Language Constructs for Data Locality:
Moving Policy Decisions from Language Definition to User Space

Brad Chamberlain, Chapel Team, Cray Inc.

PADAL Workshop, Lugano Switzerland

April 28th, 2014

Three Language Concepts for Taming Data Locality

~~Language Constructs for Data Locality:~~

~~Moving Policy Decisions from Language Definition to User Space~~

Brad Chamberlain, Chapel Team, Cray Inc.

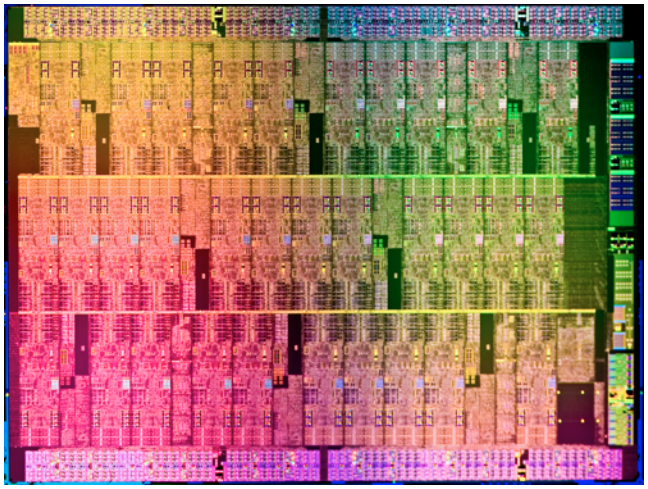
PADAL Workshop, Lugano Switzerland

April 28th, 2014

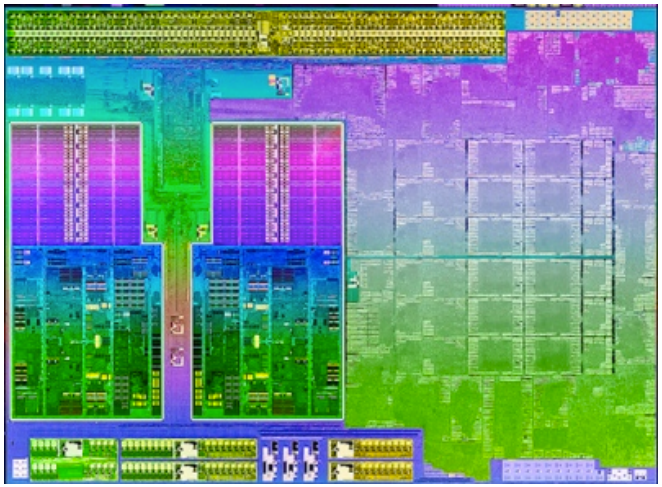
Safe Harbor Statement

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.

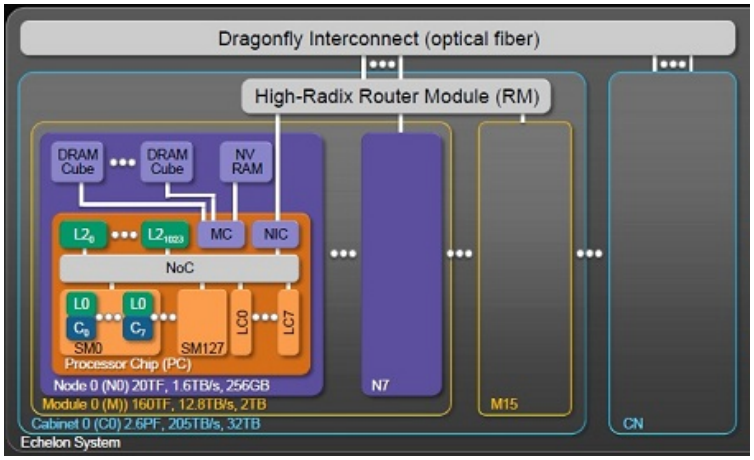
Prototypical Next-Gen Processor Technologies



