PARALLELISM IN CHAPEL, PART I

Chapel Team, edited by Michelle Strout

April 10, 2025

PLAN

- **Announcements**
  - SA7 is due Friday April 11th
  - Final projects are due Friday May 2nd (3 weeks left)
- **Last time**
  - TopHat question about graduation
  - Chapel programming basics in the context of an Nbody simulation, part II
- **Today**
  - TopHat questions about Chapel basics
  - Data parallelism in Chapel
  - Domain decomposition in Chapel

OUTLINE: OVERVIEW OF PROGRAMMING 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

RECALL PROCESSING FILES IN PARALLEL

RECALL: ANALYZING MULTIPLE FILES USING PARALLELISM

parfilekmer.chpl

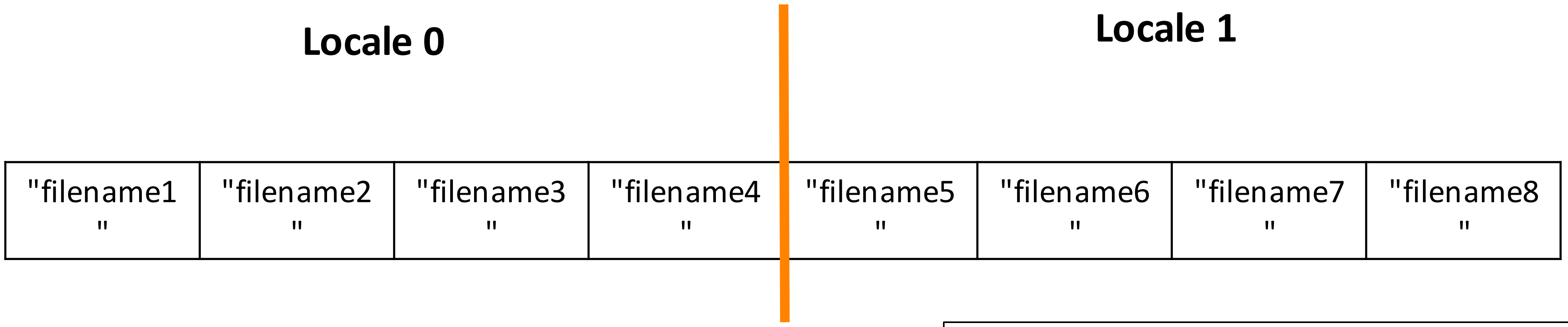
parfilekmer.chpl

```
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



Locale 0

Locale 1

"filename1"	"filename2"	"filename3"	"filename4"	"filename5"	"filename6"	"filename7"	"filename8"

```
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

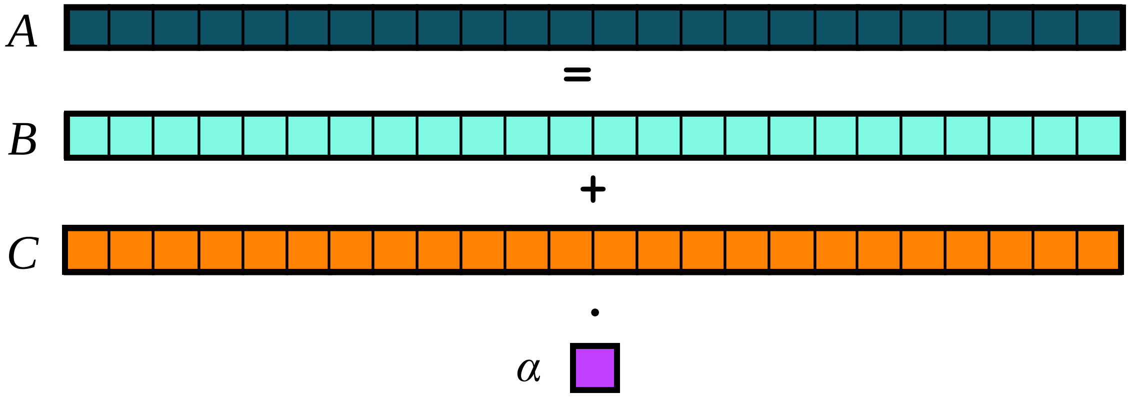
DATA PARALLELISM CONCEPTS AND EXAMPLES INCLUDING MULTI-LOCALE PARALLELISM WITH DISTRIBUTIONS

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:

