

Chapel 101

Brad Chamberlain

CHIUW 2019

June 22, 2019



bradc@cray.com



chapel-lang.org



@ChapelLanguage



CRAY®



What is Chapel?

CRAY

Chapel: A modern parallel programming language

- portable & scalable
- open-source & collaborative

Goals:

- Support general parallel programming
 - “any parallel algorithm on any parallel hardware”
- Make parallel programming at scale far more productive



What does “Productivity” mean to you?

CRAY

Recent Graduates:

“something similar to what I used in school: Python, Matlab, Java, ...”

Seasoned HPC Programmers:

“that sugary stuff which I don’t need because I require full control to get performance”

Computational Scientists:

“something that lets me express my parallel computations without having to wrestle with architecture-specific details”

Chapel Team:

“something that lets computational scientists express what they want,
without taking away the control that HPC programmers want,
implemented in a language as attractive as recent graduates want.”

Why Consider New Languages at all?

CRAY

Syntax

- High level, elegant syntax
- Improve programmer productivity

Semantics

- Static analysis can help with correctness
- We need a compiler (front-end)

Performance

- If optimizations are needed to get performance
- We need a compiler (back-end)

Algorithms

- Language defines what is easy and hard
- Influences algorithmic thinking

[Source: Kathy Yelick,
CHI UW 2018 keynote:
*Why Languages Matter
More Than Ever*]

Comparing Chapel to Other Languages

CRAY

Chapel aims 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]

Outline

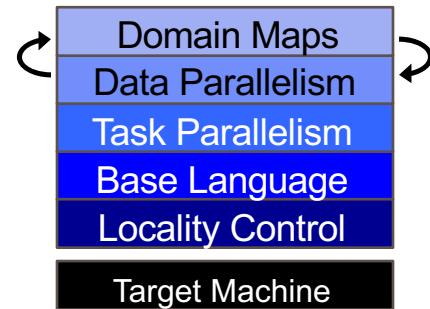
- ✓ Context and Motivation
- A Brief Tour of Chapel Features
 - Chapel Evaluations
 - Summary and Resources



Chapel Feature Areas

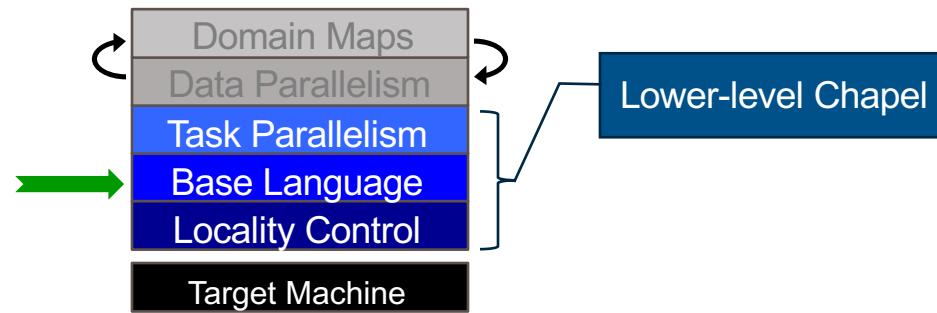
CRAY

Chapel language concepts



Base Language

CRAY



Base Language Features, by example

CRAY

```
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);
```

```
0  
1  
1  
2  
3  
5  
8  
...
```

Base Language Features, by example

CRAY

```
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);
```

```
0  
1  
1  
2  
3  
5  
8  
...
```

Configuration declarations
(support command-line overrides)
.fib --n=1000000

Base Language Features, by example

CRAY

Iterators

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

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

```
0  
1  
1  
2  
3  
5  
8  
...
```

Base Language Features, by example

Static type inference for:

- arguments
- return types
- variables

```
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);
```

```
0  
1  
1  
2  
3  
5  
8  
...
```

Base Language Features, by example

CRAY

Explicit types also supported

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

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

```
0  
1  
1  
2  
3  
5  
8  
...
```

Base Language Features, by example

CRAY

```
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);
```

```
0
1
1
2
3
5
8
...
...
```

Base Language Features, by example

CRAY

```
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);
```

```
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
...
...
```

