Chapel Language Features for Hierarchical Tiling and Exascale Architectures

Brad Chamberlain, Chapel Team, Cray Inc.

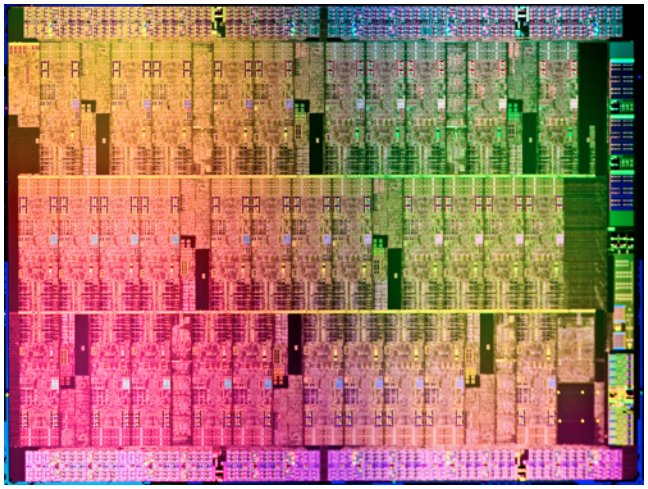
SIAM PP14, MS10: Hierarchical and Iteration Space Tiling

February 19th, 2014

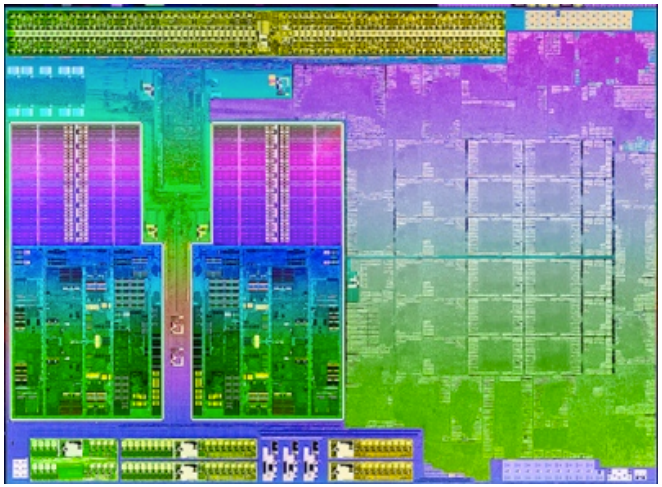
Safe Harbor Statement

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.

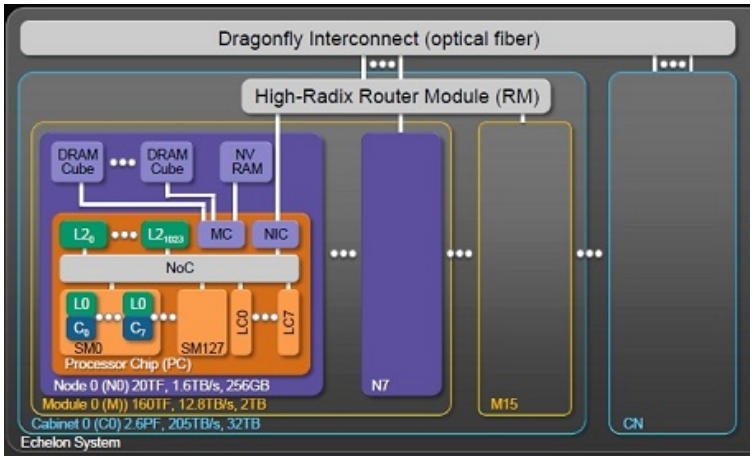
Prototypical Next-Gen Processor Technologies



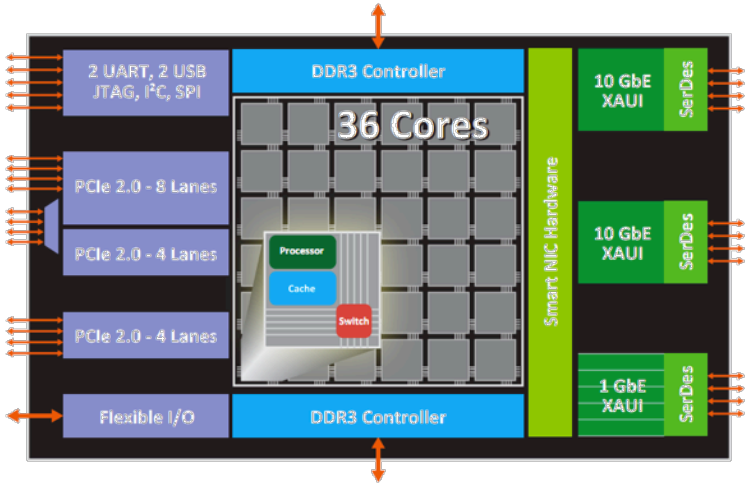
Intel MIC



AMD Trinity



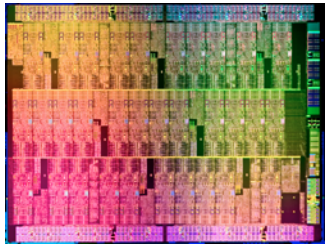
Nvidia Echelon

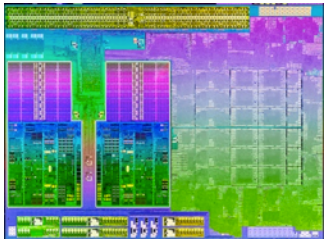


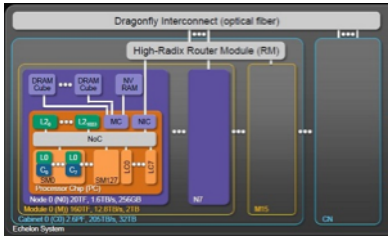
Tilera Tile-Gx

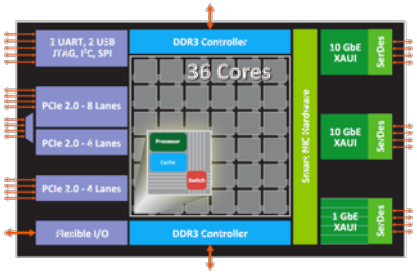
http://download.intel.com/pressroom/images/Aubrey_Isle_die.jpg http://www.zdnet.com/amds-trinity-processors-take-on-intels-ivy-bridge-3040155225/
http://insidehpc.com/2010/11/26/nvidia-reveals-details-of-echelon-gpu-designs-for-exascale/ http://tilera.com/sites/default/files/productbriefs/Tile-Gx%203036%20SB012-01.pdf

General Trends in These Architectures









- Increased hierarchy and/or sensitivity to locality
- Potentially heterogeneous processor/memory types

⇒ **Next-gen programmers will have a lot more to think about at the node level than in the past**

Why is there an exascale programming crisis?

