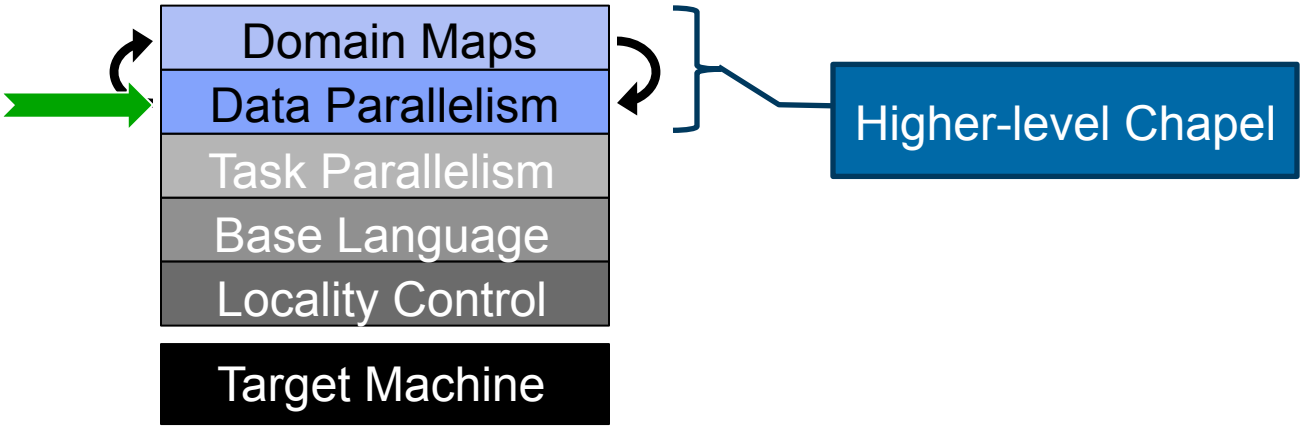
Data Parallelism, By Example

SC14
New Orleans, LA | hpc matters.

Safe Harbor Statement

This presentation may contain forward-looking statements that are based on our current expectations. Forward looking statements may include statements about our financial guidance and expected operating results, our opportunities and future potential, our product development and new product introduction plans, our ability to expand and penetrate our addressable markets and other statements that are not historical facts. These statements are only predictions and actual results may materially vary from those projected. Please refer to Cray's documents filed with the SEC from time to time concerning factors that could affect the Company and these forward-looking statements.

Data Parallel Features



Data Parallel "Hello, world!"

```
config const numIters = 1000;

forall i in 1..numIters do
  writeln("Hello, world! ",
          "from iteration ", i, " of ", numIters);
```

```
Hello, world! from iteration 500 of 1000
Hello, world! from iteration 501 of 1000
Hello, world! from iteration 502 of 1000
Hello, world! from iteration 503 of 1000
Hello, world! from iteration 1 of 1000
Hello, world! from iteration 2 of 1000
Hello, world! from iteration 3 of 1000
Hello, world! from iteration 4 of 1000
…
```

Defining our Terms

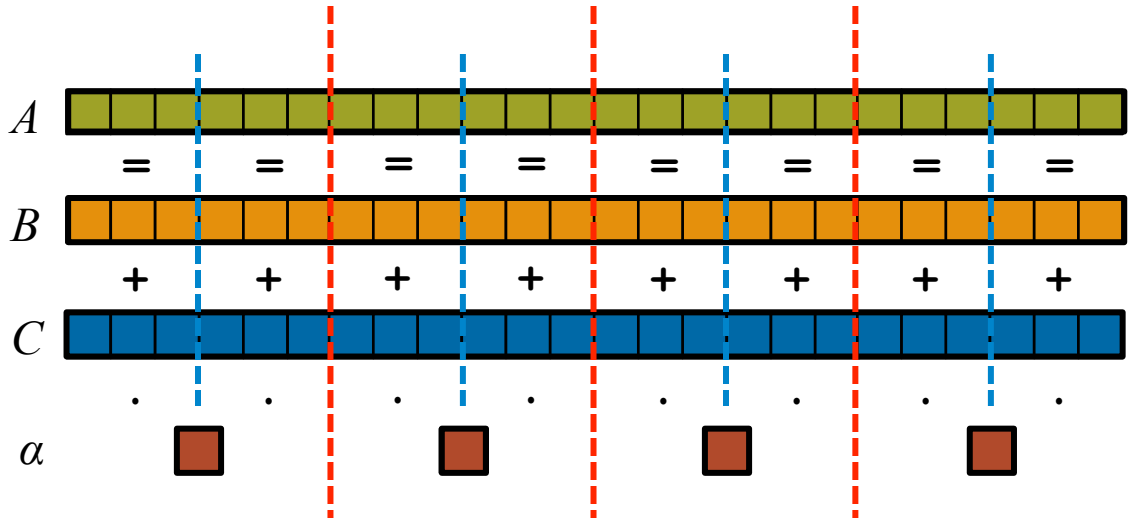
Data Parallelism: a style of parallel programming in which parallelism is driven by computations over collections of data elements or their indices

STREAM Triad: a trivial parallel computation

Given: m-element vectors A, B, C

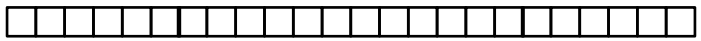
Compute: $\forall i \in 1..m,\ A_i = B_i + \alpha \cdot C_i$

In pictures, in parallel (distributed memory multicore):

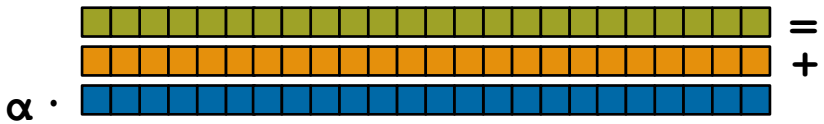


STREAM Triad in Chapel

```
const ProblemSpace = {1..m};
```



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



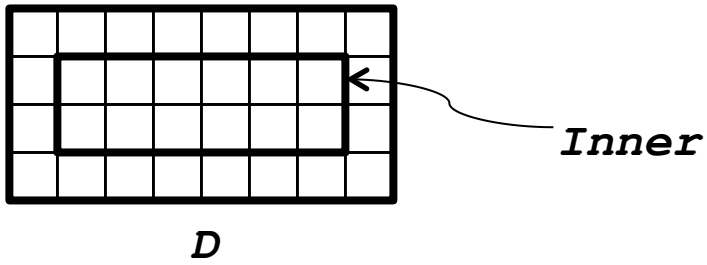

```
A = B + alpha * C;
```

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};
```

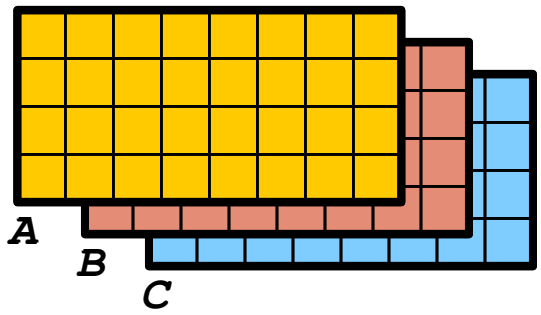


Domains

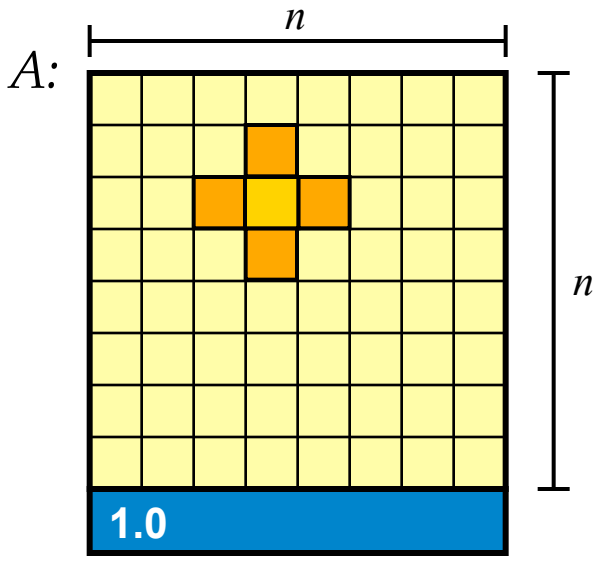
Domain:

- 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: Jacobi Iteration



repeat until max change < ε

Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
          D = BigD[1..n, 1..n],
   LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do {
  forall (i,j) in D do
    Temp[i,j] = (A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1]) / 4;

  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
          D = BigD[1..n, 1..n],
   LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;

A[La

do {
  fo
                                    ]) / 4;
  co
  A[
} wh

writ
```

Declare program parameters

const ⇒ can't change values after initialization

config ⇒ can be set on executable command-line

```
prompt> jacobi --n=10000 --epsilon=0.0001
```

note that no types are given; they're inferred from initializers

n ⇒ **default integer** (64 bits)

epsilon ⇒ **default real floating-point** (64 bits)