Zippered iteration

Base Language Features, by example

CRAY

Range types and operators

```
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);
```

```
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

CRAY

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

Tuples

```
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  
...
```

Base Language Features, by example

CRAY

```
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);
```

```
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
...
...
```

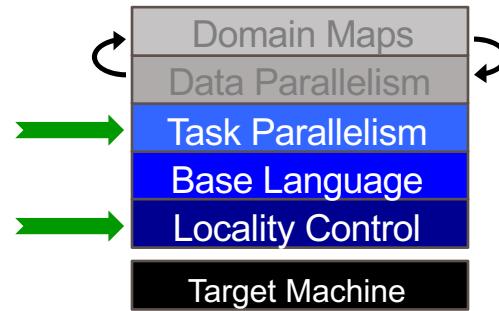
Other Base Language Features

CRAY

- **Object-oriented programming** (value- and reference-based)
 - Managed objects and lifetime checking
 - Nilable vs. non-nilable class variables
- **Generic programming / polymorphism**
- **Error-handling**
- **Compile-time meta-programming**
- **Modules** (supporting namespaces)
- **Procedure overloading / filtering**
- **Arguments:** default values, intents, name-based matching, type queries
- and more...

Task Parallelism and Locality Control

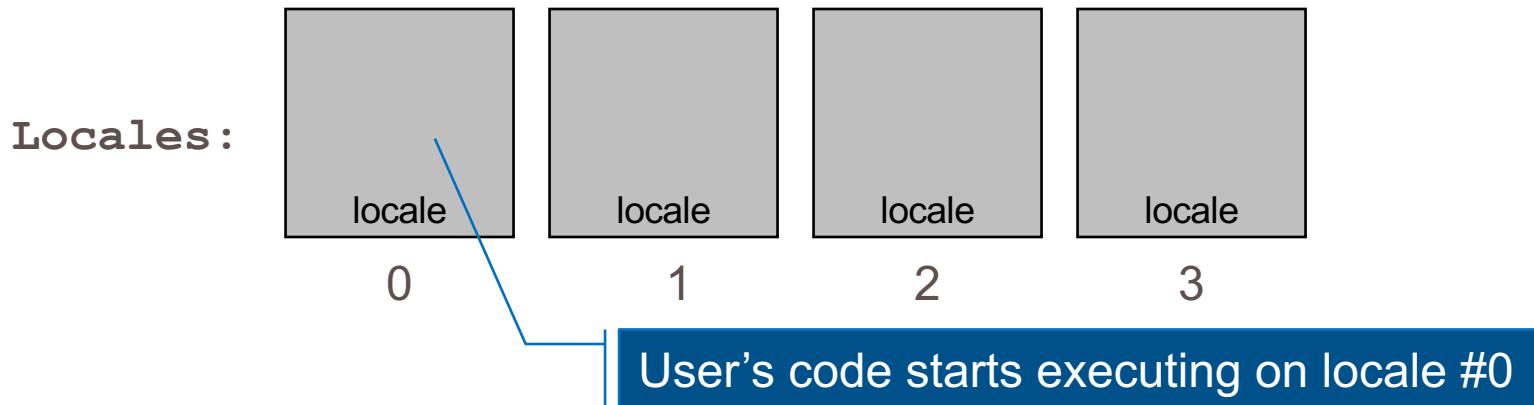
CRAY



Locales, briefly

- Locales can run tasks and store variables
 - Think “compute node”
 - The number of locales is specified on the execution command-line

```
> ./myProgram --numLocales=4      # or ` -nl 4`
```



Task Parallelism and Locality, by example

CRAY

taskParallel.chpl

```
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);
```

```
prompt> chpl taskParallel.chpl
prompt> ./taskParallel
Hello from task 2 of 2 running on n1032
Hello from task 1 of 2 running on n1032
```

Task Parallelism and Locality, by example

CRAY

Abstraction of
System Resources

taskParallel.chpl

```
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);
```

```
prompt> chpl taskParallel.chpl
prompt> ./taskParallel
Hello from task 2 of 2 running on n1032
Hello from task 1 of 2 running on n1032
```

Task Parallelism and Locality, by example

CRAY

High-Level
Task Parallelism

taskParallel.chpl

