Hewlett Packard Enterprise

ONE-DAY CHAPEL TUTORIAL SESSION 3: PARALLELISM IN CHAPEL

Chapel Team

October 16, 2023

ONE DAY CHAPEL TUTORIAL

- 9-10:30: Getting started using Chapel for parallel programming
- 10:30-10:45: break
- 10:45-12:15: Chapel basics in the context of the n-body example code
- 12:15-1:15: lunch
- 1:15-2:45: Distributed and shared-memory parallelism especially w/arrays (data parallelism)
- 2:45-3:00: break
- 3:00-4:30: More parallelism including for asynchronous parallelism (task parallelism)
- 4:30-5:00: Wrap-up including gathering further questions from attendees

OUTLINE: PARALLELISM IN CHAPEL

- Recall processing files in parallel
- Data parallelism concepts and examples including multi-locale parallelism with distributions
- Domains
- Forall Loops
- Domain Distributions
- Using a Different Domain Distribution
- Implicit Communication: Remote writes/Puts and Reads/Gets
- Parallelizing a 1D heat diffusion solver (Hands On)
- Heat 2D example with CommDiagnostics (Hands On)

RECALL PROCESSING FILES IN PARALLEL

RECALL: ANALYZING MULTIPLE FILES USING PARALLELISM

parfilekmer.chpl

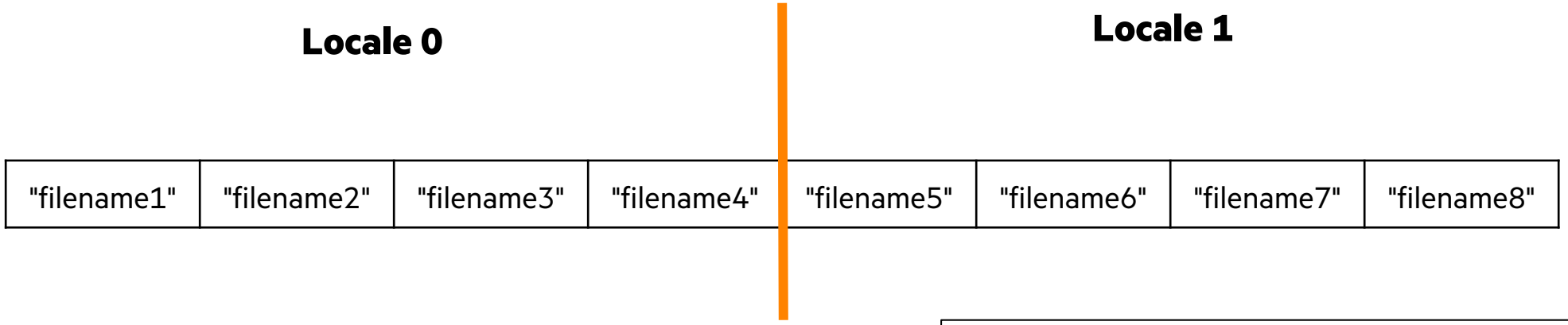
parfilekmer.chpl

```
use FileSystem;
config const dir = "DataDir";
var fList = findFiles(dir);
var filenames =
  blockDist.createArray(0..<fList.size,string);
filenames = fList;

// per file word count
forall f in filenames {
   ...
   // code from kmer.chpl
   ...
}
```

```
prompt> chpl --fast parfilekmer.chpl
prompt> ./parfilekmer -nl 1
prompt> ./parfilekmer -nl 4
```

- shared and distributed-memory parallelism using 'forall'
  - in other words, parallelism within the locale/node and across locales/nodes
- a distributed array
- command line options to indicate number of locales

RECALL: BLOCK DISTRIBUTION OF ARRAY OF STRINGS




```
prompt> chpl --fast parfilekmer.chpl
prompt> ./parfilekmer -nl 2
```

- Array of strings for filenames is distributed across locales
- 'forall' will do parallelism across locales and then within each locale to take advantage of multicore

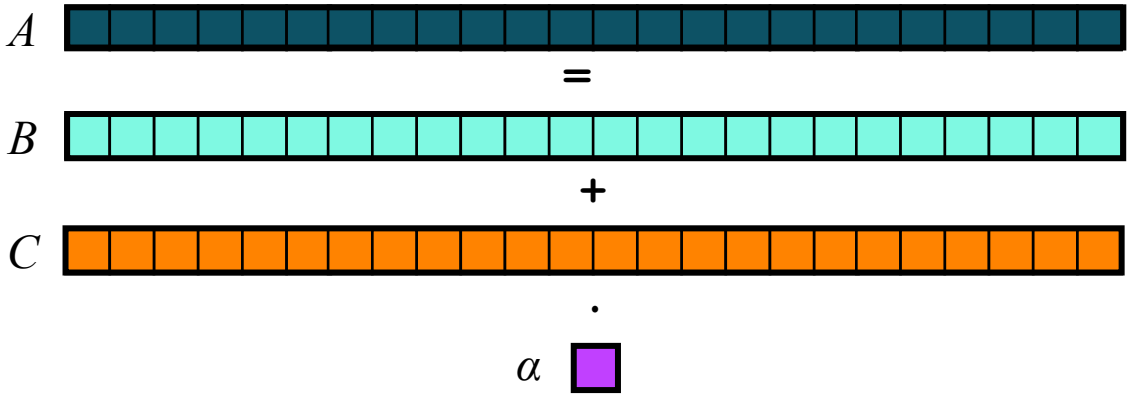
DATA PARALLELISM CONCEPTS AND EXAMPLES INCLUDING MULTI-LOCALE PARALLELISM WITH DISTRIBUTIONS

STREAM TRIAD: A PARALLEL COMPUTATION

Given: m-element vectors A, B, C

Compute: $\forall i \in 1..m,\ A_i = B_i + \alpha \cdot C_i$

In pictures:

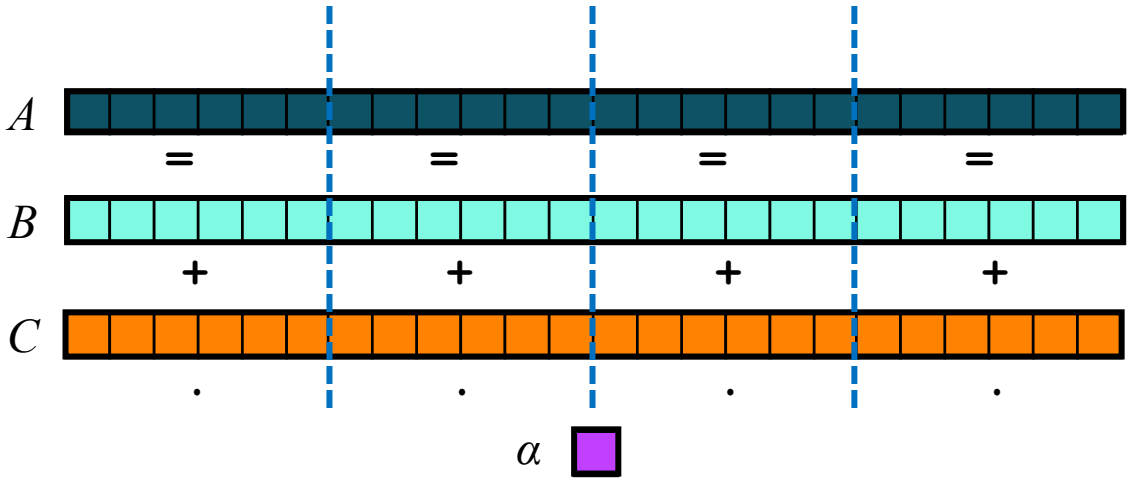


STREAM TRIAD: A PARALLEL COMPUTATION

Given: m-element vectors A, B, C

Compute: $\forall i \in 1..m,\ A_i = B_i + \alpha \cdot C_i$

In pictures, in parallel (shared memory / multicore):

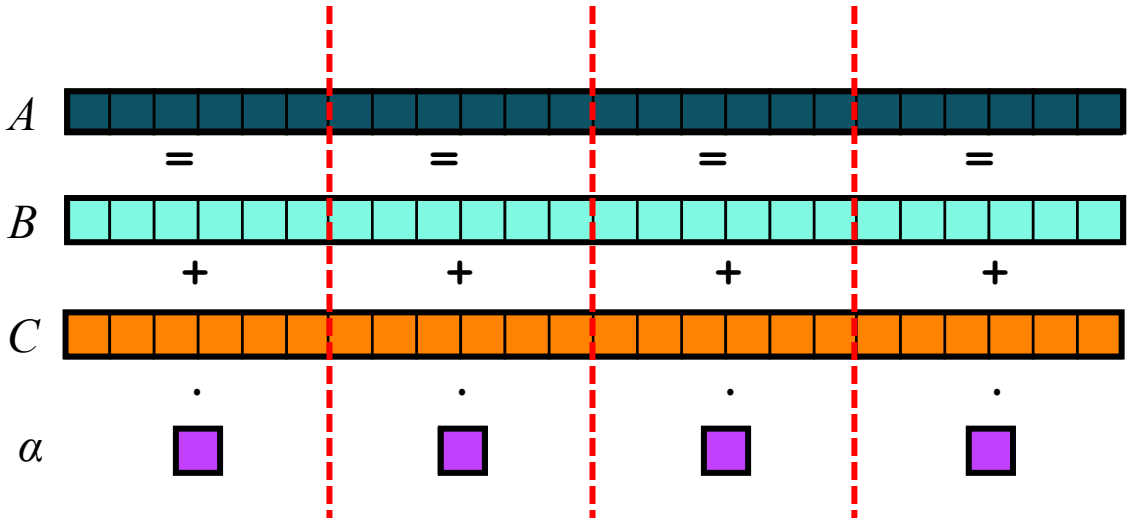


STREAM TRIAD: A PARALLEL COMPUTATION

Given: m-element vectors A, B, C

Compute: $\forall i \in 1..m,\ A_i = B_i + \alpha \cdot C_i$

In pictures, in parallel (distributed memory):

