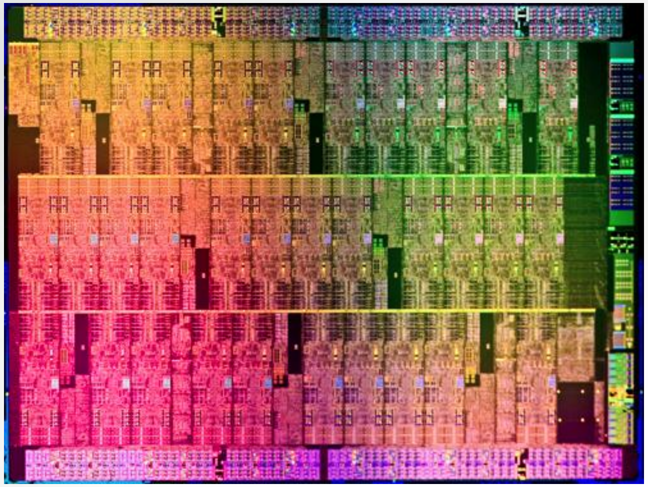
Chapel Support for Heterogeneous Architectures via Hierarchical Locales

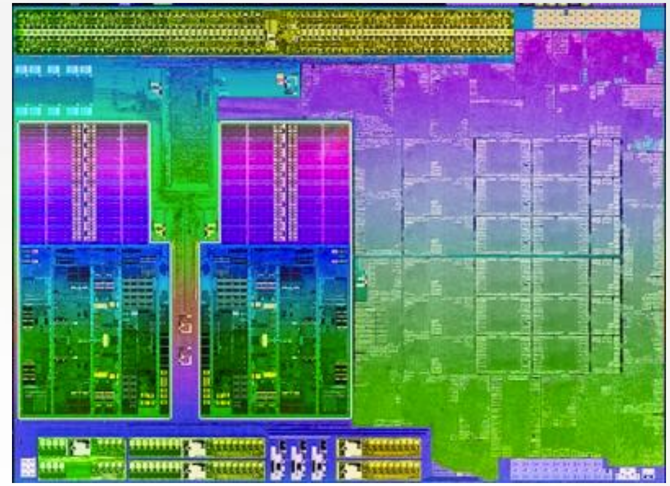
Brad Chamberlain, Tom Hildebrandt: Cray Inc.
Casey Battaglino: Georgia Tech

PGAS-X Workshop: October 10, 2012

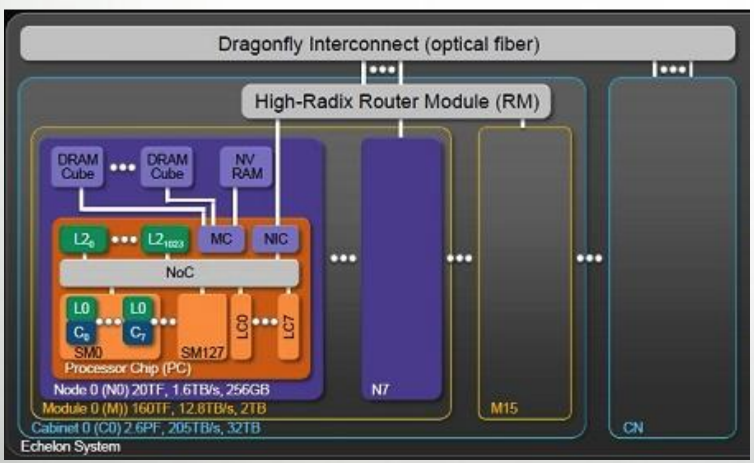
Prototypical Next-Gen Processor Technologies



