Pacific
Northwest
NATIONAL LABORATORY

Chapel Aggregation Library (CAL)

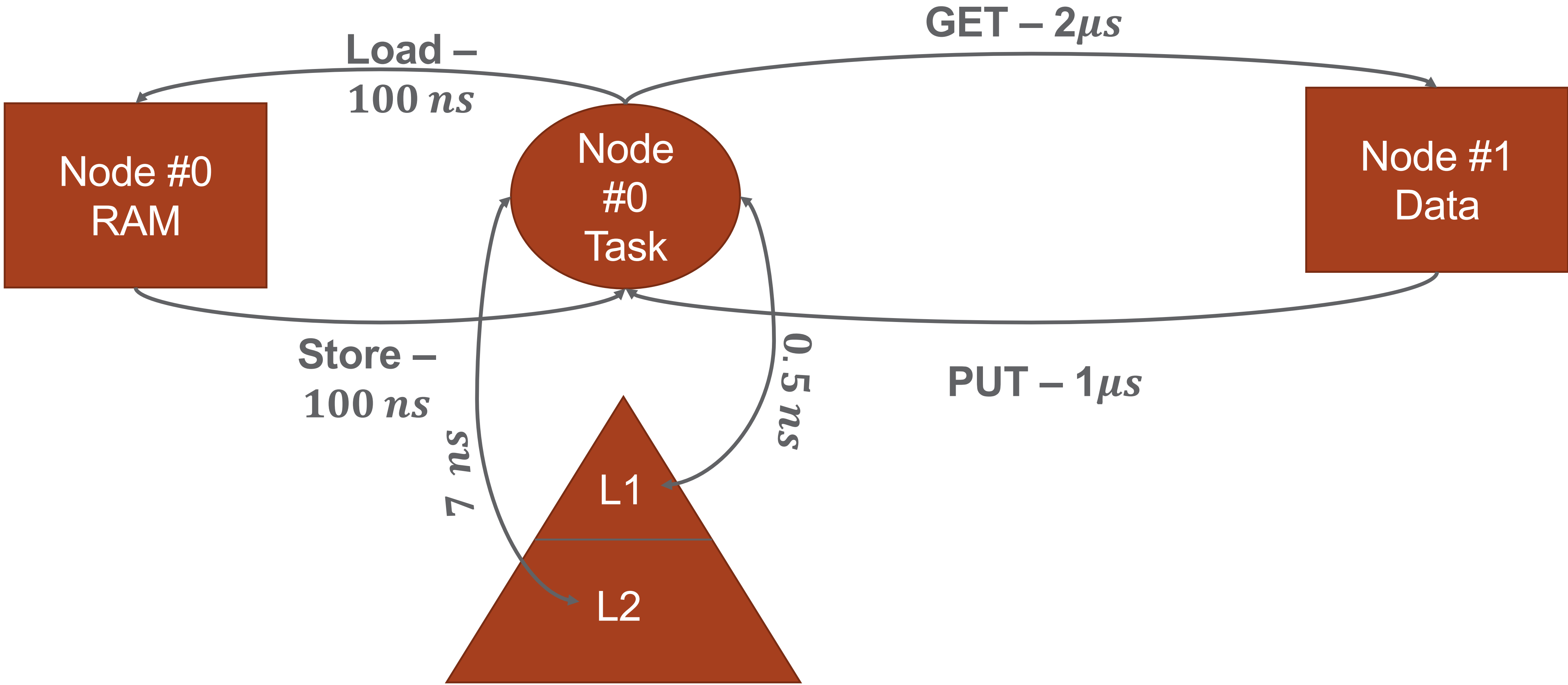
November 12, 2018

Louis Jenkins
Marcin Zalewski (Pacific Northwest National Lab.),
Michael Ferguson (Cray Inc.)

U.S. DEPARTMENT OF ENERGY BATTELLE
PNNL is operated by Battelle for the U.S. Department of Energy

The Problem

- Accessing remote data is slow
    - Multiple orders of magnitude slower to access than local memory



