Chapel Hierarchical Locales

Greg Titus, Chapel Team, Cray Inc.

SC14 Emerging Technologies

November 18th, 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.

Outline

- **Chapel introduction**
- The problem: architecture and how to express it
- The solution: *hierarchical locales*
- Locality during compilation
- Status and plans

What is Chapel?

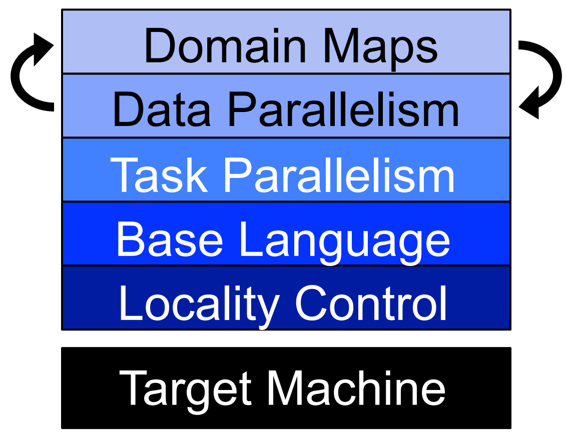
- **An emerging parallel programming language**
  - Design and development led by Cray Inc.
    - with contributions from academics, labs, industry
  - Initiated under the DARPA HPCS program
  - A work-in-progress
- **Overall goal: Improve programmer productivity**
- **Open source (at GitHub), licensed as Apache software**
- **Target architectures:**
  - Cray architectures
  - multicore desktops and laptops
  - commodity clusters
  - systems from other vendors
  - working on: CPU+accelerator hybrids, manycore, …

Multiresolution Design: a Root Concept in Chapel

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

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

Chapel language concepts



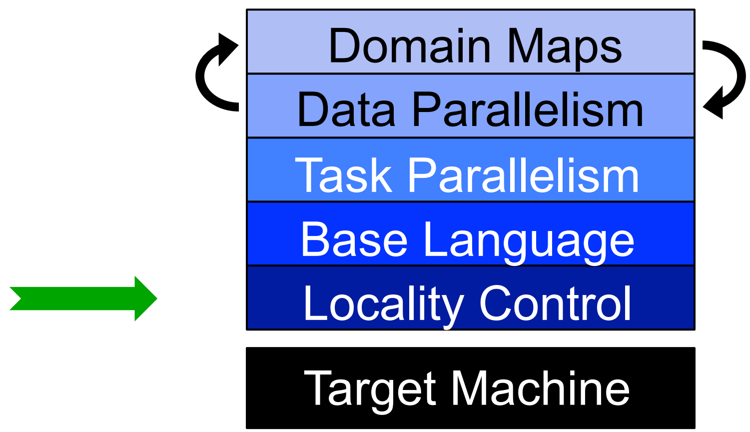
- build the higher-level concepts in terms of the lower
- permit the user to intermix layers arbitrarily

Multiresolution Design: a Root Concept in Chapel

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

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

Chapel language concepts



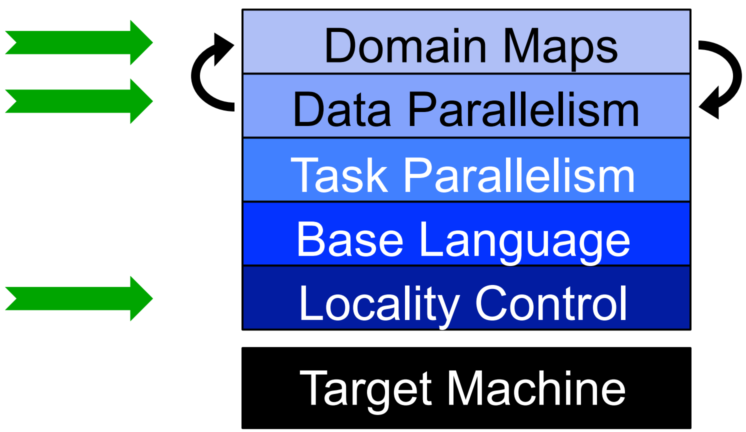
- build the higher-level concepts in terms of the lower
- permit the user to intermix layers arbitrarily

Multiresolution Design: a Root Concept in Chapel

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

Chapel in a Nutshell: Task Parallelism, etc.

Variables and types for reasoning about system resources:
Locales: the collection of compute nodes on which the program is running
here: the node on which the current task is running

Syntactic constructs for creating task parallelism:
coforall (concurrent forall): creates a task per iteration

Control over locality/affinity:
on-clauses: task migration

Static type inference (optionally):
Supports programmability with performance

taskParallel.chpl

```
coforall loc in Locales do
  on loc {
    const numTasks = here.maxTaskPar;
    coforall tid in 1..numTasks do
      writef("Hello from task %n of %n running on %s\n",
                tid, numTasks, here.name);
  }
```

```
prompt> chpl taskParallel.chpl -o taskParallel
prompt> ./taskParallel --numLocales=2
Hello from task 1 of 4 running on n1032
Hello from task 4 of 4 running on n1032
Hello from task 2 of 4 running on n1033
Hello from task 1 of 4 running on n1033
Hello from task 3 of 4 running on n1032
Hello from task 3 of 4 running on n1033
Hello from task 2 of 4 running on n1032
Hello from task 4 of 4 running on n1033
```

Chapel in a Nutshell: Data Parallelism, etc.

