Chapel’s Multiresolution Programming Model
Mixing High-level Parallel Abstractions with Lower-level Control

Brad Chamberlain, Chapel Team, Cray Inc.
Northwest C++ Users Group
February 21, 2018
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.
What is Chapel?
What is Chapel?

**Chapel:** A productive parallel programming language

- portable
- open-source
- a collaborative effort

**Goals:**

- Support general parallel programming
  - “any parallel algorithm on any parallel hardware”
- Make parallel programming at scale far more productive
Q: What do HPC programmers need from a language?
A: **Serial Code**: Software engineering and performance

- **Parallelism**: What should execute simultaneously?
- **Locality**: Where should those tasks execute?
- **Mapping**: How to map the program to the system?
- **Separation of Concerns**: Decouple these issues

**Chapel is a language designed to address these needs from first principles**
Chapel and Other Languages

Chapel strives to be as...
  ...programmable as Python
  ...fast as Fortran
  ...scalable as MPI, SHMEM, or UPC
  ...portable as C
  ...flexible as C++
  ...fun as [your favorite programming language]
CLBG Cross-Language Summary
(Oct 2017 standings)

Compressed Code Size (normalized to smallest entry)

Execution Time (normalized to fastest entry)

smaller

faster
CLBG Cross-Language Summary
(Oct 2017 standings, zoomed in)

Copyright 2018 Cray Inc.
CLBG Cross-Language Summary
(Oct 2017 standings, zoomed in)
CLBG Cross-Language Summary
(Oct 2017 standings)

Compressed Code Size (normalized to smallest entry)

Execution Time (normalized to fastest entry)

smaller

faster

Copyright 2018 Cray Inc.
Chapel Performance: HPC Benchmarks

**LCALS:** Chapel vs. C + OpenMP
Shared memory performance competitive with hand-coded

- **Serial LCALS kernels:** Chapel vs. g++
- **Parallel LCALS kernels:** Chapel vs g++ w/ OMP

**HPCC RA**

**HPCC RA Performance:** Chapel vs. MPI

**STREAM Triad:** Chapel vs. MPI+OpenMP

**ISx Performance:** Chapel vs. MPI, SHMEM

**Stencil PRK Scalability**

Nightly performance graphs online at: [https://chapel-lang.org/perf](https://chapel-lang.org/perf)
Chapel Performance: HPC Benchmarks

- **Local loop kernels**
- **STREAM Triad**
- **HPCC RA**
- **Isx Peformance: Chapel vs. MPI, SHMEM**
- **Stencils PRK Scalability**
- **Global Random Updates**

**Embarrassing/Pleasing Parallelism**

**Bucket-Exchange Pattern**

**Stencil Boundary Exchanges**

Nightly performance graphs online at: [https://chapel-lang.org/perf](https://chapel-lang.org/perf)
The Chapel Team at Cray (May 2017)

14 full-time employees + ~2 summer interns
Chapel Community Partners

(and several others…)

https://chapel-lang.org/collaborations.html
Tonight’s Plan

● Cover features that we haven’t in this forum before
  ● base language features of potential interest to C++ users
  ● multiresolution features for user control over parallel abstractions
    ● parallel iterators
    ● domain maps
    ● locale models

● Review core features along the way
  ● goal: quicker than in previous talks
  ● help refresh memories / bring new attendees up-to-speed

● Please ask questions as we go
Chapel language feature areas

Chapel language concepts

- Domain Maps
- Data Parallelism
- Task Parallelism
- Base Language
- Locality Control
- Target
Base Language
Base Language Features, by example

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

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

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

```
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Copyright 2018 Cray Inc.
Base Language Features, by example

Configuration declarations
(to avoid command-line argument parsing)
./a.out --n=1000000

```plaintext
const n = 10;

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

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

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

0 1 1 2 3 5 8 ...

Copyright 2018 Cray Inc.
Base Language Features, by example

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

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

Modern iterators
Base Language Features, by example

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

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

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

Static type inference for:
- arguments
- return types
- variables

0 1 1 2 3 5 8 ...