Load –
$100\ ns$

GET – $2\mu s$

Node #0
RAM

Node
#0
Task

Node #1
Data

Store –
$100\ ns$

$0.5\ ns$

PUT – $1\mu s$

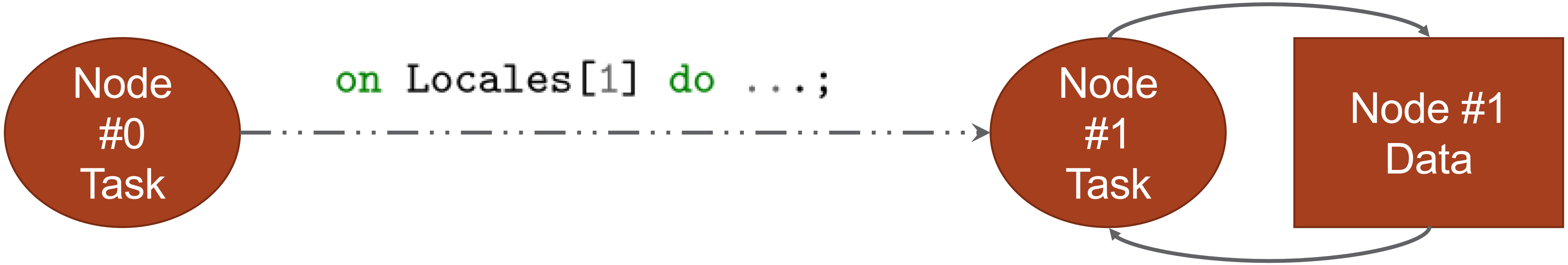
$7\ ns$

L1

L2

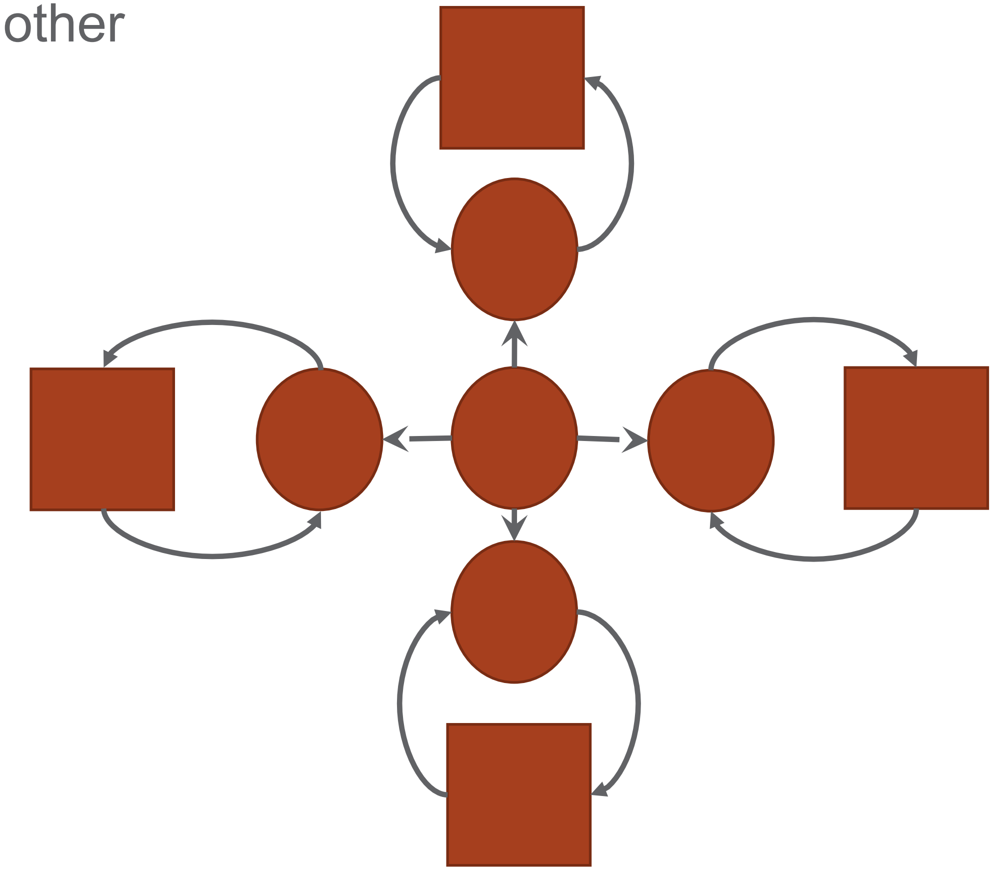
The Problem

- Accessing remote data is slow
  - Multiple orders of magnitude slower to access than local memory
- "Moving the computation to the data" not always the best solution
  - Using an *on* statement requires migrating tasks to another locale



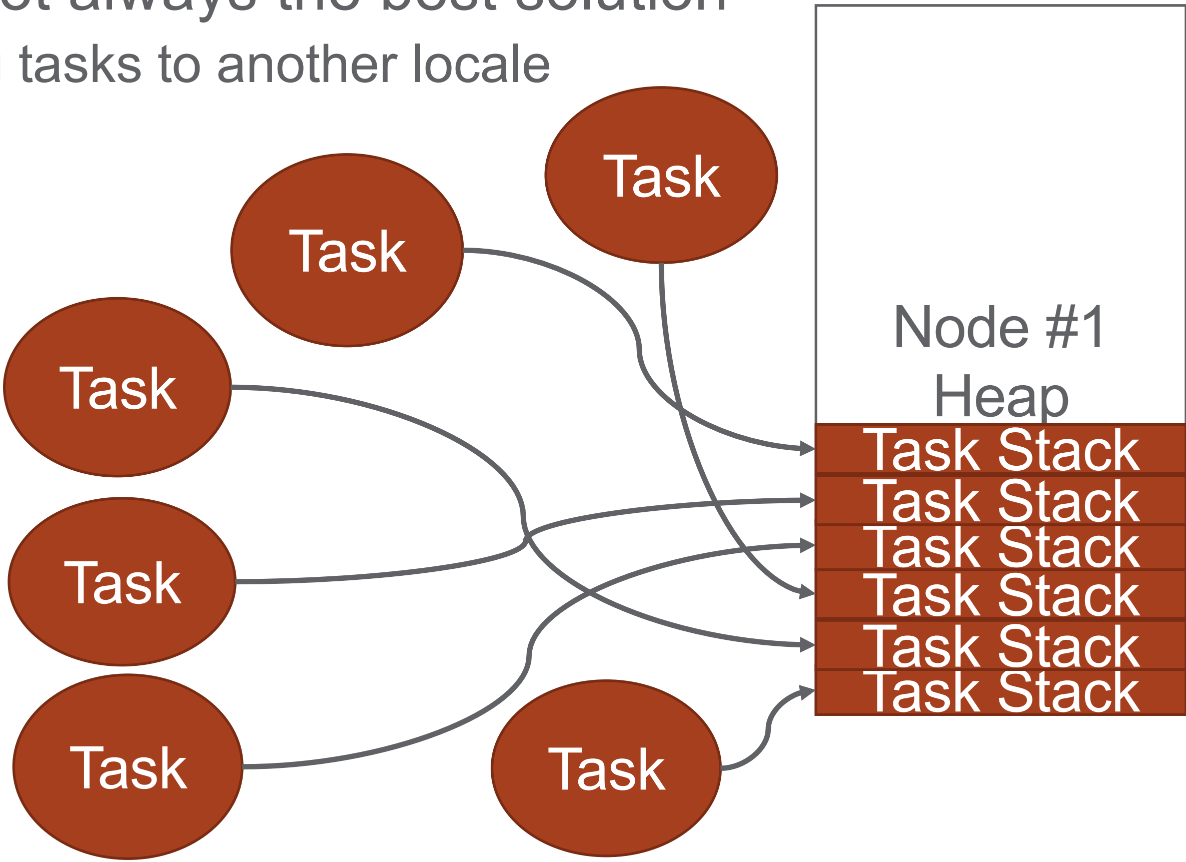
The Problem

- Accessing remote data is slow
    - Multiple orders of magnitude slower to access than local memory
- “Moving the computation to the data” not always the best solution
    - Using an *on* statement requires migrating tasks to another locale
        - ✓ Can become bottleneck if fine-grained



