Chapel: Locality and Affinity

Brad Chamberlain

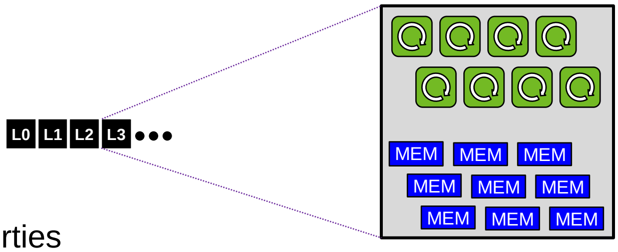
PRACE Winter School
12 February 2009

Outline

- Basics of Multi-Locale Chapel
  - The **`locale`** type and **`Locales`** array
  - The **`on`** statement, **`here`** locale, and communication
  - The **`local`** block
- Domain and Array Distributions
- Sample Uses of Distributed Domains/Arrays

The `locale` Type

- Definition
  - An abstract unit of the target architecture
  - Supported to permit reasoning about locality
  - Has capacity for processing and storage



- Properties
  - Threads within a locale have ~uniform access to local memory
  - Memory within other locales is accessible, but at a price
  - Locales are defined for a given architecture by a Chapel compiler
    - e.g., a multicore processor or SMP node could be a locale

Locales and Program Startup

- Chapel users specify # locales on executable command-line

```
prompt> myChapelProg -nl=8      # run using 8 locales
```

- Chapel launcher bootstraps program execution:
  - obtains necessary machine resources
    - *e.g.*, requests 8 nodes from the job scheduler
  - loads a copy of the executable onto the machine resources
  - starts running the program. Conceptually…

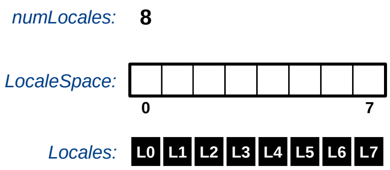
    …locale #0 starts running program's entry point (`main()`)

    …other locales wait for work to arrive

Locale Variables

Built-in variables represent a program's set of locales:

```
config const numLocales: int;             // number of locales
const LocaleSpace = [0..numLocales-1],   // locale indices
      Locales: [LocaleSpace] locale;     // locale values
```

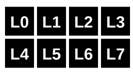


Locale Views

Using standard array operations, users can create their own locale views:

```
var TaskALocs = Locales[..numTaskALocs];
var TaskBLocs = Locales[numTaskALocs+1..];
```


```
var CompGrid = Locales.reshape([1..gridRows,
                                1..gridCols]);
```



Locale Methods

- The locale type supports built-in methods:

```
def locale.id: int;                      // index in LocaleSpace
def locale.name: string;                 // similar to uname -n
def locale.numCores: int;                // # of processor cores
def locale.physicalMemory(…): …;         // amount of memory
…
```


Executing on Remote Locales

- Syntax

```
on-stmt:
  on expr { stmt }
```

- Semantics
  - Executes *stmt* on the locale specified by *expr*
  - Does not introduce concurrency
- Example

```
var A: [LocaleSpace] int;
coforall loc in Locales do on loc {
  A(loc.id) = computation(loc.id);
}
```

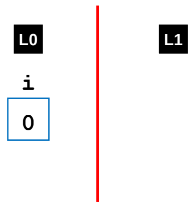
Querying a variable's locale

- Syntax

```
locale-query-expr:
  var-expr . locale
```

- Semantics
  - Returns the locale on which *var-expr* is allocated
- Example

```
var i: int;
write(i.locale.id);
on Locales(1) do
  write(i.locale.id);
```



- Output

```
00
```


Serial on-clause example

on clauses: indicate where code should execute

```
// Chapel programs begin running on locale 0 by default

var x, y: real;           // allocate x & y on locale 0

on Locales(1) {           // migrate task to locale 1
  var z: real;            // allocate z on locale 1

  writeln(x.locale.id);   // prints "0"
  writeln(z.locale.id);   // prints "1"

  z = x + y;              // requires "get" for x and y

  on Locales(0) do        // migrate back to locale 0
    z = x + y;            // requires "get" for z
                          // return to locale 1
}                         // return to locale 0
```