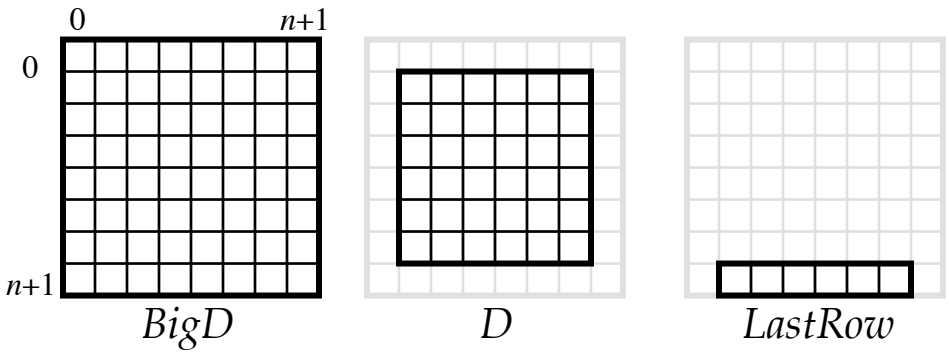
Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
          D = BigD[1..n, 1..n],
   LastRow = D.exterior(1,0);
```

Declare domains (first class index sets)

{lo..hi, lo2..hi2} ⇒ 2D rectangular domain, with 2-tuple indices

Dom1[Dom2] ⇒ computes the intersection of two domains



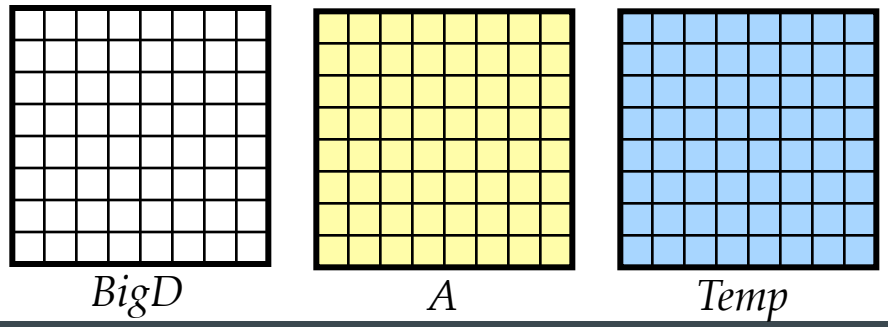
.exterior() ⇒ one of several built-in domain generators

Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
          D = BigD[1..n, 1..n],
   LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;
```

Declare arrays

var ⇒ can be modified throughout its lifetime
: [*Dom*] T ⇒ array of size *Dom* with elements of type *T*
(no initializer) ⇒ values initialized to default value (0.0 for reals)

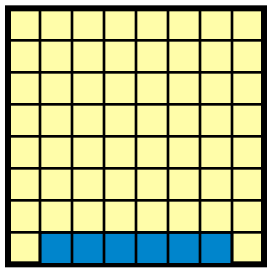


Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
          D = BigD[1..n, 1..n],
   LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do

                                        j+1]) / 4;

}

wr
```

Set Explicit Boundary Condition

Arr[Dom] ⇒ refer to array slice (“forall i in Dom do …Arr[i]…”)



A

Jacobi Iteration in Chapel

```
config const n = 6,
```

Compute 5-point stencil

forall ***ind*** **in** ***Dom*** ⇒ parallel forall expression over *Dom*'s indices, binding them to *ind*
(here, since *Dom* is 2D, we can de-tuple the indices)

```
do {
  forall (i,j) in D do
    Temp[i,j] = (A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1]) / 4;

  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;
```

Compute maximum change

***op* reduce** ⇒ collapse aggregate expression to scalar using *op*

Promotion: *abs()* and – are scalar operators; providing array operands results in parallel evaluation equivalent to:

```
forall (a,t) in zip(A,Temp) do abs(a - t)
```

```
do {
  forall (i,j) in D do
    Temp[i,j] = (A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1]) / 4;

  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
      D = BigD[1..n, 1..n],
```

Copy data back & Repeat until done

uses slicing and whole array assignment
standard *do…while* loop construct

```
do {
  forall (i,j) in D do
    Temp[i,j] = (A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1]) / 4;

  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
          D = BigD[1..n, 1..n],
   LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do {
  forall (i,j) in D do
    Temp[i,j] = (A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1]) / 4;

  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

Write array to console

Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
          D = BigD[1..n, 1..n],
   LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do {
  forall (i,j) in D do
    Temp[i,j] = (A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1]) / 4;

  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

Chapel Data Parallel Operations

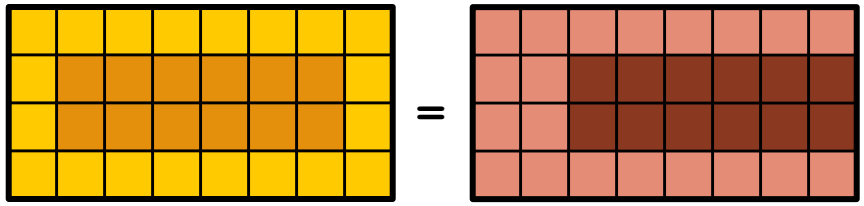
- **Data Parallel Iteration**

```
forall (i,j) in D 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

- **Array Slicing; Domain Algebra**

```
A[InnerD] = B[InnerD+(0,1)];
```



- **Promotion of Scalar Functions and Operators**

```
A = exp(B, C);
```

```
A = foo("hi", B, C);
```

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

- **And many others: reductions, scans, reallocation, reshaping, remapping, set operations, aliasing, …**

Tiled Row-Major Order Iterator

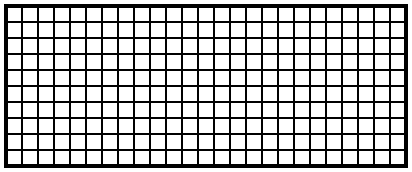
```
//
// iterate over domain D using tilesize x tilesize tiles in row-major order
//
iter tiledRMO(D, tilesize) {
  const tile = {0..#tilesize, 0..#tilesize};

  for base in D by tilesize do
    for ij in D[tile.translate(base)] do
      yield ij;
}
```

```
for ij in tiledRMO({1..m, 1..n}, 2) do
  write(ij);
```

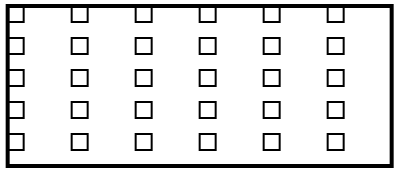
Output:

```
(1,1)(1,2)(2,1)(2,2)(1,3)(1,4)(2,3)(2,4)…
(3,1)(3,2)(4,1)(4,2)(3,3)(3,4)(4,3)(4,4)…
```

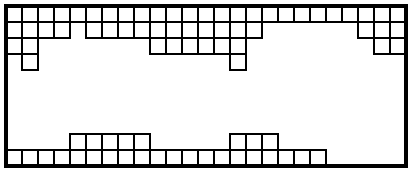
Chapel Domain Types



dense

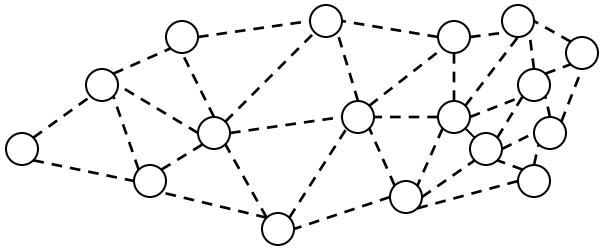


strided



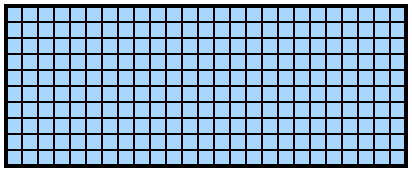
sparse

associative

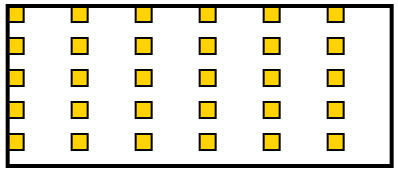


unstructured

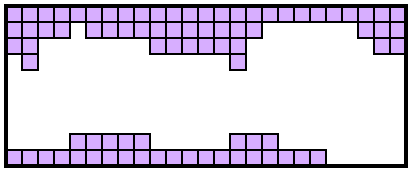
Chapel Array Types



dense

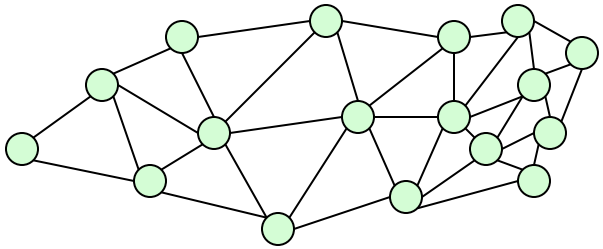


strided



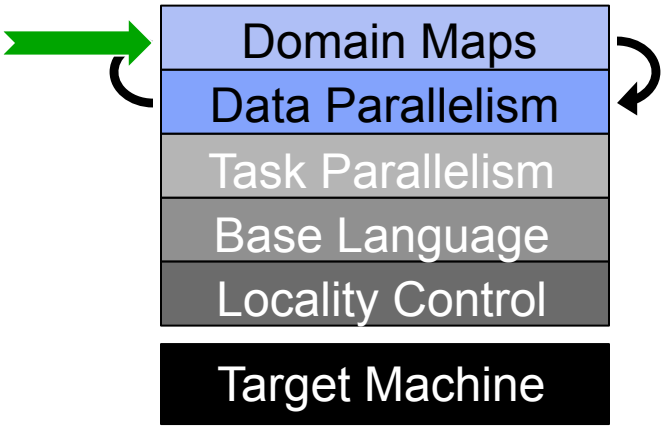
sparse

associative



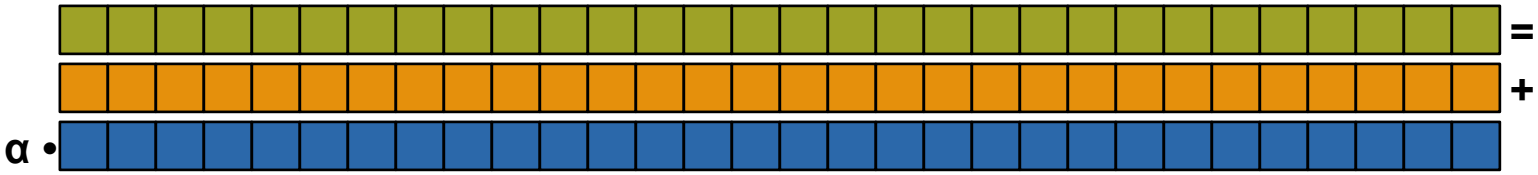
unstructured

Domain Maps



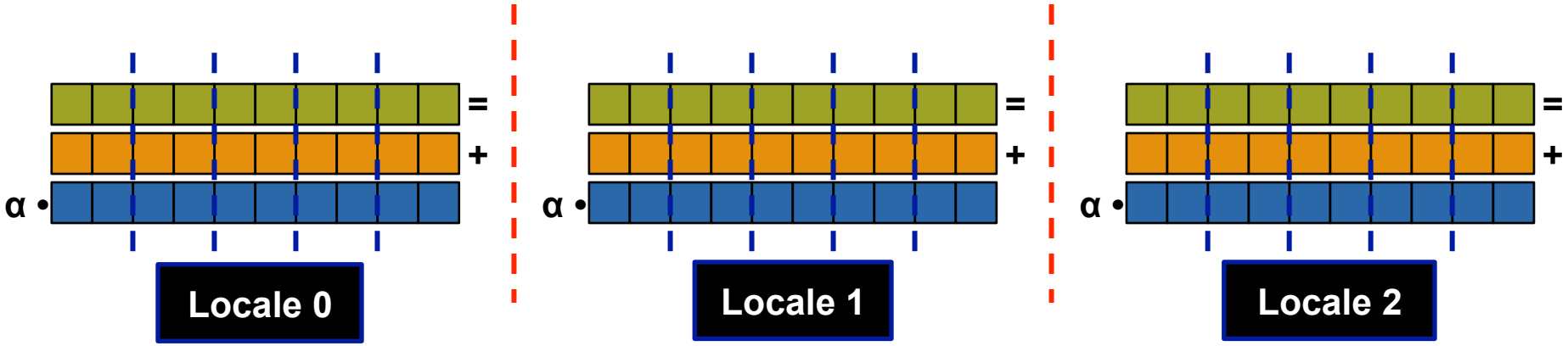
Domain Maps

Domain maps 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:



Shared Memory Data Parallel "Hello, world!"

