



Chapel: A Modern Language for Modern Architectures (and Modern Programmers)

Sung-Eun Choi, Chapel Team
LLNL Co-Design Summer School
August 21, 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.



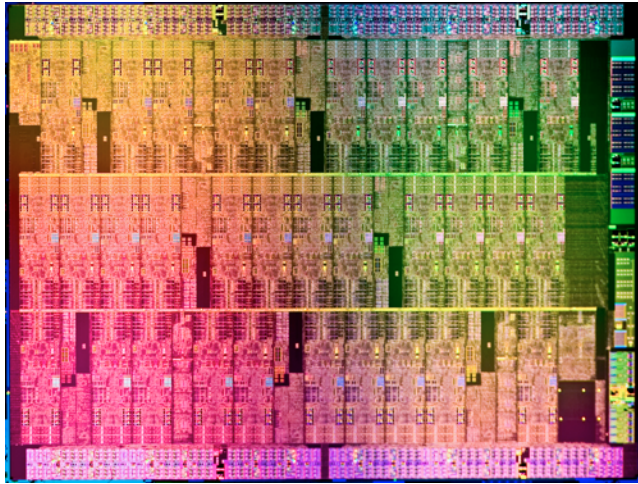


Quick Survey

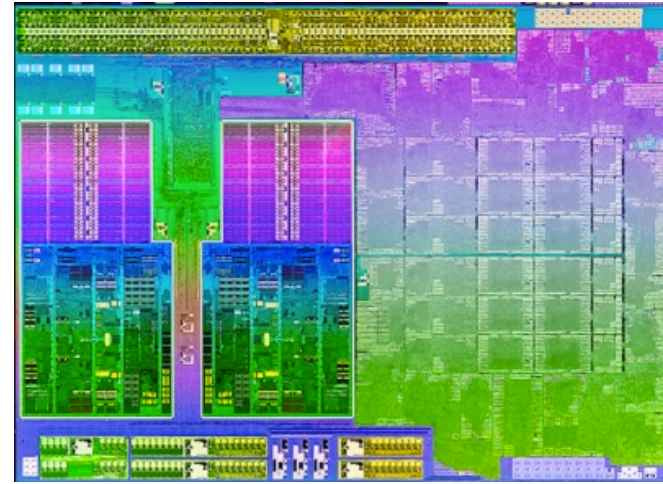
- Major? CS or an “actual” science?
- Have you heard of Chapel?
- Have you used Chapel?
- Day-to-day programming language of choice?



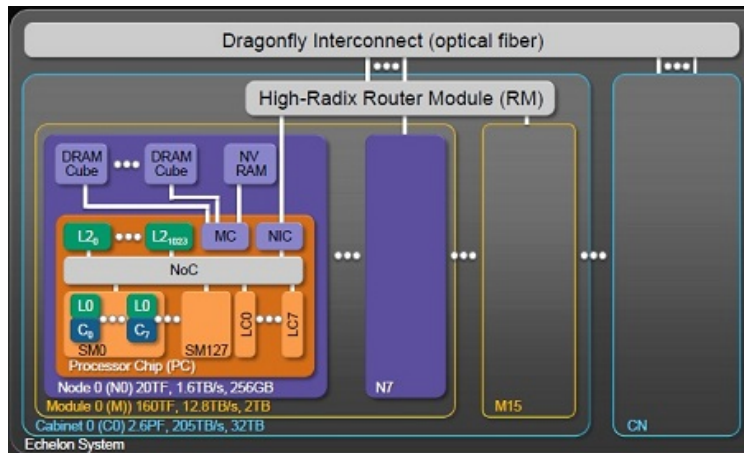
Modern Processor Architectures



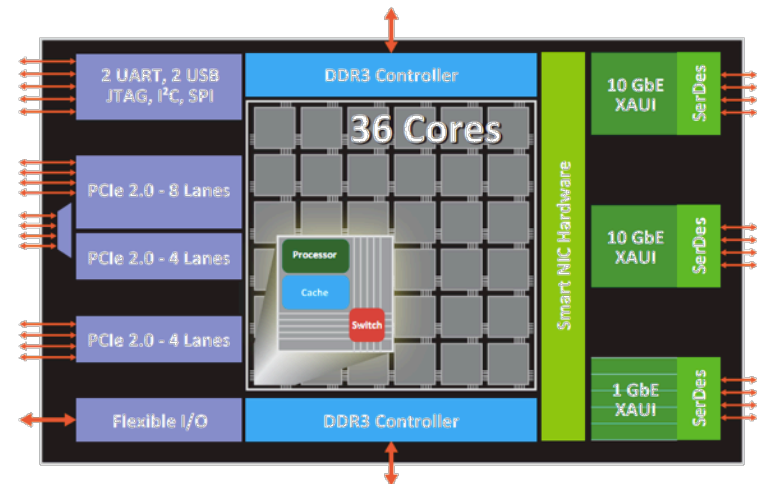
Intel MIC



AMD APU

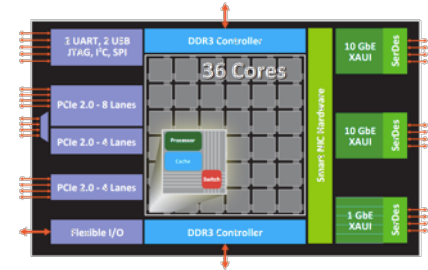
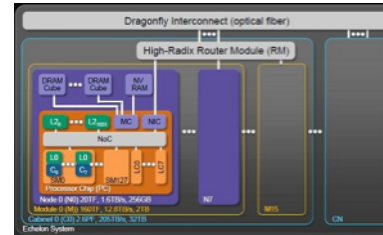
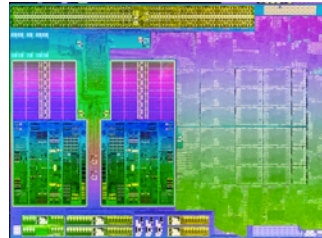
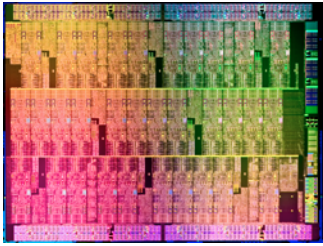


NVIDIA Echelon



Tilera Tile-Gx

Why do we need data locality control?



