The Chapel Parallel Programming Language

Brad Chamberlain, Chapel Team, Cray Inc.  
Pacific Northwest Numerical Analysis Seminar  
October 19th, 2013
The Chapel Parallel Programming Language

Brad Chamberlain, Chapel Team, Cray Inc.
Pacific Northwest Numerical Analysis Seminar
October 19th, 2013
Chapel: *The* Parallel Programming Language

Brad Chamberlain, Chapel Team, Cray Inc.
Pacific Northwest Numerical Analysis Seminar
October 19th, 2013
Chapel: The Parallel Programming Language of the Future!

Brad Chamberlain, Chapel Team, Cray Inc.
Pacific Northwest Numerical Analysis Seminar
October 19th, 2013
Chapel: The Parallel Programming Language of the Future! (?)

Brad Chamberlain, Chapel Team, Cray Inc.
Pacific Northwest Numerical Analysis Analysis Seminar
October 19th, 2013
Kanye West’s 1st Solo Tour in 5 Years
(kicks off in Seattle tonight)

Chapel: “That $#!^’s Cray”
My Employer: CRAY
THE SUPERCOMPUTER COMPANY
Titan (ORNL)

- compute nodes: 18,688
- processors: 16-core AMD/node = 299,008 cores
- GPUs: 18,688 NVIDIA Tesla K20s
- memory: 32 + 6 GB/node = 710 TB total
- peak speed: 20+ petaflops
- floorspace: 4,352 square feet

For more information: [http://www.olcf.ornl.gov/titan/](http://www.olcf.ornl.gov/titan/)
Blue Waters

Blue Waters (NCSA)
- compute nodes: 25,712
- processors: 386,816 AMD cores
- GPUs: 3,072 NVIDIA Kepler GPUs
- memory: 1.476 PB total
- peak speed: 11.61 petaflops

https://bluewaters.ncsa.illinois.edu/
Sustained Performance Milestones

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

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

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

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

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

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

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

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

Intel MIC

AMD Trinity

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%202012-01.pdf
General Characteristics of These Architectures

- Increased hierarchy and/or sensitivity to locality
- Potentially heterogeneous processor/memory types

⇒ Next-gen programmers will have a lot more to think about at the node level than in the past
Sustained Performance Milestones

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

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

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

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

Or, perhaps something completely different?
STREAM Triad: a trivial parallel computation

**Given:** $m$-element vectors $A$, $B$, $C$

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

In pictures:
STREAM Triad: a trivial parallel computation

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

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

In pictures, in parallel:
STREAM Triad: a trivial parallel computation

**Given:** $m$-element vectors $A$, $B$, $C$

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

In pictures, in parallel (distributed memory):
STREAM Triad: a trivial parallel computation

**Given:** $m$-element vectors $A$, $B$, $C$

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

In pictures, in parallel (distributed memory multicore):
#include <hpcc.h>

static int VectorSize;
static double *a, *b, *c;

int HPCC_StarStream(HPCC_Params *params) {
    int myRank, commSize;
    int rv, errCount;
    MPI_Comm comm = MPI_COMM_WORLD;

    MPI_Comm_size( comm, &commSize );
    MPI_Comm_rank( comm, &myRank );

    rv = HPCC_Stream( params, 0 == myRank);
    MPI_Reduce( &rv, &errCount, 1, MPI_INT, MPI_SUM, 0, comm);

    return errCount;
}

int HPCC_Stream(HPCC_Params *params, int doIO) {
    register int j;
    double scalar;

    VectorSize = HPCC_LocalVectorSize( params, 3, sizeof(double), 0 );

    a = HPCC_XMALLOC( double, VectorSize );
    b = HPCC_XMALLOC( double, VectorSize );
    c = HPCC_XMALLOC( double, VectorSize );

    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>
#ifdef _OPENMP
#include <omp.h>
#endif

static int VectorSize;
static double *a, *b, *c;

int HPCC_StarStream(HPCC_Params *params) {
    int myRank, commSize;
    int rv, errCount;
    MPI_Comm comm = MPI_COMM_WORLD;

    MPI_Comm_size( comm, &commSize );
    MPI_Comm_rank( comm, &myRank );

    rv = HPCC_Stream( params, 0 == myRank);
    MPI_Reduce( &rv, &errCount, 1, MPI_INT, MPI_SUM, 0, comm );

    return errCount;
}

int HPCC_Stream(HPCC_Params *params, int doIO) {
    register int j;
    double scalar;