Serial on-clause example (data-driven)

on clauses: indicate where code should execute

```
// Chapel programs begin running on locale 0 by default

var x, y: real;            // allocate x & y on locale 0

on Locales(1) {            // migrate task to locale 1
  var z: real;             // allocate z on locale 1

  writeln(x.locale.id);    // prints "0"
  writeln(z.locale.id);    // prints "1"

  z = x + y;               //          t" for x and y

  on ~~Locales(0)~~ x do       // migrate back to locale 0
    z = x + y;             // requires "get" for z
                           // return to locale 1
}                          // return to locale 0
```

optionally, in a data-driven manner

Parallel on-clause examples

on clauses: indicate where code should execute

By naming locales explicitly…

```
cobegin {
  on TaskALocs  do computeTaskA(…);   L0 L1               computeTaskA()
  on TaskBLocs  do computeTaskB(…);   L2 L3 L4 L5 L6 L7   computeTaskB()
  on Locales[0] do computeTaskC(…);   L0                  computeTaskC()
}
```

…or in a data-driven manner…

```
computePivot(data, lo, hi);
cobegin {
  on data[lo]    do Quicksort(data, lo, pivot);
  on data[pivot] do Quicksort(data, pivot, hi);
}
```

Here

- Built-in locale function

```
def here: locale;
```

- Semantics
  - Returns the locale on which the task is executing
- Example

```
writeln(here.id);
on Locales(1) do
  writeln(here.id);
```

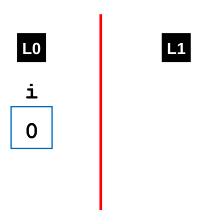
- Output

```
0
1
```


Remote Reads and Writes

- Example

```
var i = 0;
on Locales(1) {
  writeln((here.id, i.locale.id, i));
  i = 1;
  writeln((here.id, i.locale.id, i));
}
writeln((here.id, i.locale.id, i));
```



- Output

```
(1, 0, 0)
(1, 0, 1)
(0, 0, 1)
```

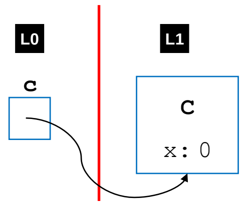
Remote Classes

- Example

```
class C {
  var x: int;
}

var c: C;
on Locales(1) do c = new C();

writeln((here.id, c.locale.id, c));
```



- Output

```
(0, 1, {x = 0})
```


Local Blocks

- Syntax

```
local-stmt:
  local stmt
```

- Semantics
  - Asserts there are no remote references in *stmt*
  - Checked at runtime by default; can be disabled for performance
- Example

```
c = Root.child(1);
on c do local {
  traverseTree(c);
}
```

```
local {
  A[D] = B[D];
}
```

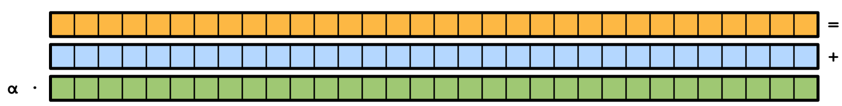
Outline

- Basics of Multi-Locale Chapel
- Domain and Array Distributions
  - overview
  - a case study: Block1D
- Sample Uses of Distributed Domains/Arrays

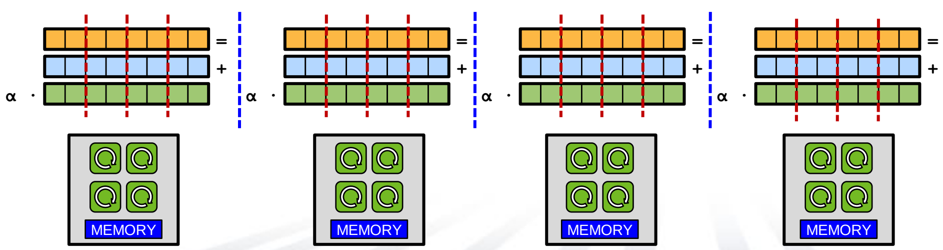
Chapel Distributions

Distributions: "Recipes for parallel, distributed arrays"

- help the compiler map from the computation's global view…



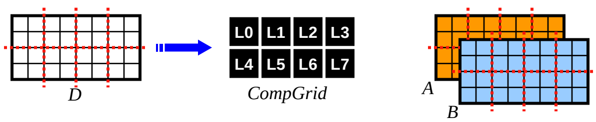
…down to the *fragmented*, per-processor implementation



Domain Distribution

Domains may be distributed across locales

```
var D: domain(2) distributed Block on CompGrid = …;
```



A distribution implies…

…ownership of the domain's indices (and its arrays' elements)

