Productive Programming in Chapel:
A Computation-Driven Introduction

Data Parallelism with Jacobi

Michael Ferguson and Lydia Duncan
Cray Inc,
SC15 November 15th, 2015

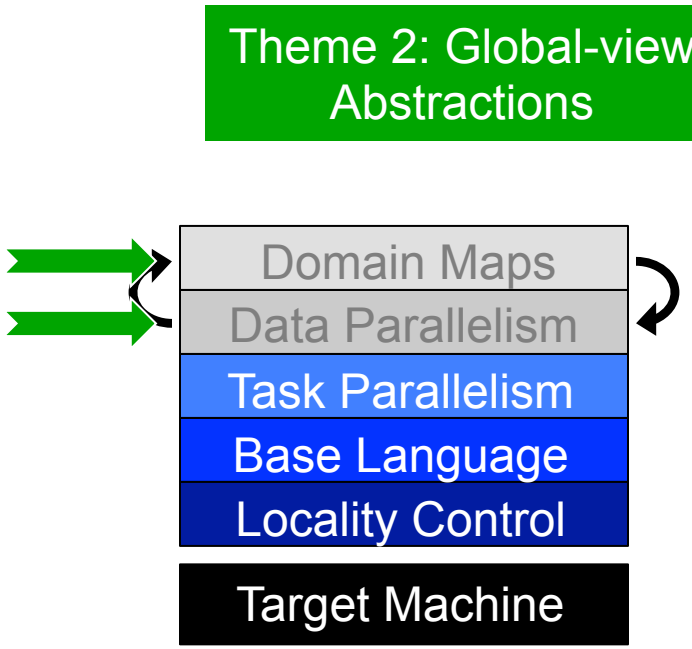
Safe Harbor Statement

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.

Copyright 2015 Cray Inc.

Outline

- ✓ Motivation
- ✓ Chapel Background and Themes
- ✓ Learning the Base Language with n-body
- ✓ Short Introduction to Task Par
- ✓ Hands-On 1: Hello World
- ✓ Short Introduction to Locality
- ➢ **Data Parallelism with Jacobi**
- **Hands-On 2: Mandelbrot**
- **Project Status, Next Steps**



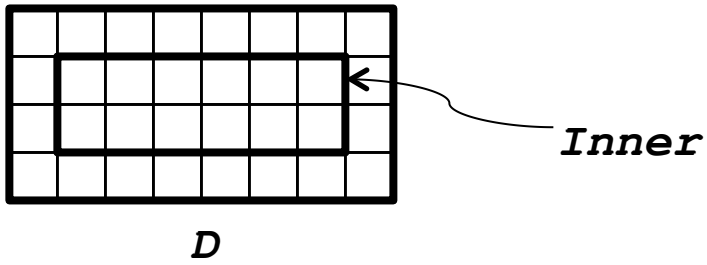
Copyright 2014 Cray Inc.

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};
```



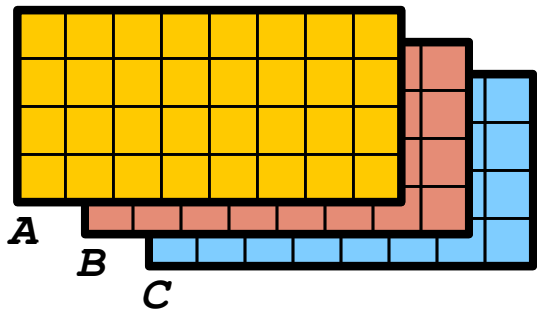
Copyright 2015 Cray Inc.

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;
```



Copyright 2015 Cray Inc.

Chapel Data Parallel Operations

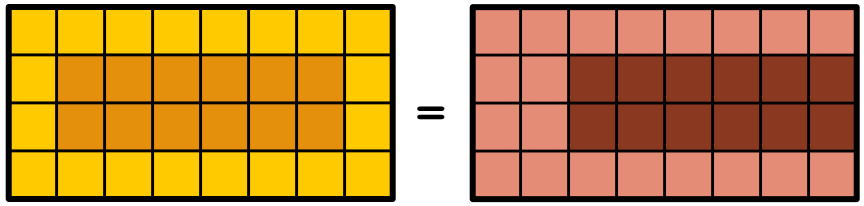
- **Data Parallel Iteration**

```
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);
```

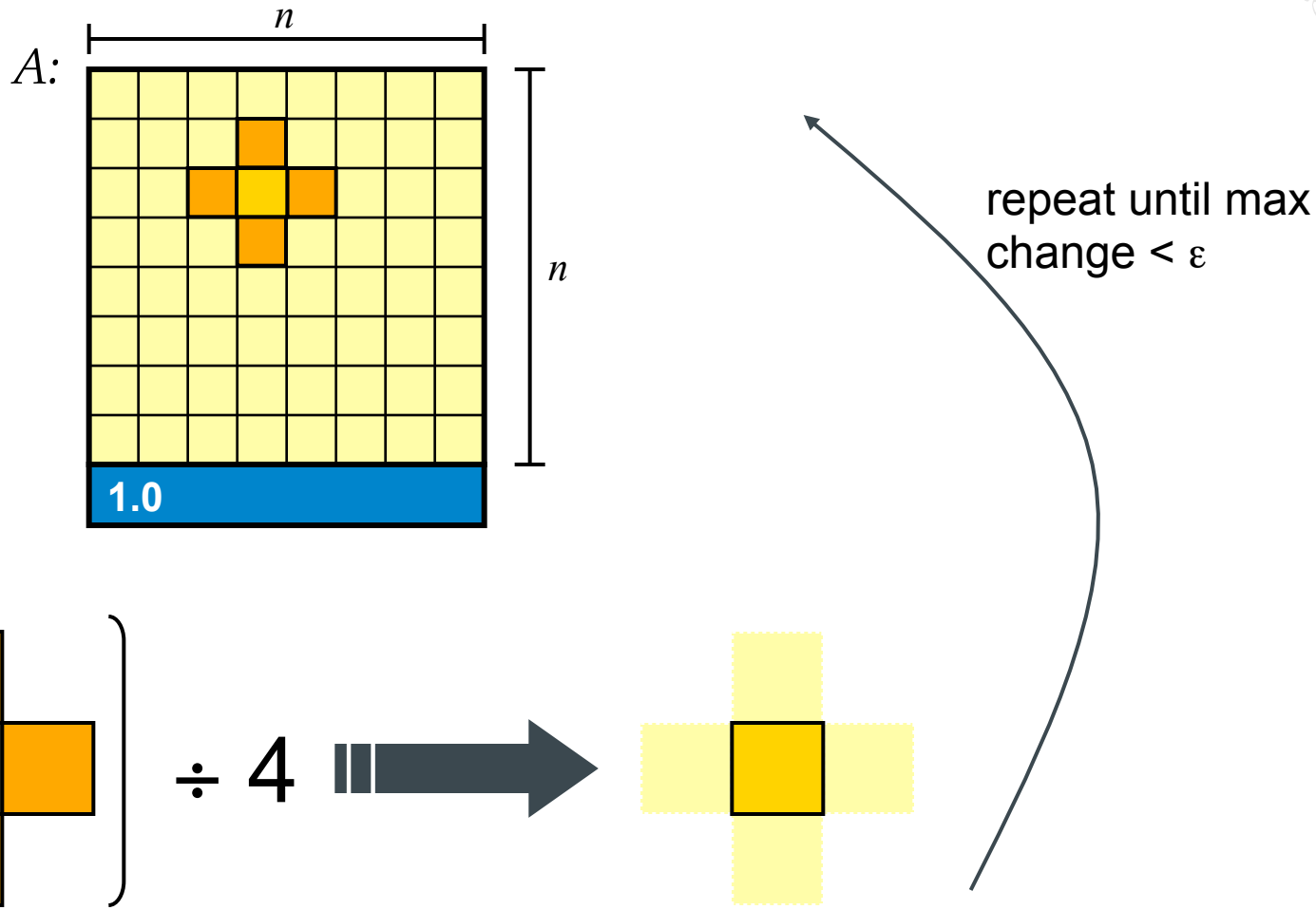
```
A = foo("hi", B, C);
```

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

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

Copyright 2015 Cray Inc.