    VectorSize = HPCC_LocalVectorSize( params, 3, sizeof(double), 0 );

    a = HPCC_XMALLOC( double, VectorSize );
    b = HPCC_XMALLOC( double, VectorSize );
    c = HPCC_XMALLOC( double, VectorSize );

    if (!a || !b || !c) {
        if (c) HPCC_free(c);
        if (b) HPCC_free(b);
        if (a) HPCC_free(a);
        if (doIO) {
            fprintf( outFile, "Failed to allocate memory (%d).\n", VectorSize );
            fclose( outFile );
        }
        return 1;
    }

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

    scalar = 3.0;

    #ifdef _OPENMP
    #pragma omp parallel for
    #endif
    for (j=0; j<VectorSize; j++)
        a[j] = b[j]+scalar*c[j];

    HPCC_free(c);
    HPCC_free(b);
    HPCC_free(a);
}
```
**STREAM Triad: MPI+OpenMP vs. CUDA**

**MPI + OpenMP**

```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;
}
int HPCC_Stream(HPCC_Params *params, int doIO) {
    int j;
    double scalar;
    VectorSize = HPCC_LocalVectorSize( params, 3, sizeof(double), 0 );
    a = HPCC_XMALLOC( double, VectorSize );
    b = HPCC_XMALLOC( double, VectorSize );
    c = HPCC_XMALLOC( double, VectorSize );
    if (!a || !b || !c) {
        if (c) HPCC_free(c);
        if (b) HPCC_free(b);
        if (a) HPCC_free(a);
        if (doIO) {
            fprintf( outFile, "Failed to allocate memory (%d).\n", VectorSize );
            fclose( outFile );
        }
        return 1;
    }
    #ifdef _OPENMP
    #pragma omp parallel for
    #endif
    for (j=0; j<VectorSize; j++) {
        b[j] = 2.0;
        c[j] = 0.0;
    }
    scalar = 3.0;
    #ifdef _OPENMP
    #pragma omp parallel for
    #endif
    for (j=0; j<VectorSize; j++)
        a[j] = b[j]+scalar*c[j];
    HPCC_free(c);
    HPCC_free(b);
    HPCC_free(a);
    return 0;
}
```

**CUDA**

```c
#define N 2000000

tvoid main() {
    float *d_a, *d_b, *d_c;
    float scalar;
    cudaMemcpy((void**)&d_a, sizeof(float)*N);
    cudaMemcpy((void**)&d_b, sizeof(float)*N);
    cudaMemcpy((void**)&d_c, sizeof(float)*N);

    dim3 dimGrid(N/dimBlock.x);
    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);
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
}
```

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

**Examples:**

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

**benefits:** lots of control; decent generality; easy to implement
**downsides:** lots of user-managed detail; brittle to changes
(“Glad I’m not an HPC Programmer!”)

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

The Unfortunate Reality:
- Performance-minded mainstream programmers will increasingly deal with parallelism
- And, as chips become more complex, locality too
Rewinding a few slides...

**MPI + OpenMP**

```c
#include <omp.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;
    }
    #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];
    if (doIO) {
        fprintf( outFile, "Failed to allocate memory (%d).\n", VectorSize );
        fclose( outFile );
    }
    return 0;
}
```

**CUDA**

```c
#define main 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);

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);
}
```

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

if( N % dimBlock.x != 0 ) dimGrid
**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 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.

---

**MPI + OpenMP**

```
#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif
static int VectorSize;
static double *a, *b, *c;

int HPCC_StartStream(HPCC_Params *params, int doIO)
{
    int 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;
    return 0;
}
```

---

The special sauce
Outline

✓ Motivation

➤ Chapel Background and Themes
  ● Survey of Chapel Concepts
  ● Project Status and Next Steps
What is Chapel?

● **An emerging parallel programming language**
  ● Design and development led by Cray Inc.
    ● in collaboration with academia, labs, industry
  ● Initiated under the DARPA HPCS program

● **Overall goal: Improve programmer productivity**
  ● Improve the programmability of parallel computers
  ● Match or beat the performance of current programming models
  ● Support better portability than current programming models
  ● Improve the robustness of parallel codes

● **A work-in-progress**
Chapel's Implementation

- Being developed as open source at SourceForge
- Licensed as BSD software

**Target Architectures:**
- Cray architectures
- multicore desktops and laptops
- commodity clusters
- systems from other vendors
- *in-progress*: CPU+accelerator hybrids, manycore, …
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/m 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

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

“Why don’t I have more control?”
3) Multiresolution Design

**Multiresolution Design:** Support multiple tiers of features
- higher levels for programmability, productivity
- lower levels for greater degrees of control

*Chapel language concepts*

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

- build the higher-level concepts in terms of the lower
- permit the user to intermix layers arbitrarily
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
```chapel
const pi = 3.14, // pi is a real
  coord = 1.2 + 3.4i, // coord is a complex...
  coord2 = pi*coord, // ...as is coord2
  name = "brad", // name is a string
  verbose = false; // verbose is boolean