Domain maps
Describe how iterations over domains/arrays are mapped to locales

Modules for namespace management:
CyclicDist: standard module providing cyclic distributions

Configuration variables and constants:
Never write an argument parser again (unless you want to)

Domains and Arrays:
Index sets and arrays that can optionally be distributed

Data parallel forall loops and operations:
Use available parallelism for data-driven computations
Domain map *iterator* controls domain traversal, including parallelism and locality

dataParallel.chpl

```
use CyclicDist;
config const n = 1000;
var D = {1..n, 1..n}
        dmapped Cyclic(startIdx = (1,1));
var A: [D] real;
forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

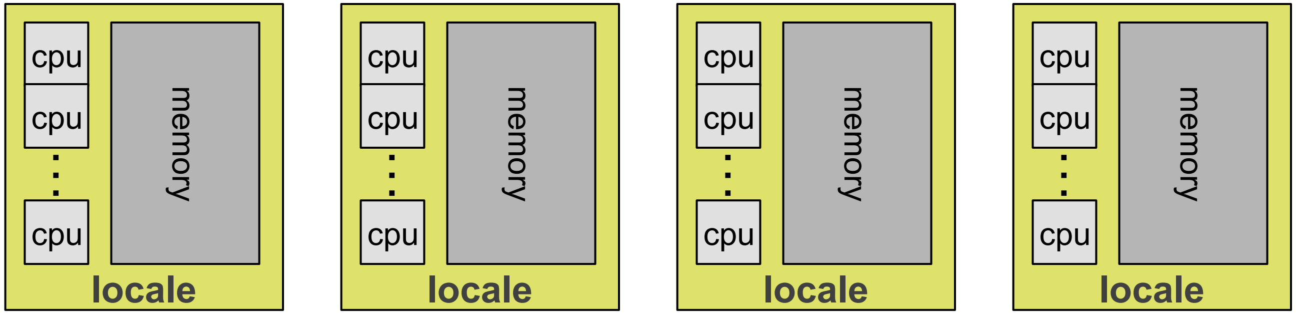
```
prompt> chpl dataParallel.chpl -o dataParallel
prompt> ./dataParallel --numLocales=4 --n=5
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```

Outline

- ✓ **Chapel introduction**
- ➢ **The problem: architecture and how to express it**
- **The solution: *hierarchical locales***
- **Locality during compilation**
- **Status and plans**

Architecture Used to Be So Simple

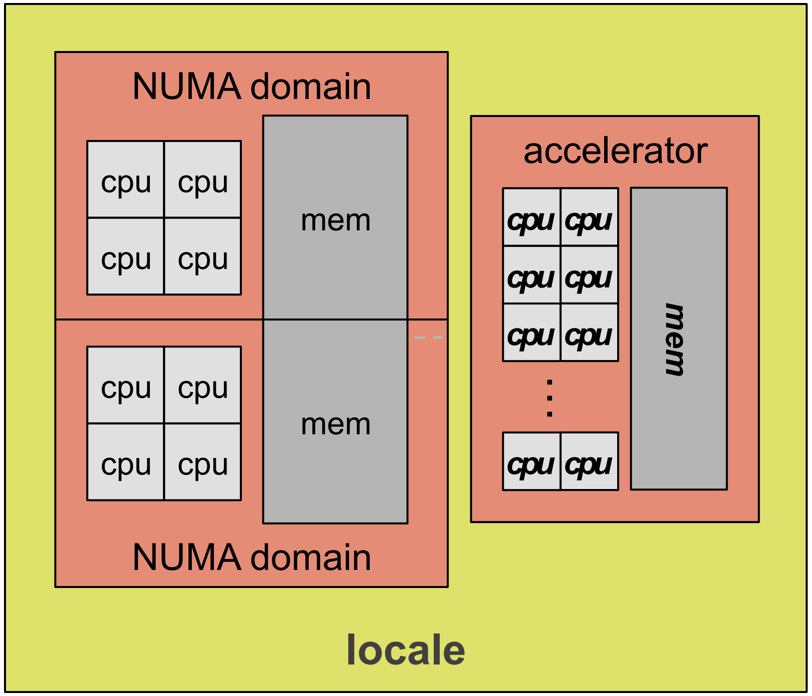
- **Traditionally, Chapel supported only a 1D array of locales**
  - Users could reshape/slice to suit their computation's needs



- **Apart from queries, no further visibility into locales**
  - No mechanism to refer to 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)

But the Old Model Was Really *too* Simple

- (HPC) architectures are varied and evolving rapidly
- Intra-node architecture has become important
  - Hierarchical (example: NUMA)
  - Heterogeneous (example: GPUs)
- Performance requires using *all* architecture effectively
- How to deal with this?



