CSE 501 Language Issues
Languages for High Performance Computing (HPC) and Parallelization

Brad Chamberlain, Chapel Team, Cray Inc.
UW CSE 501, Spring 2015
May 5th, 2015
Chapel: The Language for HPC/Parallelization*

Brad Chamberlain, Chapel Team, Cray Inc.
UW CSE 501, Spring 2015
May 5th, 2015

* NOTE: speaker may be somewhat biased
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.
“Who is this guy Alvin dumped on us?”

2001: graduated from UW CSE with a PhD
  ● worked on the ZPL parallel programming language
  ● advisor: Larry Snyder (now Emeritus)

2001-2002: spent a lost/educational year at a startup

2002-present: have been working at Cray Inc.
  ● Hired to help with the HPCS program (see 2\textsuperscript{nd} slide following)
  ● Convinced execs/customers that we should do a language

Also a UW CSE affiliate faculty member
Ground Rules

- Please feel encouraged to ask questions as we go
  - I’ll throttle as necessary (my slides or the questions)

- Optionally: Grab lunch afterwards
Chapel’s Origins: HPCS

DARPA HPCS: High Productivity Computing Systems

- **Goal**: improve productivity by a factor of 10x
- **Timeframe**: Summer 2002 – Fall 2012
- Cray developed a new system architecture, network, software stack…
  - this became the very successful Cray XC30™ Supercomputer Series

...and a new programming language: Chapel
What is Chapel?

● An emerging parallel programming language
  ● Design and development led by Cray Inc.
    ● in collaboration with academia, labs, industry; domestically & internationally

● A work-in-progress
  ● Being developed as open source at GitHub
    ● Uses Apache v2.0 license
  ● Portable design and implementation, targeting:
    ● multicore desktops and laptops
    ● commodity clusters and the cloud
    ● HPC systems from Cray and other vendors
      ● in-progress: manycore processors, CPU+accelerator hybrids, …

Goal: Improve productivity of parallel programming
What does “Productivity” mean to you?

Recent Graduates:
“something similar to what I used in school: Python, Matlab, Java, …”

Seasoned HPC Programmers:
“that sugary stuff that I don’t need because I was born to suffer”
“want full control to ensure performance”

Computational Scientists:
“something that lets me express my parallel computations without having to wrestle with architecture-specific details”

Chapel Team:
“something that lets computational scientists express what they want, without taking away the control that HPC programmers need, implemented in a language as attractive as recent graduates want.”
A Stencil Computation in Chapel
Chapel Stencil Example: Jacobi Iteration

\[
A: \sum \begin{bmatrix}
1.0
\end{bmatrix} \div 4
\]

repeat until max change < \(\varepsilon\)
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
    D = BigD[1..n, 1..n],
    LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;
A[LastRow] = 1.0;

do {
    forall (i,j) in D do

    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
    D = BigD[1..n, 1..n],
    LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;
A[LastRow] = 1.0;
do {
    forall (i,j) in D do
    const delta = max reduce abs(A[D] - Temp[D];
    A[D] = Temp[D];
} while (delta > epsilon);
writeln(A);
```

Declare program parameters

- **config** ⇒ can’t change values after initialization
- **const** ⇒ can be set on executable command-line
  ```chapel
  prompt> jacobi --n=10000 --epsilon=0.0001
  ```

Note that no types are given; they’re inferred from initializers

- **n** ⇒ default integer (64 bits)
- **epsilon** ⇒ default real floating-point (64 bits)
Jacobi Iteration in Chapel

```chapel
config const n = 6,
epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
D = BigD[1..n, 1..n],
LastRow = D.exterior(1,0);

var A, Temp : [BigD]real;
A[LastRow] = 1.0;
do {
    forall (i,j) in D do
    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
} while (delta > epsilon);
writeln(A);
```

**Declare domains (first class index sets)**

{lo..hi, lo2..hi2} ⇒ 2D rectangular domain, with 2-tuple indices

Dom1[Dom2] ⇒ computes the intersection of two domains

.\texttt{exterior()} ⇒ one of several built-in domain generators
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
    D = BigD[1..n, 1..n],
    LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;
do {
    forall (i,j) in D do
    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
} while (delta > epsilon);
writeln (A);
```

Declare arrays

- `var` ⇒ can be modified throughout its lifetime
- `: [Dom] T` ⇒ array of size `Dom` with elements of type `T`
- `(no initializer)` ⇒ values initialized to default value (0.0 for reals)
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
    D = BigD[1..n, 1..n],
    LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do {
    forall (i,j) in D do
    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
} while (delta > epsilon);

writeln (A);
```

Set Explicit Boundary Condition

`Arr[Dom]` ⇒ refer to array slice (“forall i in Dom do …Arr[i]…”)