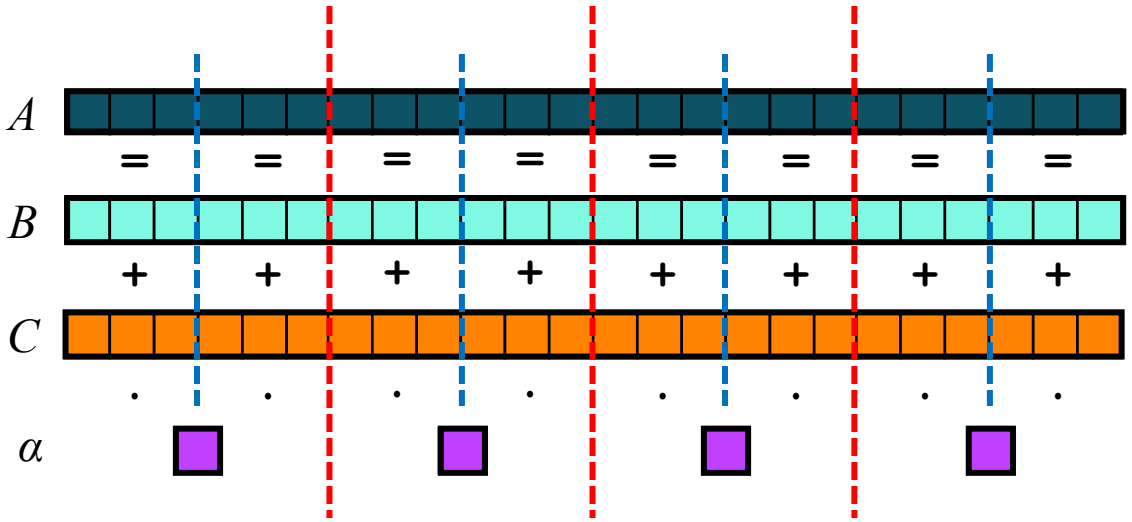


STREAM TRIAD: A PARALLEL COMPUTATION

Given: m-element vectors A, B, C

Compute: $\forall i \in 1..m,\ A_i = B_i + \alpha \cdot C_i$

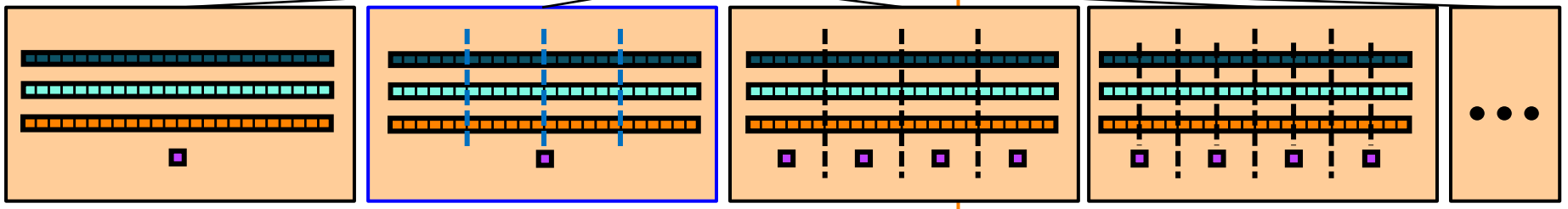
In pictures, in parallel (distributed memory multicore):



STREAM TRIAD: CHAPEL

```
use BlockDist;

config const m = 1000,
             alpha = 3.0;

const ProblemSpace = blockDist.createDomain({1..m});

var A, B, C: [ProblemSpace] real;

B = 2.0;
C = 1.0;

A = B + alpha * C;
```

The special sauce:
How should this index set—and any arrays and computations over it—be mapped to the system?



Philosophy: Good, *top-down* language design can tease system-specific implementation details away from an algorithm, permitting the compiler, runtime, applied scientist, and HPC expert to each focus on their strengths.

DATA PARALLELISM, BY EXAMPLE

03-domain-distributions.chpl

```
config const n = 1000;
var D = {1..n, 1..n};

var A: [D] real;
forall (i,j) in D with (ref A) do
  A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

```
prompt> chpl dataParallel.chpl
prompt> ./dataParallel -nl 1 --n=5
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```

DOMAINS

DATA PARALLELISM, BY EXAMPLE

03-domain-distributions.chpl

Domains (Index Sets)

```
config const n = 1000;
var D = {1..n, 1..n};

var A: [D] real;
forall (i,j) in D with (ref A) do
  A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

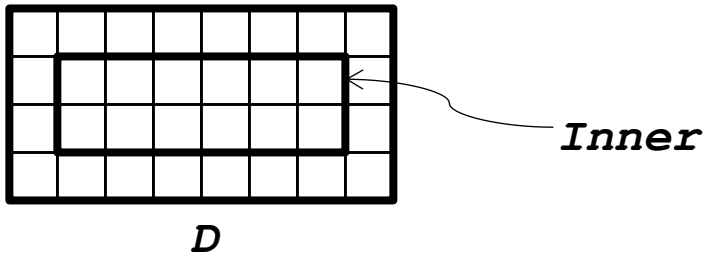
```
prompt> chpl dataParallel.chpl
prompt> ./dataParallel -nl 1 --n=5
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```

DOMAINS

Domain:

- A first-class index set
- The fundamental Chapel concept for data parallelism

```
config const m = 4, n = 8;

const D = {1..m, 1..n};
const Inner = {2..m-1, 2..n-1};
```

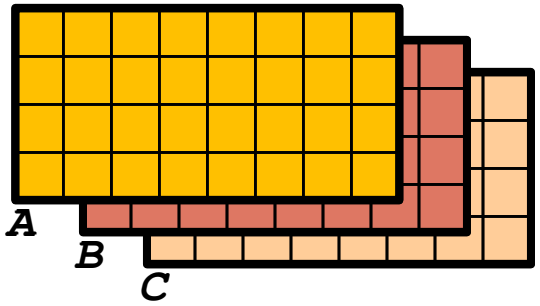


DOMAINS

Domain:

- A first-class index set
- The fundamental Chapel concept for data parallelism
- Useful for declaring arrays and computing with them

```
config const m = 4, n = 8;

const D = {1..m, 1..n};
const Inner = {2..m-1, 2..n-1};

var A, B, C: [D] real;
```



DATA PARALLELISM, BY EXAMPLE

03-domain-distributions.chpl

Arrays

```
config const n = 1000;
var D = {1..n, 1..n};

var A: [D] real;
forall (i,j) in D with (ref A) do
  A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

```
prompt> chpl dataParallel.chpl
prompt> ./dataParallel -nl 1 --n=5
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```

FORALL LOOPS

DATA PARALLELISM, BY EXAMPLE

03-domain-distributions.chpl

Data-Parallel Forall Loops

```
config const n = 1000;
var D = {1..n, 1..n};

var A: [D] real;
forall (i,j) in D with (ref A) do
  A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

```
prompt> chpl dataParallel.chpl
prompt> ./dataParallel -nl 1 --n=5
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```

FORALL LOOPS

Forall loops: Central concept for data parallel computation

- Like for-loops, but parallel
- Implementation details determined by iterand (e.g., *D* below)
  - specifies number of tasks, which tasks run which iterations, …
  - in practice, typically uses a number of tasks appropriate for target HW

```
forall (i,j) in D with (ref A) do
  A[i,j] = i + j/10.0;
```

1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8
2.1	2.2	2.3	2.4	2.5	2.6	2.7	2.8
3.1	3.2	3.3	3.4	3.5	3.6	3.7	3.8
4.1	4.2	4.3	4.4	4.5	4.6	4.7	4.8

Forall loops assert…

…parallel safety: OK to execute iterations simultaneously

…order independence: iterations could occur in any order

…serializability: all iterations could be executed by one task

- e.g., can't have synchronization dependences between iterations

COMPARISON OF LOOPS: FOR, FORALL, AND COFORALL

For loops: executed using one task

- use when a loop must be executed serially
- or when one task is sufficient for performance

Forall loops: typically executed using 1 < #tasks << #iters

- use when a loop *should* be executed in parallel…
- …but *can* legally be executed serially
- use when desired # tasks << # of iterations

Coforall loops: executed using a task per iteration

- use when the loop iterations *must* be executed in parallel
- use when you want # tasks == # of iterations
- use when each iteration has substantial work

DATA PARALLELISM, BY EXAMPLE

03-domain-distributions.chpl

This is a shared memory program

Nothing has referred to remote locales, explicitly or implicitly

```
config const n = 1000;
var D = {1..n, 1..n};

var A: [D] real;
forall (i,j) in D with (ref A) do
  A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