Data Parallelism by Example: Jacobi Iteration



Copyright 2015 Cray Inc.

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);
```


Copyright 2015 Cray Inc.

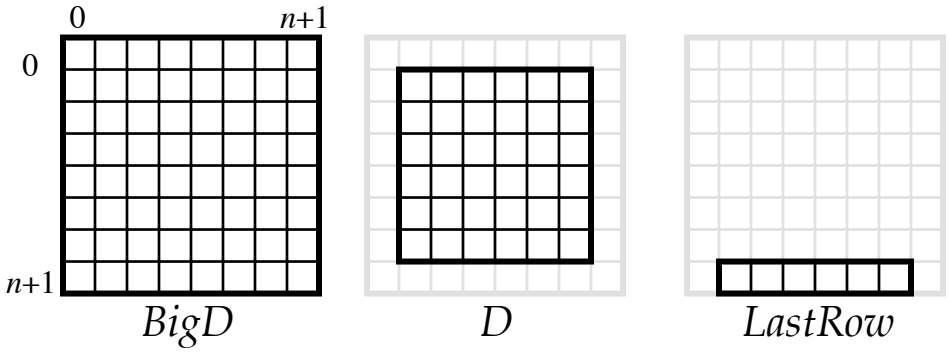
Jacobi Iteration in Chapel

```
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

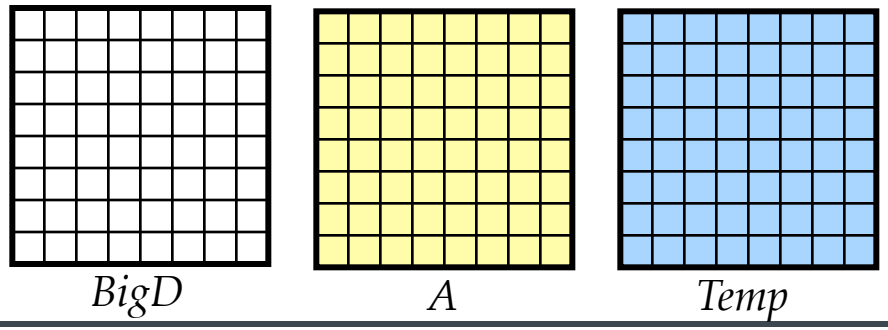
Copyright 2015 Cray Inc.

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;
```

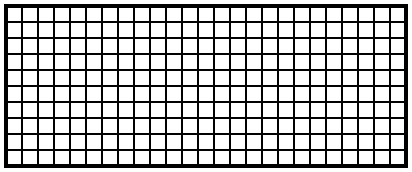
Declare arrays

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)

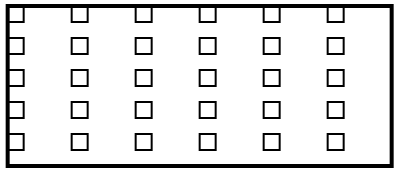


Copyright 2015 Cray Inc.

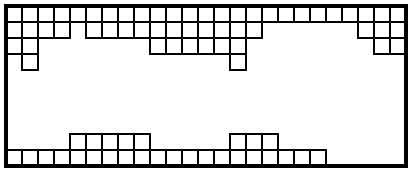
Chapel Domain Types



dense

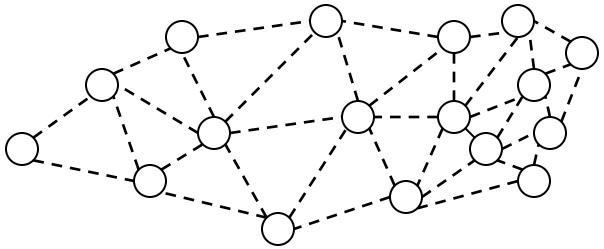


strided



sparse

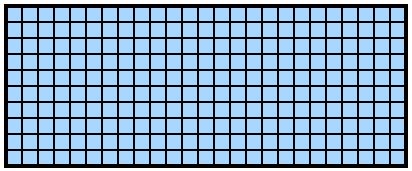
associative



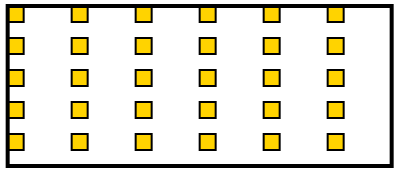
unstructured

Copyright 2015 Cray Inc.

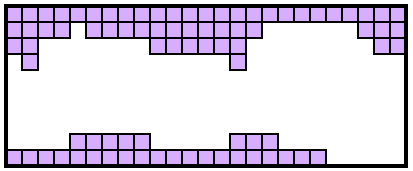
Chapel Array Types



dense

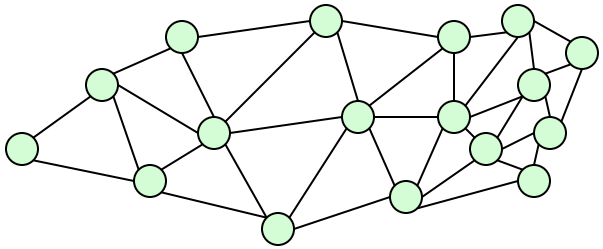


strided



sparse

associative



unstructured

Copyright 2015 Cray Inc.

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);
```


Copyright 2015 Cray Inc.

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

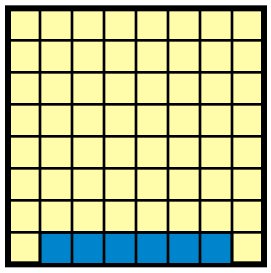
Copyright 2015 Cray Inc.

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

Copyright 2015 Cray Inc.

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);
```


Copyright 2015 Cray Inc.

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);
```


Copyright 2015 Cray Inc.

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

Copyright 2015 Cray Inc.

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;
```

By default, domains and their arrays are mapped to a single locale.
Any data parallelism over such domains/ arrays will be executed by the cores on that locale.
Thus, this is a shared-memory parallel program.

