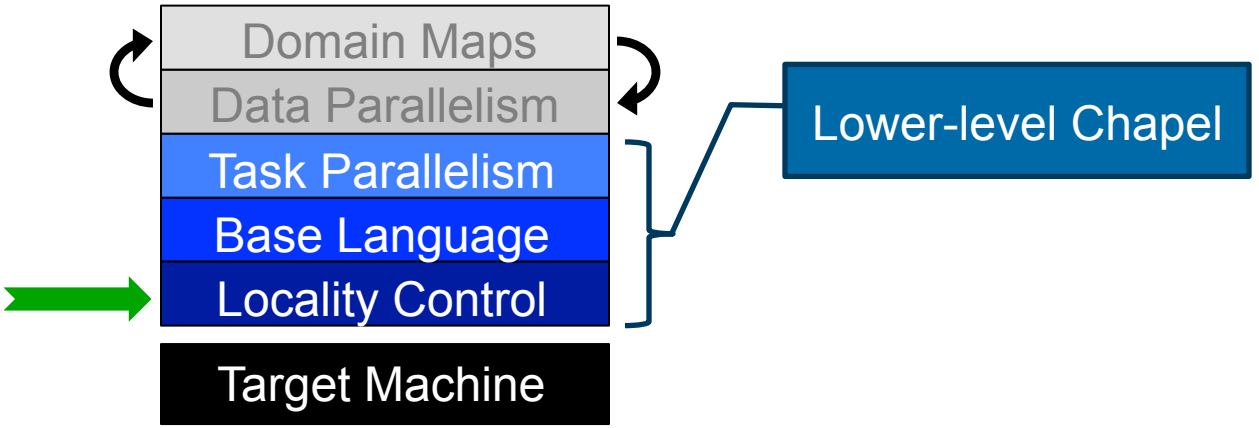
Locality By Example

SC14
New Orleans, LA | hpc matters.

Safe Harbor Statement

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

Locality Features



"Hello World" in Chapel: a Multi-Locale Version

- **Multi-locale Hello World**

```
coforall loc in Locales do
  on loc do
    writeln("Hello, world! ",
            "from node ", loc.id, " of ", numLocales);
```

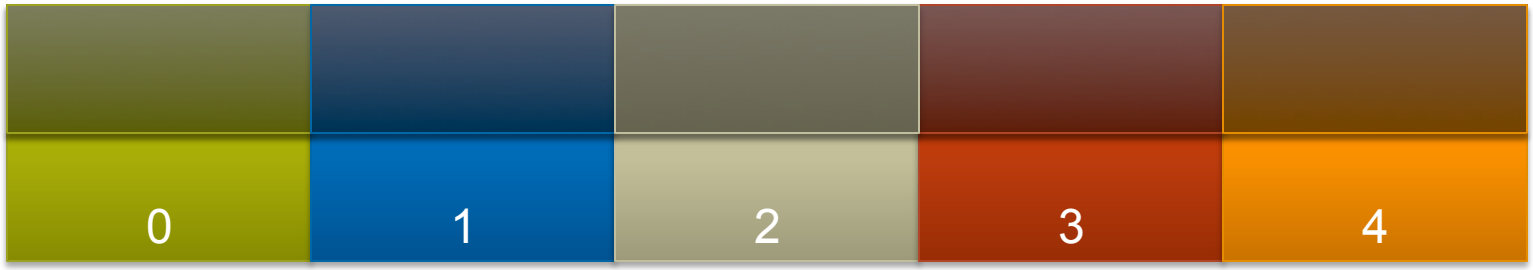
```
Hello, world! from node 3 of 6
Hello, world! from node 0 of 6
Hello, world! from node 5 of 6
Hello, world! from node 2 of 6
Hello, world! from node 1 of 6
Hello, world! from node 4 of 6
```

The Locale Type

Definition:

- Abstract unit of target architecture
- Supports reasoning about locality
  - defines “here vs. there” / “local vs. remote”
- Capable of running tasks and storing variables
  - i.e., has processors and memory

Typically: A compute node (multicore processor or SMP)



Locales (think: “compute nodes”)

Getting started with locales

- Specify # of locales when running Chapel programs

```
% a.out --numLocales=8
```

```
% a.out -nl 8
```

- Chapel provides built-in locale variables

```
config const numLocales: int = …;
const Locales: [0..#numLocales] locale = …;
```

- User's `main()` begins executing on locale #0

Locale Operations: System Queries

- Locale methods support queries about the target system:

```
proc locale.physicalMemory(…) { … }
proc locale.numCores { … }
proc locale.maxTaskPar { … }
proc locale.id { … }
proc locale.name { … }
```

```
const nodeMemory     = here.physicalMemory();
const systemMemory   = + reduce Locales.physicalMemory();
const maxNodeMemory = max reduce Locales.physicalMemory();
const minNodeMemory = min reduce Locales.physicalMemory();

if (minNodeMemory != maxNodeMemory) then
  writeln("My Locales have different amounts of memory");
```

Parallelism and Locality: Orthogonal in Chapel

- This is a **parallel**, but local program:

```
begin writeln("Hello world!");
writeln("Goodbye!");
```

- This is a **distributed**, but serial program:

```
writeln("Hello from locale 0!");
on Locales[1] do writeln("Hello from locale 1!");
writeln("Goodbye from locale 0!");
```

- This is a **distributed**, **parallel** program:

```
begin on Locales[1] do writeln("Hello from locale 0!");
on Locales[2] do begin writeln("Hello from locale 1!");
writeln("Goodbye from locale 0!");
```

Data-Driven On-Clauses

- **In practice, on-clauses typically refer to variables rather than specific locales:**

```
begin on A[i,j] do
        bigComputation(A);

begin on node.left do
        search(node.left);
```