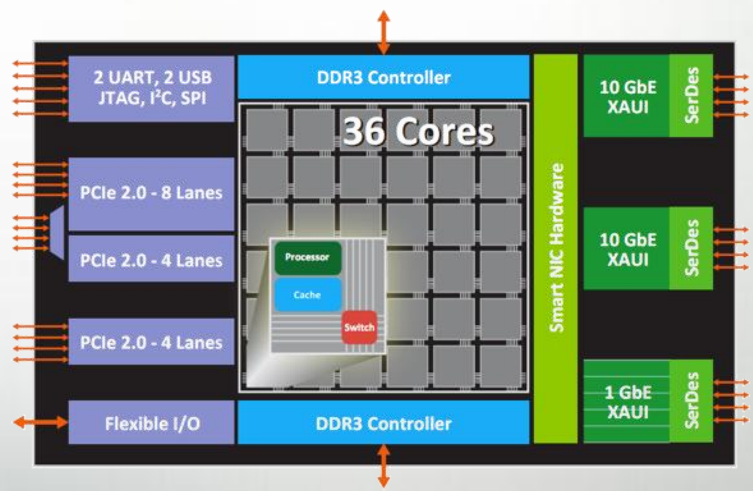
Intel MIC



AMD Trinity

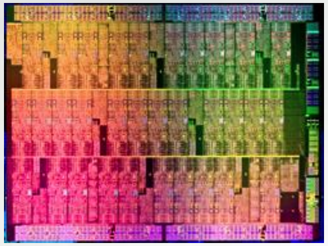


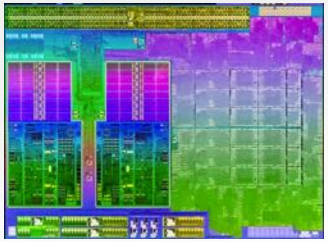
Nvidia Echelon

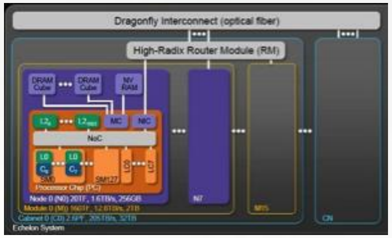


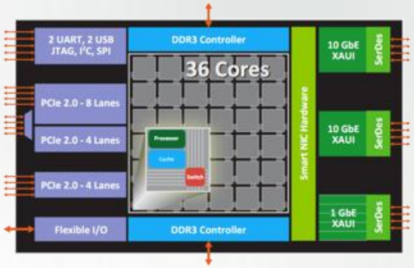
Tilera Tile-Gx

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

Next-Gen Tech Programming Model Wishlist

performance: (naturally)

portability: specifically, to/between next-generation architectures

programmability features: because you know you want them

general parallelism:

- **data parallelism:** to take advantage of SIMD HW units; for simplicity
- **task parallelism:** for asynchronous computations; data-driven algorithms
- **varying granularities/nestings:** for algorithmic and architectural generality

locality control: to tune for locality/affinity across the machine (inter- and intra-node)

resilience-/energy-aware features: to deal with emerging issues at system scale

user extensibility: to be ready for next-generation unknowns

Next-Gen Scorecard for HPC Programming Models

	Fortran	C/C++	MPI	OpenMP	UPC
performance					
portability (to next-gen)					
programmability					
data parallelism					
task parallelism					
parallel nesting/granularities					
locality control					
resilience					
energy-awareness					
user-extensibility					

Next-Gen Scorecard for HPC Programming Models

	Fortran	C/C++	MPI	OpenMP	UPC
performance	✓	✓	✓	✓	~
portability (to next-gen)					
programmability					
data parallelism					
task parallelism					
parallel nesting/granularities					
locality control					
resilience					
energy-awareness					
user-extensibility					

Next-Gen Scorecard for HPC Programming Models

	Fortran	C/C++	MPI	OpenMP	UPC
performance	✓	✓	✓	✓	~
portability (to next-gen)	✓	✓			
programmability					
data parallelism					
task parallelism					
parallel nesting/granularities					
locality control					
resilience					
energy-awareness					
user-extensibility					

Next-Gen Scorecard for HPC Programming Models

	Fortran	C/C++	MPI	OpenMP	UPC
performance	✓	✓	✓	✓	~
portability (to next-gen)	✓	✓	~	~	~
programmability	X	X	X	~	X
data parallelism	~	X	X	~	~
task parallelism	X	X	X	~	X
parallel nesting/granularities	X	X	X	~	X
locality control	X	X	~	X	~
resilience	X	X	~	X	X
energy-awareness	X	X	X	X	X
user-extensibility	X	X	X	X	X

Chapel: Well-Positioned for Next-Gen

performance	~
portability (to next-gen)	~*
programmability	✓
data parallelism	✓
task parallelism	✓
parallel nesting/granularities	✓
locality control	~*
resilience	X
energy-awareness	X
user-extensibility	✓

* (The work in this talk strives to address these items)

Outline

- ✓ Motivation
- ➢ Chapel Background
- Hierarchical Locales in Chapel
- Approach, Status, and Summary

What is Chapel?

