Chapel
Overview and Status

Brad Chamberlain, Chapel Team, Cray Inc.
Intel Extreme Scale Technical Review Meeting
November 11th, 2014
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.
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 the STREAM Triad computation](image)
STREAM Triad: a trivial parallel computation

Given: \( m \)-element vectors \( A, B, C \)

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

In pictures, in parallel (distributed memory multicore):
STREAM Triad: MPI

```c
#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>
#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

Copyright 2014 Cray Inc.
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/m multicore</td>
<td>OpenMP/pthreads</td>
<td>iteration/task</td>
</tr>
<tr>
<td>Instruction-level v/t</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>

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

### 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);
    if (N % dimBlock.x != 0) dimBlock.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);
}
```

### 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;
}

__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];
}

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

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

HPC suffers from too many distinct notations for expressing parallelism and locality
STREAM Triad: Chapel

**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

✓ Motivation

➢ Chapel Background and Themes

● Survey of Chapel Concepts
  ● Quick run-through of basic concepts
  ● Slightly more detail on advanced/research-y concepts

● Project Status and Next Steps
  ● including “current events” section
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**

- **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.”
Chapel's Implementation

- **Being developed as open source at GitHub**
  - Licensed as Apache v2.0 software

- **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, …
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
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>
3) Multiresolution Design: Motivation

"Why is everything so tedious/difficult?"
"Why 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
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

✓ Motivation
✓ Chapel Background and Themes
➢ Survey of Chapel Concepts
● Project Status and Next Steps
Lower-Level Features

Chapel language concepts

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

Lower-level Chapel

Target Machine
Base Language Features
Sample Base Language Features

Static Type Inference for:
- arguments
- return types
- variables

CLU-style iterators

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

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

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
...

swap operator

range types and operators

zippered iteration
Task Parallel Features
Task Parallelism: Begin Statements

```plaintext
begin writeln(“hello world”); // create a fire-and-forget task
writeln(“goodbye”);
```

Possible outputs:

- hello world
- goodbye
- goodbye
- hello world
Sample Task Parallel Feature: Coforall Loops

```chapel
coforall t in 0..#numTasks { // create a task per iteration
    writeln("Hello from task ", t, " of ", numTasks);
} // wait for its tasks to complete before proceeding

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) \textit{atomic variables}: support atomic operations (as in C++)
   - e.g., compare-and-swap; atomic sum, mult, etc.

2) \textit{single-assignment variables}: reads block until assigned

3) \textit{synchronization variables}: store full/empty state
   - by default, reads/writes block until the state is full/empty
Locality Features

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

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
  
  ```
  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);
```
Chapel and PGAS

- Chapel is a PGAS language...
  - ...but unlike most, it’s not restricted to SPMD
    - never think in terms of “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”)
Chapel: Scoping and Locality

```plaintext
var i : int;
```

Locales (think: “compute nodes”)

C O M P U T E      |     S T O R E      |     A N A L Y Z E
Chapel: Scoping and Locality

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

Locales (think: “compute nodes”)
var i: int;
on Loci[i] {
  var j: int;

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

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

![Diagram of Locales with indices i and j]
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
Higher-Level Features

Chapel language concepts

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

Higher-level Chapel

Copyright 2014 Cray Inc.
Higher-Level Features

Chapel language concepts

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

Target Machine

Higher-level Chapel

Theme 2: Global-view Abstractions
Data Parallel Features (by example)
STREAM Triad: Chapel

### Chapel

```chapel
cfg 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 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.
STREAM Triad in Chapel

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

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

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

```
const ProblemSpace = {1..m}
    dmapped Cyclic(startIdx=1);

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

A = B + alpha * C;
```
Data Parallelism by Example: Jacobi Iteration

\[ \sum \left( \begin{array}{cccc} 1.0 \\ \frac{1}{4} \end{array} \right) \div 4 \]

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

```
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

```
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

.dom().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]...")