…the default work ownership for operations on the domains/arrays

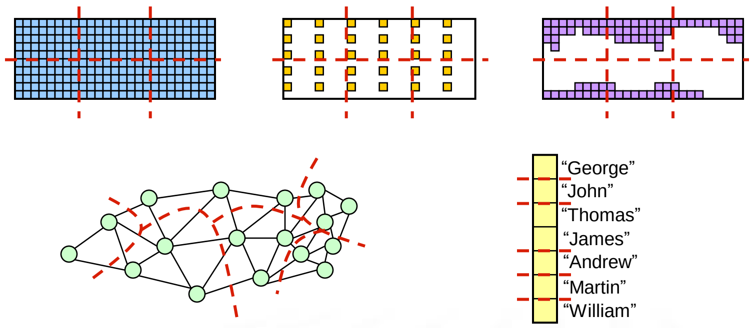
- e.g., forall loops or promoted operations over domains/arrays

Authoring Distributions

- (Advanced) Programmers can write distributions in Chapel
- Chapel will support a standard library of distributions
  - *research goal:* using the same mechanism that users would
  - our compiler should have no knowledge of specific distributions
  - only its structural interface—how to…
    - …create domains and arrays using that distribution
    - …map indices to locales
    - …access array elements
    - …iterate over indices/array elements
      - sequentially
      - in parallel
      - in parallel and zippered with other parallel iteratable types
    - …and so forth…
- Distributions are built using the concepts we've already seen
  - on clauses for expressing what data & tasks each locale owns
  - begins, cobegins, coforalls to express inter- & intra-locale parallelism

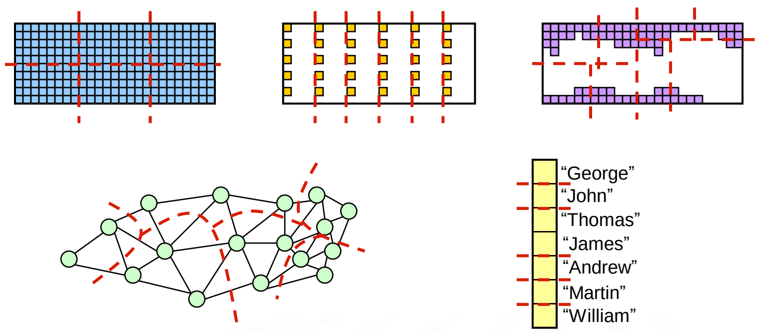
Distributions

- All the domain types we've seen will support distributions
- Domain/array semantics are independent of distribution
  - performance and parallelism may vary greatly as distributions change



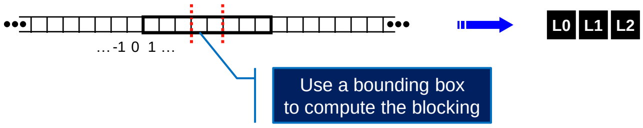
Distributions

- All the domain types we've seen will support distributions
- Domain/array semantics are independent of distribution
  - performance and parallelism may vary greatly as distributions change



A Simple Distribution: Block1D

- **Goal:** block a 1D index space across a set of locales



Distributions vs. Domains

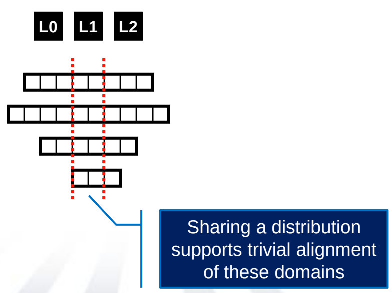
Q1: Why distinguish between distributions and domains?

Q2: Why do distributions map an index *space* rather than a fixed index set?

A: To permit several domains to share a single distribution

- amortizes the overheads of storing a distribution
- supports trivial domain/array alignment and compiler optimizations

```
const D      : …distributed B1 = [1..8],
      outerD: …distributed B1 = [0..9],
      innerD: subdomain(D)     = [2..7],
      slideD: subdomain(D)     = [4..6];
```



Distributions vs. Domains

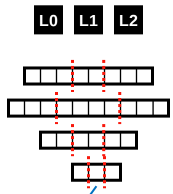
Q1: Why distinguish between distributions and domains?

Q2: Why do distributions map an index *space* rather than a fixed index set?

