Parallel Programming Models

Elliott Slaughter

CME 213 lecture 2021-06-02

U.S. DEPARTMENT OF ENERGY | Stanford University

SLAC NATIONAL ACCELERATOR LABORATORY

About Me

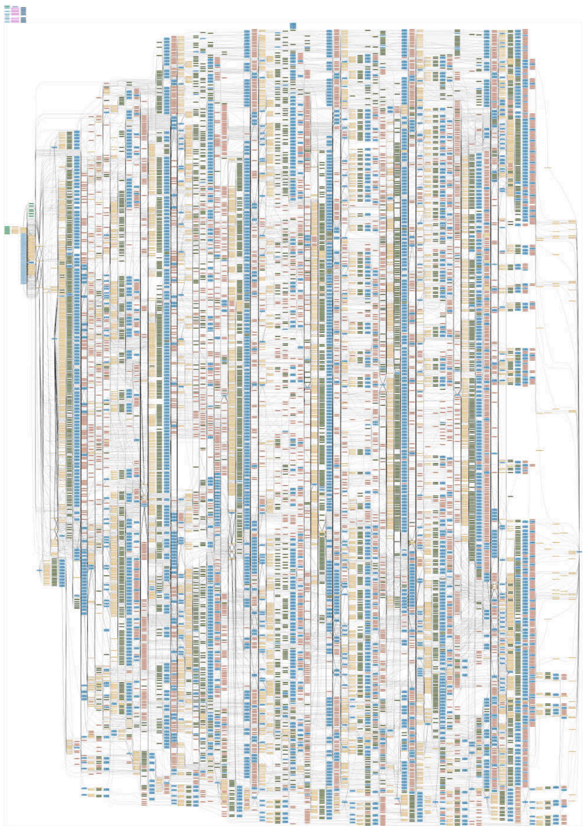
- Stanford CS PhD, 2017 (with Alex Aiken)
- SLAC CS research group since 2017

Today's HPC Landscape

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

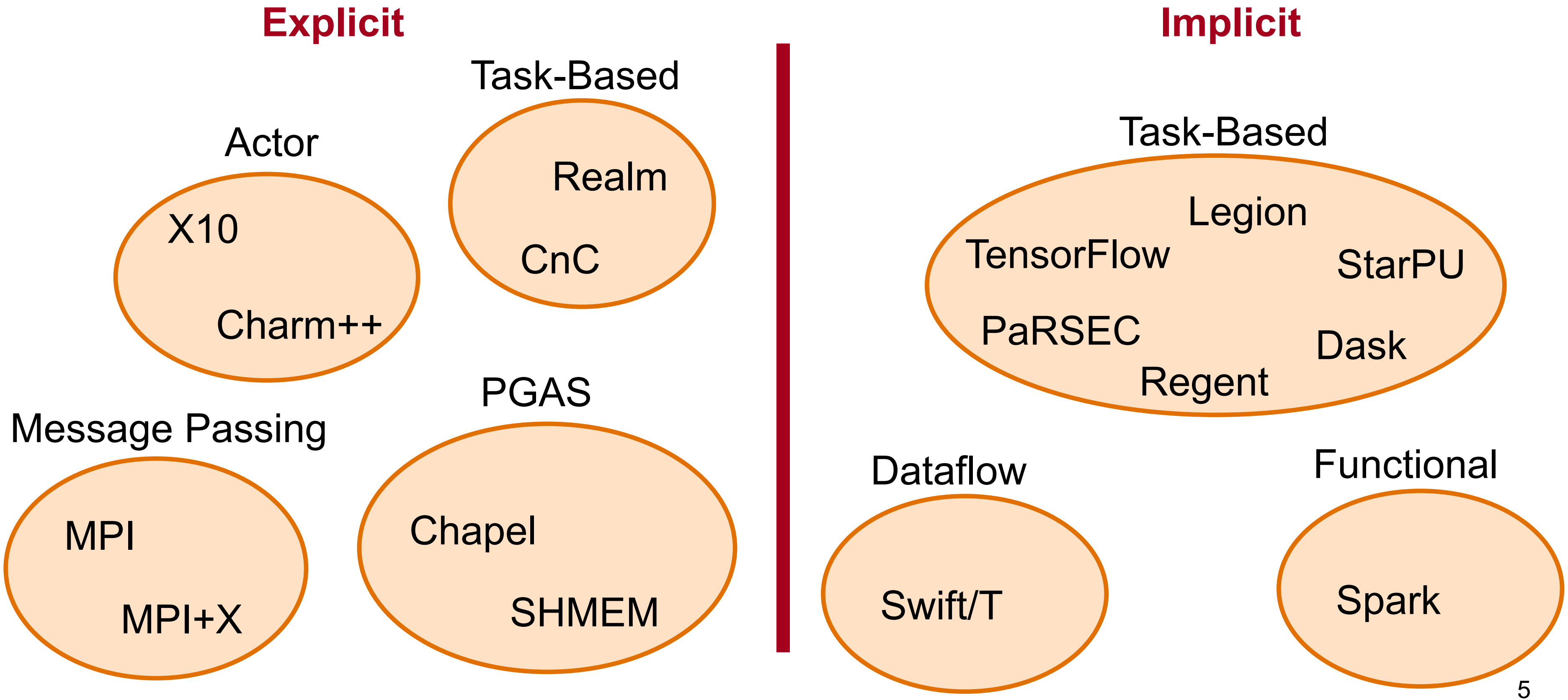
The Good (and Bad) News About Parallelism

