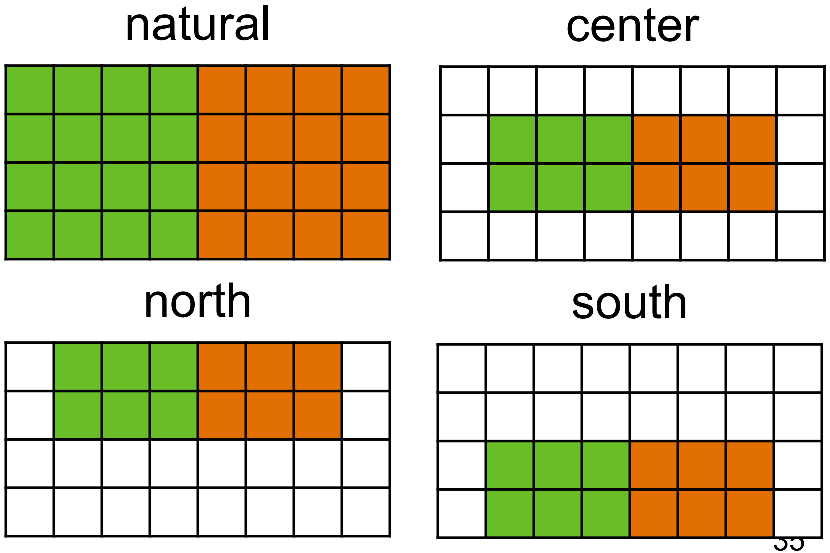
cuNumeric:

- NumPy on Legion
  - “Change one line …”
  - Product by NVIDIA
- NumPy programs have sequential semantics!
- Runs on:
  - Multi-core CPU
  - GPU
  - Distributed (CPU+GPU)

cuNumeric: Regions and Partitions

- Every array *shape* becomes a region
  - Arrays are fields on the region (an optimization to reduce overheads)
- Views become partitions
- Every region also has a “natural” partition for parallelism





```
center = grid[1:-1, 1:-1]
north  = grid[0:-2, 1:-1]
east   = grid[1:-1, 2:  ]
west   = grid[1:-1, 0:-2]
south  = grid[2:  , 1:-1]
```

cuNumeric: Turning Operations into Tasks

- Each NumPy operation becomes a task
- Operations launch in sequential order (Legion handles dependencies)

```
center = grid[1:-1, 1:-1]
north  = grid[0:-2, 1:-1]
east   = grid[1:-1, 2:  ]
west   = grid[1:-1, 0:-2]
south  = grid[2:  , 1:-1]

for t in range(T):
   total = center + north +
        east + west + south
   center[:] = 0.2*total
```

Create partitions

Tasks:

add_task(center, north, tmp)

add_task(tmp, east, tmp)

add_task(tmp, west, tmp)

add_task(tmp, south, tmp)

cuNumeric: Parallelism

- Tasks are split for parallelism to match partitions

```
center = grid[1:-1, 1:-1]
north  = grid[0:-2, 1:-1]
east   = grid[1:-1, 2:  ]
west   = grid[1:-1, 0:-2]
south  = grid[2:  , 1:-1]

for t in range(T):
   total = center + north +
        east + west + south
   center[:] = 0.2*total
```

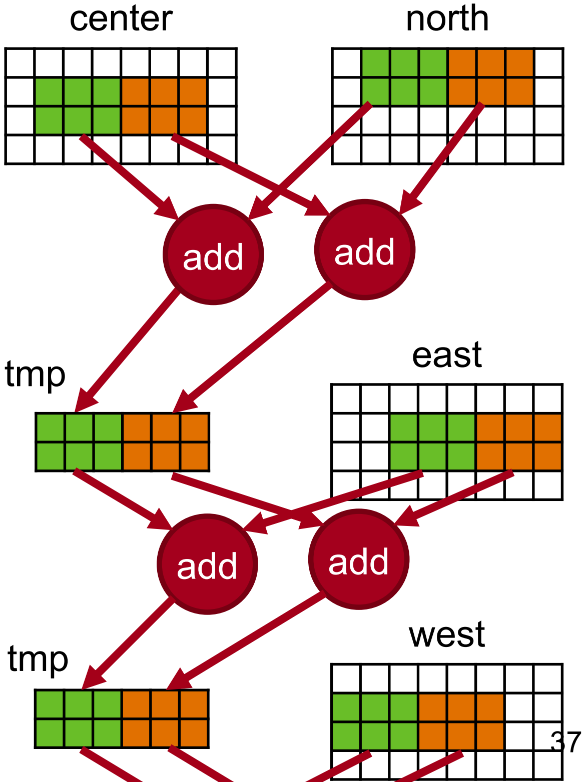
Tasks:

add_task(center, north, tmp)

add_task(tmp, east, tmp)