A: To permit several domains to share a single distribution

- amortizes the overheads of storing a distribution
- supports trivial domain/array alignment and compiler optimizations

```
const D      : …distributed B1 = [1..8],
      outerD: …distributed B2 = [0..9],
      innerD: …distributed B3 = [2..7],
      slideD: …distributed B4 = [4..6];
```

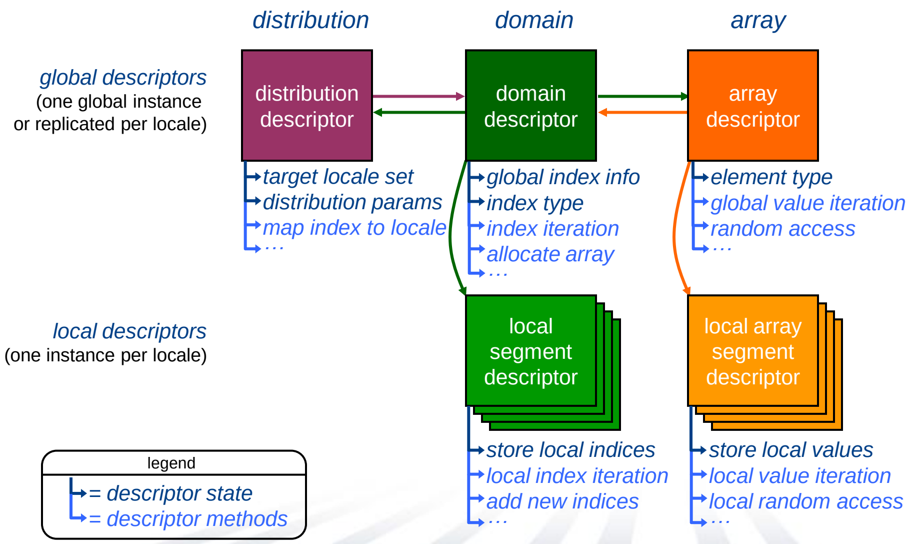


When each domain is given its own distribution, the alignment between indices is less obvious.

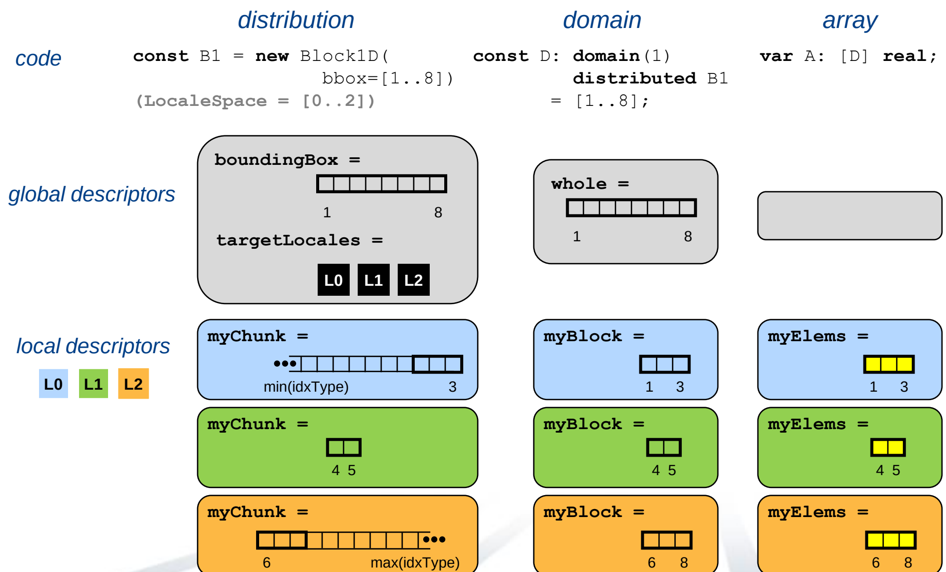
Chapel's Distribution Architecture

	distribution	*domain*	*array*
global descriptors (one global instance or replicated per locale)	**Responsibility:** Mapping of indices to locales	**Responsibility:** How to store, iterate over domain indices	**Responsibility:** How to store, access, iterate over array elements
local descriptors (one instance per locale)		**Responsibility:** How to store, iterate over *local* domain indices	**Responsibility:** How to store, access, iterate over *local* array elements