Because adopted HPC programming models…

…have poor support for parallel work decomposition and scheduling
(for this minisymposium, serial/parallel/distributed tiled iterators)

…have poor support for array layouts and distributed data structures
(tiling-oriented data layouts and distributions)

…tend to be too tied to the architectural capabilities they target
(portability challenges in the face of new / divergent architectures)

Chapel differs in that it provides frameworks for end-users to define such policies and abstractions for using them.

What is Chapel?

- **An emerging parallel programming language**
  - Design and development led by Cray Inc.
    - in collaboration with academia, labs, industry
  - Initiated under the DARPA HPCS program

- **A work-in-progress**

- **Chapel's overall goal: Improve programmer productivity**
  - Improve the programmability of parallel computers
  - Match or beat the performance of current programming models
  - Support better portability than current programming models
  - Improve the robustness of parallel codes

Chapel's Implementation

- **Being developed as open source at SourceForge**
- **Licensed as BSD software**
- **A Community Effort**
  - version 1.8 saw 19 developers from 8 organizations and 5 countries
- **Target Architectures:**
  - multicore desktops and laptops
  - commodity clusters and the cloud
  - HPC systems from Cray and other vendors
  - *in-progress:* exascale-era architectures

What does "Productivity" mean to you?

Recent Graduate:

"something similar to what I used in school: Python, Matlab, Java, …"

Seasoned HPC Programmer:

"that sugary stuff which I don't need because I ~~was born to suffer~~"

want full control/performance

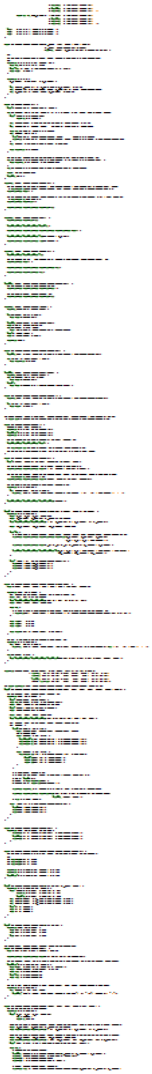
Computational Scientist:

"something that lets me focus on my parallel computational algorithms without having to wrestle with architecture-specific details"

Chapel Team:

"something that lets the computational scientist express what they want, without taking away the control an HPC programmer would want, implemented in a language as attractive as recent graduates want."

LULESH in Chapel



LULESH in Chapel

1288 lines of source code

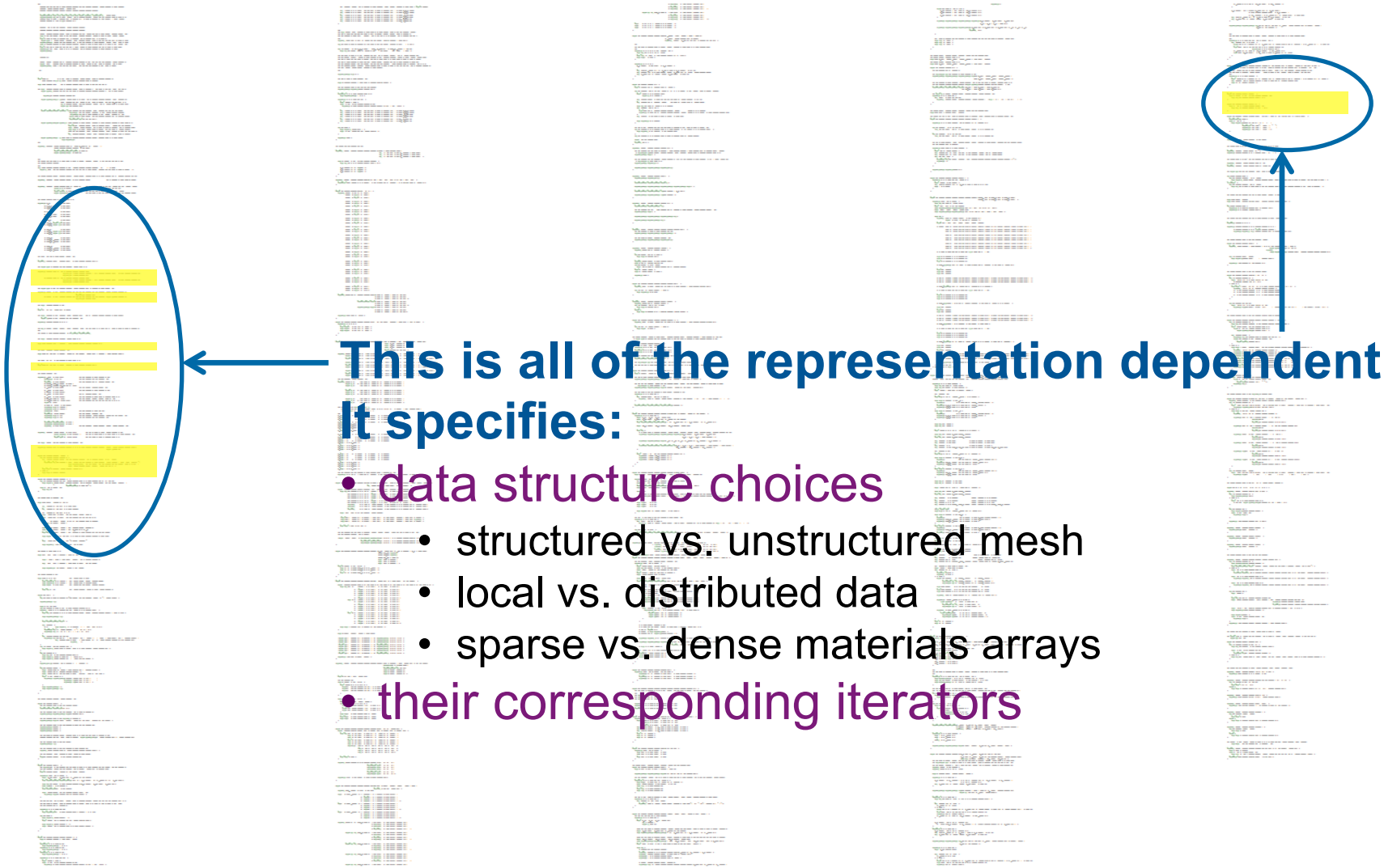
plus 266 lines of comments

487 blank lines

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

This is trunk/test/release/examples/benchmarks/lulesh/*.chpl in the SourceForge repository, as of r22745 (2/16/14).