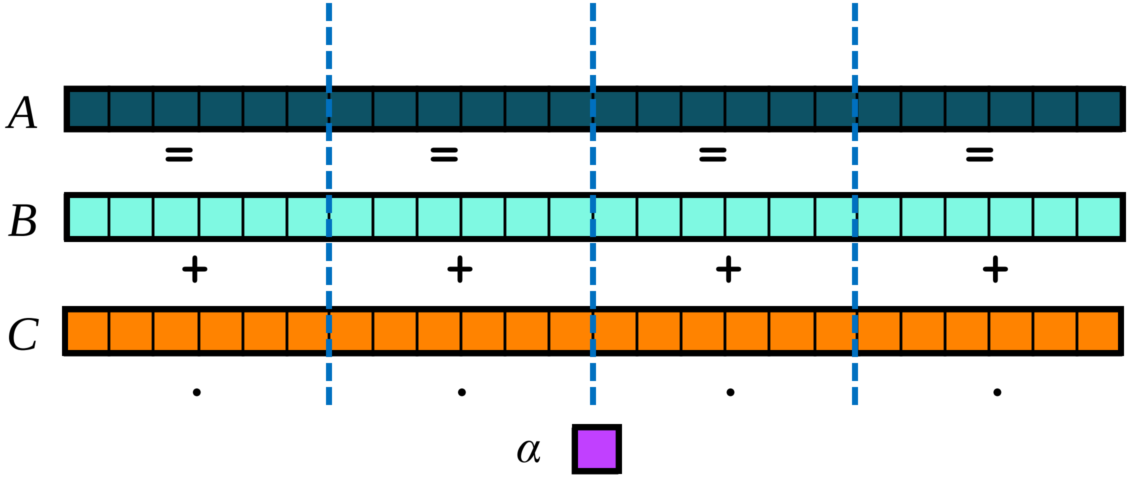


STREAM TRIAD: A PARALLEL COMPUTATION

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):

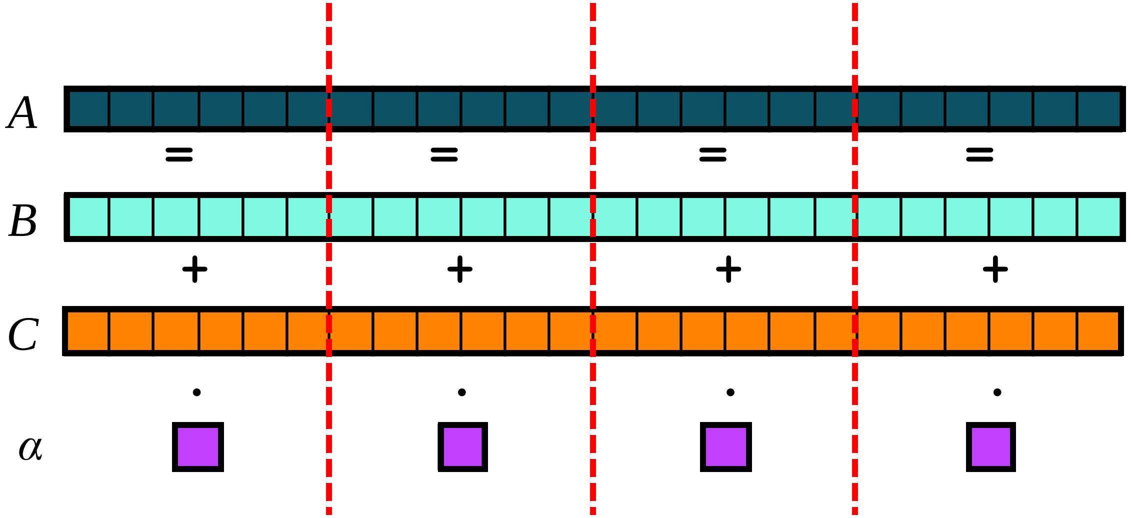


STREAM TRIAD: A PARALLEL COMPUTATION

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):