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?"





Outline

- ✓ Motivation
- Chapel Background and Themes
- Three Features for Tackling Locality
- Summary, Project Status and Next Steps





What is Chapel?

- **An emerging parallel programming language**
 - Design and development led by Cray Inc.
 - in collaboration with academia, labs, industry; domestically & internationally
- **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 want full control to ensure 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 GitHub**
 - Licensed as BSD software (future releases will be Apache)
- **Portable design and implementation, targeting:**
 - multicore desktops and laptops
 - commodity clusters and the cloud
 - HPC systems from Cray and other vendors
 - *in-progress*: manycore processors, CPU+accelerator hybrids, ...



Motivating Themes

- 1) General Parallel Programming
- 2) Global-View Abstractions
- 3) Multiresolution Design
- 4) Control over Locality/Affinity
- 5) Reduce HPC \leftrightarrow Mainstream Language Gap



1) General Parallel Programming

With a unified set of concepts...

...express any parallelism desired in a user's program

- **Styles:** data-parallel, task-parallel, concurrency, nested, ...
- **Levels:** model, function, loop, statement, expression

...target any parallelism available in the hardware

- **Types:** machines, nodes, cores, instructions

Type of HW Parallelism	Programming Model	Unit of Parallelism
Inter-node	Chapel	executable/task
Intra-node/multicore	Chapel	iteration/task
Instruction-level vectors/threads	Chapel	iteration
GPU/accelerator	Chapel	SIMD function/task



2) Global-View Abstractions

In pictures: “Apply a 3-Point Stencil to a vector”

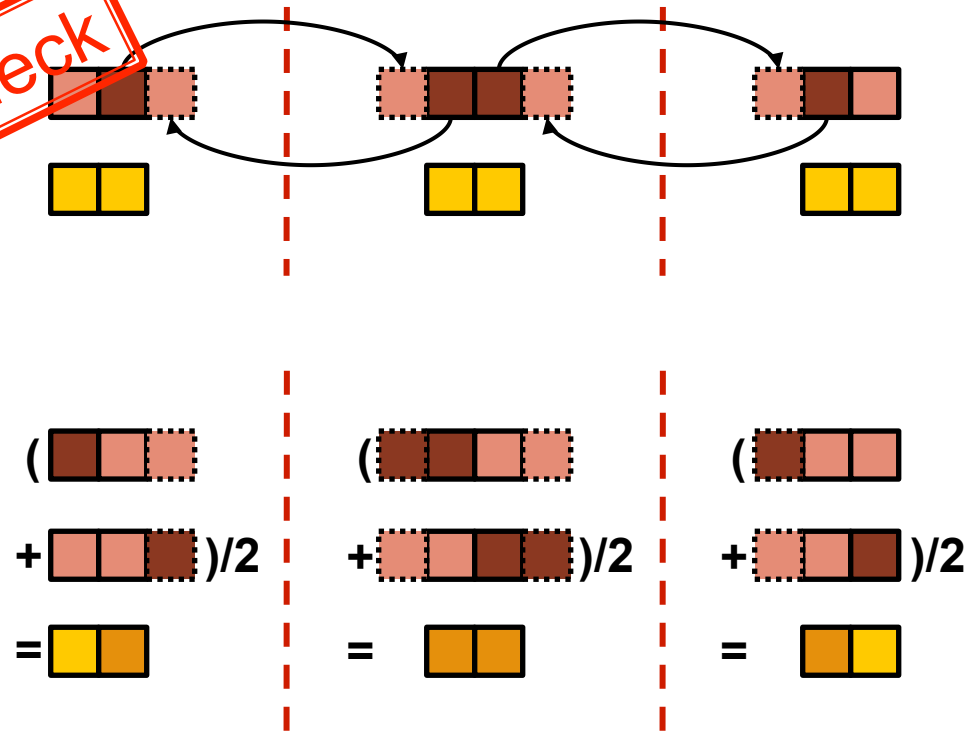
want

Global-View

$$\begin{aligned}
 & (\text{[vector of 6 elements]} \\
 & + \text{[vector of 6 elements]}) / 2 \\
 & = \text{[vector of 6 elements]}
 \end{aligned}$$

block

Local-View

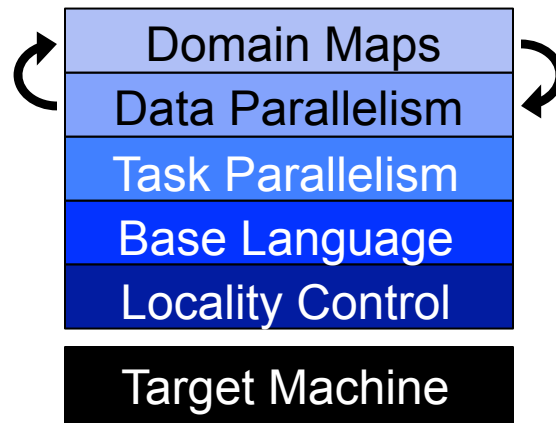


3) Multiresolution Design

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



4) Control over Locality/Affinity

Consider:

- Scalable architectures package memory near processors
- Remote accesses take longer than local accesses

Therefore:

- Placement of data relative to tasks affects scalability
- Give programmers control of data and task placement

Note:

- Parallelism is an orthogonal concept to locality





5) Reduce HPC ↔ Mainstream Language Gap

Consider:

- Students graduate with training in Java, Matlab, Python, etc.
- Yet HPC programming is dominated by Fortran, C/C++, MPI

We'd like to narrow this gulf with Chapel:

- to leverage advances in modern language design
- to better utilize the skills of the entry-level workforce...
- ...while not alienating the traditional HPC programmer
 - e.g., support object-oriented programming, but make it optional





Outline

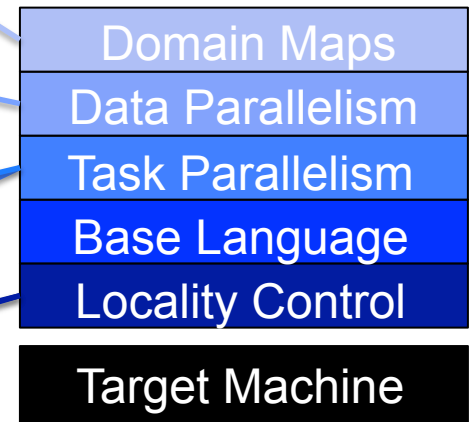
- ✓ Motivation
- ✓ Chapel Background and Themes
- **Three Features for Tackling Locality**
- **Summary, Project Status and Next Steps**



Chapel in 1 Slide!

```
const D = {1..n} dmapped Cyclic(startIdx=1);
var A, B, C: [D] real;
forall (a,b,c) in zip(A,B,C) do
```

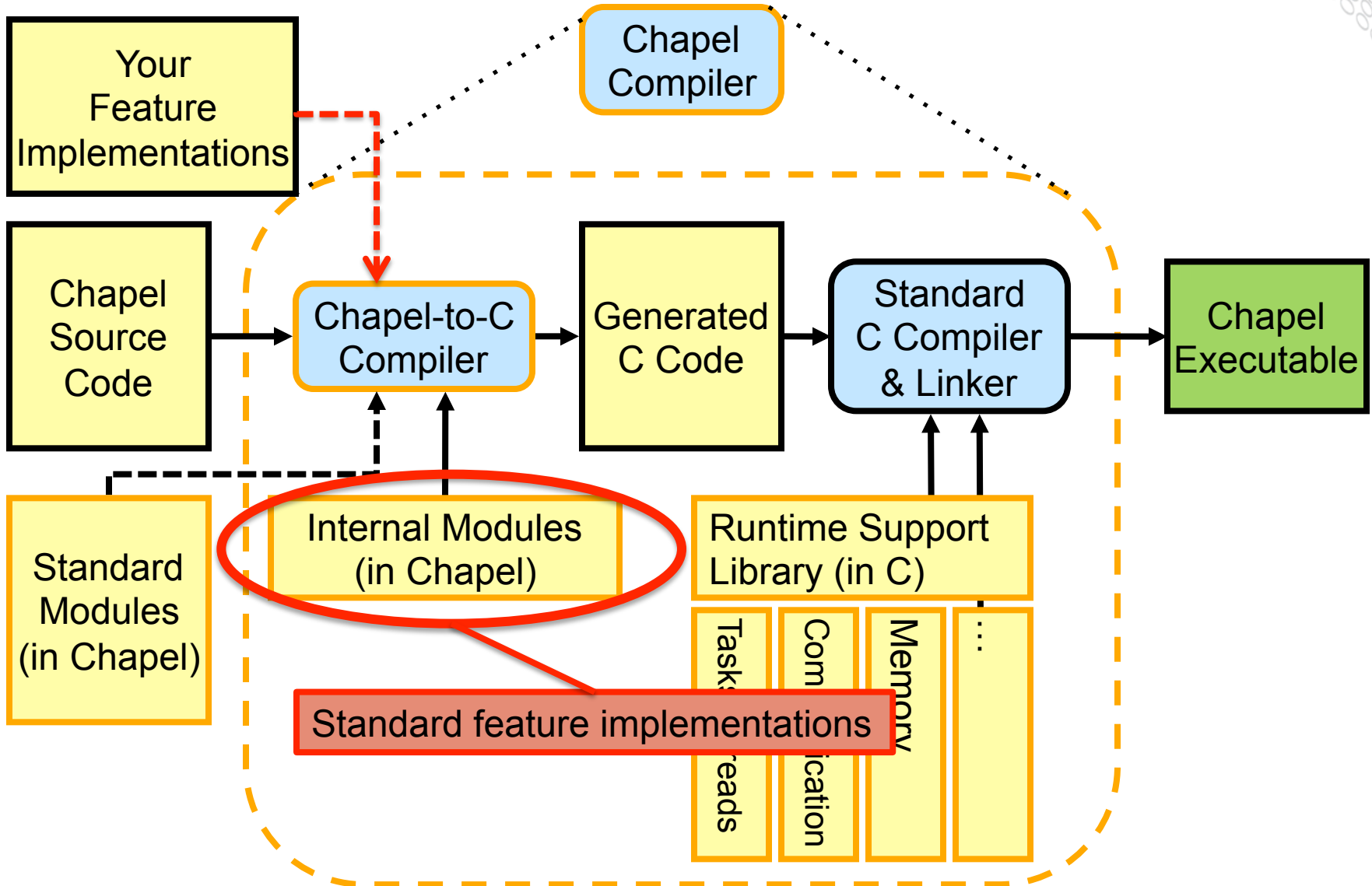
Chapel language concepts



a = b + alpha * c;
 High-level features implemented...
 • in Chapel
 • using lower-level features
 • by end-users