```
config const numIters = 100000;

const D = {1..numIters};

forall i in D do
  writeln("Hello, world! ",
          "from iteration ", i, " of ", numIters);
```

Distributed Memory Data Parallel "Hello, world!"

```
config const numIters = 100000;

const D = {1..numIters} dmapped Cyclic(startIdx=1);

forall i in D do
  writeln("Hello, world! ",
          "from iteration ", i, " of ", numIters);
```

Layouts and Distributions

Domain Maps fall into two major categories:

layouts:

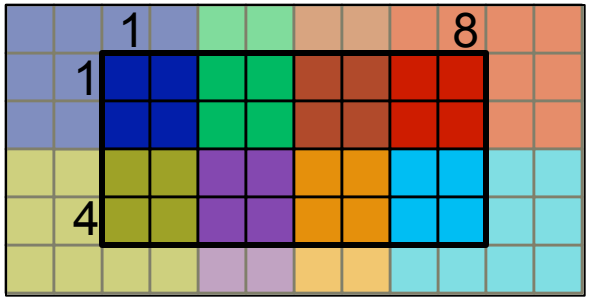
- target a shared memory
- **examples:** row- and column-major order, tilings, compressed sparse row, space-filling curves

distributions:

- map indices/elements to distributed memories
- **examples:** Block, Cyclic, Block-Cyclic, Recursive Bisection, …

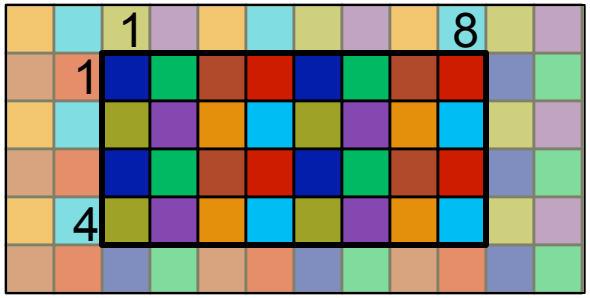
Sample Distributions: Block and Cyclic

```
var Dom = {1..4, 1..8} dmapped Block( {1..4, 1..8} );
```



distributed to

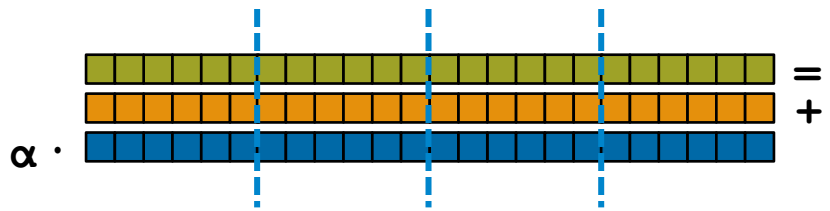

```
var Dom = {1..4, 1..8} dmapped Cyclic( startIdx=(1,1) );
```



distributed to

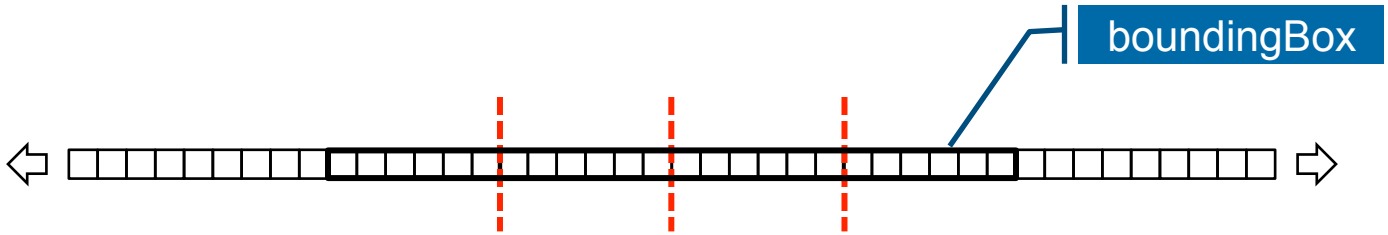
STREAM Triad: Chapel (multicore)

```
const ProblemSpace = {1..m};

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

A = B + alpha * C;
```

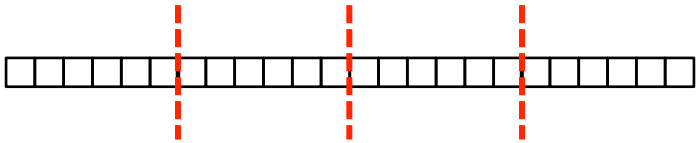


No domain map specified => use default layout
- current locale owns all domain indices and array values
- computation will execute using local processors only

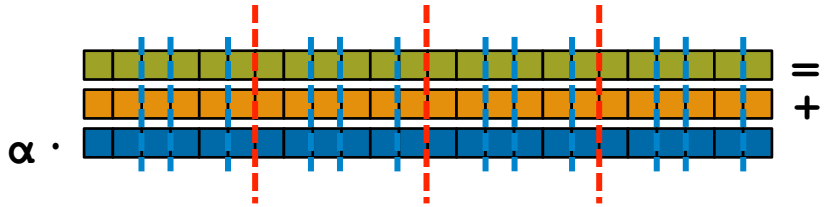
STREAM Triad: Chapel (distributed, blocked)




```
const ProblemSpace = {1..m}
                     dmapped Block(boundingBox={1..m});
```

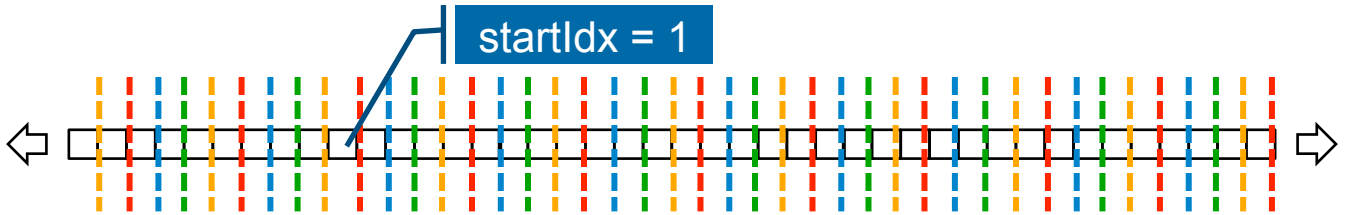


```
var A, B, C: [ProblemSpace] real;
```

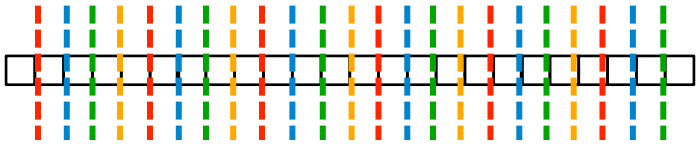



```
A = B + alpha * C;
```

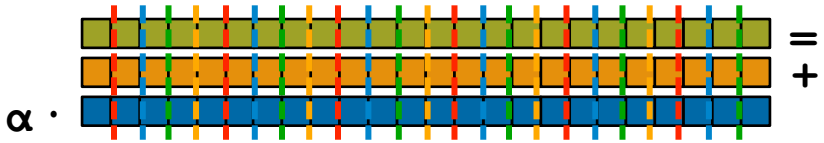
STREAM Triad: Chapel (distributed, cyclic)




