Chapel:
Parallel Programmability
from Desktops to Supercomputers

Brad Chamberlain, Chapel Team, Cray Inc.
University of Copenhagen
February 4th, 2016
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.
Motivation

Q: Why doesn’t HPC programming have an equivalent to Python / Matlab / Java / C++ / (your favorite programming language here)?
   ● one that makes it easy to get programs up and running quickly
   ● one that is portable across system architectures and scales
   ● one that bridges the HPC, data analysis, and mainstream communities

A: We believe this is due not to any particular technical challenge, but rather a lack of sufficient…
   …long-term efforts
   …resources
   …community will
   …co-design between developers and users
   …patience

Chapel is our attempt to change this
What is Chapel?

**Chapel:** An emerging parallel programming language
- extensible
- portable
- open-source
- a collaborative effort
- a work-in-progress

**Goals:**
- Support general parallel programming
  - “any parallel algorithm on any parallel hardware”
- Make parallel programming far more productive
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 want,
implemented in a language as attractive as recent graduates want.”
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:

\[
\begin{align*}
A &= \underbrace{\ldots}^{m} \\
B &= \underbrace{\ldots}^{m} \\
C &= \underbrace{\ldots}^{m} \\
\alpha &= \text{constant}
\end{align*}
\]
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):

```
<table>
<thead>
<tr>
<th>A</th>
<th>=</th>
<th>=</th>
<th>=</th>
<th>=</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>C</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>\alpha</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>
```
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):

- \(A\)
- \(B\)
- \(C\)
- \(\alpha\)
#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

```c
#include <hpcc.h>
#endif _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);
}
```

#include `<hpcc.h>
 ifndef _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);
HPC suffers from too many distinct notations for expressing parallelism and locality.
Why so many programming models?

HPC tends to approach programming models bottom-up:
Given a system and its core capabilities...
...provide features that can access the available performance.
- portability, generality, programmability: not strictly necessary.

<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 / Open[CL</td>
<td>MP</td>
</tr>
</tbody>
</table>

**benefits:** lots of control; decent generality; easy to implement
**downsides:** lots of user-managed detail; brittle to changes
Rewinding a few slides...

**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;
}
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);
    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

Philosophy: Good, *top-down* language design can tease system-specific implementation details away from an algorithm, permitting the compiler, runtime, applied scientist, and HPC expert to each focus on their strengths.
Outline

✔ Motivation

➢ Chapel's Design Themes
  ● Survey of Chapel Concepts
  ● Project Status and Next Steps
Design Themes for Chapel

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

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

We’ll cover these as we go
1) General Parallel Programming

With a unified set of concepts...

...target any hardware parallelism available in the system

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

...express any software parallelism desired by the user

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

<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>task (or executable)</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 2016 Cray Inc.
2) 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
3) Multiresolution Design: Motivation

"Why is everything so tedious/difficult?"
"Why don’t my programs trivially port to new systems?"

"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
Outline

- Motivation
- Chapel's Design Themes

➢ Survey of Chapel Concepts

- Project Status and Next Steps
**Outline**

- Motivation
- Chapel's Design Themes
- Survey of Chapel Concepts

**● Project Status and Next Steps**
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)
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(i, " ");
    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

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

for f in fibonacci(7) do writeln(f);

0
1
1
2
3
5
8
```

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

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

✓ Motivation
✓ Chapel's Design Themes
➢ Survey of Chapel Concepts

● Project Status and Next Steps
Task Parallelism: Begin Statements

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

Possible outputs:

```
hello world
goodbye
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
```
Other Task Parallel Concepts

- **cobegins**: create tasks using compound statements

- **atomic variables**: support atomics ops, similar to modern C++

- **sync/single variables**: support producer/consumer patterns

- **sync statements**: join unstructured tasks

- **serial statements**: conditionally squash parallelism
Outline

✓ Motivation
✓ Chapel's Design Themes
➢ Survey of Chapel Concepts

● Project Status and Next Steps

Theme 4: Control over Locality/Affinity
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

  % 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");
  ```
  ```chapel
  on A[i,j] do
    bigComputation(A);
  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** and **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!”);
  ```
Chapel: Scoping and Locality

```chapel
var i: int;
```

Locales (think: “compute nodes”)
var i: int;
on Locales[1] {
Chapel: Scoping and Locality