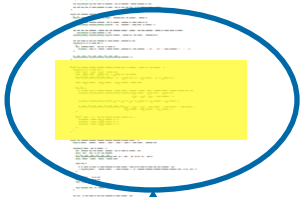
LULESH in Chapel



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
- their corresponding 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

All of this is independent of:

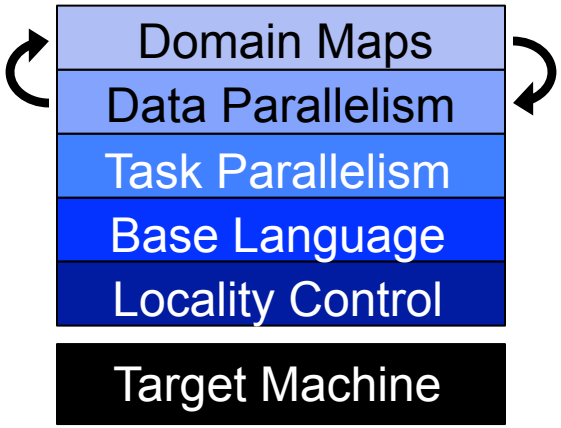
- structured vs. unstructured mesh
- shared vs. distributed data
- sparse vs. dense representation

Multiresolution Design

***Multiresolution Design:* Support multiple tiers of features**

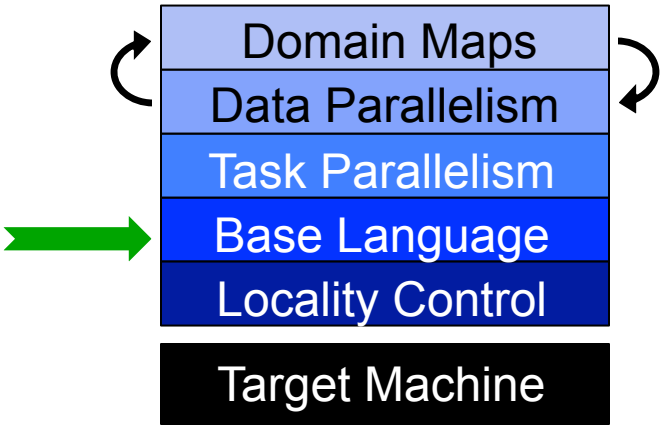
- higher levels for programmability, productivity
- lower levels for greater degrees of control

Chapel language concepts



- build the higher-level concepts in terms of the lower
- permit the user to intermix layers arbitrarily

Base Language Features



Sample Base Language Features (from LULESH)

CLU-style iterators

```
//
// yield each node corresponding to an element, along with its index
//
iter elemToNodesTuple(e) {
  for i in 1..nodesPerElem do
    yield (elemToNode[e][i], i);
}
```

inferred types

tuples

```
//
// sample invocation from within integrateStressForElems()
//
for (noi, t) in elemToNodesTuple(k) {
  …
}
```

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;
}
```

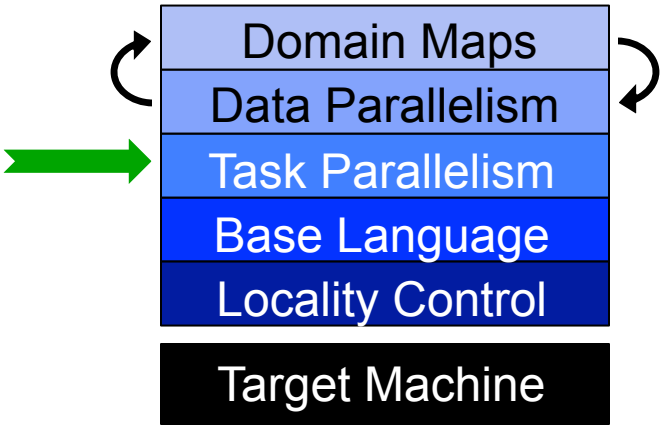
first-class domains (index sets) and supporting operations

```
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)…
```

Task Parallel Features



Sample Task Parallel Concept: Coforall Loops

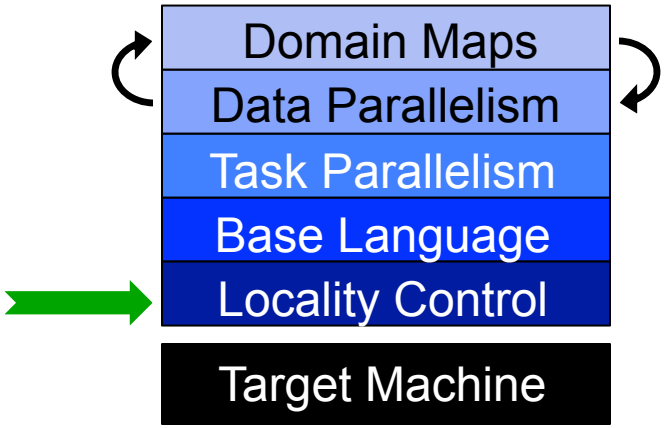
```
coforall t in 0..#numTasks {
  writeln("Hello from task ", t, " of ", numTasks);
} // implicit join of the coforall tasks here