The Problem

- Accessing remote data is slow
  - Multiple orders of magnitude slower to access than local memory
- “Moving the computation to the data” not always the best solution
  - Using an *on* statement requires migrating tasks to another locale
    - ✓ Can become bottleneck if fine-grained
    - ✓ Task creation is relatively expensive
      - Tasks are too large to spawn in a fire-and-forget manner (issue #9984)
      - Migrating tasks require individual active messages (issue #9727)



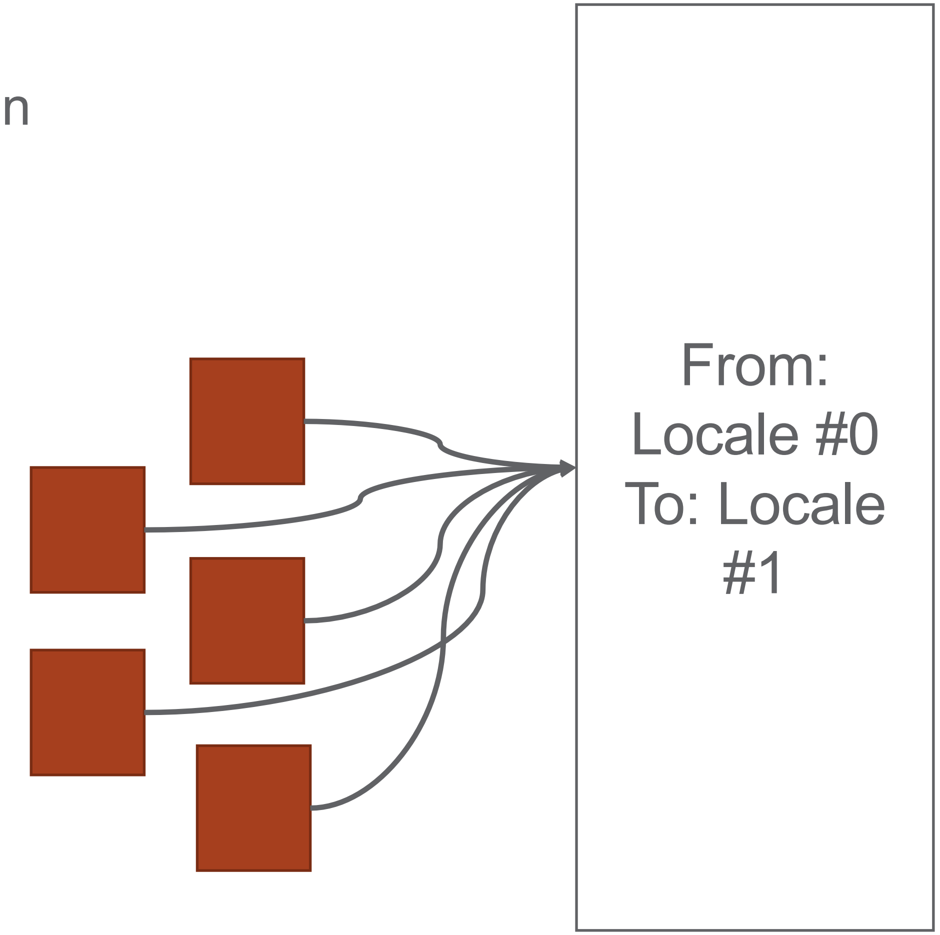
A Solution

- Coarsen the granularity of the data
    - Buffer units of data to be sent to a locale in *destination buffers*

From: Locale #0 To: Locale #1

A Solution

- Coarsen the granularity of the data
  - Buffer units of data to be sent to a locale in *destination buffers*