```
const ProblemSpace = {1..m}
                       dmapped Cyclic(startIdx=1);
```



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




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

Jacobi Iteration in Chapel (shared memory)

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
          D = BigD[1..n, 1..n],
   LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do {
  forall (i,j) in D do
    Temp[i,j] = (A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1]) / 4;

  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

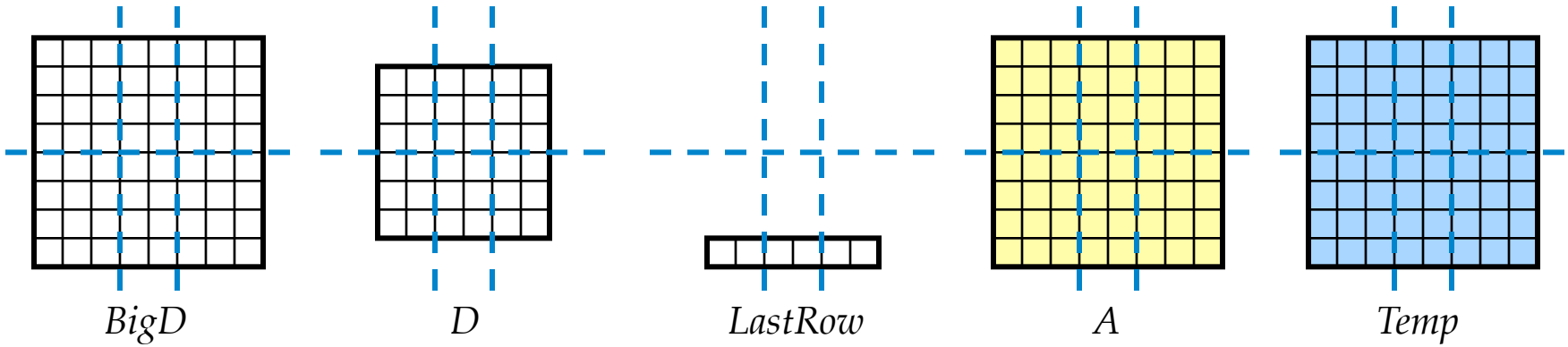
Jacobi Iteration in Chapel (distributed memory)

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1} dmapped Block({1..n, 1..n}),
          D = BigD[1..n, 1..n],
   LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;
```

With this simple change, we specify a mapping from the domains and arrays to locales

Domain maps describe the mapping of domain indices and array elements to *locales*

- specifies how array data is distributed across locales
- specifies how iterations over domains/arrays are mapped to locales



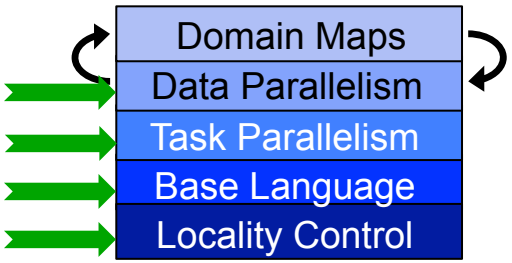
Jacobi Iteration in Chapel (distributed memory)

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1} dmapped Block({1..n, 1..n}),
          D = BigD[1..n, 1..n],
    LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do {
  forall (i,j) in D do
    Temp[i,j] = (A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1]) / 4;

  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);