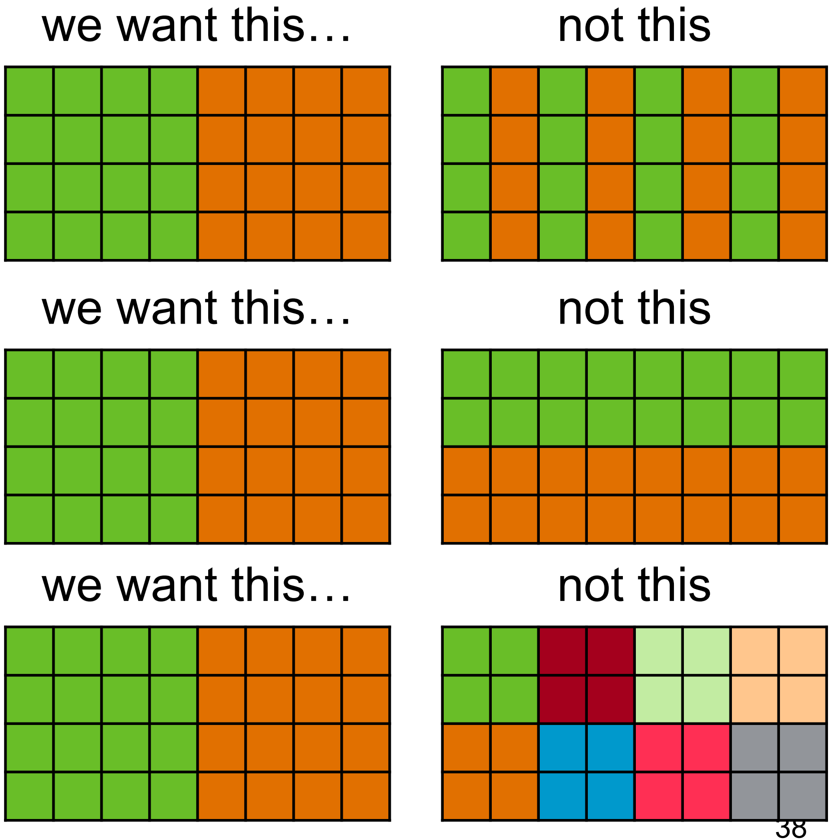
add_task(tmp, west, tmp)

add_task(tmp, south, tmp)



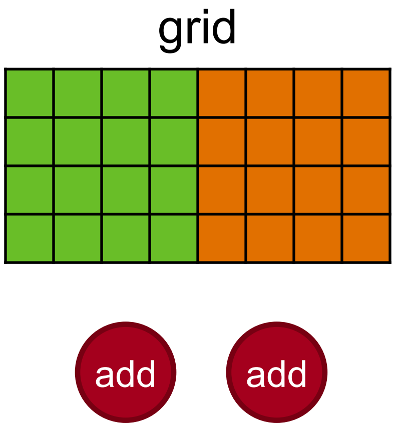
cuNumeric: Selecting Partitions Automatically

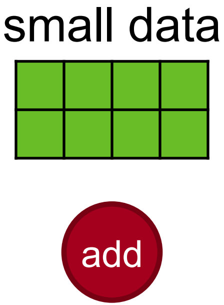
- Users don’t know anything about partitions or distributed computing
- cuNumeric chooses partitions **automatically**
  - There is no way to do this **optimally** in all cases
  - We must make some **educated guesses**
- Heuristics:
  - Minimize surface to volume ratio
  - Minimum granularity
  - Maximum parallelism