Summarizing the Requirements (and Desires)

- **Really just 3 classes of ops have to do with architecture:**
  - **Memory management (allocate, free, etc.)**
  - **Task support (initiate, move, etc.)**
  - **Communication**
  - Also helpful: we do not need very many operations from each class
- **Solution must be adaptive/flexible**
  - Must not require Chapel core team involvement
    - We are not architecture specialists
    - Others must be able to describe new architectures for Chapel
  - Knowing Chapel + architecture and being motivated should be enough
  - Must support experimentation and prototyping
- **Thus: fairly well constrained, not too-large problem**

Outline

- ✓ **Chapel introduction**
- ✓ **The problem: architecture and how to express it**
- ➢ **The solution: *hierarchical locales***
- **Locality during compilation**
- **Status and plans**

Chapel Hierarchical Locales

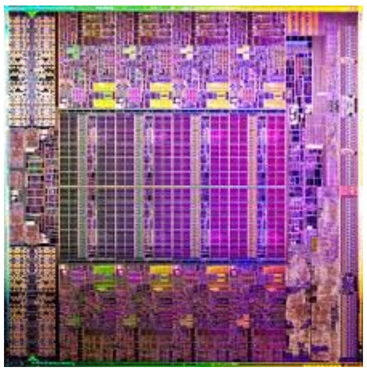
- **The key ideas:**
  - ➢ Define standardized Chapel class to describe CPU+mem architecture
  - ➢ Make it composable, to reflect hierarchy

```
class LocaleModel { … }
```

- **Has a required interface**
  - Functions for:
    - Memory management, task support, and communication operations
    - Parents and children
  - A few variables
    - "Has children?", e.g.
  - Compiler-generated code calls this required interface
  - May be implemented however you like
    - Typically in terms of other LocaleModel instances or runtime calls