- 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



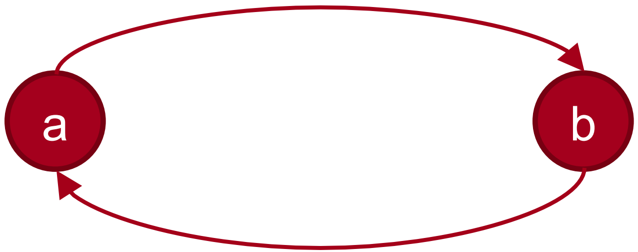
This Lecture

- Focus on two categories:
  - Actors (Charm++)
  - (Implicit) Task-Based (Legion/Regent, StarPU, PaRSEC)

Actors: The Big Idea (1/3)

- Big idea: communicating objects

Objects can call methods on other objects



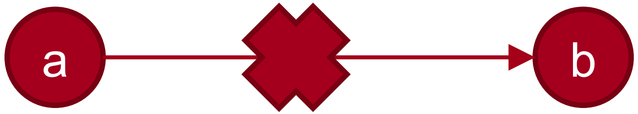

```
class A:
  def some_method():
    b.method()
    x = b.method()
```

Methods can return results

Actors: The Big Idea (2/3)

- Big idea: no shared state

No direct access to the state of other objects

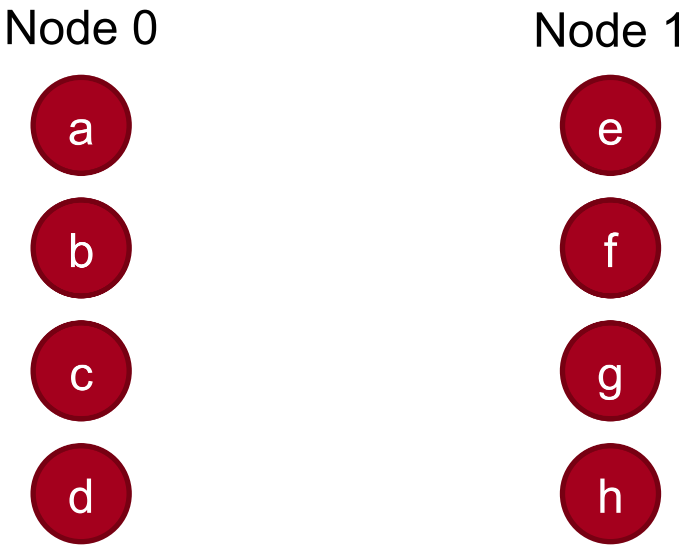



```
class A:
  def some_method():
    b.x = ... # error
```

Actors: The Big Idea (3/3)

- Big idea: seamless migration

Objects can migrate to other nodes

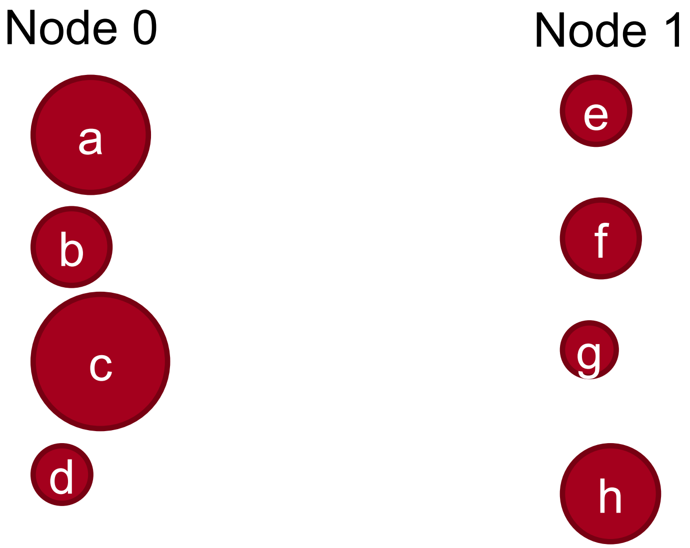


Objects are balanced,
no need to migrate

Actors: The Big Idea (3/3)