```chapel
A
```
Jacobi Iteration in Chapel

config const n = 6,

Compute 5-point stencil

forall ind in Dom ⇒ parallel forall expression over Dom’s indices, binding them to ind
(here, since Dom is 2D, we can de-tuple the indices)

\[
\sum \left( \begin{array}{c}
\text{orange} \\
\text{yellow} \\
\text{green} \\
\text{blue}
\end{array} \right) \div 4
\]

do {
  forall (i,j) in D do

  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
D = BigD[1..n, 1..n],
LastRow = D.exterior(1, 0);

var A, Temp : [BigD]

var A[LastRow] = 1.0;

do {
    forall (i,j) in D do

    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

**Compute maximum change**

*op reduce* ⇒ collapse aggregate expression to scalar using *op*

**Promotion:** *abs()* and − are scalar operators; providing array operands results in *promotion*—parallel evaluation equivalent to:

```chapel
do {
    forall (a,t) in zip(A,Temp) do abs(a - t)
}
```

Copyright 2015 Cray Inc.
Jacobi Iteration in Chapel

```chapel
config const n = 6,
     epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
D = BigD[1..n, 1..n],

LastRow = D.exterior(1, 0);

var A, Temp : [BigD]real;
A[LastRow] = 1.0;

do {
    forall (i,j) in D do

    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

Copy data back & Repeat until done

uses slicing and whole-array assignment
standard `do...while` loop construct
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
    D = BigD[1..n, 1..n],
    LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;
A[LastRow] = 1.0;

do {
    forall (i,j) in D do

    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

Write array to console
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 (compute nodes)?
- Completely local to one locale? Or distributed?
- If distributed… In a blocked manner? cyclically? block-cyclically? recursively bisected? dynamically rebalanced? …?
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 (compute nodes)?
- 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
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
    D = BigD[1..n, 1..n],
    LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;
do {
    forall (i,j) in D do
    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

By default, domains and their arrays are mapped to a single locale. Any data parallelism over such domains/arrays will be executed by the cores on that locale. Thus, this is a shared-memory/multi-core parallel program.
Jacobi Iteration in Chapel

```chapel
config const n = 6,
       epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1} dmapped Block({1..n, 1..n}),
            D = BigD[1..n, 1..n],
            LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;

With this simple change, we specify a mapping from the domains and arrays to locales:
- Domain maps describe the mapping of domain indices and array elements to locales.
- Specifies how array data is distributed across locales.
- Specifies how iterations over domains/arrays are mapped to locales.
```
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1} dmapped Block({1..n, 1..n}),
    D = BigD[1..n, 1..n],
    LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;

do {
    forall (i,j) in D do
    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
} while (delta > epsilon);

writeln (A);
```

With this simple change, we specify a mapping from the domains and arrays to locales. Domain maps describe the mapping of domain indices and array elements to locales. Specifies how array data is distributed across locales. Specifies how iterations over domains/arrays are mapped to locales. …including multicore parallelism.
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1} mapped Block({1..n, 1..n}),
    D = BigD[1..n, 1..n],
    LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;
A[LastRow] = 1.0;

do {
    forall (i,j) in D do

    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);

use BlockDist;
```
**Goal:** Solve one octant of the spherical Sedov problem (blast wave) using Lagrangian hydrodynamics for a single material

pictures courtesy of Rob Neely, Bert Still, Jeff Keasler, LLNL
LULESH in Chapel
LULESH in Chapel

1288 lines of source code
plus 266 lines of comments
487 blank lines

(the corresponding C+MPI+OpenMP version is nearly 4x bigger)

This can be found in Chapel v1.9 in examples/benchmarks/lulesh/*chpl
This is all of the representation dependent code. It specifies:

- data structure choices
- structured vs. unstructured mesh
  - local vs. distributed data
- sparse vs. dense materials arrays
- a few supporting iterators

Chapel’s *domain maps* make this possible.
Domain Maps

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

$A = B + \alpha \cdot C$;

…to the target locales’ memory and processors:
Chapel Domain Types

- **dense**
- **strided**
- **sparse**
- **associative**
- **unstructured**
Chapel Array Types

- dense
- strided
- sparse
- associative
- unstructured
All Domain Types Support Domain Maps

- dense
- strided
- sparse
- associative
- unstructured
Chapel’s Domain Map Philosophy

1. Chapel provides a library of standard domain maps
   ● to support common array implementations effortlessly

2. Expert users can write their own domain maps in Chapel
   ● to cope with any 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
Motivating Chapel Themes

1) General Parallel Programming
2) Global-View Abstractions
3) Multiresolution Design
4) Control over Locality/Affinity
5) Reduce HPC ↔ Mainstream Language Gap
Motivating Chapel Themes

1) General Parallel Programming
2) Global-View Abstractions
3) Multiresolution Design
4) Control over Locality/Affinity
5) Reduce HPC ↔ Mainstream Language Gap
3) Multiresolution Design: Motivation

“Why is everything so tedious/difficult?”
“What don’t my programs port trivially?”