cuNumeric: Selecting Processors Automatically

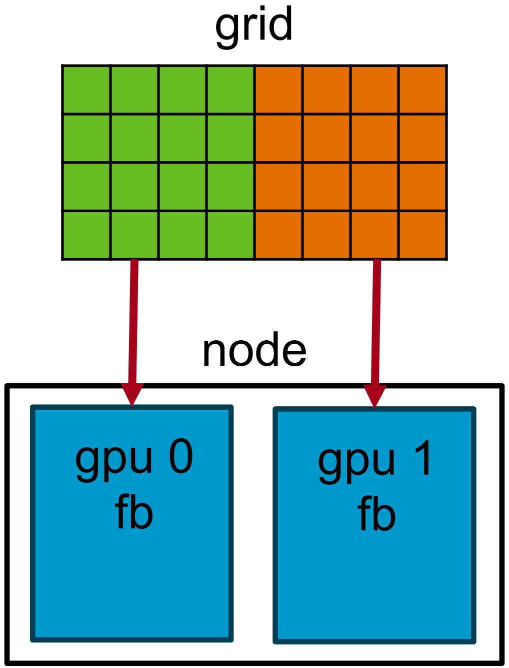
- Parallelism:
  - Given partitioning, parallelism is a function of the number of subregions
  - Prioritize **largest** region (move tasks to data)
- cuNumeric must choose **where** to run each task
  - Generally, NumPy operations are memory-bandwidth bound
  - So run each task on processor with highest bandwidth (usually GPU)
  - One task per subregion (unless data is small)





cuNumeric: Choosing Memories Automatically

- cuNumeric must also pick a memory per subregion
  - Choose memory **closest** to processor (e.g., GPU framebuffer)
  - If it's full, choose a **nearby** memory, but only if the processor can access it (e.g., another GPU's framebuffer on the same node)



cuNumeric: Task Implementation

- Every NumPy function must be implemented in cuNumeric
  - There are **hundreds** of APIs!
  - All of these need to be implemented
- cuNumeric tasks are currently hand-written
  - Three versions: C++ (CPU), OpenMP (CPU), CUDA (GPU)
  - Using existing math libraries where possible (e.g., cuBLAS)
  - While there are portability layers, they increase compile time (fast compile times are more important than minimizing code)

cuNumeric: Summary

- None of this would be possible without a task-based runtime
  - Sequential semantics are critical!
- Even so, it’s still a huge amount of work
  - Only possible because NVIDIA is investing
- Makes it dramatically easier for non-experts to access distributed GPU computing
- Open research problems remain: code generation, optimization, task scheduling, …

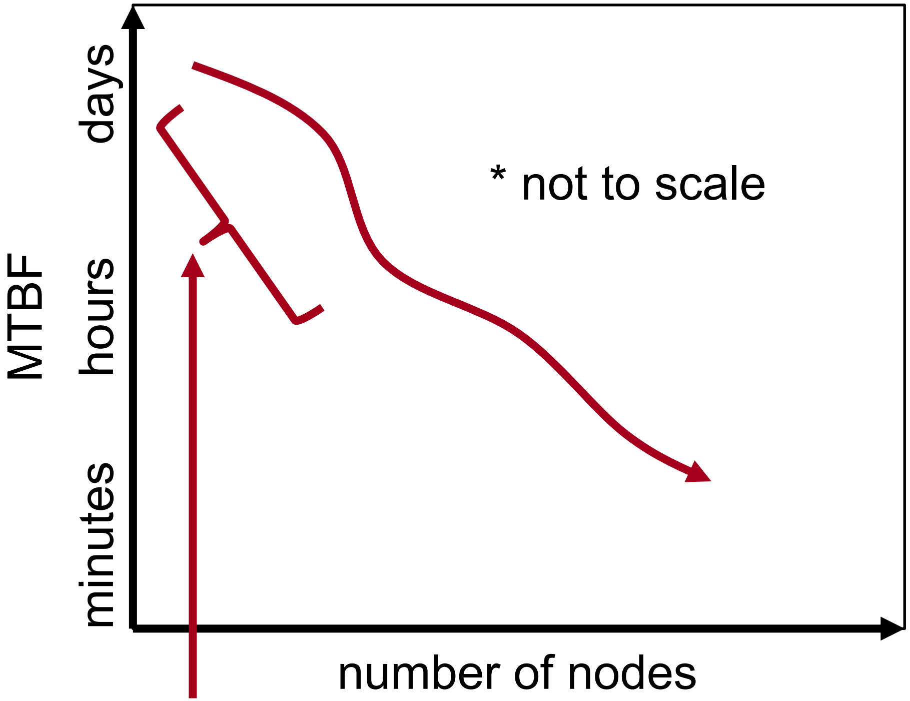
Part 3

Relight

Automatic checkpointing of task-based programs

Mean Time Between Failure (MTBF)