```
    Temp[1,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);
```


Copyright 2015 Cray Inc.

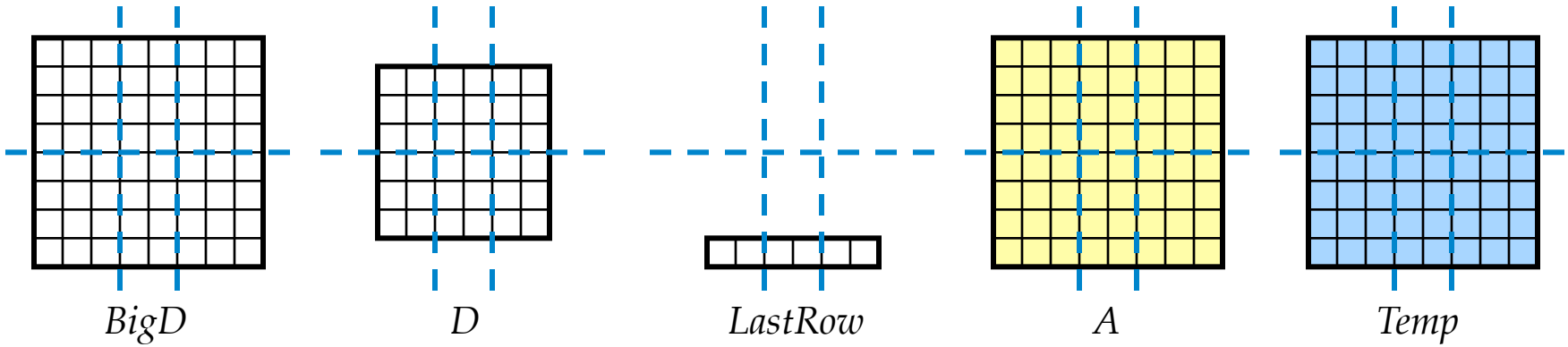
Jacobi Iteration in Chapel

```
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



Copyright 2015 Cray Inc.

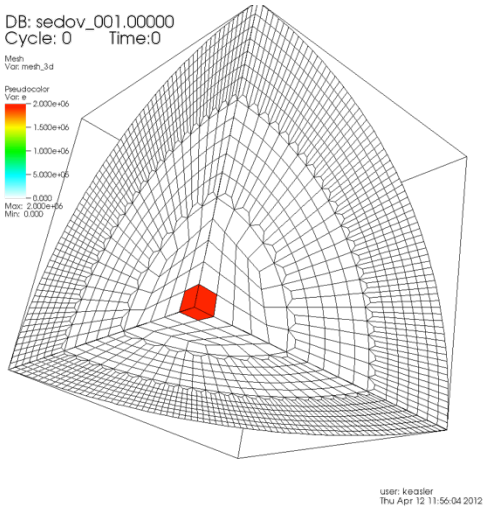
Jacobi Iteration in Chapel

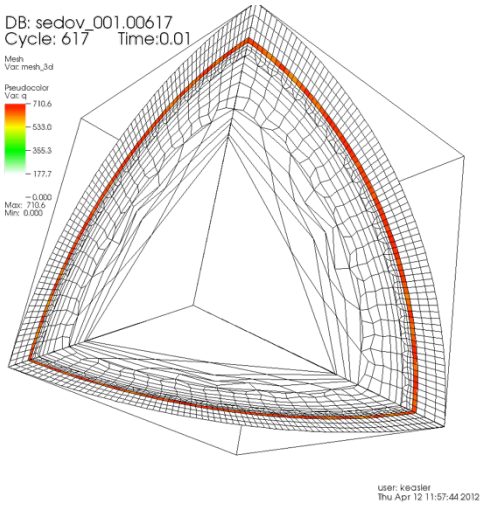
```
config const n = 6,
             epsilon = 1.0e-5;

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;
```

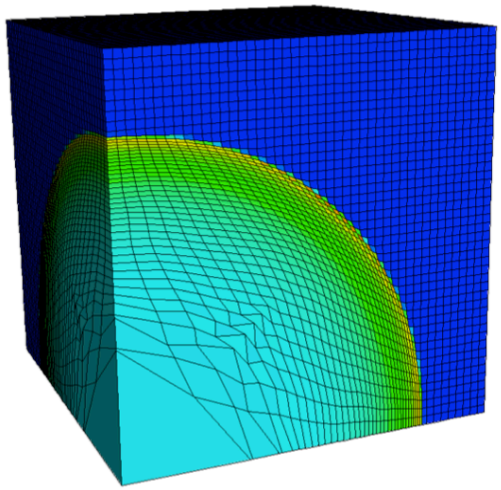

Copyright 2015 Cray Inc.

LULESH: a DOE Proxy Application

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







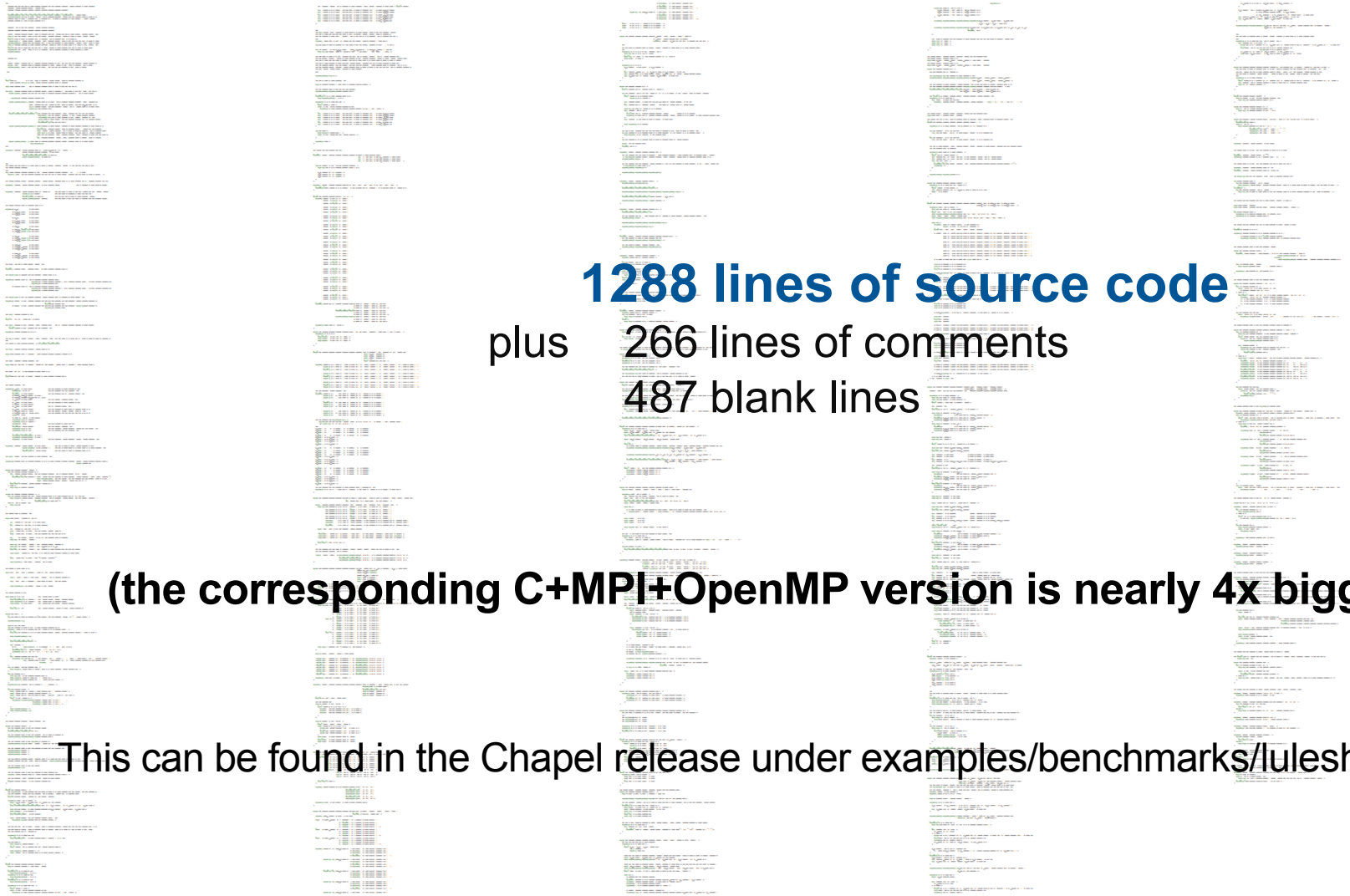
pictures courtesy of Rob Neely, Bert Still, Jeff Keasler, LLNL

Copyright 2015 Cray Inc.