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

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

- A work-in-progress

Chapel's Implementation

- Being developed as open source at SourceForge
- Licensed as BSD software
- **Target Architectures:**
  - Cray architectures
  - multicore desktops and laptops
  - commodity clusters
  - systems from other vendors
  - *in-progress:* CPU+accelerator hybrids, manycore, …

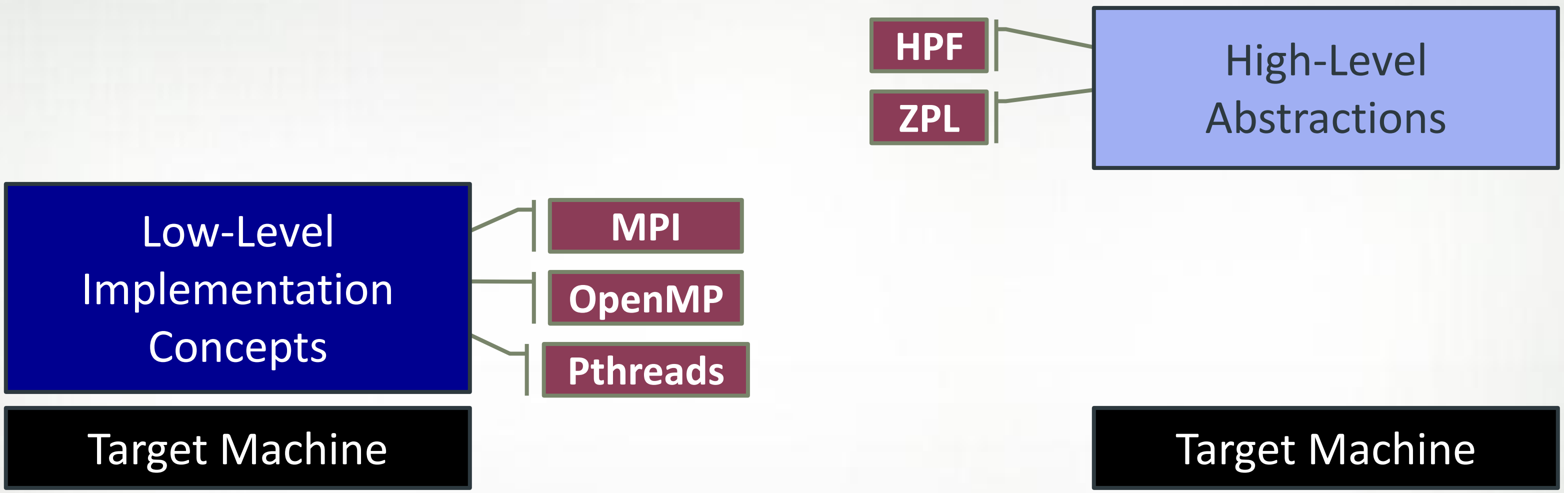
Chapel's Greatest Hits

- Multiresolution Language Design Philosophy
- User-Defined Parallel Iterators, Layouts, and Distributions
- Distinct Concepts for Parallelism and Locality
- Multithreaded Execution Model
- Unification of Data- and Task-Parallelism
- Productive Base Language Features
  - type inference, iterators, tuples, ranges
- Portable Design, Open-Source Implementation
  - Yet, able to take advantage of HW-specific capabilities
- Helped revitalize Community Interest in Parallel Languages

Chapel's Greatest Hits

- Multiresolution Language Design Philosophy
- User-Defined Parallel Iterators, Layouts, and Distributions
- Distinct Concepts for Parallelism and Locality
- Multithreaded Execution Model
- Unification of Data- and Task-Parallelism
- Productive Base Language Features
  - type inference, iterators, tuples, ranges
- Portable Design, Open-Source Implementation
  - Yet, able to take advantage of HW-specific capabilities
- Helped revitalize Community Interest in Parallel Languages

Multiresolution Design: Motivation



"Why is everything so tedious/difficult?"