“Why don’t I have more control?”
3) 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
- permit the user to intermix layers arbitrarily
Domain Maps Summary

- Data locality requires mapping arrays to memory well
  - distributions between distinct memories
  - layouts within a single memory

- Most languages define a single data layout & distribution
  - where the distribution is often the degenerate “everything’s local”

- Domain maps…
  …move such policies into user-space…
  …exposing them to the end-user through high-level declarations

```c
const Elems = {0..#numElems} dmapped Block(...)
```
Two Other Thematically Similar Features

1) **parallel iterators:** Define parallel loop policies

2) **locale models:** Define target architectures

Like domain maps, these are…

...written in Chapel by expert users using lower-level features

- e.g., task parallelism, on-clauses, base language features, …

...available to the end-user via higher-level abstractions

- e.g., forall loops, on-clauses, lexically scoped PGAS memory, …
Multiresolution Summary

Chapel’s multiresolution philosophy allows users to write…
…custom array implementations via domain maps

…custom parallel iterators via parallel iterators

…custom architectural models via hierarchical locales

The result is a language that decouples crucial policies for managing data locality out of the language’s definition and into an expert user’s hand…

…while making them available to end-users through high-level abstractions
For More Information on…

…domain maps


…parallel iterators

*User-Defined Parallel Zippered Iterators in Chapel [slides]*, Chamberlain, Choi, Deitz, Navarro; PGAS 2011, October 2011.

…hierarchical locales


**Status:** all of these concepts are in-use in every Chapel program today (pointers to code/docs in the release available by request)
Outline

- Setting
- Chapel By Example: Jacobi Stencil
- Multiresolution Philosophy: Domain Maps and such

- Chapel Motivation
  - Parallel Programming Model Taxonomy, Pluses/Minuses
  - Chapel Motivating Themes
  - Survey of Chapel Concepts
  - Compiling Chapel
  - Project Status and Next Steps
Sustained Performance Milestones

1 GF – 1988: Cray Y-MP; 8 Processors
- Static finite element analysis

1 TF – 1998: Cray T3E; 1,024 Processors
- Modeling of metallic magnet atoms

1 PF – 2008: Cray XT5; 150,000 Processors
- Superconductive materials

1 EF – ~2018: Cray ____; ~10,000,000 Processors
- TBD
Sustained Performance Milestones

1 GF – 1988: Cray Y-MP; 8 Processors
- Static finite element analysis
- Fortran77 + Cray autotasking + vectorization

1 TF – 1998: Cray T3E; 1,024 Processors
- Modeling of metallic magnet atoms
- Fortran + MPI (Message Passing Interface)

1 PF – 2008: Cray XT5; 150,000 Processors
- Superconductive materials
- C++/Fortran + MPI + vectorization

1 EF – ~2018: Cray ____; ~10,000,000 Processors
- TBD
- TBD: C/C++/Fortran + MPI + OpenMP/OpenACC/CUDA/OpenCL?

Or, perhaps something completely different?
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: 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:
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):**

![Diagram of STREAM Triad computation]
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):
#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 );

    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);
}

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;
}
#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);
Prototypical Next-Gen Processor Technologies

Intel Phi

AMD APU

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
(“Glad I’m not an HPC Programmer!”)

A Possible Reaction:
“This is all well and good for HPC users, but I’m a mainstream desktop programmer, so this is all academic for me.”

The Unfortunate Reality:
● Performance-minded mainstream programmers will increasingly need to deal with parallelism and locality too.
HPC suffers from too many distinct notations for expressing parallelism and locality

(Check how much more different these would be if we were actually doing something interesting, like a stencil!)
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

<table>
<thead>
<tr>
<th>Type of HW Parallelism</th>
<th>Programming Model</th>
<th>Unit of Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inter-node</td>
<td>MPI</td>
<td>executable</td>
</tr>
<tr>
<td>Intra-node/multicore</td>
<td>OpenMP/pthreads</td>
<td>iteration/task</td>
</tr>
<tr>
<td>Instruction-level</td>
<td>pragmas</td>
<td>iteration</td>
</tr>
<tr>
<td>Instruction-level</td>
<td>CUDA/OpenCL/OpenACC</td>
<td>SIMD function/task</td>
</tr>
</tbody>
</table>

benefits: lots of control; decent generality; easy to implement
downsides: lots of user-managed detail; brittle to changes
What a bunch of gear we have to carry around! This is getting old…

I guess we need a canoe?!

Oops, need my ice axe

OK, let’s upgrade to hiking boots

OK, got my walking shoes on!

By Analogy: Let’s Cross the United States!
By Analogy: Let’s Cross the United States!

...Hey, what’s that sound?
Rewinding a few slides...