LULESH in Chapel

Copyright 2015 Cray Inc.

LULESH in Chapel



1288 lines of source code

plus 266 lines of comments

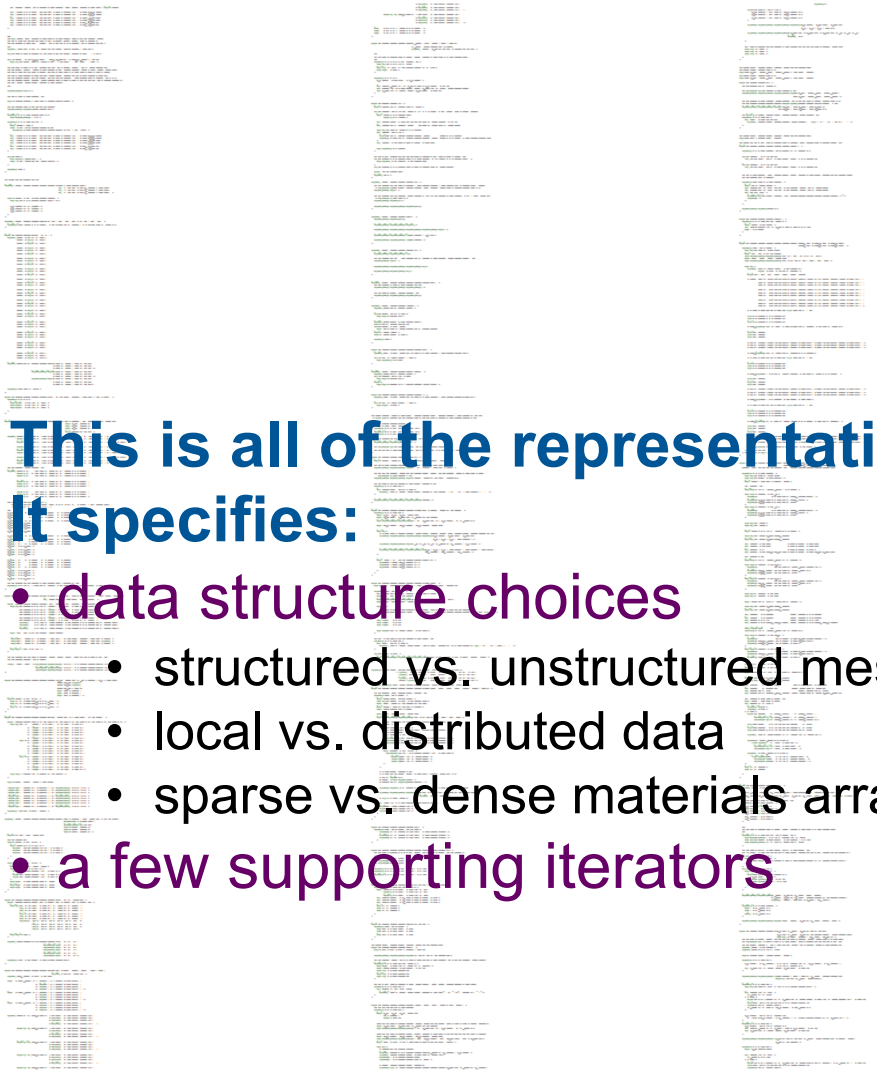
487 blank lines

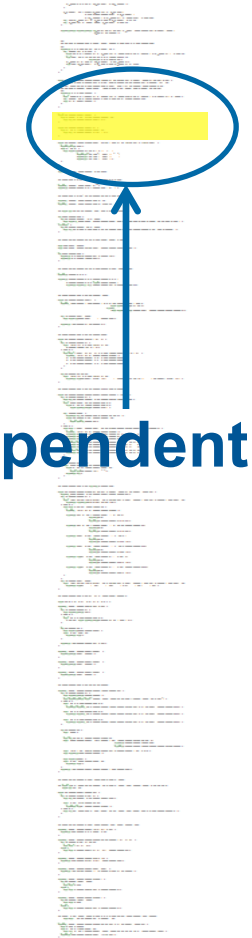
(the corresponding C+MPI+OpenMP version is nearly 4x bigger)

This can be found in the Chapel release under examples/benchmarks/lulesh/*.chpl

Copyright 2015 Cray Inc.