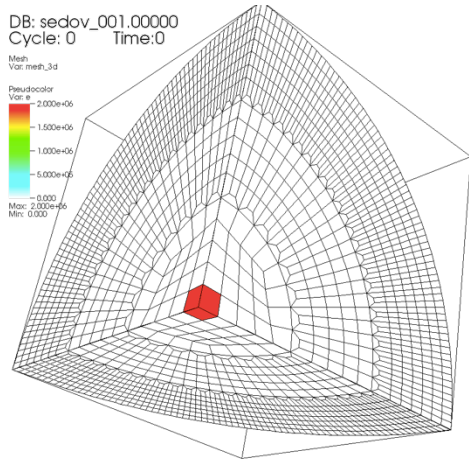
```
var buffer$: [0..numElts] sync real,
cobegin {
  on Locales[1] do producer(buffer$);
  on A[i] do consumer(buffer$);
}
```

Chapel Compiler Architecture

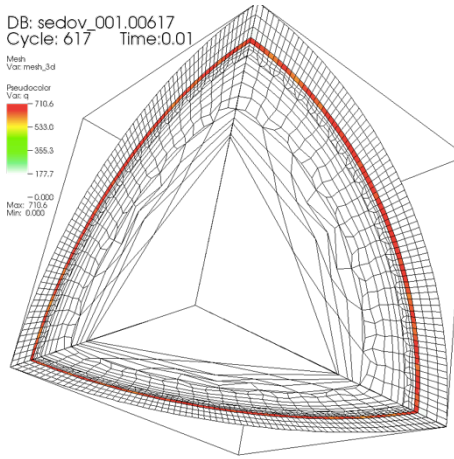


LULESH: a DOE Proxy Application

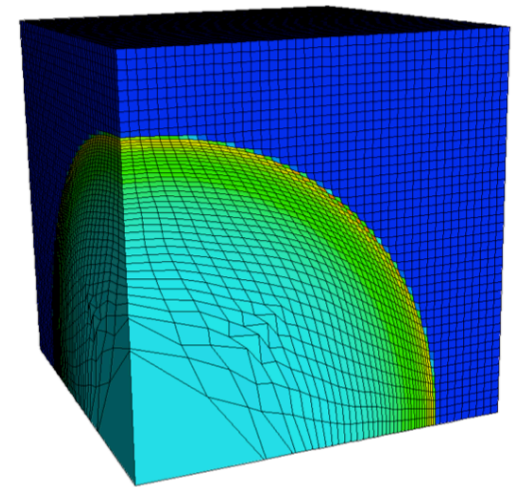
Goal: Solve one octant of the spherical Sedov problem (blast wave) using Lagrangian hydrodynamics for a single material



user: keasler
Thu Apr 12 11:56:04 2012

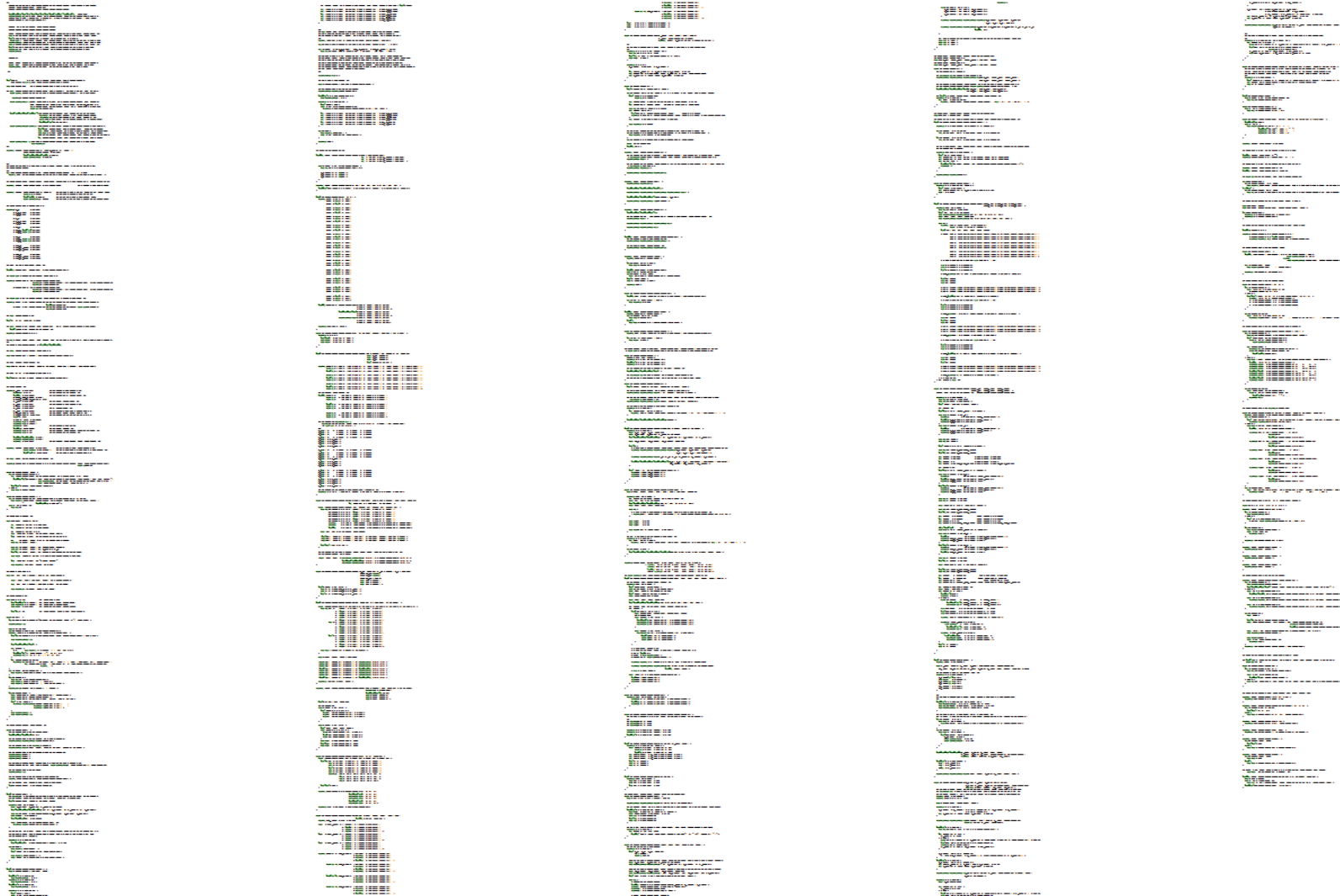
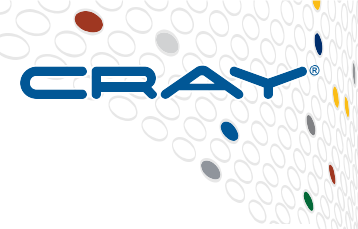


