Hierarchical Locales: Exposing Node-Level Locality in Chapel

Brad Chamberlain, Chapel Team, Cray Inc.

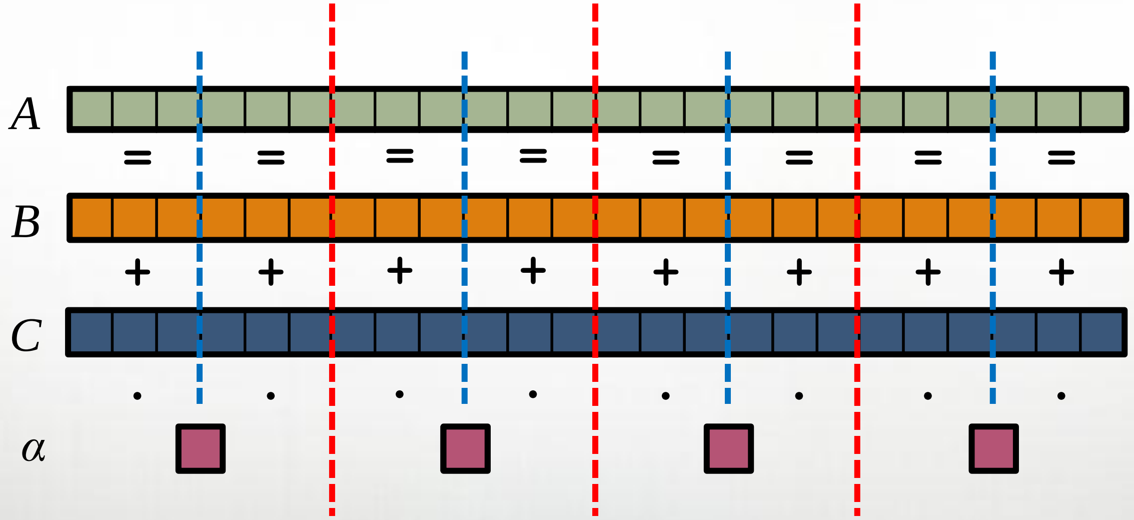
UIUC, May 16, 2013

STREAM Triad: a trivial parallel computation

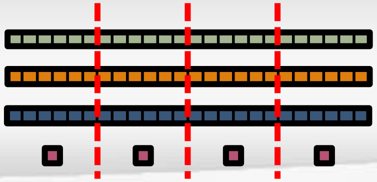
Given: m-element vectors A, B, C

Compute: $\forall i \in 1..m$, $A_i = B_i + \alpha \cdot C_i$

In pictures, in parallel (distributed memory multicore):



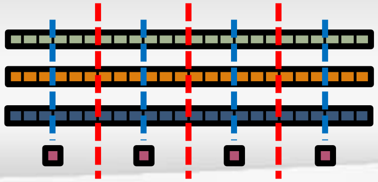
STREAM Triad: MPI



MPI

```
#include <hpcc.h>




static int VectorSize;
static double *a, *b, *c;

int HPCC_StarStream(HPCC_Params *params) {
  int myRank, commSize;
  int rv, errCount;
  MPI_Comm comm = MPI_COMM_WORLD;

  MPI_Comm_size( comm, &commSize );
  MPI_Comm_rank( comm, &myRank );

  rv = HPCC_Stream( params, 0 == myRank);
  MPI_Reduce( &rv, &errCount, 1, MPI_INT, MPI_SUM,
    0, comm );

  return errCount;
}

int HPCC_Stream(HPCC_Params *params, int doIO) {
  register int j;
  double  scalar;

  VectorSize = HPCC_LocalVectorSize( params, 3,
    sizeof(double), 0 );

  a = HPCC_XMALLOC( double, VectorSize );
  b = HPCC_XMALLOC( double, VectorSize );
  c = HPCC_XMALLOC( double, VectorSize );

  if (!a || !b || !c) {
    if (c) HPCC_free(c);
    if (b) HPCC_free(b);
    if (a) HPCC_free(a);
    if (doIO) {
      fprintf( outFile, "Failed to allocate memory
    (%d).\n", VectorSize );
      fclose( outFile );
    }
    return 1;
  }




  for (j=0; j<VectorSize; j++) {
    b[j] = 2.0;
    c[j] = 0.0;
  }

  scalar = 3.0;




  for (j=0; j<VectorSize; j++)
    a[j] = b[j]+scalar*c[j];

  HPCC_free(c);
  HPCC_free(b);
  HPCC_free(a);

  return 0;
}
```

STREAM Triad: MPI+OpenMP



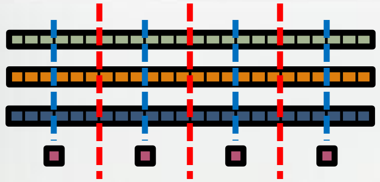
MPI + OpenMP

```
#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif

static int VectorSize;
static double *a, *b, *c;

int HPCC_StarStream(HPCC_Params *params) {
  int myRank, commSize;
  int rv, errCount;
  MPI_Comm comm = MPI_COMM_WORLD;

  MPI_Comm_size( comm, &commSize );
  MPI_Comm_rank( comm, &myRank );

  rv = HPCC_Stream( params, 0 == myRank);
  MPI_Reduce( &rv, &errCount, 1, MPI_INT, MPI_SUM,
    0, comm );

  return errCount;
}

int HPCC_Stream(HPCC_Params *params, int doIO) {
  register int j;
  double  scalar;

  VectorSize = HPCC_LocalVectorSize( params, 3,
    sizeof(double), 0 );

  a = HPCC_XMALLOC( double, VectorSize );
  b = HPCC_XMALLOC( double, VectorSize );
  c = HPCC_XMALLOC( double, VectorSize );
```

```
  if (!a || !b || !c) {
    if (c) HPCC_free(c);
    if (b) HPCC_free(b);
    if (a) HPCC_free(a);
    if (doIO) {
      fprintf( outFile, "Failed to allocate memory
    (%d).\n", VectorSize );
      fclose( outFile );
    }
    return 1;
  }

#ifdef _OPENMP
#pragma omp parallel for
#endif
  for (j=0; j<VectorSize; j++) {
    b[j] = 2.0;
    c[j] = 0.0;
  }

  scalar = 3.0;

#ifdef _OPENMP
#pragma omp parallel for
#endif
  for (j=0; j<VectorSize; j++)
    a[j] = b[j]+scalar*c[j];

  HPCC_free(c);
  HPCC_free(b);
  HPCC_free(a);

  return 0;
}
```