LULESH in Chapel



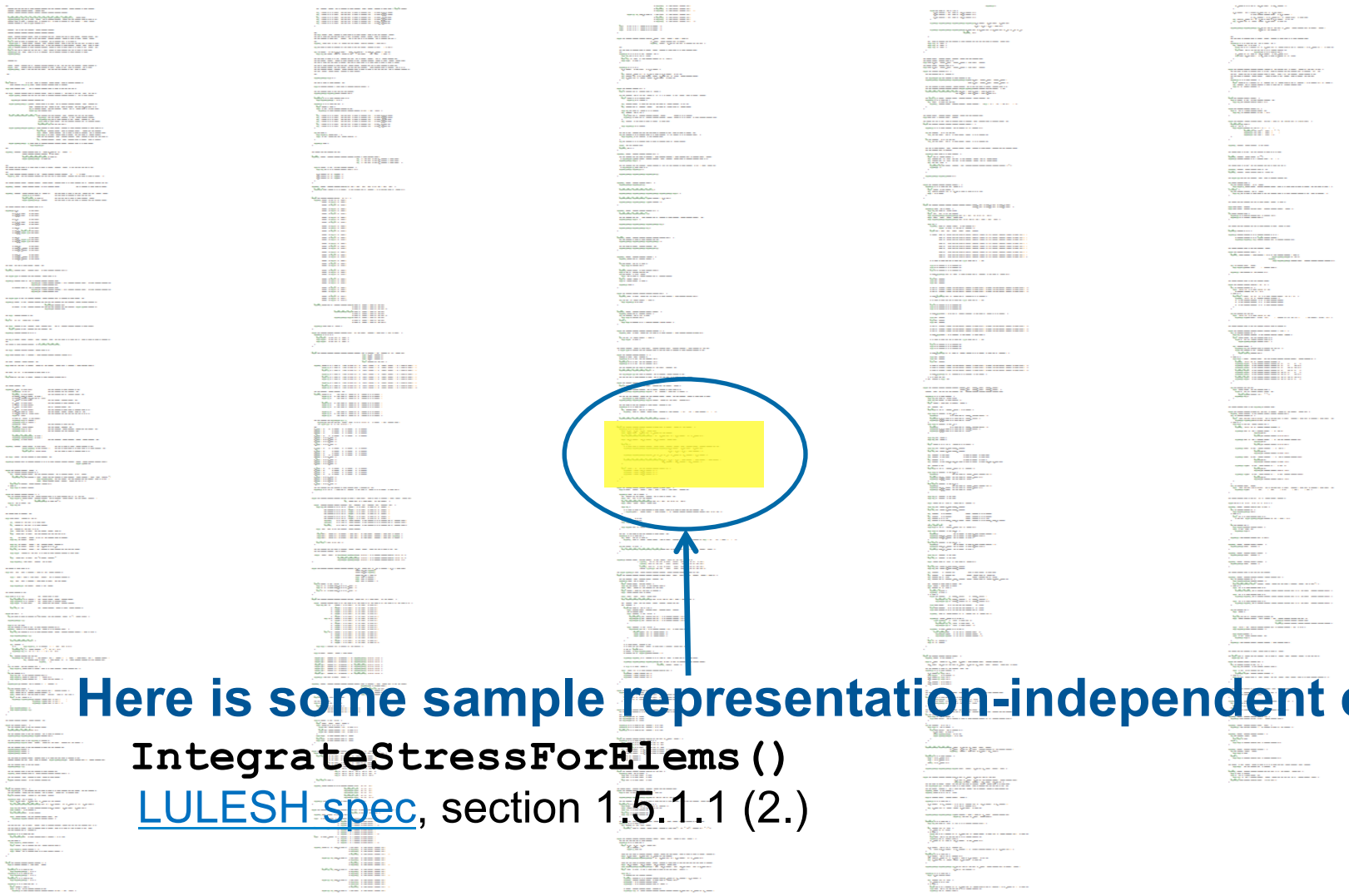


This is all of the representation dependent code. It specifies:

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

Copyright 2015 Cray Inc.

LULESH in Chapel



Here is some sample representation-independent code

`IntegrateStressForElems()`

LULESH spec, section 1.5.1.1 (2.)

Copyright 2015 Cray Inc.

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

Copyright 2015 Cray Inc.

STREAM Triad: Chapel

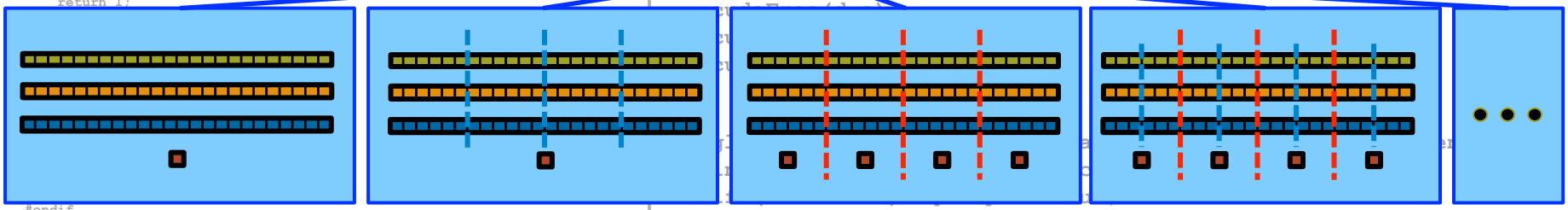
MPI + OpenMP

```
#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif

static int VectorSize;
static double *a, *b, *c;

int HPCC_StarStream(HPCC_Params *
  int myRank, commSize;
  int rv, errCount;
  MPI_Comm comm = MPI_COMM_WORLD;

  MPI_Comm_size( comm, &commSize );
  MPI_Comm_rank( comm, &myRank );

  rv = HPCC_Stream( params, 0 == myR
  MPI_Reduce( &rv, &errCount, 1, MPI

  return errCount;
}

int HPCC_Stream(HPCC_Params *params,
  register int j;
  double  scalar;

  VectorSize = HPCC_LocalVectorSize(

  a = HPCC_XMALLOC( double, VectorSi
  b = HPCC_XMALLOC( double, VectorSi
  c = HPCC_XMALLOC( double, VectorSi

  if (!a || !b || !c) {
    if (c) HPCC_free(c);
    if (b) HPCC_free(b);
    if (a) HPCC_free(a);
    if (doIO) {
      fprintf( outFile, "Failed to a
      fclose( outFile );
    }
    return 1;
```

Chapel

```
config const m = 1000,
             alpha = 3.0;

const ProblemSpace = {1..m} dmapped …;

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

B = 2.0;
C = 3.0;

A = B + alpha * C;
```

the special sauce

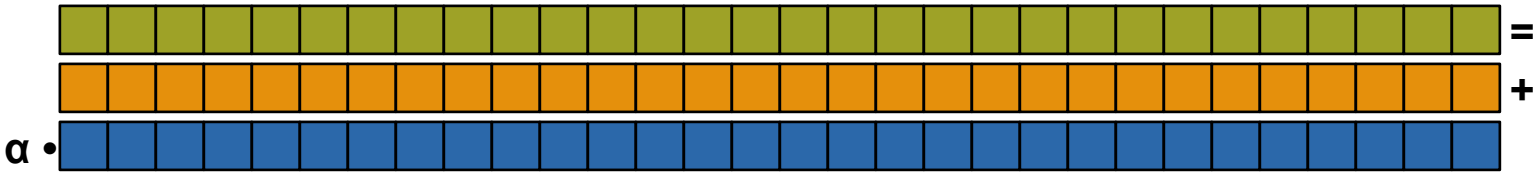


Philosophy: Good language design can tease details of locality and parallelism away from an algorithm, permitting the compiler, runtime, applied scientist, and HPC expert to each focus on their strengths.

Copyright 2015 Cray Inc.

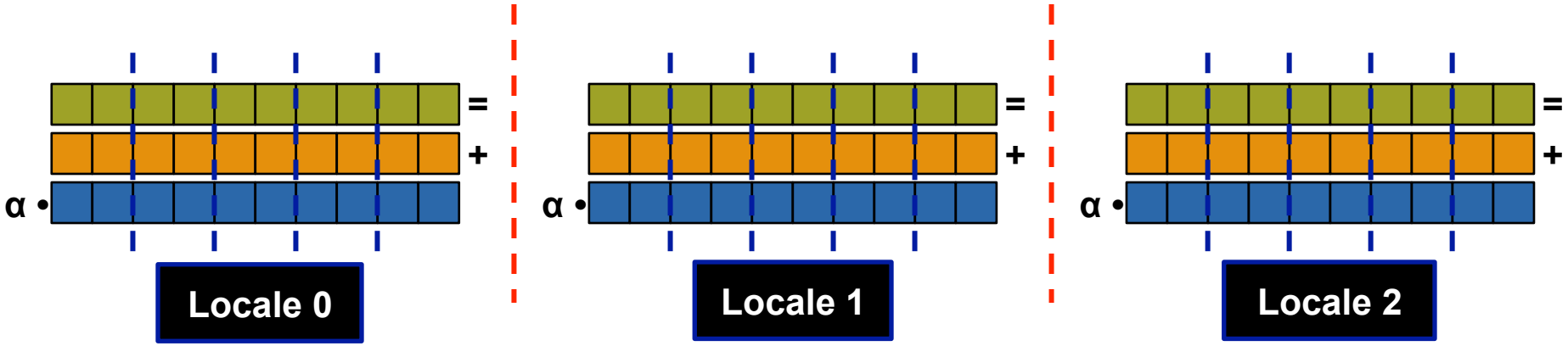
Domain Maps

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:

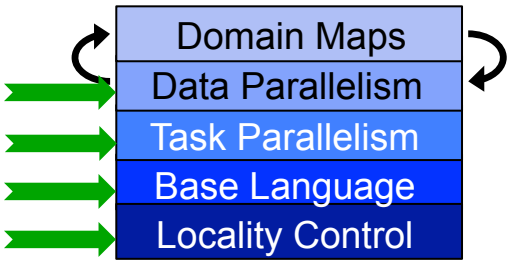


Copyright 2015 Cray Inc.

