Chapel:
Productive, Multiresolution Parallel Programming

Brad Chamberlain, Chapel Team, Cray Inc.
ATPESC 2016
August 3rd, 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 for Chapel

**Q:** Can a single language be...
   ...as productive as Python?
   ...as fast as Fortran?
   ...as portable as C?
   ...as scalable as MPI?
   ...as fun as <your favorite language here>?

**A:** We believe so.
Chapel: Putting the “Whee!” back in HPC

Brad Chamberlain, Chapel Team, Cray Inc.
ATPESC 2016
August 3rd, 2016
The Challenge

**Q:** So why don’t we have such languages already?

**A:** Technical challenges?

- while they exist, we don’t think this is the main issue…

**A:** Due to a lack of…

- long-term efforts
- resources
- community will
- co-design between developers and users
- patience

*Chapel is our attempt to reverse this trend*
Chapel: Putting the “We” back in HPC

Brad Chamberlain, Chapel Team, Cray Inc.
ATPESC 2016
August 3rd, 2016
What is Chapel?

**Chapel**: A productive 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 need, 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:**

- $A$
- $B$
- $C$
- $\alpha$

$A = B + \alpha \cdot C$
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):

\[
\begin{align*}
A &= + + + + =
\end{align*}
\]
\[
\begin{align*}
B &= + + + + =
\end{align*}
\]
\[
\begin{align*}
C &= + + + + =
\end{align*}
\]
\[
\begin{align*}
\alpha &= \cdot \cdot \cdot =
\end{align*}
\]
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 multicore):

![Diagram of STREAM Triad](image)
#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);
#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

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

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];
}
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 required.

<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/m 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
#define openmp
#include <omp.h>
#define VectorSize
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);
    if (doIO) {
        fprintf(outFile, "Failed to allocate memory (%d).\n", VectorSize );
        fclose(outFile );
    }
    return 1;
}
#endif
#pragma omp parallel for
for (j=0; j<VectorSize; j++) {
    b[j] = 2.0;
    c[j] = 0.0;
}
scalar = 3.0;
#endif
#pragma omp parallel for
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;

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

    __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**

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

cnst ProblemSpace = {1..m};

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.

---

**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_StarStream( 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 (%ld).\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;
```

---

**Chapel**

```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];
}
```
Outline

✓ Motivation

➢ Survey of Chapel Concepts
  ● Chapel Project and Characterizations
  ● Chapel Resources
Chapel’s Multiresolution Philosophy

**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
Base Language

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

Lower-level Chapel

Target Machine
Base Language Features, by example

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

```plaintext
config const n = 10;

for f in fib(n) do
    writeln(f);
```

0
1
1
2
3
5
8
...
Base Language Features, by example

CLU-style iterators

```plaintext
iter fib(n) {
    var current = 0,
         next = 1;

    for i in 1..n {
        yield current;
        current += next;
        current <=> next;
    }
}
```

```plaintext
config const n = 10;

for f in fib(n) do
    writeln(f);
```

0
1
1
2
3
5
8
...
Configuration declarations
(to avoid command-line argument parsing)
./a.out -n=1000000

iter fib(n) {
  var current = 0,
  next = 1;

  for i in 1..n {
    yield current;
    current += next;
    current <=> next;
  }
}

config const n = 10;

for f in fib(n) do
  writeln(f);

0
1
1
2
3
5
8...

Base Language Features, by example
Base Language Features, by example

```plaintext
config const n = 10;
for f in fib(n) do writeln(f);
```

```plaintext
iter fib(n) {
 var current = 0,
   next = 1;
   for i in 1..n {
     yield current;
     current += next;
     current <=> next;
   }
}
```
Base Language Features, by example

```plaintext
iter fib(n) {
  var current = 0,
     next  = 1;

  for i in 1..n {
    yield current;
    current += next;
    current <=> next;
  }
}
```

```plaintext
config const n = 10;

for (i,f) in zip(0..#n, fib(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
...
```
Base Language Features, by example

iter fib(n) {
    var current = 0,
    next = 1;
    for i in 1..n {
        yield current;
        current += next;
        current <=> next;
    }
}

config const n = 10;
for (i, f) in zip(0..#n, fib(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
...
Base Language Features, by example

```c
iter fib(n) {
    var current = 0,
        next = 1;
    for i in 1..n {
        yield current;
        current += next;
        current <=> next;
    }
}
```

```c
config const n = 10;
for (i,f) in zip(0..#n, fib(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
...
Base Language Features, by example

```
iter fib(n) {
    var current = 0,
    next = 1;

    for i in 1..n {
        yield current;
        current += next;
        current <=> next;
    }
}
```

```
config const n = 10;

for (i,f) in zip(0..#n, fib(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

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

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

// 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: Co forall Loops

```c
// 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

- **atomic variables**: support atomic operations
  - e.g., compare-and-swap; atomic sum, multiply, etc.
  - similar to C/C++