- Problem: as supercomputers get bigger, failures happen more frequently
- Standard solution: **checkpointing**
  - Save program state and restore it



* not to scale

current deployments of U.S. supercomputers (approximate)

Manual Checkpointing

- The de facto approach to checkpointing is manual
- You must:
    - Identify all data to be saved
    - Identify all control state to be saved (e.g., local variables)
    - Save it (requires I/O)
    - On restore, load it and put it back
    - Synchronize, or your data is invalid

```
int t = 0;

if (restore_checkpoint) {
  t = load_checkpoint(...);
  update_ghost(grid, ghost);
  communicate(ghost);
}

for (; t < T; ++t) {
  do_physics(grid, ghost);
  update_ghost(grid, ghost);
  communicate(ghost);
  MPI_Barrier(...);
  save_checkpoint(t, grid);
  MPI_Barrier(...);
}
```

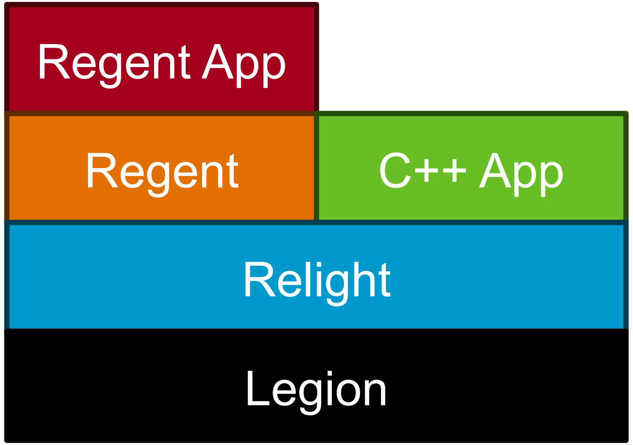
Automatic Checkpointing

- Need to save:
  - Local variables
  - Heap data structures
  - Contents of network buffers
  - Synchronize, or you lose data
- Examples:
  - DMTCP, BLCR
  - Stop process, save entire address space (and network buffers)
  - Works but very expensive

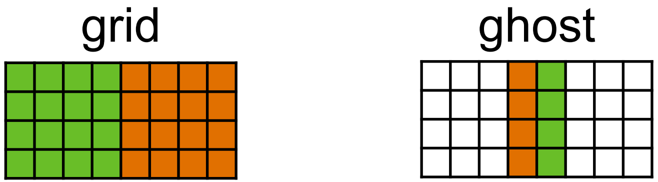
- Alternative:
  - Task-based systems allow us to capture what matters (regions)
  - Sequential semantics saves us from synchronization
  - Need: a way to restore main task

Relight: Design

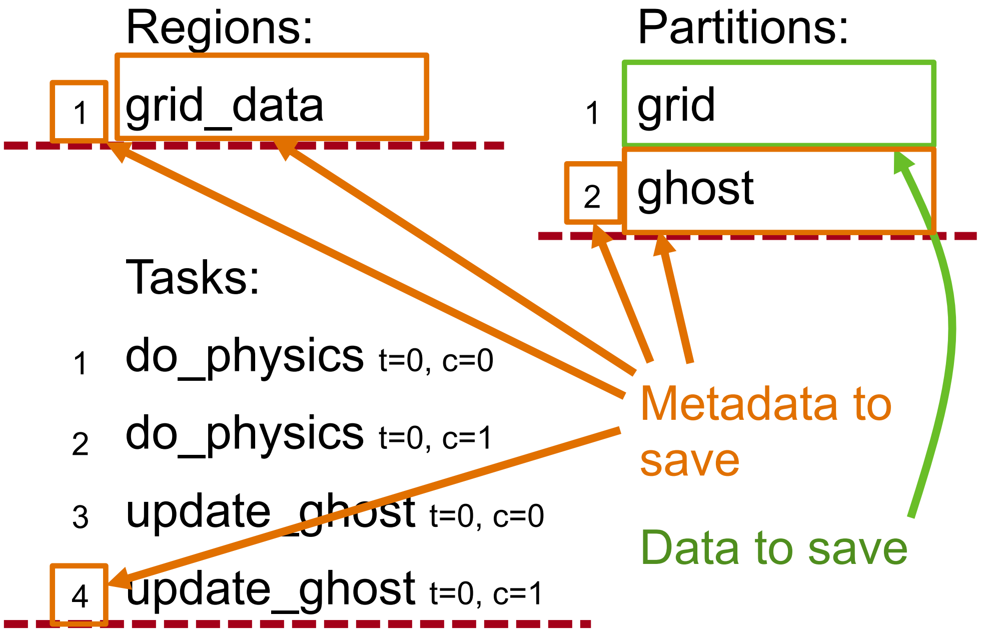
- Sits between application and runtime
  - Intercept runtime calls
  - Thus we know all regions, partitions, tasks, etc.
- On checkpoint:
  - List all regions
  - Save them to disk
  - Save metadata



