Hierarchical Locales: Exposing the Node Architecture in Chapel

Sung-Eun Choi, Cray Inc.

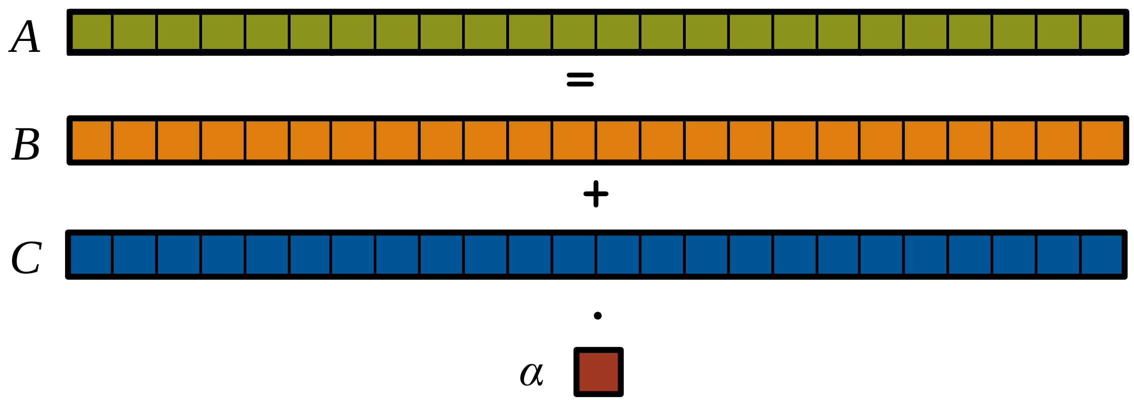
2nd KIISE-KOCSEA SIG HPC Workshop
November 19, 2013

STREAM Triad: a trivial parallel computation

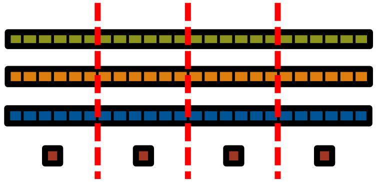
Given: m-element vectors A, B, C

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

In pictures:



STREAM Triad: MPI

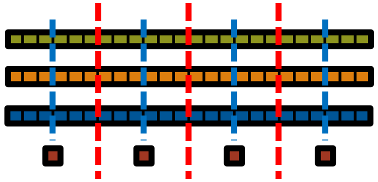


MPI

```
#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);
```

STREAM Triad: MPI+OpenMP



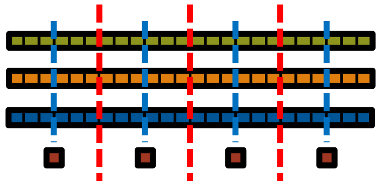
MPI + OpenMP

```
#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);
```

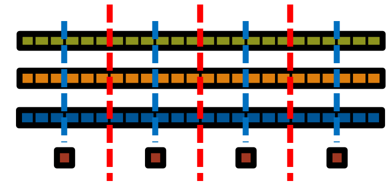
STREAM Triad: MPI+OpenMP vs. CUDA

MPI + OpenMP



```
#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

  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];
```