A
Jacobi Iteration in Chapel

```
config const n = 6,

Compute 5-point stencil
forall ind in Dom \Rightarrow parallel forall expression over Dom's indices, binding them to ind
(here, since Dom is 2D, we can de-tuple the indices)

\sum \begin{bmatrix}
\text{Top Left} & \text{Middle Left} \\
\text{Bottom Left} & \text{Center} & \text{Bottom Right} \\
\text{Top Right} & \text{Middle Right} \\
\end{bmatrix} \div 4  \rightarrow \begin{bmatrix}
\text{Bottom Left} & \text{Bottom Right} \\
\text{Top Right} & \text{Middle Right} & \text{Center} & \text{Top Left} \\
\end{bmatrix}
```

```
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;

Compute maximum change

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

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

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

```chapel
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);
```

Copy data back & Repeat until done

uses slicing and whole array assignment
standard `do...while` loop construct
Jacobi Iteration in 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
Jacobi Iteration in Chapel (shared memory)

```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 {
    foreach (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 parallel program.
Jacobi Iteration in Chapel (distributed memory)

```
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;
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
LULESH in Chapel

Here is some sample representation-independent code

IntegrateStressForElems()

LULESH spec, section 1.5.1.1 (2.)
Representation-Independent Physics

```chapel
proc IntegrateStressForElems(sigxx, sigyy, sigzz, determ) {
    forall k in Elems {
        var b_x, b_y, b_z: 8*real;
        var x_local, y_local, z_local: 8*real;
        localizeNeighborNodes(k, x, x_local, y, y_local, z, z_local);

        var fx_local, fy_local, fz_local: 8*real;

        local {
            /* Volume calculation involves extra work for numerical consistency. */
            CalcElemShapeFunctionDerivatives(x_local, y_local, z_local,
                b_x, b_y, b_z, determ[k]);

            CalcElemNodeNormals(b_x, b_y, b_z, x_local, y_local, z_local);

            SumElemStressesToNodeForces(b_x, b_y, b_z, sigxx[k], sigyy[k], sigzz[k],
                fx_local, fy_local, fz_local);
        }

        for (noi, t) in elemToNodesTuple(k) {
            fx[noi].add(fx_local[t]);
            fy[noi].add(fy_local[t]);
            fz[noi].add(fz_local[t]);
        }
    }
}
```

Because of domain maps, this code is independent of:
- structured vs. unstructured mesh
- shared vs. distributed data
- sparse vs. dense representation
Domain Maps

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

- “steve”
- lee
- sung
- “david”
- jacob
- albert
- “brad”

**unstructured**
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

\[
\text{const Elems = } \{0..\#numElems\} \text{ dmapped Block(...)}
\]
Two Other Thematically Similar Features

1) **leader-follower 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, …
Implementing Data Parallel Loops

Q: How are parallel loops implemented?
   (how many tasks? executing where? how are iterations divided up?)

```plaintext
forall k in Elems { ... }
```

Q2: What about zippered data parallel operations?
   (how to reconcile potentially conflicting parallel implementations?)

```plaintext
forall (k,d) in zip(Elems, determ) { ... }
x += xd * dt;
```

A: Via Feature #2 (leader-follower iterators)…
Leader-Follower Iterators: Definition

● Chapel defines forall loops using leader-follower iterators:
  ● leader iterators: create parallelism, assign iterations to tasks
  ● follower iterators: serially execute work generated by leader

● Given…

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

...A is defined to be the leader

...A, B, and C are all defined to be followers

● Domain maps support default leader-follower iterators
  ● specify parallel traversal of a domain’s indices/array’s elements
  ● typically written to leverage affinity
Writing Leaders and Followers

Leader iterators are defined using task/locality features:

```chapel
iter BlockArr.lead() {
    coforall loc in Locales do
        on loc do
            coforall tid in here.numCores do
                yield computeMyChunk(loc.id, tid);
}
```

Follower iterators simply use serial features:

```chapel
iter BlockArr.follow(work) {
    for i in work do
        yield accessElement(i);
}
```
Leader-Follower Summary

● **Data locality requires parallel loops to execute intelligently**
  ● appropriate number and placement of tasks
  ● good data-task affinity

● **Most languages define fixed parallel loop styles**
  ● where “no parallel loops” is a common choice

● **Leader-follower iterators...**
  ...move such policies into user-space
  ...expose them to the end-user through data parallel abstractions

\[
\text{forall } k \text{ in } \text{Elems} \{ \ldots \}
\]
\[
x += xd \times dt;
\]
Prototypical Next-Gen Processor Technologies

Intel Phi

AMD APU

Nvidia Echelon

Tilera Tile-Gx

Sources:
http://download.intel.com/pressroom/images/Aubrey_Isle_die.jpg
http://www.zdnet.com/amds-trinity-processors-take-on-intels-ivy-bridge-3040155225/
http://tilera.com/sites/default/files/productbriefs/Tile-Gx%203036%20SB012-01.pdf
Architectural Trends

Emerging compute node designs…

…the increasingly locality-sensitive

…potentially have multiple processor/memory types
Traditional Locales

Concept:

- Traditionally, Chapel has supported a 1D array of locales

- Supports inter-node locality well, but not intra-node
  - (which, of course, is becoming increasingly important)
Recent Work: Hierarchical Locales

Concept:

- Support locales within locales to describe architectural sub-structures within a node (e.g., memories, processors)

- As with top-level locales, on-clauses and domain maps map tasks and variables to sub-locales

- Locale models are defined using Chapel code
Defining Hierarchical Locales

1) Define the processor’s abstract block structure

2) Define how to run a task on any sublocale

3) Define how to allocate/access memory on any sublocale
Hierarchical Locale Summary

- Data locality requires flexibility w.r.t. future architectures
  - due to uncertainty in processor design
  - to support portability between approaches

- Most programming models assume certain features in the target architecture
  - this is why MPI/OpenMP/UPC/CUDA/… have restricted applicability

- Hierarchical Locales
  …move the definition of new architectural models to user space
  …are exposed to the end-user via Chapel’s traditional locality features