use BlockDist;
```

Chapel's Domain Map Philosophy

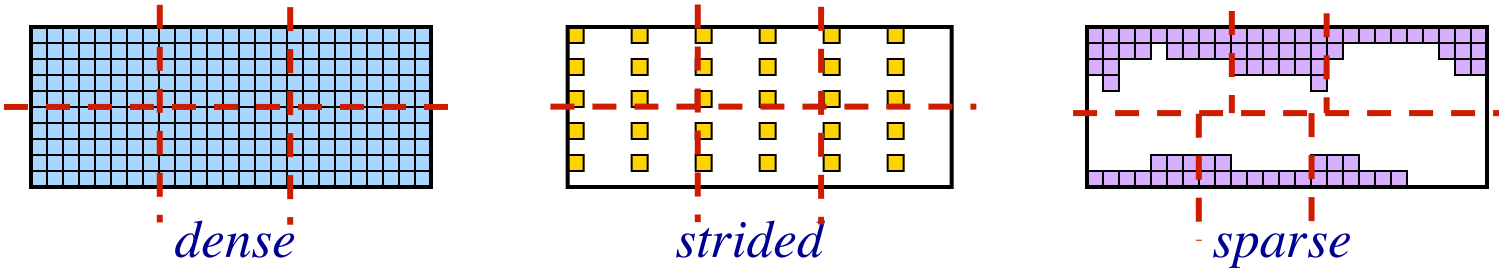
1. **Chapel provides a library of standard domain maps**
   - to support common array implementations effortlessly

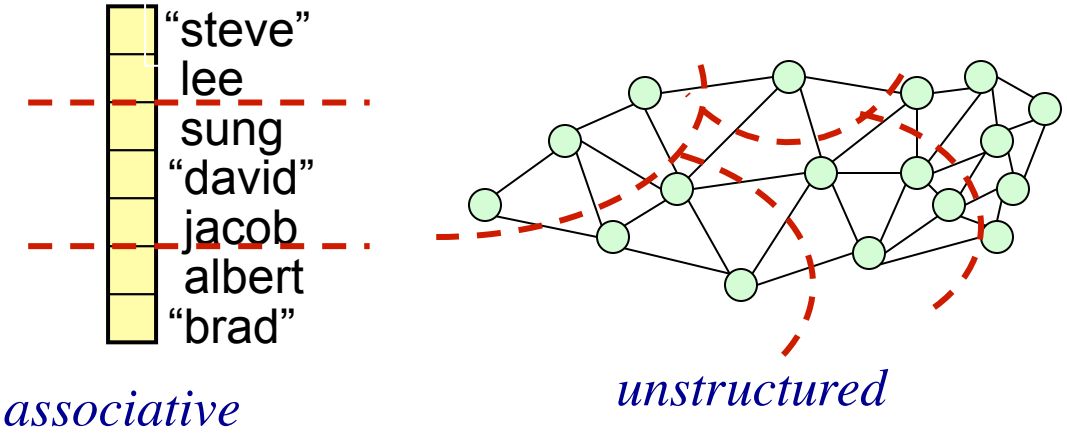
2. **Expert users can write their own domain maps in Chapel**
   - to cope with any shortcomings in our standard library



3. **Chapel's standard domain maps are written using the same end-user framework**
   - to avoid a performance cliff between "built-in" and user-defined cases

All Domain Types Support Domain Maps





Domain Map Descriptor Classes

Domain Map

Represents: a domain map value

Generic w.r.t.: index type

State: the domain map's representation

Typical Size: Θ(1)

Required Interface:

- create new domains

Domain

Represents: a domain

Generic w.r.t.: index type

State: representation of index set

Typical Size: Θ(1) → Θ(*numIndices*)

Required Interface:

- create new arrays
- queries: size, members
- iterators: serial, parallel
- domain assignment
- index set operations

Array

Represents: an array

Generic w.r.t.: index type, element type

State: array elements

Typical Size: Θ(*numIndices*)

Required Interface:

- (re-)allocation of elements
- random access
- iterators: serial, parallel
- slicing, reindexing, aliases
- get/set of sparse "zero" values

For More Information on Domain Maps

HotPAR'10: *User-Defined Distributions and Layouts in Chapel: Philosophy and Framework*
Chamberlain, Deitz, Iten, Choi; June 2010

CUG 2011: *Authoring User-Defined Domain Maps in Chapel*
Chamberlain, Choi, Deitz, Iten, Litvinov; May 2011

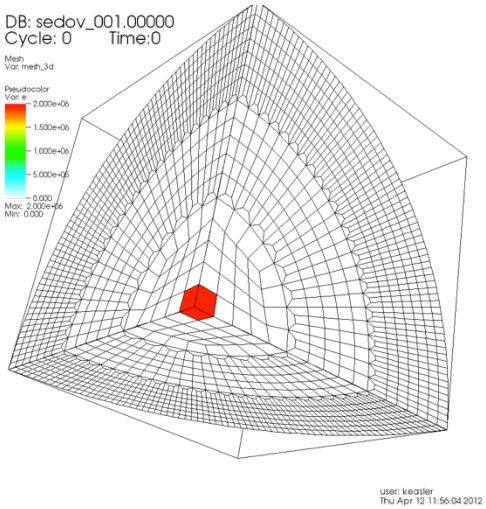
Chapel release:

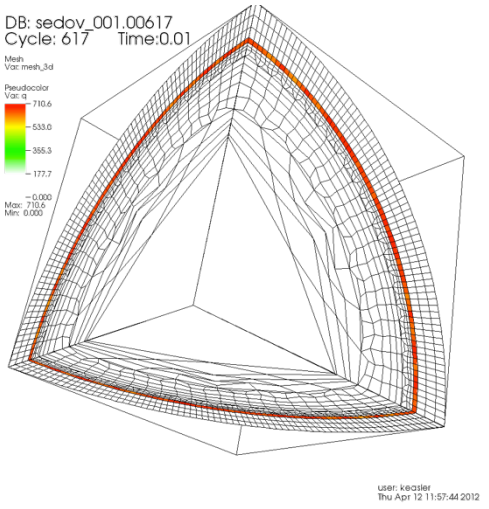
- Current domain maps:
  $CHPL_HOME/modules/dists/*.chpl
  layouts/*.chpl
  internal/Default*.chpl
- Technical notes detailing the domain map interface for implementers:
  $CHPL_HOME/doc/technotes/README.dsi

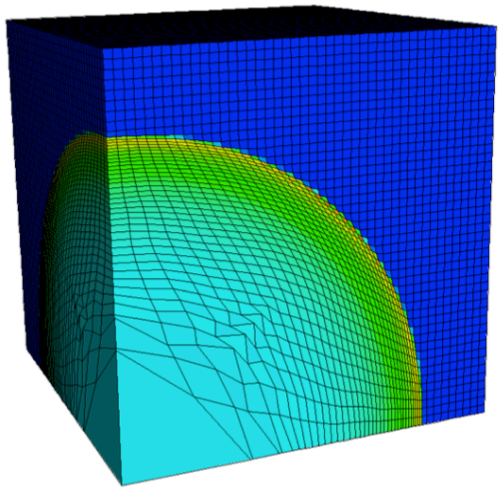
LULESH Case Study: Domain Maps in Action

LULESH: a DOE Proxy Application

Goal: Solve one octant of the spherical Sedov problem (blast wave) using Lagrangian hydrodynamics for a single material

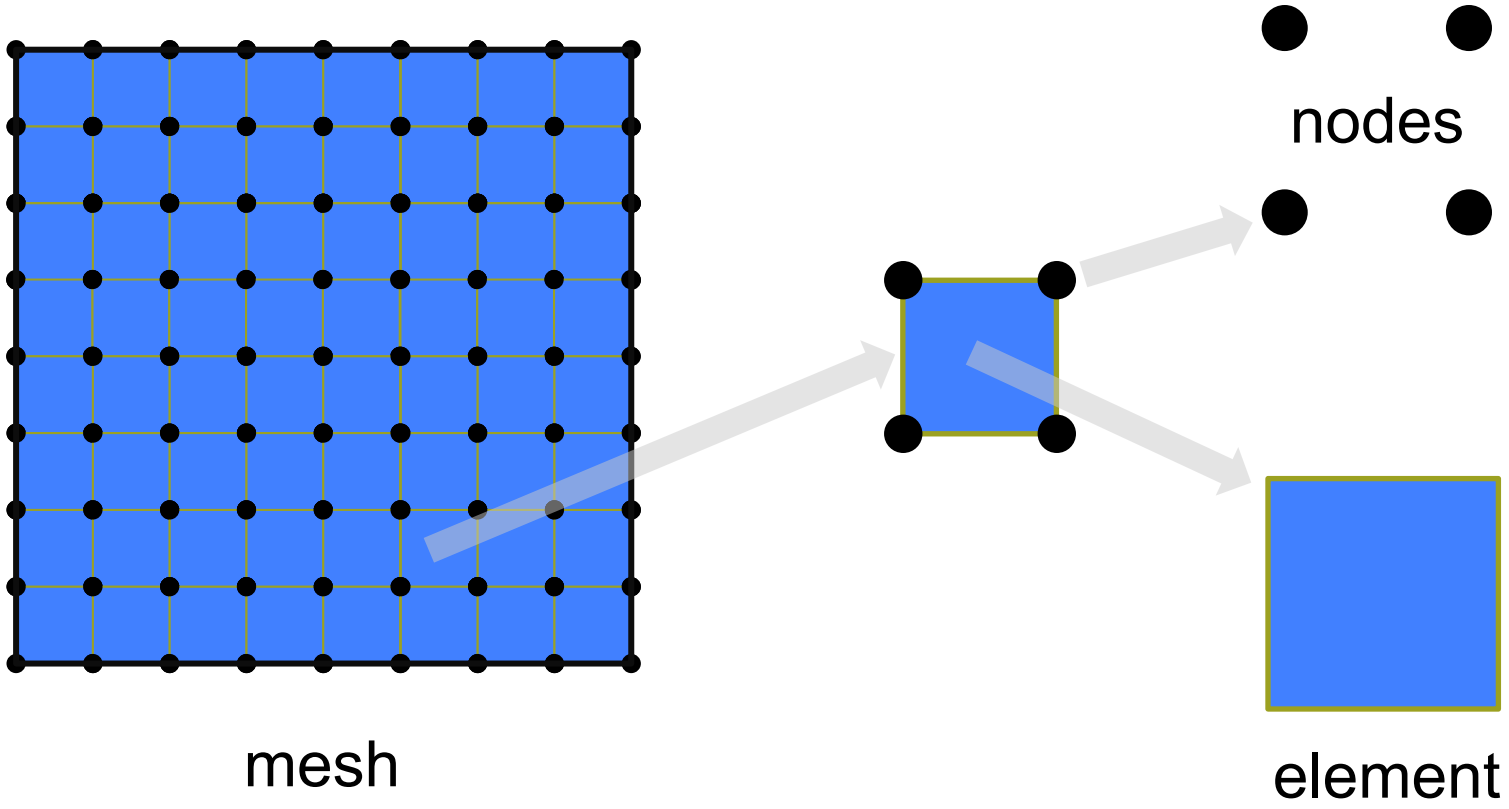






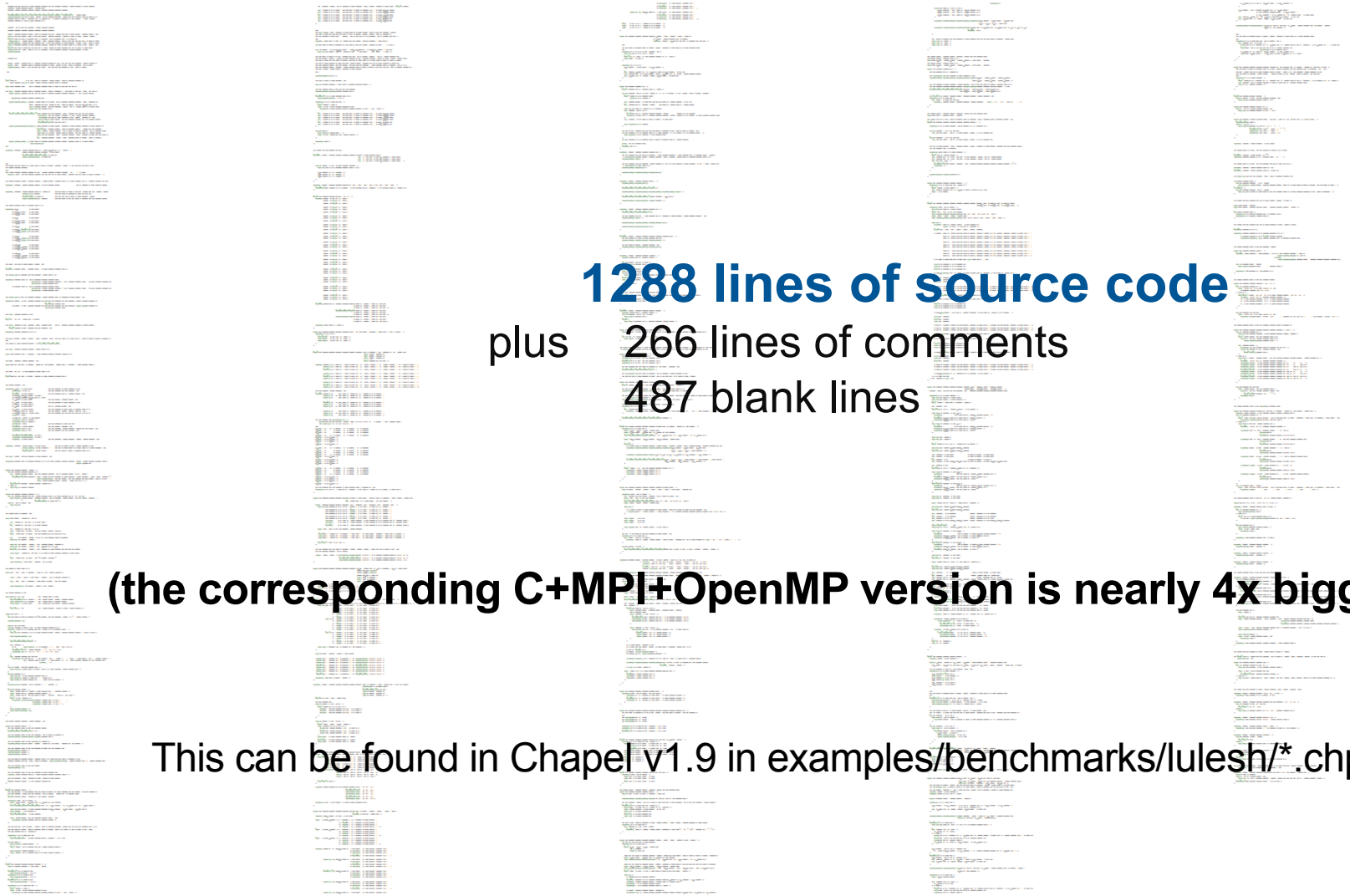
pictures courtesy of Rob Neely, Bert Still, Jeff Keasler, LLNL

Fundamental LULESH Concepts/Terminology



LULESH in Chapel

LULESH in Chapel



1288 lines of source code

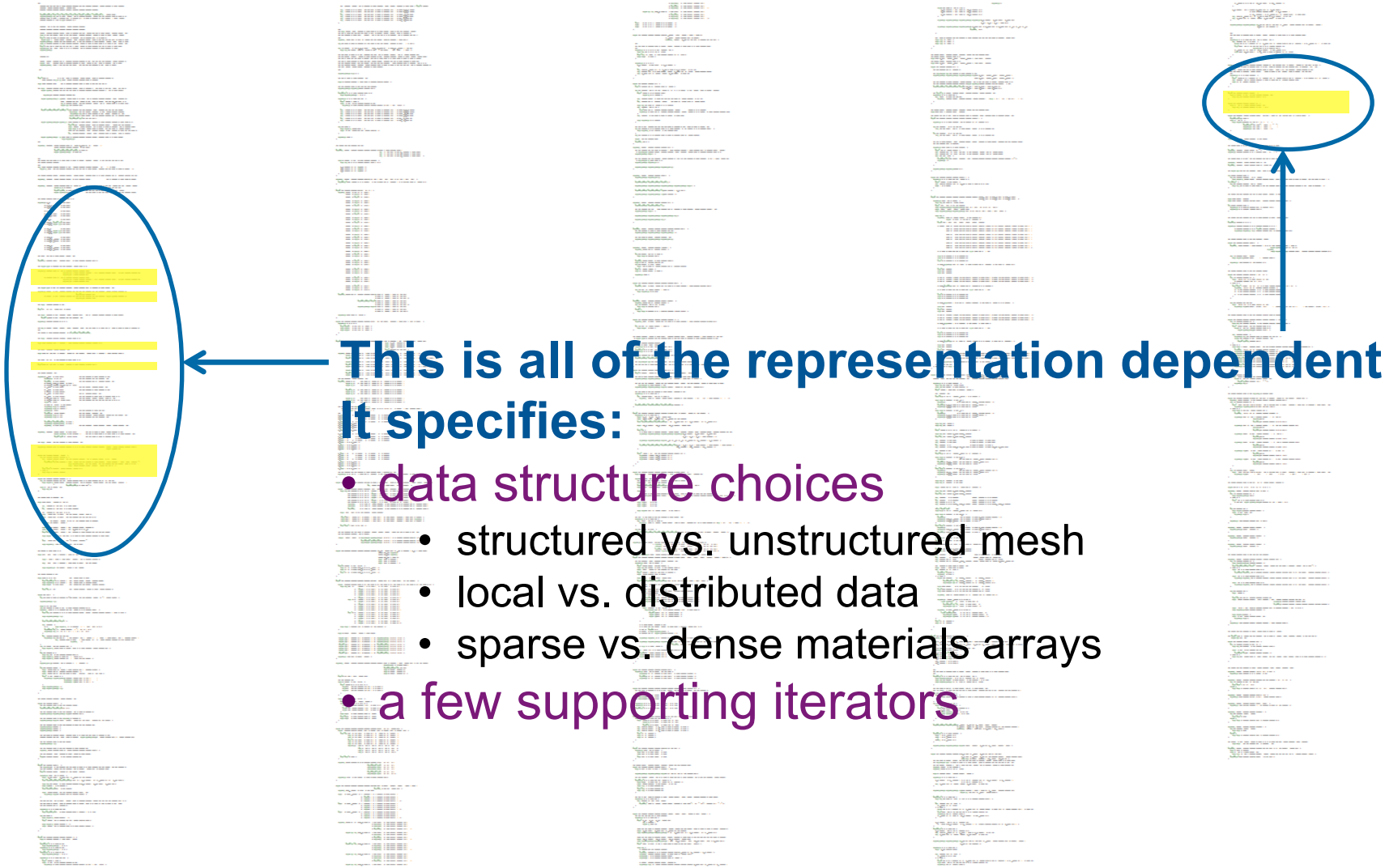
plus 266 lines of comments

487 blank lines

(the corresponding C+MPI+OpenMP version is nearly 4x bigger)

This can be found in Chapel v1.9 in examples/benchmarks/lulesh/*.chpl

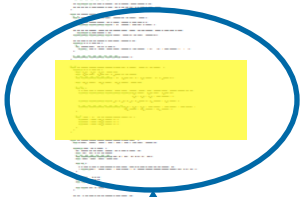
LULESH in Chapel



This is all of the representation dependent code. It specifies:

- data structure choices
  - structured vs. unstructured mesh
  - local vs. distributed data
  - sparse vs. dense materials arrays
- a few supporting iterators

LULESH in Chapel



Here is some sample representation-independent code

`IntegrateStressForElems()`

LULESH spec, section 1.5.1.1 (2.)

Representation-Independent Physics