Relight: Checkpoint



```
var grid_data = region(...)
var grid = partition(...)
var ghost = partition(...)
for t = 0, T do
  for c = 0, 2 do
    do_physics(grid[c], ghost[c])
  end
  for c = 0, 2 do
    update_ghost(grid[c])
  end
  __checkpoint()
end
```



Regions:

1 grid_data

Partitions:

1 grid

2 ghost

Tasks:

1 do_physics t=0, c=0

2 do_physics t=0, c=1

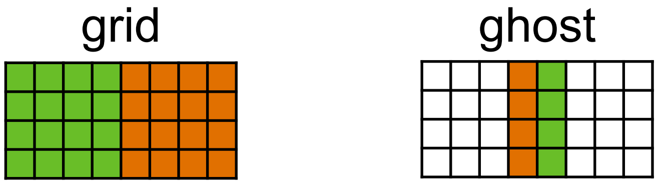
3 update_ghost t=0, c=0

4 update_ghost t=0, c=1

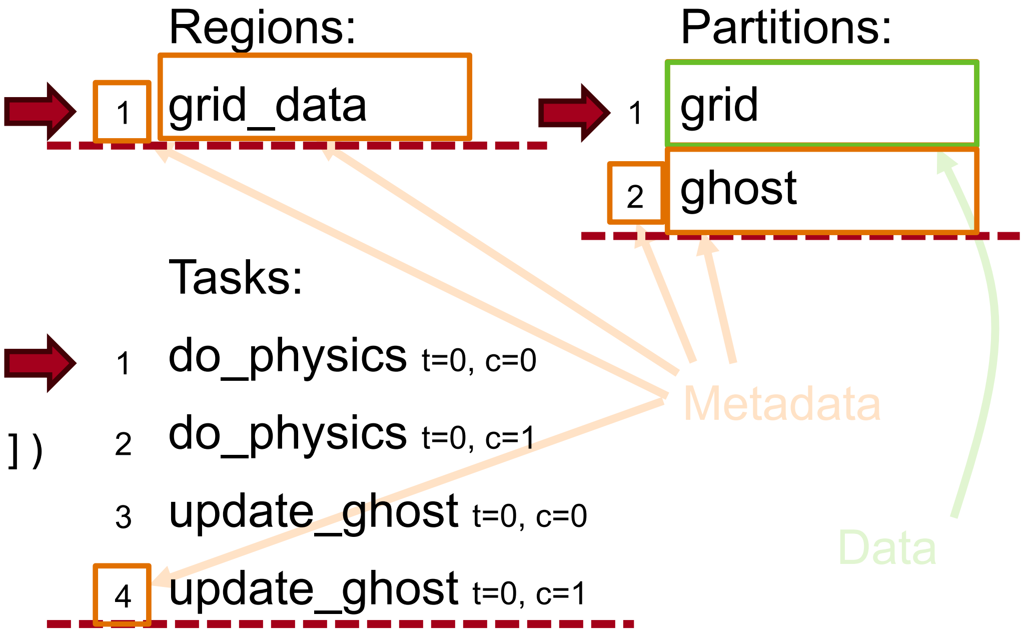
Metadata to save

Data to save

Relight: Restore



```
var grid_data = region(...)
var grid = partition(...)
var ghost = partition(...)
for t = 0, T do
  for c = 0, 2 do
    do_physics(grid[c], ghost[c])
  end
  for c = 0, 2 do
    update_ghost(grid[c])
  end
  __checkpoint()
end
```