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

```
#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);
    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];
}
```
**STREAM Triad: Chapel**

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

class ProblemSpace = {1..m};

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

B = 2.0;
C = 3.0;

A = B + alpha * C;
```

**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.
Outline

✓ Setting
✓ Chapel By Example: Jacobi Stencil
✓ Multiresolution Philosophy: Domain Maps and such
✓ Chapel Motivation

➢ Parallel Programming Model Taxonomy, Pluses/Minuses
  ● Chapel Motivating Themes
  ● Survey of Chapel Concepts
  ● Compiling Chapel
  ● Project Status and Next Steps
Global Address Space Programming Models (Shared Memory)

e.g., OpenMP, Pthreads

+ support dynamic, fine-grain parallelism
+ considered simpler, more like traditional programming
  - “if you want to access something, simply name it”
  - no support for expressing locality/affinity; limits scalability
  - bugs can be subtle, difficult to track down (race conditions)
  - tend to require complex memory consistency models
Message Passing Programming Models (Distributed Memory)

e.g., MPI

- a more constrained model; can only access local data
- runs on most large-scale parallel platforms
  - and for many of them, can achieve near-optimal performance
- is *relatively* easy to implement
- can serve as a strong foundation for higher-level models
- users have been able to get real work done with it
Message Passing Programming Models (Distributed Memory)

e.g., MPI

- communication must be used to get copies of remote data
  - tends to reveal too much about how to transfer data, not simply what
- only supports “cooperating executable”-level parallelism
- couples data transfer and synchronization
- has frustrating classes of bugs of its own
  - e.g., mismatches between sends/recvs, buffer overflows, etc.
Hybrid Programming Models

e.g., MPI+OpenMP/Pthreads/CUDA, UPC+OpenMP, …

+ supports a division of labor: each handles what it does best
+ permits overheads to be amortized across processor cores, as compared to using MPI alone
  – requires multiple notations to express a single logical parallel algorithm, each with its own distinct semantics
Traditional PGAS Languages

e.g., Co-Array Fortran, UPC
+ support a shared namespace, like shared-memory
+ support a strong sense of ownership and locality
  • each variable is stored in a particular memory segment
  • tasks can access any visible variable, local or remote
  • local variables are cheaper to access than remote ones
+ implicit communication eases user burden; permits compiler to use best mechanisms available
Traditional PGAS Languages

e.g., Co-Array Fortran, UPC

- restricted to SPMD programming and execution models
- data structures not as flexible/rich as one might like
- retain many of the downsides of shared-memory
  - error cases, memory consistency models
Next-Generation PGAS Languages

e.g., Chapel (also Charm++, X10, Fortress, …)

+ breaks out of SPMD mold via global multithreading
+ richer set of distributed data structures
  – retains many of the downsides of shared-memory
    ● error cases, memory consistency models
Outline

✓ Setting
✓ Chapel By Example: Jacobi Stencil
✓ Multiresolution Philosophy: Domain Maps and such
✓ Chapel Motivation
✓ Parallel Programming Model Taxonomy, Pluses/Minuses

➢ Chapel Motivating Themes
  ● Survey of Chapel Concepts
  ● Compiling Chapel
  ● Project Status and Next Steps
Motivating Chapel Themes

1) General Parallel Programming
2) Global-View Abstractions
3) Multiresolution Design
4) Control over Locality/Affinity
5) Reduce HPC ↔ Mainstream Language Gap
1) 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 any parallelism available in the hardware
  ● **Types:** machines, nodes, cores, instructions

<table>
<thead>
<tr>
<th>Type of HW Parallelism</th>
<th>Programming Model</th>
<th>Unit of Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inter-node</td>
<td>MPI</td>
<td>executable</td>
</tr>
<tr>
<td>Intra-node/multicore</td>
<td>OpenMP/pthreads</td>
<td>iteration/task</td>
</tr>
<tr>
<td>Instruction-level vectors/threads</td>
<td>pragmas</td>
<td>iteration</td>
</tr>
<tr>
<td>GPU/accelerator</td>
<td>CUDA/OpenCL/OpenACC</td>
<td>SIMD function/task</td>
</tr>
</tbody>
</table>
1) 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 any parallelism available in the hardware

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

<table>
<thead>
<tr>
<th>Type of HW Parallelism</th>
<th>Programming Model</th>
<th>Unit of Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inter-node</td>
<td>Chapel</td>
<td>executable/task</td>
</tr>
<tr>
<td>Intra-node/multicore</td>
<td>Chapel</td>
<td>iteration/task</td>
</tr>
<tr>
<td>Instruction-level vectors/threads</td>
<td>Chapel</td>
<td>iteration</td>
</tr>
<tr>
<td>GPU/accelerator</td>
<td>Chapel</td>
<td>SIMD function/task</td>
</tr>
</tbody>
</table>

Copyright 2015 Cray Inc.
2) Global-View Abstractions

In pictures: “Apply a 3-Point Stencil to a vector”

Global-View

\[
\frac{(\text{Global-View})}{2} + \frac{(\text{Local-View})}{2}
\]

Local-View

\[
\frac{(\text{Global-View})}{2}
\]

\[
\frac{(\text{Local-View})}{2}
\]
2) Global-View Abstractions

In pictures: “Apply a 3-Point Stencil to a vector”
2) Global-View Abstractions

In code: “Apply a 3-Point Stencil to a vector”

Global-View