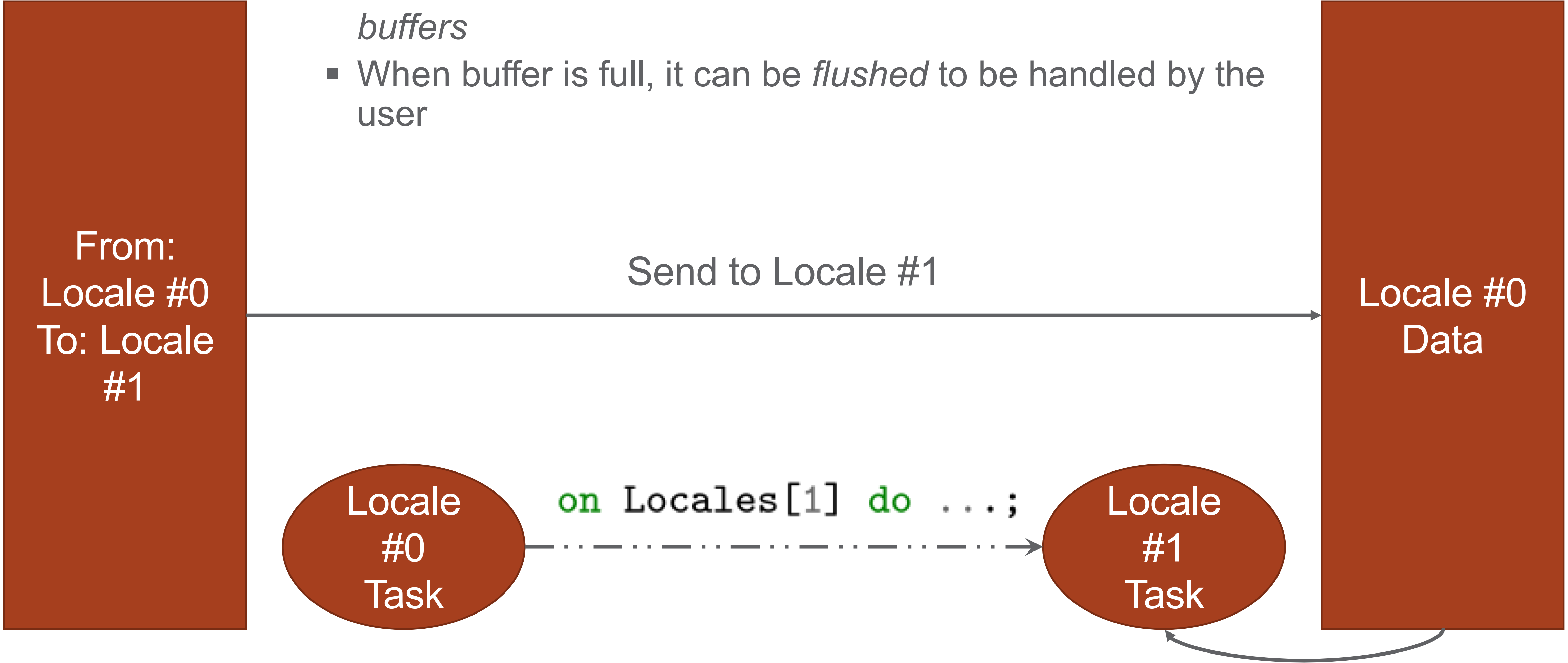


A Solution

- Coarsen the granularity of the data
  - Buffer units of data to be sent to a locale in *destination buffers*

A Solution

- Coarsen the granularity of the data
  - Buffer units of data to be sent to a locale in *destination buffers*
  - When buffer is full, it can be *flushed* to be handled by the user