- **sync variables**: store full-empty state along with value
  - by default, reads/writes block until full/empty, leave in opposite state
Other Task Parallel Concepts

- **cobegins**: create tasks using compound statements
- **single variables**: like sync variables, but write-once
- **sync statements**: join unstructured tasks
- **serial statements**: conditionally squash parallelism
Locality Control
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

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

- Chapel provides built-in locale variables

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

- `main()` starts execution as a task on locale #0
Locale Operations

● Locale methods support queries about the target system:

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

● **On-clauses** support placement of computations:

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

on Locales[1] do
  writeln("now on locale 1");
writeln("on locale 0 again");

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
coforall i in 1..msgs do
  writeln("Hello from task ", i);
  ```

- This is a **distributed**, but serial program:
  ```chapel
writeln("Hello from locale 0!");
on Locales[1] do writeln("Hello from locale 1!");
on Locales[2] do writeln("Hello from locale 2!");
  ```

- This is a **distributed parallel** program:
  ```chapel
coforall i in 1..msgs do
  on Locales[i%numLocales] do
  writeln("Hello from task ", i,
  " running on locale ", here.id);
  ```
Chapel: Scoping and Locality

var i: int;

Locales (think: "compute nodes")
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”)

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 {
```

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

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

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

```chapel
var i: int;
on Locales[1] {
    var j: int;
    coforall loc in Locales {
        on loc {
            var k: int;
            k = 2*i + j;
        }
    }
}
```

OK to access `i`, `j`, and `k` wherever they live

```
k = 2*i + j;
```

**Locales** (think: “compute nodes”)
var i: int;
on Locales[1] {
  var j: int;
  coforall loc in Locales {
    on loc {
      var k: int;
      k = 2*i + j;
    }
  }
}

here, i and j are remote, so the compiler + runtime will transfer their values

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

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

            ...here...  // query the locale on which this task is running
            ...j.locale...  // query the locale on which j is stored
        }
    }
}
```

Locales (think: “compute nodes”)

Reasoning about Communication

- Though implicit, users can reason about communication
  - semantic model is explicit about where data is placed / tasks execute
  - execution-time queries support reasoning about locality
    - e.g., `here, x.locale`
  - tools should also play a role here
    - e.g., `chplvis`, contained in the release (developed by Phil Nelson, WWU)
Data Parallelism

Higher-level Chapel

- Domain Maps
- Data Parallelism
- Task Parallelism
- Base Language
- Locality Control
- Target Machine
const ProblemSpace = {1..m};

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

forall (a,b,c) in zip(A,B,C) do
  a = b + alpha*c;
**Data Parallelism By Example: STREAM Triad**

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

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

A = B + alpha * C;  // equivalent to the zippered forall
```

// Diagram/Visual Representation

---

Copyright 2016 Cray Inc.
Other Data Parallel Features

- **Rich Domain/Array Types:**
  - multidimensional
  - strided
  - sparse
  - associative

- **Slicing:** Refer to subarrays using ranges/domains
  - \( A[2..n-1, lo..#b] \)
  - \( A[\text{ElementsOfInterest}] \)

- **Promotion:** Call scalar functions with array arguments
  - \( \text{pow}(A, B) \)  // equivalent to: \( \text{forall} \ (a,b) \in \text{zip}(A,B) \ do \ \text{pow}(a,b) \)

- **Reductions/Scans:** Apply operations across collections
  - \( + \ \text{reduce} \ A \)
  - \( \text{myReduceOp} \ \text{reduce} \ A \)
Domain Maps

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

Higher-level 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, cyclic)

\[
\text{const ProblemSpace} = \{1..m\}, \text{dmapped Cyclic(startIdx=1)};
\]

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

\[
\alpha \cdot \text{startIdx = 1}
\]

\[
A = B + \alpha \cdot C;
\]
STREAM Triad: Chapel (multilocale, blocked)