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

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

writeln((sum, fullname));
```

(4.14, bradford)
Range Types and Algebra

```
const r = 1..10;

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

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

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

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

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

- Motivation
- Chapel Background and Themes
  - Survey of Chapel Concepts
    - Domain Maps
    - Data Parallelism
    - Task Parallelism
    - Base Language
    - Locality Control
- Project Status and Next Steps
Task Parallelism: Begin Statements

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

Possible outputs:

<table>
<thead>
<tr>
<th>hello world</th>
<th>good bye</th>
</tr>
</thead>
<tbody>
<tr>
<td>good bye</td>
<td>hello world</td>
</tr>
</tbody>
</table>
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
```
Bounded Buffer Producer/Consumer Example

```chapel
begin producer();
consumer();

// 'sync' types store full/empty state along with value
var buff$: [0..#buffersize] sync real;

proc producer() {
  var i = 0;
  for ... {
    i = (i+1) % buffersize;
    buff$[i] = ...; // writes block until empty, leave full
  }
}

proc consumer() {
  var i = 0;
  while ... {
    i= (i+1) % buffersize;
    ...buff$[i]...; // reads block until full, leave empty
  }
}
```
Outline

- Motivation
- Chapel Background and Themes
  - Survey of Chapel Concepts
    - Domain Maps
    - Data Parallelism
    - Task Parallelism
    - Base Language
    - Locality Control

- Project Status and Next Steps

Theme 4: Control over Locality/Affinity
The Locale Type

**Definition:**
- Abstract unit of target architecture
- Supports reasoning about locality
- Capable of running tasks and storing variables
  - i.e., has processors and memory

**Typically:** A compute node (multicore processor or SMP)
Defining 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
  begin on A[i,j] do
    bigComputation(A);
  begin on node.left do
    search(node.left);
  ```
Chapel: Scoping and Locality

var i: int;
var i: int;
on Locales[1] {
  // Code here
}
var i: int;
on Locales[1] {
  var j: int;
}
Chapel: Scoping and Locality

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

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

- **Motivation**
- **Chapel Background and Themes**
  - Survey of Chapel Concepts

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

- **Project Status and Next Steps**
Domains

**Domain:**
- A first-class index set
- The fundamental Chapel concept for data parallelism

```chapel
config const m = 4, n = 8;

var D: domain(2) = {1..m, 1..n};

var Inner: subdomain(D) = {2..m-1, 2..n-1};
```
Chapel Domain Types

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

- **dense**
- **strided**
- **sparse**
- **associative**
- **unstructured**
Chapel Domain/Array Operations

- Data Parallel Iteration (as well as serial and coforall)

\[
A = \text{forall } (i,j) \text{ in } D \text{ do } (i + j/10.0);
\]

- Array Slicing; Domain Algebra

\[
A[\text{InnerD}] = B[\text{InnerD} + (0,1)];
\]

- Promotion of Scalar Operators and Functions

\[
A = B + \alpha \times C;
\quad A = \exp(B, C);
\]

- And many others: indexing, reallocation, set operations, remapping, aliasing, queries, …
Data Parallelism Implementation Qs

Q1: How are arrays laid out in memory?
- Are regular arrays laid out in row- or column-major order? Or…?
- How are sparse arrays stored? (COO, CSR, CSC, block-structured, …?)

Q2: How are arrays stored by the locales?
- Completely local to one locale? Or distributed?
- If distributed… In a blocked manner? cyclically? block-cyclically? recursively bisected? dynamically rebalanced? …?
Q1: How are arrays laid out in memory?
- Are regular arrays laid out in row- or column-major order? Or…?
- How are sparse arrays stored? (COO, CSR, CSC, block-structured, …?)

Q2: How are arrays mapped to the locales?
- Completely local to one locale? Or distributed?
- If distributed… In a blocked manner? cyclically? block-cyclically? recursively bisected? dynamically rebalanced? …?