Chapel's Domain Map Philosophy

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

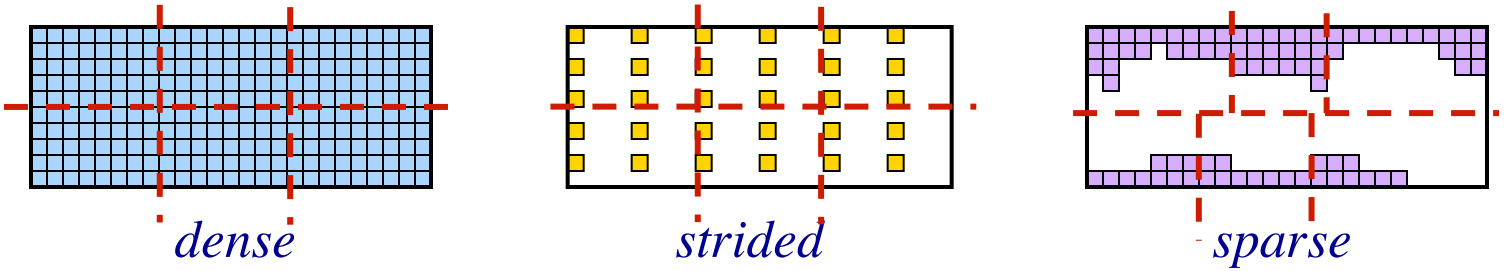
2. **Expert users can write their own domain maps in Chapel**
   - to cope with any shortcomings in our standard library

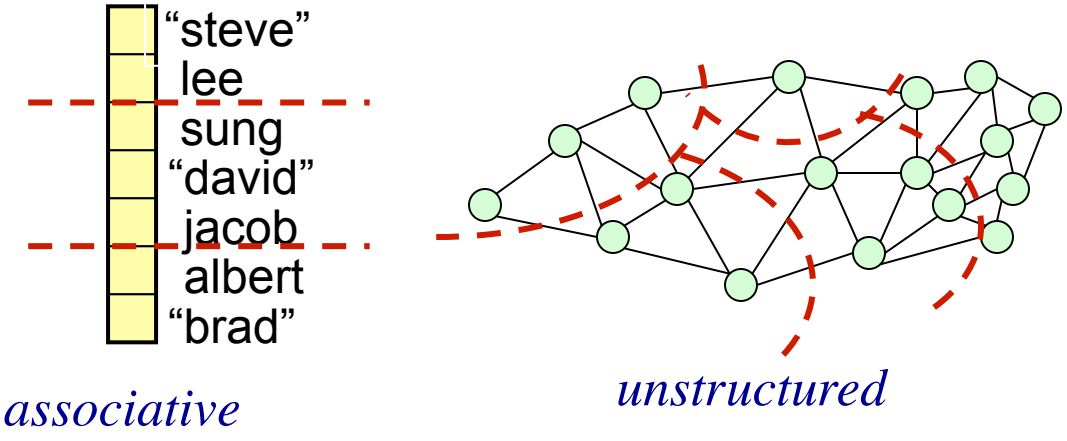


3. **Chapel's standard domain maps are written using the same end-user framework**
   - to avoid a performance cliff between "built-in" and user-defined cases

Copyright 2015 Cray Inc.

All Domain Types Support Domain Maps





Copyright 2015 Cray Inc.

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;
```


Copyright 2015 Cray Inc.

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;
```


Copyright 2015 Cray Inc.

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];
```

⇒

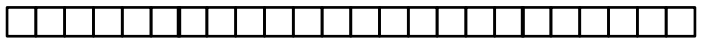
```
forall (a1, a2, a3)
  in (A[D], A[D-one], A[D+one]) do
    a1 = a2 + a3;
```

Read/write race!

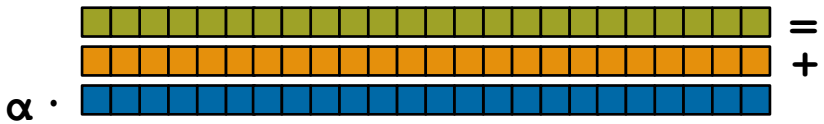
Copyright 2015 Cray Inc.

STREAM Triad in Chapel

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



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



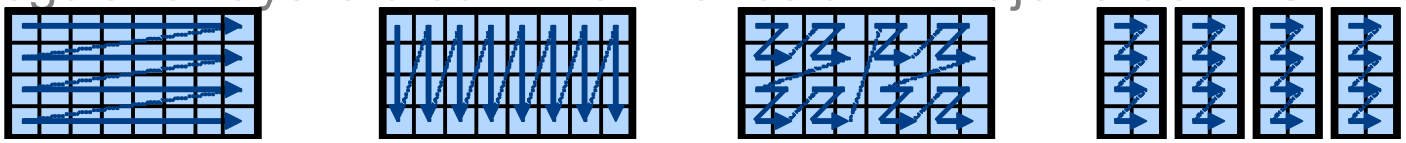