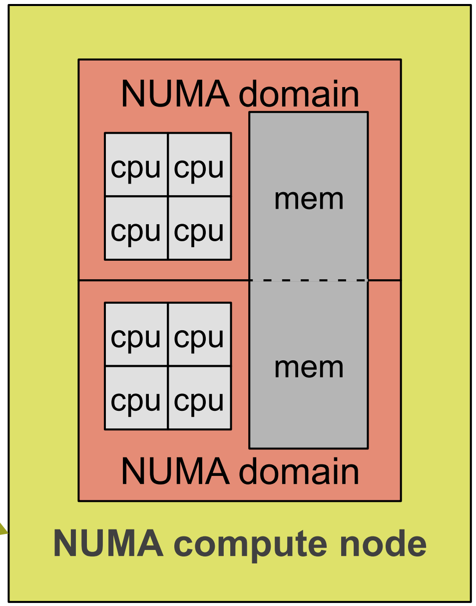
An Example: The numa Locale Model

physical



http://www1.pcmag.com/media/images/337192-intel-xeon-e5-chip.jpg?thumb=y

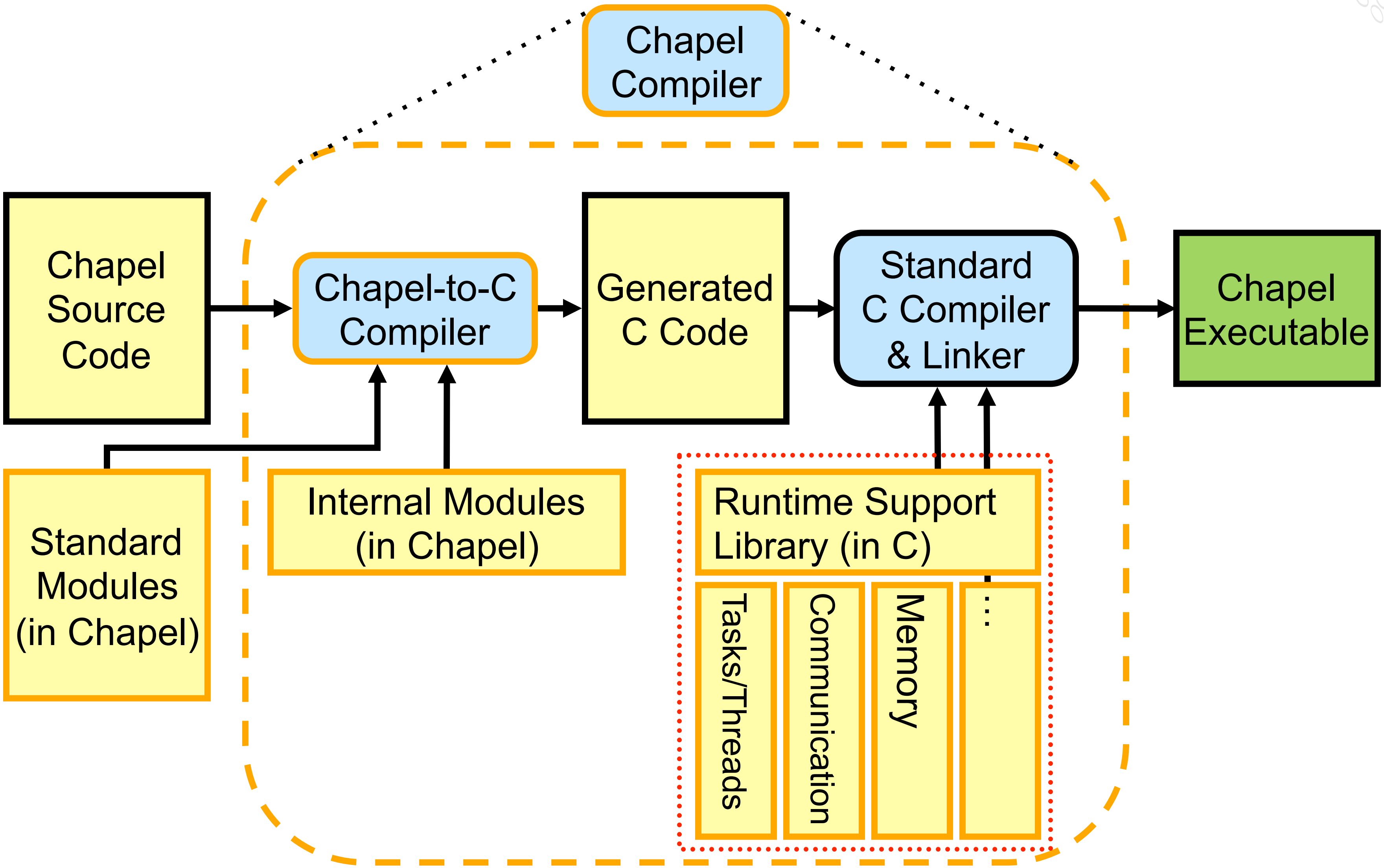
conceptual



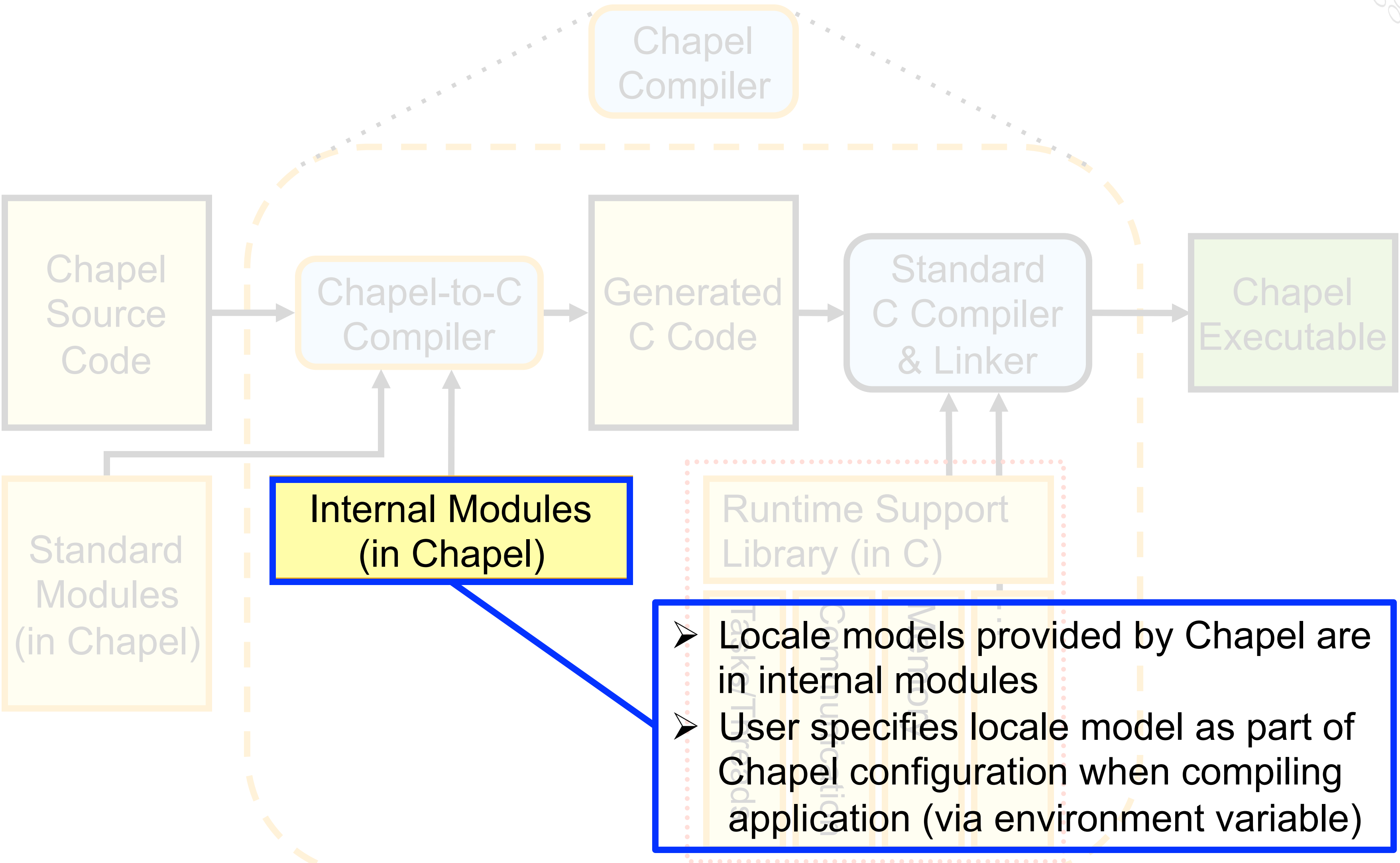
$CHPL_HOME/modules/…/numa/LocaleModel.chpl

```
class NumaDomain : AbstractLocaleModel {
  const sid: chpl_sublocID_t;
}

// The node model
class LocaleModel : AbstractLocaleModel {
  const numSublocales: int;
  var childSpace: domain(1);
  var childLocales: [childSpace] NumaDomain;
}

// support for memory management
proc chpl_here_alloc(size:int, md:int(16)) { … }

// support for "on" statements
proc chpl_executeOn
      (loc: chpl_localeID_t, // target locale
       fn: int,              // on-body func idx
       args: c_void_ptr,     // func args
       args_size: int(32)    // args size
      ) { … }

// support for tasking stmts: begin, cobegin, coforall
proc chpl_taskListAddCoStmt
      (subloc_id: int,         // target subloc
       fn: int,                // body func idx
       args: c_void_ptr,       // func args
       ref tlist: _task_list,  // task list
       tlist_node_id: int      // task list owner
      ) { … }
```