"Why don't my programs port trivially?"

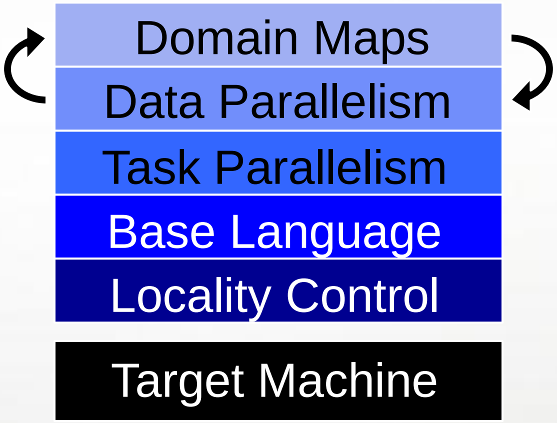
"Why don't I have more control?"

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
  - examples: array distributions and layouts; forall loop implementations
- permit the user to intermix layers arbitrarily

Distinct Concepts for Parallelism and Locality

Consider:

- Most HPC languages couple parallelism and locality
  - e.g., I can't create parallelism in MPI/UPC without also introducing locality
- Or, they don't support a concept for locality at all
  - e.g., OpenMP (though it's working on improving this)

Yet these are distinct, important things!

(and, getting more important with time)

- parallelism: "Please execute these at the same time"
- locality: "Do this here rather than there"

For this reason, Chapel supports distinct concepts

- parallelism: tasks
- locality: locales

The Locale Type

Definition:

- Abstract unit of target architecture
- Supports reasoning about locality
- Capable of running tasks and storing variables
  - i.e., has processors and memory

Typically: A compute node (multi-core processor or SMP node)

Defining 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 = …;
```

Locales: | L0 | L1 | L2 | L3 | L4 | L5 | L6 | L7 |

Locale Operations

- Locale methods support queries about target system:

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

- *On-clauses* support placement of computations:

```
writeln("on locale 0");

on Locales[1] do
  writeln("now on locale 1");

writeln("on locale 0 again");
```

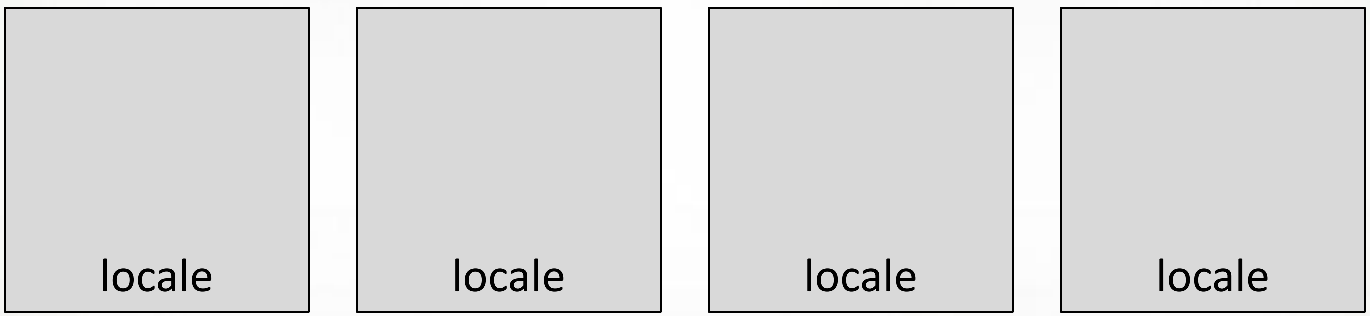
```
cobegin {
  on A[i,j] do
    bigComputation(A);

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

Locales Today

Concept:

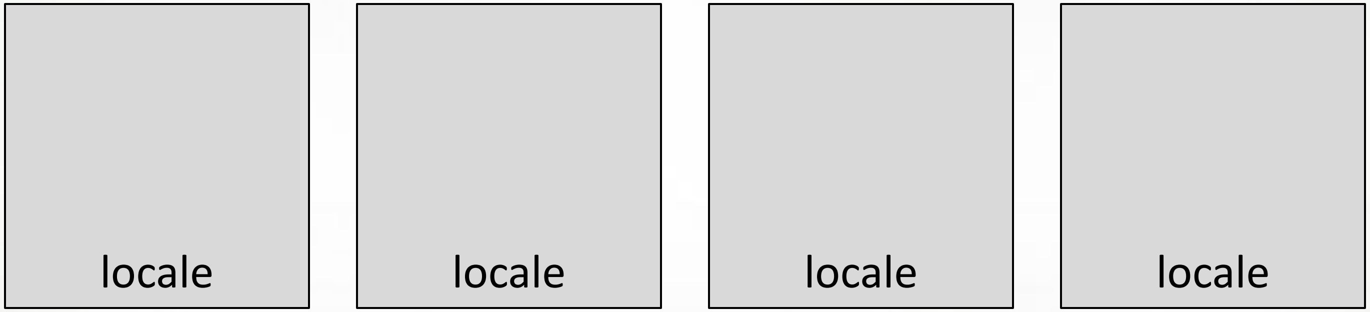
- Today, Chapel supports a 1D array of locales
  - users can reshape/slice to suit their computation's needs



Locales Today

Concept:

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

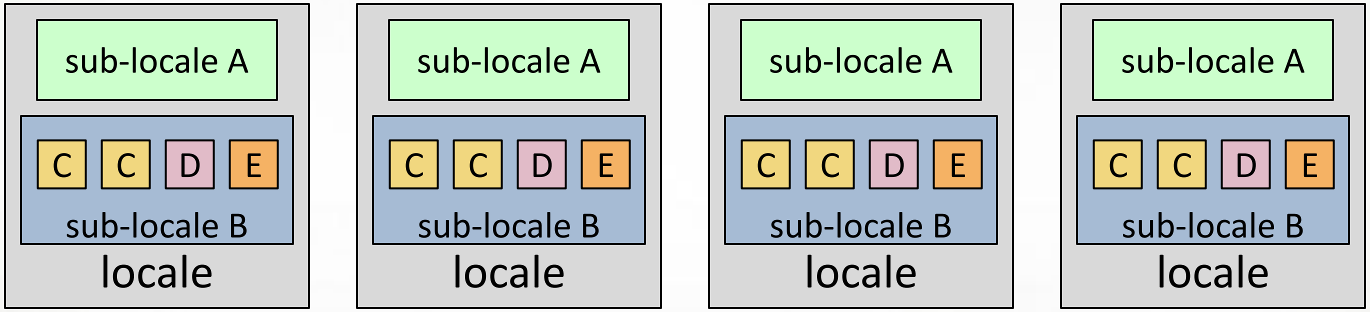


- Apart from locale queries, no further visibility into locale
  - no mechanism to refer to specific NUMA domains, processors, memories, …
  - assumption: compiler, runtime, OS, HW can handle intra-locale concerns

Current Work: Hierarchical Locales

Concept:

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

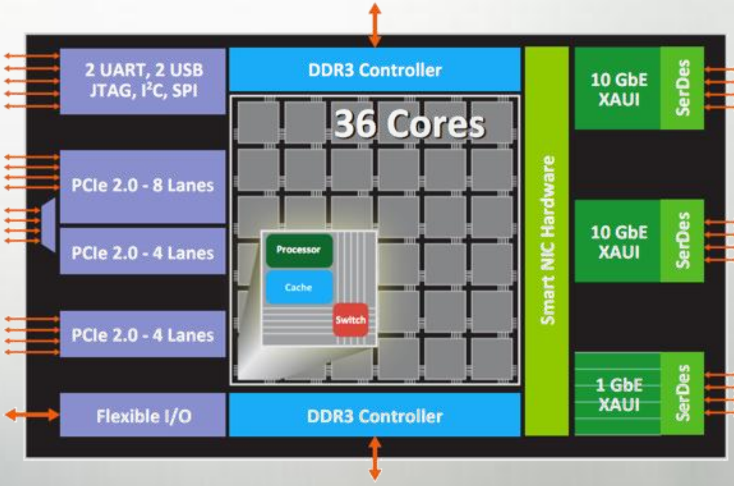


- As with traditional locales, *on-clauses* and *domain maps* should be used to map tasks and variables to a sub-locale's memory and processors
- Locale structure is defined as Chapel code
  - permits implementation policies to be specified in-language
  - introduces a new Chapel role: *architectural modeler*

