Chapel: An Emerging Parallel Programming Language

Thomas Van Doren, Chapel Team, Cray Inc.
Northwest C++ Users’ Group
April 16th, 2014
My Employer: CRAY

THE SUPERCOMPUTER COMPANY
Parallel Challenges

Genome Sequencing
Photo: umn.edu

Virus modeling
Photo: HPCWire.com

Square-Kilometer Array
Photo: www.phy.cam.ac.uk

https://en.wikipedia.org/wiki/Human_Genome_Project
http://www.hpcwire.com/2013/12/12/supercomputing-fundamental-ska-project/
Hardware Progression

● 1988: 8 Processors - 512 MB memory
● 1998: 1,024 Processors - 1 TB memory
● 2008: 150,000 Processors - 360 TB memory
● 2020: 10,000,000 Processors? - ? memory
A trivial parallel computation

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

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

In pictures:
A 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:
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):

```
A = = = =
B + + + +
C . . . .
alpha . . . .
```
A trivial parallel computation

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

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

In pictures, in parallel (distributed memory multicore):

![Diagram](image_url)
Existing programming models?

- MPI – C, Fortran, Java
- CUDA, OpenACC for GPUs
- MapReduce
- Custom solutions

Limitations:
- Closely tied to hardware.
- Support single type of parallelism.
What is Chapel?

● An emerging parallel programming language
  ● Design and development led by Cray Inc.

● A work-in-progress

● Chapel’s overall goal: Improve programmer productivity
Chapel's Implementation

- **Open source at SourceForge**
  - Moving to Github soon!

- **BSD license**

- **Target Architectures:**
  - multicore desktops and laptops
  - commodity clusters and the cloud
  - supercomputers

- **Compiler and runtime written in C++/C**
- **Standard library and language features written in Chapel**
Chapel Community

Users:
- 1000+ downloads of each release
- 200 subscribers to chapel-users mailing list

Developers (last release):
- 19 developers
- 8 organizations
- 5 countries
Multiresolution Design

- Domain Maps
- Data Parallelism
- Task Parallelism
- Base Language
- Locality Control
- Target Machine
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*coord, // ...as is coord2
    name = “foo”, // 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, “bar”); // fullname is a string

writeln((sum, fullname));

(4.14, foobar)
Range Types and Algebra

```chapel
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(i, " ");
        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

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

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

```
0 1 1 2 3 5 8
```
Zippered Iteration

```chapel
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

...
class Circle {
    var x, y: int;
    var r: real;
}
var c = new Circle(r=2.0);

proc Circle.area()
    return pi*r**2;
writeln(c.area());

class Oval: Circle {
    var r2: real;
}
proc Oval.area()
    return pi*r*r2;

var o: Circle = new Oval(r=1.0, r2=2.0);
writeln(o.area());
Other Base Language Features

- tuple types and values
- records
- modules
- unions
- enums
- interoperability features
Tasks: discrete units of computation that can, and should, be executed in parallel.
Coforall Loops

```chapel
coforall t in 0..#numTasks {
    writeln("Hello from task ", t, " of ", numTasks);
} // implicit join of the numTasks tasks here

writeln("All tasks done");
```

Sample output:

Hello from task 2 of 4
Hello from task 0 of 4
Hello from task 3 of 4
Hello from task 1 of 4
All tasks done
Other Task Parallel Features

- begin, cobegin
- synchronization blocks
- atomic and synchronization variables
Data Parallel Features

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

```plaintext
forall a in 1..n do
    writeln("Here is an a: ", a);
```

Typically $1 \leq \#\text{Tasks} << \#\text{Iterations}$

```plaintext
forall (a, i) in zip(A, 1..n) do
    a = i/10.0;
```

Forall-loops may be zippered, like for-loops
• Corresponding iterations will match up
Domains

Domain:
- First-class index set
- Used to declare and operate on arrays
- Drive iteration

```
config const m = 4, n = 8;
var D: domain(2) = {1..m, 1..n};
```
config const m = 4, n = 8;