Where Predefined Locale Models Live



Where Predefined Locale Models Live



- Locale models provided by Chapel are in internal modules
- User specifies locale model as part of Chapel configuration when compiling application (via environment variable)

Hierarchical Locales Create A New Chapel Role

- **Application programmer: work on applications**
  - Express solutions in a natural way
  - Use forall statements to expose data parallelism
  - Use domain maps to inform Chapel about locality and affinity

Hierarchical Locales Create A New Chapel Role

- **Application programmer: work on applications**
  - Express solutions in a natural way
  - Use forall statements to expose data parallelism
  - Use domain maps to inform Chapel about locality and affinity

- **Domain map specialist: work on locality**
  - In a general or conceptual way, not an architecture-specific one

Hierarchical Locales Create A New Chapel Role

- **Application programmer: work on applications**
  - Express solutions in a natural way
  - Use forall statements to expose data parallelism
  - Use domain maps to inform Chapel about locality and affinity

- **Domain map specialist: work on locality**
  - In a general or conceptual way, not an architecture-specific one

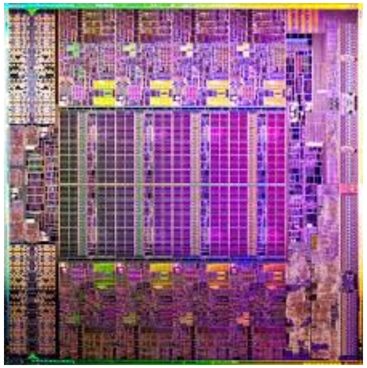
★ **Architecture modeler: work on architectural mappings**
  - Describe architectural hierarchy
  - Implement functional interfaces at various levels

Outline

- ✓ Chapel introduction
- ✓ The problem: architecture and how to express it
- ✓ The solution: *hierarchical locales*
- ➢ Locality during compilation
- Status and plans

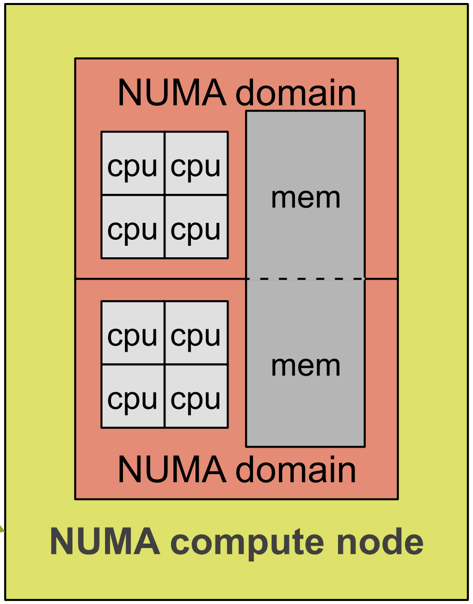
Context: We're Using the numa Locale Model

physical



http://www1.pcmag.com/media/images/337192-intel-xeon-e5-chip.jpg?thumb=y

conceptual

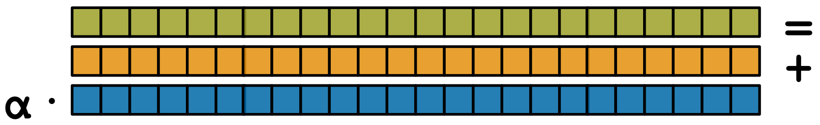


$CHPL_HOME/modules/…/numa/LocaleModel.chpl

```
class NumaDomain : AbstractLocaleModel {
  const sid: chpl_sublocID_t;
}

// The node model
class LocaleModel : AbstractLocaleModel {
  const numSublocales: int;
  var childSpace: domain(1);
  var childLocales: [childSpace] NumaDomain;
}

// support for memory management
proc chpl_here_alloc(size:int, md:int(16)) { … }

// support for "on" statements
proc chpl_executeOn
     (loc: chpl_localeID_t, // target locale
      fn: int,              // on-body func idx
      args: c_void_ptr,     // func args
      args_size: int(32)    // args size
     ) { … }

// support for tasking stmts: begin, cobegin, coforall
proc chpl_taskListAddCoStmt
     (subloc_id: int,        // target subloc
      fn: int,               // body func idx
      args: c_void_ptr,      // func args
      ref tlist: _task_list, // task list
      tlist_node_id: int     // task list owner
     ) { … }
```