Sublocales: Tiled Processor Example

```
class locale: AbstractLocale {
  const xt = 6, yt = xTiles;
  const sublocGrid: [0..#xt, 0..#yt] tiledLoc = …;
  const allSublocs: [0..#xt*yt] tiledLoc = …;
  …memory interface…
  …tasking interface…
}

class tiledLoc: AbstractLocale {
  …memory interface…
  …tasking interface…
}
```



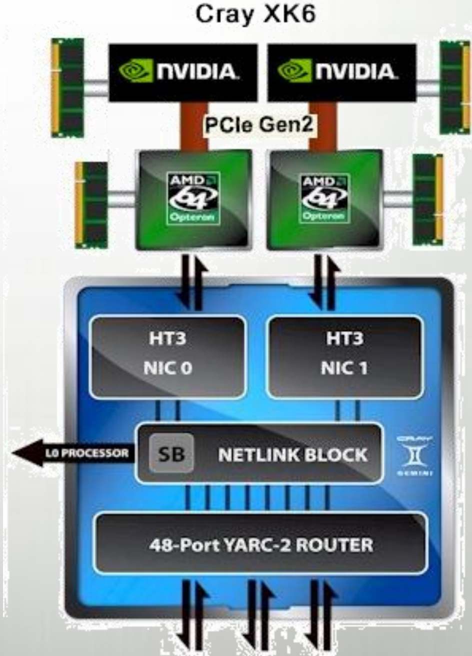
Tilera Tile-Gx

Sublocales: Hybrid Processor Example

```
class locale: AbstractLocale {
  const numCPUs = 2, numGPUs = 2;
  const cpus: [0..#numCPUs] cpuLoc = …;
  const gpus: [0..#numGPUs] gpuLoc = …;
  …memory interface…
  …tasking interface…
}

class cpuLoc: AbstractLocale { … }

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



Sample tasking/memory interface

Memory Interface:

```
proc AbstractLocale.malloc(size_t size) { … }
proc AbstractLocale.realloc(size_t size) { … }
proc AbstractLocale.free(size_t size) { … }
…
```

Tasking Interface:

```
proc AbstractLocale.taskBegin(…) { … }
proc AbstractLocale.tasksCobegin(…) { … }
…
```

In practice, we expect the guts of these to be implemented via calls out to external C routines

Policy Questions

Memory Policy Questions:

- If a sublocale is out of memory, what happens?
  - out-of-memory error?
  - allocate elsewhere? sibling? parent? somewhere else? (on-node v. off?)
- What happens on locales with no memory?
  - illegal? allocate on sublocale? somewhere else?

Tasking Policy Questions:

- Can a task that's placed on a specific sublocale migrate?
  - to where? sibling? parent? somewhere else?
- What happens on locales with no processors?
  - illegal? allocate on sublocale? parent locale?
  - using what heuristic? sublocale[0]? round-robin? dynamic load balance?

Goal: Any of these policies should be possible

Tasking Policy Example

Q: What happens to tasks on locales with no processors?

e.g., a sublocale representing a memory resource

Tasking Policy Example

Q: What happens to tasks on locales with no processors?

e.g., a sublocale representing a memory resource

A1: Throw an error?

```
proc TextureMemLocale.taskBegin(…) {
  halt("You can't run tasks on texture memory!");
}
```

Downside: potential user inconvenience:

```
on Locales[2].gpuLoc.texMem do var X: [1..n, 1..n] int;
on X[i,j] do begin refine(X);
```

Tasking Policy Example

Q: What happens to tasks on locales with no processors?

e.g., a sublocale representing a memory resource

A2: Defer to parent?