```
A = B + alpha * C;
```


Copyright 2015 Cray Inc.

Data Parallelism Implementation Qs

Q1: How are arrays laid out in memory?

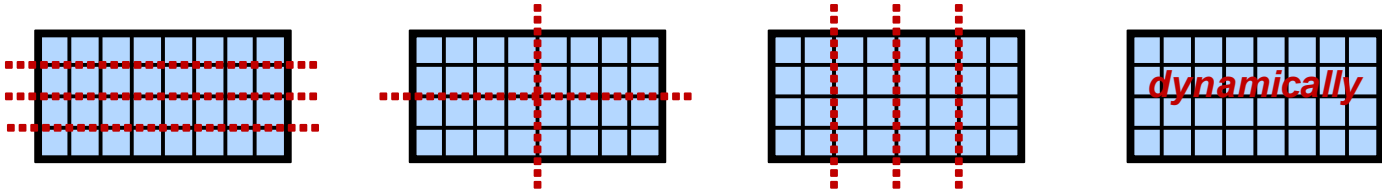
- Are regular arrays laid out in row- or column-major order? Or…?



- How are sparse arrays stored? (COO, CSR, CSC, block-structured, …?)

Q2: How are arrays stored by the locales?

- Completely local to one locale? Or distributed?
- If distributed… In a blocked manner? cyclically? block-cyclically? recursively bisected? dynamically rebalanced? …?

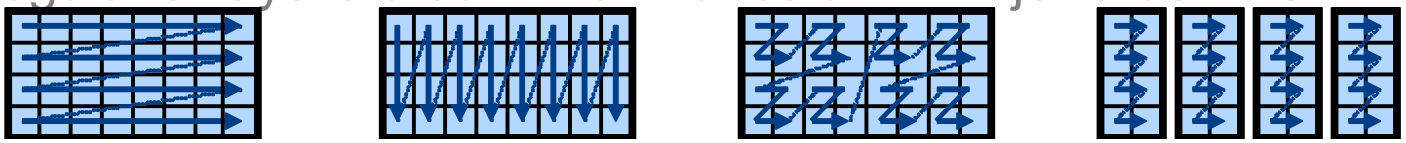


Copyright 2015 Cray Inc.

Data Parallelism Implementation Qs

Q1: How are arrays laid out in memory?

- Are regular arrays laid out in row- or column-major order? Or…?



- How are sparse arrays stored? (COO, CSR, CSC, block-structured, …?)

Q2: How are arrays stored by the locales?

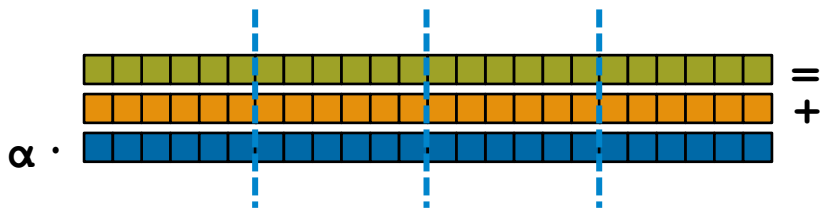
- Completely local to one locale? Or distributed?
- If distributed… In a blocked manner? cyclically? block-cyclically? recursively bisected? dynamically rebalanced? …?

A: Chapel's *domain maps* are designed to give the user full control over such decisions

Copyright 2015 Cray Inc.

STREAM Triad: Chapel (multicore)

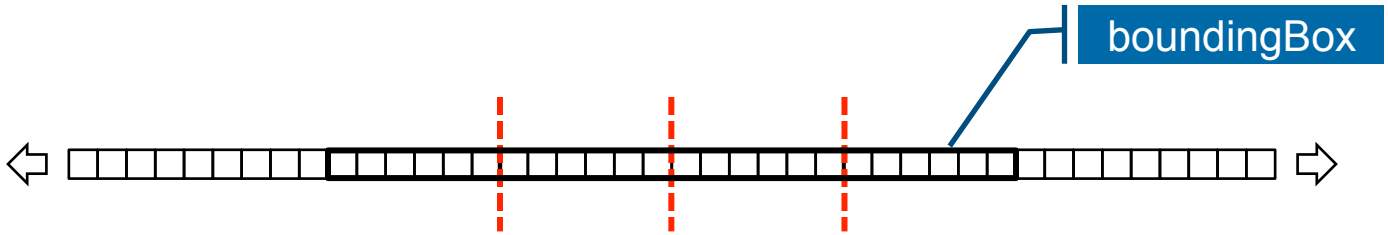
```
const ProblemSpace = {1..m};

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

A = B + alpha * C;
```



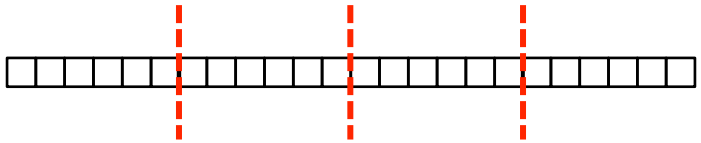
No domain map specified => use default layout
- current locale owns all domain indices and array values
- computation will execute using local processors only

Copyright 2015 Cray Inc.

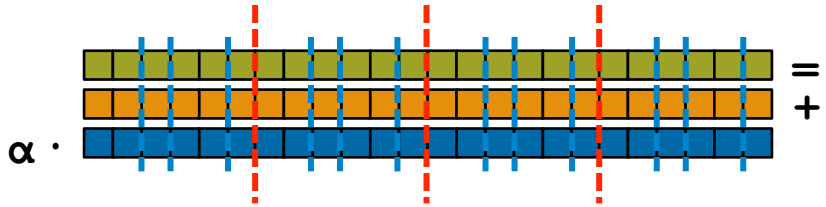
STREAM Triad: Chapel (multilocale, blocked)




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



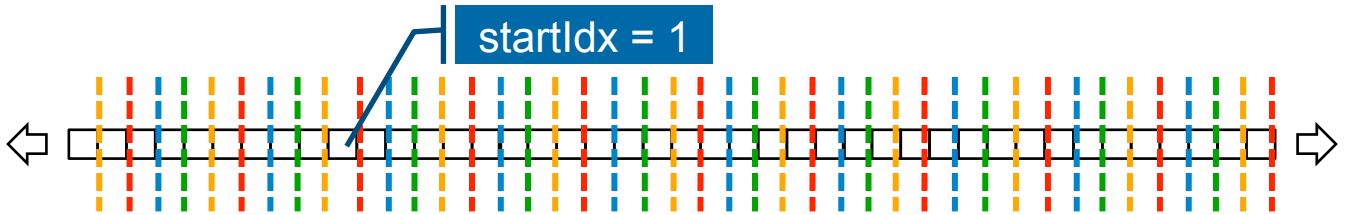
```
var A, B, C: [ProblemSpace] real;
```



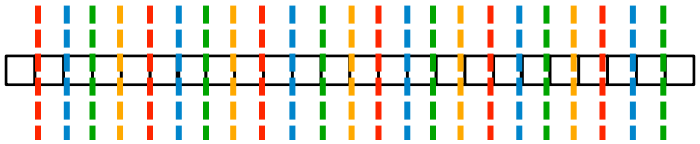

```
A = B + alpha * C;
```


Copyright 2015 Cray Inc.

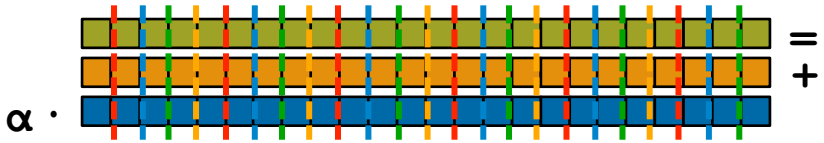
STREAM Triad: Chapel (multilocale, cyclic)




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



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

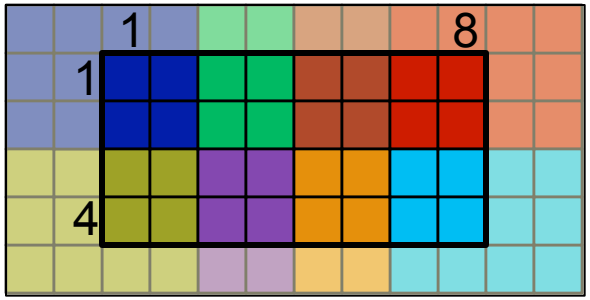