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

**Locales** (think: “compute nodes”)

- **i**: 0
- **j**: 1
- 2
- 3
- 4
```chapel
var i: int;
on Locales[1] {
  var j: int;
  coforall loc in Locales {
    on loc {
      i j Locales
    }
  }
}
```
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. For example:
            k = 2*i + j;
            // The implementation manages any communication.
        }
    }
}
```

Locales (think: “compute nodes”)
Chapel: Locality queries

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

            ...here...
            ...j.locale...
        }
    }
}
```

Locales (think: “compute nodes”)
Outline

✓ Motivation
✓ Chapel's Design Themes
➢ Survey of Chapel Concepts

● Project Status and Next Steps
Outline

- Motivation
- Chapel's Design Themes
  - Survey of Chapel Concepts

- Project Status and Next Steps
Data Parallelism By Example: STREAM Triad

```chapel
const ProblemSpace = {1..m};

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

A = B + alpha * C;
```
**STREAM Triad: Chapel (multicore)**

```chapel
const ProblemSpace = {1..m};

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

A = B + alpha * C;
```

No domain map specified ⇒ use default layout
- current locale owns all domain indices and array values
- computation will execute using local processors only
STREAM Triad: Chapel (multilocale, blocked)

```
const ProblemSpace = {1..m}
  dmapped Block(boundingBox={1..m});

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

A = B + alpha * C;
```
**STREAM Triad: Chapel (multilocal, cyclic)**

\[
\text{const ProblemSpace} = \{1..m\}\]

\[
\text{dmapped Cyclic(startIdx=1)};
\]

\[
\text{var A, B, C: [ProblemSpace] real};
\]

\[
A = B + \alpha \cdot C;
\]

\[
\alpha \cdot \]

\[
= \]

\[
+ \]

\[
= \]

\[
\text{startIdx} = 1\]
**STREAM Triad: Chapel**

```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;
```

**Philosophy:** Good, *top-down* language design can tease system-specific implementation details away from an algorithm, permitting the compiler, runtime, applied scientist, and HPC expert to each focus on their strengths.
Chapel has Many Types of Domains/Arrays

- dense
- strided
- sparse

- associative
- unstructured
LULESH: a DOE Proxy Application

**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

<table>
<thead>
<tr>
<th>Compute</th>
<th>Store</th>
<th>Analyze</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Copyright 2016 Cray Inc.
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 the only 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

Domain maps insulate the rest of the application from these choices
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’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
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 the domain map interface for implementers: http://chapel.cray.com/docs/latest/technotes/dsi.html
- Current domain maps:
  - $CHPL_HOME/modules/dists/*.chpl
  - layouts/*.chpl
  - internal/Default*.chpl
Two Other Thematically Similar Features

1) **parallel iterators:** Permit users to specify the parallelism and work decomposition used by forall loops
   - including zippered forall loops

2) **locale models:** Permit users to model the target architecture and how Chapel should be implemented on it
   - e.g., how to manage memory, create tasks, communicate, …

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, …
Language Summary

**HPC programmers deserve better programming models**

**Higher-level programming models can help insulate algorithms from parallel implementation details**
- yet, without necessarily abdicating control
- Chapel does this via its multiresolution design
  - domain maps, parallel iterators, and locale models are all examples
  - avoids locking crucial policy decisions into the language definition

**We believe Chapel can greatly improve productivity**
  …for current and emerging HPC architectures
  …for HPC users and mainstream uses of parallelism at scale
Outline

✓ Motivation
✓ Chapel's Design Themes
✓ Survey of Chapel Concepts
➢ Project Status and Next Steps
Chapel is Portable

- Chapel’s design is intended to be hardware-independent

- **The current release requires:**
  - a C/C++ compiler
  - a *NIX environment (Linux, OS X, BSD, Cygwin, …)
  - POSIX threads
  - (for distributed execution): support for RDMA, MPI, or UDP

- **Chapel can run on…**
  - laptops and workstations
  - commodity clusters
  - the cloud
  - HPC systems from Cray and other vendors
  - modern processors like Intel Xeon Phi, GPUs*, etc.