Chapel's Distribution Architecture

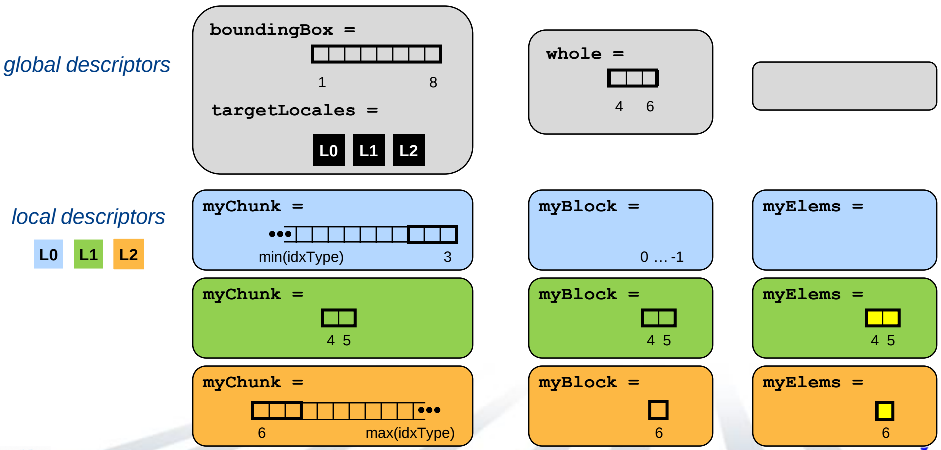


Block1D Distribution Classes



Block1D Distribution Classes

	distribution	*domain*	*array*
code	`const B1 = new Block1D( bbox=[1..8]) (LocaleSpace = [0..2])`	`const sliceD: domain(1) distributed B1 = [4..6];`	`var A2: [sliceD] real;`



Outline

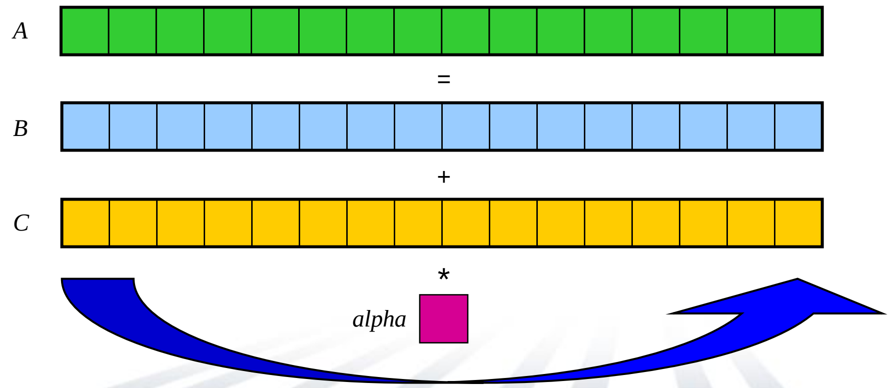
- Basics of Multi-Locale Chapel
- Domain and Array Distributions
- Sample Uses of Distributed Domains/Arrays
  - HPCC Stream Triad
  - HPCC Random Access (RA)

Introduction to STREAM Triad

Given: m-element vectors A, B, C

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

Pictorially:

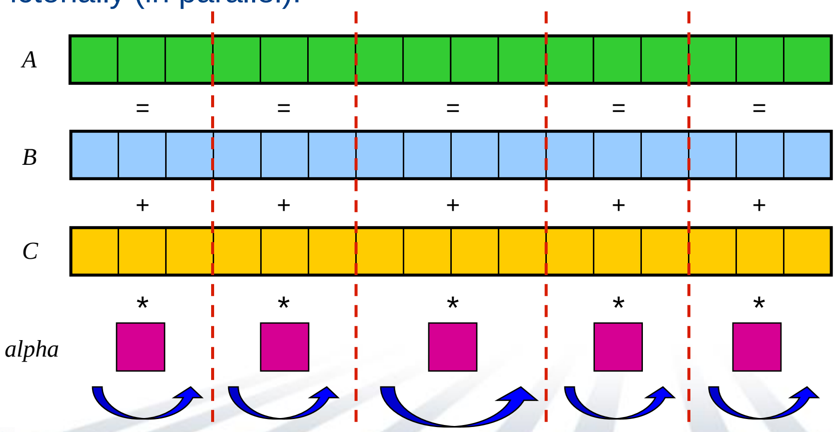


Introduction to STREAM Triad

Given: m-element vectors A, B, C

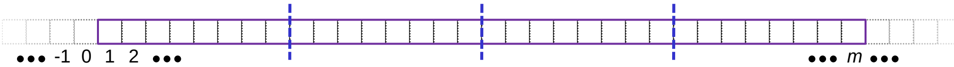
Compute: $\forall i \in 1..m$, $A_i = B_i + \alpha \cdot C_i$

Pictorially (in parallel):