```chapel
proc main() {
    var n = 1000;
    var A, B: [1..n] real;

    forall i in 2..n-1 do
        B[i] = (A[i-1] + A[i+1])/2;
}
```

Local-View (SPMD)

```chapel
proc main() {
    var n = 1000;
    var p = numProcs(),
        me = myProc(),
        myN = n/p,
    var A, B: [0..myN+1] real;

    if (me < p-1) {
        send(me+1, A[myN]);
        recv(me+1, A[myN+1]);
    }
    if (me > 0) {
        send(me-1, A[1]);
        recv(me-1, A[0]);
    }
    forall i in 1..myN do
        B[i] = (A[i-1] + A[i+1])/2;
}
```

Bug: Refers to uninitialized values at ends of A
2) Global-View Abstractions

In code: “Apply a 3-Point Stencil to a vector”

Global-View

```
proc main() {
    var n = 1000;
    var A, B: [1..n] real;
    forall i in 2..n-1 do
        B[i] = (A[i-1] + A[i+1])/2;
}
```

Local-View (SPMD)

```
proc main() {
    var n = 1000;
    var p = numProcs(),
        me = myProc(),
        myN = n/p,
        myLo = 1,
        myHi = myN;
    var A, B: [0..myN+1] real;
    if (me < p-1) {
        send(me+1, A[myN]);
        recv(me+1, A[myN+1]);
    } else
        myHi = myN-1;
    if (me > 0) {
        send(me-1, A[1]);
        recv(me-1, A[0]);
    } else
        myLo = 2;
    forall i in myLo..myHi do
        B[i] = (A[i-1] + A[i+1])/2;
}
```

Communication becomes geometrically more complex for higher-dimensional arrays

Assumes p divides n
2) Global-View Programming: A Final Note

- A language may support both global- and local-view programming — in particular, Chapel does

```plaintext
proc main() {
    coforall loc in Locales do
        on loc do
            MySPMDProgram(loc.id, Locales.numElements);
}

proc MySPMDProgram(myImageID, numImages) {
    ...
}
```
4) Control over Locality/Affinity

Consider:
- Scalable architectures package memory near processors
- Remote accesses take longer than local accesses

Therefore:
- Placement of data relative to tasks affects scalability
- Give programmers control of data and task placement

Note:
- Over time, we expect locality to matter more and more within the compute node as well
Partitioned Global Address Space (PGAS) Languages

(Or perhaps: partitioned global namespace languages)

- **abstract concept:**
  - support a shared namespace on distributed memory
    - permit parallel tasks to access remote variables by naming them
  - establish a strong sense of ownership
    - every variable has a well-defined location
    - local variables are cheaper to access than remote ones

- **traditional PGAS languages have been SPMD in nature**
  - best-known examples: Co-Array Fortran, UPC

```
<table>
<thead>
<tr>
<th>partitioned shared name-/address space</th>
</tr>
</thead>
<tbody>
<tr>
<td>private space 0</td>
</tr>
<tr>
<td>private space 1</td>
</tr>
<tr>
<td>private space 2</td>
</tr>
<tr>
<td>private space 3</td>
</tr>
<tr>
<td>private space 4</td>
</tr>
</tbody>
</table>
```
Chapel and PGAS

● Chapel is PGAS, but unlike most, it’s not restricted to SPMD

⇒ never think about “the other copies of the program”
⇒ “global name/address space” comes from lexical scoping
  ● as in traditional languages, each declaration yields one variable
  ● variables are stored on the locale where the task declaring it is executing

Locales (think: “compute nodes”)
5) Reduce HPC ↔ Mainstream Language Gap

Consider:
- Students graduate with training in Java, Matlab, Python, etc.
- Yet HPC programming is dominated by Fortran, C/C++, MPI

We’d like to narrow this gulf with Chapel:
- to leverage advances in modern language design
- to better utilize the skills of the entry-level workforce...
- ...while not alienating the traditional HPC programmer
  - e.g., support object-oriented programming, but make it optional
Outline

- Blah blah blah
- Blah blah blah

- Survey of Chapel Concepts

- Domain Maps
- Data Parallelism
- Task Parallelism
- Base Language
- Locality Control

- Target Machine

- Blah blah blah
const pi = 3.14,       // pi is a real
    coord = 1.2 + 3.4i, // coord is a complex...
    coord2 = pi*coord,  // ...as is coord2
    name = “brad”,     // name is a string
    verbose = false;   // verbose is boolean

proc addem(x, y) {       // addem() has generic arguments
    return x + y;        // and an inferred return type
}

var sum = addem(1, pi),       // sum is a real
    fullname = addem(name, “ford”); // fullname is a string

writeln((sum, fullname));

(4.14, bradford)
Range Types and Algebra