Copyright 2018 Cray Inc.
Base Language Features, by example

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

config const n = 10;

for (i,f) in zip(0..#n, fib(n)) do
    writeln("fib #", i, " is ", f);
```

Zippered iteration

| fib #0 is 0 |
| fib #1 is 1 |
| fib #2 is 1 |
| fib #3 is 2 |
| fib #4 is 3 |
| fib #5 is 5 |
| fib #6 is 8 |
| ...       |
Base Language Features, by example

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

```javascript
config const n = 10;
for (i, f) in zip(0..#n, fib(n)) do
  writeln("fib #", i, " is ", f);
```

Range types and operators

```
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
...```

Copyright 2018 Cray Inc.
iter fib(n) {
    var current = 0,
        next = 1;
    for i in 1..n {
        yield current;
        current += next;
        current <=> next;
    }
}

config const n = 10;
for (i, f) in zip(0..#n, fib(n)) do
    writeln("fib #", i, " is ", f);

<table>
<thead>
<tr>
<th>fib</th>
<th>is</th>
</tr>
</thead>
<tbody>
<tr>
<td>#0</td>
<td>0</td>
</tr>
<tr>
<td>#1</td>
<td>1</td>
</tr>
<tr>
<td>#2</td>
<td>1</td>
</tr>
<tr>
<td>#3</td>
<td>2</td>
</tr>
<tr>
<td>#4</td>
<td>3</td>
</tr>
<tr>
<td>#5</td>
<td>5</td>
</tr>
<tr>
<td>#6</td>
<td>8</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Base Language Features, by example

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

```typescript
config const n = 10;
for (i, f) in zip(0..#n, fib(n)) do
    writeln("fib ", i, " is ", f);
```

```
fib #0 is 0
fib #1 is 1
fib #2 is 1
fib #3 is 2
fib #4 is 3
fib #5 is 5
fib #6 is 8
...```
Other Base Language Features of Potential Interest to C++ Users
Other Base Language Features: OOP

Two flavors of object-oriented types:

- **Value-based:**
  ```plaintext
type Circle =
  record {
    var radius: real;
    proc area() { ... }
  }

var myCircle = new Circle(radius = 1.0),
  myCircle2 = myCircle;
myCircle.radius = 2.0;
writeln(myCircle2.area()); // 1.0 for record, 4.0 for class
```

- **Reference-based:**
  ```plaintext
type Circle =
  class {
    var radius: real;
    proc area() { ... }
  }

var myCircle = new Circle(radius = 1.0),
  myCircle2 = myCircle;
myCircle.radius = 2.0;
writeln(myCircle2.area()); // 1.0 for record, 4.0 for class
```
Other Base Language Features: Generics

- Support for generic types and functions
  - w.r.t. types and statically known values (`param`s)

```plaintext
class Arr {
  param numDims: int;  // number of dimensions
  type eltType;         // element type
  var size: numDims*int; // tuple storing per-dimension size
}

var myArr = new Arr(2, string, (100, 200)),
myArr2 = new Arr(3, real, (500, 500, 500));
```
Support for generic types and functions

- w.r.t. types and statically known values (`param`s)

```chapel
proc mypow(type t, x: t, param exponent: int) {
    var result = 1:t;
    for param i in 1..exponent do
        result *= x;
    return result;
}
```

```chapel
var twoSquared = mypow(int, 2, 2);
var piCubed = mypow(real, 3.14159265, 3);
```

Note: this is an utterly artificial and over-engineered way to write this function in Chapel, done merely to demonstrate type/param args in ~6 lines…
Other Base Language Features: Meta-Programming

- Compile-time procedures to compute types / params

```chapel
proc computePacketSize(type t1, type t2) param {
    return numBits(t1) + numBits(t2);
}
proc c_intToChapelInt() type {
    return int(numBits(c_int));
}
```

- Also, support for config types / params

```chapel
config param bitsPerInt = 16;
config type eltType = int(bitsPerInt);
```