- Big idea: seamless migration

Objects can migrate to other nodes



Objects are imbalanced, can migrate to balance load

Implementations of Actors

Common implementations in HPC:

- Charm++ (covered this lecture)
- X10

Elsewhere:

- Erlang (telephony, fault-tolerant distributed systems)
- Stackless Python (“microthreads”)
- Go (“goroutines”)
- … (and many more)

Charm++ Basics

- **Chare**: an object/actor
- **Chare Array**: array of chares
  - Used for distribution, collectives

```
class Hello(Chare):
  def hi(self):
    print("hello")

def main(args):
  a = Array(Hello, 2)
  a[0].hi()

charm.start(main)
```

Note: code uses Charm4py interface for Python

Charm++: Returning Values

```python
class Fib(Chare):
  def fib(self):
    n = self.thisIndex
    if n <= 1:
      return n
    a = self.thisProxy[n-1].fib(ret=True)
    b = self.thisProxy[n-2].fib(ret=True)
    return a.get() + b.get()

def main(args):
  a = Array(Fib, 10)
  print(a[9].fib(ret=True).get())

charm.start(main)
```

This is the index of the chare in its array

Method calls **do not block**

By default, methods throw away any return value. Request the result with ret=True

get() blocks on the remote method call

Charm++: Collectives

```python
class Sum(Chare):
  def __init__(self):
    self.data = ...

  def work(self):
    self.reduce(
      self.thisProxy[0].do_something,
      self.data,
      Reducer.sum)

  def do_something(self, result):
    print("the result is",result)

def main(args):
  a = Array(Sum, 10)
  a.work()
```

Collectives are performed in the context of an array

Callback to be executed when collective is complete

Predefined reduction sum operator

A Simple Timestep Loop in Charm++?

```
for t in range(0, T):
  ghosts = []
  for n in neighbors:
    ghosts.append(
      n.get_ghost(self.thisIndex, ret=True))
  values = [g.get() for g in ghosts]
  self.do_physics(values)
```

This code has a bug!

The physics step doesn't wait for sending ghosts to complete before continuing

Why does this happen? Because messages are one-sided (not like MPI!)

One last feature: Channels

```
class Send(Chare):
  def sender(self):
    ch = Channel(self, remote=self.thisProxy[1])
    ch.send("hi")

  def receiver(self):
    ch = Channel(self, remote=self.thisProxy[0])
    print(ch.recv()) # blocks on result

def main(args):
  a = Array(Send, 2)
  a[0].sender()
  a[1].receiver()
```

Channels are two-sided, send and recv (like MPI)

recv blocks (like MPI)

A Simple Timestep Loop in Charm++ (with Channels)

```
for t in range(0, T):
  values = []
  for n in neighbors:
    n.out_ch.send(self.local_data)
  for n in neighbors:
    values.append(n.in_ch.recv())
  self.do_physics(values)
```

Lesson learned: one-sided messages are tricky to use in practice

Summary: Actors

Pros:

- More flexibility than traditional models like MPI
  - Can create chare arrays at runtime, for example
- Automatic migration to adapt to load imbalance

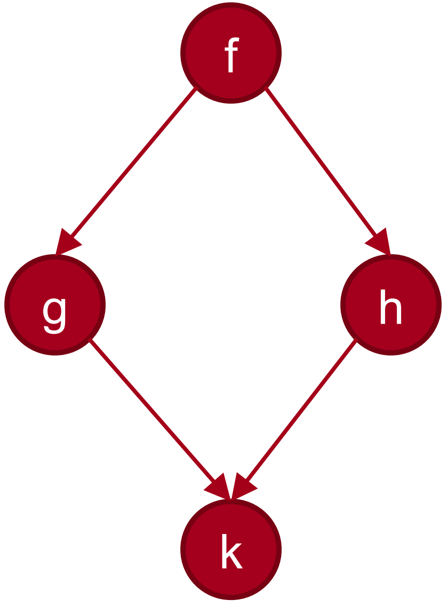
Cons:

- You can still get synchronization wrong!
  - Particularly with one-sided messages
- You still have to manually decompose your application into chares

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)
```

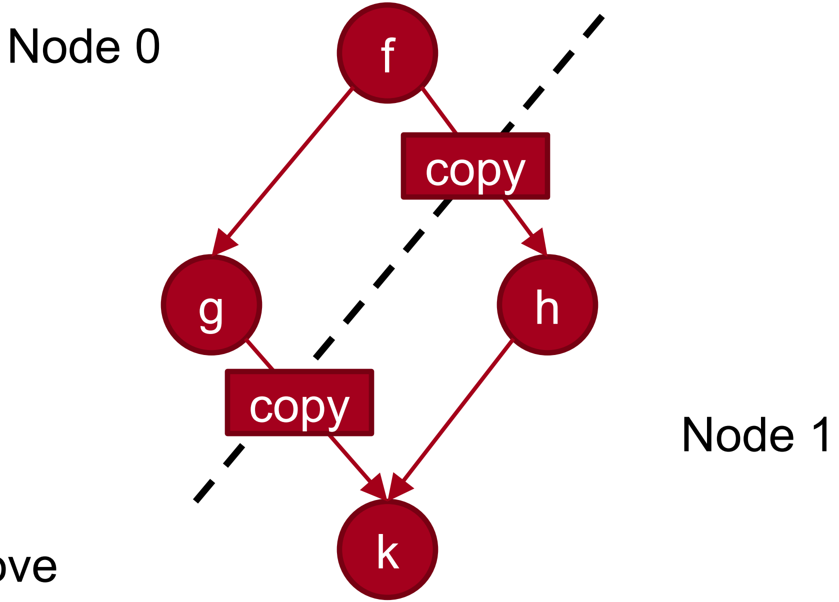


Sequential semantics means no way to get the synchronization wrong!

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)
```

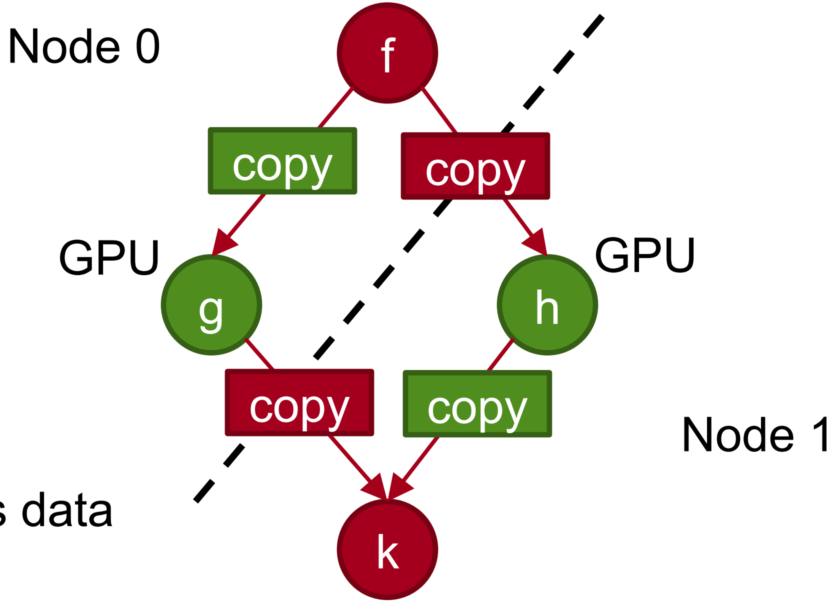


The system determines when messages need to be sent to move data between nodes

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
- Spark

Regent Basics

- This lecture will use Regent syntax
- But concepts apply to Legion, StarPU, PaRSEC

```
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

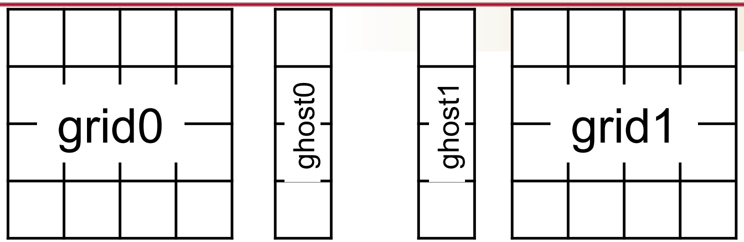
r	r	r	r	g	g	g	g	b	b	b	b
r	r	r	r	g	g	g	g	b	b	b	b
r	r	r	r	g	g	g	g	b	b	b	b
r	r	r	r	g	g	g	g	b	b	b	b

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?



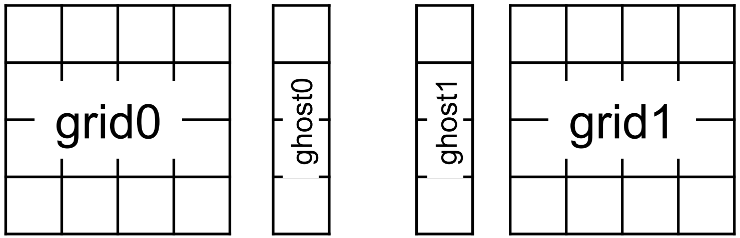