user: keasler
Thu Apr 12 11:57:44 2012



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

LULESH in Chapel



COMPUTE | STORE | ANALYZE

Copyright 2014 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 Chapel v1.9 in `examples/benchmarks/lulesh/*.chpl`

LULESH in Chapel

← This is the only representation-dependent code. It specifies:

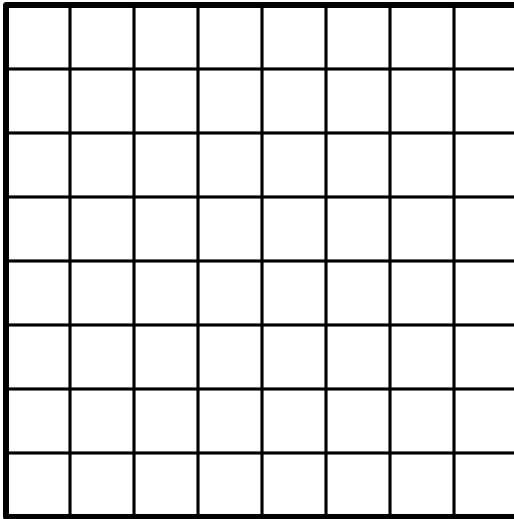
- data structure choices
 - structured vs. unstructured mesh
 - local vs. distributed data
 - sparse vs. dense materials arrays
- a few supporting iterators

Data Parallelism in LULESH (Structured)

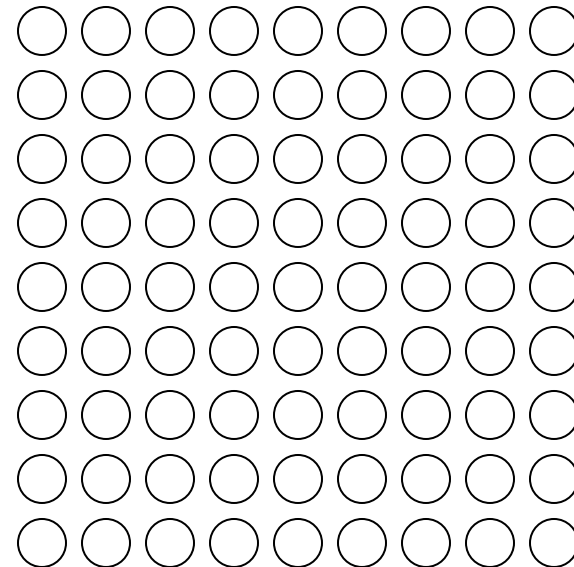
```
const Elms = {0..#elemsPerEdge, 0..#elemsPerEdge},
        Nodes = {0..#nodesPerEdge, 0..#nodesPerEdge};

var determ: [Elms] real;

forall k in Elms { ...determ[k]... }
```



Elms



Nodes

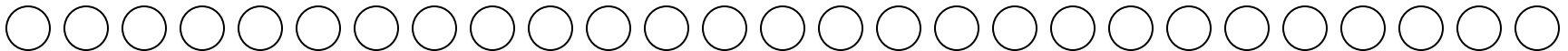


Data Parallelism in LULESH (Unstructured)

```
const Elms = {0..#numElms},  
      Nodes = {0..#numNodes};  
  
var determ: [Elms] real;  
var elemToNode: [Elms] nodesPerElem*index(Nodes);  
  
forall k in Elms { ...determ[k]... }
```



Elms



Nodes





Implementing Domains and Arrays

Q: How are domains and arrays implemented?

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

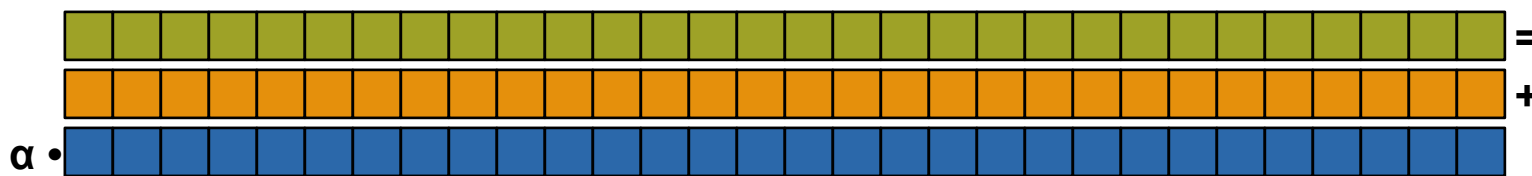
```
const Elems = {0..#numElems},  
        Nodes = {0..#numNodes};  
  
var determ: [Elems] real;
```

A: Via Feature #1 (domain maps)...