The Application, Unburdened by Architecture

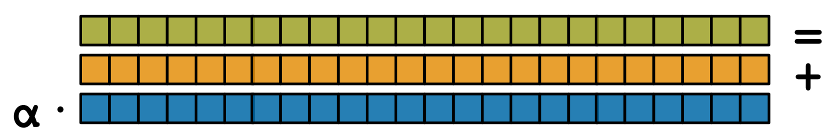
```
// Stream Triad
config const m = 1000,
              alpha = 3.0;
const ProblemSpace = {1..m} dmapped Block(…);
var A, B, C: [ProblemSpace] real;
B = 2.0;
C = 3.0;
A = B + alpha * C;
```



The Application, Unburdened by Architecture

Express parallelism abstractly, without referring to physical architecture

```
// Stream Triad
config const m = 1000,
             alpha = 3.0;
const ProblemSpace = {1..m} dmapped Block(…);
var A, B, C: [ProblemSpace] real;
B = 2.0;
C = 3.0;
A = B + alpha * C;
```

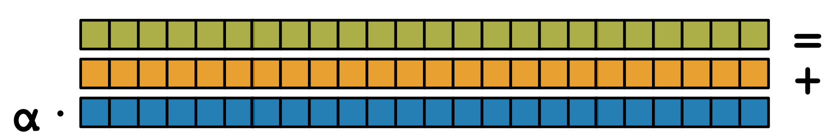


The Application, Unburdened by Architecture

Specify domain map in application code

Express parallelism abstractly, without reference to architecture

```
// Stream Triad
config const m = 1000,
             alpha = 3.0;
const ProblemSpace = {1..m} dmapped Block(…);
var A, B, C: [ProblemSpace] real;
B = 2.0;
C = 3.0;
A = B + alpha * C;
```



Locality & Affinity in the Domain Map

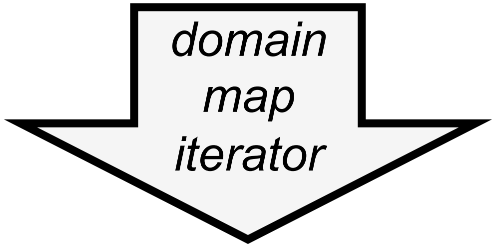
```
// Block domain map
class Block: BaseDist {
  var targetLocDom: domain(rank);
  var targetLocales: [targetLocDom] locale;
  var dataParTasksPerLocale: int;
  var dataParIgnoreRunningTasks: bool;
  var dataParMinGranularity: int;
}
…
iter these(param tag: iterKind,
       tasksPerLocale = dataParTasksPerLocale,
       ignoreRunning = dataParIgnoreRunningTasks,
       minIndicesPerTask = dataParMinGranularity)
{
  const numSublocs = here.getChildCount();
  if locModelHasSublocs && numSublocs != 0 {
    … _computeChunkStuff(min(numSublocs,
                             here.maxTaskPar),
                         ignoreRunning,
                         minIndicesPerTask,
                         ranges);
    …
  }
}
```

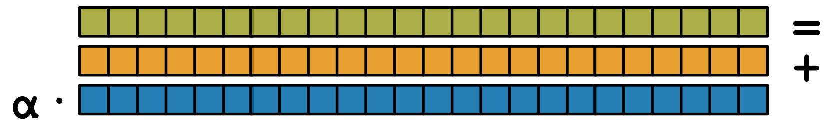
Domain map:

- Describes distribution of indices (block, cyclic, etc.)
- Ties together locality, affinity, parallelism via iterators for forall-stmts
- Interrogates locale model to learn about resources
- Has a standardized interface, referenced by compiler-generated code
- Is typically coded by a specialist

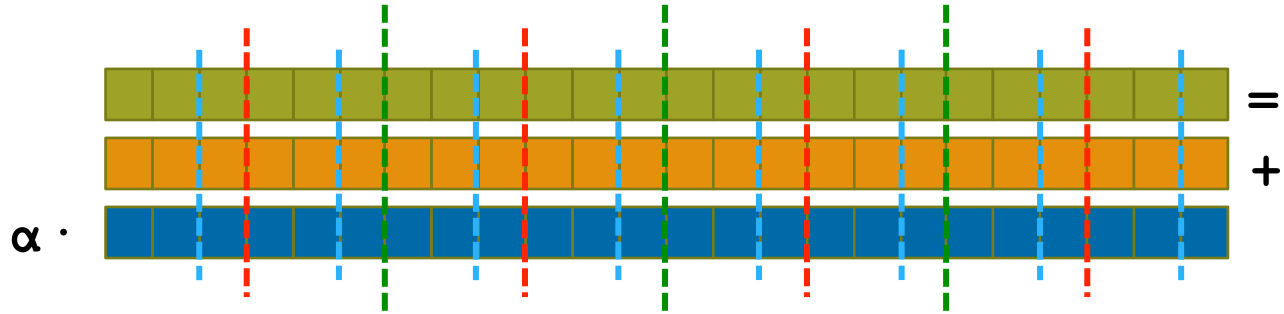
The Application, Translated by the Domain Map

```
const ProblemSpace = {1..m} dmapped Block(…);
var A, B, C: [ProblemSpace] real;
A = B + alpha * C;
```