From: Locale #0 To: Locale #1

Send to Locale #1

Locale #0 Data

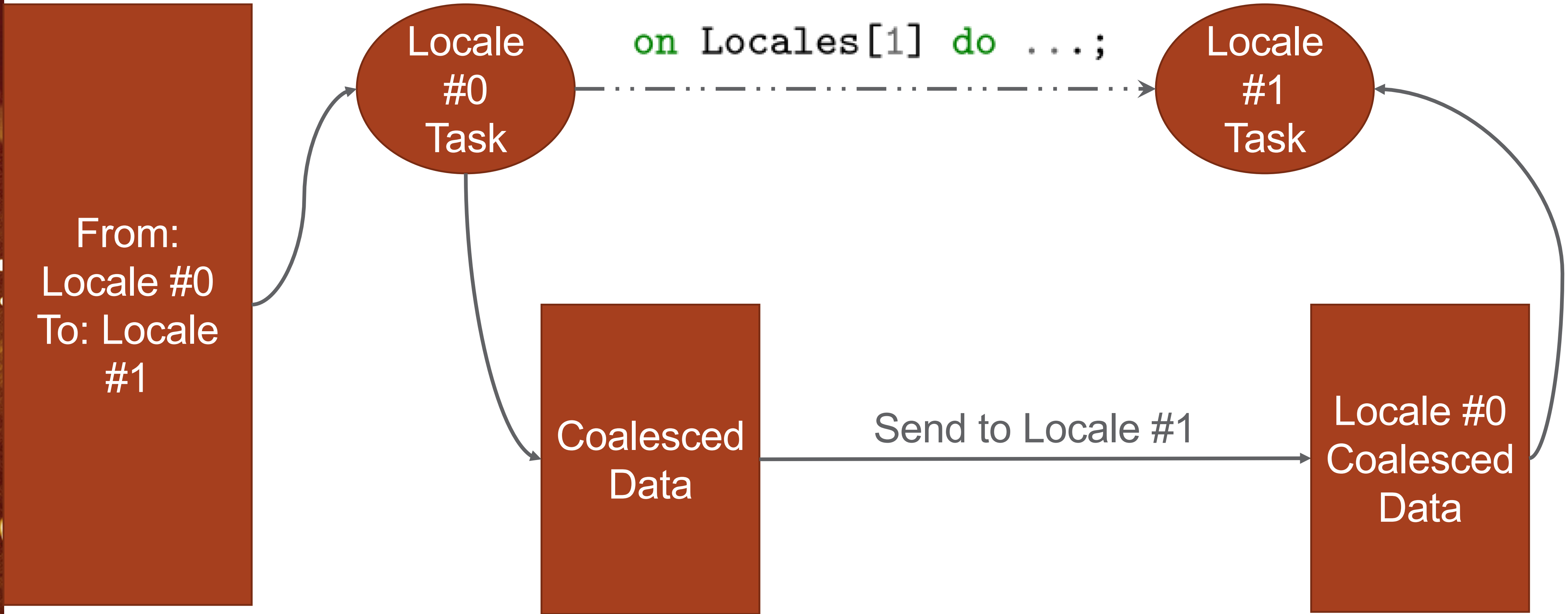
Locale #0 Task

`on Locales[1] do ...;`

Locale #1 Task

A Solution

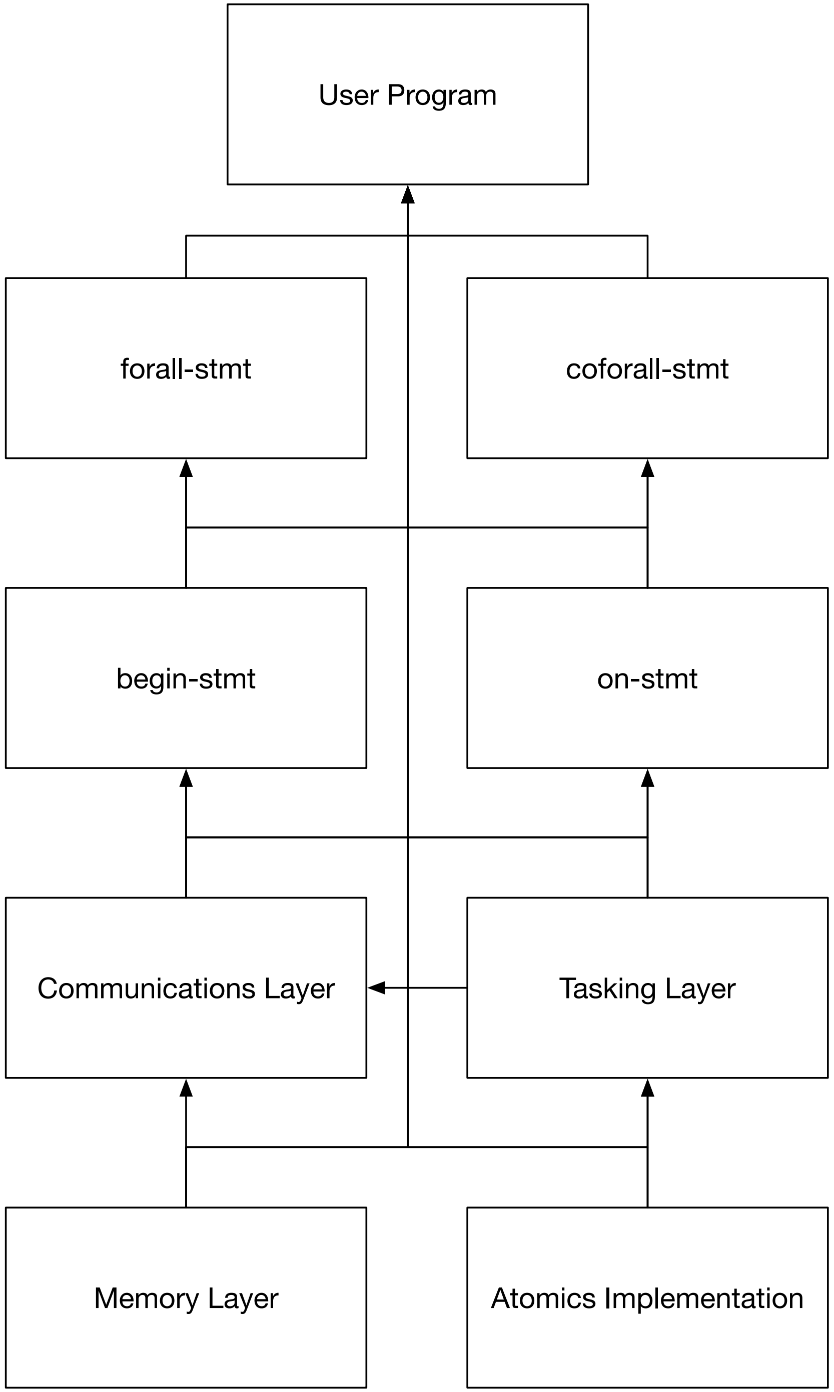
- Coarsen the granularity of the data
  - Buffer units of data to be sent to a locale in *destination buffers*
  - When buffer is full, it can be *flushed* to be handled by the user
  - User can perform *coalescing* to combine aggregated data



Pacific Northwest
NATIONAL LABORATORY

Chapel’s Multiresolution Design Philosophy

- Higher Level composed of Lower Level abstractions, features, and language constructs
    - Changes to lower level propagate up to higher level
    - User free to use either
        - ✓ High-Level for convenience
        - ✓ Low-Level for performance



Global-View Programming

- Abstracts locality for the user
  - No need to think: “What portion of the array does this task own?”
  - Array can be accessed from any locale, even if it is not distributed over that locale…
    - ✓ Remote references are resolved into remote PUT/GET *implicitly*

Chapel

```
var sum : float;
forall a in arr with (+ reduce sum) {
  sum += a;
}
```

MPI

```
float globalSum = 0;
float localSum = 0;
for (int i = localStart; i < localEnd; i++) {
  localSum += arr[i];
}
MPI_REDUCE(&localSum, &globalSum, ...);
```

Global-View Programming

- Abstracts locality for the user
  - No need to think: “What portion of the array does this task own?”
  - Array can be accessed from any locale, even if it is not distributed over that locale…
    - ✓ Remote references are resolved into remote PUT/GET *implicitly*
- Multiresolution: More Abstraction

Chapel

```
var sum = + reduce arr;
```

MPI

```
float globalSum = 0;
float localSum = 0;
for (int i = localStart; i < localEnd; i++) {
  localSum += arr[i];
}
MPI_REDUCE(&localSum, &globalSum, ...);
```

Global-View Programming

- Abstracts locality for the user
  - No need to think: “What portion of the array does this task own?”
  - Array can be accessed from any locale, even if it is not distributed over that locale…
    - ✓ Remote references are resolved into remote PUT/GET *implicitly*
- Multiresolution: Less Abstraction

Chapel

```
var sum : float;
coforall loc in Locales with (+ reduce sum) do on loc {
  coforall tid in 0..#here.maxTaskPar with (+ reduce sum) {
    for i in computeRange(arr.domain.localSubdomain(), tid) {
      sum += arr[i];
    }
  }
}
```

MPI

```
float globalSum = 0;
float localSum = 0;
for (int i = localStart; i < localEnd; i++) {
  localSum += arr[i];
}
MPI_REDUCE(&localSum, &globalSum, ...);
```

Chapel Aggregation Library (CAL)

- Written in Chapel, for Chapel
  - **Minimal** and User-Friendly
    - ✓ Unassuming of how data is handled
    - ✓ Designed specifically for Chapel
  - **Distributed**, Scalable, and Parallel-Safe
    - ✓ Supports Global-View Programming
    - ✓ Usable with Chapel’s parallel and locality constructs
  - **Modular**, Reusable, and Generic
    - ✓ Generic on user-defined type
    - ✓ Easy to use and ’plug in’