```
on loc do
  coforall tid in here.numCores do
```
Multiresolution Summary

Chapel’s multiresolution philosophy allows users to write…

…custom array implementations via domain maps

…custom parallel iterators via leader-follower 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


*Authoring User-Defined Domain Maps in Chapel* [slides], Chamberlain, Choi, Deitz, Iten, Litvinov; Cug 2011, May 2011.

...leader-follower 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)
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
  - leader-follower 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
Outline

✓ Motivation
✓ Chapel Background and Themes
✓ Survey of Chapel Concepts
➢ Project Status and Next Steps
Major Chapel Successes Under HPCS

SSCA#2 demonstration on the prototype Cray XC30
- unstructured graph compact application
- clean separation of computation from data structure choices
- fine-grain latency-hiding runtime
- use of XC30’s network AMOs via Chapel’s ‘atomic’ types

Clean, general parallel language design
- unified data-, task-, concurrent-, nested-parallelism
- distinct concepts for parallelism and locality
- multiresolution language design philosophy

Portable design and implementation
- while still being able to take advantage of Cray-specific features

Revitalization of Community Interest in Parallel Languages
- HPF-disenchantment became interest, cautious optimism, enthusiasm
Implementation Status -- Version 1.10.0 (Oct 2014)

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

This is a good time to:
- Try out the language and compiler
- Use Chapel for non-performance-critical projects
- Give us feedback to improve Chapel (or contribute code)
- Use Chapel for parallel programming education
A Year in the Life of Chapel

- **Two major releases per year** (April / October)
  - latest release: version 1.10, October 2\textsuperscript{nd}, 2014
  - \~a month later: detailed release notes
    - version 1.10 release notes: [http://chapel.cray.com/download.html#releaseNotes](http://chapel.cray.com/download.html#releaseNotes)

- **CHIUW**: Chapel Implementers and Users Workshop (May-June)
  - talk-based workshop focusing on community efforts

- **SC** (Nov)
  - Chapel tutorials (most years)
  - lightning talks BoF (now in its 4\textsuperscript{th} year)
  - CHUG meet-up / happy hour (now in its 5\textsuperscript{th} year)
  - emerging technology exhibit (now in its 2\textsuperscript{nd} year)
  - educators forum (past two years)
  - talks, posters, panels, etc. (whenever possible)

- **Talks, tutorials, research visits, blogs, ...** (year-round)
CHIUW 2014 Talks and Speakers

User Experiences with a Chapel Implementation of UTS
Jens Breitbart, Technische Universität München

Evaluating Next Generation PGAS Languages for Computational Chemistry
Daniel Chavarria-Miranda, Pacific Northwest National Laboratory

Programmer-Guided Reliability in Chapel
David E. Bernholdt, Oak Ridge National Laboratory

Towards Interfaces for Chapel
Chris Wailes, Indiana University

Affine Loop Optimization using Modulo Unrolling in Chapel
Aroon Sharma, University of Maryland

Keynote: Walking to the Chapel
Robert Harrison, Stony Brook University / Brookhaven National Laboratory

LLVM Optimizations for PGAS Programs
Akihiro Hayashi, Rice University

Opportunities for Integrating Tasking and Communication Layers
Dylan T. Stark, Sandia National Laboratories

Caching in on Aggregation
Michael Ferguson, Laboratory for Telecommunication Sciences
Chapel at SC14

Chapel Tutorial (Sun @ 8:30)
  ● A Computation-Driven Introduction to Parallel Computing in Chapel

Emerging Technologies Booth on Show floor (all week)
  ● Booth #233: Chapel Hierarchical Locales poster, staffed by Chapel team

Chapel Lightning Talks BoF (Tues @ 12:15)
  ● 5-minute talks on Chapel + HSA, HDFS/Lustre/cURL, tilings, LLVM, ExMatEx, Python

Chapel Hierarchical Locales Talk (Tues @ 4:30)
  ● 30-minute talk describing the use of hierarchical locales in Chapel

Poster (Tues @ 5:15)
  ● Ian Bertolacci (Colorado State University) on advanced tilings in Chapel

Chapel Users Group BoF (Wed @ 5:30)
  ● Chapel intro + current events followed by community discussion

Happy Hour (Wed @ 7:15): 5th annual Chapel Users Group (CHUG) Happy Hour
  ● at Mulate’s (201 Julia St, just across the way): open to public, dutch treat

Participation in other BoFs:
  ● LLVM in HPC (Tues @ 12:15)
  ● Programming Abstractions for Data Locality (Wed @ 12:15)
  ● PGAS: Partitioned Address Space Programming Model (Wed @ 12:15)
Chapel Current Events
Chapel is Alive and Well

- Based on positive customer response to Chapel under HPCS, Cray is undertaking a five-year effort to improve it
  - we’re currently partway into our second year

- Focus Areas:
  1. Improving performance and scaling
  2. Fixing immature aspects of the language and implementation
     - e.g., strings, RAII/memory model, error handling, …