writeln("All tasks done");
```

Sample output:

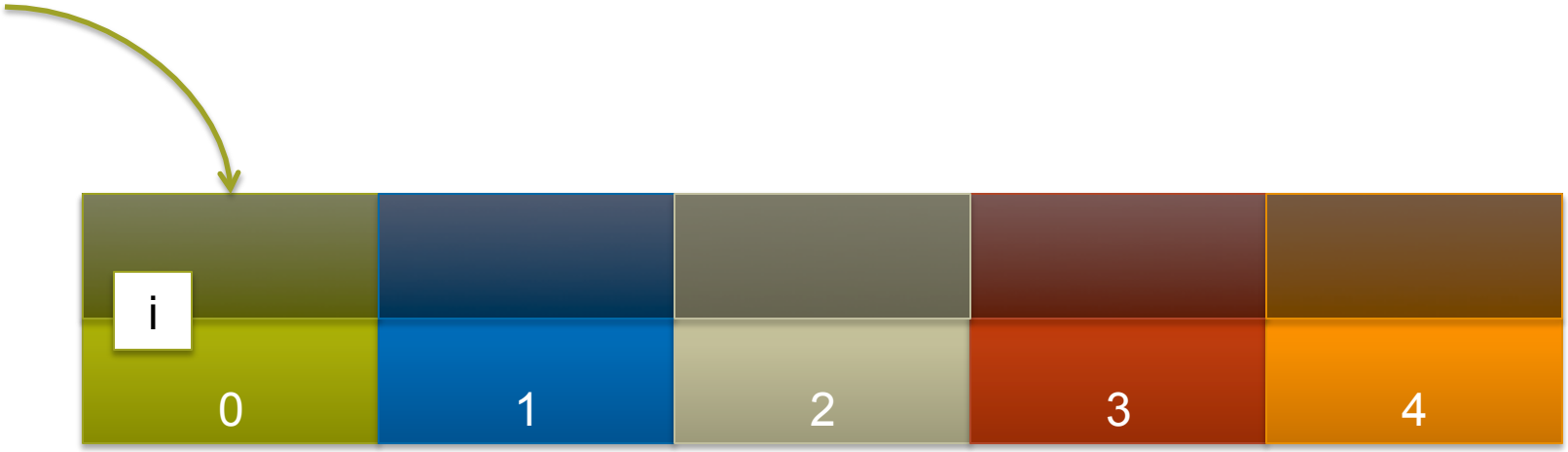
```
Hello from task 2 of 4
Hello from task 0 of 4
Hello from task 3 of 4
Hello from task 1 of 4
All tasks done
```

Locality Control Features



Locality Control and Scoping

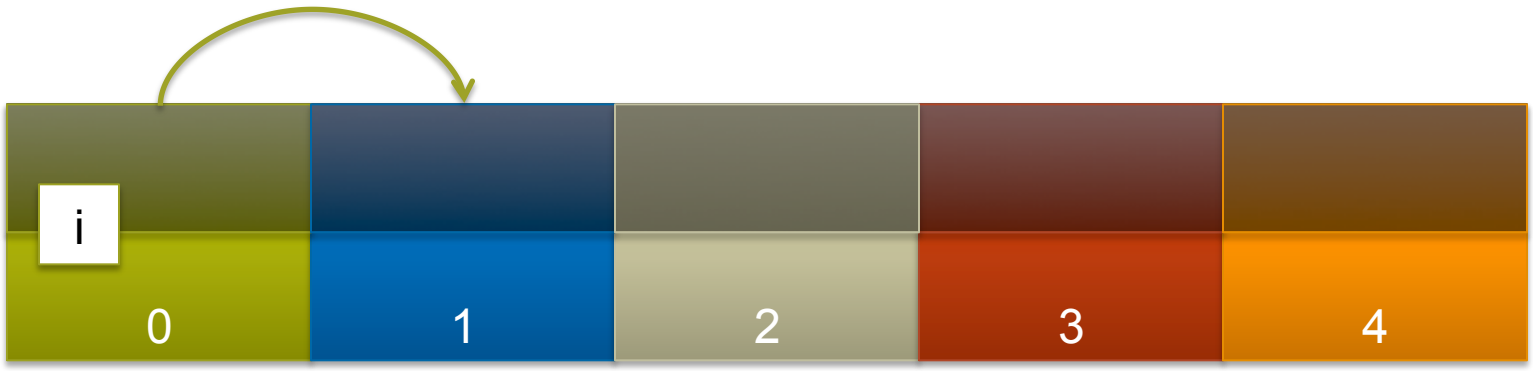
```
var i: int;
```



Locales (think: “compute nodes”)

Locality Control and Scoping

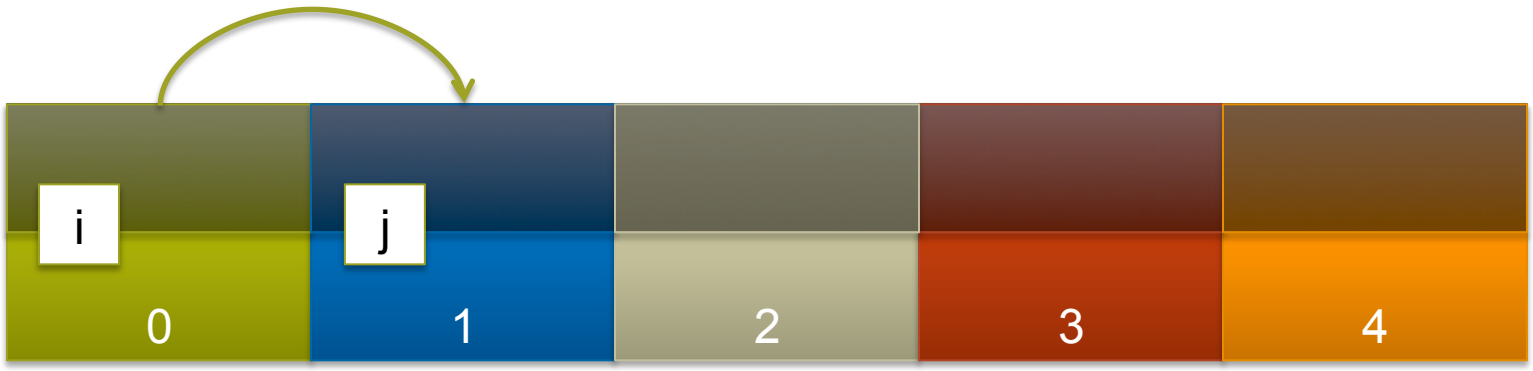
```
var i: int;
on Locales[1] {
```



Locales (think: “compute nodes”)

Locality Control and Scoping

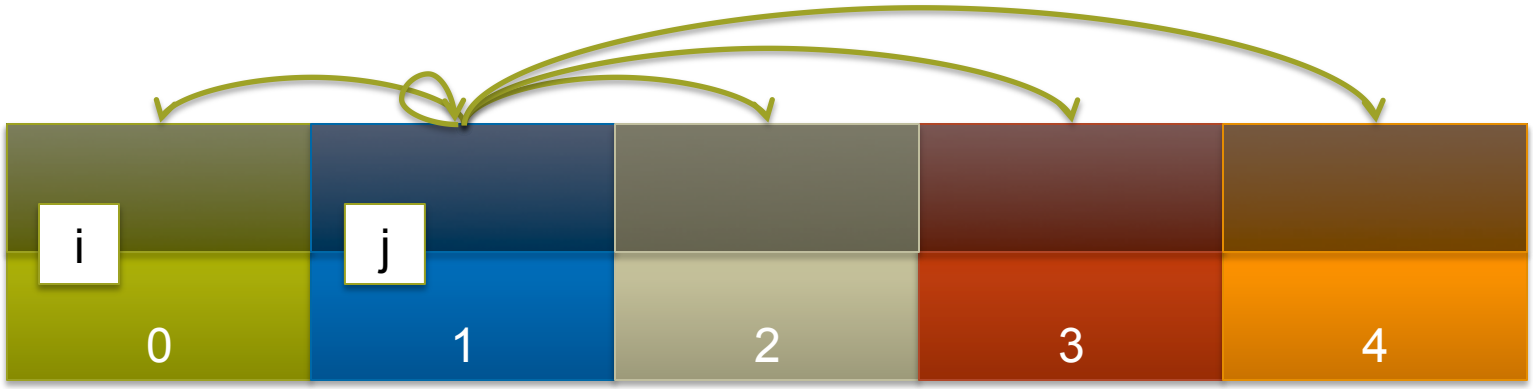
```
var i: int;
on Locales[1] {
  var j: int;
```



Locales (think: “compute nodes”)

Locality Control and Scoping