```
for t = 0, T do
  do_physics(grid0, ghost1)
  do_physics(grid1, ghost0)

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

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

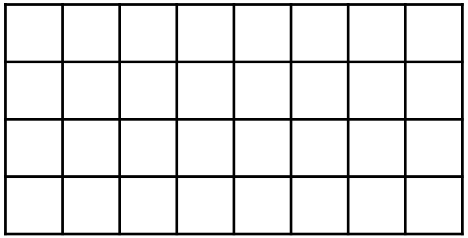
```
task do_physics(
    grid : region(...),
    ghost : region(...))
where reads writes(grid),
      reads(ghost)
do ... end

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

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

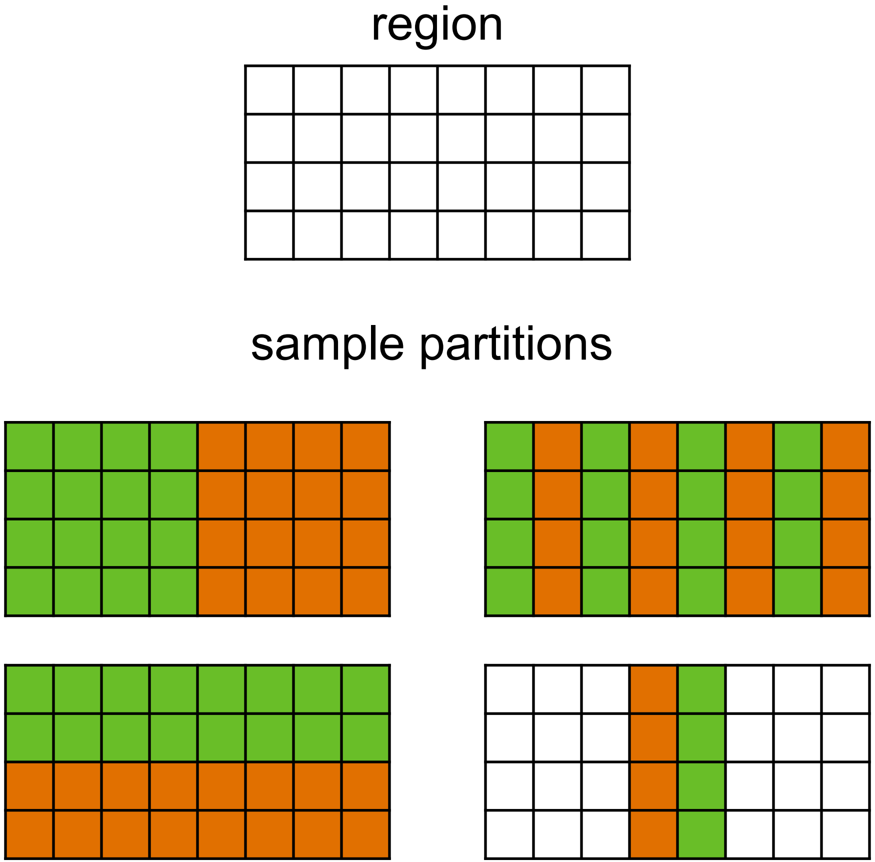


grid (the whole thing)

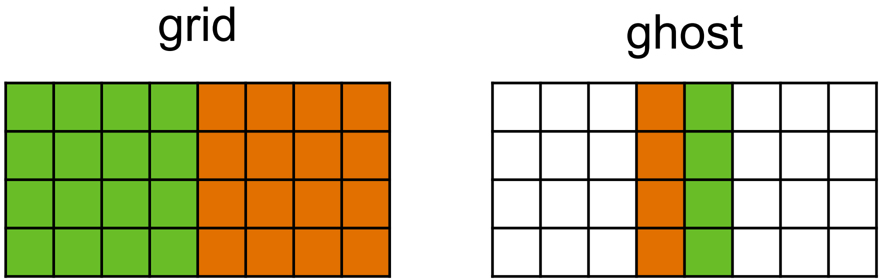


Regent: Partitioning

- 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



A Simple Timestep Loop in Regent (with Partitioning)



These partition the same region

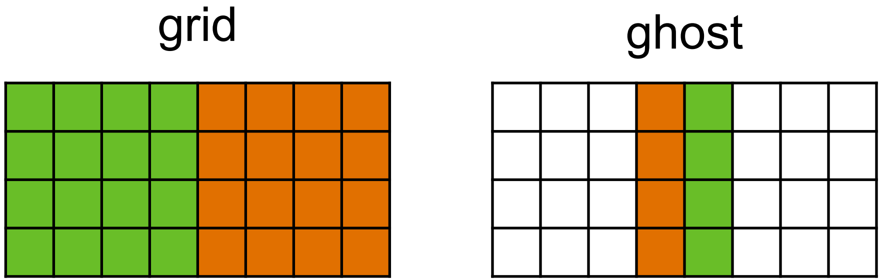
```
for t = 0, T do
  for c = 0, 2 do
    do_physics(grid[c], ghost[c])
  end