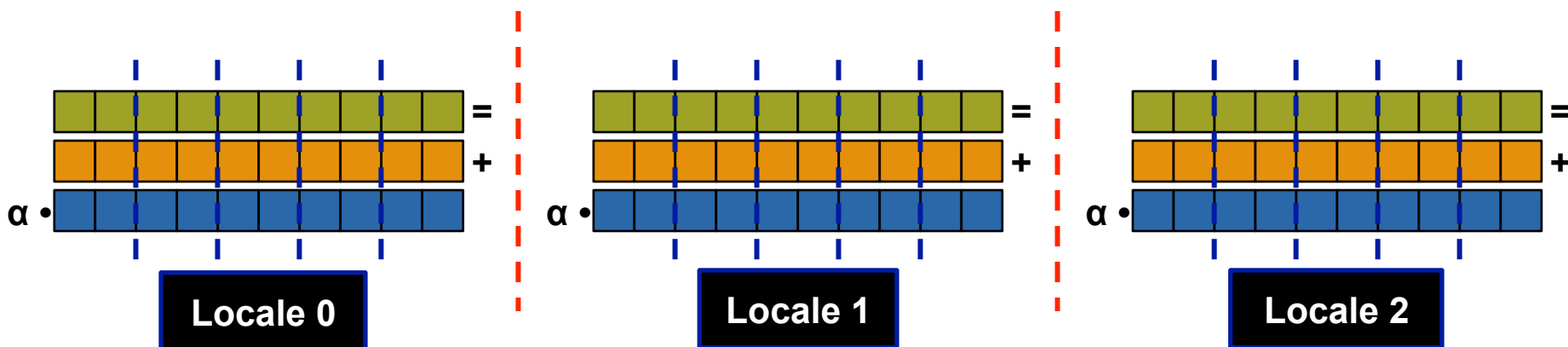
Domain Maps: Concept

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 { ... }
```



Elems

No domain map specified \Rightarrow use default layout

- current locale owns all indices and values
- computation will execute using local processors only



Nodes

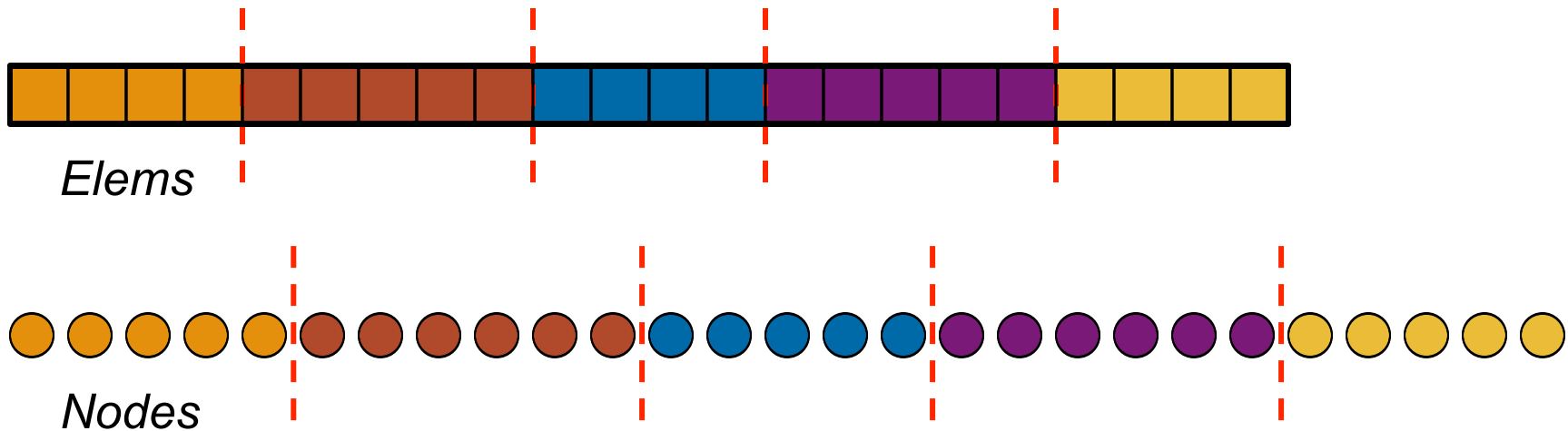
LULESH Data Structures (distributed, block)

```

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 Elms = {0..#numElms} dmapped Cyclic(...),
      Nodes = {0..#numNodes} dmapped Cyclic(...);

var determ: [Elms] real;

forall k in Elms { ... }
```



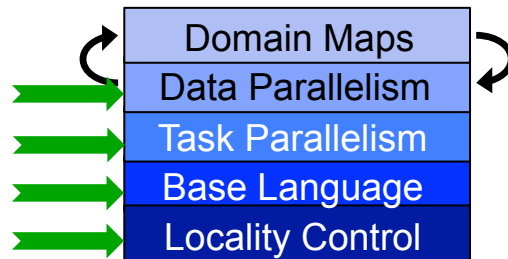
Elms



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 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 Chapel code
 - ...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 Elms { ... }
```

Q2: What about zippered data parallel operations?