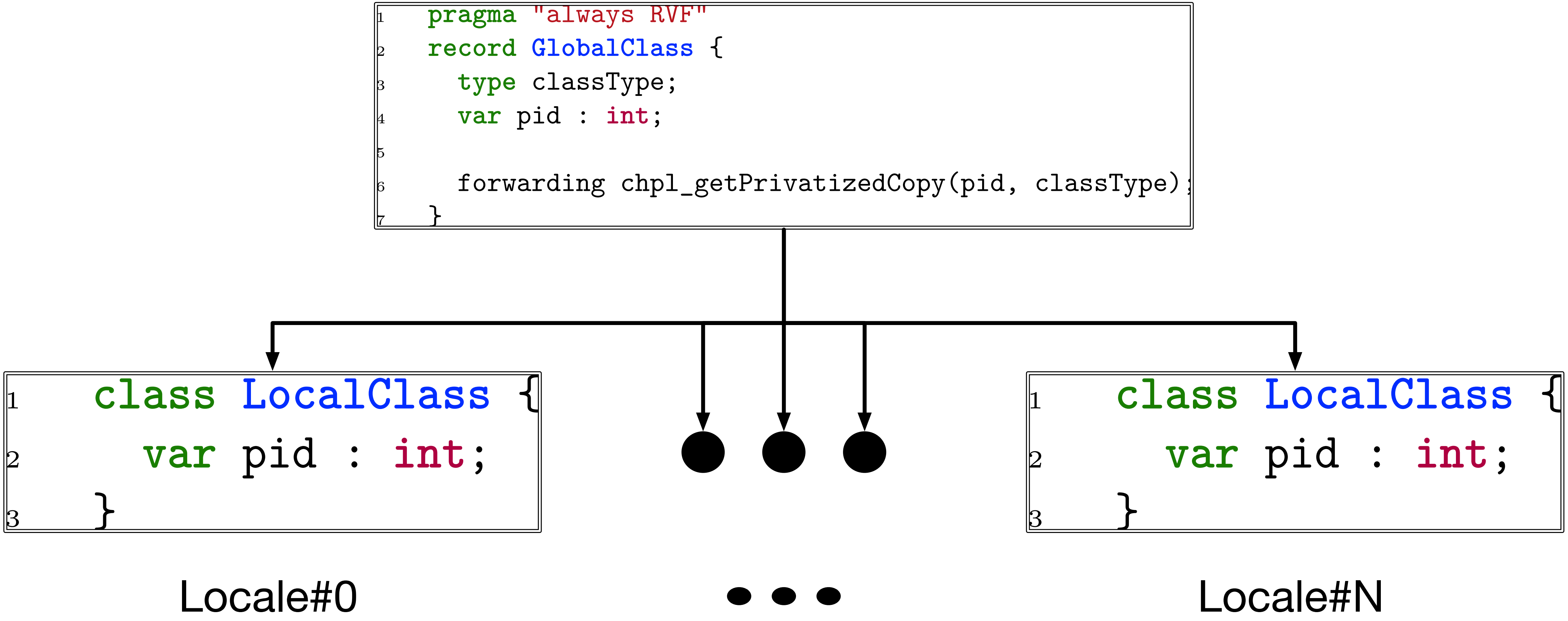
Minimalism

- CAL is an aggregation library
  - Processing of the aggregated data is deferred to the user
  - Buffer is returned to the last task that filled it

```
const msg = "From Locale#0 to Locale#1";
const loc = Locales[1];
var aggregator = new Aggregator(string);
var buffer = aggregator.aggregate(msg, loc);
if buffer != nil then handleBuffer(buffer);
[(buf, loc) in aggregator.flush()] on loc do handleBuffer(buf);
```

Distributed Object Pattern

- Use privatization to enable global-view programming
  - GlobalClass forwards access to per-locale LocalClass *privatized instances*
  - Each privatized instance can communicate and coordinate with others



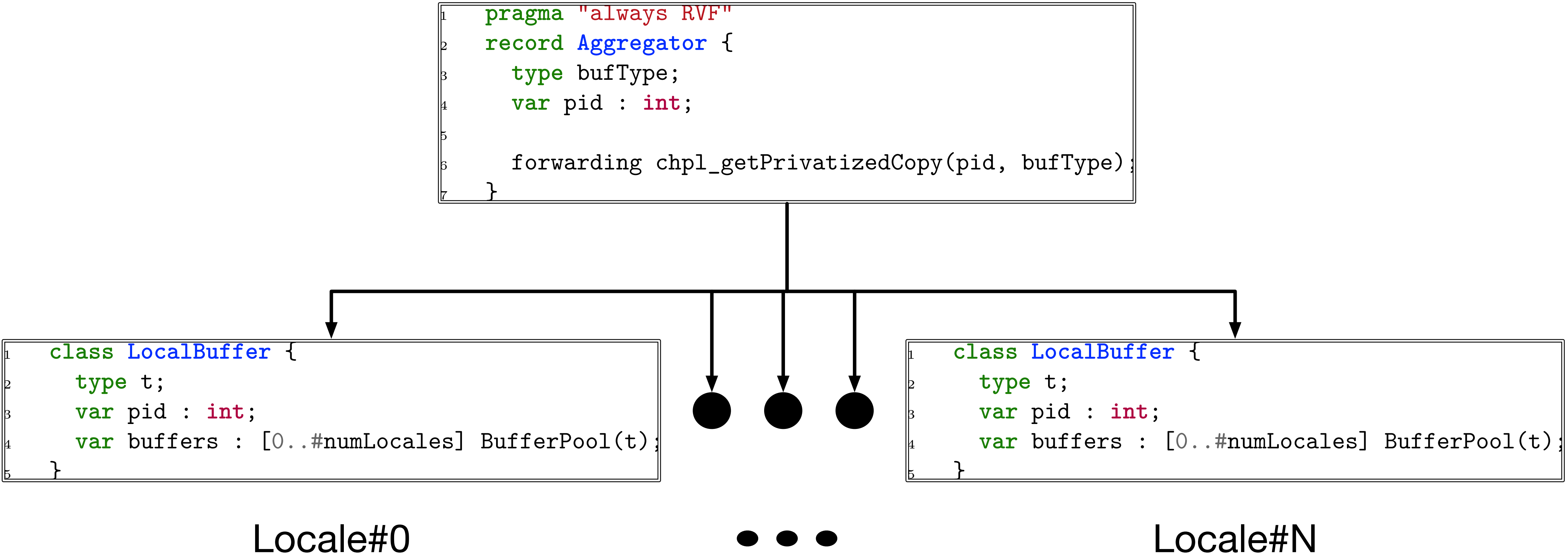
```
pragma "always RVF"
record GlobalClass {
  type classType;
  var pid : int;

  forwarding chpl_getPrivatizedCopy(pid, classType);
}
```

```
class LocalClass {
  var pid : int;
}
```

```
class LocalClass {
  var pid : int;
}
```