```
prompt> chpl dataParallel.chpl
prompt> ./dataParallel -nl 1 --n=5
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```

DOMAIN DISTRIBUTIONS

DISTRIBUTED DATA PARALLELISM, BY EXAMPLE

03-domain-distributions.chpl

Domain Distribution
(Map Data Parallelism to the System)

```
use CyclicDist;
config const n = 1000;
var D = cyclicDist.createDomain({1..n, 1..n});

var A: [D] real;
forall (i,j) in D with (ref A) do
  A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

```
prompt> chpl dataParallel.chpl
prompt> ./dataParallel --n=5 -nl 4
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```

DISTRIBUTED DATA PARALLELISM, BY EXAMPLE

03-domain-distributions.chpl

High-level distributed and shared memory parallelism

```
use CyclicDist;
config const n = 1000;
var D = cyclicDist.createDomain({1..n, 1..n});

var A: [D] real;
forall (i,j) in D with (ref A) do
  A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

Provides programmability and control

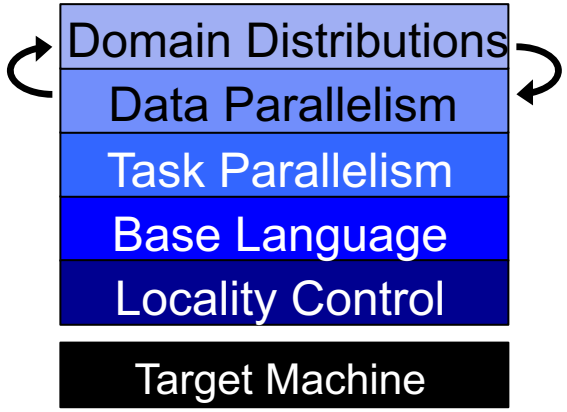
- Lowering of code is well-defined
- User can control details
- Part of Chapel's *multiresolution philosophy*…

```
prompt> chpl dataParallel.chpl
prompt> ./dataParallel --n=5 --nl 4
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```

CHAPEL'S MULTIRESOLUTION PHILOSOPHY

Multiresolution Design: Support multiple tiers of features

- higher levels for programmability, productivity
- lower levels for greater degrees of control
- build the higher-level concepts in terms of the lower
- permit users to intermix layers arbitrarily



DISTRIBUTED DATA PARALLELISM, BY EXAMPLE

03-domain-distributions.chpl

Chapel's prescriptive approach:

```
forall (i,j) in D do…
```

⇒ invoke and inline D's default parallel iterator

- defined by D's type / domain distribution

default domain distribution

- create a task per local core
- block indices across tasks

```
config const n = 1000;
var D = {1..n, 1..n};

var A: [D] real;
forall (i,j) in D with (ref A) do
  A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

```
prompt> chpl dataParallel.chpl
prompt> ./dataParallel --n=5 -nl 1
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```

DISTRIBUTED DATA PARALLELISM, BY EXAMPLE

03-domain-distributions.chpl

Chapel's prescriptive approach:

```
forall (i,j) in D do…
```

⇒ invoke and inline D's default parallel iterator

- defined by D's type / domain distribution

```
use CyclicDist;
config const n = 1000;
var D = cyclicDist.createDomain({1..n, 1..n});

var A: [D] real;
forall (i,j) in D with (ref A) do
  A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

cyclic domain distribution

on each target locale…

- create a task per core
- block local indices across tasks

```
prompt> chpl dataParallel.chpl
prompt> ./dataParallel --n=5 -nl=4
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```

DISTRIBUTED DATA PARALLELISM, BY EXAMPLE

03-domain-distributions.chpl

Chapel's prescriptive approach:

```
forall (i,j) in D do…
```

What if I don't like D's iteration strategy?

```
use CyclicDist;
config const n = 1000;
var D = cyclicDist.createDomain({1..n, 1..n});
var A: [D] real;
forall (i,j) in D with (ref A) do
  A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

Write and call your own parallel iterator:

```
forall (i,j) in myParIter(D) do…
```

DISTRIBUTED DATA PARALLELISM, BY EXAMPLE

03-domain-distributions.chpl

Chapel's prescriptive approach:

```
forall (i,j) in D do…
```

What if I don't like D's iteration strategy?

```
use CyclicDist;
config const n = 1000;
var D = cyclicDist.createDomain({1..n, 1..n});
var A: [D] real;
forall (i,j) in D with (ref A) do
  A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

Write and call your own parallel iterator:

```
forall (i,j) in myParIter(D) do…
```

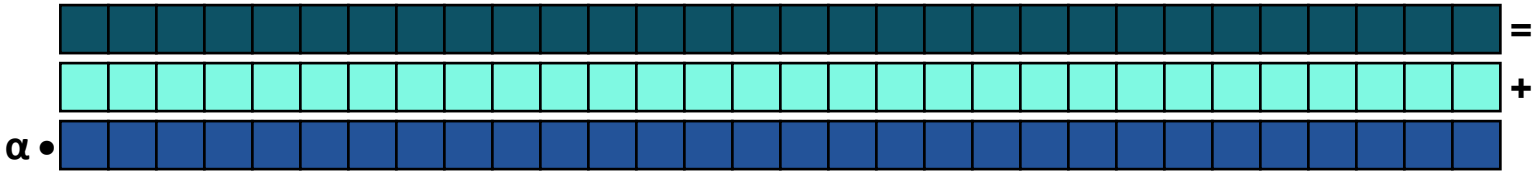
Or use a different domain distribution:

```
var D = blockDist.createDomain({1..n, 1..n});
```

USING A DIFFERENT DOMAIN DISTRIBUTION

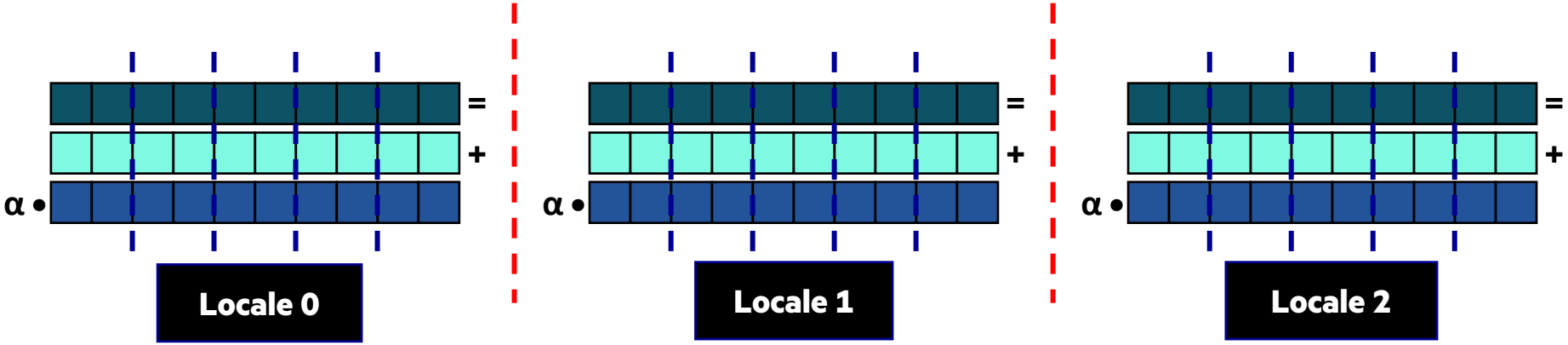
DOMAIN DISTRIBUTIONS: A MULTIRESOLUTION FEATURE

Domain distributions are "recipes" that instruct the compiler how to map the global view of a computation…



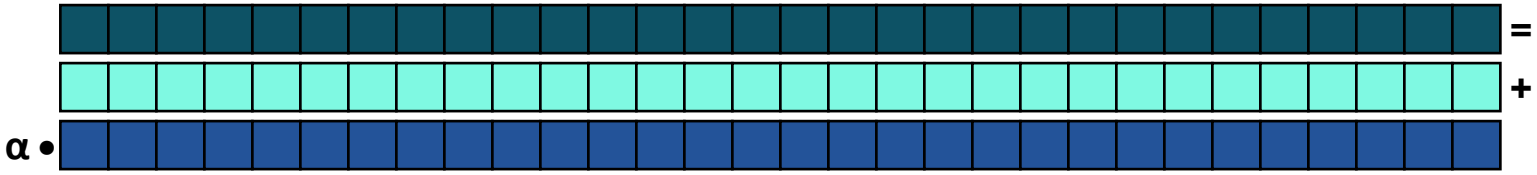

```
A = B + alpha * C;
```

…to the target locales' memory and processors:



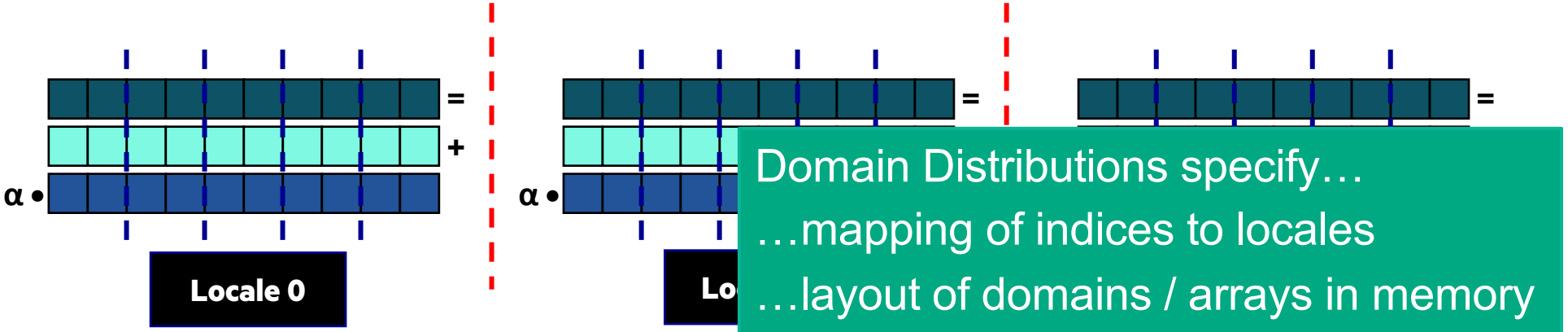
DOMAIN DISTRIBUTIONS: A MULTIRESOLUTION FEATURE

Domain distributions are "recipes" that instruct the compiler how to map the global view of a computation…




```
A = B + alpha * C;
```

…to the target locales' memory and processors:



Domain Distributions specify…

…mapping of indices to locales

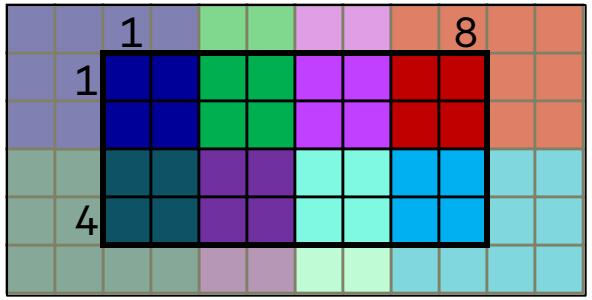
…layout of domains / arrays in memory

…parallel iteration strategies

…core operations on arrays / domains

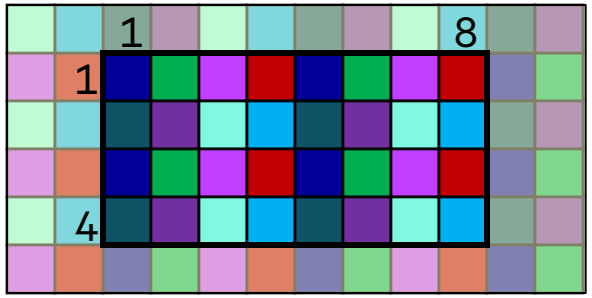
SAMPLE DOMAIN DISTRIBUTIONS: BLOCK AND CYCLIC

```
var Dom = blockDist.createDomain({1..4, 1..8});
```



distributed to


```
var Dom = cyclicDist.createDomain({1..4, 1..8});
```



distributed to

IMPLICIT COMMUNICATION:
REMOTE WRITES/PUTS AND READS/GETS

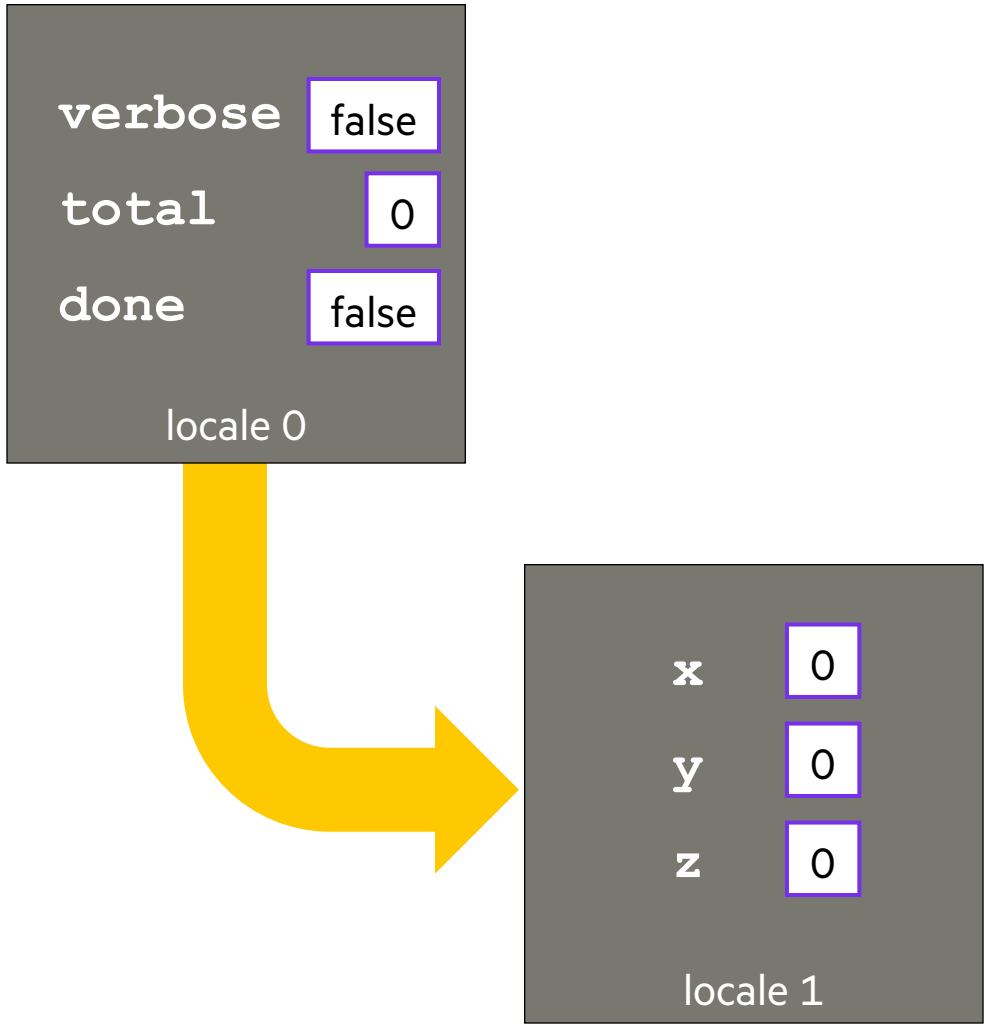
CHAPEL SUPPORTS A GLOBAL NAMESPACE WITH PUTS AND GETS

Note 1: Variables are allocated on the locale where the task is running

03-onClause.chpl

03-onClause.chpl

```
config const verbose = false;
var total = 0,
    done = false;

…

on Locales[1] {
  var x, y, z: int;
  …



}
```



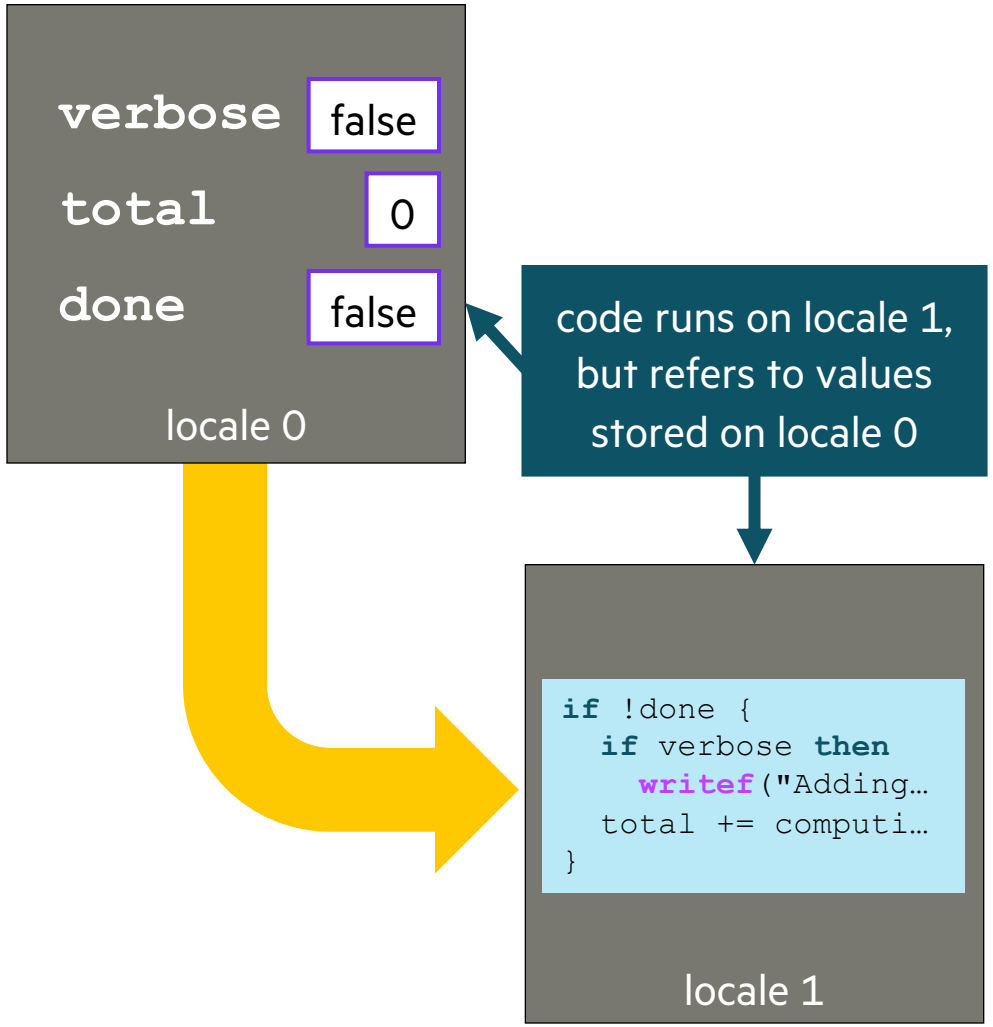
CHAPEL SUPPORTS A GLOBAL NAMESPACE WITH PUTS AND GETS

Note 2: Tasks can refer to lexically visible variables, whether local or remote

03-onClause.chpl

03-onClause.chpl

```
config const verbose = false;
var total = 0,
    done = false;

…

on Locales[1] {
  if !done {
    if verbose then
      writef("Adding locale 1's contribution");
    total += computeMyContribution();
  }
}
```