(how to reconcile potentially conflicting parallel implementations?)

```
forall (k,d) in zip(Elms, 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*

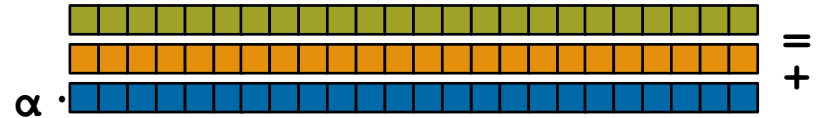
- Domain maps support default leader-follower iterators**
 - specify parallel traversal of a domain's indices/array's elements
 - typically written to leverage affinity



Leader-Follower Iterators: Rewriting

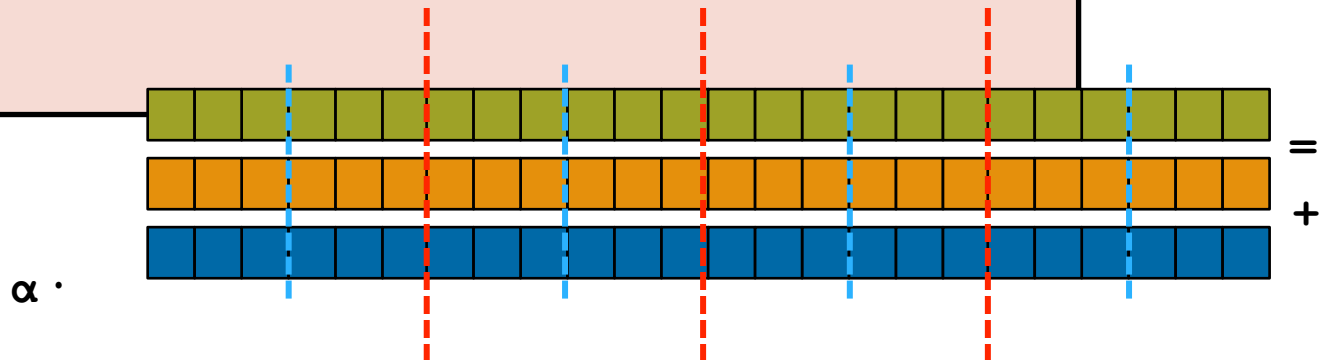
Conceptually, the Chapel compiler translates:

$$A = B + \alpha * C$$



into:

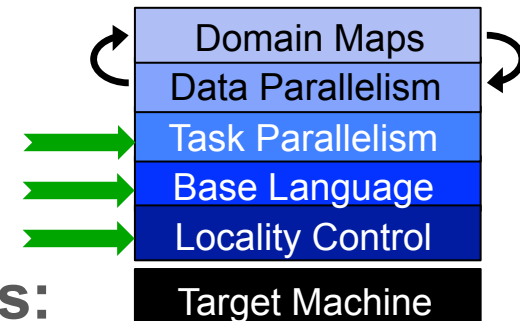
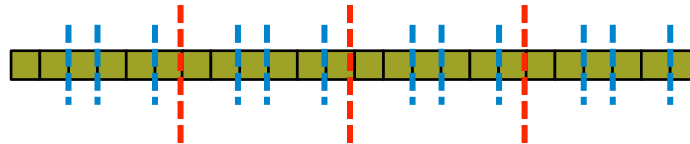
```
coforall loc in targetLocales do on loc {  
  coforall tid in here.numCores {  
    for (a,b,c) in zip(A,B,C) {  
      a = b + alpha * c;  
    }  
  }  
}
```



Writing Leaders and Followers

Leader iterators are defined using task/locality features:

```
iter BlockArr.lead() {
    coforall loc in targetLocales 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);
}
```



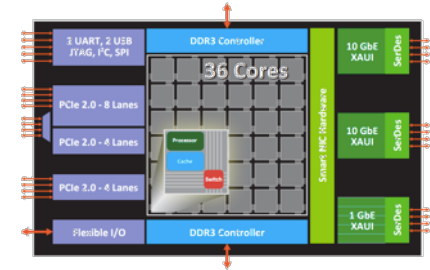
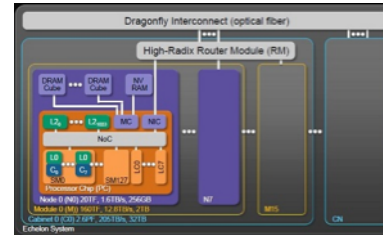
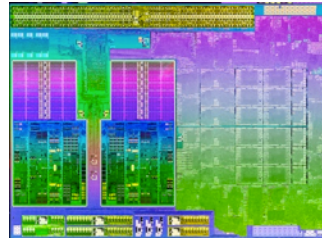
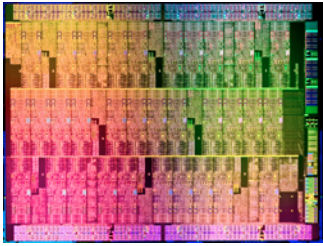

Leader-Follower Summary

- **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 Chapel code
 - ...expose them to the end-user through data parallel abstractions