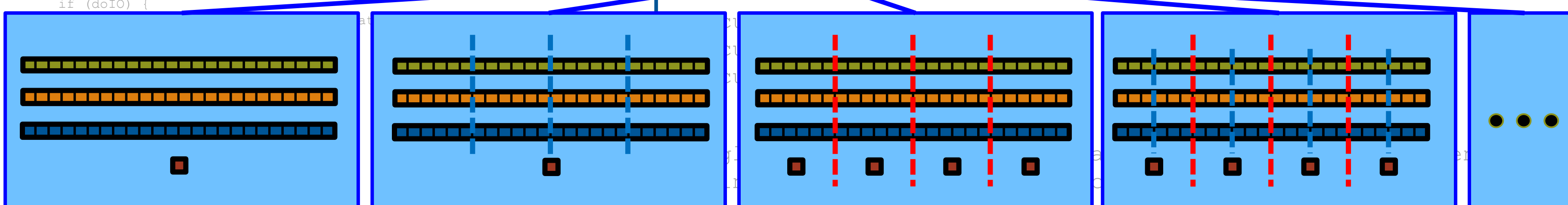
STREAM Triad: Chapel

MPI + OpenMP

```
#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) {
```

```
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.

Outline

- ✓ Motivation
- ➢ Chapel Background
- Exposing the node architecture: locale models
- Project Status and Next Steps

What is Chapel?

- **An emerging parallel programming language**
  - Design and development led by Cray Inc.
    - in collaboration with academia, labs, industry
  - 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:**
  - multicore desktops and laptops
  - Cray architectures
  - commodity clusters
  - systems from other vendors
  - *in-progress:* CPU+accelerator hybrids, …

Chapel Design Goal

Write code like this…

```
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;
```

Have it perform like this…

```
#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 );
    }
```

…on any architecture available

```
#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];

…
```

How can this be achieved?

Don’t expose low-level mechanisms

Chapel provides syntax to express locality and parallelism, which are portable to any architecture

Don’t expect the compiler to do everything

Chapel’s locality and parallelism mechanisms are implemented in Chapel modules not our compiler

Don’t keep the implementation too close

Chapel is an open-source project with contributors from around the world

Outline

- ✓ Motivation
- ✓ Chapel Background
- ➢ Exposing the node architecture: locale models
- Project Status and Next Steps

Exposing the node architecture: Locale models

- **Locale models**
  - The locale type
  - Locality/affinity control
  - Task parallelism
- **Example: The NUMA locale model**
- **STREAM Triad revisited**
  - Domain maps
  - Leader-follower iterators

The Locale Model

- **Every Chapel program employs a locale model to describe the system architecture**

- **Locale models define:**
  - Abstract node architecture (the *locale* type)
  - Memory allocation
  - Task scheduling policies
  - Locality/affinity control

- **Locale models are written in Chapel**
  - Class and procedural interface for the compiler
  - Utilizes runtime interfaces (e.g., communication, tasking/threading)
  - Programmer role: Architecture specialist

The locale type

- **The *locale* type**
  - an abstract unit of architecture
    - Can have memory, processing units, etc.
  - can be nested (hierarchical)
  - typically a compute node (multicore processor or SMP) or your laptop

- **Chapel programs run on one or more locales**

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

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

Locale Operations

- Locale methods support queries about the target system:

```
proc locale.numCores { … }
proc locale.id { … }
proc locale.name { … }
```

- *On* statements support placement of computations:

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

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

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

Locale Operations

- Locales can be nested:

```
proc locale.getChild(…) { … }
proc locale.getChildCount { … }
```

- Children (sublocales) are locales

- *On* statements support placement of computations on sublocales:

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

on here.getChild(1) do
  writeln("now on locale 0, sublocale 1");

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

On statements: Rewriting

***Conceptually*, the Chapel compiler translates:**

```
on here.getChild(1) do
  writeln("now on locale 0, sublocale 1");
```

into:

```
proc on_fn() {
  writeln("now on locale 0, sublocale 1");
}

chpl_executeOn(here.getChild(1), on_fn, …);
```

chpl_executeOn() is defined by the Locale Model

Task parallelism

- **Tasks are the basic unit of parallelism**
- **Tasks can be created by the following Chapel statements:**
  - begin
  - cobegin
  - coforall

Task parallelism: begin statements

```
// create a fire-and-forget task for a statement
begin writeln("hello world");
writeln("good bye");
```

Possible outputs:

```
hello world
good bye
```

```
good bye
hello world
```

Task parallelism: cobegin statements

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

Task parallelism: coforall loops

```
// create a task per iteration
coforall t in 0..#numTasks {
  writeln("Hello from task ", t, " of ", numTasks);
} // implicit join of the numTasks tasks here

writeln("All tasks done");
```

Sample output:

```
Hello from task 2 of 4
Hello from task 0 of 4
Hello from task 3 of 4
Hello from task 1 of 4
All tasks done
```

Begin statement: Rewriting

***Conceptually*, the Chapel compiler translates:**

```
begin writeln("hello world");
```

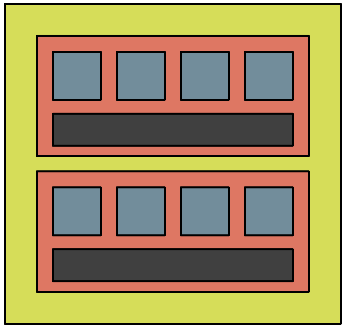
into:

```
proc begin_fn() {
  writeln("hello world");
}

chpl_taskListAddBegin(begin_fn, …);
```

chpl_taskListAddBegin() is defined by the Locale Model

Example: The NUMA locale model

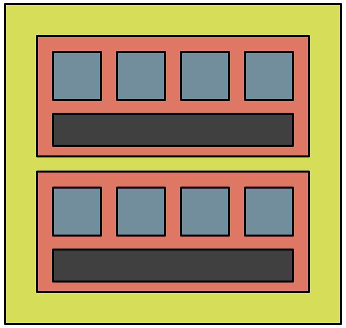
- **The first interesting Locale Model prototype**
- ***n* NUMA domains per locale (sublocale)**
  - *m* cores per NUMA domain
- **Memory divided between NUMA domains**
  - shared by both NUMA domains



Example: The NUMA locale model

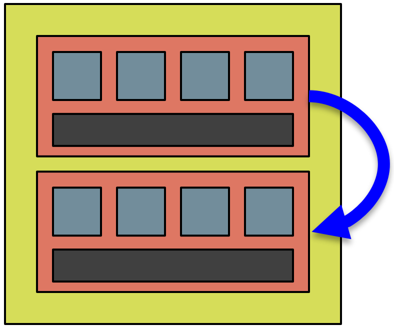
```
class LocaleModel: AbstractLocaleModel {
  var numSublocales: int;
  var childLocales: [0..#numSublocales] NumaDomain;
  …
  proc getChild(i) return childLocales[i];
  …
}
```

```
class NumaDomain: AbstractLocaleModel {
  …
  proc getChild(i) return nil;
  …
}
```



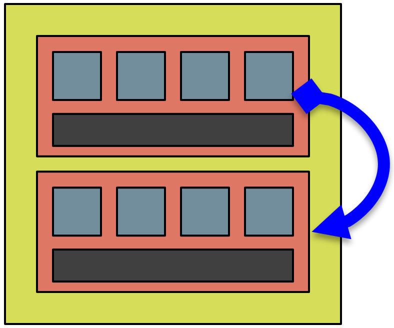
Example: chpl_executeOn()

```
// no off-node case, locale ids simplified
proc chpl_executeOn(loc, fn, args) {
  if (loc == here) {
    // run directly on this numa domain
    chpl_ftable_call(fn, args);
  } else {
    // move to a different numa domain
    var orig = here.sublocid;
    chpl_task_setSubloc(loc.sublocid);
    chpl_ftable_call(fn, args);
    chpl_task_setSubloc(orig);
   }
}
```



Example: chpl_taskListAddBegin()

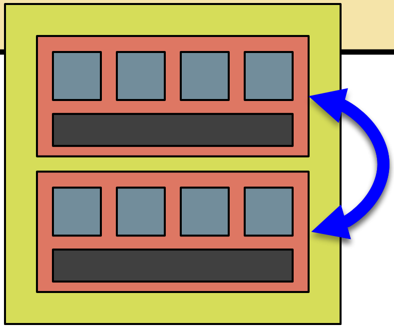
```
// Variant 1:
// Start the task on the specified sublocale
// Let the tasking layer decide what to do with "any"
proc chpl_taskListAddBegin(loc, fn, args) {
  chpl_task_addToTaskList(fn, args, loc.sublocid, …);
}
```



Example: chpl_taskListAddBegin()

```
// Variant 1:
```

```
// Variant 2:
// Start the task on the specified sublocale
// Schedule "any" in a round robin fashion
proc chpl_taskListAddBegin(loc, fn, args) {
  const sublocid
      = if loc.sublocid != subloc_any then
          loc.sublocid; // use specified subloc
        else
          loc.nextSubloc.fetchAdd(1) % loc.numSublocales;
                                 // round robin using atomic count
  chpl_task_addToTaskList(fn, args, sublocid, …);
}
```

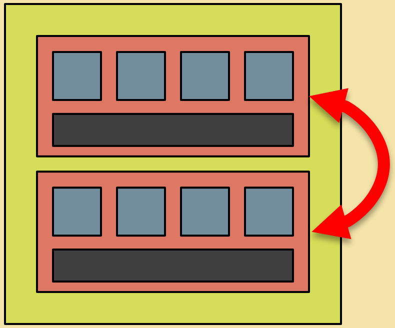


Example: chpl_taskListAddBegin()

// Variant 1:

// Variant 2:

```
// Variant 3:
// Start the task on the specified sublocale
// Schedule "any" based on load
proc chpl_taskListAddBegin(loc, fn, args) {
  const sublocid
    = if loc.sublocid != subloc_any then
        loc.sublocid; // use specified subloc
      else
        getBestSubloc(loc);
  chpl_task_addToTaskList(fn, args, sublocid, …);
}

proc getBestSubloc(loc) {
  const (, sublocid)
           = minloc reduce (loc.numTasks(), 0..#numSubLocs);
  return sublocid;
}
```



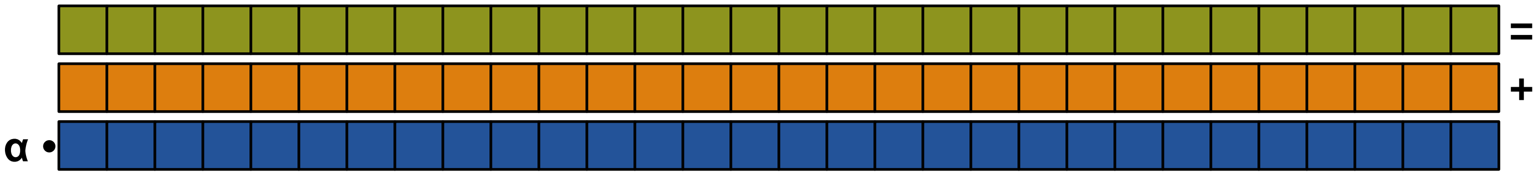
Back to STREAM Triad

```
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

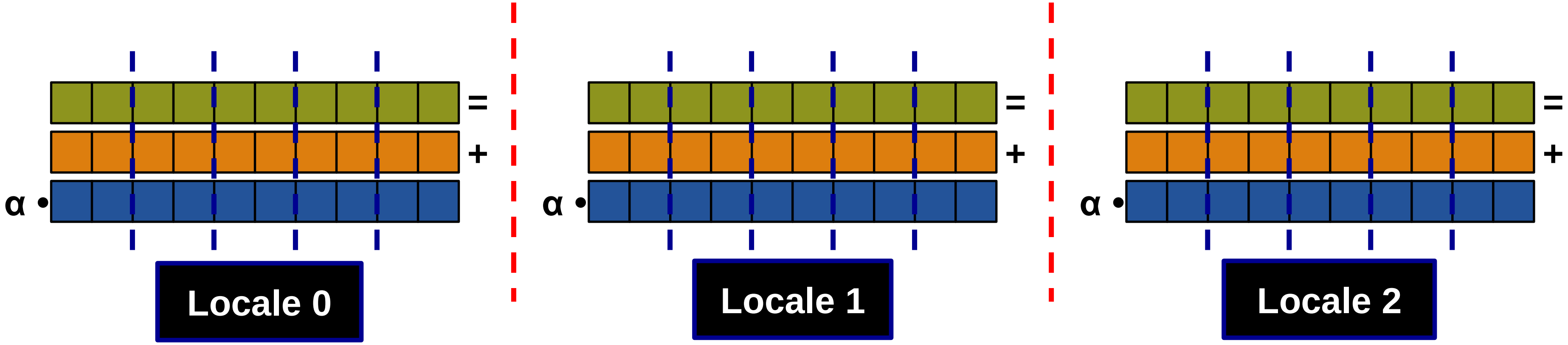
The Special Sauce: Domain maps

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:



Domain maps in a nutshell

- **Domain maps specify**
  - data allocation and layout
  - parallel iteration
  - low level implementation of array and domain operations
    - e.g., slicing, reallocating, reshaping

- **Domain maps are written in Chapel**
  - Class and procedural interface for the compiler
  - Utilizes lower level concepts like coforall and on statements
  - Programmer role: HPC specialist

Back to STREAM Triad

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

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

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

B = 2.0;
C = 3.0;