ARRAY-BASED PARALLELISM AND LOCALITY

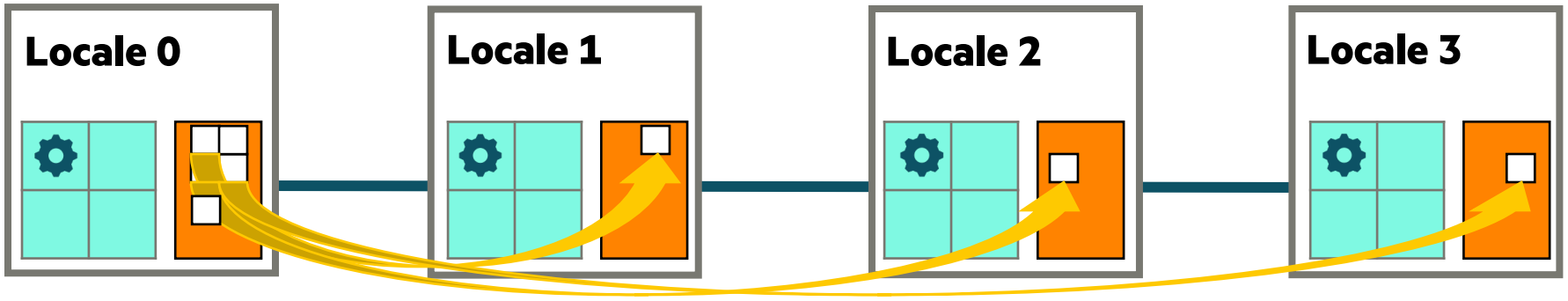
03-basics-distarr.chpl

03-basics-distarr.chpl

```
writeln("Hello from locale ", here.id);

var A: [1..2, 1..2] real;

use BlockDist;

var D = blockDist.createDomain({1..2, 1..2});
var B: [D] real;
B = A;
```

Chapel also supports distributed domains (index sets) and arrays

They also result in parallel distributed computation

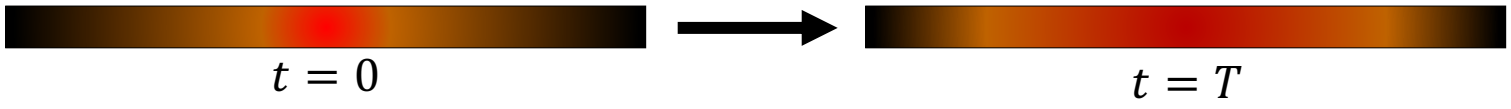


PARALLELIZING A 1D HEAT DIFFUSION SOLVER (HANDS ON)

1D HEAT EQUATION EXAMPLE

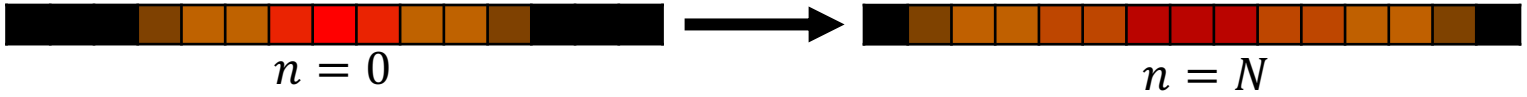
heat-1D.chpl

Differential equation: $\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}$



Discretized (finite difference) equation: $u_i^{n+1} = u_i^n + \alpha\,(u_{i-1}^n - 2u_i^n + u_{i+1}^n)$

- where $i \in \Omega \subset \mathbb{R}^1$ are discrete points in space, and $(n, n+1, \ldots)$ are discrete instances in time



Finite difference algorithm:

- define Ω to be a set of discrete points along the x-axis
- define $\widehat{\Omega}$ over the same points, excluding the boundaries
- define an array u to over Ω
- set some initial conditions
- create a temporary copy of u, named un
- for N timesteps:
  (1) swap u and un
  (2) compute u in terms of un over $\widehat{\Omega}$

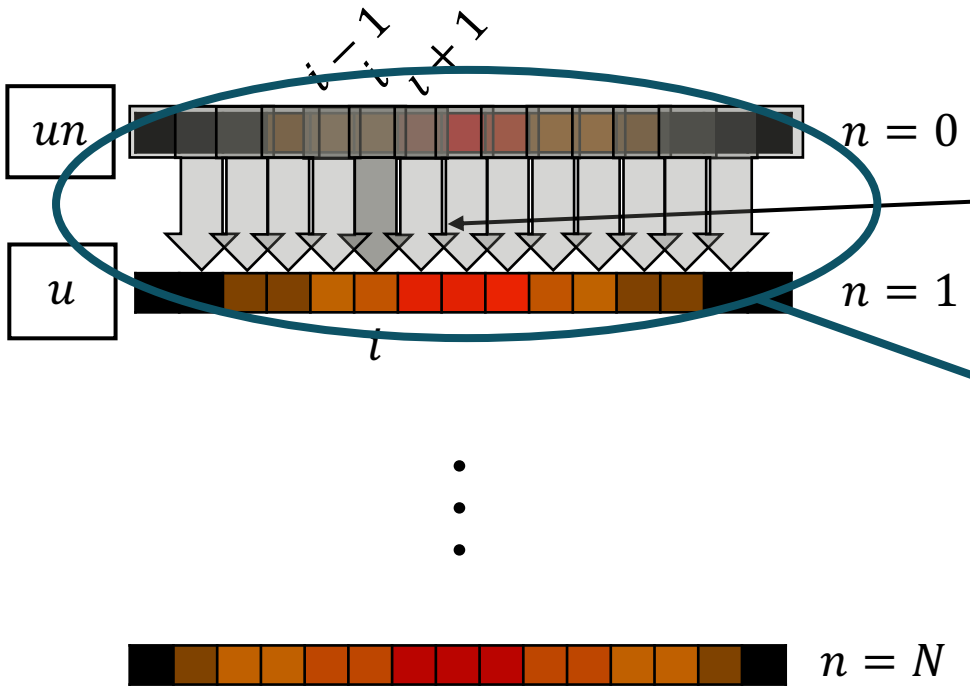
```
const omega = {0..<nx},
      omegaHat = omega.expand(-1);
var u: [omega] real = 1.0;
u[nx/4..3*nx/4] = 2.0;
var un = u;
for 1..N {
  un <=> u;
  forall i in omegaHat do
    u[i] = un[i] + alpha *
           (un[i-1] - 2*un[i] + un[i+1]);
}
```

1D HEAT EQUATION EXAMPLE

heat-1D.chpl

This pattern is often referred to as a *Stencil Computation*

- The values in the array can be computed by applying a "stencil" to its previous state
- Note that in this case, the stencil can be applied to the entire array in parallel
  - each value in un depends strictly on values in u



"stencil"

$$u_i^{n+1} = u_i^n + \alpha\,(u_{i-1}^n - 2u_i^n + u_{i+1}^n)$$

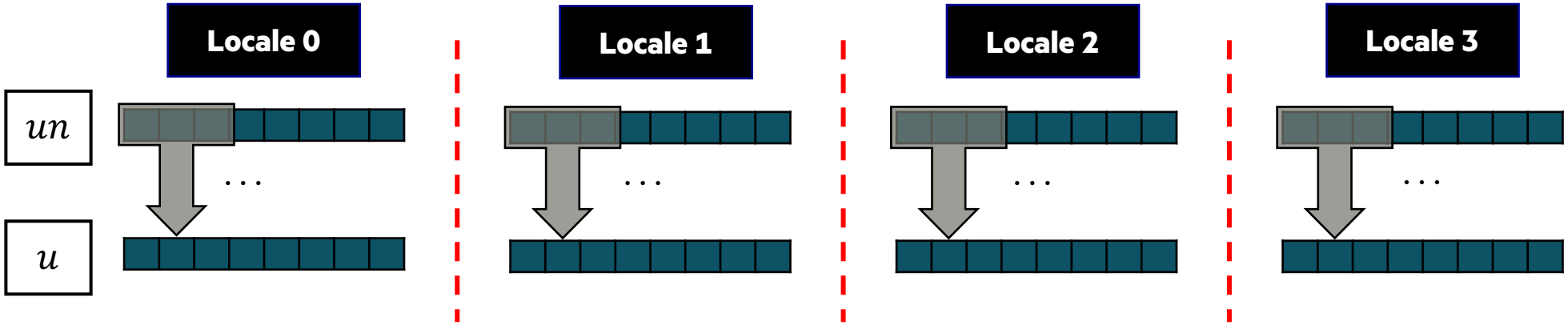
```
...
forall i in omegaHat do
  u[i] = un[i] + alpha *
         (un[i-1] - 2*un[i] + un[i+1]);
...
```