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

var A: [D] real;
var B: [D] string;

forall (i, j) in D {
    A(i, j) = i + j/10.0;
    B(i, j) = i + "," + j;
}
Chapel supports several types of domains (index sets):

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

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

- **Array Slicing; Domain Algebra**

  ```
  var InnerD: subdomain(D) = {2..m-1, 2..n-1};
  A[InnerD] = B[InnerD+(1,1)];
  ```

- **Promotion of Scalar Operators and Functions**

  ```
  A = B + alpha * C;
  A = exp(B, C);
  A = foo(B, C);
  ```
Promoted functions/operators are defined in terms of zippered forall loops in Chapel. For example...

\[ A = B; \]

\[ \Rightarrow \]

\[ \text{forall } (a, b) \text{ in zip}(A, B) \text{ do } a = b; \]

Whole-array operations are implemented element-wise.

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

\[ \Rightarrow \]

\[ \text{forall } (a, b, c) \text{ in } (A, B, C) \text{ do } a = b + \alpha \cdot c; \]
Trivial Example: Chapel (multicore)

\[\text{const ProblemDomain} = \{1..m\};\]

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

\[A = B + \alpha \cdot C;\]
Locale: abstract unit of target architecture that can run tasks and store variables.
Defining Locales

- Specify # of locales when running Chapel programs
  
  ```
  % a.out --numLocales=8
  ```

- Chapel provides built-in locale variables
  
  ```
  config const numLocales: int = ...;
  const Locales: [0..#numLocales] locale = ...;
  ```

- User’s `main()` begins executing on locale #0
Chapel: Scoping and Locality

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

```bash
% a.out --numLocales=5
```
Chapel: Scoping and Locality

```chapel
% a.out --numLocales=5

var i: int;
on Locales[1] {
```
Chapel: Scoping and Locality

```plaintext
% a.out --numLocales=5

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

- `i` and `j` are declared at the beginning.
- `i` is bound to locale 0, and `j` is bound to locale 1.

The diagram shows the scoping and locality of `i` and `j` across different locales.
Chapel: Scoping and Locality

```chapel
% a.out --numLocales=5

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

```
0 1 2 3 4
```

```
i j
```

```
& a.out --numLocales=5
```

```chapel
}`n
}`n
```
Chapel: Scoping and Locality

```chapel
%! a.out --numLocales=5

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

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

// Diagram showing the allocation of variables across locales:

```
0 1 2 3 4
i  k  j  k  k  k  k
```

---

% a.out --numLocales=5

```chapel
%! a.out --numLocales=5

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

      // within this scope, i, j can be referenced;
      // the implementation manages the communication
    }
  }
}
```
Trivial Example: Chapel (multicore)

\texttt{const ProblemDomain} = \{1..m\};

\texttt{var A, B, C: [ProblemDomain] real;}

A = B + alpha * C;
Domain Map Features

- 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:
Trivial Example: Chapel

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

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

A = B + alpha * C;
```
Trivial Example: Chapel (multicore)

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

var A, B, C: [ProblemDomain] 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
Trivial Example: Chapel (multilocale, blocked)

```chapel
const ProblemDomain = {1..m}

dmapped Block({1..m});

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

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

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

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

A = B + alpha * C;
```
LULESH in Chapel

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
- their corresponding iterators
Implementation Status -- Version 1.9.0 (Apr 2014)

Overall Status:
● Most features work at a functional level
● Many performance optimizations remain

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
Chapel: the next five years

- Harden Prototype to Production-grade
- Target more complex/modern compute node types
- Continue to grow the user and developer communities
For More Information: Online Resources

Chapel project page:  http://chapel.cray.com

Chapel SourceForge page:  https://sourceforge.net/projects/chapel/

Mailing Aliases:

- chapel_info@cray.com: contact the team at Cray
- chapel-announce@lists.sourceforge.net: announcement list
- 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 more detailed overview of Chapel’s history, motivating themes, features

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

...is a collaborative effort — join us!