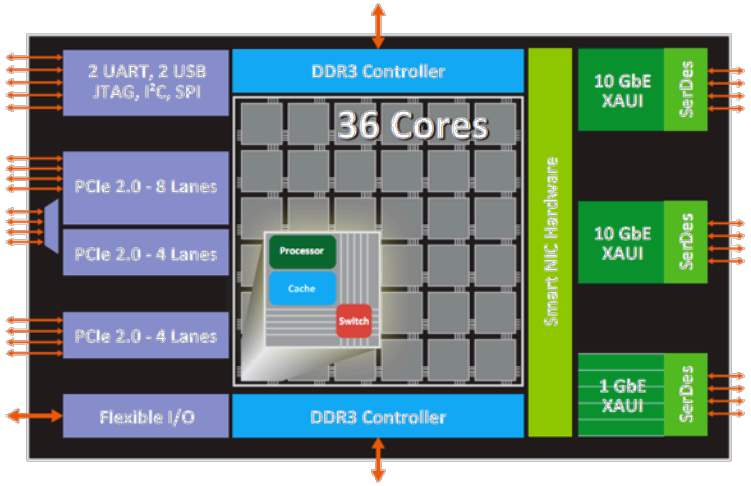
Intel MIC



AMD APU



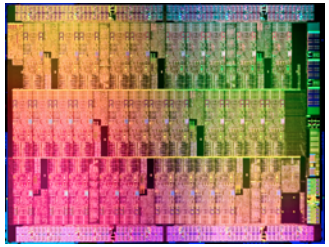
Nvidia Echelon

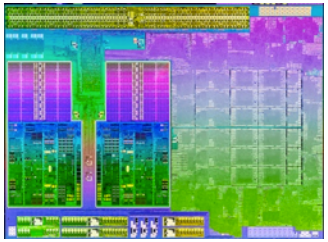


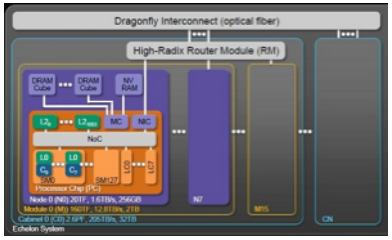
Tilera Tile-Gx

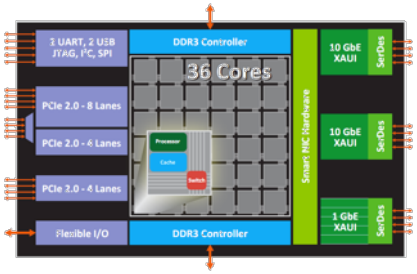
http://download.intel.com/pressroom/images/Aubrey_Isle_die.jpg http://www.zdnet.com/amds-trinity-processors-take-on-intels-ivy-bridge-3040155225/
http://insidehpc.com/2010/11/26/nvidia-reveals-details-of-echelon-gpu-designs-for-exascale/ http://tilera.com/sites/default/files/productbriefs/Tile-Gx%203036%20SB012-01.pdf

Why do we need data locality control?









Emerging processor designs…

…are increasingly locality-sensitive

…potentially have multiple processor/memory types

Data Locality Control in Current HPC Models

Q: Why are current HPC models lacking w.r.t. data locality?

A: Because they…

…lock key data locality policies into the language

- e.g., array layouts, parallel scheduling

…lack support for users to create new policy abstractions

…expose too much about their target architectures

In Chapel, we're striving to improve upon this status quo

"How can we define a language that supports high level abstractions and enables users to plug in their own implementations?"

What is Chapel?

- **An emerging parallel programming language**
  - Design and development led by Cray Inc.
    - in collaboration with academia, labs, industry
- **A work-in-progress**
- **Goal: Improve productivity of parallel programming**

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 that I can't use because I require full control to ensure good performance"

Computational Scientist:

"something that lets me express my parallel computations 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 the HPC programmer wants, implemented in a language as attractive as recent graduates want."

Chapel's Implementation

- **Being developed as open source at SourceForge**
  - Licensed as BSD software

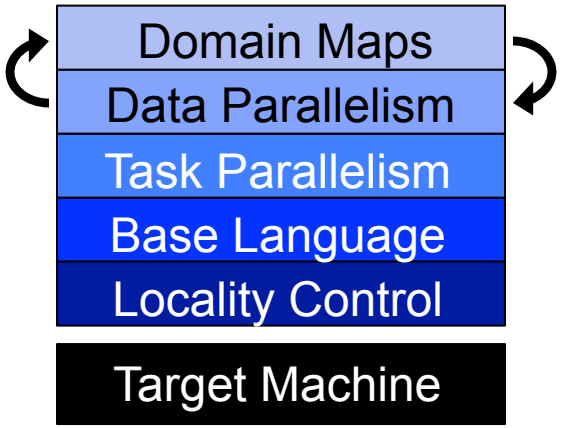
- **Portable design and implementation, targeting:**
  - multicore desktops and laptops
  - commodity clusters and the cloud
  - HPC systems from Cray and other vendors
  - *in-progress:* exascale-era architectures