```
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);
```

```
prompt> chpl taskParallel.chpl
prompt> ./taskParallel
Hello from task 2 of 2 running on n1032
Hello from task 1 of 2 running on n1032
```

Task Parallelism and Locality, by example

CRAY

So far, this is a shared memory program
Nothing refers to remote locales,
explicitly or implicitly

taskParallel.chpl

```
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);
```

```
prompt> chpl taskParallel.chpl
prompt> ./taskParallel
Hello from task 2 of 2 running on n1032
Hello from task 1 of 2 running on n1032
```

Task Parallelism and Locality, by example

CRAY

taskParallel.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);
    }
```

```
prompt> chpl taskParallel.chpl
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

CRAY

Abstraction of
System Resources

taskParallel.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);
    }
```

```
prompt> chpl taskParallel.chpl
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

CRAY

High-Level
Task Parallelism

taskParallel.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);
    }
```

```
prompt> chpl taskParallel.chpl
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

CRAY

Control of Locality/Affinity

taskParallel.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);
    }
```

```
prompt> chpl taskParallel.chpl
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

CRAY

taskParallel.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);
    }
```

```
prompt> chpl taskParallel.chpl
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
```

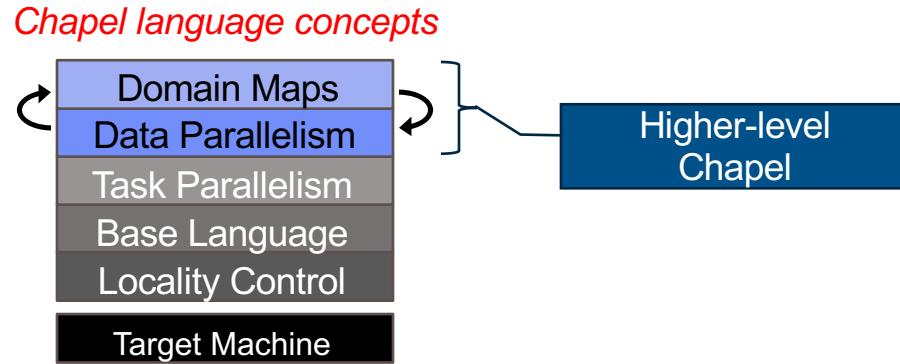
Other Task Parallel Features

CRAY

- **atomic / synchronized variables:** for sharing data & coordination
- **begin / cobegin statements:** other ways of creating tasks
- **task intents:** for specifying how outer-scope variables are passed to tasks

Data Parallelism in Chapel

CRAY



Data Parallelism, by example

CRAY

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
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

CRAY

Domains (Index Sets)

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
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

CRAY

Arrays

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
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

CRAY

Data-Parallel Forall Loops

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
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

CRAY

So far, this is a shared memory program
Nothing refers 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
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

CRAY

Domain Maps
(Map Data Parallelism to the System)

dataParallel.chpl

```
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
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

CRAY

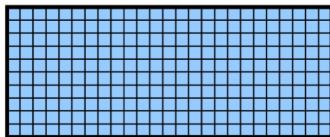
dataParallel.chpl

```
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);
```

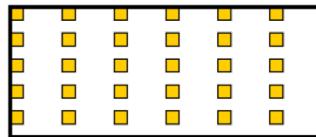
```
prompt> chpl dataParallel.chpl
prompt> ./dataParallel --n=5 --numLocales=4
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```

Other Data Parallel Features

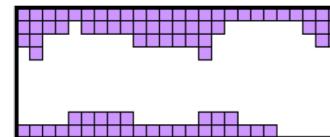
- **Parallel Iterators and Zippering**
- **Slicing:** refer to subarrays using ranges / domains
- **Promotion:** execute scalar functions in parallel using array arguments
- **Reductions:** collapse arrays to scalars or subarrays
- **Scans:** parallel prefix operations
- **Several Domain/Array Types:**



dense



strided



sparse



associative

Chapel Evaluations



Computer Language Benchmarks Game (CLBG)



The Computer Language Benchmarks Game

Which programs are faster?

Will your toy benchmark program be faster if you write it in a different programming language? It depends how you write it!