```
A = B + alpha * C;
```


Copyright 2015 Cray Inc.

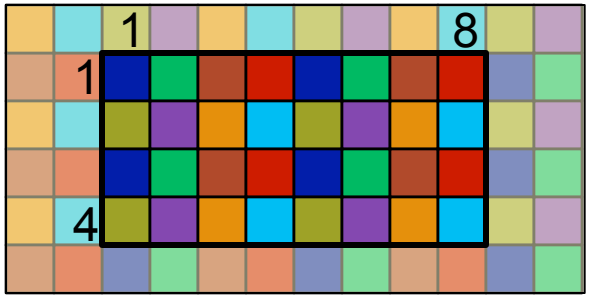
Sample Distributions: Block and Cyclic

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



distributed to

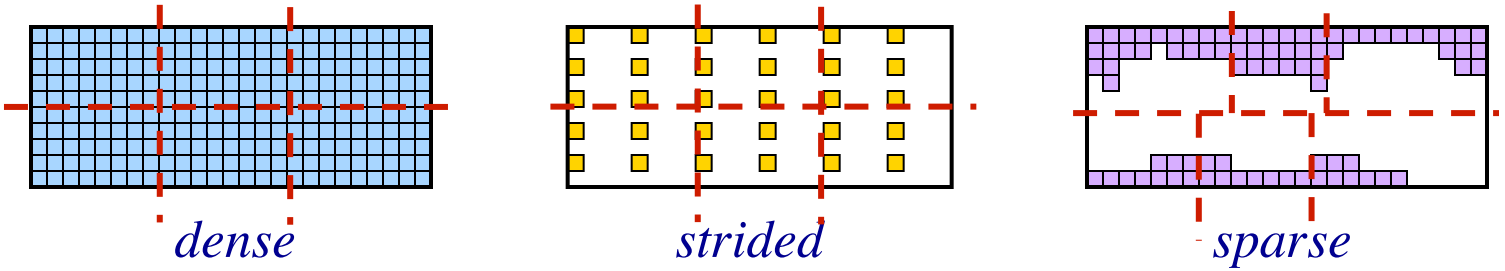

```
var Dom = {1..4, 1..8} dmapped Cyclic( startIdx=(1,1) );
```

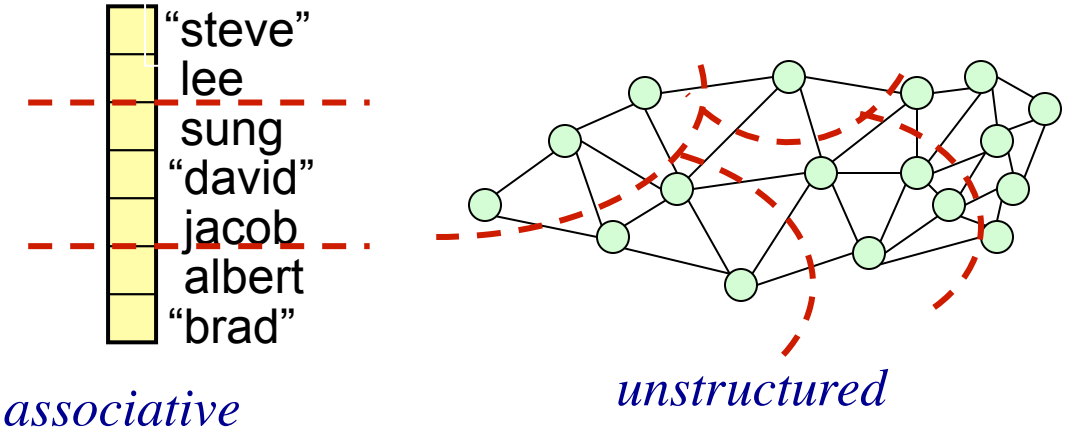


distributed to

Copyright 2015 Cray Inc.

All Domain Types Support Domain Maps





Copyright 2015 Cray Inc.

Layouts and Distributions

Domain Maps fall into two major categories:

layouts:

- e.g., a desktop machine or multicore node
- **examples:** row- and column-major order, tilings, compressed sparse row, space-filling curves

distributions:

- e.g., a distributed memory cluster or supercomputer
- **examples:** Block, Cyclic, Block-Cyclic, Recursive Bisection, …

Copyright 2015 Cray Inc.

Domain Map Roles

They define data storage:

- Mapping of domain indices and array elements to locales
- Layout of arrays and index sets in each locale's memory

…as well as operations:

- random access, iteration, slicing, reindexing, rank change, …
- the Chapel compiler generates calls to these methods to implement the user's array operations

Copyright 2015 Cray Inc.

Domain Maps: Next Steps

- **More advanced uses of domain maps:**
  - Dynamically load balanced domains/arrays
  - Resilient data structures
  - *in situ* interoperability with legacy codes
  - out-of-core computations

- **Further compiler optimization via optional interfaces**
  - particularly communication idioms (stencils, reductions, …)

Copyright 2015 Cray Inc.

Two Other Thematically Similar Features

1) **parallel iterators:** Permit users to specify the parallelism and work decomposition used by forall loops
   - including zippered forall loops

2) **locale models:** Permit users to model the target architecture and how Chapel should be implemented on it
   - e.g., how to manage memory, create tasks, communicate, …

Like domain maps, these are…

…written in Chapel by expert users using lower-level features
- e.g., task parallelism, on-clauses, base language features, …

…available to the end-user via higher-level abstractions
- e.g., forall loops, on-clauses, lexically scoped PGAS memory, …

Copyright 2015 Cray Inc.

More Data Parallelism Implementation Qs

Q1: How are forall loops implemented?

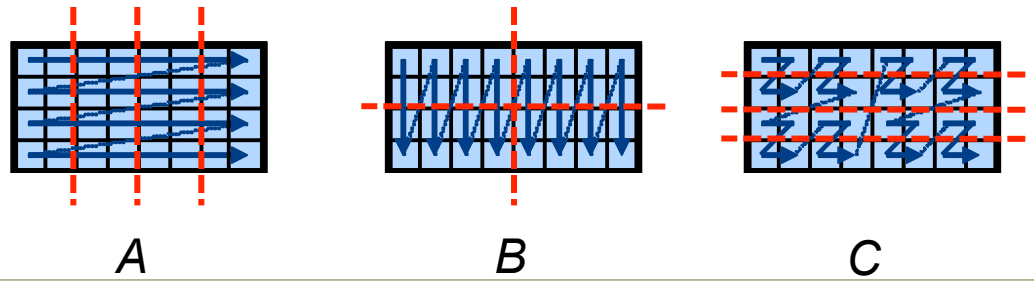
```
forall i in B.domain do B[i] = i/10.0;
```

- How many tasks? Where do they execute?
- How is the iteration space divided between the tasks?

Q2: How are parallel zippered loops implemented?

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

- Particularly given that the iterands might have incompatible distributions, memory layouts, and parallelization strategies



Copyright 2015 Cray Inc.

Summary of this Section

- **Chapel avoids locking crucial implementation decisions into the language specification**
  - local and distributed parallel array implementations
  - parallel loop scheduling policies
  - target architecture models

- **Instead, these can be…**
  …specified in the language by an advanced user
  …swapped between with minimal code changes

- **The result cleanly separates the roles of domain scientist, parallel programmer, and compiler/runtime**

Copyright 2015 Cray Inc.

Summary

Higher-level programming models can help insulate algorithms from parallel implementation details

- yet, without necessarily abdicating control
- Chapel does this via its multiresolution design
  - domain maps, parallel iterators, and locale models are all examples
  - avoids locking crucial policy decisions into the language definition

We believe Chapel can greatly improve productivity

…for current and emerging HPC architectures

…for HPC users and mainstream uses of parallelism at scale

Copyright 2015 Cray Inc.

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:

- Documentation of current domain maps:
  http://chapel.cray.com/docs/latest/modules/distributions.html
  http://chapel.cray.com/docs/latest/modules/layouts.html
  $CHPL_HOME/modules/internal/Default*.chpl
- Technical notes detailing the domain map interface for implementers:
  http://chapel.cray.com/docs/latest/technotes/dsi.html

Copyright 2015 Cray Inc.

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, and URIKA. 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.

Copyright 2015 Cray Inc.


CRAY
CRAY CHAPEL
COMPUTE | STORE | ANALYZE
http://chapel.cray.com chapel_info@cray.com http://github.com/chapel-lang/chapel/