Multiresolution Design

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

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

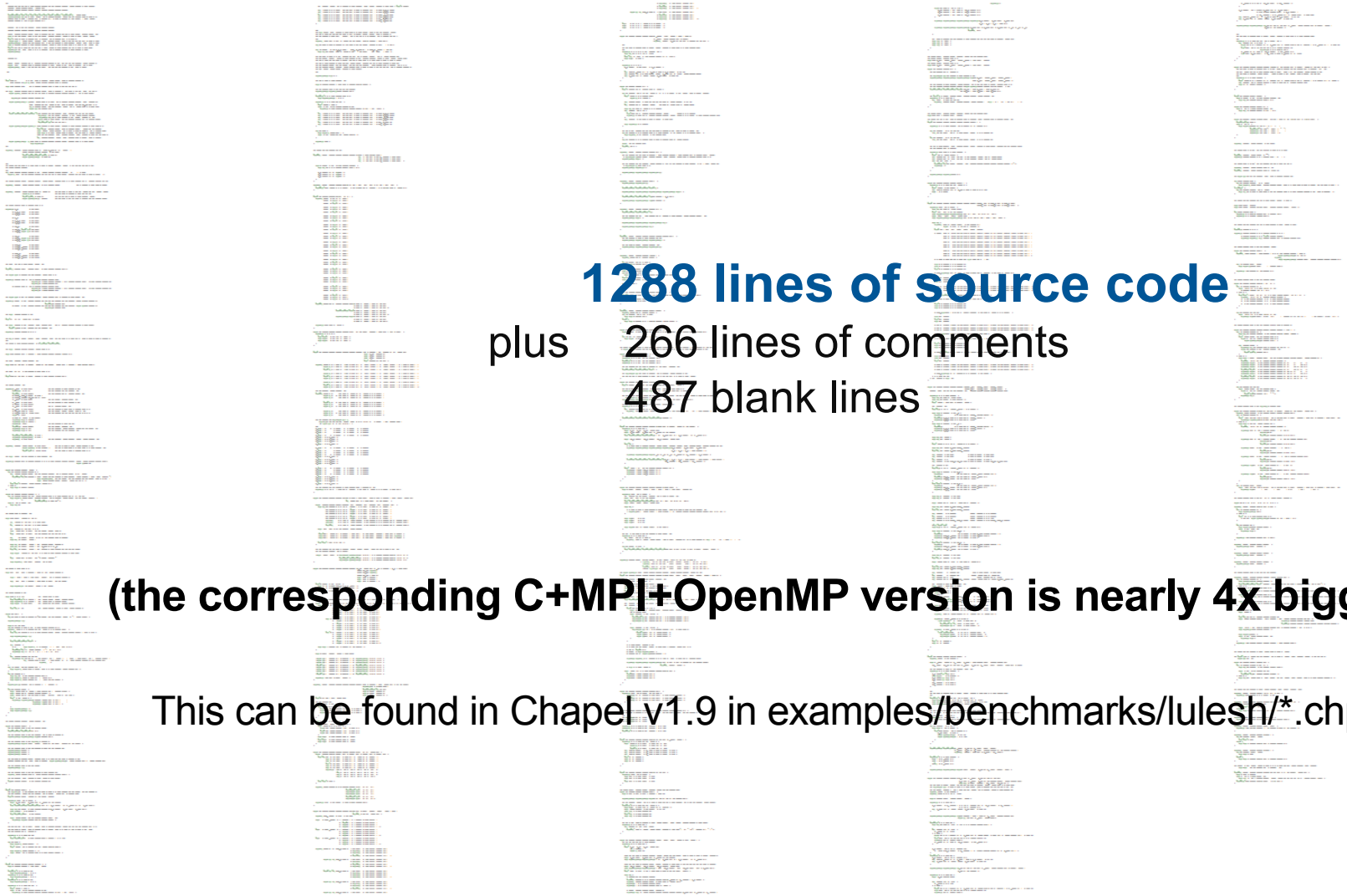
Chapel language concepts



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

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 the only representation-dependent code. It specifies:

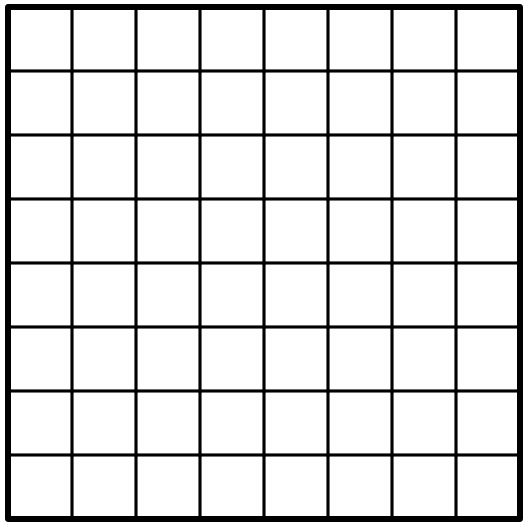
- data structure choices
  - structured vs. unstructured mesh
  - local vs. distributed data
  - sparse vs. dense materials arrays
- a few supporting iterators

Data Parallelism in LULESH (Structured)

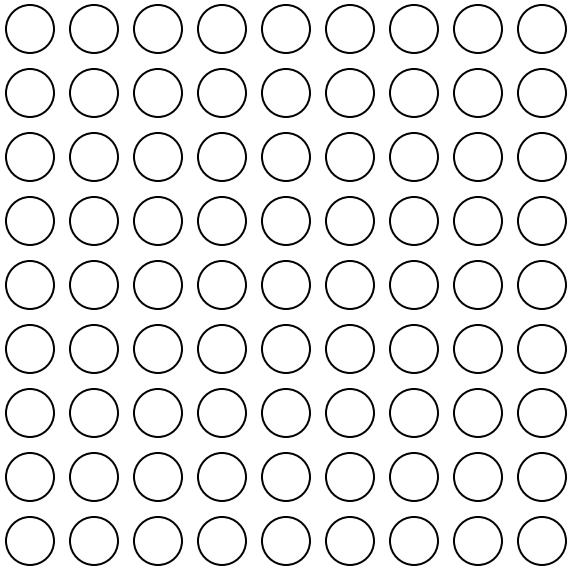
```
const Elems = {0..#elemsPerEdge, 0..#elemsPerEdge},
      Nodes = {0..#nodesPerEdge, 0..#nodesPerEdge};

var determ: [Elems] real;


forall k in Elems { …determ[k]… }
```



Elems



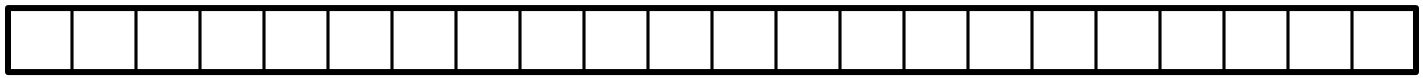
Nodes

Data Parallelism in LULESH (Unstructured)

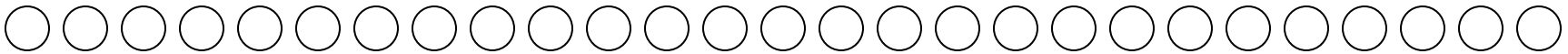
```
const Elems = {0..#numElems},
      Nodes = {0..#numNodes};

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

forall k in Elems { …determ[k]… }
```



Elems



Nodes

Implementing Domains and Arrays

Q: How are domains and arrays implemented?

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

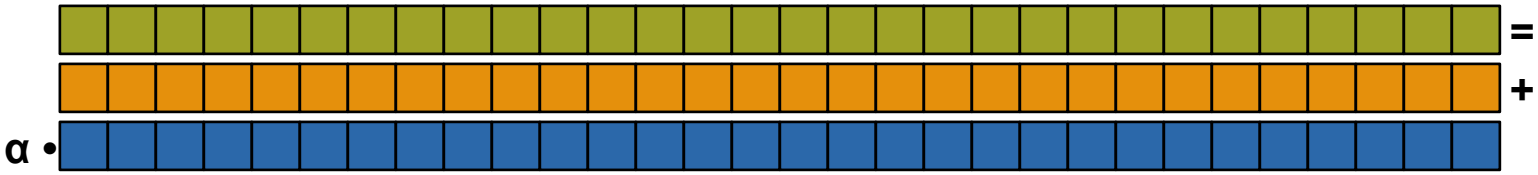
```
const Elems = {0..#numElems},
      Nodes = {0..#numNodes};

var determ: [Elems] real;
```