  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
```

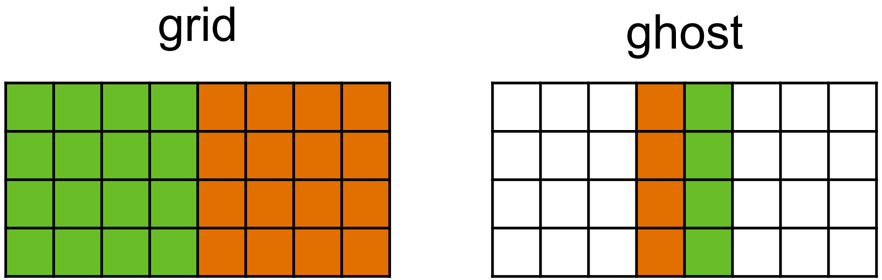
Privileges are updated to include fields

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

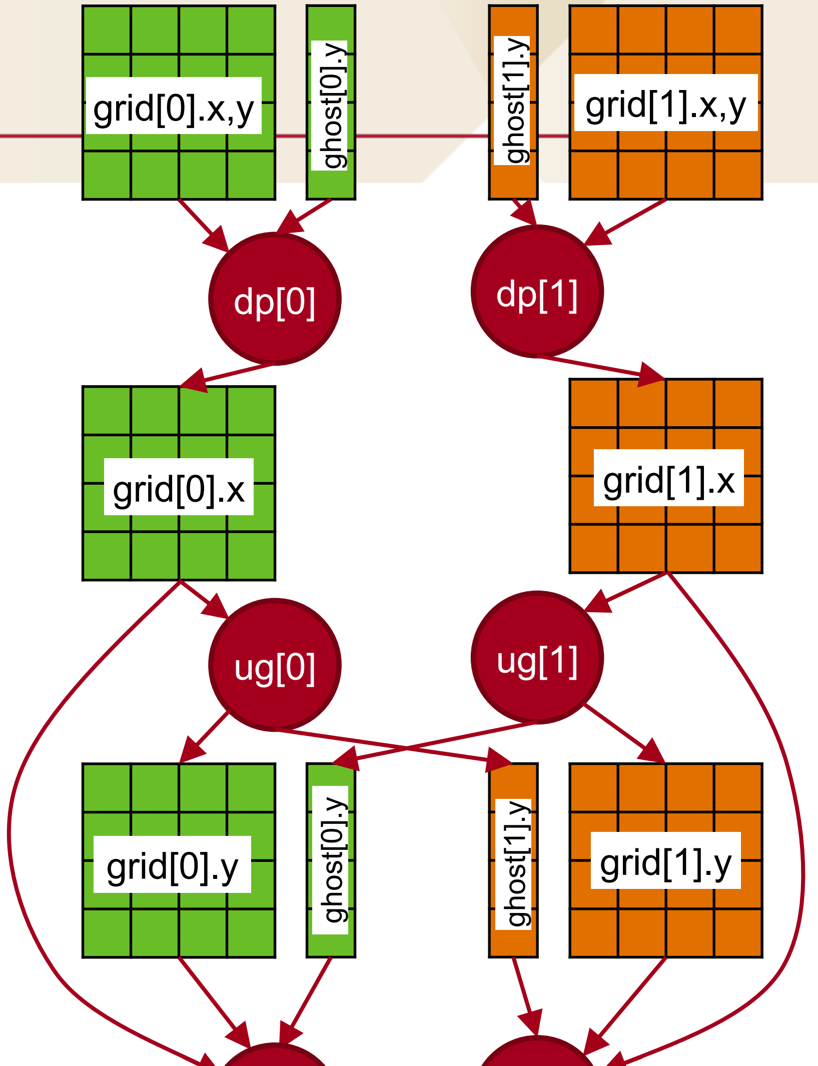
Important: use different fields, otherwise tasks cannot run in parallel!

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

Timestep Loop: Execution



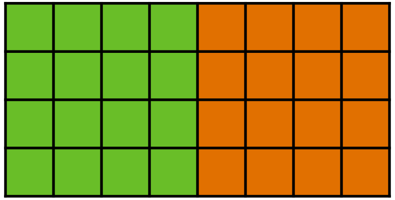