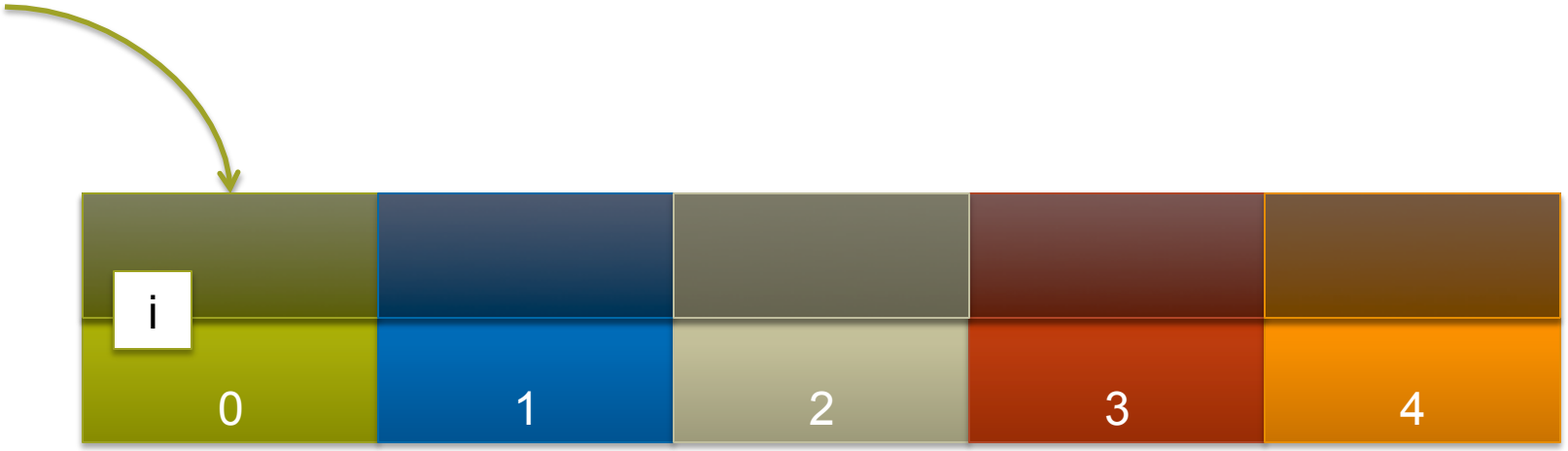
Q: How does data get mapped to locales to begin with?

A (high-level): distributions (see “data parallelism” section)

A (low-level): on-clauses and lexical scoping…

Chapel: Scoping and Locality

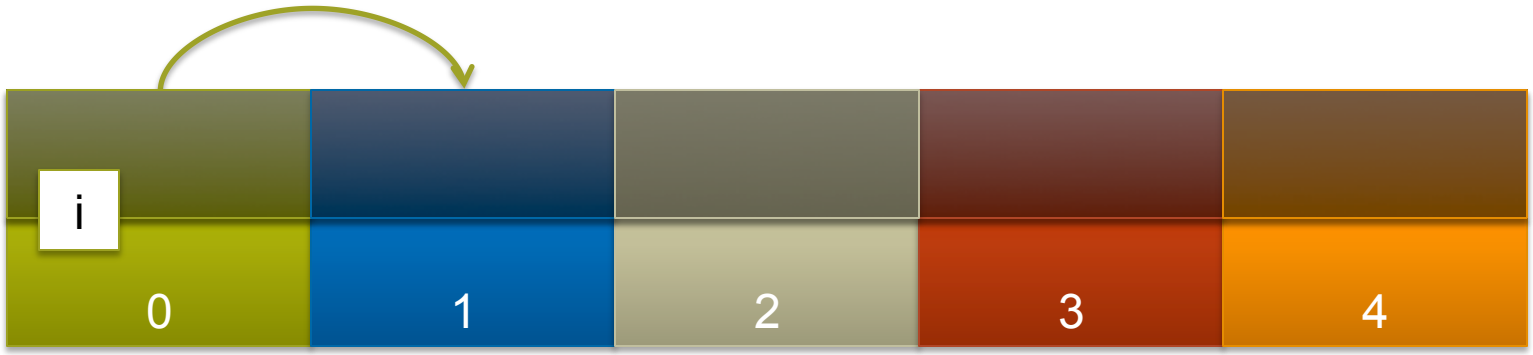
```
var i: int;
```



Locales (think: "compute nodes")

Chapel: Scoping and Locality

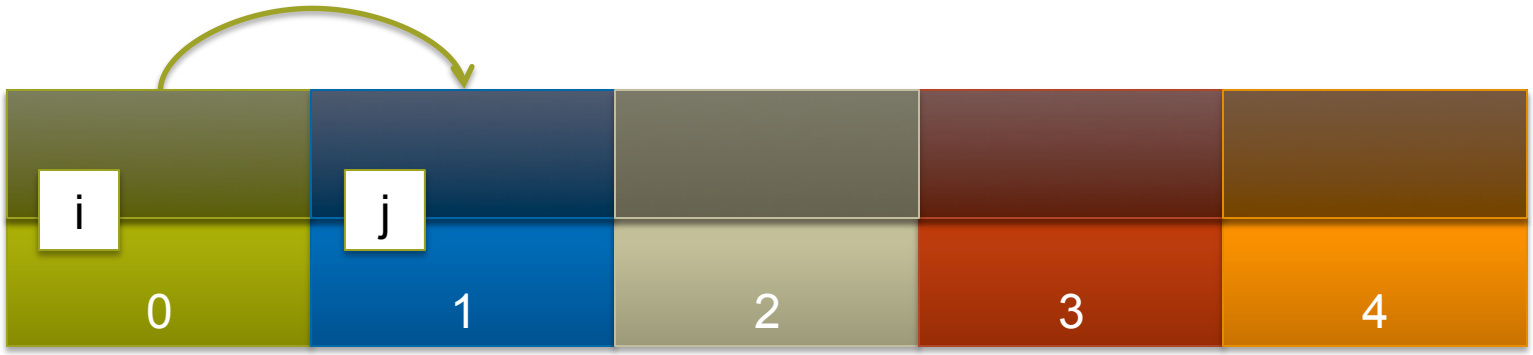
```
var i: int;
on Locales[1] {
```



Locales (think: “compute nodes”)