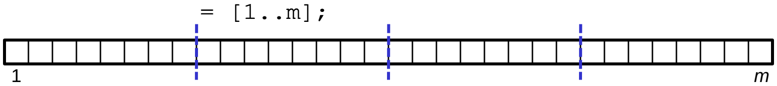


STREAM Triad in Chapel

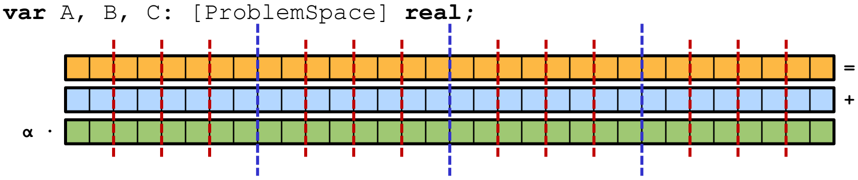
```
const BlockDist = new Block1D(bbox=[1..m], tasksPerLocale=…);
```




```
const ProblemSpace: domain(1, int(64)) distributed BlockDist
                  = [1..m];
```




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



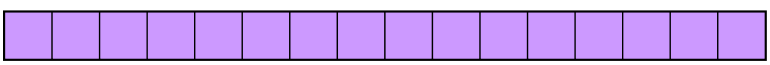

```
forall (a, b, c) in (A, B, C) do
  a = b + alpha * c;
```


Introduction to Random Access

Given: m-element table T (where $m = 2^n$ and initially $T_i = i$)

Compute: N_U random updates to the table using bitwise-xor

Pictorially:

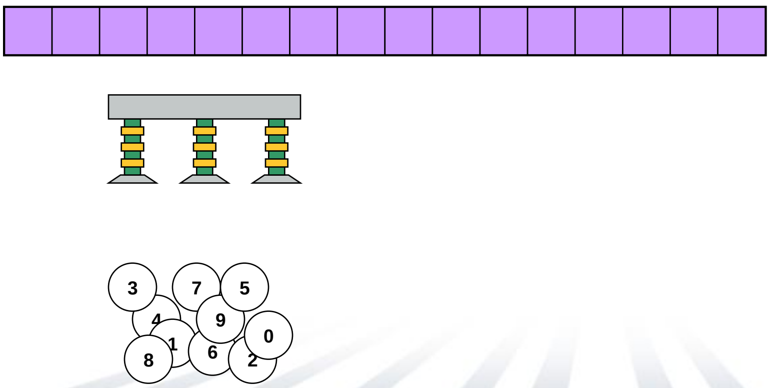


Introduction to Random Access

Given: m-element table T (where $m = 2^n$ and initially $T_i = i$)

Compute: N_U random updates to the table using bitwise-xor

Pictorially:

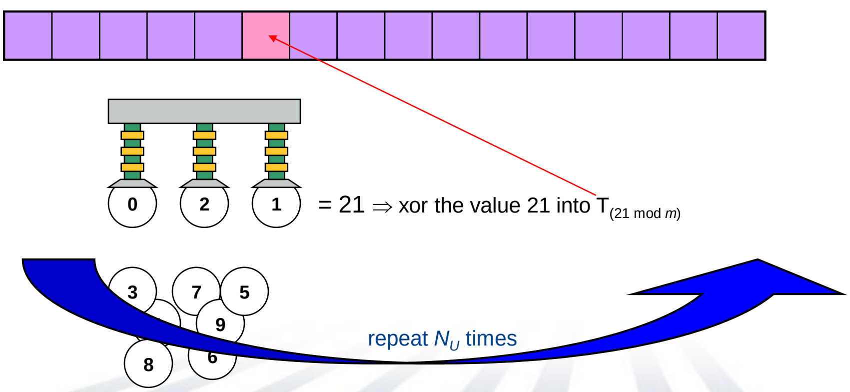


Introduction to Random Access

Given: m-element table T (where $m = 2^n$ and initially $T_i = i$)

Compute: N_U random updates to the table using bitwise-xor

Pictorially:

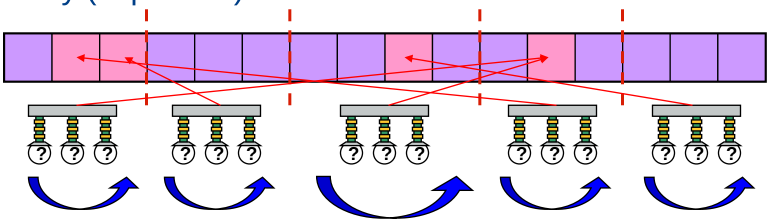


Introduction to Random Access

Given: m-element table T (where $m = 2^n$ and initially $T_i = i$)