Aggregator

- Aggregator forwards all accesses to per-locale privatized instances
- Distributed and parallel access is abstracted
  - Supports global-view programming



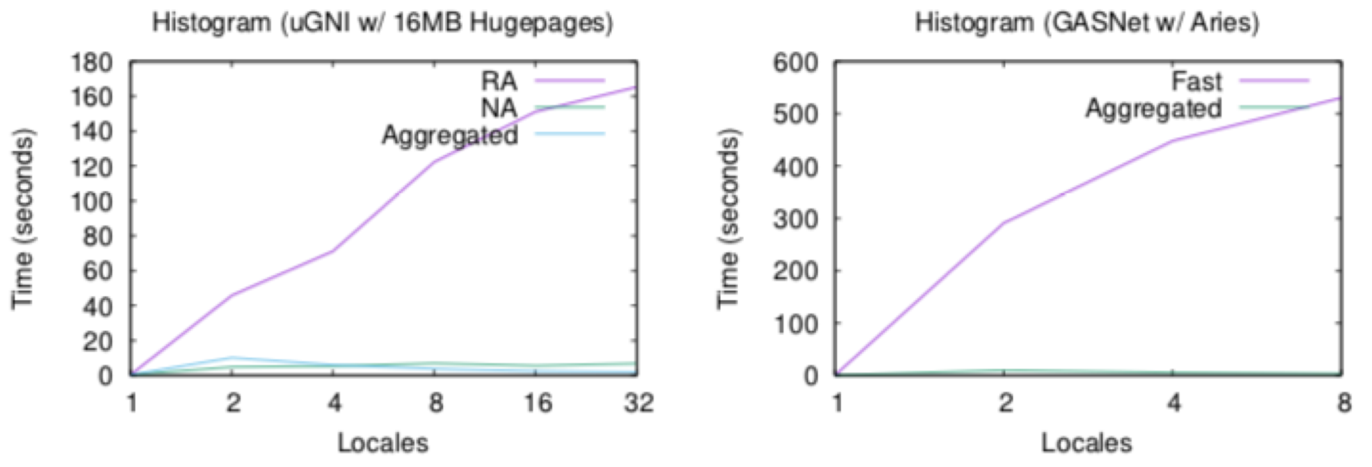
```
pragma "always RVF"
record Aggregator {
  type bufType;
  var pid : int;

  forwarding chpl_getPrivatizedCopy(pid, bufType);
}
```

```
class LocalBuffer {
  type t;
  var pid : int;
  var buffers : [0..#numLocales] BufferPool(t);
}
```

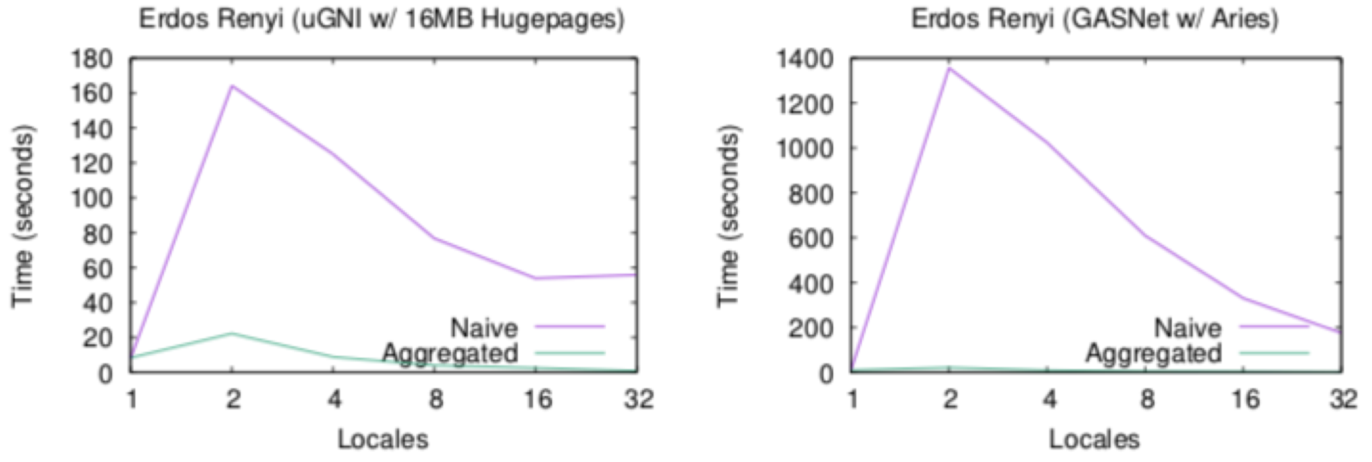
```
class LocalBuffer {
  type t;
  var pid : int;
  var buffers : [0..#numLocales] BufferPool(t);
}
```