```
var i: int;
on Locales[1] {
  var j: int;
  coforall loc in Locales {
    on loc {
```

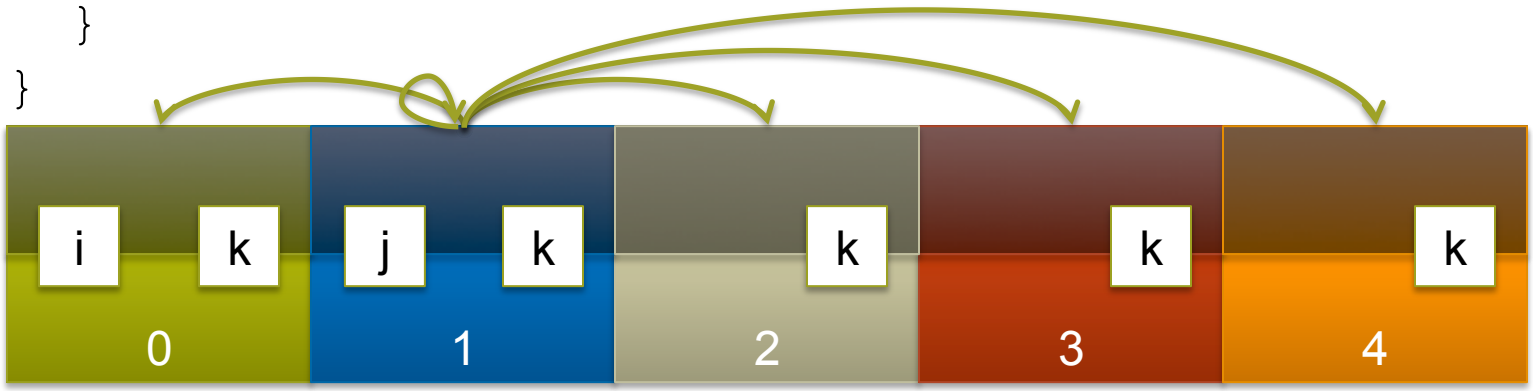


Locales (think: “compute nodes”)

Locality Control and Scoping

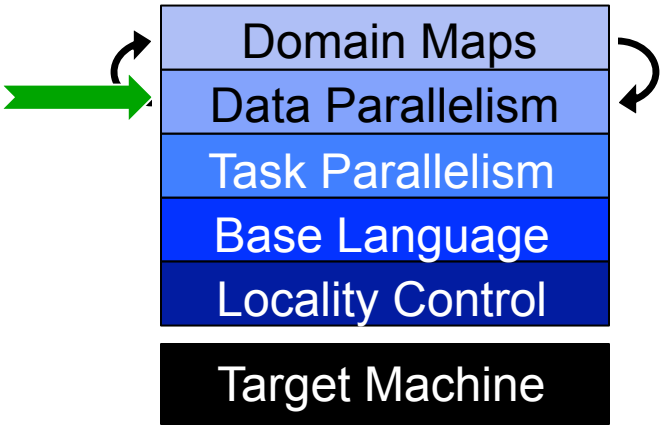
```
var i: int;
on Locales[1] {
  var j: int;
  coforall loc in Locales {
    on loc {
      var k: int;

        // within this scope, i,j,k can be referenced, as in any language;
        // the communication is managed by the compiler and runtime
    }
  }
}
```



Locales (think: "compute nodes")

Data Parallel Features

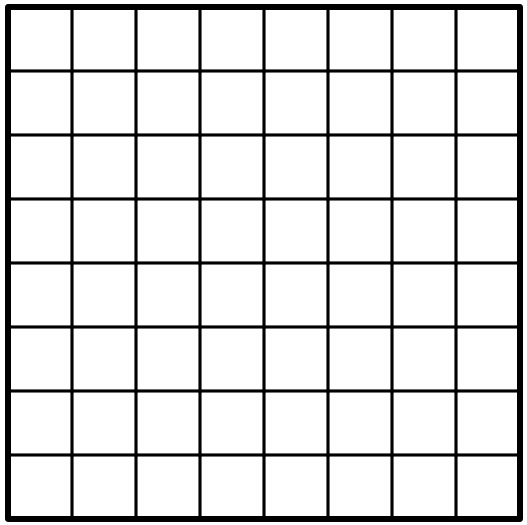


Data Parallelism in LULESH (Structured)

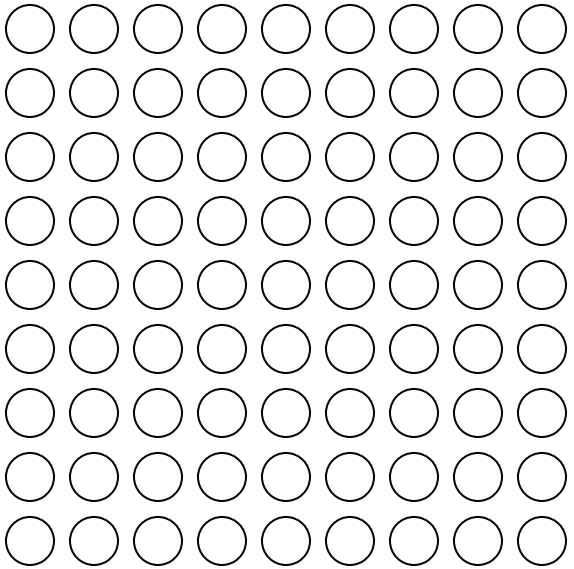
```
const Elems = {0..#elemsPerEdge, 0..#elemsPerEdge},
      Nodes = {0..#nodesPerEdge, 0..#nodesPerEdge};

var determ: [Elems] real;

forall k in Elems { …determ[k]… }
```



Elems



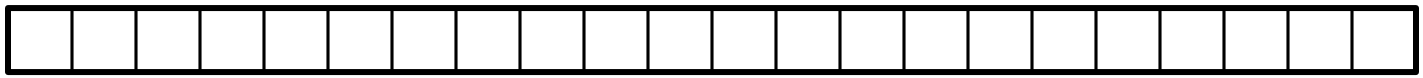
Nodes

Data Parallelism in LULESH (Unstructured)

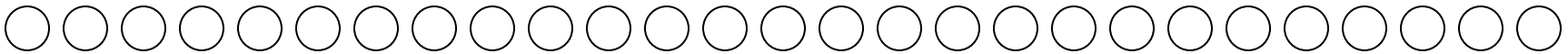
```
const Elems = {0..#numElems},
      Nodes = {0..#numNodes};

var determ: [Elems] real;

forall k in Elems { …determ[k]… }
```



Elems



Nodes

Data Parallelism: Multiresolution in Action

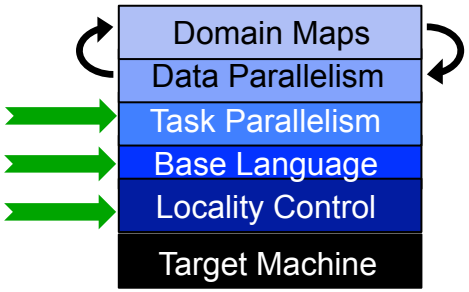
```
forall k in Elems { … }
```

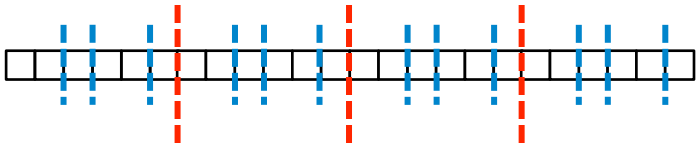
Q: How are data parallel loops implemented?

(how many tasks? running where? owning which iterations?)

A: Via parallel iterators defined using task/locality features:

```
//
//  statically blocked iterator across nodes and cores
//
iter BlockDom.these(…) {
  coforall loc in Locales do
    on loc do
      coforall tid in here.numCores do
        yield computeMyChunk(loc.id, tid);
}
```





Data Parallel Iterators: Multiresolution in Action

Q: How are domains and arrays implemented?

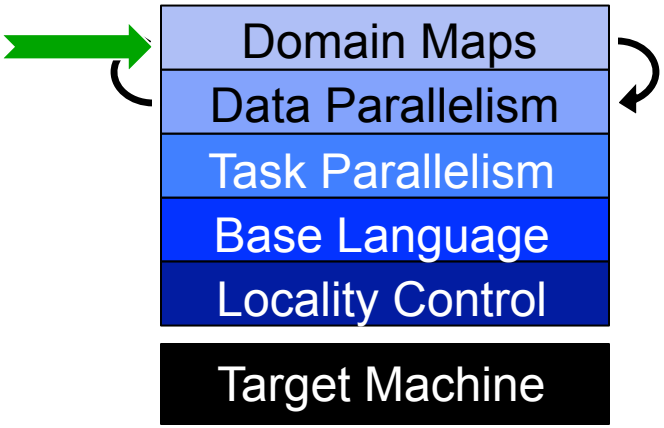
(distributed or local? distributed how? stored in memory how?)

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

var determ: [Elems] real;
```

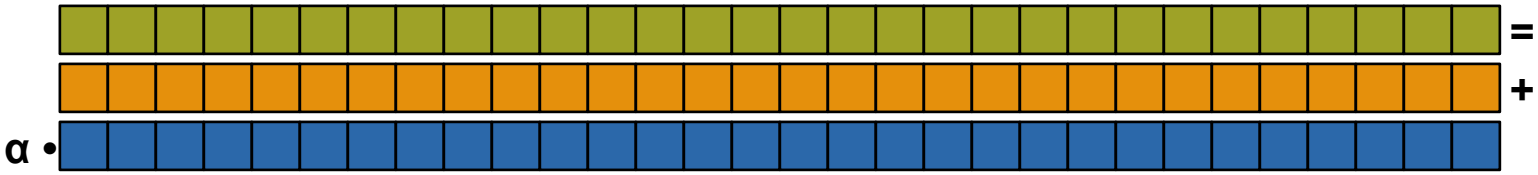
A: Via domain maps…

Domain Maps



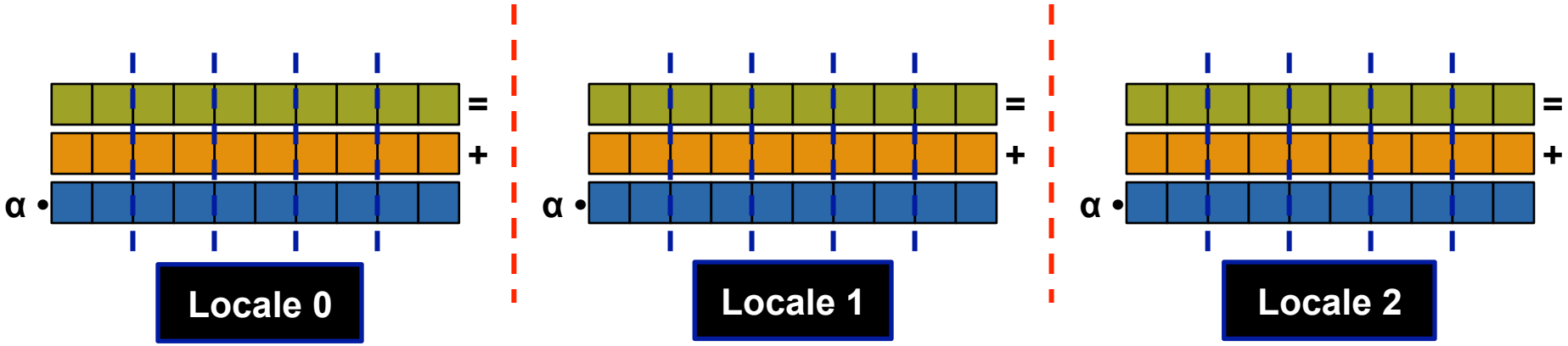
Domain Maps: Concept

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:

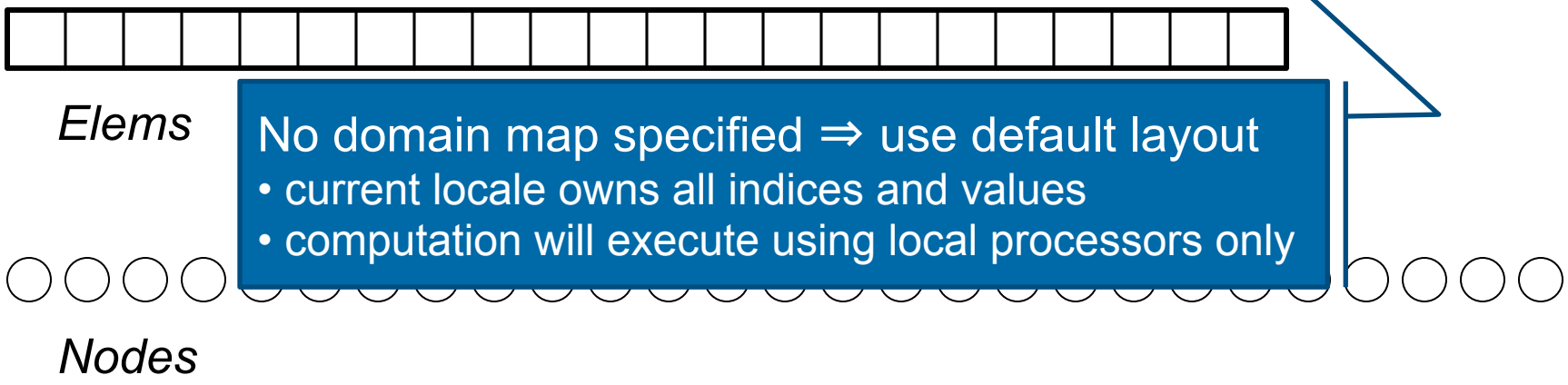


LULESH Data Structures (local)