Chapel: Scoping and Locality

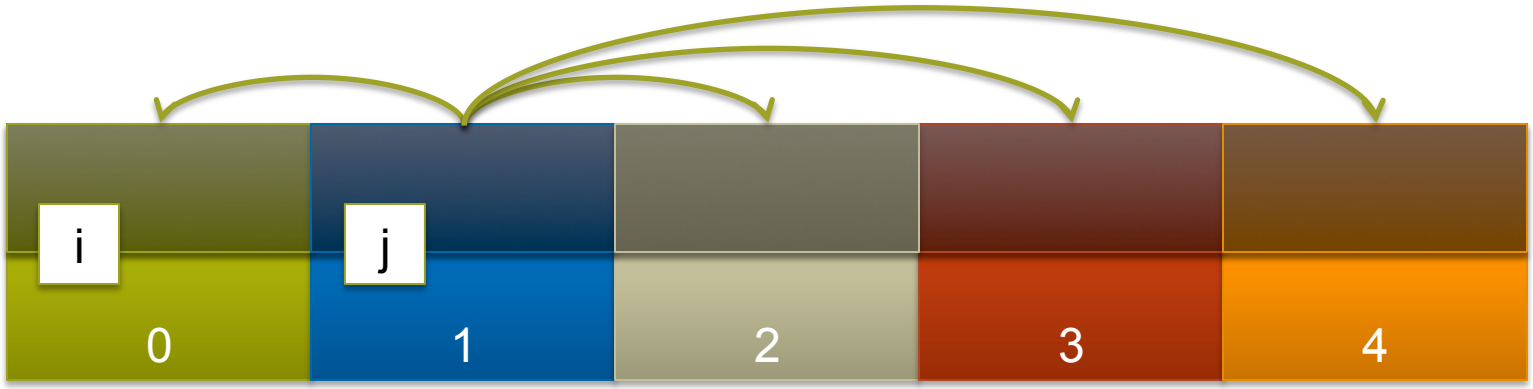
```
var i: int;
on Locales[1] {
  var j: int;
```



Locales (think: “compute nodes”)

Chapel: Scoping and Locality

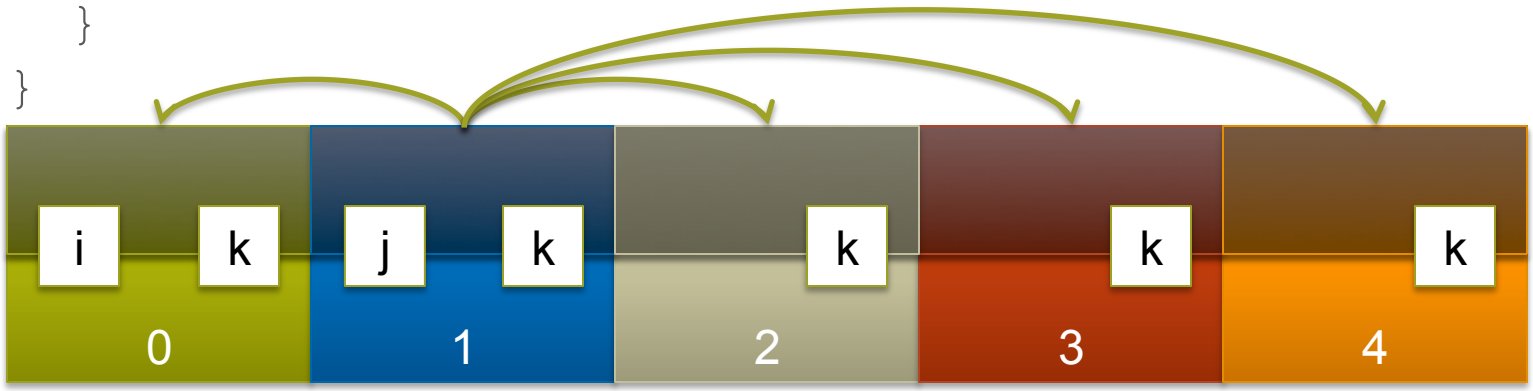
```
var i: int;
on Locales[1] {
  var j: int;
  coforall loc in Locales {
    on loc {
```



Locales (think: “compute nodes”)

Chapel: Scoping and Locality

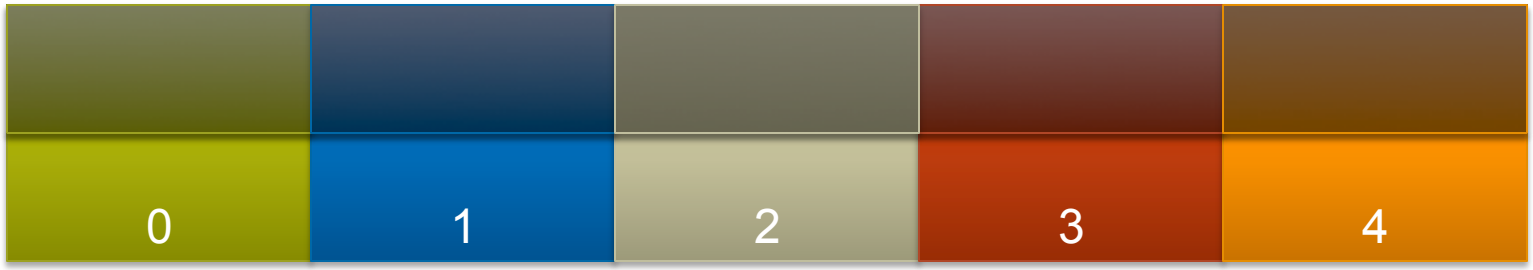
```
var i: int;
on Locales[1] {
  var j: int;
  coforall loc in Locales {
    on loc {
      var k: int;
      // within this scope, i, j, and k can be referenced;
      // the implementation manages the communication for i and j
      // (Chapel is a PGAS language)
    }
  }
}
```



Locales (think: "compute nodes")

Chapel and PGAS

- **Chapel is PGAS, but unlike most, it's not limited to SPMD**
  - ⇒ never think about "the other copies of the program"
  - ⇒ "global name/address space" comes from lexical scoping
    - as in traditional languages, each declaration creates a variable
    - variables are stored on the locale where the task declaring it is executing



Locales (think: "compute nodes")