Ada C Chapel C# C++ Dart
Erlang F# Fortran Go Hack
Haskell Java JavaScript Lisp Lua
OCaml Pascal Perl PHP Python
Racket Ruby Rust Smalltalk Swift
 TypeScript

Which are fast? Trust, and verify

{ for researchers }

Website supporting cross-language comparisons

- 10 toy benchmark programs ×
- ~27 languages ×
 - several implementations
- specific approach prescribed

Chapel's approach to the CLBG:

- striving for elegance over heroism
 - ideally: “Want to learn how program xyz works?
Read the Chapel version.”

CLBG: Website

CRAY

Can sort results by various metrics: execution time, code size, memory use, CPU use:

The Computer Language Benchmarks Game						
pidigits						
description						
program source code, command-line and measurements						
x	source	secs	mem	gz	cpu	cpu load
1.0	<u>Chapel #2</u>	1.62	6,484	423	1.63	99% 1% 1% 2%
1.0	<u>Chapel</u>	1.62	6,488	501	1.63	99% 1% 1% 1%
1.1	<u>Free Pascal #3</u>	1.73	2,428	530	1.72	0% 2% 100% 1%
1.1	<u>Rust #3</u>	1.74	4,488	1366	1.74	1% 100% 1% 0%
1.1	<u>Rust</u>	1.74	4,616	1420	1.74	1% 100% 1% 0%
1.1	<u>Rust #2</u>	1.74	4,636	1306	1.74	1% 100% 0% 0%
1.1	<u>C gcc</u>	1.75	2,728	452	1.74	1% 2% 0% 100%
1.1	<u>Ada 2012 GNAT #2</u>	1.75	4,312	1068	1.75	1% 0% 100% 0%
1.1	<u>Swift #2</u>	1.76	8,492	601	1.76	1% 100% 1% 0%
1.1	<u>Lisp SBCL #4</u>	1.79	20,196	940	1.79	1% 2% 1% 100%
1.2	<u>C++ g++ #4</u>	1.89	4,284	513	1.88	5% 0% 1% 100%
1.3	<u>Go #3</u>	2.04	8,976	603	2.04	1% 0% 100% 0%
1.3	<u>PHP #5</u>	2.12	10,664	399	2.11	100% 0% 1% 1%
1.3	<u>PHP #4</u>	2.12	10,512	389	2.12	100% 0% 0% 2%

The computer Language Benchmarks Game						
pidigits						
description						
program source code, command-line and measurements						
x	source	secs	mem	gz	cpu	cpu load
1.0	<u>Perl #4</u>	3.50	7,348	261	3.50	100% 1% 1% 1%
1.5	<u>Python 3 #2</u>	3.51	10,500	386	3.50	1% 1% 0% 100%
1.5	<u>PHP #4</u>	2.12	10,512	389	2.12	100% 0% 0% 2%
1.5	<u>Perl #2</u>	3.83	7,320	389	3.83	2% 1% 100% 1%
1.5	<u>PHP #5</u>	2.12	10,664	399	2.11	100% 0% 1% 1%
1.6	<u>Chapel #2</u>	1.62	6,484	423	1.63	99% 1% 1% 2%
1.7	<u>C gcc</u>	1.75	2,728	452	1.74	1% 2% 0% 100%
1.7	<u>Racket</u>	27.58	124,156	453	27.56	100% 0% 0% 100%
1.8	<u>OCaml #5</u>	6.72	19,836	458	6.71	1% 2% 0% 100%
1.8	<u>Perl</u>	15.45	10,876	463	15.44	0% 81% 19% 1%
1.9	<u>Ruby #5</u>	3.29	277,496	485	6.58	8% 63% 32% 100%
1.9	<u>Lisp SBCL #3</u>	11.99	325,776	493	11.96	0% 1% 100% 0%
1.9	<u>Chapel</u>	1.62	6,488	501	1.63	99% 1% 1% 1%
1.9	<u>PHP #3</u>	2.14	10,672	504	2.14	1% 0% 0% 100%

gz == code size metric
strip comments and extra whitespace, then gzip

CLBG: Website

CRAY

Can also compare languages pair-wise:

- but only sorted by execution speed...