  3. Porting to emerging architectures
     - Intel Phi, accelerators, heterogeneous processors and memories, …
  4. Improving interoperability
  5. Growing the Chapel user and developer community
     - including non-HPC communities
  6. Transitioning the governance to neutral, external group
The Cray Chapel Team has doubled in size
The Broader Chapel Community is also Growing

(there’s been an uptick in interest from industrial users/developers as well)

http://chapel.cray.com/collaborations.html
Chapel version 1.10 is now available

**Highlights Include:**

- lighter-weight tasking via Sandia’s Qthreads
- initial support for Intel Xeon Phil Knights Corner (KNC)
- renewed focus on standard libraries
- support for Lustre and cURL-based data channels
- expanded array capabilities
- improved semantic checks, bug fixes, third-party packages, …
- significant performance improvements…
Execution Time is Improving (lower is better)
Nightly Performance Graphs are Now Public

- What this means:
  - You can stalk our performance changes over time
  - You can submit your own performance tests and monitor them
  - You can see the performance impacts of patches you commit

http://chapel.sourceforge.net/perf/
Chapel Language Shootout* Entry Underway

* = ahem… The Computer Language Benchmarks Game
http://benchmarksgame.alioth.debian.org/
Shootout entry improving with each release

X worse than reference

Chapel 1.8 release
Chapel 1.9 release
Chapel 1.10 release
Multilocale Performance is also improving.

- **STREAM (GB/s)**
  - EP, ugni+muxed
  - Global, ugni+muxed

- **RA, atomic version (GUPS)**
  - ugni+muxed

- **HPL, study version (GF/s)**
  - ugni+muxed
  - gasnet+mpi
  - gasnet+aries

- **FFT (GF/s)**
  - ugni+muxed
  - gasnet+mpi
  - gasnet+aries
Scalability is Improving as well

Performance of STREAM
(ugni+muxed)

GB/s

Locales

Global (1.9)  
EP (1.9)  
Global (1.10)  
EP (1.10)  

Cray Proprietary -- Copyright 2014 Cray Inc.
Memory Leaks are Being Plugged
Correctness Testing Now Public Too

- Nightly regression tests sent to SourceForge mailing lists:
  - chapel-test-results-regressions@
  - chapel-test-results-all@

http://sourceforge.net/p/chapel/mailman/
Chapel Developers Join 21st Century

- Migrated from SVN/SourceForge to Git/GitHub
- Converted testing from crontabs to Jenkins
- Began using Travis for pre-commit sanity checks
- Began using Coverity scan to catch code quality issues
- Started tracking tasks in Pivotal
- Kicked off a Facebook page
- Started a #chapel-developers IRC channel
- Created/Owned a Chapel project in OpenHUB

Next up: Modern, online documentation and more…
Chapel is now Apache

Historically:

- License: BSD
- Contributor Agreement: Cray-specific

As of version 1.10:

- License: Apache v2.0
- Contributor Agreement: Apache v2.0

Rationale:

- BSD doesn’t have a contributor agreement
- Cray agreement has been a stumbling block for some developers

http://www.apache.org/licenses/LICENSE-2.0.html
Chapel: It’s not just for HPC anymore

● “Big data” programmers want productive languages too
  ● MapReduce, Pig, Hive, HBase have their place, but also drawbacks
  ● Wouldn’t a general, locality-aware parallel language be nice here too?

● Chapel support for HDFS*: A first step
  ● Developed by Tim Zakian (Indiana University), summer 2013
  ● Summer 2014: extended support to include Lustre, cURL

● Questions:
  ● What killer apps/demos to focus on?

*HDFS = Hadoop Distributed File System

For some time, we’ve claimed Chapel is ideal for education:

For 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: Attractive for Education

And now, educators are helping make the argument for us:

http://chapel.cray.com/education.html
Interactive Chapel

- What if you could work with Chapel interactively:
  ```chapel
  chpl> var A: [1..n] real;
  OK.
  chpl> [i in 1..n] A = i / 2.0;
  OK.
  chpl> writeln(A);
  0.5 1.0 1.5 2.0 2.5 3.0
  chpl> proc foo(x) { x *= 2; }
  OK.
  ```

- What if this worked not only on your desktop, but by offloading onto compute nodes as well:
  ```chapel
  chpl> var myLocales = getNodes(100);
  OK.
  chpl> var MyDist = new Block({1..1000000}, myLocales);
  OK.
  ```

- We’ve recently started an effort to implement such a capability
Working toward “The Chapel Foundation”

● If Chapel remains Cray-controlled, its chances of succeeding are much lower

● The intention has always been to “turn it over to the community” when it’s ready
  ● finding the correct timing is tricky

● We’ve started the brainstorming process of what such a model would look like (“The Chapel Foundation”)
  ● membership roles
  ● governance
  ● funding models

● If you have thoughts on this, we’re interested in them
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 1.10.0 release, browse source repository

**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, 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 2014 Cray Inc.