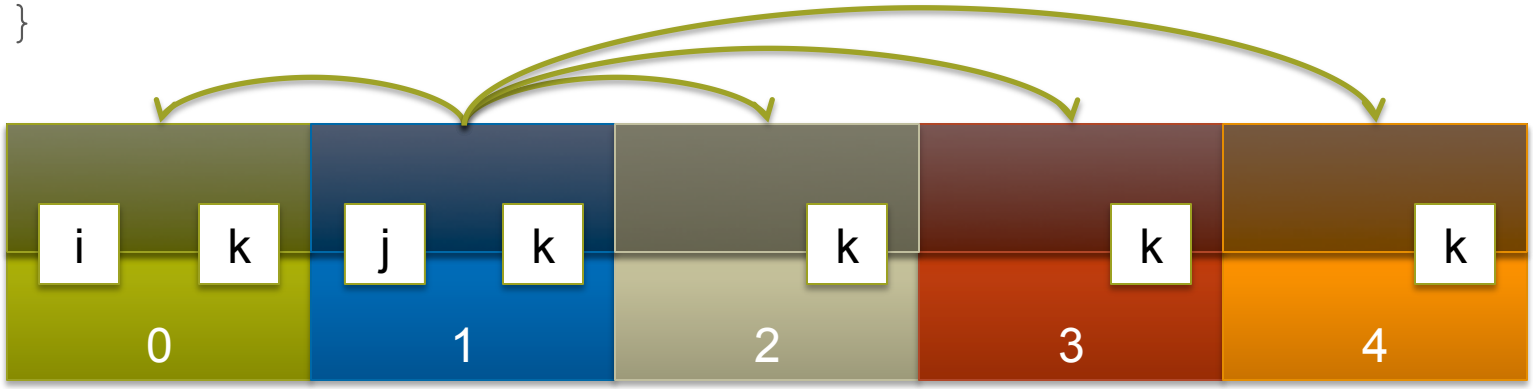
Chapel: Locality Queries

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

      if (j.locale == here) then …
    }
  }
}
```

Apply '.locale' to any expression to query the locale where it lives

'here' represents the locale where the current task is running



Locales (think: "compute nodes")

Rearranging Locales

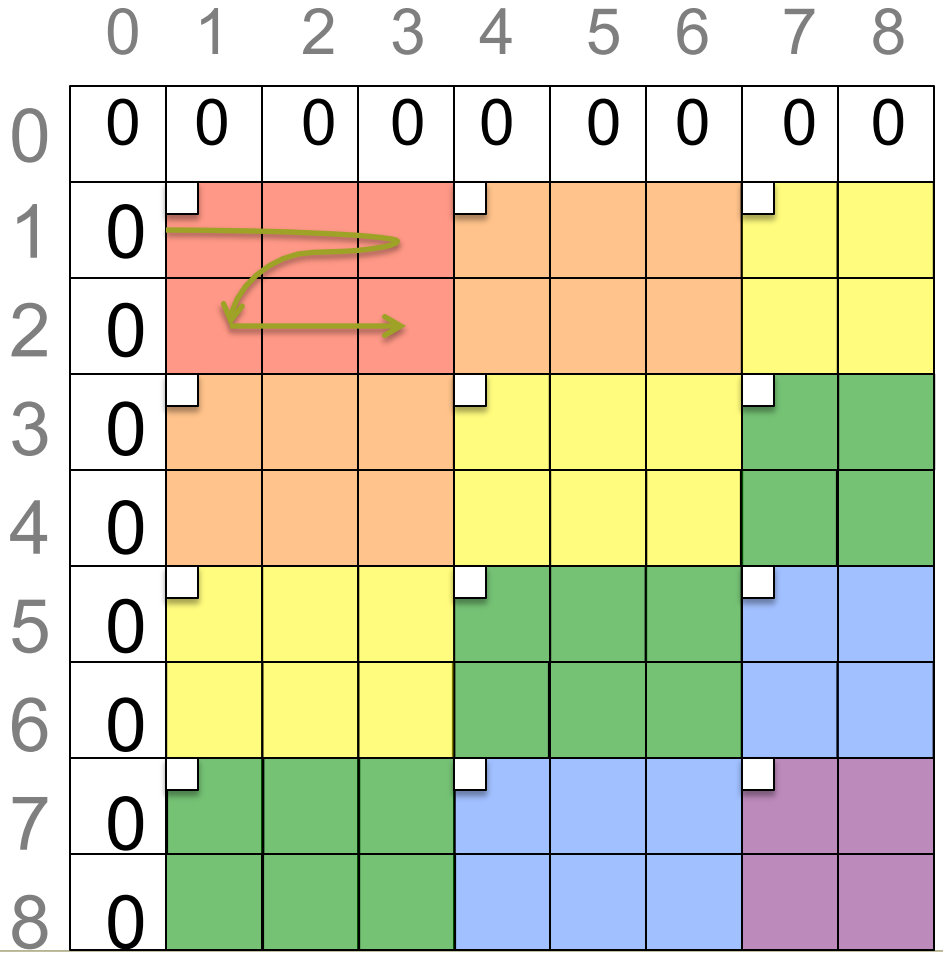
Create locale views with standard array operations:

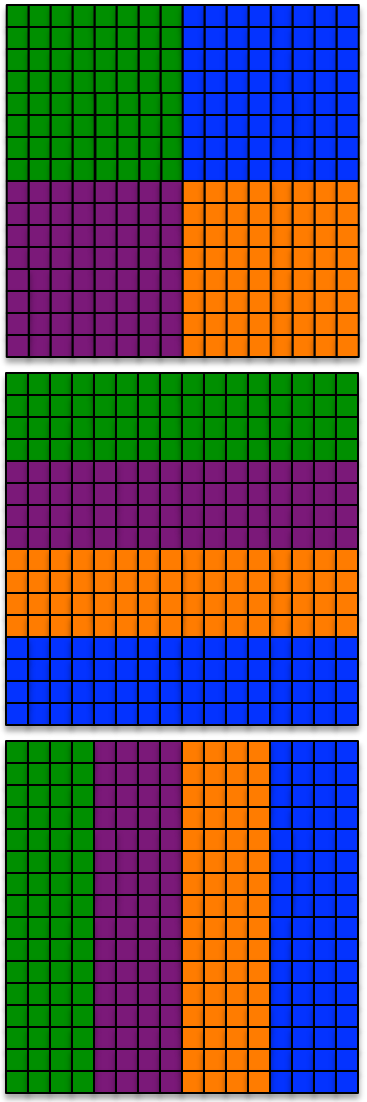
```
var TaskALocs = Locales[0..1];
var TaskBLocs = Locales[2..];