```
for t = 0, T do
  for c = 0, 2 do
    do_physics(grid[c],      ghost[c])
    --          W(x) R(y),    R(y)
  end

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



More on Partitioning

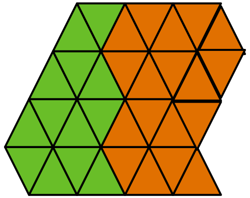
Equal partitioning

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

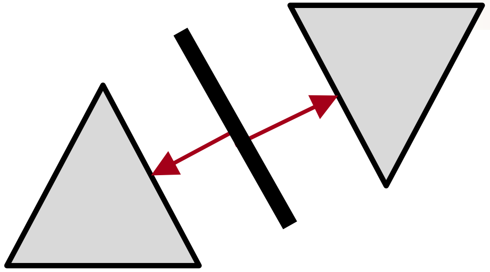


Partition by field (e.g., METIS)

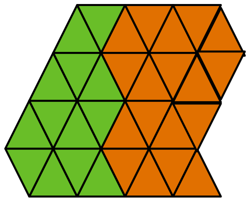
```
run_metis(r) -- W(color)
partition(r.color,
    ispace(int1d, 2))
```



Dependent Partitioning



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



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

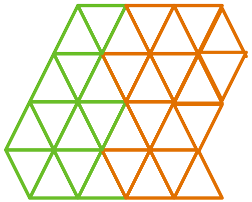
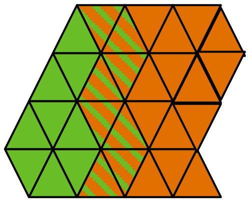
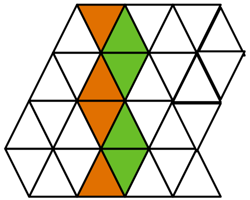


Image (partition of cells)
u = image(cell, t, edge.cell)