The Computer Language Benchmarks Game

Chapel versus C++ g++ fastest programs

vs C vs C++ vs Go vs Java vs Python

by faster benchmark performance

reverse-complement

source	secs	mem	gz	cpu	cpu load
Chapel	2.20	1,497,876	707	5.10	96% 42% 58% 38%
C++ g++	2.95	980,472	2280	4.56	50% 41% 16% 50%

pidigits

source	secs	mem	gz	cpu	cpu load
Chapel	1.62	6,488	501	1.63	99% 1% 1% 1%
C++ g++	1.89	4,284	513	1.88	5% 0% 1% 100%

fannkuch-redux

source	secs	mem	gz	cpu	cpu load
Chapel	12.07	4,556	728	48.05	100% 100% 100% 100%
C++ g++	10.62	2,040	980	41.91	100% 95% 100% 100%

The Computer Language Benchmarks Game

Chapel versus Python 3 fastest programs

vs C vs C++ vs Go vs Java vs Python

by faster benchmark performance

mandelbrot

source	secs	mem	gz	cpu	cpu load
Chapel	5.09	36,328	620	20.09	99% 99% 99% 99%
Python 3	279.68	49,344	688	1,117.29	100% 100% 100% 100%

spectral-norm

source	secs	mem	gz	cpu	cpu load
Chapel	3.97	5,488	310	15.75	99% 99% 99% 99%
Python 3	193.86	50,556	443	757.23	98% 98% 99% 99%

fannkuch-redux

source	secs	mem	gz	cpu	cpu load
Chapel	12.07	4,556	728	48.05	100% 100% 100% 100%
Python 3	547.23	48,052	950	2,162.70	99% 100% 97% 100%

CLBG: Qualitative Code Comparisons

CRAY

Can also browse program source code (*but this requires actual thought!*):

```
proc main() {
    printColorEquations();

    const group1 = [i in 1..popSize1] new Chameneos(i, ((i-1)*3):Color);
    const group2 = [i in 1..popSize2] new Chameneos(i, colors10[i]);

    cobegin {
        holdMeetings(group1, n);
        holdMeetings(group2, n);
    }

    print(group1);
    print(group2);

    for c in group1 do delete c;
    for c in group2 do delete c;
}

// Print the results of getNewColor() for all color pairs.
//
proc printColorEquations() {
    for c1 in Color do
        for c2 in Color do
            writeln(c1, " + ", c2, " -> ", getNewColor(c1, c2));
    writeln();
}

// Hold meetings among the population by creating a shared meeting
// place, and then creating per-chameneos tasks to have meetings.
//
proc holdMeetings(population, numMeetings) {
    const place = new MeetingPlace(numMeetings);

    coforall c in population do // create a task per chameneos
        c.haveMeetings(place, population);

    delete place;
}
```

excerpt from 1210.gz Chapel entry

```
void get_affinity(int* is_smp, cpu_set_t* affinity1, cpu_set_t* affinity2)
{
    cpu_set_t
    FILE*
    char
    char const*
    int
    int
    int
    int
    int
    size_t
    size_t

    active_cpus;
    f;
    buf [2048];
    pos;
    cpu_idx;
    physical_id;
    core_id;
    cpu_cores;
    apic_id;
    cpu_count;
    i;

    char const*
    size_t
    char const*
    size_t
    char const*
    size_t
    char const*
    size_t
    char const*
    size_t

    processor_str      = "processor";
    processor_str_len = strlen(processor_str);
    physical_id_str   = "physical id";
    physical_id_str_len = strlen(physical_id_str);
    core_id_str        = "core id";
    core_id_str_len   = strlen(core_id_str);
    cpu_cores_str     = "cpu cores";
    cpu_cores_str_len = strlen(cpu_cores_str);

    CPU_ZERO(&active_cpus);
    sched_getaffinity(0, sizeof(active_cpus), &active_cpus);
    cpu_count = 0;
    for (i = 0; i != CPU_SETSIZE; i += 1)
    {
        if (CPU_ISSET(i, &active_cpus))
        {
            cpu_count += 1;
        }
    }

    if (cpu_count == 1)
    {
        is_smp[0] = 0;
        return;
    }

    is_smp[0] = 1;
    CPU_ZERO(affinity1);
```