```
chpl -sbitsPerInt=64 -seltType=real(32) myProc.chpl
```
Other Base Language Features: Meta-Programming

● Ability to unroll loops / fold conditionals or ‘void’ exprs

```cpp
for param i in (1, 2.3, "hello", (5,7)) do
    writeln("i: ", i, " has type: ", i.type:string);
```

● Reflection module:

- “Can this function / method be resolved”
- “Iterate over all fields in this record giving me their names / types”
- …
Other Base Language Features

- Error-handling
- Modules (namespaces)
- Overloading, filtering
- Default args, arg intents, keyword-based arg passing
- Argument type queries / pattern-matching
- ...

Copyright 2018 Cray Inc.
Base Language Features: What’s Missing?

- **better initializer (constructor) features**
  - currently being implemented and refined

- **delete-free programming / borrow-checking**
  - currently being designed and implemented

- **first-class functions**
  - prototyped, need strengthening

- **constrained generics / interfaces / concepts**
  - proposal drafted but not implemented

- **anti-function hijacking features**
  - currently under consideration
Task Parallelism and Locality Control
Locales

- Unit of the target system useful for reasoning about locality
  - Each locale can run tasks and store variables
    - Has processors and memory (or can defer to something that does)
  - For most HPC systems, locale == compute node

![Locales diagram]

User’s main() executes on locale #0
Task Parallelism and Locality, by example

taskParallel.chpl

coforall loc in Locales do
  on loc {
    const numTasks = here.numPUs();
    coforall tid in 1..numTasks do
      printf("Hello from task %n of %n "+
        "running on %s\n",
        tid, numTasks, here.name);
  }

prompt> chpl taskParallel.chpl -o taskParallel
prompt> ./taskParallel --numLocales=2
Hello from task 1 of 2 running on n1033
Hello from task 2 of 2 running on n1032
Hello from task 2 of 2 running on n1033
Hello from task 1 of 2 running on n1032
Task Parallelism and Locality, by example

```
coforall loc in Locales do
  on loc {
    const numTasks = here.numPUs();
    coforall tid in 1..numTasks do
      printf("Hello from task %n of %n "+
              "running on %s\n", tid, numTasks, here.name);
  }
```

Prompt> chpl taskParallel.chpl -o taskParallel
Prompt> ./taskParallel --numLocales=2
Hello from task 1 of 2 running on n1033
Hello from task 2 of 2 running on n1032
Hello from task 2 of 2 running on n1033
Hello from task 1 of 2 running on n1032
Task Parallelism and Locality, by example

High-Level Task Parallelism

```chpl
coforall loc in Locales do
  on loc {
    const numTasks = here.numPUs();
    coforall tid in 1..numTasks do
      writef("Hello from task %n of %n "+
             "running on %s\n", tid, numTasks, here.name);
  }
```

```bash
prompt> chpl taskParallel.chpl -o taskParallel
prompt> ./taskParallel --numLocales=2
Hello from task 1 of 2 running on n1033
Hello from task 2 of 2 running on n1032
Hello from task 2 of 2 running on n1033
Hello from task 1 of 2 running on n1032
```
Task Parallelism and Locality, by example

Control of Locality/Affinity

```
taskParallel.chpl

coforall loc in Locales do
  on loc {
    const numTasks = here.numPUs();
    coforall tid in 1..numTasks do
      printf("Hello from task \%n of \%n "+
        "running on \%s\n", tid, numTasks, here.name);
  }
```

```
prompt> chpl taskParallel.chpl -o taskParallel
prompt> ./taskParallel --numLocales=2
Hello from task 1 of 2 running on n1033
Hello from task 2 of 2 running on n1032
Hello from task 2 of 2 running on n1033
Hello from task 1 of 2 running on n1032
```
Task Parallelism and Locality, by example

```
coforall loc in Locales do
  on loc {
    const numTasks = here.numPUs();
    coforall tid in 1..numTasks do
      printf("Hello from task %n of %n +
          "Running on %s\n",
          tid, numTasks, here.name);
  }
```