Subtract (partition of cells)
v = u - s



Regent Optimization: Index Launches

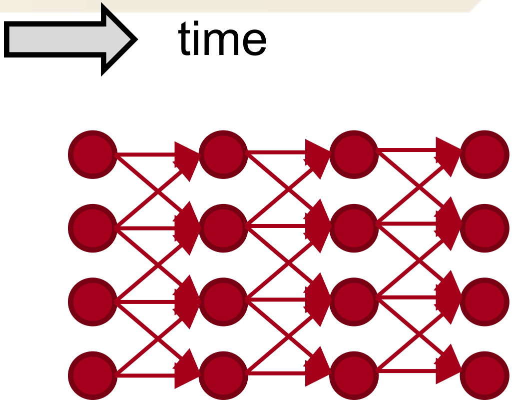
```
for t = 0, T do
  for c = 0, 4 do -- index launch
    do_physics(grid[c], ghost[c])
  end

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

These loops are index launches

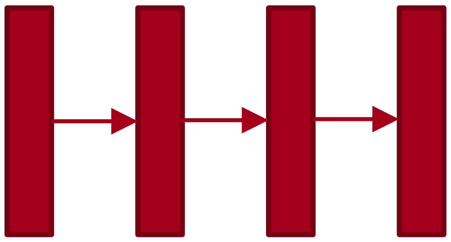
This is an automatic optimization, no input required by the user

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



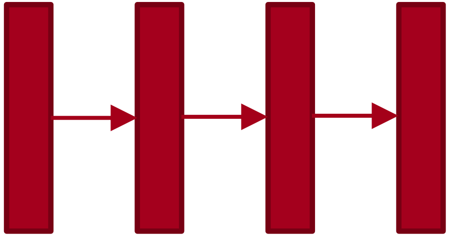
With optimization

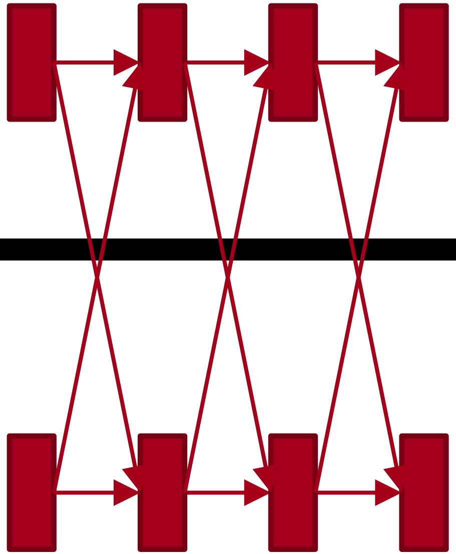
Index launches reduce overhead in the runtime

Regent Optimization: Control Replication

(Without optimization)

Node 0





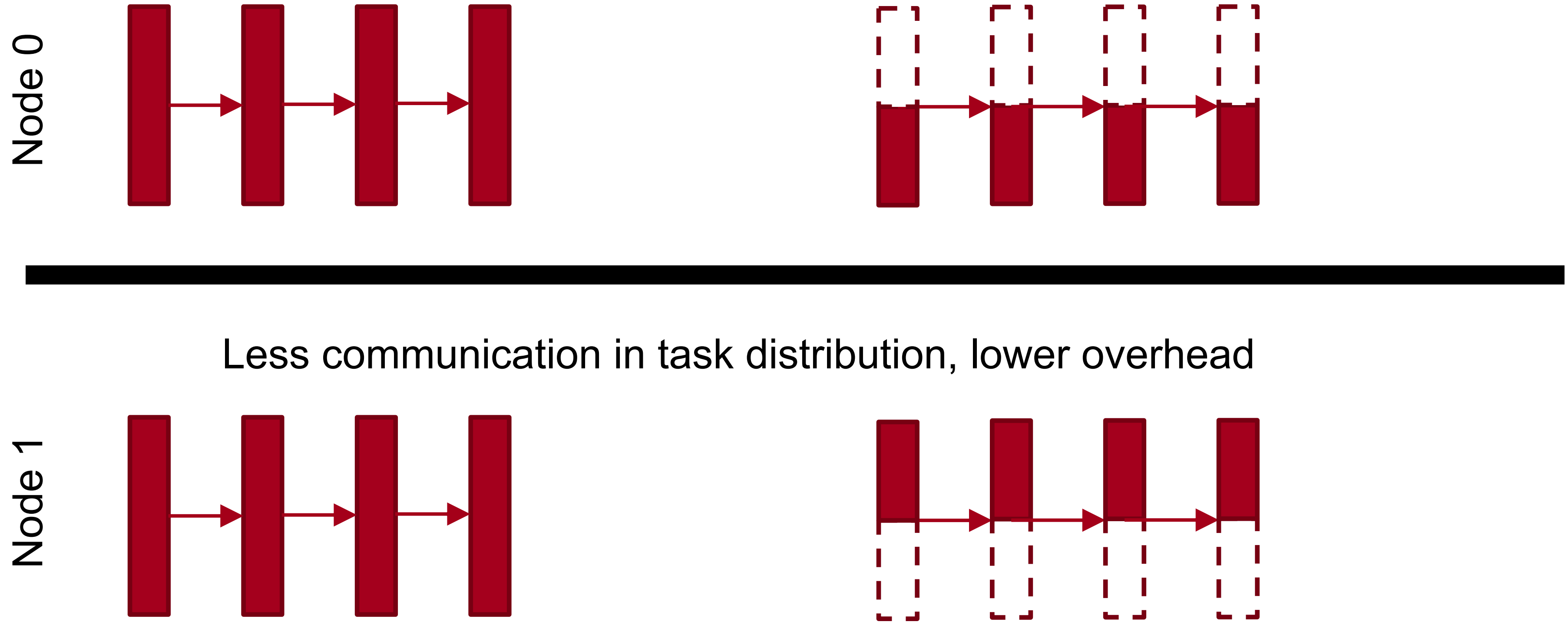
Node 1

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: Task-Based Systems

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

Resources

Charm++:

- http://charm.cs.illinois.edu/research/charm
- Charm4py (Python interface) https://charm4py.readthedocs.io/en/latest/

Legion/Regent:

- https://legion.stanford.edu/
- http://regent-lang.org/

StarPU:

- https://starpu.gitlabpages.inria.fr/

PaRSEC:

- http://icl.utk.edu/parsec/

Questions