```chapel
const r = 1..10;

printVals(r);
printVals(r # 3);
printVals(r by 2);
printVals(r by -2);
printVals(r by 2 # 3);
printVals(r # 3 by 2);
printVals(0.. #n);

proc printVals(r) {
    for i in r do
        write(r, " ");
        writeln();
}
```

```
1 2 3 4 5 6 7 8 9 10
1 2 3
1 3 5 7 9
10 8 6 4 2
1 3 5
1 3
0 1 2 3 4 ... n-1
```
Iterators

```plaintext
iter fibonacci(n) {
    var current = 0,
        next = 1;
    for 1..n {
        yield current;
        current += next;
        current <=> next;
    }
}
```

```plaintext
for f in fibonacci(7) do writeln(f);
0 1 1 2 3 5 8
```

```plaintext
iter tiledRMO(D, tiles) {
    const tile = {0..#tiles, 0..#tiles};
    for base in D by tiles do
        for ij in D[tile + base] do
            yield ij;
}
```

```plaintext
for ij in tiledRMO({1..m, 1..n}, 2) do write(ij);
(1,1) (1,2) (2,1) (2,2)
(1,3) (1,4) (2,3) (2,4)
(1,5) (1,6) (2,5) (2,6)
...
(3,1) (3,2) (4,1) (4,2)
```
Zippered Iteration

```plaintext
for (i, f) in zip(0..#n, fibonacci(n)) do
  writeln("fib ", i, " is ", f);

fib #0 is 0
fib #1 is 1
fib #2 is 1
fib #3 is 2
fib #4 is 3
fib #5 is 5
fib #6 is 8
...
```
Other Base Language Features

- tuple types and values
- rank-independent programming features
- interoperability features
- compile-time features for meta-programming
  - e.g., compile-time functions to compute types, parameters
- OOP (value- and reference-based)
- argument intents, default values, match-by-name
- overloading, where clauses
- modules (for namespace management)
- …
Outline

- Blah blah blah
- Blah blah blah

➢ Survey of Chapel Concepts

- Domain Maps
- Data Parallelism
- Task Parallelism
- Base Language
- Locality Control
- Target Machine

- Blah blah blah
Defining our Terms

**Task:** a unit of computation that can/should execute in parallel with other tasks

**Task Parallelism:** a style of parallel programming in which parallelism is driven by programmer-specified tasks

(in contrast with):

**Data Parallelism:** a style of parallel programming in which parallelism is driven by computations over collections of data elements or their indices
Task Parallelism: Begin Statements

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

Possible outputs:

```
hello world
goodbye
hello world
```
Task Parallelism: Coforall Loops

```chapel
// 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
```
Task Parallelism: Data-Driven Synchronization

1) **atomic variables**: support atomic operations (as in C++)
   - e.g., compare-and-swap; atomic sum, mult, etc.

2) **single-assignment variables**: reads block until assigned

3) **synchronization variables**: store full/empty state
   - by default, reads/writes block until the state is full/empty
Outline

- Blah blah blah
- Blah blah blah

Survey of Chapel Concepts

<table>
<thead>
<tr>
<th>Domain Maps</th>
<th>Data Parallelism</th>
<th>Task Parallelism</th>
<th>Base Language</th>
<th>Locality Control</th>
<th>Target Machine</th>
</tr>
</thead>
</table>

- Blah blah blah
Partitioned Global Address Space (PGAS) Languages

(Or perhaps: partitioned global namespace languages)

- **abstract concept:**
  - support a shared namespace on distributed memory
    - permit parallel tasks to access remote variables by naming them
  - establish a strong sense of ownership
    - every variable has a well-defined location
    - local variables are cheaper to access than remote ones

- **traditional PGAS languages have been SPMD in nature**
  - best-known examples: Co-Array Fortran, UPC

<table>
<thead>
<tr>
<th>partitioned shared name-/address space</th>
</tr>
</thead>
<tbody>
<tr>
<td>private space 0</td>
</tr>
<tr>
<td>private space 1</td>
</tr>
<tr>
<td>private space 2</td>
</tr>
<tr>
<td>private space 3</td>
</tr>
<tr>
<td>private space 4</td>
</tr>
</tbody>
</table>
Chapel and PGAS

- Chapel is PGAS, but unlike most, it’s not restricted to SPMD
  - never think about “the other copies of the program”
  - “global name/address space” comes from lexical scoping
    - as in traditional languages, each declaration yields one variable
    - variables are stored on the locale where the task declaring it is executing

![Locales diagram](Image)
Chapel: Scoping and Locality

\[
\text{var } i : \text{ int;}
\]
Chapel: Scoping and Locality

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

Locales (think: “compute nodes”)
Chapel: Scoping and Locality

```chapel
var i: int;

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

Locales (think: “compute nodes”)
```chapel
var i: int;
on Locales[1] {
    var j: int;
    coforall loc in Locales {
        on loc {
```