A: Via Feature #1 (domain maps)…

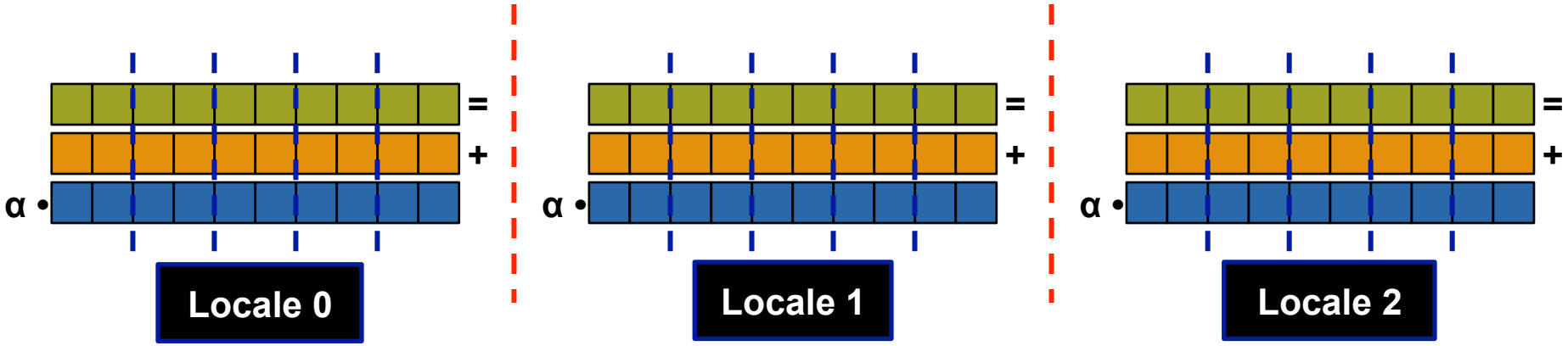
Domain Maps: Concept

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:

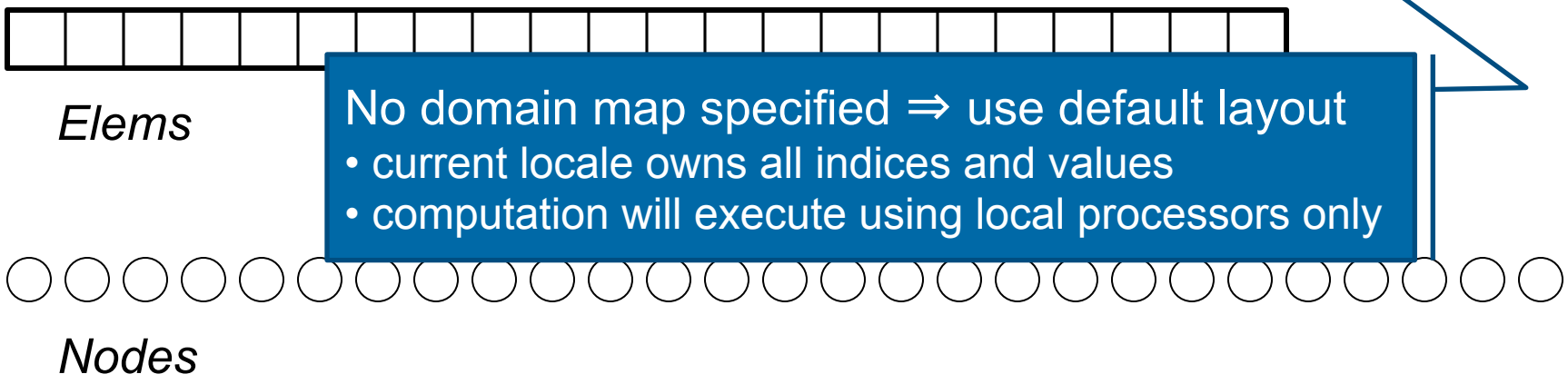


LULESH Data Structures (local)

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

var determ: [Elems] real;