var Grid2D = reshape(Locales, {1..2, 1..4});
```

Distributed Smith-Waterman

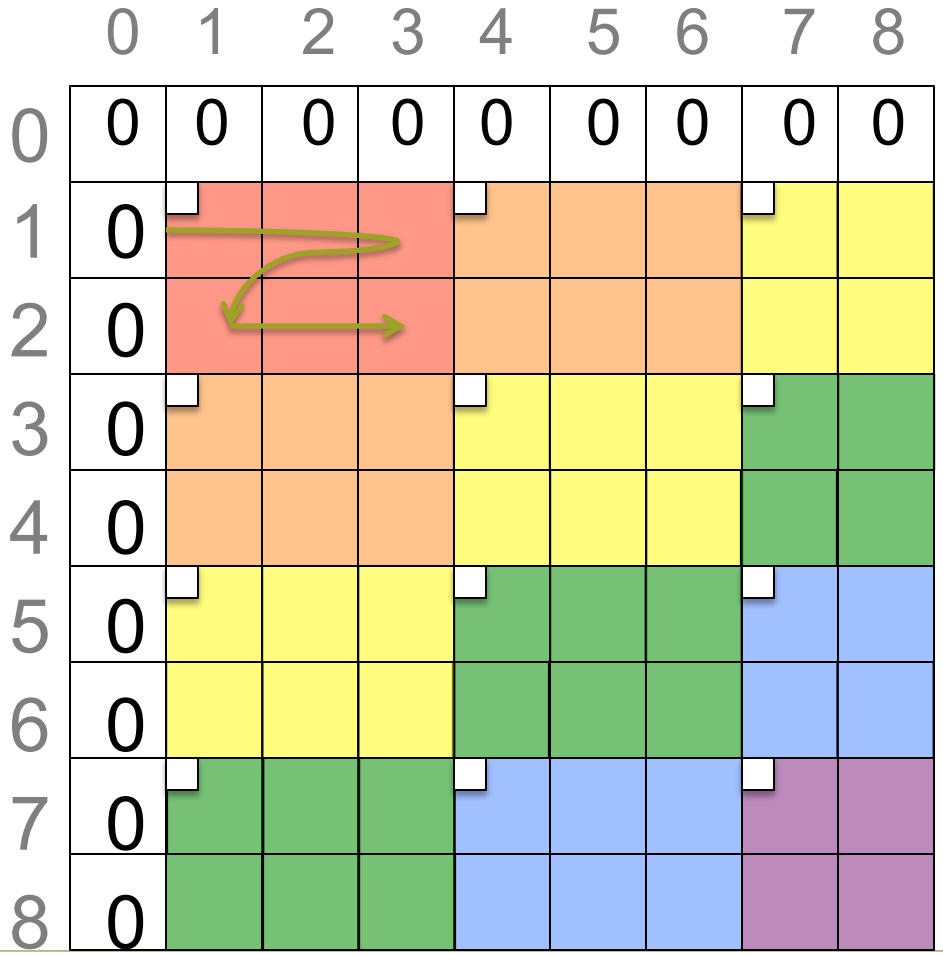
Now, what about distributed memory?

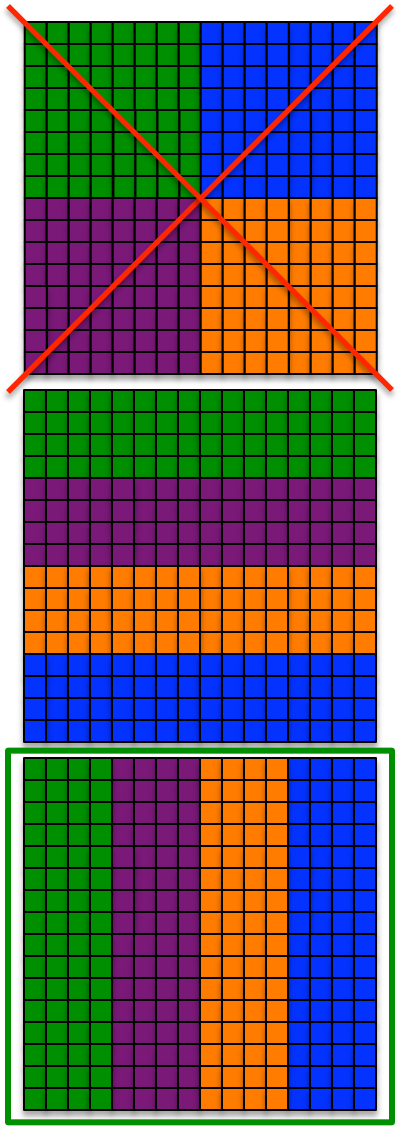




Distributed Smith-Waterman

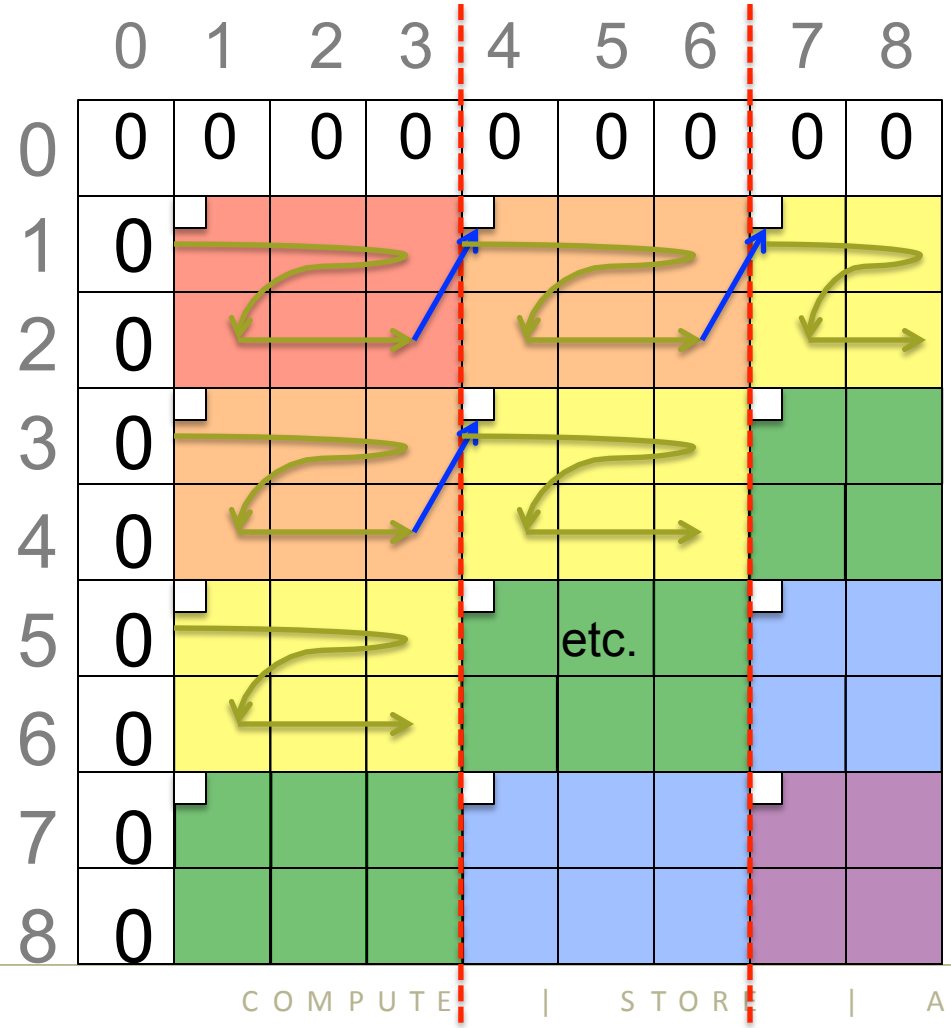
Now, what about distributed memory?





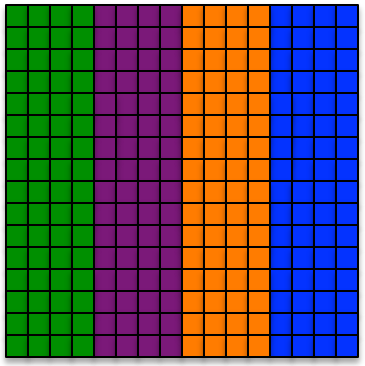
Distributed Smith-Waterman

Now, what about distributed memory?



Advantages:

- Good cache behavior: Nice fat blocks of data touchable in memory order
- Pipeline parallelism: Good utilization once pipeline is filled



Distributed Smith-Waterman

Distributed Chunked Data-Driven Task-Parallel Approach:

```
const Hspace = {0..n, 0..n};
const LocaleGrid = Locales.reshape({0..#numLocales, 0..0});
const DistHSpace = Hspace dmapped Block(Hspace, LocaleGrid);
var H: [DistHSpace] int;

proc computeH(H: [] int) {
  const ProbSpace = H.domain.translate(1,1);
  const StrProbSpace = ProbSpace by (rowsPerChunk, colsPerChunk);
  var NeighborsDone: [StrProbSpace] atomic int;
…
  proc computeHHelp(x,y) {
    on H[x,y] {
      for (i,j) in ProbSpace[x..#rowsPerChunk, y..#colsPerChunk] do
        H[i,j] = f(H[i-1,j-1], H[i-1,j], H[i,j-1]);
    const eastReady =  NeighborsDone[x,                y+colsPerChunk].fetchAdd(1);
    …etc…
    if (eastReady == 2)  then begin computeHHelp(x,                y+colsPerChunk);
    …etc…
} } }
```

Reshape the 1D Locales array into a 2D column

Block-distribute the data space across the column of locales

Compute each chunk on the locale that owns its initial element

Summary: Primary Locality Concepts

Locales: our abstraction for distributed system resources

On-clauses: a fine-grained way to move data/tasks to locales

Distributions/Domain Maps: a global-view way to distribute computation and data to locales

More Multiresolution Concepts

Chapel's Three Core Multiresolution Concepts

- ☑ **domain maps:** Permit users to specify how domains and arrays are mapped to machine resources
- ☐ **leader-follower iterators:** Permit users to specify the parallelism and work decomposition used by forall loops
  - including zippered forall loops
- ☐ **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, …

Motivation for Leader-Follower Iterators

Q: How are parallel loops implemented?

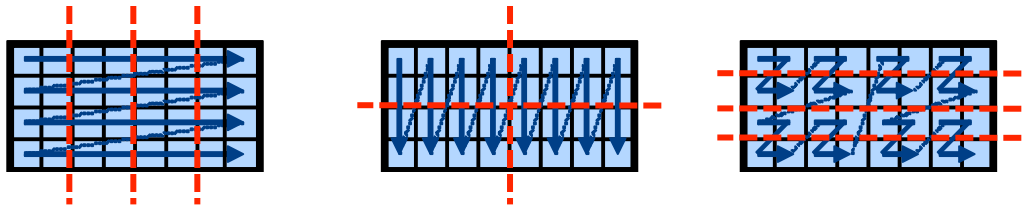
(how many tasks? executing where? how are iterations divided up?)

```
forall k in Elems { … }
```

Q2: What about zippered data parallel operations?

(how to reconcile potentially conflicting parallel implementations?)

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



A *B* *C*

A: Via Leader-Follower Iterators…

Leader-Follower Iterators: Definition

- **Chapel defines forall loops using *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:**

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

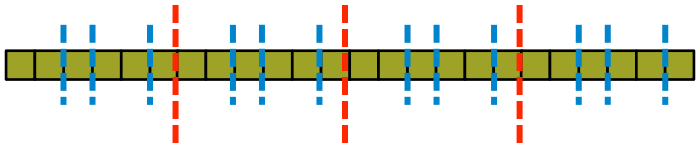
into:

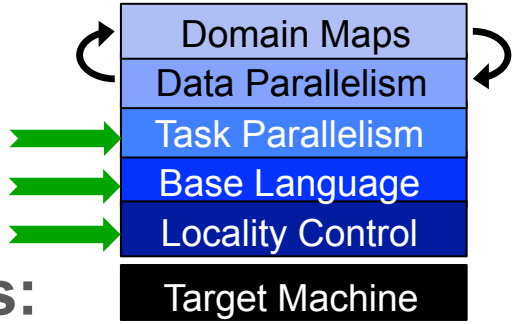
```
inlined A.lead() iterator, which creates tasks that yield work {
  for (a,b,c) in zip(A.follow(work),
                     B.follow(work)
                     C.follow(work)) do
    a = b + alpha * c;
}
```

Writing Leaders and Followers (notionally)

Leader iterators are defined using task/locality features:

```
iter BlockArr.lead() {
  coforall loc in Locales do
    on loc do
      coforall tid in 0..#here.maxTaskPar do
        yield computeMyChunk(loc.id, tid);
}
```





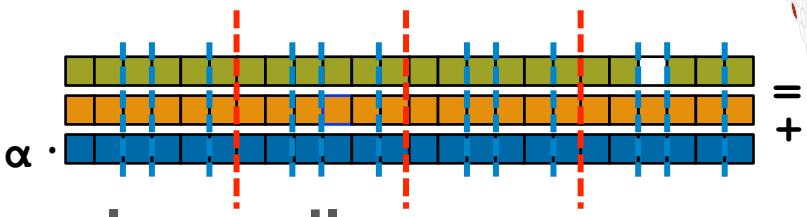
Follower iterators simply use serial features:

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

Leader-Follower Iterators: Rewriting (notionally)

- Putting it all together, the following loop…

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



…would get rewritten by the Chapel compiler as:

```
coforall loc in Locales do
  on loc do
    coforall tid in 0..#here.maxTaskPar {
      const work = computeMyChunk(loc.id, tid);
      for (a,b,c) in zip(A.follow(work),
                         B.follow(work)
                         C.follow(work)) do
        a = b + alpha * c;
    }
```

Leader-Follower Summary

- **Most languages define a fixed menu of parallel loop styles**
  - where “no parallel loops” is a common choice
- **Leader-follower iterators…**

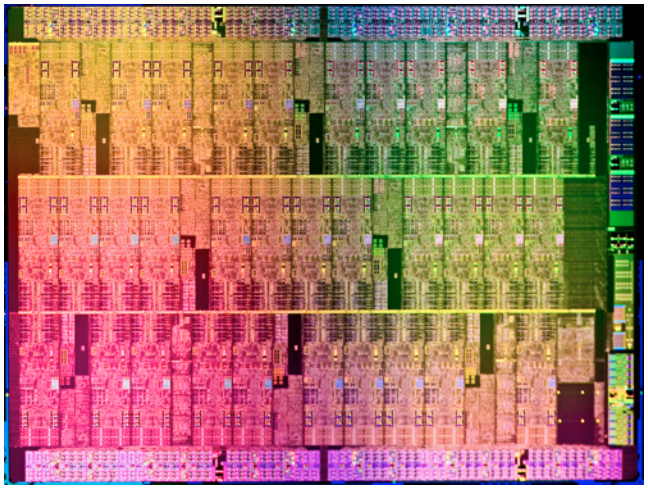
  …move such policies into user-space

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

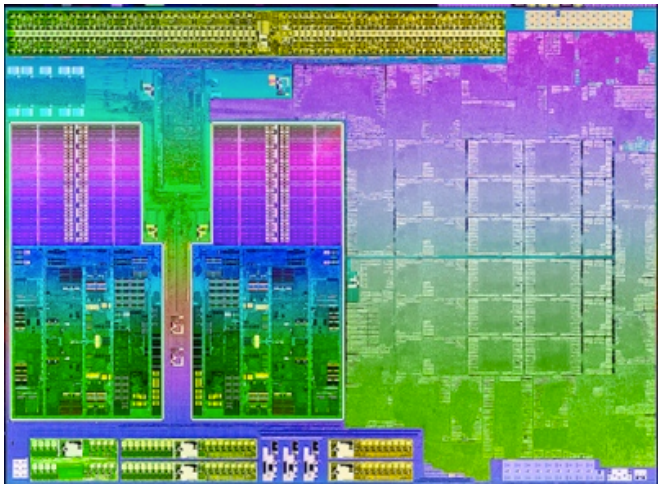
Chapel's Three Core Multiresolution Concepts

- ☑ **domain maps:** Permit users to specify how domains and arrays are mapped to machine resources
- ☑ **leader-follower iterators:** Permit users to specify the parallelism and work decomposition used by forall loops
  - including zippered forall loops
- ☐ **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, …

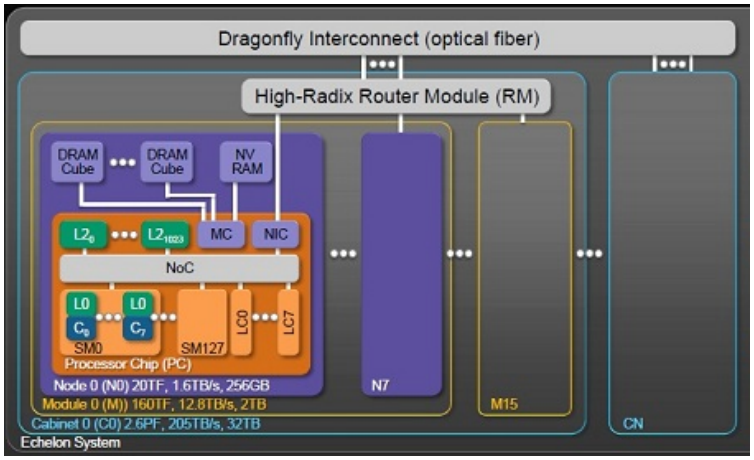
Prototypical Next-Gen Processor Technologies



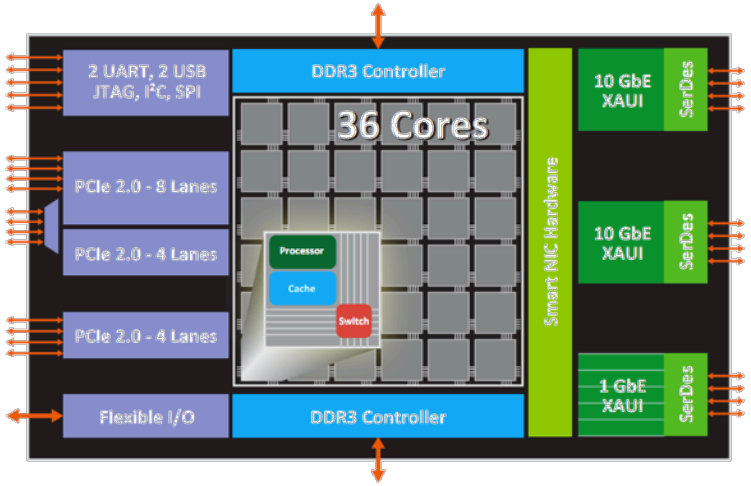
Intel MIC



AMD APU



Nvidia Echelon

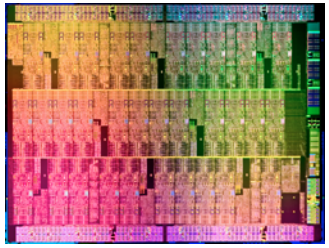


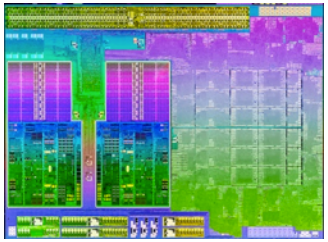
Tilera Tile-Gx

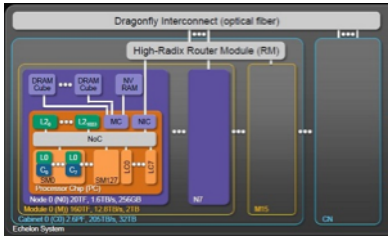
http://download.intel.com/pressroom/images/Aubrey_Isle_die.jpg http://www.zdnet.com/amds-trinity-processors-take-on-intels-ivy-bridge-3040155225/
http://insidehpc.com/2010/11/26/nvidia-reveals-details-of-echelon-gpu-designs-for-exascale/ http://tilera.com/sites/default/files/productbriefs/Tile-Gx%203036%20SB012-01.pdf

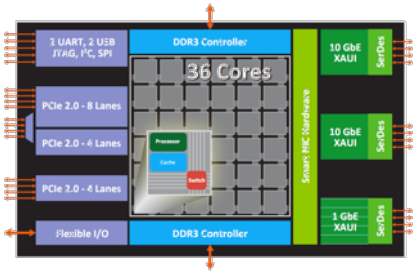
COMPUTE | STORE | ANALYZE

General Characteristics of These Architectures









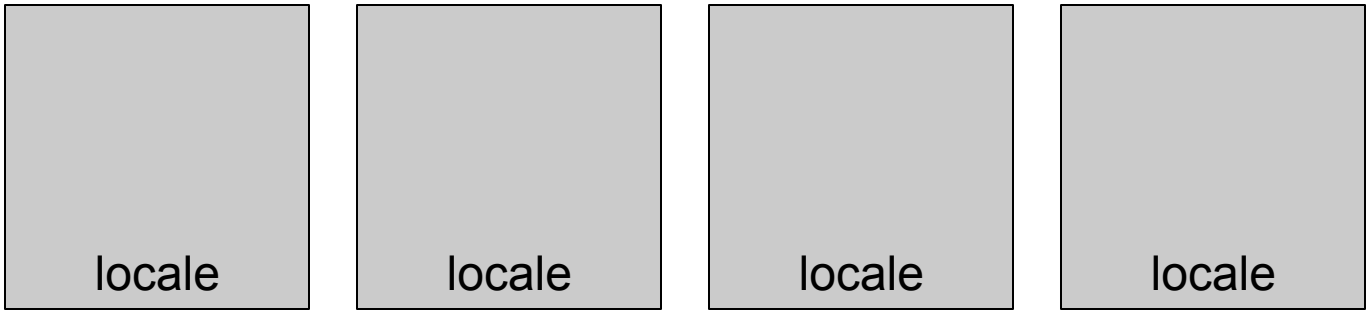
- **Increased hierarchy and/or sensitivity to locality**
- **Potentially heterogeneous processor/memory types**

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

Locales, Traditionally

Concept:

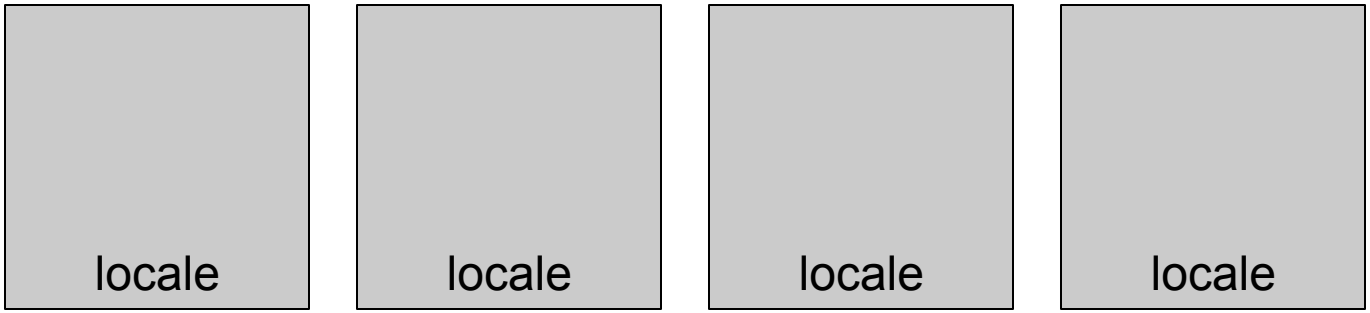
- Traditionally, Chapel has supported a 1D array of locales
  - users can reshape/slice to suit their computation's needs