**Locales** (think: “compute nodes”)

- **i**
- **j**
- 0
- 1
- 2
- 3
- 4
Chapel: Scoping and Locality

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

      // within this scope, i, j, and k can be referenced;
      // the implementation manages the communication for i and j
    }
  }
}
```

Locales (think: “compute nodes”)
The Locale Type

Definition:

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

Typically: A compute node (multicore processor or SMP)
Getting started with locales

- Specify # of locales when running Chapel programs

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

- Chapel provides built-in locale variables

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

- User’s `main()` begins executing on locale #0
Locale Operations

- Locale methods support queries about the target system:

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

- *On-clauses* support placement of computations:

  ```chapel
  writeln("on locale 0");
  on Locales[1] do
    writeln("now on locale 1");
  writeln("on locale 0 again");
  begin on A[i,j] do
    bigComputation(A);
  begin on node.left do
    search(node.left);
  ```
Parallelism and Locality: Orthogonal in Chapel

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

```chapel
begin writeln(“Hello world!”);
writeln(“Goodbye!”);
```

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

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

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

```chapel
begin on Locales[1] do writeln(“Hello from locale 1!”);
on Locales[2] do begin writeln(“Hello from locale 2!”);
writeln(“Goodbye from locale 0!”);
```
Outline

✓ Blah blah blah
✓ Blah blah blah

Survey of Chapel Concepts
- You’ve had a good taste of this, but there’s more as well…

● Compiling Chapel
Notes on Forall Loops

```
forall a in A do
writeln("Here is an element of A: ", a);
```

Typically:
- \(1 \leq \#\text{Tasks} \ll \#\text{Iterations}\)
- \(\#\text{Tasks} \approx \) amount of HW parallelism

```
forall (a, i) in zip(A, 1..n) do
  a = i / 10.0;
```

Like for loops, forall-loops may be zippered, and corresponding iterations will match up
Promotion Semantics

Promoted functions/operators are defined in terms of zippered forall loops in Chapel. For example...

```plaintext
A = B;
```

...is equivalent to:

```plaintext
forall (a,b) in zip(A,B) do
  a = b;
```
Impact of Zippered Promotion Semantics

Whole-array operations are implemented element-wise...

\[
A = B + \text{alpha} \times C; \quad \Rightarrow \quad \text{forall } (a,b,c) \text{ in } (A,B,C) \text{ do } \quad a = b + \text{alpha} \times c;
\]

...rather than operator-wise.

\[
A = B + \text{alpha} \times C; \quad \Rightarrow \quad T1 = \text{alpha} \times C; \quad A = B + T1;
\]

⇒ No temporary arrays required by semantics
⇒ No surprises in memory requirements
⇒ Friendlier to cache utilization

⇒ Differs from traditional array language semantics

\[
A[D] = A[D\text{-}one] + A[D\text{+}one]; \quad \Rightarrow \quad \text{forall } (a1, a2, a3) \text{ in } (A[D], A[D\text{-}one], A[D\text{+}one]) \text{ do } \quad a1 = a2 + a3;
\]

Read/write race!
Sample Distributions: Block and Cyclic

var Dom = {1..4, 1..8} dmapped Block( {1..4, 1..8} );

distributed to

var Dom = {1..4, 1..8} dmapped Cyclic( startIdx=(1,1) );

distributed to
Domain Map Descriptors

**Domain Map**

- **Represents:** a domain map value
- **Generic w.r.t.:** index type
- **State:** the domain map’s representation
- **Typical Size:** $\Theta(1)$
- **Required Interface:**
  - create new domains

**Domain**

- **Represents:** a domain
- **Generic w.r.t.:** index type
- **State:** representation of index set
- **Typical Size:** $\Theta(1) \rightarrow \Theta(\text{numIndices})$
- **Required Interface:**
  - create new arrays
  - queries: size, members
  - iterators: serial, parallel
  - domain assignment
  - index set operations

**Array**

- **Represents:** an array
- **Generic w.r.t.:** index type, element type
- **State:** array elements
- **Typical Size:** $\Theta(\text{numIndices})$
- **Required Interface:**
  - (re-)allocation of elements
  - random access
  - iterators: serial, parallel
  - slicing, reindexing, aliases
  - get/set of sparse “zero” values
Outline

- Setting
- Chapel By Example: Jacobi Stencil
- Multiresolution Philosophy: Domain Maps and such
- Chapel Motivation
- Parallel Programming Model Taxonomy, Pluses/Minuses
- Chapel Motivating Themes
- Survey of Chapel Concepts
- Compiling Chapel
- Project Status and Next Steps
Compiling Chapel

Chapel Source Code → chpl → Chapel Executable

Standard Modules (in Chapel)
Chapel Compiler Architecture

Chapel Source Code → Chapel-to-C Compiler → Generated C Code → Standard C Compiler & Linker → Chapel Executable

- Standard Modules (in Chapel)
- Internal Modules (in Chapel)
- Runtime Support Library (in C)
  - Tasks/Threads
  - Communication
  - Memory
  - ...

Copyright 2015 Cray Inc.
Key Compiler Passes Required by Chapel

- Transform higher-level Chapel constructs to C
  - iterators
  - overloading, classes, generics, where clauses, tuples, …

- Transform parallel constructs to C routines

- Transform on-clauses to C routines
Key Compiler Analyses Required by Chapel

- **Static type inference + Function Resolution**
- **Multiresolution Optimizations**
  - Given plug-in nature of…
    - domain maps
    - parallel iterators
    - locale models
  - …how to get performance competitive with C/Fortran?
- **Locality Analysis (in the “locale” sense)**
  - What might be referred to remotely? What is known to be local?
- **Communication Optimizations**
  - Overlap of communication and computation to hide latency
  - Combining similar/Eliminating redundant communications
  - (still haven’t caught up to ZPL work, in this regard)
- **Plus, traditional optimizations (LICM, DCE, scalar repl., …)**
Outline

 ✓ Setting
 ✓ Chapel By Example: Jacobi Stencil
 ✓ Multiresolution Philosophy: Domain Maps and such
 ✓ Chapel Motivation
 ✓ Parallel Programming Model Taxonomy, Pluses/Minuses
 ✓ Chapel Motivating Themes
 ✓ Survey of Chapel Concepts
 ✓ Compiling Chapel

 ➤ Project Status and Next Steps
The Cray Chapel Team (Summer 2014)
Chapel...

...is a collaborative effort — join us!
A Year in the Life of Chapel

- **Two major releases per year** (April / October)
  - latest release: version 1.11, April 2nd, 2015
  - ~a month later: detailed release notes
    - version 1.11 release notes: [http://chapel.cray.com/download.html#releaseNotes](http://chapel.cray.com/download.html#releaseNotes)

- **CHIUW**: Chapel Implementers and Users Workshop (May-June)
  - workshop focusing on community efforts, code camps
  - this year will be held in Portland, June 13-14

- **SC** (Nov)
  - the primary conference for the HPC industry
  - we give tutorials, BoFs, talks, etc. to show off year’s work

- **Talks, tutorials, research visits, blogs, ...** (year-round)
Implementation Status -- Version 1.11 (Apr 2015)

Overall Status:

● **User-facing Features:** generally in good shape
  ● some require additional attention (e.g., strings, memory mgmt)
● **Multiresolution Features:** in use today
  ● their interfaces are likely to continue evolving over time
● **Error Messages:** not always as helpful as one would like
  ● correct code works well, incorrect code can be puzzling
● **Performance:** hit-or-miss depending on the idioms used
  ● Chapel designed to ultimately support competitive performance
  ● to-date, we’ve focused primarily on correctness and local perf.

This is a great 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
When 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 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: the five-year push

- **Harden prototype to production-grade**
  - add/improve lacking features
  - optimize performance
  - improve interoperability

- **Target more complex/modern compute node types**
  - e.g., Intel Phi, CPU+GPU, AMD APU, ...

- **Continue to grow the user and developer communities**
  - including nontraditional circles: desktop parallelism, “big data”
  - transition Chapel from Cray-managed to community-governed
Summary

**Higher-level programming models can help insulate algorithms from parallel implementation details**

- yet, without necessarily abdicating control
- Chapel does this via its multiresolution design
  - here, we saw it principally in domain maps
  - parallel iterators and locale models are other examples
  - these avoid locking crucial policy decisions into the language

**We believe Chapel can greatly improve productivity**

...for current and emerging HPC architectures

...for emerging mainstream needs for parallelism and locality
For More Information: Online Resources

Chapel project page: [http://chapel.cray.com](http://chapel.cray.com)
- overview, papers, presentations, language spec, …

Chapel GitHub page: [https://github.com/chapel-lang](https://github.com/chapel-lang)
- download 1.11.0 release, browse source repository

Chapel Facebook page: [https://www.facebook.com/ChapelLanguage](https://www.facebook.com/ChapelLanguage)
For More Information: Community Resources

Chapel SourceForge page: [https://sourceforge.net/projects/chapel/](https://sourceforge.net/projects/chapel/)
- join community mailing lists; alternative release download site

Mailing Aliases:
- chapel_info@cray.com: contact the team at Cray
- chapel-announce@lists.sourceforge.net: list for announcements only
- chapel-users@lists.sourceforge.net: user-oriented discussion list
- chapel-developers@lists.sourceforge.net: developer discussion
- chapel-education@lists.sourceforge.net: educator discussion
- chapel-bugs@lists.sourceforge.net: public bug forum
Overview Papers:

  ● *a detailed overview of Chapel’s history, motivating themes, features*

  ● *a higher-level overview of the project, summarizing the HPCS period*
For More Information: Lighter Reading

Blog Articles:

  - a short-and-sweet introduction to Chapel

  - a current series of articles answering common questions about why we are pursuing Chapel in spite of the inherent challenges

  - a series of technical opinion pieces designed to combat standard arguments against the development of high-level parallel languages
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, ECOPLEX, 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.

Copyright 2015 Cray Inc.