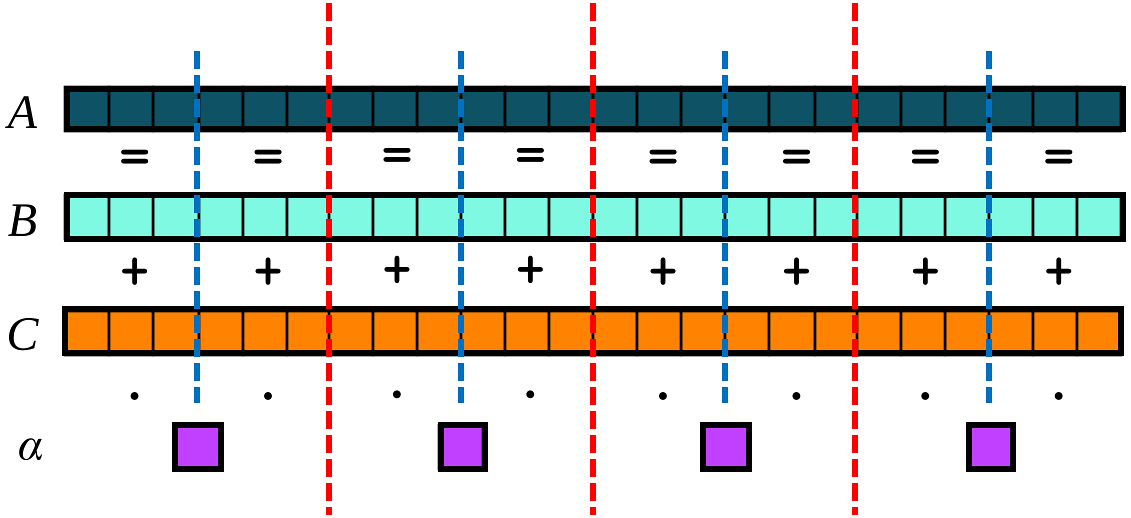


STREAM TRIAD: A PARALLEL COMPUTATION

Given: m-element vectors A, B, C

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

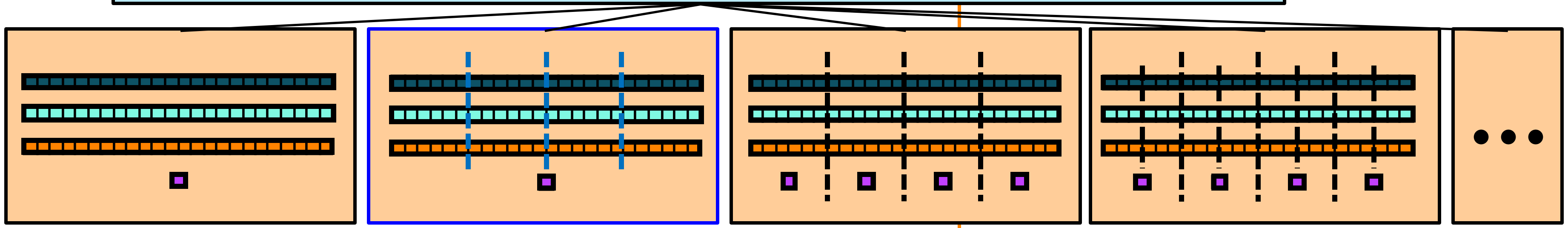
In pictures, in parallel (distributed memory multicore):



STREAM TRIAD: CHAPEL (SEE TOPHAT QUESTION)

```
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

Question: What happens when you remove the ”with (ref A)”?

```
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
```

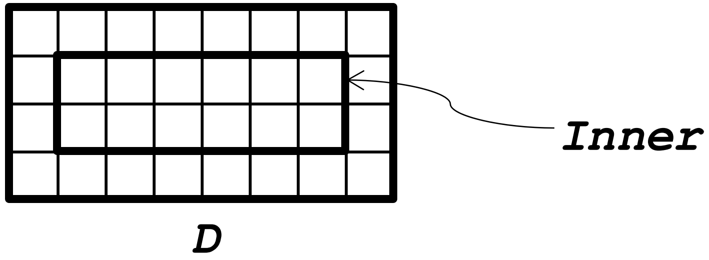
DOMAINS

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};
```

Question: What do ‘D’ and ‘Inner’ look like when you print them out?

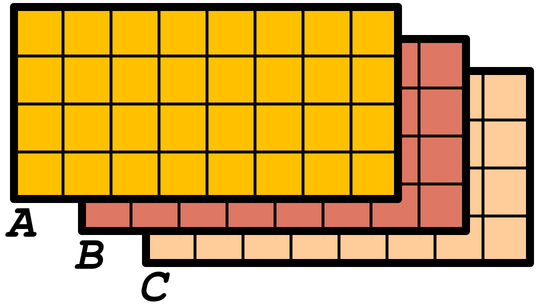


DOMAINS

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 (TOPHAT QUESTION)

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

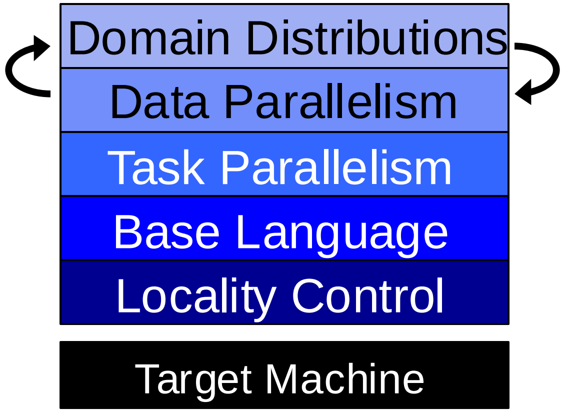
- Lowering of code is well-defined
- User can control details
- Part of Chapel‘s *multiresolution philosophy…*

```
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
- Round robin 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…
```