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

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

A = B + alpha * C;
```

Copyright 2016 Cray Inc.
The Chapel code snippet for the STREAM Triad problem demonstrates a top-down language design that 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. The Chapel code includes a configuration block with constants and variables, followed by a CUDA device code that sets up arrays and performs operations on them. The Chapel code snippet is written in a way that abstracts away the complexities of the underlying system, allowing for more focus on the algorithm itself rather than the details of the system implementation. The philosophy behind Chapel is that good, top-down language design can facilitate this separation, enabling each component to specialize in its particular expertise.
Chapel Has Several Domain/Array Types

- dense
- strided
- sparse
- associative
- unstructured

All Chapel domain types support domain maps.
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

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 the Chapel release in examples/benchmarks/lulesh/
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 ("the science") 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
Two Other Thematically Similar Features

1) **parallel iterators:** Permit users to specify forall-loop policies
   ● e.g., parallelism, work decomposition, and locality
   ● including zippered forall loops

2) **locale models:** Permit users to target new architectures
   ● e.g., how to manage memory, create tasks, communicate, …

Like domain maps, these are…
   …written in Chapel by expert users
   …exposed to the end-user via higher-level abstractions
Chapel is Extensible

Advanced users can create their own…

...parallel loop schedules...
...array layouts and distributions...
...models of the target architecture...

...as Chapel code, without modifying the compiler.

Why? To create a future-proof language.

This has been our main research challenge: How to create a language that does not lock these policies into the implementation without sacrificing performance?
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

✓ Survey of Chapel Concepts

➢ Chapel Project and Characterizations

● Chapel Resources
A Year in the Life of Chapel

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

- **CHIUW: Chapel Implementers and Users Workshop** (May/June)
  - held three years so far, typically at IPDPS
  - CHIUW 2017 proposal being submitted this week

- **SC** (Nov)
  - tutorials, BoFs, panels, posters, educator sessions, exhibits, …
  - annual [CHUG](#) (Chapel Users Group) happy hour
  - for **SC16**:
    - full-day Chapel tutorial (Sunday)
    - Chapel Lightning Talks BoF proposal submitted
    - likely to be additional events as well…

- **Talks, tutorials, collaborations, social media, …** (year-round)
Chapel is Portable

● Chapel is designed to be hardware-independent

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

● 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](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](http://chapel.cray.com/download.html) to get started
14 full-time employees + 2 summer interns + 1 contracting professor (one of each started after this photo was taken)
Chapel is a Collaborative Effort

(and several others…)

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

- **Currently being picked up by early adopters**
  - Last two releases got ~3500 downloads total in a year
  - Users who try it generally like what they see

- **Most current features are functional and working well**
  - some areas need improvements, particularly object-oriented features

- **Performance is improving, but remains hit-or-miss**
  - shared memory performance is often competitive with C+OpenMP
  - distributed memory performance continues to need more work

- **We are actively working to address these lacks**
A notable early adopter

Chapel in the (Cosmological) Wild

Nikhil Padmanabhan, Yale University Professor, Physics & Astronomy

Abstract: This talk aims to present my personal experiences using Chapel in my research. My research interests are in observational cosmology; more specifically, I use large surveys of galaxies to constrain the evolution of the Universe and to probe the physics underlying that evolution. Operationally, this involves measuring a number of spatial statistics of the distribution of galaxies, both on actual observations, but also on large numbers of simulated universes.

I'll start by presenting a whirlwind introduction to cosmology, the problems that keep me up at night and our approaches to solving these. I'll then discuss what attracted me to Chapel—the ability to prototype algorithms quickly and the promised ease and flexibility of writing parallel programs. I'll then present a worked example of Chapel being used in a real-world application, discussing some of these aspects as well highlighting its interoperability with existing libraries, as well as some of the challenges. I'll conclude with what it would take for me to switch over to using Chapel all of the time.
Chapel is a Work-in-Progress

- Currently being picked up by early adopters
  - Last two releases got ~3500 downloads total in a year
  - Users who try it generally like what they see

- Most current features are functional and working well
  - Some areas need improvements, particularly object-oriented features

- Performance is improving, but can be hit-or-miss
  - Shared memory performance is often competitive with C+OpenMP
  - Distributed memory performance continues to need more work

- We are actively working to address these lacks
Chapel’s 5-year push

- Based on positive user response to Chapel in its research phase, Cray undertook a five-year effort to improve it
  - we’ve just completed our third year

- **Focus Areas:**
  1. Improving **performance** and scaling
  2. **Fixing** immature aspects of the language and implementation
     - e.g., strings, memory management, error handling, …
  3. **Porting** to emerging architectures
     - Intel Xeon Phi, accelerators, heterogeneous processors and memories, …
  4. Improving **interoperability**
  5. Growing the Chapel user and developer **community**
     - including non-scientific computing communities
  6. Exploring transition of Chapel **governance** to a neutral, external body
Outline

✓ Motivation

✓ Survey of Chapel Concepts

✓ Chapel Project and Characterizations

➢ Chapel Resources
Chapel Websites

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

GitHub: https://github.com/chapel-lang
- download Chapel; browse source repository; contribute code

Facebook: https://www.facebook.com/ChapelLanguage

Twitter: https://twitter.com/ChapelLanguage
Suggested Reading

Chapel chapter from *Programming Models for Parallel Computing*

- a detailed overview of Chapel’s history, motivating themes, features
- published by MIT Press, November 2015
- edited by Pavan Balaji (Argonne)
- chapter is now also available online

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

**Chapel Springs into a Summer of Code**, Cray Blog, April 2016.
- coverage of recent events

**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 Lists

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

**community lists:**
- 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

(subscribe at SourceForge: [http://sourceforge.net/p/chapel/mailman/](http://sourceforge.net/p/chapel/mailman/))

**To contact the Cray team:**
- chapel_info@cray.com: contact the team at Cray
- chapel_bugs@cray.com: for reporting non-public bugs
Chapel: Productive, Multiresolution Parallel Programming

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, and URIKA. 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.