Aggregator - Performance

- 10x – 100x speedup at 32 nodes
  - Histogram
  - Hypergraph Generation



(a) uGNI Histogram (b) GASNet Histogram



(a) uGNI Erdos Renyi (b) GASNet Erdos Renyi

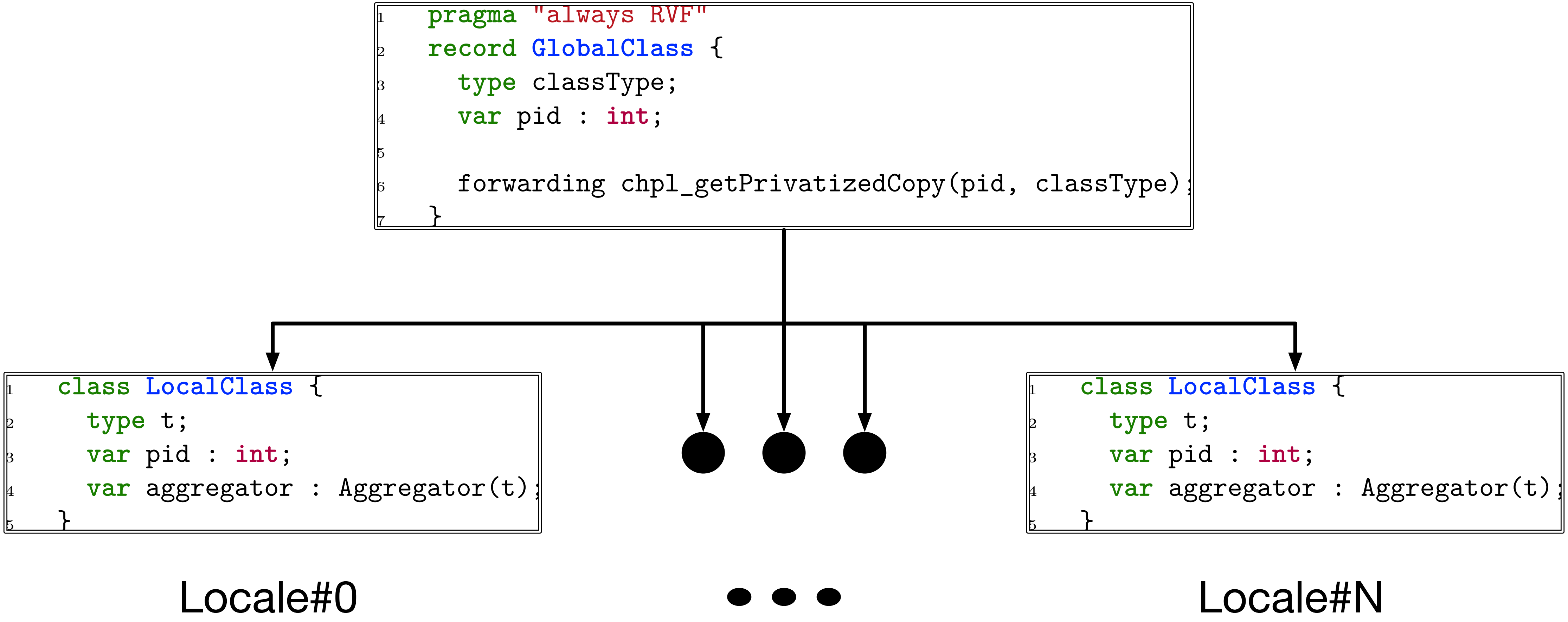
Distributed - Example

- Aggregator is allocated on Locale#0, but accessible from Locale#1
  - Accesses are forwarded to Locale#1’s privatized instance
  - Global-View Programming
- Implicit parallelism (line 9) vs Explicit parallelism (line 11)

```
1   var aggregator = new Aggregator(int);
2   // Migrate to Locale #1 from Locale #0
3   on Locales[1] {
4     // Aggregate single value to Locale #0
5     var buffer = aggregator.aggregate(0, Locales[0]);
6     // If non-nil, then handle buffer.
7     if buffer != nil then handleBuffer(buffer);
8     // Aggregate multiple units of data via Chapel's implicit parallelism
9     var buffers = aggregator.aggregate(1..1024, Locales[0]);
10    // Check if any of the buffers are nil
11    [buf in buffers] if buf != nil then handleBuffer(buf);
12  }
```

Modularity

- Composition of Distributed Objects
  - Aggregator can be used within other global-view data structures
  - Future of Distributed Object Oriented Programming (?)



```
pragma "always RVF"
record GlobalClass {
  type classType;
  var pid : int;

  forwarding chpl_getPrivatizedCopy(pid, classType);
}
```

```
class LocalClass {
  type t;
  var pid : int;
  var aggregator : Aggregator(t);
}
```

```
class LocalClass {
  type t;
  var pid : int;
  var aggregator : Aggregator(t);
}
```

Future Works

- Software release of CAL
  - Currently only available as module under Chapel HyperGraph Library (CHGL)
    - ✓ github.com/pnnl/chgl
  - Independent release coming soon (?)
- Integration into Chapel
  - Mason package or Standard Module (?)
  - Run-time integration
- Aggregation handlers as first-class functions
  - Once Chapel has better first-class function support

Potential Application
Light Weight Tasks (LWT)

- Chapel Tasks are infeasible to use in fire-and-forget manner
  - Stack size of tasks in Chapel are static and large (8MB default)
  - Task migration can be made asynchronous but is not aggregated
- Solution – Make a library for LWT
  - Use Distributed Object pattern for Global-View programming
  - Use *Aggregator* for aggregation
  - Use First-Class Functions (once improved) to represent a lightweight task

```
var lwt = new LWT(visit);
proc visit(v : Vertex) {
  for vv in neighbors(v) {
    if hasProperty(vv) {
      lwt.spawn(vv, vv.locale);
    }
  }
}
forall v in vertices {
  if hasProperty(v) {
    lwt.spawn(v);
  }
}
```

Vertex Degree Distribution

```
// Find largest degree of all vertices in distributed graph
var N = max reduce [v in graph.getVertices()] graph.degree(v);
// Histogram is cyclically distributed over all locales
var histogramDomain = {1..N} dmapped Cyclic(startIdx=1);
var histogram : [histogramDomain] atomic int;

// Aggregate increments to histogram
var aggregator = new Aggregator(int);
forall v in graph.getVertices() {
  const deg = graph.degree(v);
  const loc = histogram[deg].locale;
  var buffer = aggregator.aggregate(deg, loc);
  if buffer != nil {
    on loc do [deg in buffer] histogram[deg].add(1);
    buffer.done();
  }
}

// Flush
forall (buf, loc) in aggregator.flush() {
  on loc do [deg in buf] histogram[deg].add(1);
  buffer.done();
}
```