Chapel Hands-On

Brad Chamberlain
Cray Inc.

Discovery 2015: September 16th, 2011
What is Chapel?

- A new parallel programming language
  - Design and development led by Cray Inc.
  - Started 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:**
- multicore desktops and laptops
- commodity clusters
- Cray architectures
- systems from other vendors
- (in-progress: CPU+accelerator hybrids, manycore, ...)
Why Chapel?

**Dynamic, arbitrary, multithreaded execution**
- Contrast with UPC/SHMEM: single-threaded SPMD

**Explicit parallel concepts in source code for (composable) data and task parallelism**
- Contrast with UPC/SHMEM: all parallelism stems from implicitly running multiple copies of the program

**Distinct concepts for locality vs. parallelism**
- Contrast with UPC/SHMEM in which the program images represent locality in addition to parallelism

**Productivity Features**
- type inference, iterator functions, rich array types, OOP, ...
This Session’s Goals:

- Teach you about Chapel
  - current status
  - future directions
- Give you a chance to program in Chapel
- Answer your questions
- Get your feedback and suggestions

But realistically speaking...?

- You’re about to be hit with a firehose of information
- You’ll likely leave knowing just enough to be dangerous

**Plug:** Come to our SC11 tutorial in Seattle for a more in-depth introduction!
Outline

- Chapel Motivation
- Quick Tour of Some Chapel Features
  - Project Status and Summary
  - Bonus Topics
**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 users to intermix layers arbitrarily
Base Language Features

- Domain Maps
- Data Parallelism
- Task Parallelism
- Base Language
- Locality Control
- Target Machine
const pi = 3.14, // pi is a real  
coord = 1.2 + 3.4i, // coord is a complex...  
coord2 = pi*loc, // ...as is coord2  
name = “brad”, // name is a real  
verbose = false; // verbose is boolean

proc addem(x, y) { // addem() is generic  
    return x + y;
}

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

writeln((sum, fullname));

(4.14, bradford)
param intSize = 32;
type elementType = real(32);
const epsilon = 0.01:elementType;
var start = 1:int(intSize);
config param intSize = 32;
config type elementType = real(32);
config const epsilon = 0.01:elementType;
config var start = 1:int(intSize);

% chpl myProgram.chpl -sintSize=64 -selementType=real
% a.out --start=2 --epsilon=0.00001
Iterators

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

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

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

```plaintext
for ij in tiledRMO(D, 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)
```
const r = 1..10;

printVals(r # 3);
printVals(r # -3);
printVals(r by 2);
printVals(r by 2 align 2);
printVals(r by -2);
printVals(r by 2 # 3);
printVals(r # 3 by 2);

proc printVals(r) {
  for i in r do
    write(r, " ");
    writeln();
}
var A: [0..9] real;

for (i,j,a) in (1..10, 2..20 by 2, A) do
  a = j + i/10.0;

writeln(A);

2.1 4.2 6.3 8.4 10.5 12.6 14.7 16.8 18.9 21.0
Default and Named Arguments

```
proc foo(name="joe", weight=175, age) {
    ...
}
foo("brad", age=101);
```
Other Base Language Features

- tuple types
- compile-time features for meta-programming
  - e.g., compile-time functions to compute types and params
- rank-independent programming features
- value- and reference-based OOP
- overloading, where clauses
- modules (for namespace management)
- ...

Locality Features

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

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

- **Properties**
  - A locale’s tasks have ~uniform access to local data
  - Other locales’ data is also accessible, but at a price

- **Locale Examples**
  - A multi-core processor
  - An SMP node
Coding 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 LocaleSpace = [0..#numLocales];
  const Locales: [LocaleSpace] locale;
  ```

Locales: L0 L1 L2 L3 L4 L5 L6 L7
Locale Operations

• Locale methods support reasoning about machine resources:

```plaintext
proc locale.physicalMemory(...) { ... }
proc locale.numCores(...) { ... }
proc locale.name(...) { ... }
```

• *On-clauses* support placement of computations:

```plaintext
writeln("on locale 0");
on Locales[1] do
  writeln("now on locale 1");
writeln("on locale 0 again");
on A[i,j] do
  begin bigComputation(A);
on node.left do
  begin search(node.left);
```
Task Parallel Features

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

begin myNewTask(); // fire-and-forget
whileOriginalTaskContinues();

cobegin {
    myFirstTask();
    mySecondTask();
} // wait for these two tasks to complete

coforall tid in 0..#numTasks {
    executeTask(tid);
} // wait for these numTasks tasks to complete
// ‘sync’ types store full/empty state along with value

var result$: sync int; // initially empty
result$ = begin computeSomething(); // writes fill
computeSomethingElse();
computeThirdThingUsingResult(result$); // reads empty
cobegin {
    producer();
    consumer();
}

var buff$: [0..#buffersize] sync real;

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

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

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

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

A = B + alpha * C;
Chapel Domain/Array Operations

- Parallel and Serial Iteration
  \[ A = \forall (i,j) \in D \ do \ (i + j/10.0); \]

- Array Slicing; Domain Algebra
  \[ A[\text{InnerD}] = B[\text{InnerD}+(0,1)]; \]

- Promotion of Scalar Functions and Operators
  \[ A = B + \alpha \times C; \quad A = \exp(B, C); \]

- And several other operations: indexing, reallocation, set operations, reindexing, aliasing, queries, ...
Chapel supports several types of domains and arrays:

- **dense**
- **strided**
- **sparse**
- **unstructured**
- **associative**
Data Parallel Features

- Domain Maps
- Data Parallelism
- Task Parallelism
- Base Language
- Locality Control
- Target Machine
Q1: How are arrays laid out in memory?
   - Are regular arrays laid out in row- or column-major order? Or...?
   - What data structure is used to store sparse arrays? (COO, CSR, ...?)

Q2: How are data parallel operators implemented?
   - How many tasks?
   - How is the iteration space divided between the tasks?

...?
Data Parallelism: Implementation Qs

**Q3:** How are arrays distributed between locales?
- Completely local to one locale? Or distributed?
- If distributed... In a blocked manner? cyclically? block-cyclically? recursively bisected? dynamically rebalanced? ...

**Q4:** What architectural features will be used?
- Can/Will the computation be executed using CPUs? GPUs? both?
- What memory type(s) is the array stored in? CPU? GPU? texture? ...

**A1:** In Chapel, any of these could be the correct answer

**A2:** Chapel’s *domain maps* are designed to give the user full control over such decisions
Domain maps are "recipes" that instruct the compiler how to map the global view of a computation...

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

...to the target locales’ memory and processors:
**Domain Maps**: “recipes for implementing parallel/distributed arrays and domains”

They define data storage:
- Mapping of domain indices and array elements to locales
- Layout of arrays and index sets in each locale’s memory

...as well as operations:
- random access, iteration, slicing, reindexing, rank change, ...
- the Chapel compiler generates calls to these methods to implement the user’s array operations
STREAM Triad: Chapel ( multicore )

```chapel
const ProblemSpace = [1..m];

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

A = B + alpha * C;
```

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

```chapel
const ProblemSpace = [1..m]
    dmapped Block(boundingBox=[1..m]);