```
proc TextureMemLocale.taskBegin(…) {
  parentLocale.taskBegin(…);
}
```

Tasking Policy Example

Q: What happens to tasks on locales with no processors?

e.g., a sublocale representing a memory resource

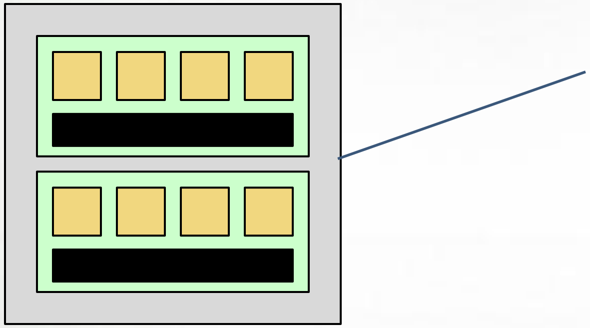
A3: Or perhaps just run directly near memory?

```
proc TextureMemLocale.taskBegin(…) {
  extern proc chpl_task_create_GPU_Task(…);
  chpl_task_create_GPU_Task(…);
}
```

Another Tasking Policy Example

Q: What happens to tasks on locales with no (direct) processors?

e.g., a locale that serves as a container for other sublocales



on “multicore NUMA Node” do begin foo()

Another Tasking Policy Example

Q: What happens to tasks on locales with no (direct) processors?

e.g., a locale that serves as a container for other sublocales

A1: Run on a fixed or arbitrary sublocale?

```
proc NUMANode.taskBegin(…) {
  numaDomain[0].taskBegin(…);
}
```

Another Tasking Policy Example

Q: What happens to tasks on locales with no (direct) processors?

e.g., a locale that serves as a container for other sublocales

A2: Schedule round-robin?

```
proc NUMANode.taskBegin(…) {
  const subloc = (nextSubLoc.fetchAdd(1))%numSubLocs;
  numaDomain[subloc].taskBegin(…);
}

class NUMANode {
  …
  var nextSubLoc: atomic int;
  …
}
```

Another Tasking Policy Example

Q: What happens to tasks on locales with no (direct) processors?

e.g., a locale that serves as a container for other sublocales

A3: Dynamically Load Balance?