```
prompt> chpl taskParallel.chpl -o taskParallel
prompt> ./taskParallel --numLocales=2
Hello from task 1 of 2 running on n1033
Hello from task 2 of 2 running on n1032
Hello from task 2 of 2 running on n1033
Hello from task 1 of 2 running on n1032
```
High-Level Task Parallelism

```
coforall loc in Locales do
  on loc {
    const numTasks = here.numPUs();
    coforall tid in 1..numTasks do
      printf("Hello from task %n of %n "+
              "running on %s\n",
      tid, numTasks, here.name);
  }
```

```
prompt> chpl taskParallel.chpl -o taskParallel
prompt> ./taskParallel --numLocales=2
Hello from task 1 of 2 running on n1033
Hello from task 2 of 2 running on n1032
Hello from task 2 of 2 running on n1033
Hello from task 1 of 2 running on n1032
```
Task Parallelism and Locality, by example

```chapel
coforall loc in Locales do
  on loc {
    const numTasks = here.numPUs();
    coforall tid in 1..numTasks do
      printf("Hello from task %d of %d running on %s\n", tid, numTasks, here.name);
  }
```

```
prompt> chpl taskParallel.chpl -o taskParallel
prompt> ./taskParallel --numLocales=2
Hello from task 1 of 2 running on n1033
Hello from task 2 of 2 running on n1032
Hello from task 2 of 2 running on n1033
Hello from task 1 of 2 running on n1032
```

Not seen here:
Data-centric task coordination via atomic and full/empty vars
Task Parallelism and Locality, by example

```
taskParallel.chpl

coforall loc in Locales do
  on loc {
    const numTasks = here.numPUs();
    coforall tid in 1..numTasks do
      printf("Hello from task %n of %n "+
             "running on %s
",
            tid, numTasks, here.name);
  }
```

prompt> chpl taskParallel.chpl -o taskParallel
prompt> ./taskParallel --numLocales=2
Hello from task 1 of 2 running on n1033
Hello from task 2 of 2 running on n1032
Hello from task 2 of 2 running on n1033
Hello from task 1 of 2 running on n1032
Chapel: Scoping and Locality

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

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

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

```chapel
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 {
```
Chapel: Scoping and Locality

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

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

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

OK to access \(i\), \(j\), and \(k\) wherever they live.

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

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

Here, $i$ and $j$ are remote, so the compiler + runtime will transfer their values.

$k = 2*i + j$;
Chapel: Locality queries

```chapel
var i: int;
on Locales[1] {
  var j: int;
  coforall loc in Locales {
    on loc {
      ...here...  // query the locale on which this task is running
      ...j.locale...  // query the locale on which j is stored
      ...here.physicalMemory(...)...  // query system characteristics
      ...here.runningTasks()...  // query runtime characteristics
    }
  }
}
```

Locales (think: “compute nodes”)
Data Parallelism in Chapel

Chapel language concepts

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

Higher-level Chapel
Data Parallelism, by example

```chpl
config const n = 1000;
var D = {1..n, 1..n};

var A: [D] real;
forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

```
prompt> chpl dataParallel.chpl -o dataParallel
prompt> ./dataParallel --n=5
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```
Data Parallelism, by example

Domains (Index Sets)

```
config const n = 1000;
var D = {1..n, 1..n};

var A: [D] real;
forall (i, j) in D do
  A[i, j] = i + (j - 0.5)/n;
writeln(A);
```

```
prompt> chpl dataParallel.chpl -o dataParallel
prompt> ./dataParallel --n=5
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```
Data Parallelism, by example

```chpl
config const n = 1000;
var D = {1..n, 1..n};

var A: [D] real;
forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