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

var determ: [Elems] real;

forall k in Elems { … }
```



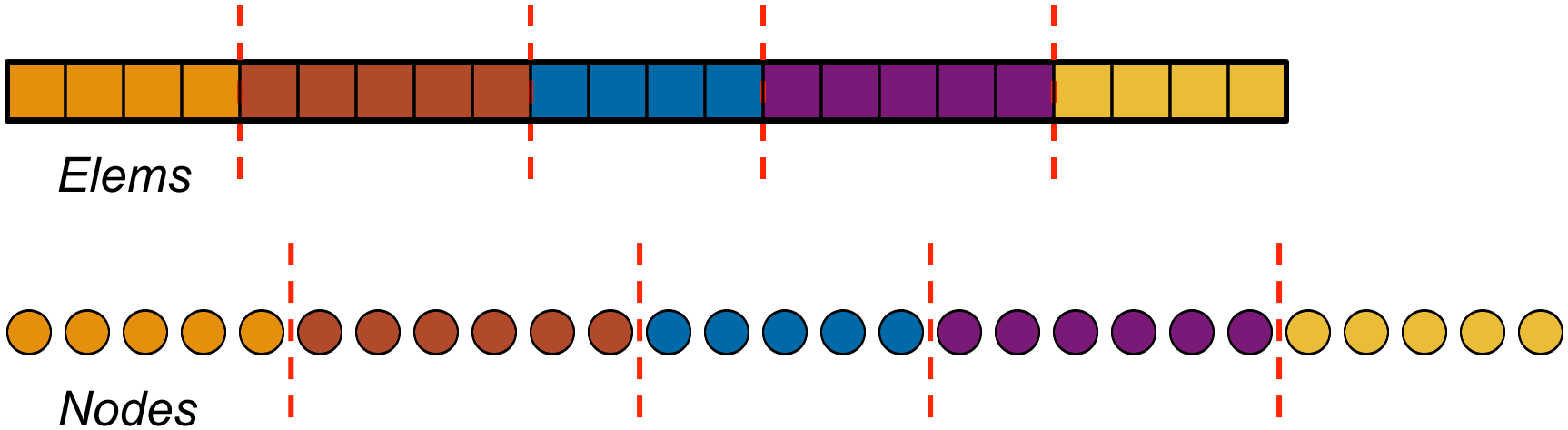
No domain map specified ⇒ use default layout
- current locale owns all indices and values
- computation will execute using local processors only

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 { … }
```

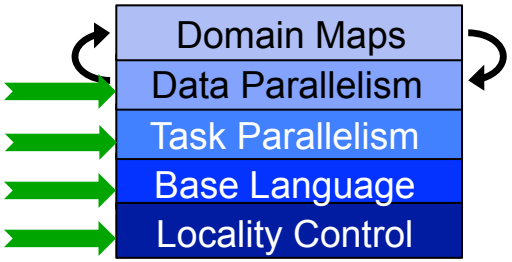
Elems

Nodes

Chapel's Domain Map Philosophy

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

2. **Advanced users can write their own domain maps in Chapel**
   - to cope with 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

So, where does that put us?

We can…

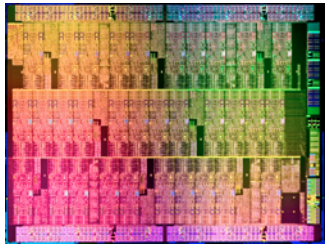
…write parameterized tiled iterators (serial or parallel)

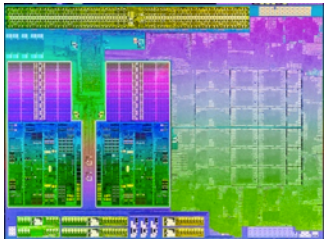
…distribute and lay out arrays in ways that support tiling

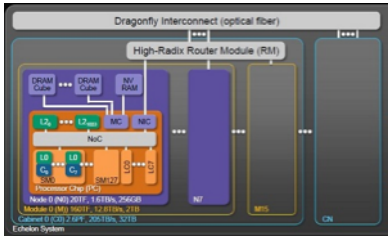
…access these abstractions using high-level abstractions

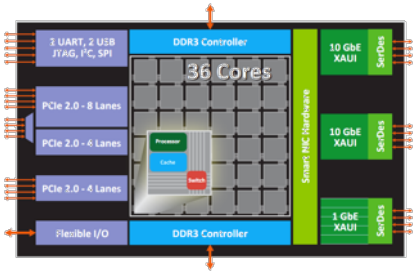
…switch implementations without rewriting computations

OK, but what about those future architectures?









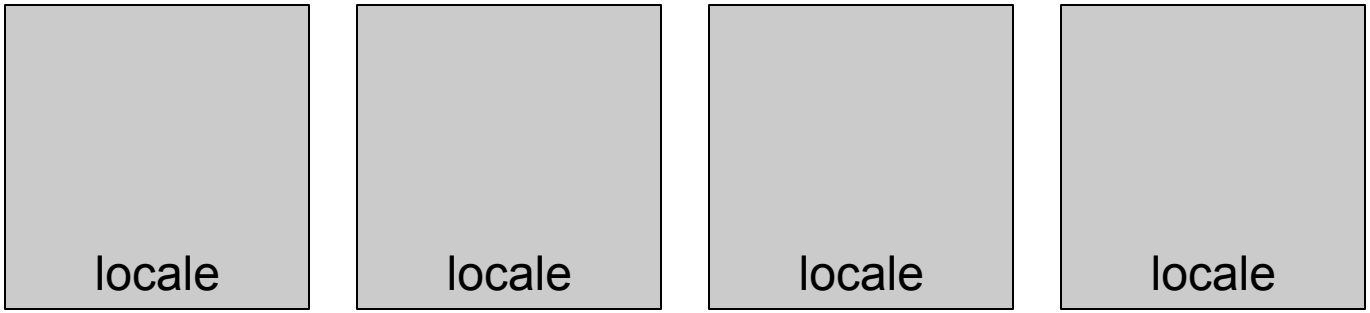
Our approach: **Hierarchical Locales**

- extends multiresolution philosophy to architectural modeling

Traditional Locales

Concept:

- Traditionally, Chapel has supported a 1D array of locales
  - users can reshape/slice to suit their computation's needs