excerpt from 2863.gz C gcc entry

CLBG: Qualitative Code Comparisons

CRAY

Can also browse program source code (*but this requires actual thought!*):

```
proc main() {
    printColorEquations();

    const group1 = [i in 1..popSize1] new Chameneos(i, 0);
    const group2 = [i in 1..popSize2] new Chameneos(i, 0);

    cobegin {
        holdMeetings(group1, n);
        holdMeetings(group2, n);
    }

    print(group1);
    print(group2);

    for c in group1 do delete c;
    for c in group2 do delete c;
}

// Print the results of getNewColor() for all colors
// proc printColorEquations() {
//     for c1 in Color do
//         for c2 in Color do
//             writeln(c1, " + ", c2, " = ", getNewColor(c1, c2));
//             writeln();
// }

// Hold meetings among the population by creating a shared
// place, and then creating per-chameneos tasks to have
// them meet
proc holdMeetings(population, numMeetings) {
    const place = new MeetingPlace(numMeetings);

    coforall c in population do // create a task
        c.haveMeetings(place, population);

    delete place;
}
```

```
void get_affinity(int* is_smp, cpu_set_t* affinity1, cpu_set_t* affinity2)
{
    active_cpus;
    f;
    buf [2048];
    pos;
    cpu_idx;
    physical_id;
    core_id;
    cpu_cores;
    apic_id;
    cpu_count;
    i;

    processor_str      = "processor";
    processor_str_len = strlen(processor_str);
    physical_id_str   = "physical id";
    physical_id_str_len = strlen(physical_id_str);
    core_id_str        = "core id";
    core_id_str_len   = strlen(core_id_str);
    cores              = "cores";
    cpu_cores_str      = "cpu_cores";
    cpu_cores_str_len = strlen(cpu_cores_str);
}
```

```
proc holdMeetings(population, numMeetings) {
    const place = new MeetingPlace(numMeetings);

    coforall c in population do // create a task
        c.haveMeetings(place, population);

    delete place;
}
```

```
is_smp[0] = 1;
CPU_ZERO(affinity1);
```

excerpt from 1210.gz Chapel entry

excerpt from 2863.gz C gcc entry

CLBG: Qualitative Code Comparisons

CRAY

Can also browse program source code (*but this requires actual thought!*):

```
proc main() {
    char const* core_id_str = "core id";
    size_t core_id_str_len = strlen(core_id_str);
    char const* cpu_cores_str = "cpu cores";
    size_t cpu_cores_str_len = strlen(cpu_cores_str);

    CPU_ZERO(&active_cpus);
    sched_getaffinity(0, sizeof(active_cpus), &active_cpus);
    cpu_count = 0;
    for (i = 0; i != CPU_SETSIZE; i += 1)
    {
        if (CPU_ISSET(i, &active_cpus))
        {
            cpu_count += 1;
        }
    }

    if (cpu_count == 1)
    {
        is_smp[0] = 0;
        return;
    }
}
```