```
prompt> chpl dataParallel.chpl -o dataParallel
prompt> ./dataParallel --n=5
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```
Data Parallelism, by example

```
config const n = 1000;
var D = {1..n, 1..n};

var A: [D] real;
forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

prompt> chpl dataParallel.chpl -o dataParallel
prompt> ./dataParallel --n=5

1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
Data Parallelism, by example

This is a shared memory program
Nothing has referred to remote locales, explicitly or implicitly

```
dataParallel.chpl

config const n = 1000;
var D = {1..n, 1..n};

var A: [D] real;
forall (i,j) in D do
    A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

```
prompt> chpl dataParallel.chpl -o dataParallel
prompt> ./dataParallel --n=5
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```
Data Parallelism, by example

This is a shared memory program
Nothing has referred to remote locales, explicitly or implicitly

```
config const n = 1000;
var D = {1..n, 1..n};

var A: [D] real;
forall (i, j) in D do
    A[i, j] = i + (j - 0.5)/n;
writeln(A);
```

```
prompt> chpl dataParallel.chpl -o dataParallel
prompt> ./dataParallel --n=5
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```
Distributed Data Parallelism, by example

```
use CyclicDist;
config const n = 1000;
var D = {1..n, 1..n}
dmapped Cyclic(startIdx = (1,1));
var A: [D] real;
forall (i,j) in D do
    A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

Prompt>
```
chpl dataParallel.chpl -o dataParallel
```

Prompt>
```
./dataParallel --n=5 --numLocales=4
```

```
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```
Distributed Data Parallelism, by example

```
use CyclicDist;
config const n = 1000;
var D = {1..n, 1..n}
dmapped Cyclic(startIdx = (1,1));
var A: [D] real;
forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

Prompt > chpl dataParallel.chpl -o dataParallel
Prompt > ./dataParallel --n=5 --numLocales=4

```
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```

Not at all...
- Lowering of code is well-defined
- User can control details
- Part of Chapel's multiresolution philosophy...

**magic!**  **HPF-like!**  **descriptive!**
Chapel’s Multiresolution Design: Motivation

“Why is everything so tedious/difficult?”

“Why don’t I have more control?”

“Why don’t my programs trivially port to new systems?”
Multiresolution Design: Support multiple tiers of features

- higher levels for programmability, productivity
- lower levels for greater degrees of control

- build the higher-level concepts in terms of the lower
- permit users to intermix layers arbitrarily
Distributed Data Parallelism, by example

Chapel’s prescriptive approach:

```chpl
forall (i, j) in D do...
```

⇒ invoke and inline D’s default parallel iterator
- defined by D’s type / domain map

```
dataParallel.chpl
forall (i, j) in D do...
```

```
config const n = 1000;
var D = {1..n, 1..n};

var A: [D] real;
forall (i, j) in D do
  A[i, j] = i + (j - 0.5)/n;
writeln(A);
```

```
prompt> chpl dataParallel.chpl -o dataParallel
prompt> ./dataParallel --n=5 --numLocales=4
```

```
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```
Distributed Data Parallelism, by example

**Chapel’s prescriptive approach:**

```chapel
forall (i,j) in D do...
```

⇒ invoke and inline D’s default parallel iterator
  - defined by D’s type / domain map

```chapel
use CyclicDist;
config const n = 1000;
var D = {1..n, 1..n}
dmapped Cyclic(startIdx = (1,1));
var A: [D] real;
forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

**default domain map**

**cyclic domain map on each target locale...**
  - create a task per core
  - block local indices across tasks

```chapel
prompt> chpl dataParallel.chpl -o dataParallel
prompt> ./dataParallel --n=5 --numLocales=4
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```
Distributed Data Parallelism, by example

Chapel’s prescriptive approach:

```chapel
forall (i,j) in D do...
```

**What if I don’t like D’s iteration strategy?**

- **Write and call your own parallel iterator:**
  ```chapel
  forall (i,j) in myParIter(D) do...
  ```
- **Or, use a different domain map:**
  ```chapel
  var D = {1..n, 1..n} dmapped Block(...);
  ```
- **Or, write and use your own domain map:**
  ```chapel
  var D = {1..n, 1..n} dmapped MyDomMap(...);
  ```