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



… and Translated Again, by the Compiler

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

Chapel code

Chapel compiler

```
void main(…) {
  chpl_taskListAddCoStmt(fn_for_outer_coforall_stmt);
}
void fn_for_outer_coforall_stmt(…) {
  chpl_executeOn(loc, fn_for_on_stmt);
}
void fn_for_on_stmt(…) {
  chpl_taskListAddCoStmt(fn_for_middle_coforall_stmt);
}
void fn_for_middle_coforall_stmt(…) {
  chpl_taskListAddCoStmt(fn_for_inner_coforall_stmt);
}
void fn_for_inner_coforall_stmt(…) {
  for (…) { a[i] = b[i] + alpha * c[i]; }
}
```

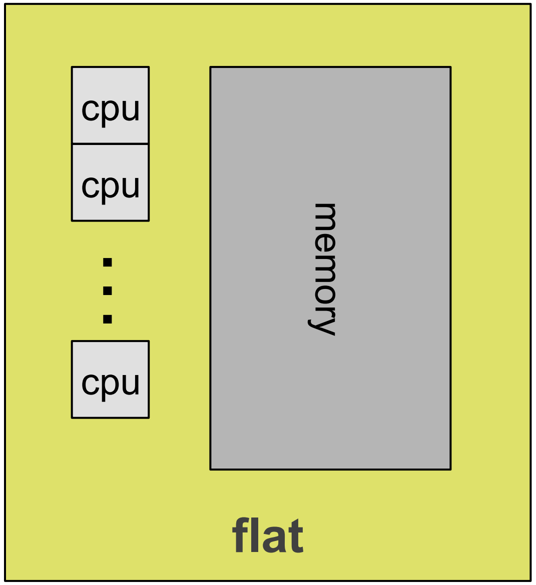
C code

Outline

- ✓ Chapel introduction
- ✓ The problem: architecture and how to express it
- ✓ The solution: *hierarchical locales*
- ✓ Locality during compilation
- ➢ Status and plans

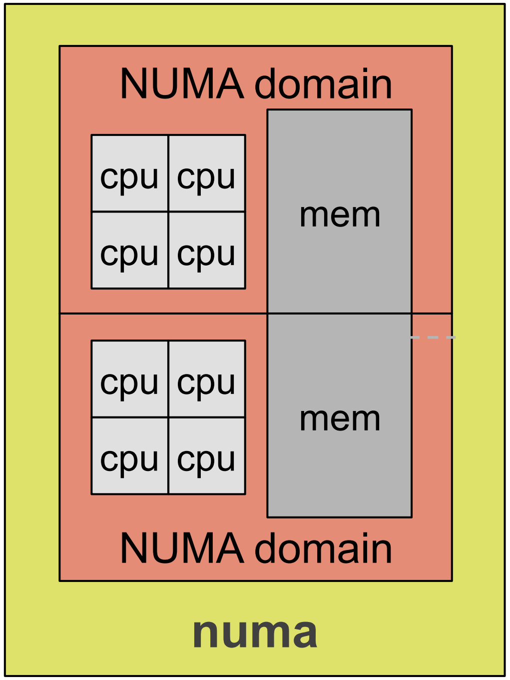
Today's Locale Models: flat

- Direct replacement for the old compiler-implemented model
- Same performance as old compiler-based architecture support
- Default in all cases



Today's Locale Models: numa

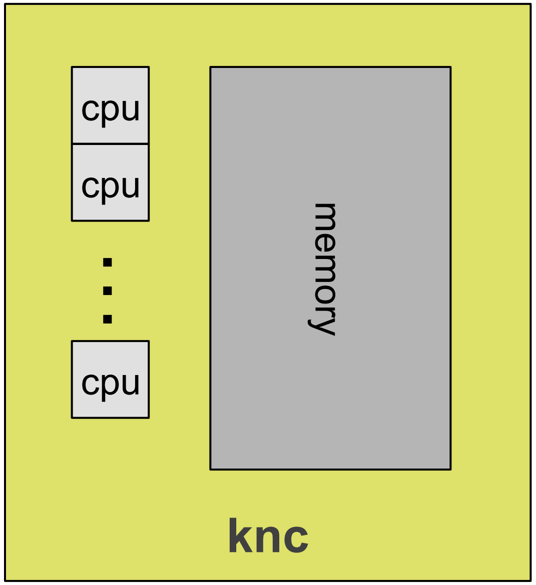
- **Fully functional**
- **Needs tuning**
  - Tasking affinity with memory locality works properly
  - But memory locality itself needs work