* = academic work only; not yet supported in the official release
Chapel is Open-Source

- Chapel’s development is hosted at GitHub
  - https://github.com/chapel-lang

- Chapel is licensed as Apache v2.0 software

- Instructions for download + install are online
  - see http://chapel.cray.com/download.html to get started
A Year in the Life of Chapel

- **Two major releases per year** (April / October)
  - ~a month later: detailed release notes

- **CHIUW: Chapel Implementers and Users Workshop** (May/June)
  - (3rd annual) [CHIUW 2016](#) will be held at IPDPS (Chicago, IL)

- **SC** (Nov)
  - tutorials, panels, BoFs, posters, educator sessions, exhibits, …
  - annual CHUG (Chapel Users Group) happy hour

- **Talks, tutorials, research visits, blog posts, …** (year-round)
The Chapel Team at Cray (Spring 2015)
Chapel is a Collaborative Effort

(and many others as well…)

http://chapel.cray.com/collaborations.html
Chapel is a Work-in-Progress

- Currently being picked up by early adopters
  - Users who try it generally like what they see
  - Last release got 1400+ downloads over six months

- Most features are functional and working well
  - some areas need improvements: strings, object-oriented features, …

- Performance can be hit-or-miss
  - shared memory performance is often competitive with C+OpenMP
  - distributed memory performance needs more work

- We are actively working to address these lacks
Stream-EP Performance Over Time

Stream EP Performance Across Chapel Releases
(128 nodes)

GB/s per node

Chapel version (six-months per release)

- MPI+OpenMP
- Chapel
Chapel Resources: For More Information
Chapel Websites

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

GitHub page: [https://github.com/chapel-lang](https://github.com/chapel-lang)
- download Chapel; browse source repository; contribute code

Facebook page: [https://www.facebook.com/ChapelLanguage](https://www.facebook.com/ChapelLanguage)
Suggested Chapel Reading

Chapel chapter from *Programming Models for Parallel Computing*

- published by MIT Press
- a detailed overview of Chapel’s history, motivating themes, features
- an early draft is available online, entitled *A Brief Overview of Chapel*

Other Chapel papers/publications available at [http://chapel.cray.com/papers.html](http://chapel.cray.com/papers.html)
Chapel Blog Articles

- a short-and-sweet introduction to Chapel

**Six Ways to Say “Hello” in Chapel** *(parts 1, 2, 3)*, Cray Blog, Sep-Oct 2015.
- a series of articles illustrating the basics of parallelism and locality in Chapel

**Why Chapel?** *(parts 1, 2, 3)*, Cray Blog, Jun-Oct 2014.
- a 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 argue against standard reasons given for not developing high-level parallel languages
Chapel Mailing Aliases

**read-only:**
- chapel-announce@lists.sourceforge.net: announcements about Chapel

**read/write:**
- chapel-users@lists.sourceforge.net: user-oriented discussion list
- chapel-developers@lists.sourceforge.net: developer discussions
- chapel-education@lists.sourceforge.net: educator discussions
- chapel-bugs@lists.sourceforge.net: public bug forum

**write-only:**
- chapel_info@cray.com: contact the team at Cray
- chapel_bugs@cray.com: for reporting non-public bugs

**Subscribe at SourceForge:** [http://sourceforge.net/p/chapel/mailman/](http://sourceforge.net/p/chapel/mailman/)
- (also serves as an alternate release download site to GitHub)
Questions?
Legal Disclaimer

Information in this document is provided in connection with Cray Inc. products. No license, express or implied, to any intellectual property rights is granted by this document.

Cray Inc. may make changes to specifications and product descriptions at any time, without notice.

All products, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

Cray hardware and software products may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Cray uses codenames internally to identify products that are in development and not yet publically announced for release. Customers and other third parties are not authorized by Cray Inc. to use codenames in advertising, promotion or marketing and any use of Cray Inc. internal codenames is at the sole risk of the user.

Performance tests and ratings are measured using specific systems and/or components and reflect the approximate performance of Cray Inc. products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.

The following are trademarks of Cray Inc. and are registered in the United States and other countries: CRAY and design, SONEXION, URIKA, and YARCDATA. The following are trademarks of Cray Inc.: ACE, APPRENTICE2, CHAPEL, CLUSTER CONNECT, CRAYPAT, CRAYPORT, ECOPHLEX, LIBSCI, NODEKARE, THREADSTORM. The following system family marks, and associated model number marks, are trademarks of Cray Inc.: CS, CX, XC, XE, XK, XMT, and XT. The registered trademark LINUX is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Other trademarks used in this document are the property of their respective owners.

Copyright 2016 Cray Inc.