```
proc IntegrateStressForElems(sigxx, sigyy, sigzz, determ) {
  forall k in Elems {
    var b_x, b_y, b_z: 8*real;
    var x_local, y_local, z_local: 8*real;
    localizeNeighborNodes(k, x, x_local, y, y_local, z, z_local);

    var fx_local, fy_local, fz_local: 8*real;

    local {
      /* Volume calculation involves extra work for numerical consistency. */
      CalcElemShapeFunctionDerivatives(x_local, y_local, z_local,
                                       b_x, b_y, b_z, determ[k]);

      CalcElemNodeNormals(b_x, b_y, b_z, x_local, y_local, z_local);

      SumElemStressesToNodeForces(b_x, b_y, b_z, sigxx[k], sigyy[k], sigzz[k],
                                  fx_local, fy_local, fz_local);
    }

    for (noi, t) in elemToNodesTuple(k) {
      fx[noi].add(fx_local[t]);
      fy[noi].add(fy_local[t]);
      fz[noi].add(fz_local[t]);
    }
  }
}
```

parallel loop over elements

collect nodes neighboring this element; localize node fields

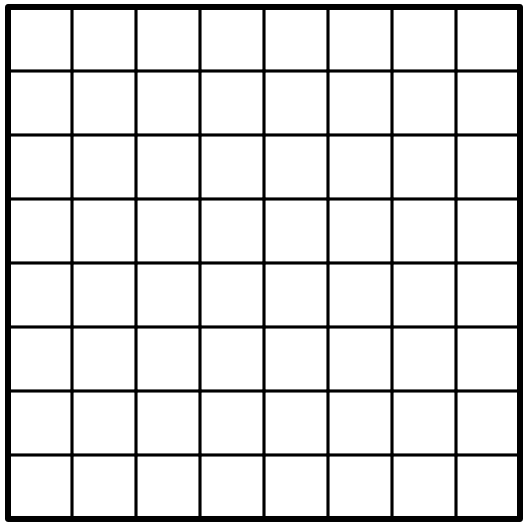
update node forces from element stresses

Because of domain maps, this code is independent of:

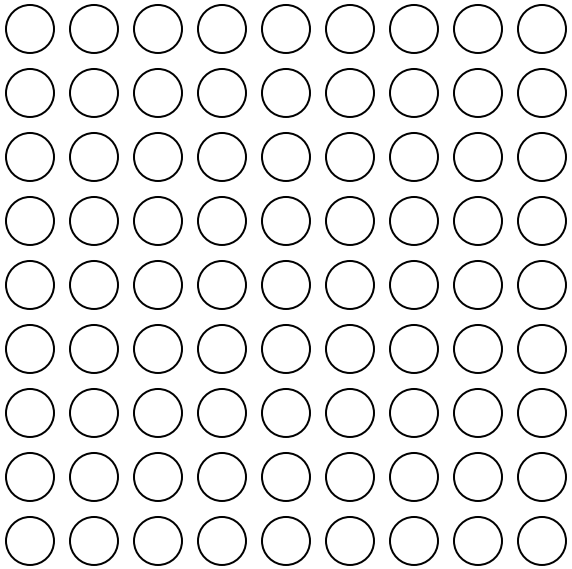
- structured vs. unstructured mesh
- shared vs. distributed data
- sparse vs. dense representation

Data Parallelism in LULESH (Structured)

```
const Elems = {0..#elemsPerEdge, 0..#elemsPerEdge},
      Nodes = {0..#nodesPerEdge, 0..#nodesPerEdge};

var determ: [Elems] real;

forall k in Elems { …determ[k]… }
```



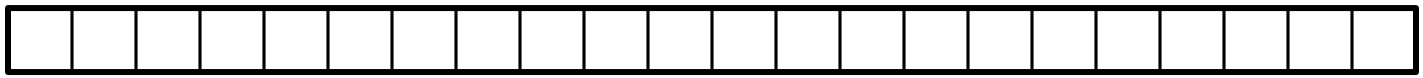
Elems



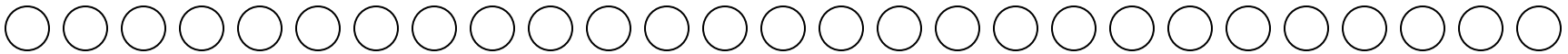
Nodes

Data Parallelism in LULESH (Unstructured)

```
const Elems = {0..#numElems},
      Nodes = {0..#numNodes};

var determ: [Elems] real;