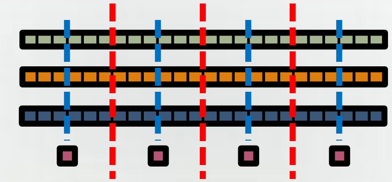
STREAM Triad: MPI+OpenMP vs. CUDA

MPI + OpenMP



```
#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif

static int VectorSize;
static double *a, *b, *c;

int HPCC_StarStream(HPCC_Params *params) {
  int myRank, commSize;
  int rv, errCount;
  MPI_Comm comm = MPI_COMM_WORLD;

  MPI_Comm_size( comm, &commSize );
  MPI_Comm_rank( comm, &myRank );

  rv = HPCC_Stream( params, 0 == myRank);
  MPI_Reduce( &rv, &errCount, 1, MPI_INT, MPI_SUM, 0, comm );

  return errCount;
}

int HPCC_Stream(HPCC_Params *params, int doIO) {
  register int j;
  double  scalar;

  VectorSize = HPCC_LocalVectorSize( params, 3, sizeof(double), 0 );

  a = HPCC_XMALLOC( double, VectorSize );
  b = HPCC_XMALLOC( double, VectorSize );
  c = HPCC_XMALLOC( double, VectorSize );

  if (!a || !b || !c) {
    if (c) HPCC_free(c);
    if (b) HPCC_free(b);
    if (a) HPCC_free(a);
    if (doIO) {
      fprintf( outFile, "Failed to allocate memory (%d).\n", VectorSize );
      fclose( outFile );
    }
    return 1;
  }

#ifdef _OPENMP
#pragma omp parallel for
#endif
  for (j=0; j<VectorSize; j++) {
    b[j] = 2.0;
    c[j] = 0.0;
  }

  scalar = 3.0;

#ifdef _OPENMP
#pragma omp parallel for
#endif
  for (j=0; j<VectorSize; j++)
    a[j] = b[j]+scalar*c[j];

  HPCC_free(c);
  HPCC_free(b);
  HPCC_free(a);

  return 0;
}
```

CUDA



```
#define N       2000000

int main() {
  float *d_a, *d_b, *d_c;
  float scalar;

  cudaMalloc((void**)&d_a, sizeof(float)*N);
  cudaMalloc((void**)&d_b, sizeof(float)*N);
  cudaMalloc((void**)&d_c, sizeof(float)*N);

  dim3 dimBlock(128);
  dim3 dimGrid(N/dimBlock.x );
  if( N % dimBlock.x != 0 ) dimGrid.x+=1;

  set_array<<<dimGrid,dimBlock>>>(d_b, .5f, N);
  set_array<<<dimGrid,dimBlock>>>(d_c, .5f, N);

  scalar=3.0f;
  STREAM_Triad<<<dimGrid,dimBlock>>>(d_b, d_c, d_a, scalar,  N);
  cudaThreadSynchronize();

  cudaFree(d_a);
  cudaFree(d_b);
  cudaFree(d_c);
}

__global__ void set_array(float *a,  float value, int len) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if (idx < len) a[idx] = value;
}

__global__ void STREAM_Triad( float *a, float *b, float *c,
                              float scalar, int len) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if (idx < len) c[idx] = a[idx]+scalar*b[idx];
}
```

HPC suffers from too many distinct notations for expressing parallelism and locality

Why so many programming models?

HPC has traditionally given users…

…low-level, *control-centric* programming models

…ones that are closely tied to the underlying hardware

…ones that support only a single type of parallelism

Examples:

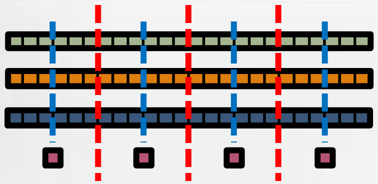
Type of HW Parallelism	Programming Model	Unit of Parallelism
Inter-node	MPI	executable
Intra-node/multicore	OpenMP/pthreads	iteration/task
Instruction-level vectors/threads	pragmas	iteration
GPU/accelerator	CUDA/OpenCL/OpenAcc	SIMD function/task

benefits: lots of control; decent generality; easy to implement

downsides: lots of user-managed detail; brittle to changes

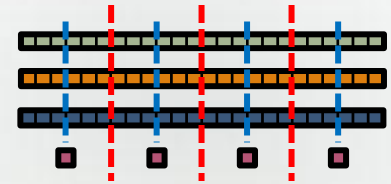
STREAM Triad: MPI+OpenMP vs. CUDA

MPI + OpenMP



```c
#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif

static int VectorSize;
static double *a, *b, *c;

int HPCC_StarStream(HPCC_Params *params) {
  int myRank, commSize;
  int rv, errCount;
  MPI_Comm comm = MPI_COMM_WORLD;

  MPI_Comm_size( comm, &commSize );
  MPI_Comm_rank( comm, &myRank );

  rv = HPCC_Stream( params, 0 == myRank);
  MPI_Reduce( &rv, &errCount, 1, MPI_INT, MPI_SUM, 0, comm );

  return errCount;
}

int HPCC_Stream(HPCC_Params *params, int doIO) {
  register int j;
  double  scalar;

  VectorSize = HPCC_LocalVectorSize( params, 3, sizeof(double), 0 );

  a = HPCC_XMALLOC( double, VectorSize );
  b = HPCC_XMALLOC( double, VectorSize );
  c = HPCC_XMALLOC( double, VectorSize );

  if (!a || !b || !c) {
    if (c) HPCC_free(c);
    if (b) HPCC_free(b);
    if (a) HPCC_free(a);
    if (doIO) {
      fprintf( outFile, "Failed to allocate memory (%d).\n", VectorSize );
      fclose( outFile );
    }
    return 1;
  }

#ifdef _OPENMP
#pragma omp parallel for
#endif
  for (j=0; j<VectorSize; j++) {
    b[j] = 2.0;
    c[j] = 0.0;
  }

  scalar = 3.0;

#ifdef _OPENMP
#pragma omp parallel for
#endif
  for (j=0; j<VectorSize; j++)
    a[j] = b[j]+scalar*c[j];

  HPCC_free(c);
  HPCC_free(b);
  HPCC_free(a);

  return 0;
}
```