```chapel
use CyclicDist;
config const n = 1000;
var D = {1..n, 1..n}
dmapped Cyclic(startIdx = (1,1));
var A: [D] real;
forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;
```

```
dataParallel.chpl
```
```
distributedDataParallel.chpl -o dataParallel
```
```
--n=5 --numLocales=4
```
Write and Call Your Own Parallel Iterator
Authoring Parallel Iterators

- Similar to serial iterators, but invoked by `forall` loops
  - Unlike serial iterators, these can contain parallel constructs

```plaintext
forall i in myParIter(D) { ... }
```

invokes:
```plaintext
iter myParIter(dom: domain, ... /* tag as a parallel iterator */) { 
  coforall tid in 0..#numTasks {
    const myChunk = computeChunk(dom, tid, numTasks);
    for i in myChunk do
      yield i;
  }
}
```
Parallel iterators can also support zippered iteration

```
forall (i,j) in zip(myParIter(D), A) { ... }
```

defined in terms of leader...

```
iter myParIter(dom: domain, ...) {
  coforall tid in 0..#numTasks do
    yield computeChunk(dom, tid, numTasks);
}
```

...and follower iterators:

```
iter myParIter(dom: domain, followThis, ...) {
  for i in followThis do yield i;
}
```
Use a Different Domain Map
Domain Maps: A Multiresolution Feature

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:
Domain Maps: A Multiresolution Feature

Domain maps are “recipes” that instruct the compiler how to map the global view of a computation...

...to the target locales’ memory

Domain Maps specify...
...mapping of indices to locales
...layout of domains / arrays in memory
...parallel iteration strategies
...core operations on arrays / domains
Sample Domain Maps: Block and Cyclic

```plaintext
var Dom = {1..4, 1..8} dmapped Block({1..4, 1..8});
```

```
var Dom = {1..4, 1..8} dmapped Cyclic(startIdx=(1,1));
```

---

Copyright 2018 Cray Inc.
Write and Use Your Own Domain Map
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 end-user framework
   ● to avoid a performance cliff between “built-in” and user-defined cases
   ● in fact every Chapel array is implemented using this framework
Domain Map Descriptors

**Domain Map**

- **Represents:** a domain map value
- **Generic w.r.t.:** index type
- **State:** the domain map’s representation
- **Typical Size:** $\Theta(1) \rightarrow \Theta(\text{numLocales})$
- **Required Interface:**
  - create new domains
  - which locale owns index $i$?

**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
  - get/set of sparse “zero” values
  - …
Chapel and Performance Portability

- Avoid locking key policy decisions into the language
  - Array memory layout?
  - Sparse storage format?
  - Parallel loop policies?
Chapel and Performance Portability

- Avoid locking key policy decisions into the language
  - Array memory layout? not defined by Chapel
  - Sparse storage format? not defined by Chapel
  - Parallel loop policies? not defined by Chapel
  - Abstract node architecture? not defined by Chapel

- Instead, permit users to specify these in Chapel itself
  - support performance portability through…
    …a separation of concerns
    …abstractions—known to the compiler, and therefore optimizable
  - goal: make Chapel a future-proof language
Historically, Chapel’s locales were black boxes
- Intra-node concerns handled by compiler, runtime, OS

This was sufficient when compute nodes were simple
Classic Locales
Classic Locales

- Classic model breaks down for more complex cases
  - E.g. multiple flavors of memory or processors

Could hope compilers will “simply get smart enough”

...but seems naïve and doesn’t match Chapel’s philosophy
Hierarchical Locales

- So, we made locales hierarchical
Hierarchical Locales

- So, we made locales hierarchical
  - Locales can now themselves contain locales
    - E.g., an accelerator sub-locale, a scratchpad memory sub-locale

- Target sub-locales with on-clauses, as before
  - on Locales[0].GPU do computationThatLikesGPUs();
  - Ideally, hide such logic in abstractions: domain maps, parallel iterators

- Introduced a new multiresolution type: locale models
Chapel’s Locale Models

● User-specified type representing locales
● Similar goals to domain maps:
  ● Support user implementation of key high-level abstractions
  ● Make language future-proof (w.r.t. emerging architectures)
Authoring a Locale Model

● Creating a locale model:
  ● Create a top-level locale object type
    ● In turn, it can contain fields representing sub-locales
  ● Each locale / sub-locale type must meet a required interface:
    ● Memory: How is it managed? (malloc, realloc, free)
    ● Tasking: How do I launch and synchronize tasks?
    ● Communication: How are data & control transferred between locales?
      ● gets, puts, active messages
      ● widening of pointers
An Example: The numa Locale Model