forall k in Elems { …determ[k]… }
```



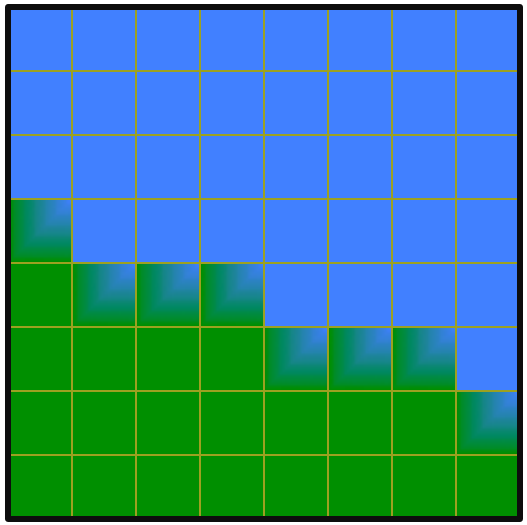
Elems



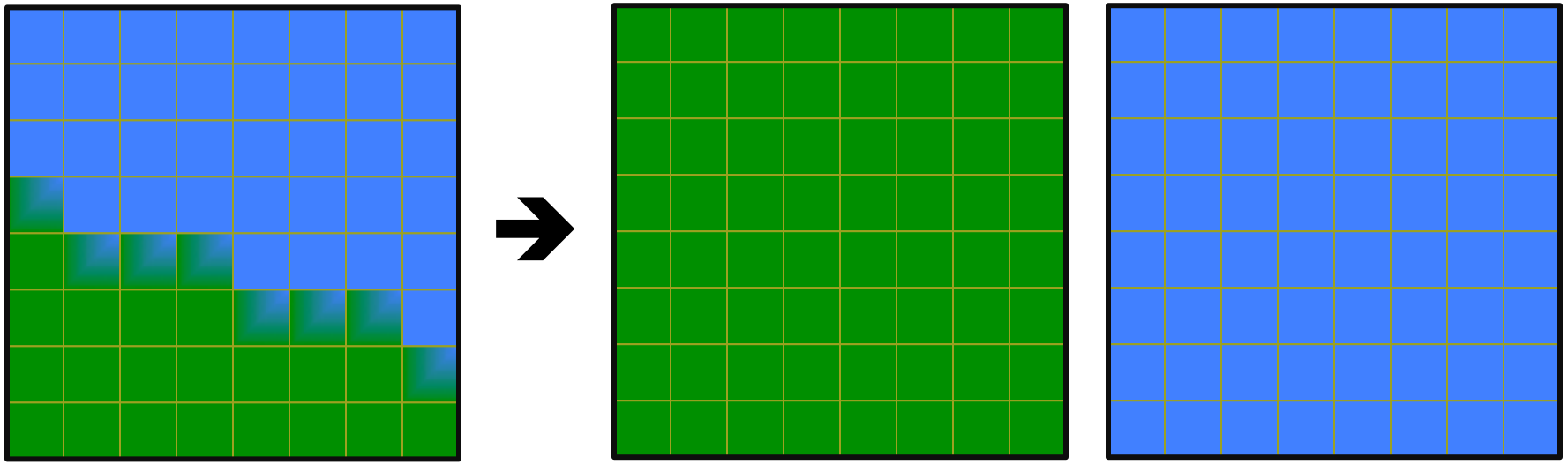
Nodes

Materials Representation

- Not all elements will contain all materials, and some will contain combinations



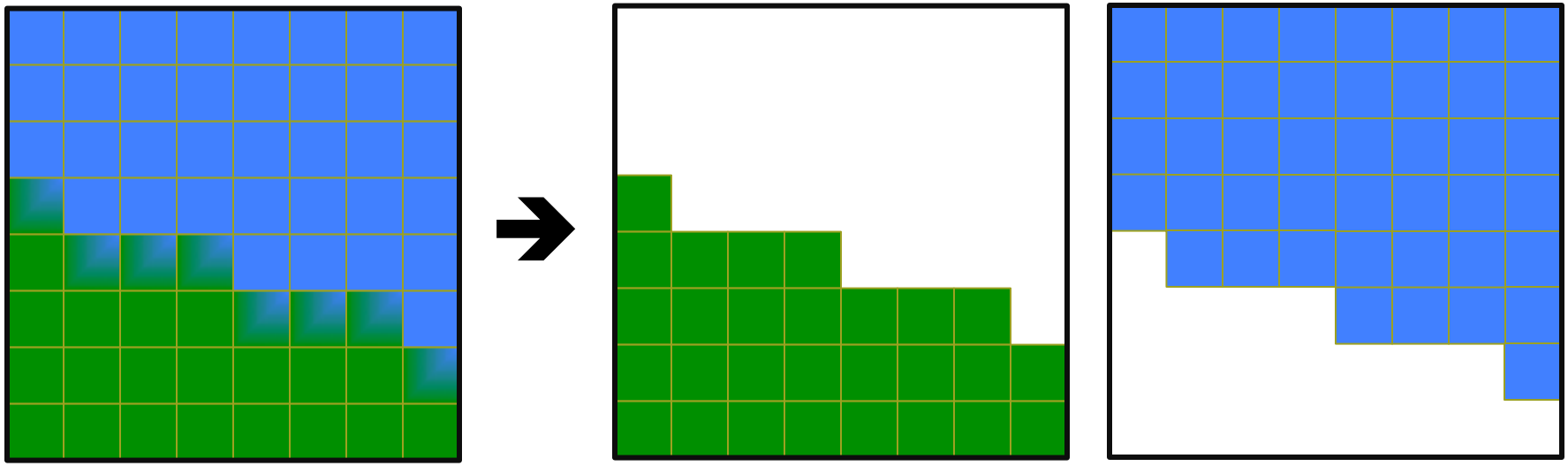
Materials Representation (Dense)



naïve approach: store all materials everywhere
(reasonable for LULESH 1.0, but not in practice)

```
const Mat1Elems = Elems,
      Mat2Elems = Elems;
```

Materials Representation (Sparse)

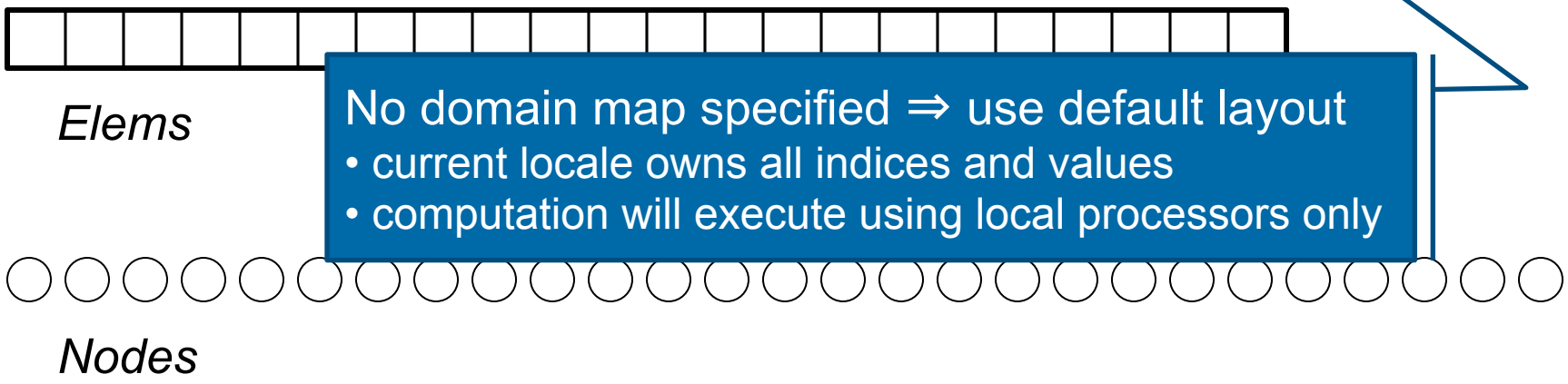


improved approach: use sparse subdomains to only store materials where necessary

```
var Mat1Elems: sparse subdomain(Elems) = enumerateMat1Locs(),
    Mat2Elems: sparse subdomain(Elems) = enumerateMat2Locs();
```

LULESH Data Structures (local)

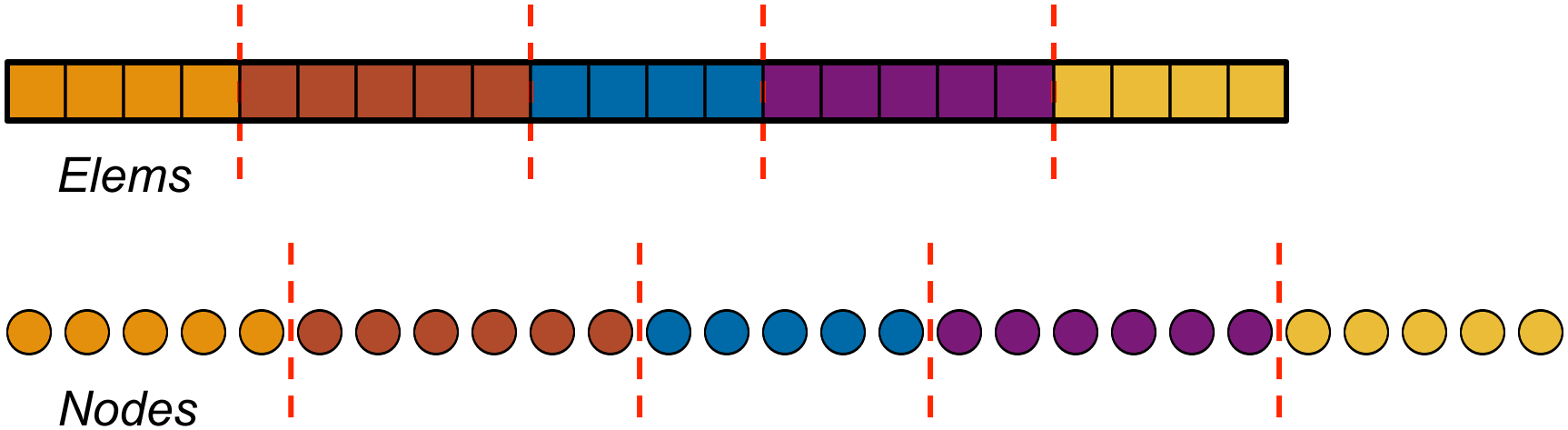
```
const Elems = {0..#numElems},
      Nodes = {0..#numNodes};

var determ: [Elems] real;

forall k in Elems { … }
```



No domain map specified ⇒ use default layout
- current locale owns all indices and values
- computation will execute using local processors only

LULESH Data Structures (distributed, block)

```
const Elems = {0..#numElems} dmapped Block(…),
      Nodes = {0..#numNodes} dmapped Block(…);

var determ: [Elems] real;

forall k in Elems { … }
```



LULESH Data Structures (distributed, cyclic)

```
const Elems = {0..#numElems} dmapped Cyclic(…),
      Nodes = {0..#numNodes} dmapped Cyclic(…);

var determ: [Elems] real;

forall k in Elems { … }
```

Elems

Nodes

Compile-time Reconfiguration in LULESH

```
config param use3DRepresentation = false,
             useBlockDist = (CHPL_COMM != "none"),
             useSparseMaterials = true;
```

Compile-time Reconfiguration in LULESH

```
const ElemSpace = if use3DRepresentation
                    then {0..#elemsPerEdge, 0..#elemsPerEdge, 0..#elemsPerEdge}
                    else {0..#numElems},
      NodeSpace = if use3DRepresentation
                    then {0..#nodesPerEdge, 0..#nodesPerEdge, 0..#nodesPerEdge}
                    else {0..#numNodes};

const Elems = if useBlockDist then ElemSpace dmapped Block(ElemSpace)
                              else ElemSpace,
      Nodes = if useBlockDist then NodeSpace dmapped Block(NodeSpace)
                              else NodeSpace;

var elemToNode: [Elems] nodesPerElem*index(Nodes);

const MatElems: MatElemsType = if sparseMaterials then enumerateMatElems()
                                                   else Elems;

proc MatElemsType type {
  if useSparseMaterials then
    return sparse subdomain(Elems);
  else
    return Elems.type;
}
```

Compile-time Reconfiguration in LULESH