CUDA



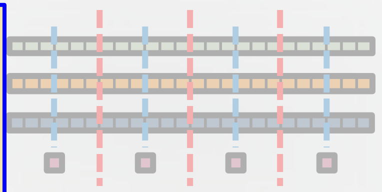
```c
#define N       2000000

int main() {
  float *d_a, *d_b, *d_c;
  float scalar;

  cudaMalloc((void**)&d_a, sizeof(float)*N);
  cudaMalloc((void**)&d_b, sizeof(float)*N);
  cudaMalloc((void**)&d_c, sizeof(float)*N);

  dim3 dimBlock(128);
  dim3 dimGrid(N/dimBlock.x );
  if( N % dimBlock.x != 0 ) dimGrid.x+=1;

  set_array<<<dimGrid,dimBlock>>>(d_b, .5f, N);
  set_array<<<dimGrid,dimBlock>>>(d_c, .5f, N);

  scalar=3.0f;
  STREAM_Triad<<<dimGrid,dimBlock>>>(d_b, d_c, d_a, scalar,  N);
  cudaThreadSynchronize();

  cudaFree(d_a);
  cudaFree(d_b);
  cudaFree(d_c);
}

__global__ void set_array(float *a,  float value, int len) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if (idx < len) a[idx] = value;
}

__global__ void STREAM_Triad( float *a, float *b, float *c,
                              float scalar, int len) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if (idx < len) c[idx] = a[idx]+scalar*b[idx];
}
```

HPC suffers from too many distinct notations for expressing parallelism and locality

STREAM Triad: Chapel

MPI + OpenMP

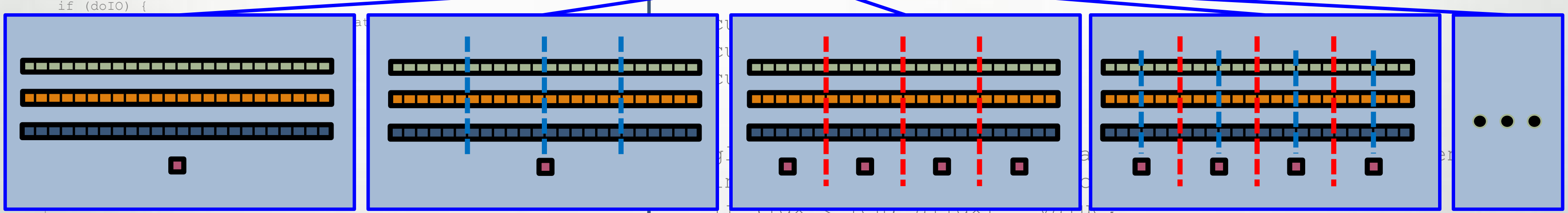
```
#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif

static int VectorSize;
static double *a, *b, *c;

int HPCC_StarStream(HPCC_Params *par
  int myRank, commSize;
  int rv, errCount;
  MPI_Comm comm = MPI_COMM_WORLD;

  MPI_Comm_size( comm, &commSize );
  MPI_Comm_rank( comm, &myRank );

  rv = HPCC_Stream( params, 0 == myR
  MPI_Reduce( &rv, &errCount, 1, MPI

  return errCount;
}

int HPCC_Stream(HPCC_Params *params,
  register int j;
  double  scalar;

  VectorSize = HPCC_LocalVectorSize(

  a = HPCC_XMALLOC( double, VectorSi
  b = HPCC_XMALLOC( double, VectorSi
  c = HPCC_XMALLOC( double, VectorSi

  if (!a || !b || !c) {
    if (c) HPCC_free(c);
    if (b) HPCC_free(b);
    if (a) HPCC_free(a);
    if (doIO) {
```



Chapel

```
config const m = 1000,
             alpha = 3.0;

const ProblemSpace = {1..m} dmapped …;

var A, B, C: [ProblemSpace] real;

B = 2.0;
C = 3.0;

A = B + alpha * C;
```

the special sauce



<u>Philosophy:</u> Good language design can tease details of locality and parallelism away from an algorithm, permitting the compiler, runtime, applied scientist, and HPC expert to each focus on their strengths.

A Chapel Goal: General Parallel Programming

With a unified set of concepts...

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

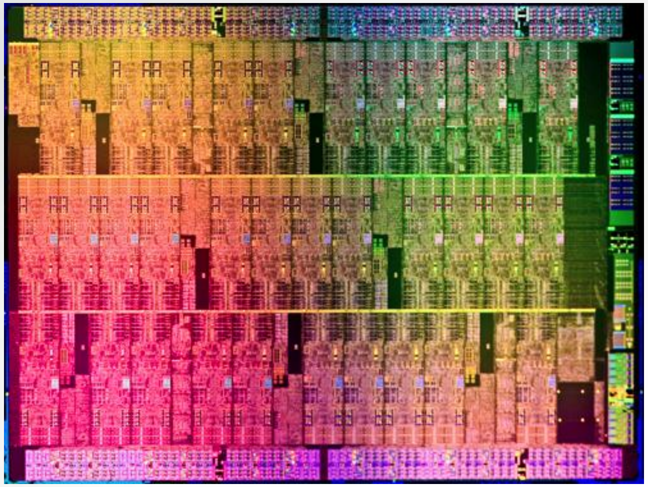
- **Styles:** data-parallel, task-parallel, concurrency, nested, …
- **Levels:** model, function, loop, statement, expression

...target all parallelism available in the hardware

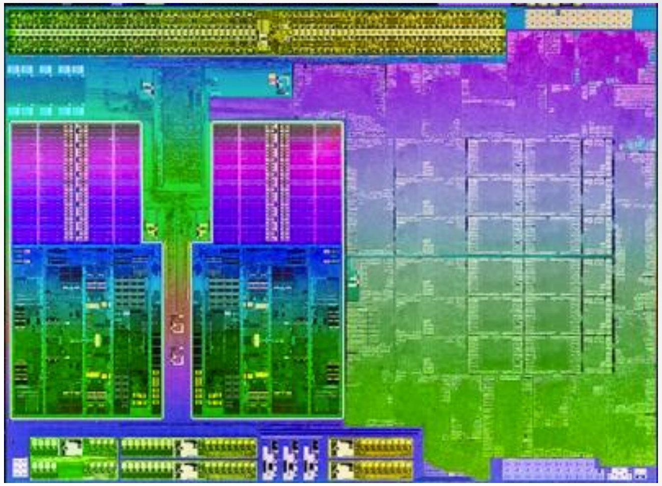
- **Types:** machines, nodes, cores, instructions