```chpl
class NumaDomain : AbstractLocaleModel {
    const sid: chpl_sublocID_t;
}

// The node model
class LocaleModel : AbstractLocaleModel {
    const numSublocales: int;
    var childSpace: domain(1);
    var childLocales: [childSpace] NumaDomain;
}

// support for memory management
proc chpl_here_alloc(size: int, md: int(16)) { … }

// support for "on" statements
proc chpl_executeOn(loc: chpl_localeID_t, // target locale
    fn: int, // on-body func idx
    args: c_void_ptr, // func args
    args_size: int(32) // args size
) { … }

// support for tasking stmts: begin, cobegin, coforall
proc chpl_taskListAddCoStmt(subloc_id: int, // target subloc
    fn: int, // body func idx
    args: c_void_ptr, // func args
    ref tlist: _task_list, // task list
    tlist_node_id: int // task list owner
) { … }
```

$CHPL_HOME/modules/…/numa/LocaleModel.chpl

NUMA compute node

physical

conceptual

 NUMA domain
 mem
 cpu
 mem
 cpu
 NUMA domain

http://www1.pcmag.com/media/images/337192-intel-xeon-e5-chip.jpg?thumb=y
Locale Models: Status

- All Chapel compilations use a locale model
  - Set via environment variable or compiler flag
- Current locale models:
  - **flat**: the default, has no sublocales (as in the classic model)
  - **numa**: supports a sub-locale per NUMA domain within the node
  - **knl**: for Intel® Xeon Phi™: numa w/ sublocale for HBM/MCDRAM
Wrapping Up
Summary

● Chapel’s design uses a multiresolution philosophy
  ● High-level for productivity
  ● Low-level for control
  ● User-extensible for flexibility, future-proof design

● Three key examples of multiresolution features:
  ● Parallel iterators: specify the implementation of forall loops
  ● Domain maps: specify the implementation of domains and arrays
  ● Locale models: specify the capabilities of the target architecture
Chapel’s Home in the Landscape of New Scientific Computing Languages (and what it can learn from the neighbours)

Jonathan Dursi, The Hospital for Sick Children, Toronto
“My opinion as an outsider… is that Chapel is important, Chapel is mature, and Chapel is just getting started. “If the scientific community is going to have frameworks for solving scientific problems that are actually designed for our problems, they’re going to come from a project like Chapel. “And the thing about Chapel is that the set of all things that are ‘projects like Chapel’ is ‘Chapel.’”

–Jonathan Dursi

Chapel’s Home in the New Landscape of Scientific Frameworks
(and what it can learn from the neighbours)
CHIUW 2017 keynote

Chapel Resources
Chapel Central: [https://chapel-lang.org/](https://chapel-lang.org/)

**The Chapel Parallel Programming Language**

**What is Chapel?**

Chapel is a modern programming language that is...

- **parallel**: contains first-class concepts for concurrent and parallel computation
- **productiv**: designed with programmability and performance in mind
- **portable**: runs on laptops, clusters, the cloud, and HPC systems
- **scalable**: supports locality-oriented features for distributed memory systems
- **open-source**: hosted on GitHub, permissively licensed

**New to Chapel?**

As an introduction to Chapel, you may want to...

- read a blog article or book chapter
- watch an overview talk or browse its slides
- download the release
- browse sample programs
- view other resources to learn how to trivially write distributed programs like this:

```plaintext
use CyclicDist; // use the Cyclic distribution library
for i in (1..n) do
    writeln("Hello from iteration ", i, " of ", n, " running on node ", here.id);";
```

**What's Hot?**

- Chapel 1.16 is now available—download a copy today!
- The CHIUW 2018 call for participation is now available!
- A recent Cray blog post reports on highlights from CHIUW 2017.
- Chapel is now one of the supported languages on Try It Online!
- Watch talks from ACCU 2017, CHIUW 2017, and ATPESC 2016 on YouTube.
- Browse slides from PADAL, EAGE, EMBRACE, ACCU, and other recent talks.
- See also: What's New?
How to Track Chapel

http://facebook.com/ChapelLanguage
http://twitter.com/ChapelLanguage
https://www.youtube.com/channel/UCHmm27bYjhknK5mU7ZzPGsQ/
chapel-announce@lists.sourceforge.net
Chapel chapter from *Programming Models for Parallel Computing*

- a detailed overview of Chapel’s history, motivating themes, features
- published by MIT Press, November 2015
- edited by Pavan Balaji (Argonne)
- chapter is now also available [online](https://chapel-lang.org/papers.html)

Other Chapel papers/publications available at [https://chapel-lang.org/papers.html](https://chapel-lang.org/papers.html)
Suggested Reading (short attention spans)

- a run-down of recent events

- a short-and-sweet introduction to Chapel

**Six Ways to Say “Hello” in Chapel** *(parts 1, 2, 3)*, Cray Blog, Sep-Oct 2015.
- a series of articles illustrating the basics of parallelism and locality in Chapel

**Why Chapel?** *(parts 1, 2, 3)*, Cray Blog, Jun-Oct 2014.
- a series of articles answering common questions about why we are pursuing Chapel in spite of the inherent challenges

*(index available on chapel-lang.org “blog posts” page)*, Apr-Nov 2012.
- a series of technical opinion pieces designed to argue against standard reasons given for not developing high-level parallel languages
Chapel StackOverflow and GitHub Issues
Where to..

Submit bug reports:
GitHub issues for chapel-lang/chapel: public bug forum
chapel_bugs@cray.com: for reporting non-public bugs

Ask User-Oriented Questions:
StackOverflow: when appropriate / other users might care
#chapel-users (irc.freenode.net): user-oriented IRC channel
chapel-users@lists.sourceforge.net: user discussions

Discuss Chapel development
chapel-developers@lists.sourceforge.net: developer discussions
#chapel-developers (irc.freenode.net): developer-oriented IRC channel

Discuss Chapel’s use in education
chapel-education@lists.sourceforge.net: educator discussions

Directly contact Chapel team at Cray: chapel_info@cray.com
Questions?
Legal Disclaimer

Information in this document is provided in connection with Cray Inc. products. No license, express or implied, to any intellectual property rights is granted by this document.

Cray Inc. may make changes to specifications and product descriptions at any time, without notice.

All products, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

Cray hardware and software products may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Cray uses codenames internally to identify products that are in development and not yet publically announced for release. Customers and other third parties are not authorized by Cray Inc. to use codenames in advertising, promotion or marketing and any use of Cray Inc. internal codenames is at the sole risk of the user.

Performance tests and ratings are measured using specific systems and/or components and reflect the approximate performance of Cray Inc. products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.

The following are trademarks of Cray Inc. and are registered in the United States and other countries: CRAY and design, SONEXION, and URIKA. The following are trademarks of Cray Inc.: ACE, APPRENTICE2, CHAPEL, CLUSTER CONNECT, CRAYPAT, CRAYPORT, ECOPHLEX, LIBSCI, NODEKARE, THREADSTORM. The following system family marks, and associated model number marks, are trademarks of Cray Inc.: CS, CX, XC, XE, XK, XMT, and XT. The registered trademark LINUX is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Other trademarks used in this document are the property of their respective owners.