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

A = B + alpha * C;
```
const ProblemSpace = [1..m]

dmapped Cyclic(startIdx=1);

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

A = B + alpha * C;
All Domain Types Support Domain Maps

dense

strided

sparse

unstructured

associative

“steve”
“lee”
“sung”
“david”
“jacob”
“albert”
“brad”
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 layouts and distributions will be written using the same user-defined domain map framework
   - to avoid a performance cliff between “built-in” and user-defined domain maps

4. Domain maps should only affect implementation and performance, not semantics
   - to support switching between domain maps effortlessly
Outline

✓ Chapel Motivation
✓ Quick Tour of Some Chapel Features
➢ Project Status and Summary
• Bonus Topics
• Everything you’ve heard about today works in the current compiler
  • (which is not to say that it’s bug-free or feature-complete)

• Performance can still be hit or miss
  • a number of optimizations remain
    • some low-hanging, some more aggressive
  • generally speaking...
    ...lower dimensional arrays perform better than higher-dimensional
    ...single-locale performs better than multi-locale
    ...multi-locale performs best with fine-grain, demand-driven communication patterns or embarrassingly parallel computations
Next Steps

No-brainers:
- Performance Optimizations
- Feature Improvements/Bug Fixes
- Complete HPCS deliverables
- Develop post-HPCS strategy/funding
- Support Collaborations and Users

Advanced Topics:
- Hierarchical Locales to target next-gen nodes
  - e.g., manycore, CPU+GPU hybrids, tiled processors, ...
  - additional hierarchy and heterogeneity warrants it
- Atomic Operations Library (local and remote)
Potential Future Work (pending interest/funding)

- Resiliency/Fault Tolerance
- Task Teams
  - with collective operations: reductions, barriers, eurekas
  - permitting distinct scheduling policies
- Improved Interoperability, Libraries
- Re-work warts based on user feedback
  - strings
  - syntax: domain/array literals, zipper iteration
- Improved Tools:
  - performance analysis, debugging, editor support
  - Chapel interpreter
- ...
Our Team

- **Cray:**
  - Brad Chamberlain
  - Sung-Eun Choi
  - Greg Titus
  - Vass Litvinov
  - Tom Hildebrandt

- **External Collaborators:**
  - Albert Sidelnik
  - Jonathan Turner
  - Srinivas Sridharan

- **Interns:**
  - Jonathan Claridge
  - Hannah Hemmaplardh
  - Andy Stone
  - Jim Dinan
  - Rob Bocchino
  - Mack Joyner

You? Your Friend/Student/Colleague?
Featured Collaborations (see http://chapel.cray.com/collaborations.html for more)

- **Sandia** (Kyle Wheeler, Rich Murphy): Chapel over Qthreads user threading
- **LTS** (Michael Ferguson): Improved I/O and strings
- **LLNL** (Tom Epperly et al.): Interoperability via Babel
- **UIUC** (David Padua, Albert Sidelnik, Maria Garzarán): CPU-GPU computing
- **U. Malaga** (Rafael Asenio, Maria Gonzales, Rafael Larossa): Parallel file I/O
- **CU Boulder** (Jeremy Siek, Jonathan Turner): Interfaces, concepts, generics
- **ORNL/Notre Dame** (Srinivas Sridharan, Jeff Vetter, Peter Kogge): Asynchronous software transactional memory over distributed memory
- **ORNL/ESSC** (Steve Poole, Matt Baker, ...): portability, performance tuning
- **BSC/UPC** (Alex Duran): Chapel over Nanos++ user-level tasking
- **Argonne** (Rusty Lusk, Rajeev Thakur, Pavan Balaji): Chapel over MPICH
- *(your name + idea here?)*
For Further Information

- **Chapel Home Page** (papers, presentations, tutorials): http://chapel.cray.com

- **Chapel Project Page** (releases, mailing lists, code): http://sourceforge.net/projects/chapel/

- **General Questions/Info:** chapel_info@cray.com (or SourceForge chapel-users list)

- **Upcoming Events:**
  
  **SC11** (November, Seattle WA):
  - Monday, Nov 14th: full-day comprehensive tutorial
  - Wednesday, Nov 16th: Chapel Lightning Talks BOF
  - Friday, Nov 18th: half-day broader engagement tutorial

  **PGAS11** (October, Galveston, TX): leader/follower iterator talk
Outline

✓ Chapel Motivation
✓ Quick Tour of Some Chapel Features
✓ Project Status and Summary

Bonus Topics
  • graph representations
  • atomic operations
  • collectives
  • I/O
  • tools
Graph Representation

- Graphs can be stored in a variety of ways in Chapel:
  - Edge lists
    - e.g., a 1D array of vertex objects, each of which stores an array of edges
  - Adjacency matrices
    - e.g., a 2D sparse $v \times v$ array whose entries represent connecting edges
  - “Pointer-based” representations
    - e.g., an unstructured/opaque array in which domain indices represent vertices and arrays of indices are used to represent edges
    - or, alternatively, a network of distributed, linked objects
  - ...or any other sensible thing you can conceive of

- As with any data structure selection, choice should be motivated by use cases, expected operations
  - and at present, maturity of implementation
Chapel currently has two main concepts for atomicity:

1) sync vars (low-level)
   - use a sync var’s full/empty state to guard critical sections
   - essentially a sugared lock