Tomorrow's Locale Models: "real" knc

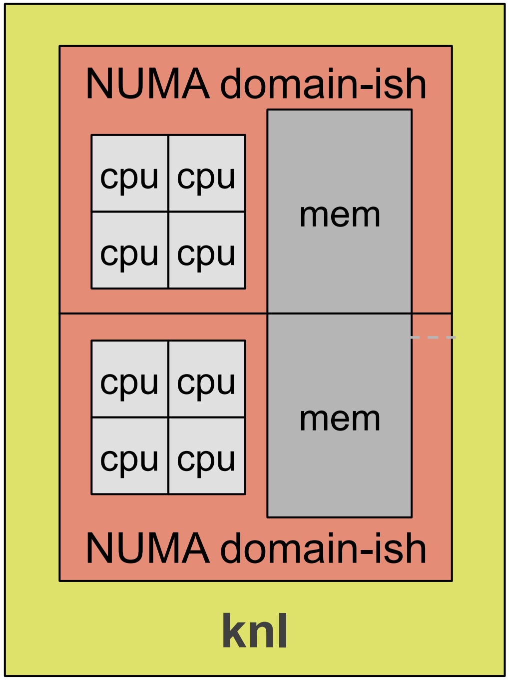
- Current Chapel Intel Xeon Phi KNC support uses "flat"

- Duplicate and tune for KNC-specific properties (breadth, e.g.)



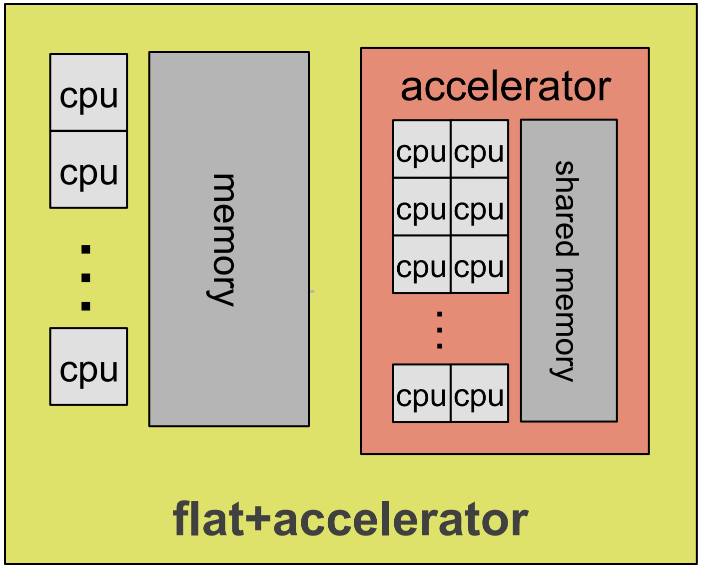
Tomorrow's Locale Models: knl

- **Intel Xeon Phi KNL would be an elaboration of numa**
  - Similar to flat → knc



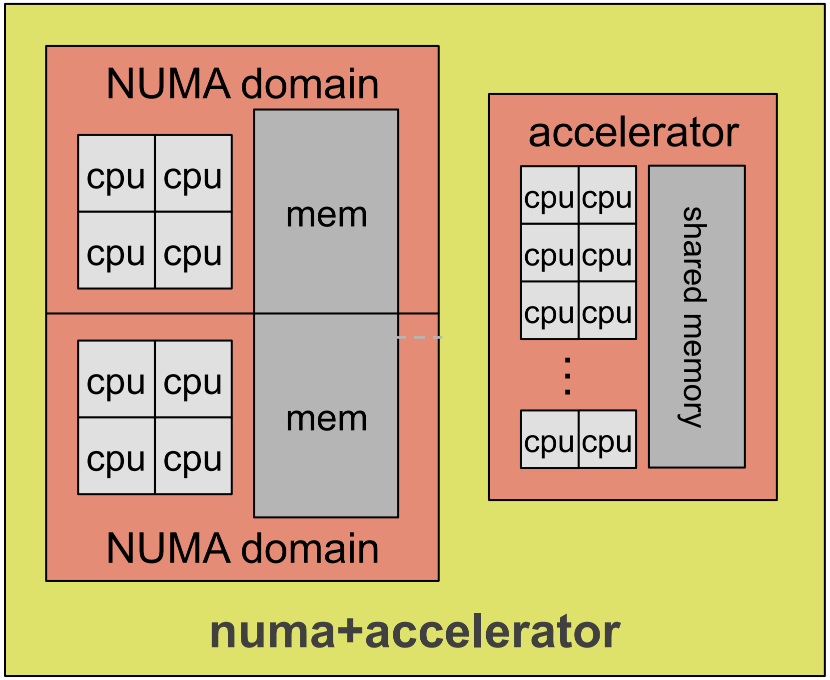
Tomorrow's Locale Models: accelerator

- Challenge: processor heterogeneity



Tomorrow's Locale Models: numa+accelerator

- **Challenge: hierarchy *and* heterogeneity**
- **Great composability test**



Improving the Implementation

- **Today's locale model implementations could be cleaner**
  - Reflect some legacy of prototyping and experimentation
- **Would like to improve things before adding more models**
  - Restructure to remove duplication
  - Split into "building block" and "compute node" instances

Summary

- Hierarchical Locales feature helps "future proof" Chapel

- Enables separation of concerns
  - **Application programmers** are freed from architecture concerns
  - **Domain map programmers** are freed from architecture concerns
  - **Compiler** is freed from architecture concerns
  - Even the **Chapel language** is freed from architectural concerns

- Puts Chapel architectural policy in the hands of those most qualified to deal with it: architecture experts

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.