HANDS ON: DISTRIBUTING THE 1D HEAT EQUATION

heat-1D-block.chpl

Imagine we want to simulate a very large domain

- We could use the Block distribution to distribute u and un across multiple locales
  - taking advantage of their memory and compute resources



Look at **heat-1D-block.chpl** and fill in the blanks to make the arrays block-distributed

Hint | Define a block-distributed domain:

```
use BlockDist;
...
const myBlockDom = blockDist.createDomain({1..10});
```

HANDS ON: DISTRIBUTING THE 1D HEAT EQUATION

heat-1D-block-solution.chpl

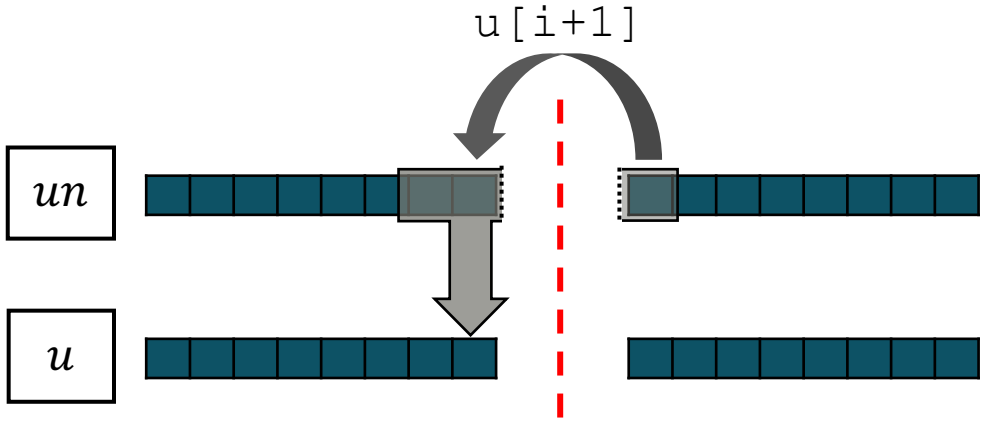
Solution: make 'omega' block-distributed:

```
omega = blockDist.createDomain({0..<nx});
```

Why does this work?

- 'omegaHat' inherits 'omega's distribution
- 'u' is block-distributed
- 'un' inherits 'u's domain (and distribution)
- 'omegaHat' invokes 'blockDist's parallel/distr. iterator
  - the body of the loop is automatically split across multiple tasks on each locale
- Communication occurs automatically when a loop references a value stored on a remote locale

```
const omega =
        blockDist.createDomain({0..<nx}),
      omegaHat = omega.expand(-1);
var u: [omega] real = 1.0;
u[nx/4..3*nx/4] = 2.0;
var un = u;
for 1..N {
  un <=> u;
  forall i in omegaHat do
    u[i] = un[i] + alpha *
           (un[i-1] - 2*un[i] + un[i+1]);
}
```



HEAT 2D EXAMPLE WITH COMMDIAGNOSTICS (HANDS ON)

2D HEAT EQUATION EXAMPLE

heat-2D.chpl

2D and 3D stencil codes are more common and practical

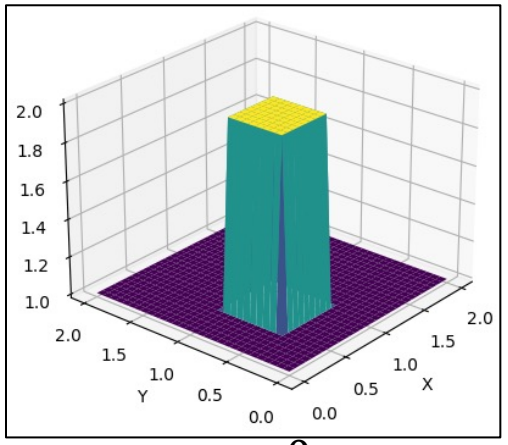
- They also present more interesting considerations for parallelization and distribution

2D heat / diffusion PDE:

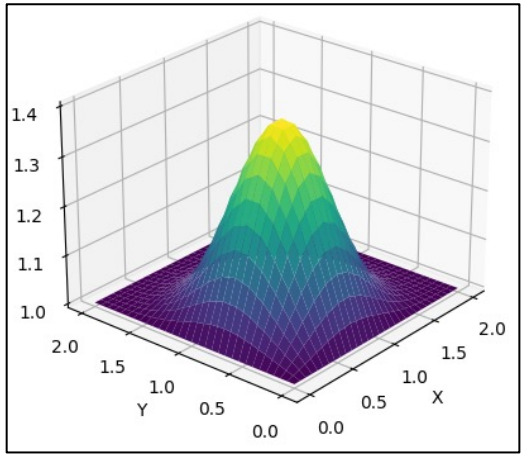
$$\frac{\partial u}{\partial t} = \alpha \Delta u = \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right)$$

Discretized (finite-difference) form:

$$u_{i,j}^{n+1} = u_{i,j}^{n} + \alpha \left(u_{i+1,j}^{n} + u_{i-1,j}^{n} - 4u_{i,j}^{n} + u_{i,j+1}^{n} + u_{i,j-1}^{n} \right)$$



$n = 0$

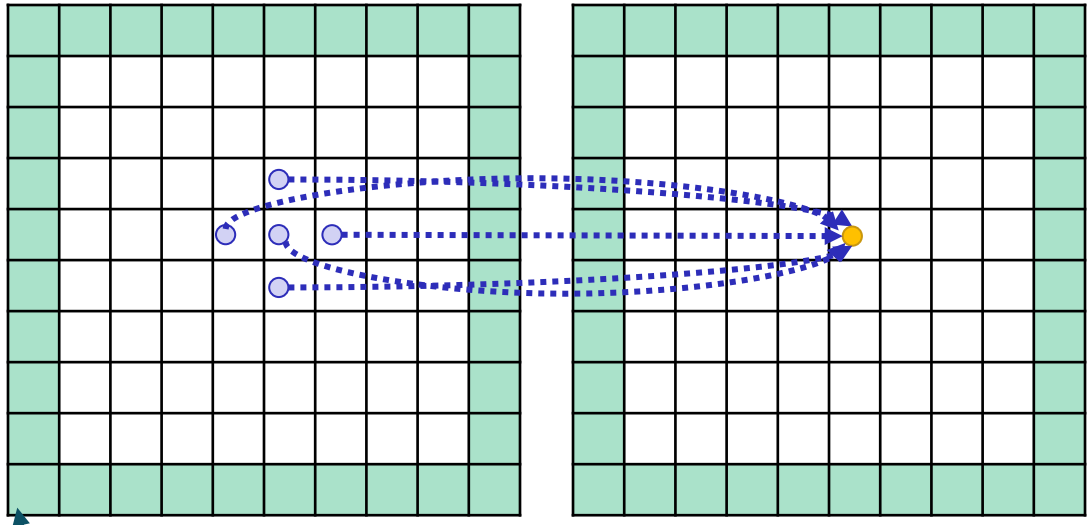


$n = N$

```
const omega = {0..<nx, 0..<ny},
      omegaHat = omega.expand(-1);
var u: [omega] real = 1.0;
u[nx/4..3*nx/4] = 2.0;
var un = u;
for 1..N {
  un <=> u
  forall (i, j) in omegaHat do
    u[i, j] = un[i, j] + alpha * (
                un[i-1, j] + un[i, j-1] +
                un[i+1, j] + un[i, j+1] -
                4 * un[i, j]);
}
```

PARALLEL 2D HEAT EQUATION

heat-2D.chpl



u^n

Stored in un

u^{n+1}

Stored in u

Fixed boundary values

- This computation uses a "5 point stencil"
- Each point in 'u' can be computed in parallel
  - this is accomplished using a 'forall' loop

```
...
  forall (i, j) in omegaHat do
    u[i, j] = un[i, j] + alpha * (
                un[i-1, j] + un[i, j-1] +
                un[i+1, j] + un[i, j+1] -
                4 * un[i, j]);
...
```

$$u_{i,j}^{n+1} = u_{i,j}^{n} + \alpha\left(u_{i-1,j}^{n} + u_{i,j-1}^{n} + u_{i+1,j}^{n} + u_{i,j+1}^{n} - 4u_{i,j}^{n}\right)$$