forall k in Elems { … }
```



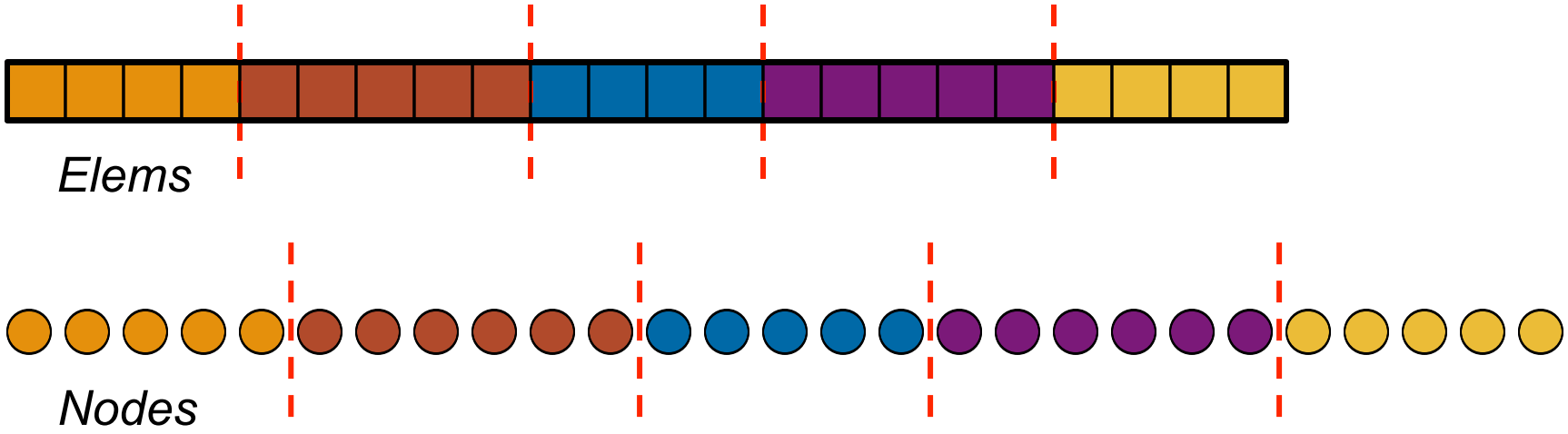
No domain map specified ⇒ use default layout
- current locale owns all indices and values
- computation will execute using local processors only

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 { … }
```

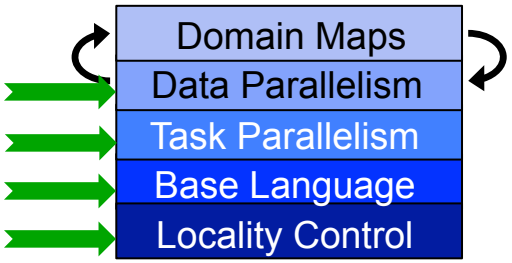
Elems

Nodes

Chapel's Domain Map Philosophy

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

2. **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 ensure that the framework works and works well

Domain Map Descriptors

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

Domain Maps Summary

- **Data locality requires mapping arrays to memory well**
  - distributions between distinct memories
  - layouts within a single memory

- **Most languages define a single data layout & distribution**
  - where the distribution is often the degenerate "everything's local"

- **Domain maps…**

  …move such policies into user-space

  …exposing them to the end-user through high-level declarations

```
const Elems = {0..#numElems} dmapped Block(…)
```

Implementing Data Parallel Loops

Q: How are parallel loops implemented?

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

```
forall k in Elems { … }
```

Q2: What about zippered data parallel operations?

(how to reconcile potentially conflicting parallel implementations?)

```
forall (k,d) in zip(Elems, determ) { … }
x += xd * dt;
```

A: Via Feature #2 (leader-follower iterators)…

Leader-Follower Iterators: Definition

- **Chapel defines all forall loops in terms of *leader-follower iterators*:**
  - *leader iterators:* create parallelism, assign iterations to tasks
  - *follower iterators:* serially execute work generated by leader

- **Given…**

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

…*A* is defined to be the *leader*

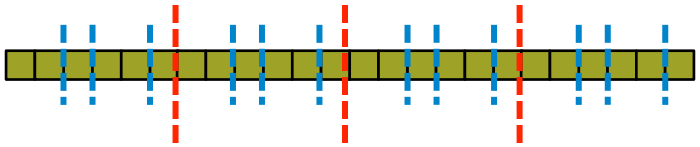
…*A*, *B*, and *C* are all defined to be *followers*

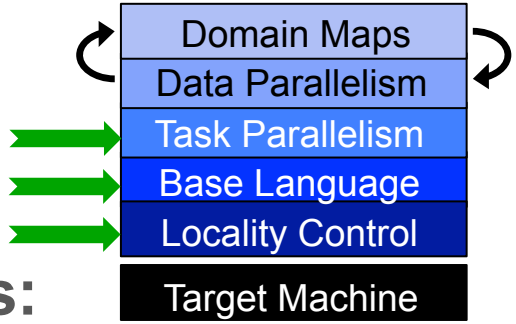
- **Domain maps support default leader-follower iterators**
  - specify parallel traversal of a domain's indices/array's elements
  - typically written to leverage affinity