```
proc NUMANode.taskBegin(…) {
  numaDomain[getBestSubLoc()].taskBegin(…);
}

proc NUMANode.getBestSubLoc() {
  const (numTasks, subloc)
          = minloc reduce (numaDomain.numTasks(),
                           0..#numSubLocs);
  return subloc;
}
```

Contrasts with Related Work

Related work:

- Sequoia (Aiken et al., Stanford)
- Hierarchical Place Trees (Sarkar et al., Rice)

Differences:

- Hierarchy only impacts locality, not semantics as in Sequoia
  - analogous to PGAS languages vs. distributed memory
- No restrictions as to what HW must live in what node
  - i.e., no "processors must live in leaf nodes" requirement
- Does not impose a strict abstract tree structure
  - e.g., `const sublocGrid: [0..#xt, 0..#yt] tiledLoc = …;`
- User-specifiable concept
  - convenience of specifying within Chapel
  - mapping policies can be defined in-language

Outline

- ✓ Motivation
- ✓ Chapel Background
- ✓ Hierarchical Locales in Chapel
- ➢ Approach, Status, and Summary

Organization of Implementation

Today:

- Locales are defined using Chapel code, but a single definition is used for all target platforms
  - see: `modules/internal/ChapelLocale.chpl`
- Task/Memory/Comm. interfaces are baked into compiler
  - can switch between multiple implementations via env. vars.
  - but each executable only supports one implementation

Organization of Implementation

Plan for this work:

- Support multiple Locale definitions, selected by env. var.
  - e.g., CHPL_LOCALE_MODEL (defaults based on CHPL_TARGET_PLATFORM)
  - store locale models in subdirectories based on CHPL_LOCALE_MODEL:
    - `modules/locales/multiNUMA/ChapelLocale.chpl`
    - `modules/locales/CPUGPU/ChapelLocale.chpl`
  - –M $CHPL_HOME/modules/standard/$CHPL_LOCALE_MODEL added to search path

- Compiler can remain ignorant of runtime interfaces
  - one binary can support multiple tasking/memory models
  - interfaces need no longer be identical across implementations

Hierarchical Locales: Implementation Challenges

Locale ID/wide pointer representation: Simple integer ID no longer suffices

Representation of 'here': Global integer in generated C code no longer suffices

- 'here' must become task-private since different tasks will have different sublocales at a given time

Communication Generation: A function of two locale types, not one

(and they may not be known at compile-time)

Hierarchical Locales: Design Challenges

Portability: Chapel code that refers to sub-locales can cause problems on systems with different model

Mitigation Strategies

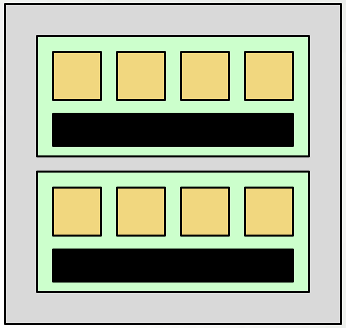
- Well-designed domain maps should buffer many typical users from these challenges
- We anticipate identifying a few broad classes of locales that characterize broad swaths of machines “well enough”
- More advanced runtime designs and compiler work could help guard most task-parallel users from this level of detail
- Not a Chapel-specific challenge, fortunately

Code Generation: Dealing with targets for which C is not the language of choice (e.g., CUDA)

Target 1: NUMA Nodes

Platform: multicore nodes with several NUMA domains

Approach:

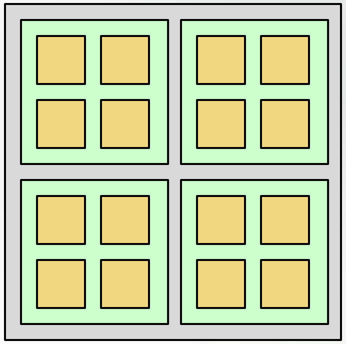


- two-level locale structure
  - outer: Complete node
  - inner: NUMA domain
  - (exposing cores/memories seems like overkill for now)
- Qthreads shepherd per NUMA domain for tasking

Why? Simple initial exercise with practical impact

Initial Goal: Support NUMA-aware STREAM Triad

Target 2: Tilera



Platform: Tilera tiled processor

Approach:

- 2-to-3 level locale structure
  - outer: Tiled processor
  - inner: OS instance (can be configured at various granularities)
  - potential for creating a sublocale per tile as well

Why? More interesting example w/ user interest

- reconfigurability, 2D layout particularly interesting

Initial Goal: Run Chapel codes using various Tilera configurations

- ideally, with single Chapel locale definition file

Status

Initial draft up and running:

- Two-level locale types defined as Chapel code
- Representing locale ID as a pair of 32-bit ints for now
- Draft memory and tasking interfaces implemented
- Sublocale-aware tasks being created
  - NUMA node locales make use of Qthreads shepherds
  - Tilera locales use OS hooks
- Initial performance improvements demonstrated
  - Yet further tuning work is required

Next Steps

- Unify the two efforts
- refine the interfaces
- promote this code to trunk
- Target increasingly heterogeneous nodes

Longer-term Directions

Represent physical machine as a hierarchical locale and represent user's locales as a *slice* of that hierarchy

- for topology-aware programming
- for jobs with dynamically-changing resource requirements
  - due to changing job needs
  - or failing HW

Combine with containment domains (Erez, UT Austin)

- the two concepts seem well-matched for each other

Summary

Next-generation nodes will likely present challenges

Chapel is better placed than current HPC languages

- Hierarchical locales should help with intra-node concerns

Hierarchical Locales have some attractive properties

- Defined in Chapel, potentially by users
- Support policy decisions
- Relaxes hard-coding of interfaces in compiler

Specification and implementation effort is underway

- Yet more work remains

The Chapel Team (Summer 2012)

For More Information

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, …

Blog Series:

Myths About Scalable Programming Languages:

https://www.ieeetcsc.org/activities/blog/

Upcoming Events:

PGAS talk: October 12th: "State of the Chapel Union"
SC12 tutorial, BoFs: November 12th-16th

Mailing Lists:

- chapel_info@cray.com: contact the team
- chapel-users@lists.sourceforge.net: user-oriented discussion list

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