BLOCK DISTRIBUTED & PARALLEL 2D HEAT EQUATION

heat-2D-block.chpl

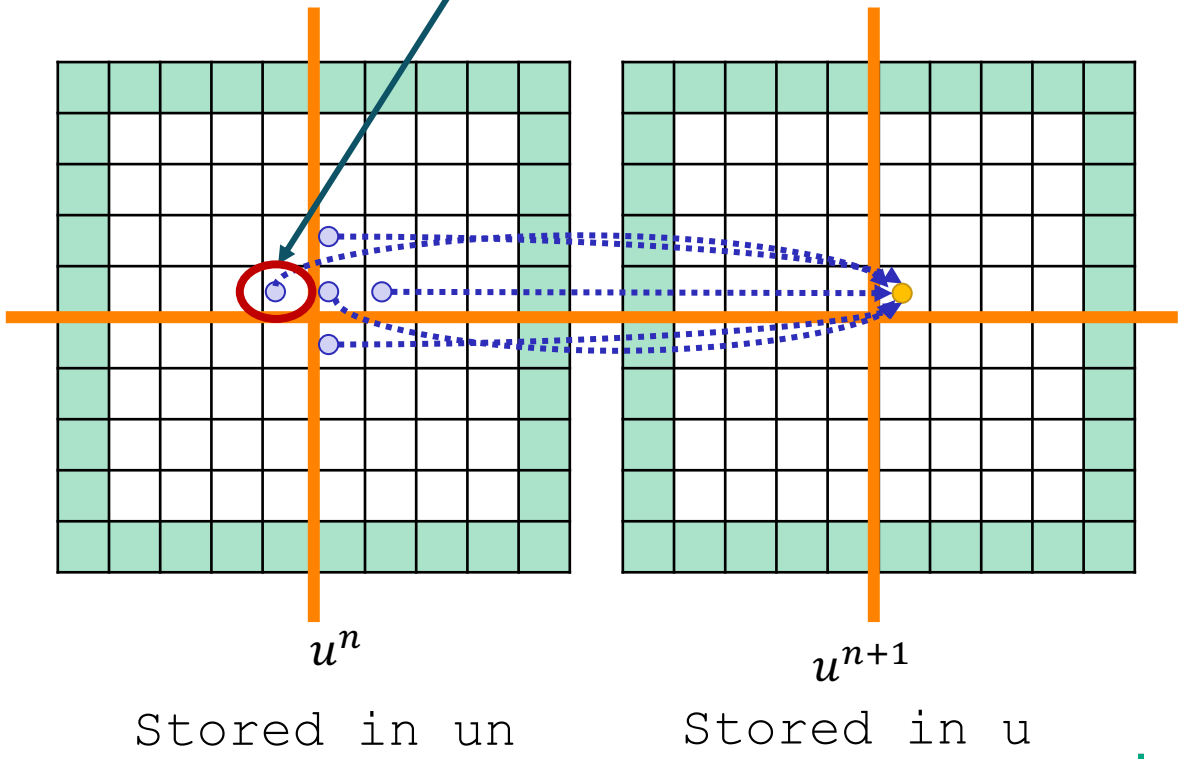
- Declaring distributed domains with the block distribution

```
const Omega = blockDist.createDomain(0..<nx, 0..<ny),
      OmegaHat = Omega.expand(-1);
```

- Distributed & Parallel loop over 'OmegaHat'

```
for 1..nt {
  u <=> un;

  forall (i, j) in OmegaHat do
    u[i, j] = un[i, j] + alpha * (
              un[i-1, j] + un[i, j-1] +
              un[i+1, j] + un[i, j+1] -
              4 * un[i, j]);
}
```

Array access across locale boundaries automatically invokes communication



STENCIL DISTRIBUTED & PARALLEL 2D HEAT EQUATION

heat-2D-stencil.chpl

- Declaring distributed domains with the stencil distribution

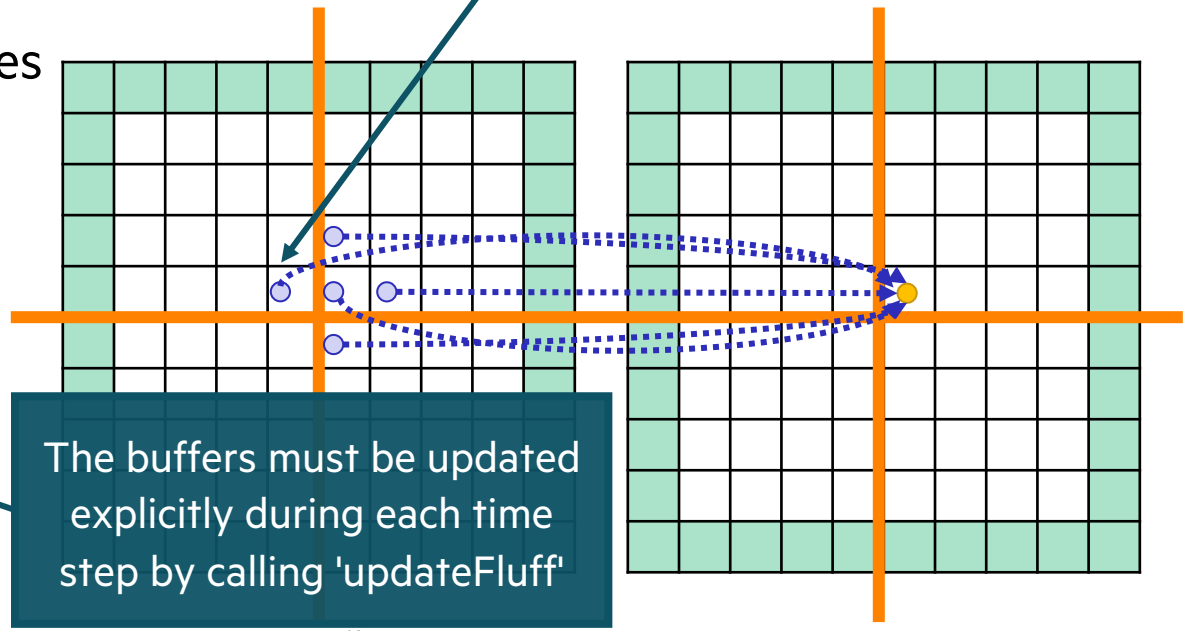
```
const Omega = stencilDist.createDomain(
                {0..<nx, 0..<ny}, fluff=(1,1)),
      OmegaHat = Omega.expand(-1);
```

- Distributed & Parallel loop including buffer updates

```
for 1..nt {
  u <=> un;
  un.updateFluff();
  forall (i, j) in OmegaHat do
    u[i, j] = un[i, j] + alpha * (
              un[i-1, j] + un[i, j-1] +
              un[i+1, j] + un[i, j+1] -
              4 * un[i, j]);
}
```

Array access across locale boundaries (within the fluff region) results in a local buffer access — no communication is required

The buffers must be updated explicitly during each time step by calling 'updateFluff'

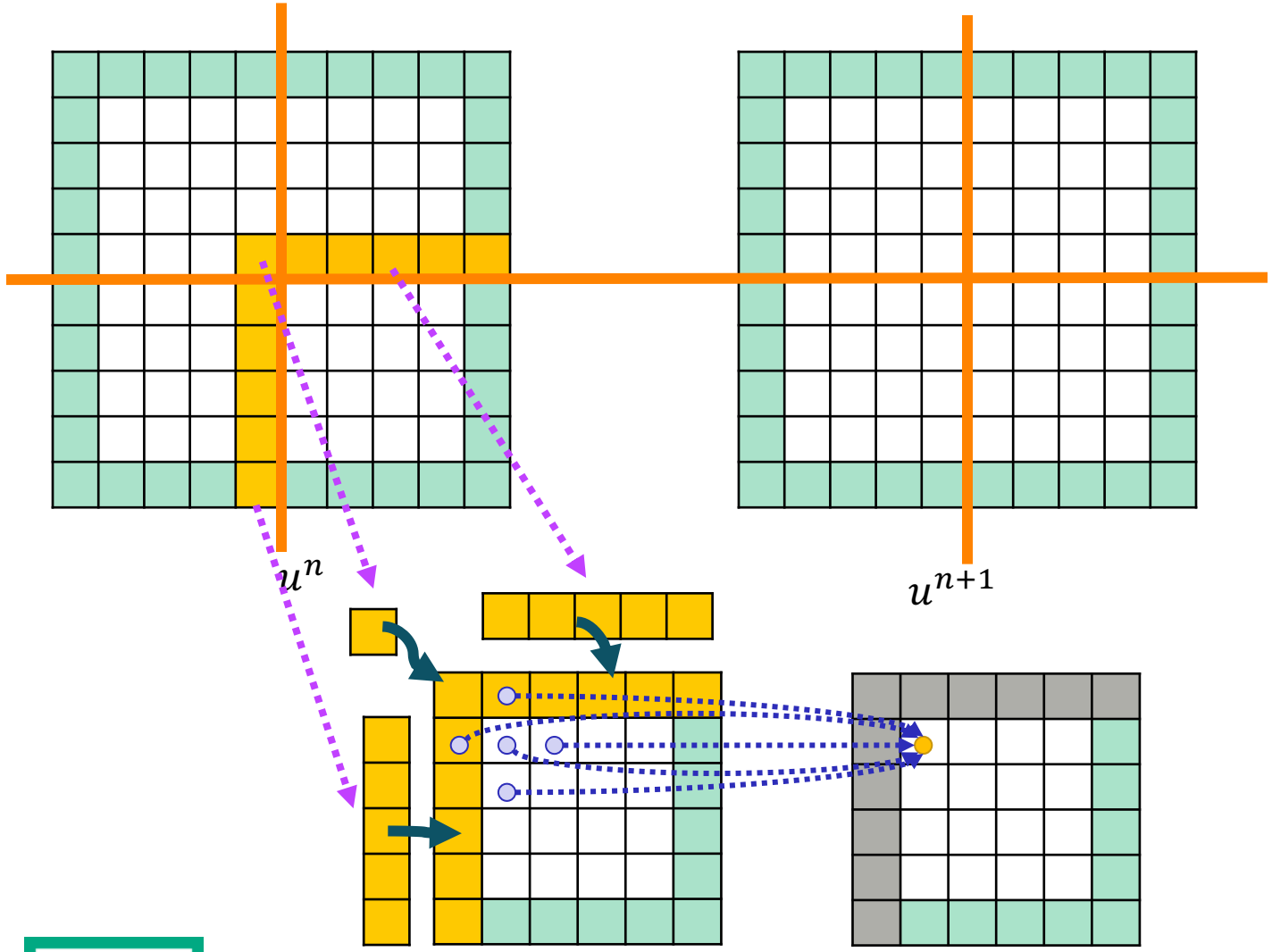


u^n Stored in un

u^{n+1} Stored in u

STENCIL DISTRIBUTED & PARALLEL 2D HEAT EQUATION

heat-2D-stencil.chpl



- Each locale owns a region of the array surrounded by a "fluff" (buffer) region
- Calling 'updateFluff' copies values from neighboring regions of the array into the local buffered region
- Subsequent accesses of those values result in a local memory access, rather than a remote communication