Style 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

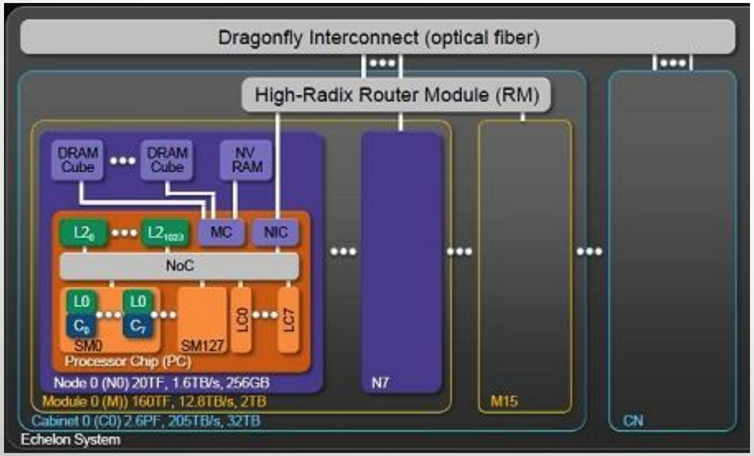
Prototypical Next-Gen Processor Technologies



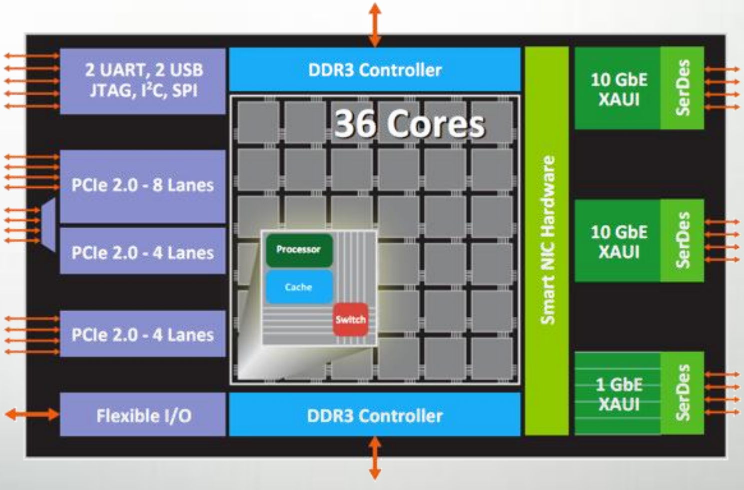
Intel MIC



AMD Trinity



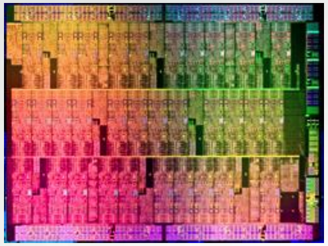
Nvidia Echelon

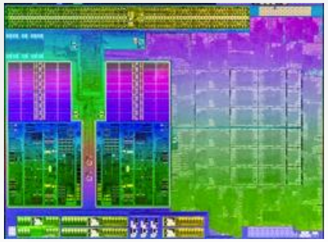


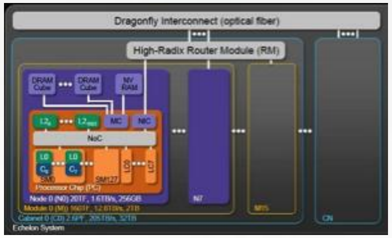
Tilera Tile-Gx

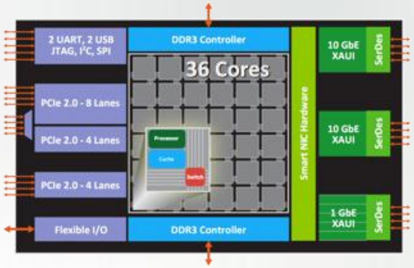
Sources: http://download.intel.com/pressroom/images/Aubrey_Isle_die.jpg, http://www.zdnet.com/amds-trinity-processors-take-on-intels-ivy-bridge-3040155225/, http://insidehpc.com/2010/11/26/nvidia-reveals-details-of-echelon-gpu-designs-for-exascale/, http://tilera.com/sites/default/files/productbriefs/Tile-Gx%203036%20SB012-01.pdf

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 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 is designed to address these items)

Outline

- ✓ Motivation
- ➢ Chapel Background
- Hierarchical Locales in Chapel
- Challenges, 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, …