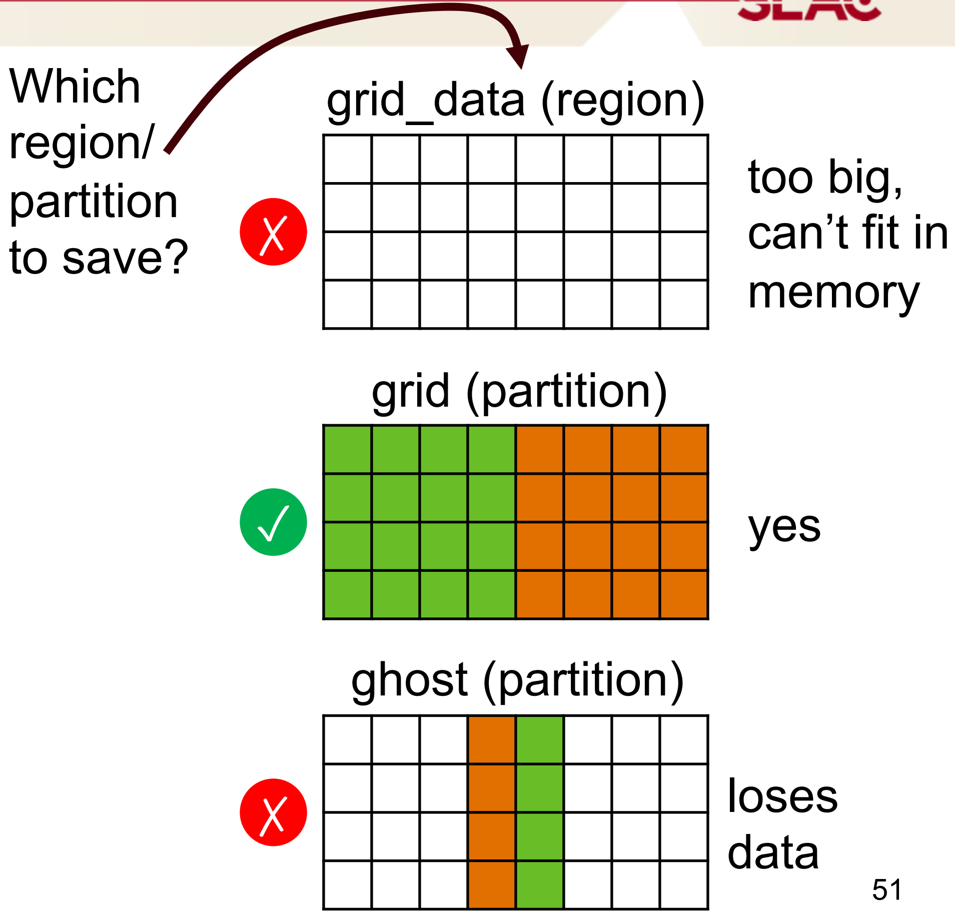


Relight: What Data to Save?

- Metadata:
  - Counters (for regions, tasks, partitions, …)
    - Sequential semantics means we uniquely identify data this way
  - Region bounds (i.e., how big it is, not contents)
  - Partition bounds (number of subregions and bounds)
- Data:
  - Contents of regions
  - Task results

Relight: Which Partitions to Save?

- We always save regions via partitions (if possible):
  - To get parallel I/O
  - To avoid overflowing one node's memory
- Partitions must be:
  - Disjoint (or we can't write in parallel)
  - Complete (or we lose data)



Which region/ partition to save?

grid_data (region)

✗ too big, can't fit in memory

grid (partition)

✓ yes

ghost (partition)

✗ loses data

Relight: Summary

- Regent/Legion abstractions enable better checkpointing:
  - Sequential semantics
    - So we can reason about program state
  - Tasks
    - So we can fast-forward execution
- This is not possible in MPI!
  - The programming model actually matters

Resources

- Legion: https://legion.stanford.edu
- Regent: https://regent-lang.org
- cuNumeric: https://developer.nvidia.com/cunumeric
- Relight: https://github.com/StanfordLegion/resilience

Questions