COMM DIAGNOSTICS

The 'CommDiagnostics' module provides functions for tracking comm between locales

- the following is a common pattern:

```
use CommDiagnostics;
...
startCommDiagnostics();
potentiallyCommHeavyOperation();
stopCommDiagnostics();
...
printCommDiagnosticsTable();
```

- which results in a table summarizing comm counts between the **start** and **stop** calls, e.g.,

```
| locale | get | put | execute_on | execute_on_nb |
| -----: | --: | --: | ---------: | ------------: |
|      0 | 10  |   0 |          6 |            12 |
|      1 | 105 |   5 |          0 |             0 |
|      2 | 105 |   4 |          0 |             0 |
|      3 | 105 |   7 |          0 |             0 |
```

- Compiling with '--no-cache-remote' before collecting comm diagnostics is recommended

HANDS ON: HEAT 2D COMM DIAGNOSTICS RESULTS

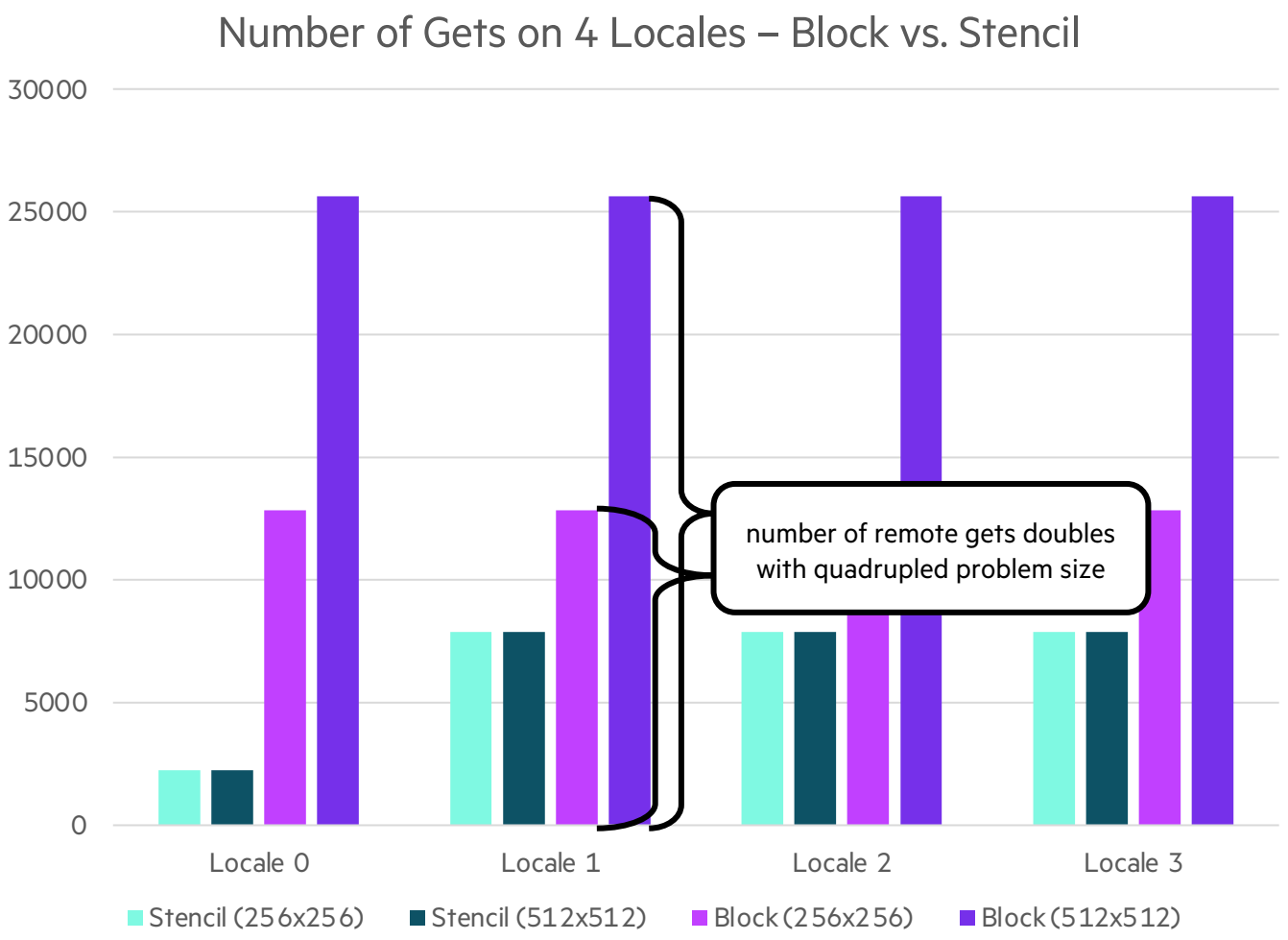
heat-2D-block.chpl, heat-2D-stencil.chpl

- Comparing comm diagnostics for:
  - heat-2D-block.chpl
  - heat-2D-stencil.chpl
- *Compilation:*

```
chpl heat-2D-block.chpl --fast
   --no-cache-remote -sRunCommDiag=true

chpl heat-2D-stencil.chpl -fast
   --no-cache-remote -sRunCommDiag=true
```

- *Execution:*

```
./heat-2D-block -nl4 --nx=256 --ny=256

./heat-2D-stencil -nl4 --nx=512 --ny=512
```



- **Block:** number of gets scales with size
- **Stencil:** static number of gets per iteration

OUTLINE: PARALLELISM IN CHAPEL

- Recall processing files in parallel
- Data parallelism concepts and examples including multi-locale parallelism with distributions
- Domains
- Forall Loops
- Domain Distributions
- Using a Different Domain Distribution
- Implicit Communication: Remote writes/Puts and Reads/Gets
- Parallelizing a 1D heat diffusion solver (Hands On)
- Heat 2D example with CommDiagnostics (Hands On)

SUMMARIZING WHAT WE LEARNED IN SESSION 3

- Data parallelism session
  - Provides shared memory and distributed memory parallelism
  - Distributions like block and cyclic can be applied to arrays of any dimension
  - Main control abstraction is the 'forall' loop
  - 'forall' loop uses default iterator over provided array or domain, but can use own iterator
    - This is an example of multi-resolution design in Chapel, i.e., the 'forall' loop is mapped down to lower-level abstractions like 'coforall'
  - CommDiagnostics module can be used to observe the number of remote puts/writes and gets/reads at runtime

LEARNING OBJECTIVES FOR TODAY'S CHAPEL TUTORIAL

- Familiarity with the Chapel execution model including how to run codes in parallel on a single node, across nodes, and both

- Learn Chapel concepts by compiling and running provided code examples
  - ✓Serial code using map/dictionary, (k-mer counting from bioinformatics)
  - ✓Parallelism and locality in Chapel
  - ✓Distributed parallelism and 1D arrays, (processing files in parallel)
  - ✓Chapel basics in the context of an n-body code
  - ✓Distributed parallelism and 2D arrays, (heat diffusion problem)
  - How to parallelize histogram
  - Using CommDiagnostics for counting remote reads and writes
  - Chapel and Arkouda best practices including avoiding races and performance gotchas

- Where to get help and how you can participate in the Chapel community

ONE DAY CHAPEL TUTORIAL

- 9-10:30: Getting started using Chapel for parallel programming
- 10:30-10:45: break
- 10:45-12:15: Chapel basics in the context of the n-body example code
- 12:15-1:15: lunch
- 1:15-2:45: Distributed and shared-memory parallelism especially w/arrays (data parallelism)
- 2:45-3:00: break
- 3:00-4:30: More parallelism including for asynchronous parallelism (task parallelism)
- 4:30-5:00: Wrap-up including gathering further questions from attendees

CHAPEL RESOURCES

Chapel homepage: https://chapel-lang.org

- (points to all other resources)

Social Media:

- Twitter: @ChapelLanguage
- Facebook: @ChapelLanguage
- YouTube: http://www.youtube.com/c/ChapelParallelProgrammingLanguage

Community Discussion / Support:

- Discourse: https://chapel.discourse.group/
- Gitter: https://gitter.im/chapel-lang/chapel
- Stack Overflow: https://stackoverflow.com/questions/tagged/chapel
- GitHub Issues: https://github.com/chapel-lang/chapel/issues