Compute: N_U random updates to the table using bitwise-xor

Pictorially (in parallel):

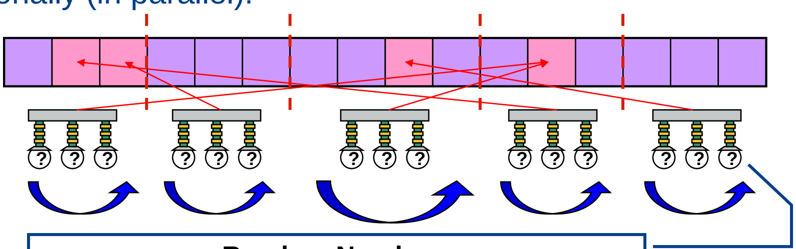


Introduction to Random Access

Given: m-element table T (where $m = 2^n$ and initially $T_i = i$)

Compute: N_U random updates to the table using bitwise-xor

Pictorially (in parallel):



Random Numbers

Not actually generated using lotto ping-pong balls!

Instead, implement a pseudo-random stream:

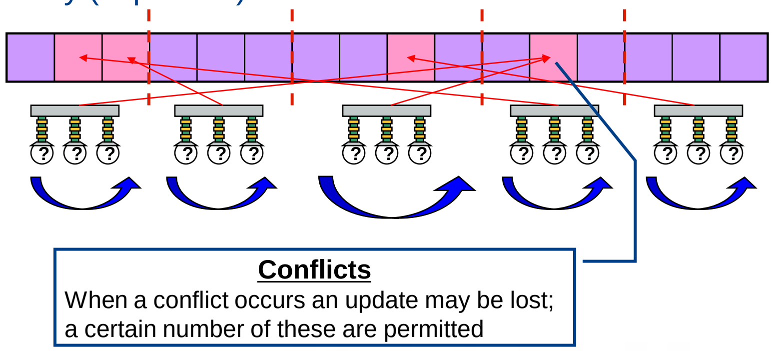
- *k*th random value can be generated at some cost
- given the *k*th random value, can generate the (*k*+1)-st much more cheaply

Introduction to Random Access

Given: m-element table T (where $m = 2^n$ and initially $T_i = i$)

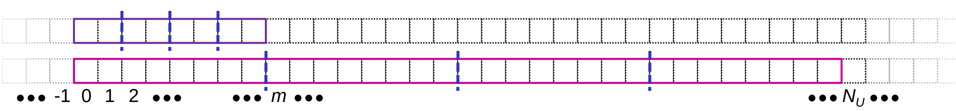
Compute: N_U random updates to the table using bitwise-xor

Pictorially (in parallel):

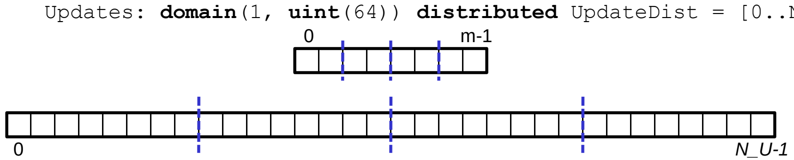


RA Declarations in Chapel

```
const TableDist  = new Block1D(bbox=[0..m-1], tasksPerLocale=…),
      UpdateDist = new Block1D(bbox=[0..N_U-1], tasksPerLocale=…);
```




```
const TableSpace: domain(1, uint(64)) distributed TableDist = [0..m-1],
      Updates: domain(1, uint(64)) distributed UpdateDist = [0..N_U-1];
```



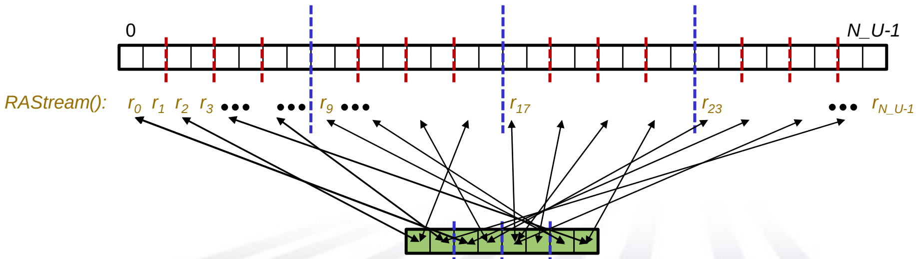

```
var T: [TableSpace] uint(64);
```

RA Computation in Chapel

```
const TableSpace: domain(1, uint(64)) distributed TableDist = [0..m-1],
      Updates: domain(1, uint(64)) distributed UpdateDist = [0..N_U-1];

var T: [TableSpace] uint(64);

forall (_, r) in (Updates, RAStream()) do
  on T(r&indexMask) do
    T(r&indexMask) ^= r;
```