A: Chapel’s domain maps are designed to give the user full control over such decisions
Outline

- Motivation
- Chapel Background and Themes
  - Survey of Chapel Concepts
  - Theme 2: Global-view Abstractions

- Project Status and Next Steps
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:
**STREAM Triad: 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 indices and values
- computation will execute using local processors only
STREAM Triad: Chapel (multilocal, blocked)

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

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

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

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

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

A = B + alpha * C;
Domain Map Types

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

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

2. Advanced users can write their own domain maps in Chapel
   ● to cope with 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 domain map interface for programmers:
  \$CHPL_HOME/doc/technotes/README.dsi
- Current domain maps:
  \$CHPL_HOME/modules/dists/*.chpl
  layouts/*.chpl
  internal/Default*.chpl
Summary of this Domain Maps Section

● Chapel avoids locking crucial implementation decisions into the language specification
  ● local and distributed array implementations
  ● parallel loop implementations

● Instead, these can be…
  …specified in the language by an advanced user
  …swapped in and out with minimal code changes

● The result separates the roles of domain scientist, parallel programmer, and implementation cleanly
Outline

✓ Motivation
✓ Chapel Background and Themes
✓ Survey of Chapel Concepts
➢ Project Status and Next Steps
Implementation Status -- Version 1.8.0 (Oct 2013)

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

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
● Use Chapel for parallel programming education
When teaching parallel programming, I like to cover:
- data parallelism
- task parallelism
- concurrency
- synchronization
- locality/affinity
- deadlock, livelock, and other pitfalls
- performance tuning
- …

I don’t think there’s been a good language out there...
- for teaching all of these things
- for teaching some of these things well at all
- until now: We believe Chapel can potentially 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 Community
(see chapel.cray.com/collaborations.html for further details and possible collaboration areas)

- **Lightweight Tasking using Qthreads**: Sandia (Dylan Stark, et al.)
  - paper at CUG, May 2011
- **Lightweight Tasking using MassiveThreads**: U Tokyo (Kenjiro Taura, Jun Nakashima)
- **I/O, regexp, LLVM back-end, etc.**: LTS/UMD (Michael Ferguson, et al.)
- **Application Studies**: LLNL (Rob Neely, Bert Still, Jeff Keasler), Sandia (Richard Barrett, et al.)
- **Chapel-MPI-3 Compatibility**: Argonne (Pavan Balaji, Rajeev Thakur, Rusty Lusk)
- **Futures/Task-based Parallelism**: Rice (Vivek Sarkar, Shams Imam, Sagnak Tasirlar, et al.)
- **Parallel File I/O, Bulk-Copy Opt**: U Malaga (Rafael Asenjo, Maria Angeles Navarro, et al.)
  - papers at ParCo, Aug 2011; SBAC-PAD, Oct 2012
- **Interoperability via Babel/BRAID**: LLNL/Rice (Tom Epperly, Shams Imam, et al.)
  - paper at PGAS, Oct 2011
- **Runtime Communication Optimization**: LBNL (Costin Iancu, et al.)
- **Energy and Resilience**: ORNL (David Bernholdt, et al.)
- **Interfaces/Generics/OOP**: CU Boulder (Jeremy Siek, et al.)
- **Model Checking and Verification**: U Delaware (Stephen Siegel, T. Zirkel, T. McClory)
  (and several others as well...)
Chapel: the next five years

- **Harden Prototype to Production-grade**
  - Performance Optimizations
  - Add/Improve Lacking Features

- **Target more complex/modern compute node types**
  - e.g., CPU+GPU, Intel MIC, …

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

- **Grow the team at Cray**
  - four positions open at present (manager, SW eng, build/test/release)
Summary

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

- yet, without necessarily abdicating control
- Chapel does this via its multiresolution design
  - Here, we saw it in domain maps and leader-follower iterators
  - These avoid locking crucial performance decisions into the language

We believe Chapel can greatly improve productivity

...for current and emerging HPC architectures
...and for the growing need for parallel programming in the mainstream
For More Information: Online Resources

Chapel project page: http://chapel.cray.com
● overview, papers, presentations, language spec, …

Chapel SourceForge page: https://sourceforge.net/projects/chapel/
● release downloads, public mailing lists, code repository, …

Mailing Aliases:
● chapel_info@cray.com: contact the team at Cray
● 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
For More Information: Suggested Reading

Overview Papers:

  - a high-level overview of the project summarizing the HPCS period

  - a more detailed overview of Chapel’s history, motivating themes, features

Blog Articles:

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