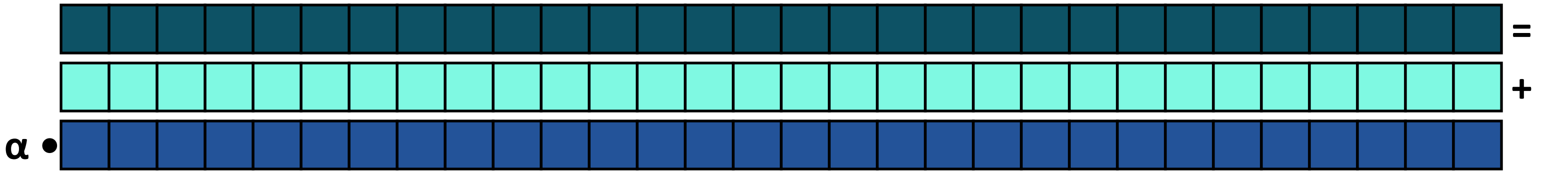
Or use a different domain distribution:

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

USING A DIFFERENT DOMAIN DISTRIBUTION

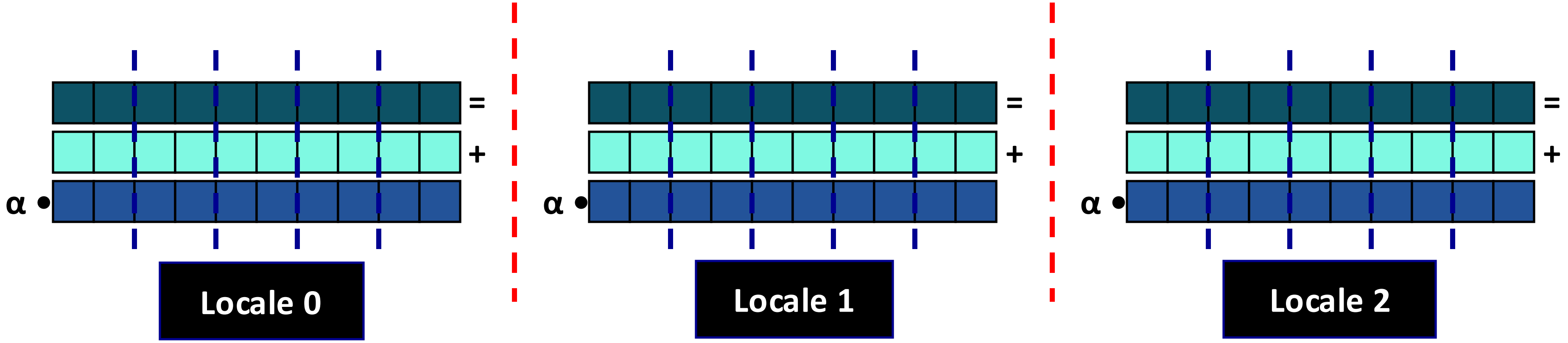
DOMAIN DISTRIBUTIONS: A MULTIRESOLUTION FEATURE

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



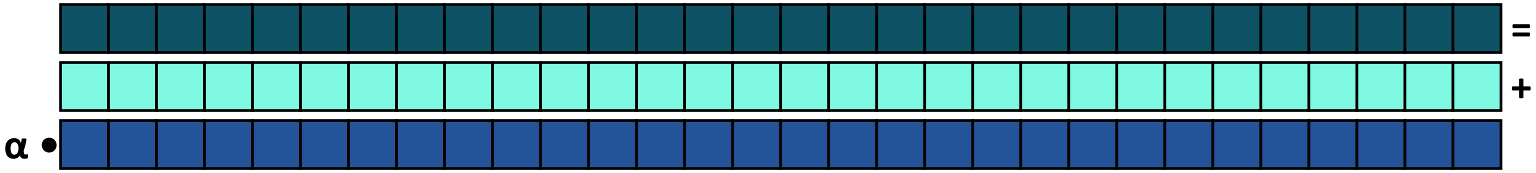
```
A = B + alpha * C;
```

…to the target locales' memory and processors:



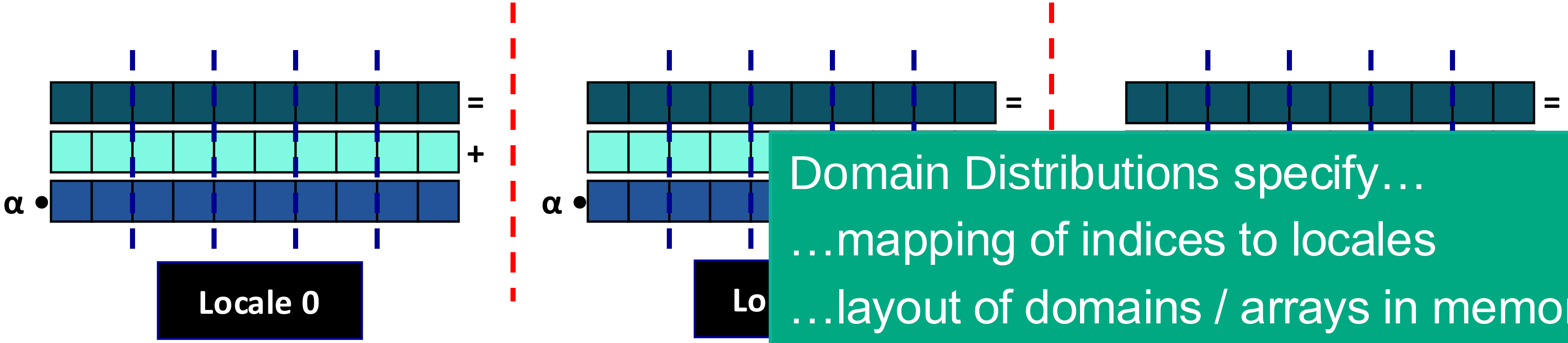
DOMAIN DISTRIBUTIONS: A MULTIRESOLUTION FEATURE

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



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

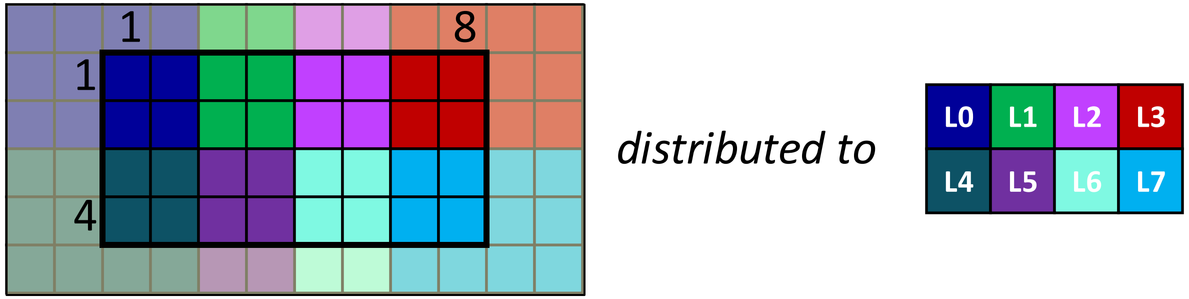
…to the target locales' memory and processors:



Domain Distributions specify…
…mapping of indices to locales
…layout of domains / arrays in memory
…parallel iteration strategies
…core operations on arrays / domains

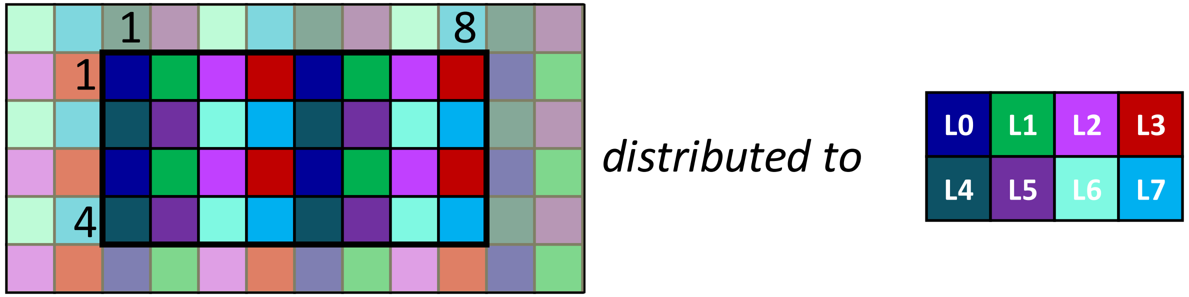
SAMPLE DOMAIN DISTRIBUTIONS: BLOCK AND CYCLIC (TOPHAT QUESTION)

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



distributed to

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



distributed to