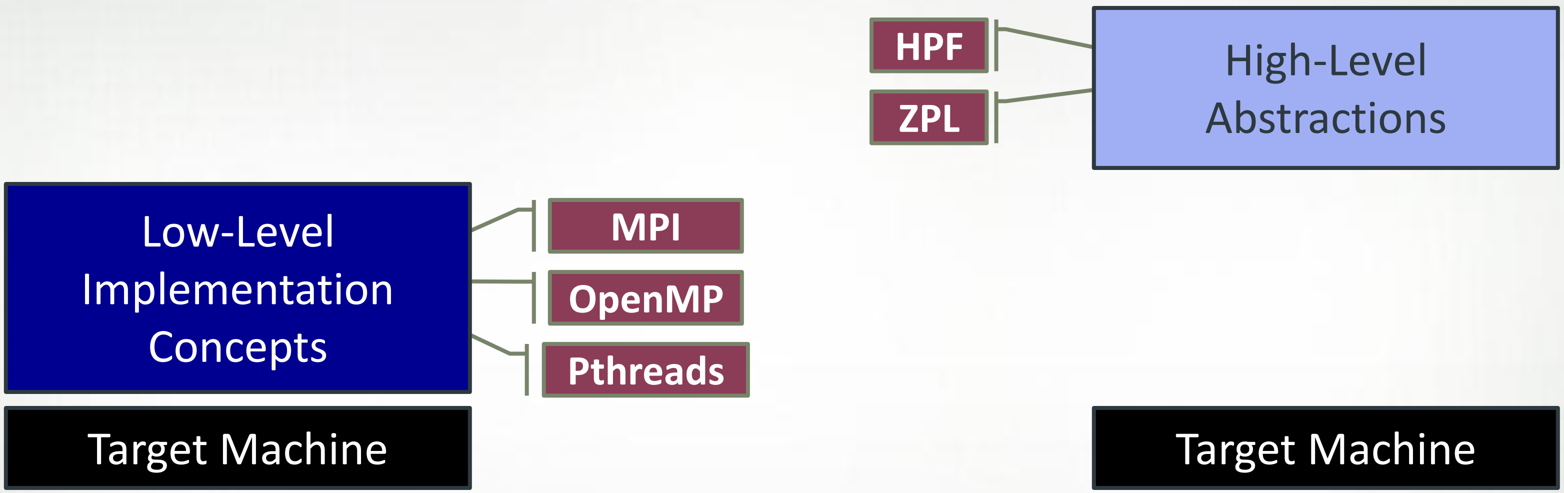
Chapel's Greatest Hits under HPCS

- 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 under HPCS

- ➢ 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

Domain Maps
Data Parallelism
Task Parallelism
Base Language
Locality Control

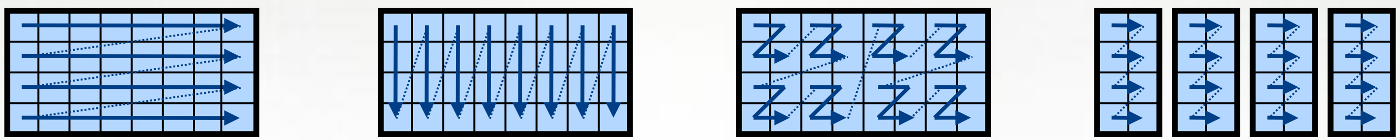
Target Machine

- 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

Data Parallelism Implementation Qs

Q1: How are arrays laid out in memory?

- Are regular arrays laid out in row- or column-major order? Or…?

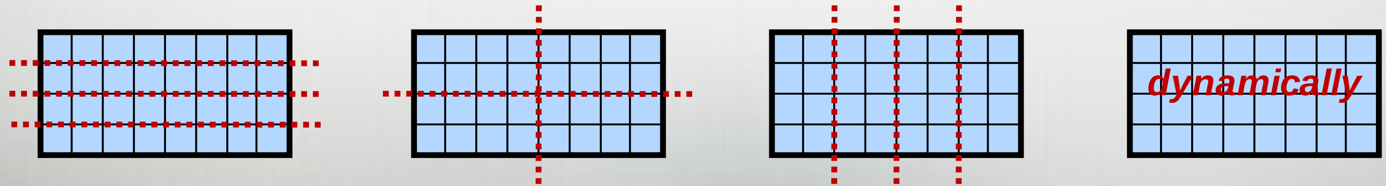


…?

- How are sparse arrays stored? (COO, CSR, CSC, block-structured, …?)

Q2: How are arrays stored by the locales?

- Completely local to one locale? Or distributed?
- If distributed… In a blocked manner? cyclically? block-cyclically? recursively bisected? dynamically rebalanced? …?

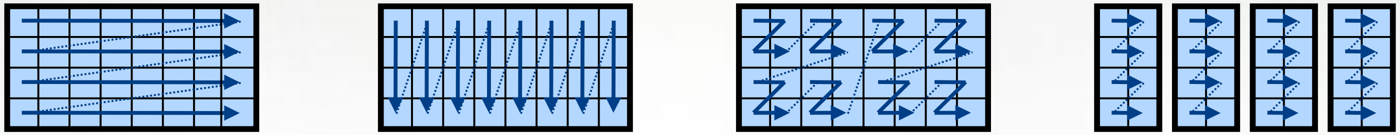


…?

Data Parallelism Implementation Qs

Q1: How are arrays laid out in memory?

- Are regular arrays laid out in row- or column-major order? Or…?



…?

- How are sparse arrays stored? (COO, CSR, CSC, block-structured, …?)

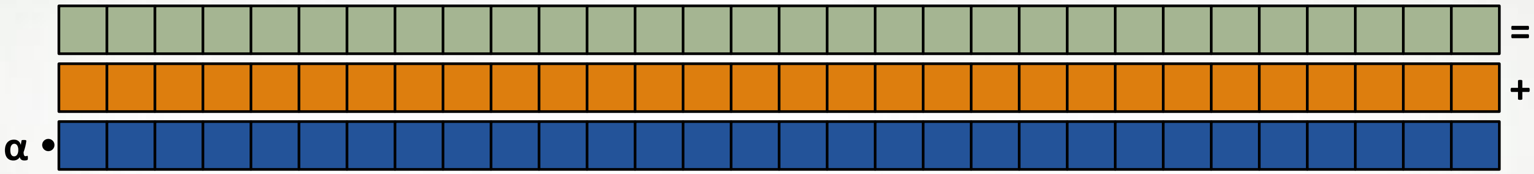
Q2: How are arrays stored by the locales?

- Completely local to one locale? Or distributed?
- If distributed… In a blocked manner? cyclically? block-cyclically? recursively bisected? dynamically rebalanced? …?

A: Chapel's *domain maps* are designed to give the user full control over such decisions

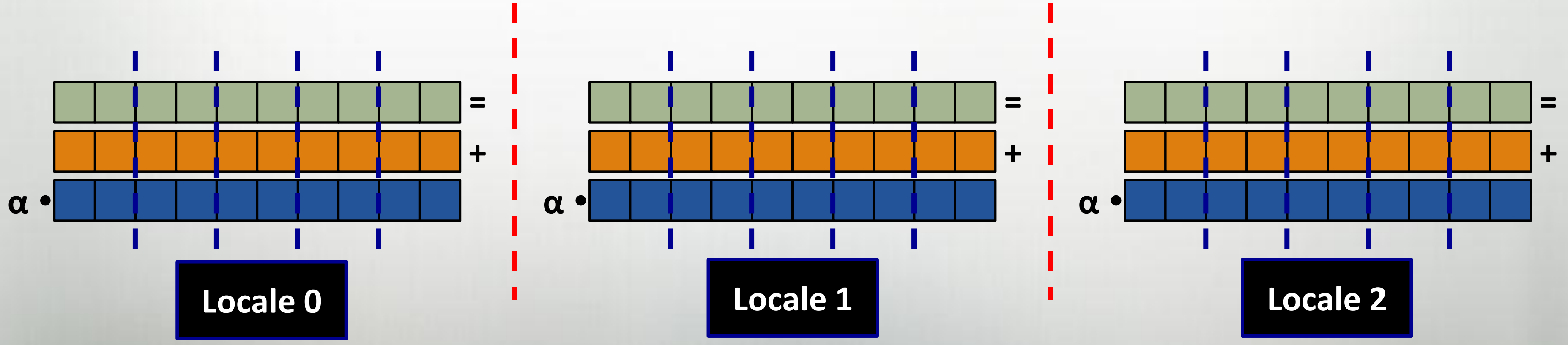
Domain Maps

Domain maps are "recipes" that instruct the compiler how to map the global view of a computation…