Locales, Traditionally

Concept:

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

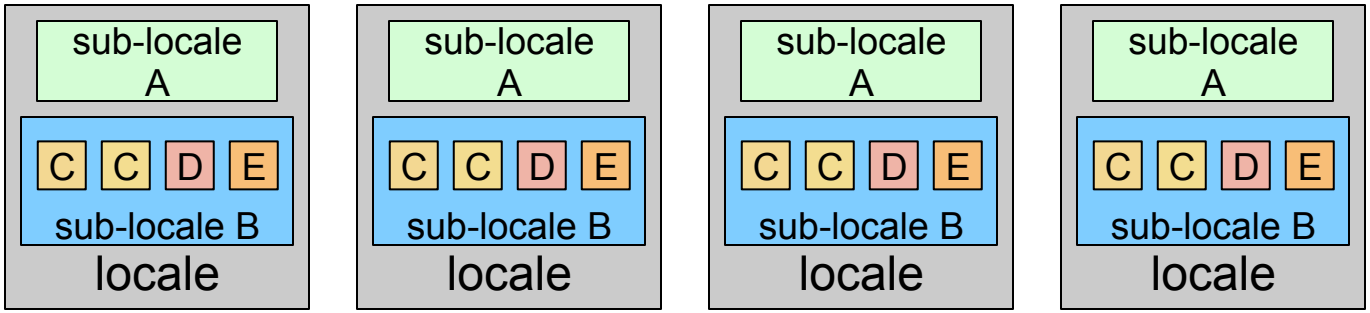


- Apart from queries, no further visibility into locales
  - no mechanism to refer to specific NUMA domains, processors, memories, …
  - assumption: compiler, runtime, OS, HW can handle intra-locale concerns
- Supports horizontal (inter-node) locality well
  - but not vertical (intra-node)

Hierarchical Locales

Concept:

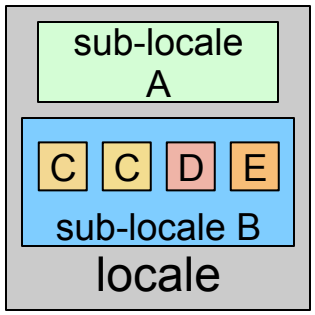
- Support locales within locales to describe architectural sub-structures within a node



- As with traditional locales, *on-clauses* and *domain maps* are used to map tasks and variables to sub-locales
- Locale structure is defined using Chapel code
  - permits architectural descriptions to be specified in-language
  - continues the multiresolution philosophy
  - introduces a new Chapel role: *architectural modeler*

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

For more information, come visit our Emerging Tech exhibit on this topic, this week on the show floor: booth #233!

Hierarchical Locale Summary

- **Most programming models assume a certain type of 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

Multiresolution 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 moves 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

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 on request)

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
  - these avoid locking crucial policy decisions into the language

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

Questions about Locality in Chapel?

Legal Disclaimer

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

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

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

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

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

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

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

CRAY
THE SUPERCOMPUTER COMPANY

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