```plaintext
enum owner = {foo, bar};
var lock$: sync owner;
proc foo() {
    lock$.writeEF(owner.foo);
    ...critical operations...
    lock$.readFE();
}
proc bar() {
    lock$ = owner.bar;
    ...critical operations...
    lock$;
}
```

- in many cases, these locks can be logically associated with algorithmic data (e.g., see earlier bounded buffer example)
Atomic Operations: High-level

2) atomic statements (high-level, not yet available)
   - designed to execute a section of code atomically w.r.t. other tasks
   ```
   atomic {
     newNode.next = node;
     newNode.prev = node.prev;
     node.prev.next = newNode;
     node.prev = newNode;
   }
   ```
   - intended that compiler would use HW-based mechanisms when applicable and fall back on SW when not (i.e., STM)
   - but STM is very much an open research area (one that we have been pursuing jointly with U. Notre Dame & ORNL)
Due to...
...the level of effort required to get general atomics working
...the desire to support lock-free programming now
...the observation that some HW atomic ops are awkward to code and have compilers recognize automatically (e.g., CAS)

...I’ve recently proposed pursuing a third, intermediate solution: a library of standard atomic ops
- e.g., atomic increments, compare and swap, math, ...
- local and remote (use processor/network atomic ops.)
- intended as a stopgap until atomic statement is complete
  - though I expect it will continue to have utility then
- main challenges: portability, design
Collectives

• Many traditional collective operations don’t make sense in a non-SPMD execution model
  • which of the arbitrarily many tasks should be involved?
• Some collective ops are supported via keywords on aggregates: reduce, scan
  • e.g., sum = + reduce A;
• Future work:
  • Introduction notion of task teams
  • Support collectives on teams
    • reductions, barriers, broadcasts, eurekas(?)


Input/Output

- **Output**
  - `write(expr-list)`: writes the argument expressions
  - `writeln(...)` variant: writes a linefeed after the arguments

- **Input**
  - `read(expr-list)`: reads values into the argument expressions
  - `read(type-list)`: reads values of given types, returns as tuple
  - `readln(...)` variant: same, but skips through next linefeed

- **Example:**

```chapel
var first, last: string;
write(“what is your name? ”);
read(first);
last = read(string);
writeln(“Hi ”, first, “ ”, last);
```

What is your name? Chapel User
Hi Chapel User

- I/O to files and strings also supported
While our current I/O story is for simple cases, it’s a bit impoverished for real applications

- (moral: to get rich I/O, create benchmarks that require it)

Some of our collaborations are striving to improve this

- **Michael Ferguson** (LTS): Re-engineering the underpinnings of Chapel I/O to support I/O to memory buffers, sockets, data streams, etc. in addition to files and strings
  - existing console I/O interface unchanged; file I/O cleaned up
  - designed with parallel access in mind
  - initial version should be available in next 1-7 months

- **Rafael Asenjo** (U. Malaga): Working on adding support for writing distributed arrays to parallel file systems efficiently
• Tools have not been a major focus in the project so far

• Current status:
  • IDEs: vim and emacs Chapel modes available
    • see $CHPL_HOME/etc
  • performance tuning / correctness debugging: existing C tools can be applied to the generated code
    • Utility varies with style of code, sophistication of user
      • e.g., Codes with heavy overloading result in name mangling
      • Compiling with --cpp-lines supports Chapel source line numbers
  • libraries/visualization: little/no intrinsic support; support for ‘extern’ calls provides a path forward