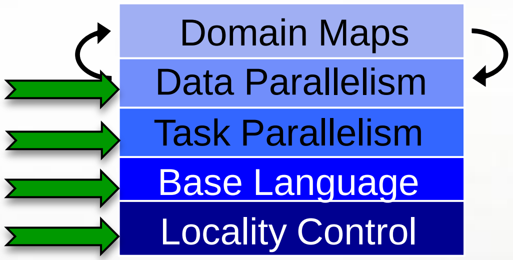

```
A = B + alpha * C;
```

…to the target locales' memory and processors:



Chapel's Domain Map Philosophy

1. Chapel provides a library of standard domain maps
   - to support common array implementations effortlessly
2. Advanced users can write their own domain maps in Chapel
   - to cope with shortcomings in our standard library

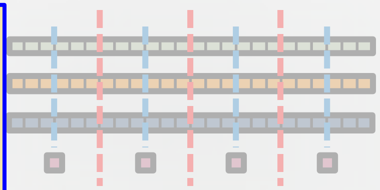


3. Chapel's standard domain maps are written using the same end-user framework
   - to avoid a performance cliff between "built-in" and user-defined cases

STREAM Triad: Chapel

MPI + OpenMP

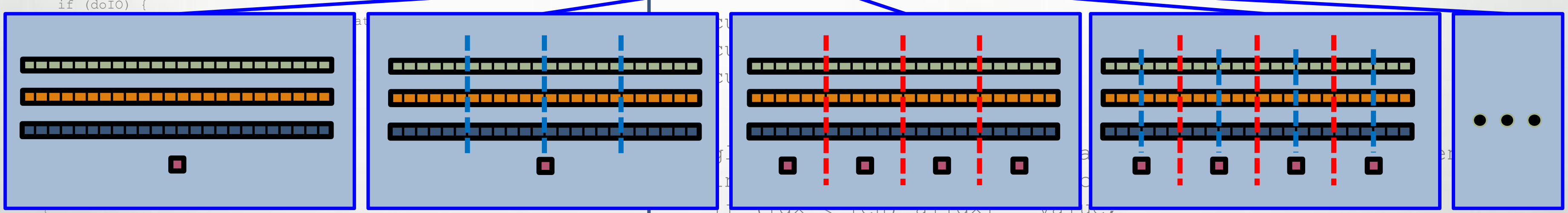
```
#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif

static int VectorSize;
static double *a, *b, *c;

int HPCC_StarStream(HPCC_Params *par
  int myRank, commSize;
  int rv, errCount;
  MPI_Comm comm = MPI_COMM_WORLD;

  MPI_Comm_size( comm, &commSize );
  MPI_Comm_rank( comm, &myRank );

  rv = HPCC_Stream( params, 0 == myR
  MPI_Reduce( &rv, &errCount, 1, MPI

  return errCount;
}

int HPCC_Stream(HPCC_Params *params,
  register int j;
  double  scalar;

  VectorSize = HPCC_LocalVectorSize(

  a = HPCC_XMALLOC( double, VectorSi
  b = HPCC_XMALLOC( double, VectorSi
  c = HPCC_XMALLOC( double, VectorSi

  if (!a || !b || !c) {
    if (c) HPCC_free(c);
    if (b) HPCC_free(b);
    if (a) HPCC_free(a);
    if (doIO) {
```



Chapel

```
config const m = 1000,
             alpha = 3.0;

const ProblemSpace = {1..m} dmapped …;

var A, B, C: [ProblemSpace] real;

B = 2.0;
C = 3.0;

A = B + alpha * C;
```

the special sauce



Philosophy: Good language design can tease details of locality and parallelism away from an algorithm, permitting the compiler, runtime, applied scientist, and HPC expert to each focus on their strengths.

For More Information on Domain Maps

HotPAR'10: *User-Defined Distributions and Layouts in Chapel: Philosophy and Framework*
Chamberlain, Deitz, Iten, Choi; June 2010

CUG 2011: *Authoring User-Defined Domain Maps in Chapel*
Chamberlain, Choi, Deitz, Iten, Litvinov; May 2011

Chapel release:

- Technical notes detailing domain map interface for programmers:
  $CHPL_HOME/doc/technotes/README.dsi
- Current domain maps:
  $CHPL_HOME/modules/dists/*.chpl
  layouts/*.chpl
  internal/Default*.chpl

More Data Parallelism Implementation Qs

Q1: How are parallel loops implemented?

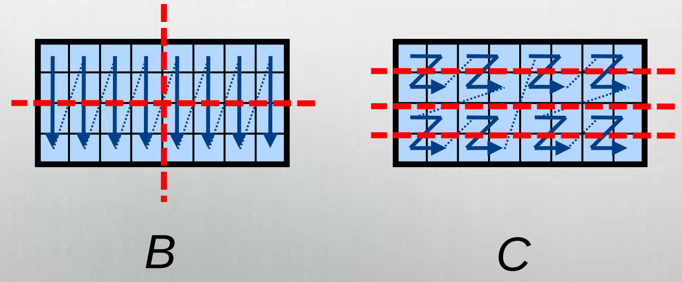
```
forall i in B.domain do B[i] = i/10.0;
forall c in C do c = 3.0;
```

- How many tasks? Where do they execute?
- How is the iteration space divided between the tasks?

Q2: How are parallel zippered loops implemented?

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

- Particularly given that the iterands might have incompatible distributions, memory layouts, and parallelization strategies



More Data Parallelism Implementation Qs

Q1: How are parallel loops implemented?

```
forall i in B.domain do B[i] = i/10.0;
forall c in C do c = 3.0;
```

- How many tasks? Where do they execute?
- How is the iteration space divided between the tasks?

Q2: How are parallel zippered loops implemented?