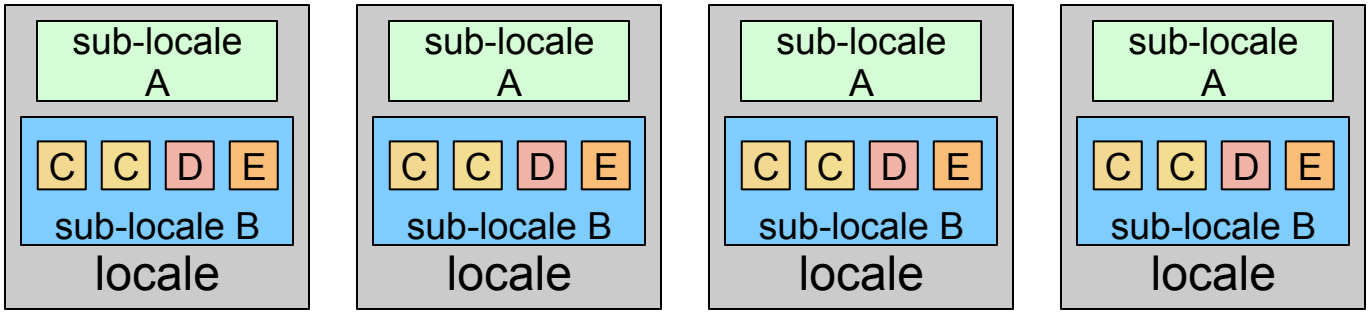


- Supports horizontal (inter-node) locality well
  - but not vertical (intra-node)

Recent Work: Hierarchical Locales

Concept:

- Support locales within locales to describe architectural sub-structures within a node (e.g., memories, processors)



- As with top-level locales, *on-clauses* and *domain maps* map tasks and variables to sub-locales
- Locale models are defined using Chapel code

Summary

Chapel's multiresolution philosophy allows users to write…

…iterators

- that support patterns like tiling
- and parallel scheduling policies

…domain maps

- that support layouts within a memory
- and distributions of data between memories
- as well as operations on those structures

…hierarchical locale models

- that permit emerging architectures to be modeled within Chapel
- and support the specification of policies for mapping to that architecture

…permitting users to decouple their algorithms from crucial policy decisions, making computations more flexible and portable to future architectures

Implementation Status -- Version 1.8.0 (Oct 2013)

Overall Status:

- Most features work at a functional level
  - some need to be improved or re-implemented (e.g., OOP)
- Many performance optimizations remain
  - particularly for distributed memory (multi-locale) execution

This is a good time to:

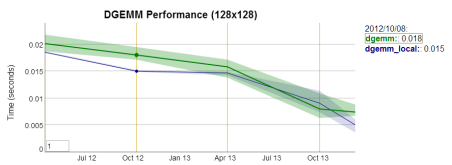
- Try out the language and compiler
- Use Chapel for non-performance-critical projects
- Give us feedback to improve Chapel
- Use Chapel for parallel programming education

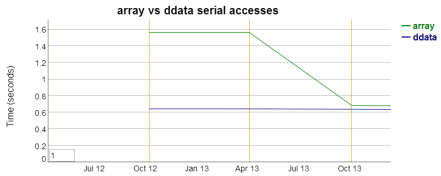
A Note on Performance

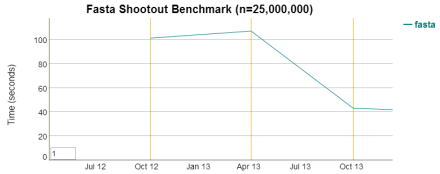
- **The generality featured in this talk is precisely why, today, Chapel's performance lags traditional approaches**
  - Where other languages have full control over policies, Chapel is open
  - "Support the general case, optimize for the common case"

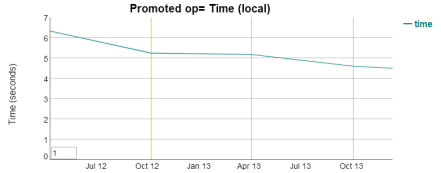
- **Yet, this lag is not inherent, just a matter of [im]maturity**
  - Chapel is designed so that ultimately it will perform well

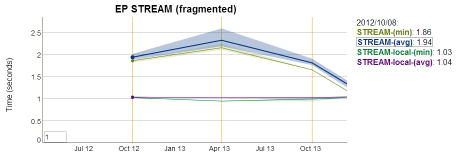
- **Execution times are improving with each release**

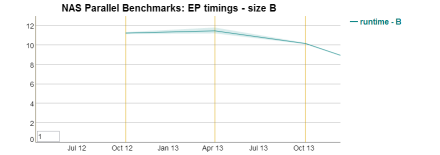


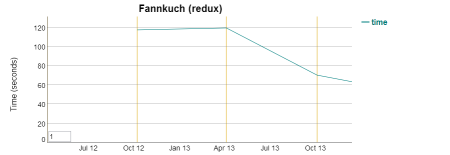


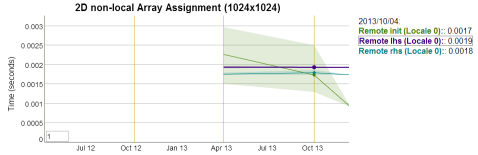








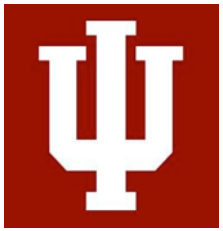




The Cray Chapel Team (Summer 2013)

Chapel…

…is a collaborative effort — join us!



For More Information: Online Resources

Chapel project page: http://chapel.cray.com

- overview, papers, presentations, language spec, …

Chapel SourceForge page: https://sourceforge.net/projects/chapel/

- release downloads, public mailing lists, code repository, …

Mailing Aliases:

- chapel_info@cray.com: contact the team at Cray
- chapel-announce@lists.sourceforge.net: announcement list
- chapel-users@lists.sourceforge.net: user-oriented discussion list
- chapel-developers@lists.sourceforge.net: developer discussion
- chapel-education@lists.sourceforge.net: educator discussion
- chapel-bugs@lists.sourceforge.net: public bug forum

For More Information: Suggested Reading

Overview Papers:

- *The State of the Chapel Union* [slides], Chamberlain, Choi, Dumler, Hildebrandt, Iten, Litvinov, Titus. CUG 2013, May 2013.
  - *a high-level overview of the project summarizing the HPCS period*
- *A Brief Overview of Chapel*, Chamberlain (pre-print of a chapter for *A Brief Overview of Parallel Programming Models*, edited by Pavan Balaji, to be published by MIT Press in 2014).
  - *a more detailed overview of Chapel's history, motivating themes, features*

Blog Articles:

- *[Ten] Myths About Scalable Programming Languages*, Chamberlain. IEEE Technical Committee on Scalable Computing (TCSC) Blog, (https://www.ieeetcsc.org/activities/blog/), April-November 2012.
  - *a series of technical opinion pieces designed to rebut standard arguments against the development of high-level parallel languages*

Chapel: the next five years

- **Harden Prototype to Production-grade**
  - Performance Optimizations
  - Add/Improve Lacking Features
- **Target more complex/modern compute node types**
  - e.g., CPU+GPU, Intel MIC, …
- **Continue to grow the user and developer communities**
  - including nontraditional circles: desktop computing, “big data”, …
  - transition Chapel from Cray-driven to community-governed
- **Grow the team at Cray**
  - currently seeking to hire a manager for the program and team

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/