```
forall k in Elms { ... }  
x += xd * dt;
```



Q: What about those modern architectures?



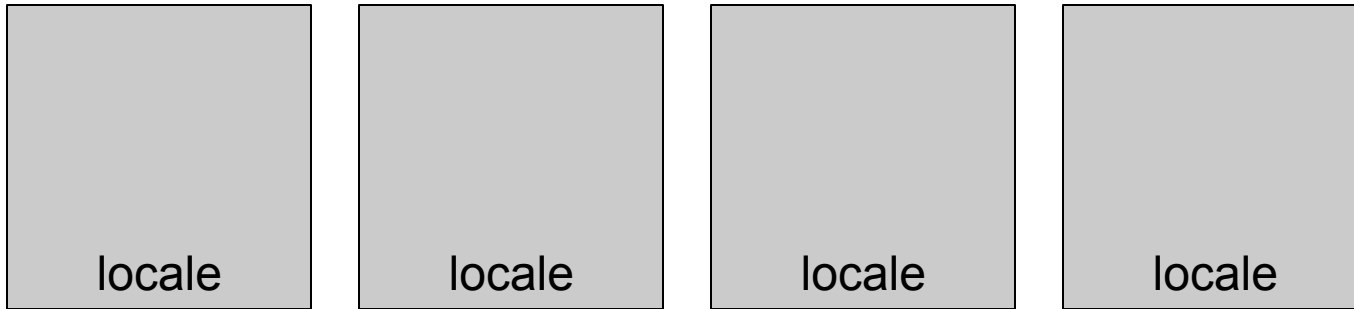
A: Feature #3 (hierarchical locales)

- extends multiresolution philosophy to architectural modeling

Traditional Locales

Concept:

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

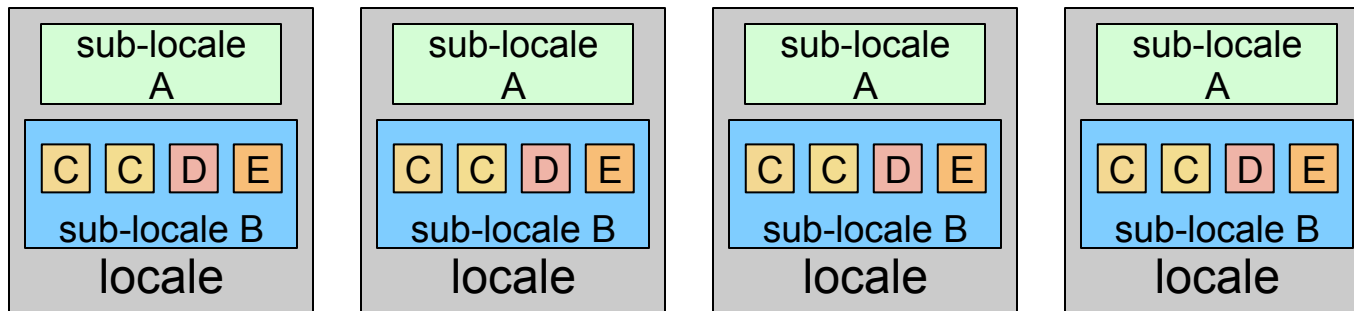


- Supports inter-node locality well, but not intra-node

Hierarchical Locales

Concept:

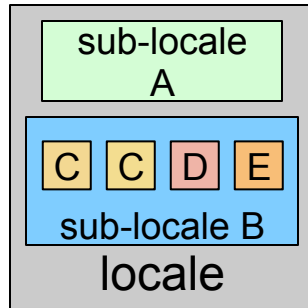
- Support locales within locales to describe architectural sub-structures within a node (e.g., memories, processors)



- 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 A Locale Model

1) Define the processor's abstract block structure



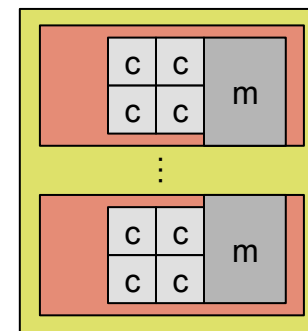
2) Define how to run a task on any sublocale

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

NUMA node Example

```
class locale: AbstractLocaleModel {
  const numNumaDomains = chpl_task_getNumSublocales();
  const sublocales: [0..#numNumaDomains] numaDomain = ...;
  ...memory interface...
  ...tasking interface...
}
```

```
class numaDomain: AbstractLocaleModel {
  const numCores = ...;
  ...memory interface...
  ...tasking interface...
}
```



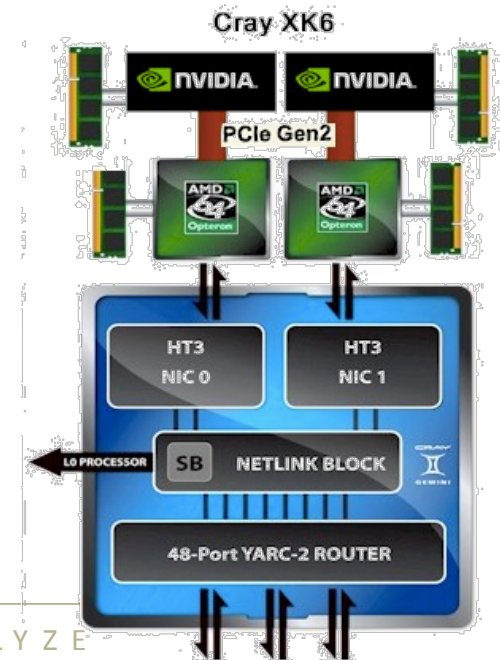
Complete, but not optimized

Hybrid Processor Example

```
class locale: AbstractLocaleModel {
    const numNumaDomains = ..., numGPUs = ...;
    const cpus: [0..#numCPUs] numaDomain = ...;
    const gpus: [0..#numGPUs] gpuLoc = ...;
    ...memory interface...
    ...tasking interface...
}
```

```
class gpuLoc: AbstractLocaleModel {
    ...sublocales for different
        memory types, thread blocks...?
    ...memory, tasking interfaces...
}
```

In progress





Hierarchical Locale Summary

- **Data locality requires flexibility w.r.t. modern 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 architectural models to Chapel code
 - ...are exposed to the end-user via Chapel's traditional locality features

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





Outline

- ✓ Motivation
- ✓ Chapel Background and Themes
- ✓ Three Features for Tackling Locality
- **Summary, Project Status and Next Steps**





Back to the title

Modern Language

- Chapel provides high-level abstractions and modern features that enable productive programming

Modern Architectures

- Chapel's Locale Models framework enables users to directly target modern architectures in Chapel code (without changing the code for their algorithms)

Modern Programmers

- (We think) people think Chapel is cool





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 programmer's hand...

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



Implementation Status -- Version 1.9.0 (Apr 2014)

Overall Status:

- **User-facing Features:** generally in good shape
 - some require additional attention (e.g., strings, OOP)
- **Multiresolution Features:** in use today
 - their interfaces are likely to continue evolving over time
- **Error Messages:** not always as helpful as one would like
 - correct code works well, incorrect code can be puzzling
- **Performance:** hit-or-miss depending on the idioms used
 - Chapel designed to ultimately support competitive performance
 - effort to-date has focused primarily on correctness

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



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!



Lawrence Berkeley
National Laboratory



COMPUTE | STORE | ANALYZE

Copyright 2014 Cray Inc.



For More Information: Online Resources

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

- overview, papers, presentations, language spec, ...

Chapel GitHub page: <https://github.com/chapel-lang>

- source code repository

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

- download 1.9.0 release, join community mailing lists

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





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.

Copyright 2014 Cray Inc.





CRAY
THE SUPERCOMPUTER COMPANY

<http://chapel.cray.com>

chapel_info@cray.com