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

- Particularly given that the iterands might have incompatible distributions, memory layouts, and parallelization strategies

A: Chapel's *leader-follower* iterators are designed to give users full control over such decisions

For More Information on Leader-Follower Iterators

PGAS 2011: *User-Defined Parallel Zippered Iterators in Chapel,*
Chamberlain, Choi, Deitz, Navarro; October 2011

Chapel release:

- Primer example introducing leader-follower iterators:
  - examples/primers/leaderfollower.chpl
- Library of dynamic leader-follower range iterators:
  - *AdvancedIters* chapter of language specification

Multiresolution Programming: Summary

- Chapel avoids locking crucial implementation decisions into the language specification
  - local and distributed array implementations
  - parallel loop implementations
- Instead, these can be…

  …specified in the language by an advanced user

  …swapped in and out with minimal code changes
- The result separates the roles of domain scientist, parallel programmer, and implementation cleanly

Chapel's Greatest Hits under HPCS

- 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

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

Task Parallelism Example: Cobegin Statements

```
cobegin {           // creates a task per child statement
  producer(1);
  producer(2);
  consumer(1);
}                   // logical join of the three tasks here
```

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:

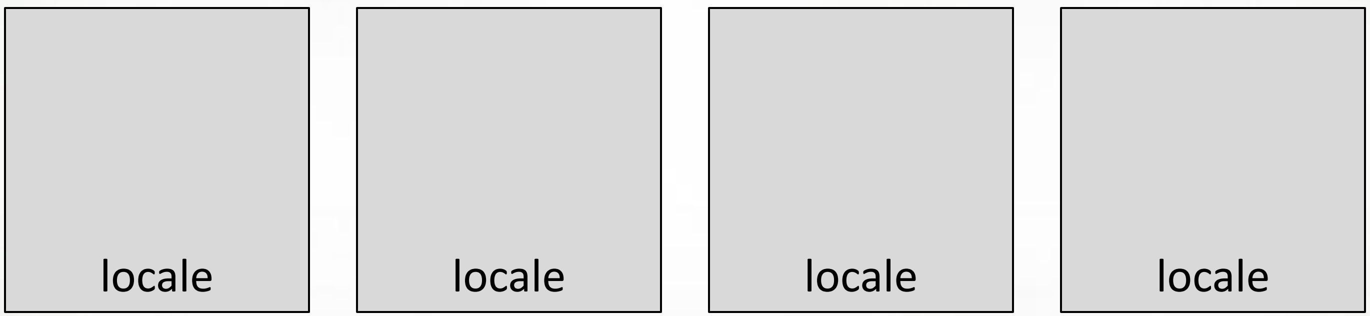
```
writeln("on locale 0");

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:

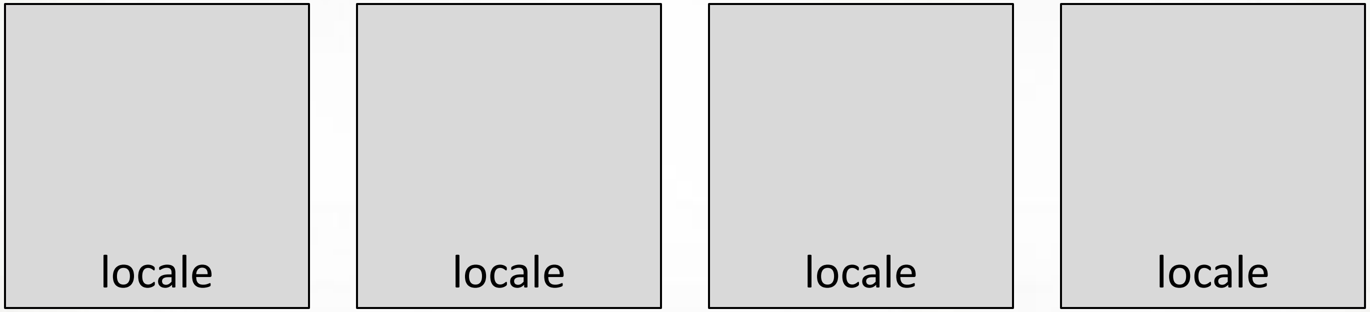
- Today, Chapel supports a 1D array of locales
  - users can reshape/slice to suit their computation's needs



Locales Today

Concept:

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



- Apart from 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)

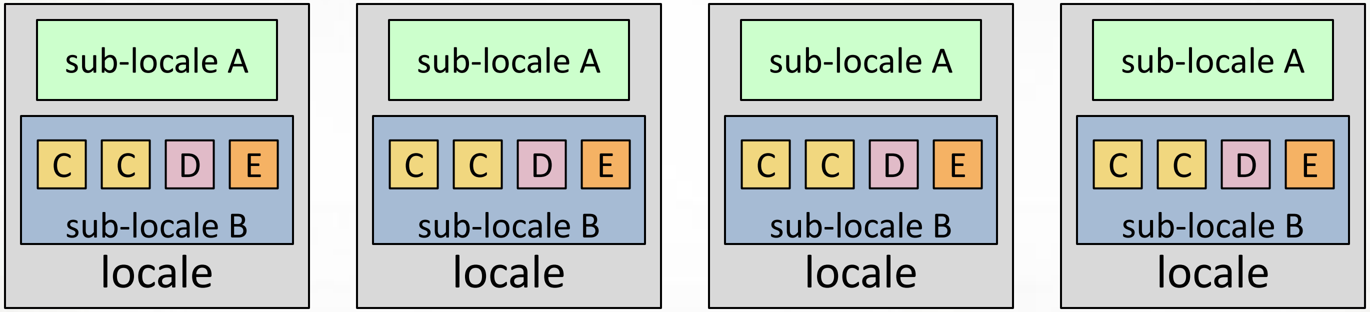
Outline

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

Current Work: Hierarchical Locales

Concept:

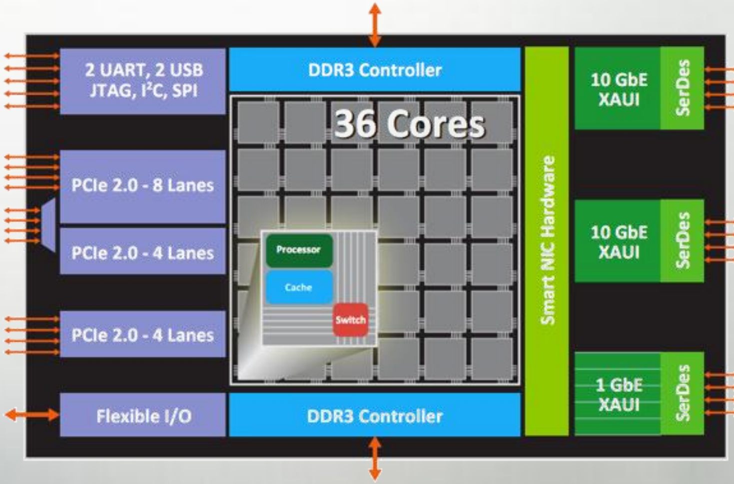
- Support locales within locales to describe architectural sub-structures within a node



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

Sublocales: Tiled Processor Example

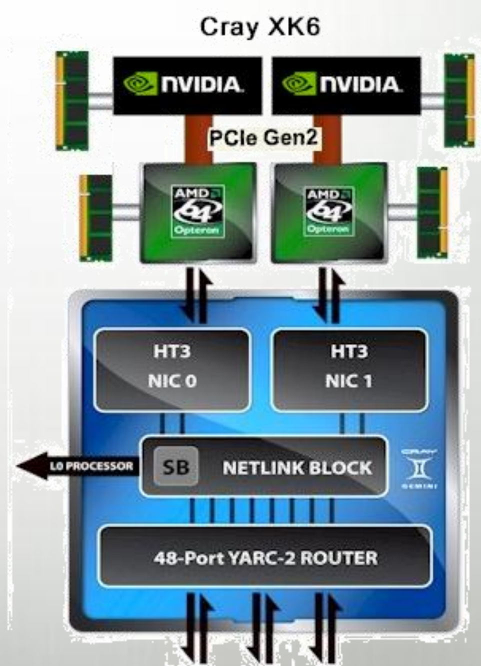
```
class locale: AbstractLocale {
  const xt = 6, yt = xTiles;
  const sublocGrid: [0..#xt, 0..#yt] tiledLoc = …;
  …memory interface…
  …tasking interface…
}