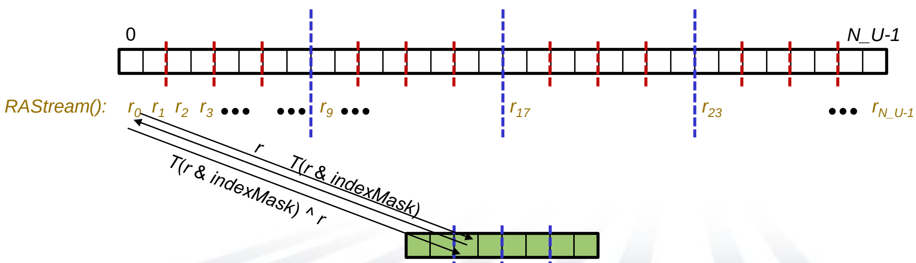


RA Computation in Chapel

```
const TableSpace: domain(1, uint(64)) distributed TableDist = [0..m-1],
      Updates: domain(1, uint(64)) distributed UpdateDist = [0..N_U-1];

var T: [TableSpace] uint(64);

forall (_, r) in (Updates, RAStream()) do
  T(r&indexMask) ^= r;
```

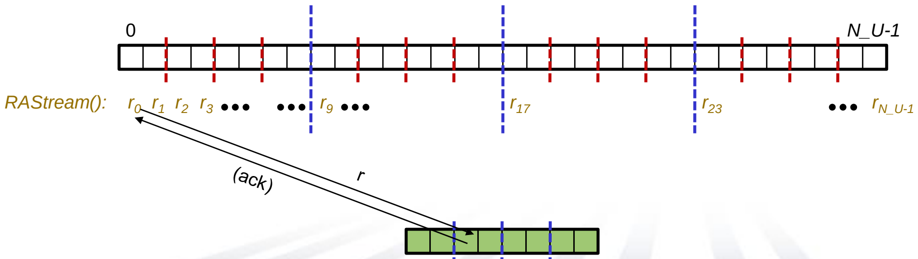


RA Computation in Chapel: tune for affinity

```
const TableSpace: domain(1, uint(64)) distributed TableDist = [0..m-1],
      Updates: domain(1, uint(64)) distributed UpdateDist = [0..N_U-1];

var T: [TableSpace] uint(64);


forall (_, r) in (Updates, RAStream()) do
  on T(r&indexMask) do
    T(r&indexMask) ^= r;
```

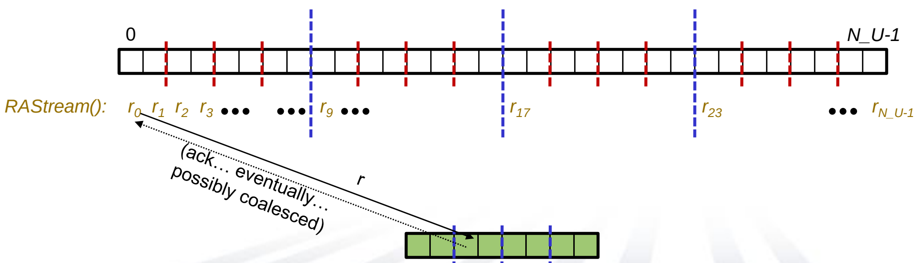


RA Computation in Chapel: fire and forget

```
const TableSpace: domain(1, uint(64)) distributed TableDist = [0..m-1],
      Updates: domain(1, uint(64)) distributed UpdateDist = [0..N_U-1];

var T: [TableSpace] uint(64);

sync {
  forall (_, r) in (Updates, RAStream()) do
    on T(r&indexMask) do
      begin T(r&indexMask) ^= r;                 }
```



Locality and Affinity Status

- Stable Features:
  - locale types, methods, and variables
  - on clauses
- Incomplete Features:
  - the local block has not been stress-tested
  - we've only just started getting our first distributions working
    - this is the reason that foralls/promotions don't result in parallelism
    - only Block1D and only for basic domain/array operations
      - see examples/hpcc/stream.chpl and ra.chpl for sample uses
- Future Directions:
  - improved support for replicated, symmetric data
  - distributions as a mechanism for software resiliency
  - richer locale types: multiple flavors and hierarchical locales
    - to better represent machine structure and heterogeneity

Questions?