```
const ElemSpace = if use3DRepresentation
                    then {0..#elemsPerEdge, 0..#elemsPerEdge, 0..#elemsPerEdge}
                    else {0..#numElems},
      NodeSpace = if use3DRepresentation
                    then {0..#nodesPerEdge, 0..#nodesPerEdge, 0..#nodesPerEdge}
                    else {0..#numNodes};

const Elems = if useBlockDist then ElemSpace dmapped Block(ElemSpace)
                              else ElemSpace,
      Nodes = if useBlockDist then NodeSpace dmapped Block(NodeSpace)
                              else NodeSpace;


var elemToNode: [Elems] nodesPerElem*index(Nodes);


const MatElems: MatElemsType = if sparseMaterials then enumerateMatElems()
                                                   else Elems;


proc MatElemsType type {
  if useSparseMaterials then
    return sparse subdomain(Elems);
  else
    return Elems.type;
}
```

domains for elements and nodes

Compile-time Reconfiguration in LULESH

```
const ElemSpace = if use3DRepresentation
                    then {0..#elemsPerEdge, 0..#elemsPerEdge, 0..#elemsPerEdge}
                    else {0..#numElems},
      NodeSpace = if use3DRepresentation
                    then {0..#nodesPerEdge, 0..#nodesPerEdge, 0..#nodesPerEdge}
                    else {0..#numNodes};
```

```
const Elems = if useBlockDist then ElemSpace dmapped Block(ElemSpace)
                              else ElemSpace,
      Nodes = if useBlockDist then NodeSpace dmapped Block(NodeSpace)
                              else NodeSpace;
```

```
var elemToNode: [Elems] nodesPerElem*index(Nodes);
```

```
const MatElems: MatElemsType = if sparseMaterials then enumerateMatElems()
                                                   else Elems;

proc MatElemsType type {
  if useSparseMaterials then
    return sparse subdomain(Elems);
  else
    return Elems.type;
}
```

potentially distributed domains for elements and nodes

Compile-time Reconfiguration in LULESH

```
const ElemSpace = if use3DRepresentation
                    then {0..#elemsPerEdge, 0..#elemsPerEdge, 0..#elemsPerEdge}
                    else {0..#numElems},
      NodeSpace = if use3DRepresentation
                    then {0..#nodesPerEdge, 0..#nodesPerEdge, 0..#nodesPerEdge}
                    else {0..#numNodes};

const Elems = if useBlockDist then ElemSpace dmapped Block(ElemSpace)
                              else ElemSpace,
      Nodes = if useBlockDist then NodeSpace dmapped Block(NodeSpace)
                              else NodeSpace;
```

```
var elemToNode: [Elems] nodesPerElem*index(Nodes);
```

```
const MatElems: MatElemsType = if sparseMaterials then enumerateMatElems()
                                                   else Elems;

proc MatElemsType type {
  if useSparseMaterials then
    return sparse subdomain(Elems);
  else
    return Elems.type;
}
```

nodes adjacent to each element

Compile-time Reconfiguration in LULESH

```
const ElemSpace = if use3DRepresentation
                    then {0..#elemsPerEdge, 0..#elemsPerEdge, 0..#elemsPerEdge}
                    else {0..#numElems},
      NodeSpace = if use3DRepresentation
                    then {0..#nodesPerEdge, 0..#nodesPerEdge, 0..#nodesPerEdge}
                    else {0..#numNodes};

const Elems = if useBlockDist then ElemSpace dmapped Block(ElemSpace)
                              else ElemSpace,
      Nodes = if useBlockDist then NodeSpace dmapped Block(NodeSpace)
                              else NodeSpace;

var elemToNode: [Elems] nodesPerElem*index(Nodes);
```

```
const MatElems: MatElemsType = if sparseMaterials then enumerateMatElems()
                                                   else Elems;

proc MatElemsType type {
  if useSparseMaterials then
    return sparse subdomain(Elems);
  else
    return Elems.type;
}
```

domain describing elements that contain the material

Compile-time Reconfiguration in LULESH

```
const ElemSpace = if use3DRepresentation
                    then {0..#elemsPerEdge, 0..#elemsPerEdge, 0..#elemsPerEdge}
                    else {0..#numElems},
      NodeSpace = if use3DRepresentation
                    then {0..#nodesPerEdge, 0..#nodesPerEdge, 0..#nodesPerEdge}
                    else {0..#numNodes};

const Elems = if useBlockDist then ElemSpace dmapped Block(ElemSpace)
                              else ElemSpace,
      Nodes = if useBlockDist then NodeSpace dmapped Block(NodeSpace)
                              else NodeSpace;

var elemToNode: [Elems] nodesPerElem*index(Nodes);

const MatElems: MatElemsType = if sparseMaterials then enumerateMatElems()
                                                   else Elems;
```

```
proc MatElemsType type {
  if useSparseMaterials then
    return sparse subdomain(Elems);
  else
    return Elems.type;
}
```

the type of the domain describing elements that contain the material

Additional Data Parallel Notes

SC14
New Orleans, LA | hpc matters.

Notes on Forall Loops

```
forall a in A do
  writeln("Here is an element of A: ", a);
```

Typically:
- 1 ≤ #Tasks << #Iterations
- #Tasks ≈ amount of HW parallelism

```
forall (a, i) in zip(A, 1..n) do
  a = i / 10.0;
```

Like for loops, forall-loops may be zippered, and corresponding iterations will match up

Motivation for Leader-Follower Iterators

Q: How are parallel loops implemented?

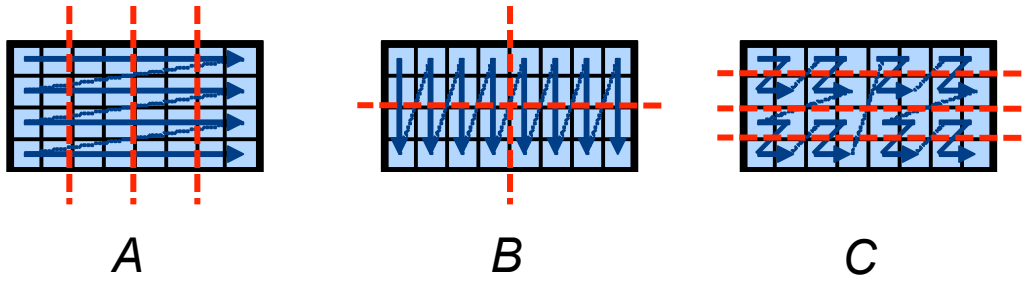
(how many tasks? executing where? how are iterations divided up?)

```
forall a in A { … }
```

Q2: What about zippered data parallel operations?

(how to reconcile potentially conflicting parallel implementations?)

```
forall (a,b,c) in zip(A,B,C) { … }
  a = b + alpha * c;
```



A: Via Leader-Follower Iterators…

- see locality section
- like domain maps, can be written by users
  - gives control over crucial scheduling and placement decisions

Chapel Promotion Semantics

Promoted functions/operators are defined in terms of zippered forall loops in Chapel. For example…

```
A = B;
```

…is equivalent to:

```
forall (a,b) in zip(A,B) do
  a = b;
```

Implication of Zippered Promotion Semantics

Whole-array operations are implemented element-wise…

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

⇒

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

…rather than operator-wise.

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

⇏

```
T1 = alpha * C;
A = B + T1;
```

Implication of Zippered Promotion Semantics

Whole-array operations are implemented element-wise…

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

⇒

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

⇒ No temporary arrays required by semantics

- ⇒ No surprises in memory requirements
- ⇒ Friendlier to cache utilization

⇒ Differs from traditional array language semantics

```
A[D] = A[D-one] + A[D+one];
```

⇒

```
forall (a1, a2, a3)
 in (A[D], A[D-one], A[D+one]) do
   a1 = a2 + a3;
```

Read/write race!

Implication of Zippered Promotion Semantics

Whole-array operations are implemented element-wise…

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

⇒

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

⇒ No temporary arrays required by semantics

- ⇒ No surprises in memory requirements
- ⇒ Friendlier to cache utilization

⇒ Differs from traditional array language semantics

```
B[D] = A[D-one] + A[D+one];
```

⇒

```
forall (b,  a2, a3)
  in (B[D], A[D-one], A[D+one]) do
    b  = a2 + a3;
```

OK!

Questions about Data Parallelism in Chapel?

Legal Disclaimer

Information in this document is provided in connection with Cray Inc. products. No license, express or implied, to any intellectual property rights is granted by this document.

Cray Inc. may make changes to specifications and product descriptions at any time, without notice.

All products, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

Cray hardware and software products may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Cray uses codenames internally to identify products that are in development and not yet publically announced for release. Customers and other third parties are not authorized by Cray Inc. to use codenames in advertising, promotion or marketing and any use of Cray Inc. internal codenames is at the sole risk of the user.

Performance tests and ratings are measured using specific systems and/or components and reflect the approximate performance of Cray Inc. products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.

The following are trademarks of Cray Inc. and are registered in the United States and other countries: CRAY and design, SONEXION, URIKA, and YARCDATA. The following are trademarks of Cray Inc.: ACE, APPRENTICE2, CHAPEL, CLUSTER CONNECT, CRAYPAT, CRAYPORT, ECOPHLEX, LIBSCI, NODEKARE, THREADSTORM. The following system family marks, and associated model number marks, are trademarks of Cray Inc.: CS, CX, XC, XE, XK, XMT, and XT. The registered trademark LINUX is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Other trademarks used in this document are the property of their respective owners.

CRAY
THE SUPERCOMPUTER COMPANY

http://chapel.cray.com chapel_info@cray.com http://sourceforge.net/projects/chapel/