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(…) { … }
proc AbstractLocale.tasksCoforall(…) { … }
…
```

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:

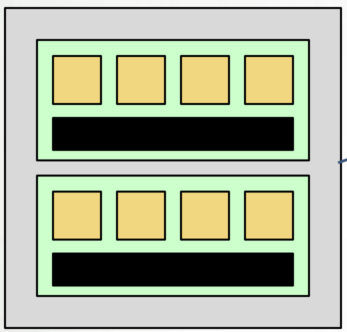
- Can a task that's placed on a specific sublocale migrate?
  - to where? sibling? parent? somewhere else?
- What happens on locales with no processors?
  - illegal? allocate on sublocale? parent locale?
  - using what heuristic? sublocale[0]? round-robin? dynamic load balance?

Goal: Any of these policies should be possible

Tasking Policy Example

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

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



on "multicore NUMA Node" do begin foo()

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(…);
}
```

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;
  …
}
```

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;
}
```

Another Tasking Policy Example

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

e.g., a sublocale representing a memory resource

Another 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);
```

Another 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(…);
}
```

Another 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(…);
}
```

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
  - e.g., 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
- ➢ Challenges, Status, and Summary

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 a different model

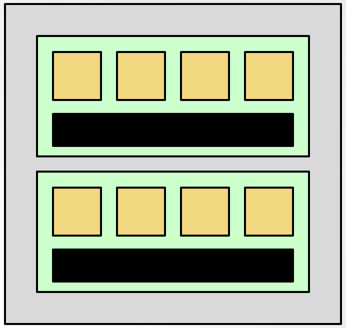
Mitigation Strategies

- 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



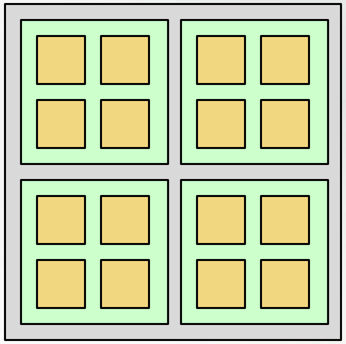
Approach:

- 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

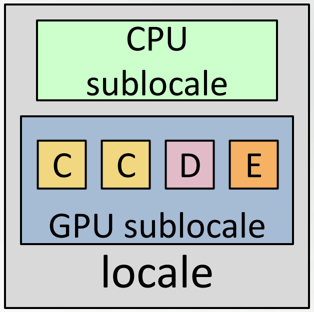
Why? More interesting example w/ user interest

- reconfigurability, 2D layout particularly interesting

Initial Goal: Run Chapel codes using various Tilera configurations

- ideally, with single Chapel locale definition file

Target 3: Clusters of CPU-GPU Compute Nodes



Platform: Cluster of CPU+GPU Nodes

Approach:

- 3-to-4 level locale structure
  - outer: Network
  - next: Compute Node
  - next: CPU vs. GPU
  - inner (potentially): distinct processor cores/memories (?)

Why? Look at #1 on the top-500

- provide a unified alternative to MPI+X

Initial Goal:

- Run some traditional CPU+GPU codes on one node
- Port some CPU+GPU cluster codes to Chapel

Status

- Proof-of-Concept draft up and running
- Working on merging concept into trunk
- Next Steps:
  - Get code into trunk
  - Ensure performance for traditional architectures isn't unduly effected
  - Port and study sample application codes

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

Technical 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)

Implementation Status -- Version 1.7.0 (Apr 2013)

In a nutshell:

- Most features work at a functional level
- Many performance optimizations remain
  - particularly for distributed memory (multi-locale) execution

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 and Education

- In teaching parallel programming, I like to cover:
  - data parallelism
  - task parallelism
  - concurrency
  - synchronization
  - locality/affinity
  - deadlock, livelock, and other pitfalls
  - performance tuning
  - …
- I don't think there's been a good language out there…
  - for teaching *all* of these things
  - for teaching some of these things well at all
  - ***until now:*** We believe Chapel can potentially play a crucial role here

(see http://chapel.cray.com/education.html for more information and http://cs.washington.edu/education/courses/csep524/13wi/ for my use of Chapel in class)

Chapel: What's Next?

- Ramp up staffing
- Fill gaps in the language design
  - exception handling, task teams, interoperability, RAII, OOP, …
- Address heterogeneous compute nodes
  - hierarchical locales to support GPUs, Intel MIC
- User-driven performance improvements
  - Scalar idioms, communication optimizations, memory leaks
- Work on transitioning governance to external entity
  - e.g., "The Chapel Foundation"

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, …

Chapel Background:

A Brief Overview of Chapel (chapter pre-print)

The State of the Chapel Union (CUG 2013)

[Ten] Myths About Scalable Programming Languages:

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

Mailing Lists:

chapel_info@cray.com: contact the team

chapel-users, chapel-education, chapel-developers: SourceForge discussion lists

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