Writing Leaders and Followers

Leader iterators are defined using task/locality features:

```
iter BlockArr.lead() {
  coforall loc in Locales do
    on loc do
      coforall tid in here.numCores do
        yield computeMyChunk(loc.id, tid);
}
```





Follower iterators simply use serial features:

```
iter BlockArr.follow(work) {
  for i in work do
    yield accessElement(i);
}
```

Leader-Follower Summary

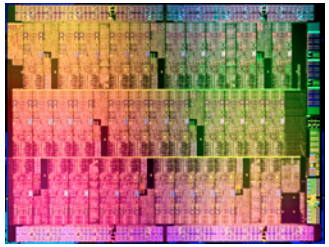
- **Data locality requires parallel loops to execute intelligently**
  - appropriate number and placement of tasks
  - good data-task affinity
- **Most languages define fixed parallel loop styles**
  - where “no parallel loops” is a common choice
- **Leader-follower iterators…**

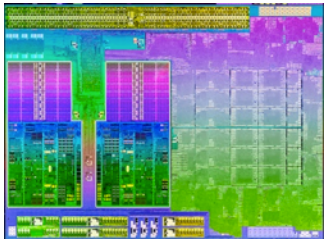
  …move such policies into user-space

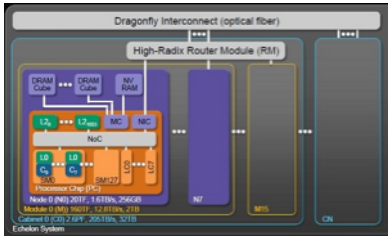
  …expose them to the end-user through data parallel abstractions

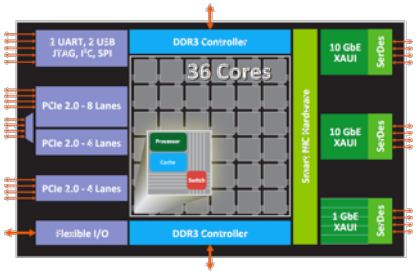
```
forall k in Elems { … }
x += xd * dt;
```

OK, but what about those future architectures?









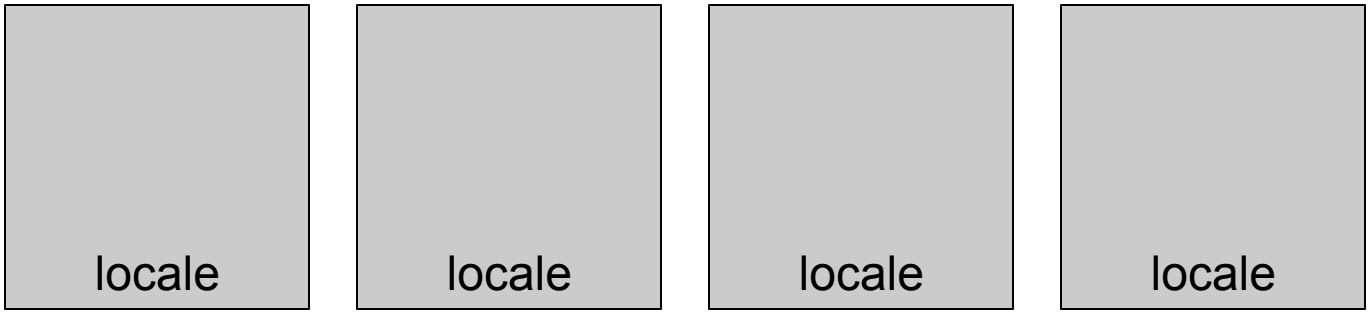
Feature #3 (hierarchical locales)

- extends multiresolution philosophy to architectural modeling

Traditional Locales

Concept:

- Traditionally, Chapel has supported a 1D array of locales

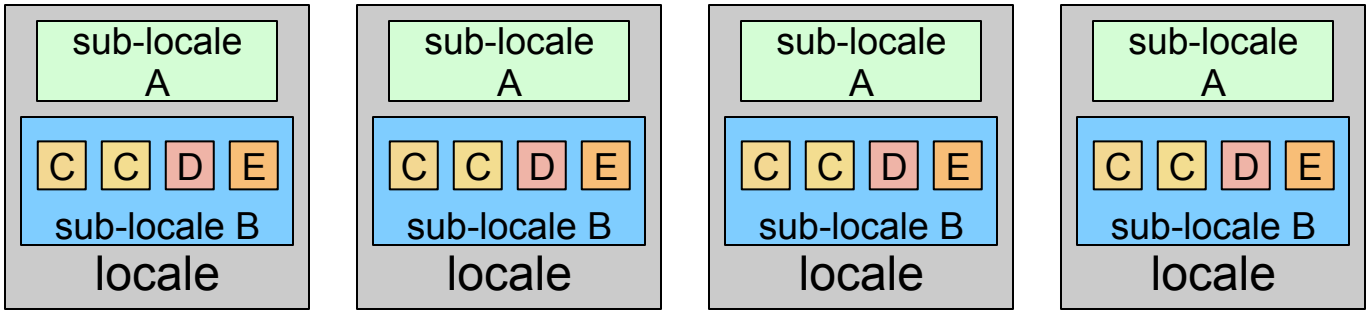


- Supports inter-node locality well, but not intra-node
  - (which, of course, is becoming increasingly important)

Recent Work: Hierarchical Locales

Concept:

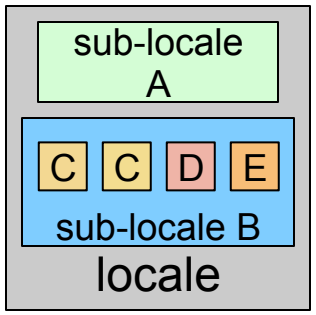
- Support locales within locales to describe architectural sub-structures within a node (e.g., memories, processors)



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

Defining Hierarchical Locales

1) Define the processor's abstract block structure



2) Define how to run a task on any sublocale

3) Define how to allocate/access memory on any sublocale

Hierarchical Locale Summary

- **Data locality requires flexibility w.r.t. future architectures**
  - due to uncertainty in processor design
  - to support portability between approaches
- **Most programming models assume certain features in the target architecture**
  - this is why MPI/OpenMP/UPC/CUDA/… have restricted applicability
- **Hierarchical Locales**

  …move the definition of new architectural models to user space

  …are exposed to the end-user via Chapel's traditional locality features

```
on loc do
  coforall tid in here.numCores do
```

Summary

Chapel's multiresolution philosophy allows users to write…

- …custom array implementations via domain maps
- …custom parallel iterators via leader-follower iterators
- …custom architectural models via hierarchical locales

The result is a language that decouples crucial policies for managing data locality out of the language's definition and into an expert user's hand…

…while making them available to end-users through high-level abstractions

Why a new language?

Q: Why develop a new language rather than a library or language extension?

A: Because…

…having custom syntax presents policies to the end-user more cleanly

…it exposes optimization opportunities to the compiler

…helps with rank-independent indexing, arr-of-struct v. struct-of-array, …

…these concepts are more difficult to write in a traditional HPC language
(due to lack of support for features like type inference, iterators, generics, …)

For More Information on…

…domain maps

User-Defined Distributions and Layouts in Chapel: Philosophy and Framework [slides], Chamberlain, Deitz, Iten, Choi; HotPar'10, June 2010.

Authoring User-Defined Domain Maps in Chapel [slides], Chamberlain, Choi, Deitz, Iten, Litvinov; Cug 2011, May 2011.

…leader-follower iterators

User-Defined Parallel Zippered Iterators in Chapel [slides], Chamberlain, Choi, Deitz, Navarro; PGAS 2011, October 2011.

…hierarchical locales

Hierarchical Locales: Exposing Node-Level Locality in Chapel, Choi; 2nd KIISE-KOCSEA SIG HPC Workshop talk, November 2013.

Status: all of these concepts are in-use in every Chapel program today
(pointers to code/docs in the release available by request)

The Cray Chapel Team (Summer 2013)

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:

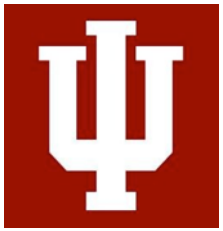
- *[Ten] Myths About Scalable Programming Languages*, Chamberlain. IEEE Technical Committee on Scalable Computing (TCSC) Blog, (https://www.ieeetcsc.org/activities/blog/), April-November 2012.
  - *a series of technical opinion pieces designed to rebut standard arguments against the development of high-level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

Chapel: the next five years

- **Harden prototype to production-grade**
  - add/improve lacking features
  - optimize performance
- **Target more complex/modern compute node types**
  - e.g., Intel MIC, CPU+GPU, AMD APU, …
- **Continue to grow the user and developer communities**
  - including nontraditional circles: desktop parallelism, “big data”
  - transition Chapel from Cray-managed to community-governed

Chapel…

…is a collaborative effort — join us!



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/