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

- **How many tasks?**
- **Where are they executed?**
- **How are the iterations assigned to tasks?**

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

Leader-Follower Iterators: Definition

- **Chapel defines all forall loops in terms of *leader-follower iterators*:**
  - *leader iterators:* create parallelism, assign iterations to tasks
  - *follower iterators:* serially execute work generated by leader

- **Given…**

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

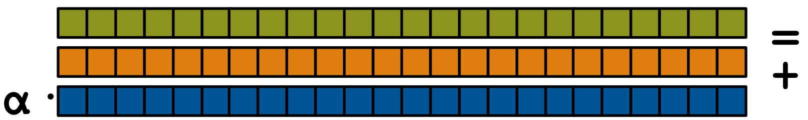
…*A* is defined to be the *leader*

…*A, B,* and *C* are all defined to be *followers*

Leader-follower iterators: Rewriting

Conceptually, the Chapel compiler translates:

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

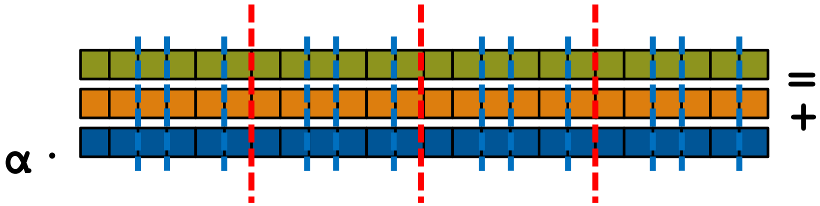


into:

Leader

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

Followers



Outline

- ✓ Motivation
- ✓ Chapel Background
- ✓ Exposing the node architecture: locale models
- ➢ Project Status and Next Steps

Implementation Status -- Version 1.8.0 (Oct 2013)

Overall Status:

- Most features work at a functional level
  - some features need to be improved or re-implemented (e.g., OOP)
- Many performance optimizations remain
  - particularly for distributed memory (multi-locale) execution

Next Steps

- **Evolve from Prototype- to Production-grade**
  - Add/Improve missing features
  - Performance optimizations
- **Target more complex compute node types**
  - e.g., CPU+GPU, Intel Phi, …
- **Continue to grow the user and developer communities**
  - Work toward transitioning Chapel from Cray-controlled to community-governed

Chapel Design Goal Revisited

Write code like this…

```
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;
```

Have it perform like this…

```
#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 );
    }
```

…on any architecture available

```
#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];

…
```

Chapel Design Goal Revisted

✓ High-level programmability

- Applications programmers expression parallelism and locality
- Chapel's locale models and domain maps can insulate programmers from details of the architecture

~ Performance

- Designed to enable good performance, results yet to be seen
- In cases where we've optimized, we've seen very promising results, e.g.,
  - Dynamic iterators that match the performance of OpenMP [Chamberlain, et al., PGAS 2011]
  - Communication aggregation implemented via the domain map interface [Sanz, et al., SBAC-PAD 2012]

Chapel at SC13

- **Emerging Technologies Booth** (all week)
  - Booth #3547: staffed by Chapel team members; poster and handouts
- **Talk** (Tues @ 3:20): *Hierarchical Locales: Exposing the Node Architecture in Chapel*
  - KISTI booth (#3713): Sung-Eun Choi (Cray Inc.)
- **Poster** (Tues @ 5:15): *Towards Co-Evolution of Auto-Tuning and Parallel Languages*
  - Posters Session: Ray Chen (University of Maryland)
- **Chapel Lightning Talks BoF** (Wed @ 12:15)
  - 5-minute talks on education, MPI-3, Big Data, Autotuning, Futures, MiniMD
- **Talk** (Wed @ 4:30): *Chapel, an Emerging Parallel Language*
  - HPC Impact Theatre (booth #3947): Brad Chamberlain (Cray Inc.)
- **Happy Hour** (Wed @ 5pm): *4th annual Chapel Users Group (CHUG) Happy Hour*
  - Pi Bar (just across the street at 1400 Welton St): open to public, dutch treat
- **HPC Education** (Thus @ 1:30pm): *High-Level Parallel Programming Using Chapel*
  - David Bunde (Knox College) and Kyle Burke (Colby College)