excerpt from 1210.gz Chapel entry

```
void get_affinity(int* is_smp, cpu_set_t* affinity1, cpu_set_t* affinity2)
{
    cpu_set_t active_cpus;
    FILE* f;
    char buf [2048];
    pos;
    cpu_idx;
    physical_id;
    core_id;
    cpu_cores;
    apic_id;
    cpu_count;
    i;

    char const* processor_str = "processor";
    size_t processor_str_len = strlen(processor_str);
    char const* physical_id_str = "physical id";
    size_t physical_id_str_len = strlen(physical_id_str);

    core_id_str = "core id";
    core_id_str_len = strlen(core_id_str);
    cpu_cores_str = "cpu cores";
    cpu_cores_str_len = strlen(cpu_cores_str);

    CPU_ZERO(&active_cpus);
    sched_getaffinity(0, sizeof(active_cpus), &active_cpus);
    cpu_count = 0;
    for (i = 0; i != CPU_SETSIZE; i += 1)
    {
        if (CPU_ISSET(i, &active_cpus))
        {
            cpu_count += 1;
        }
    }

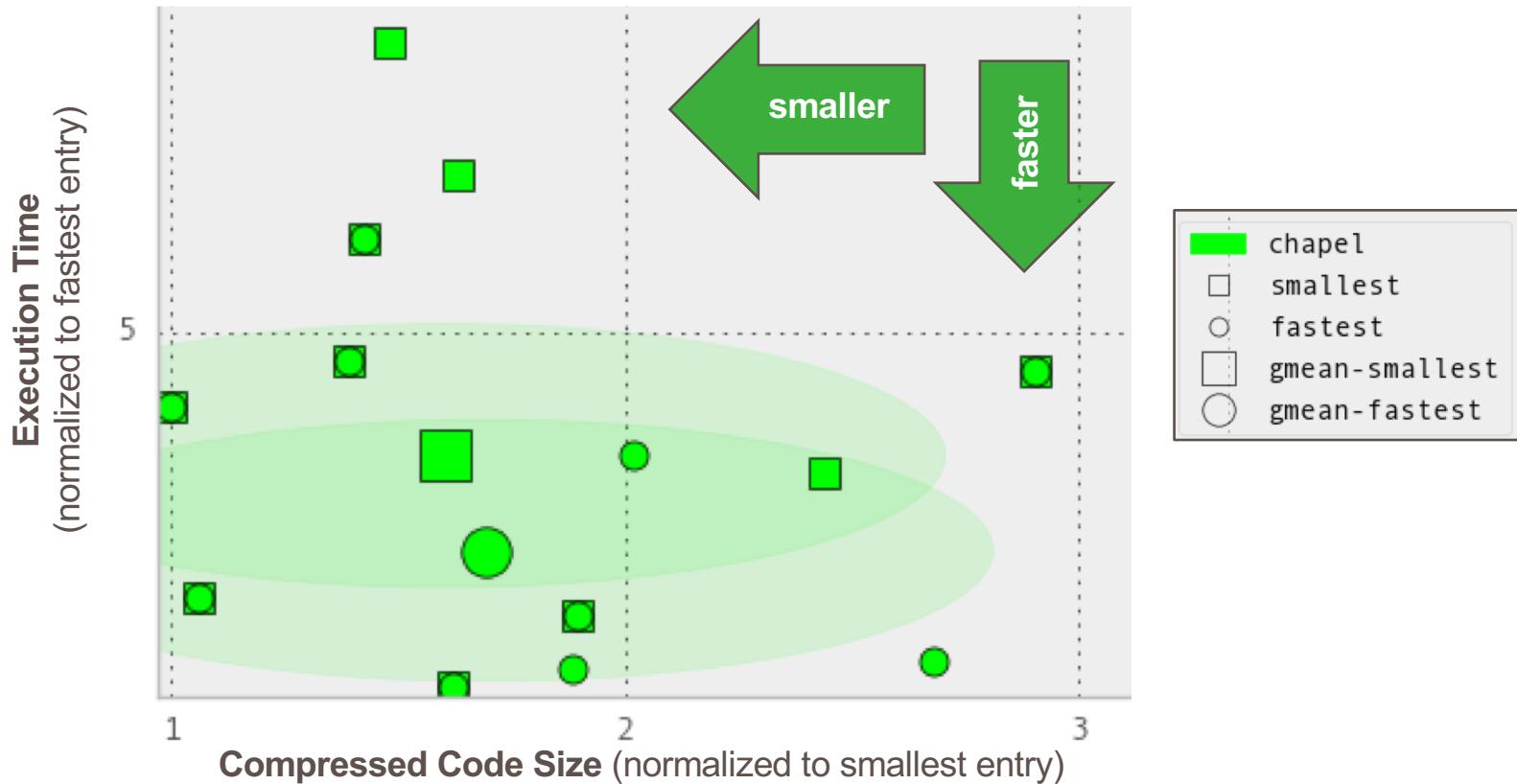
    if (cpu_count == 1)
    {
        is_smp[0] = 1;
        return;
    }
}

is_smp[0] = 1;
CPU_ZERO(affinity1);
```

excerpt from 2863.gz C gcc entry

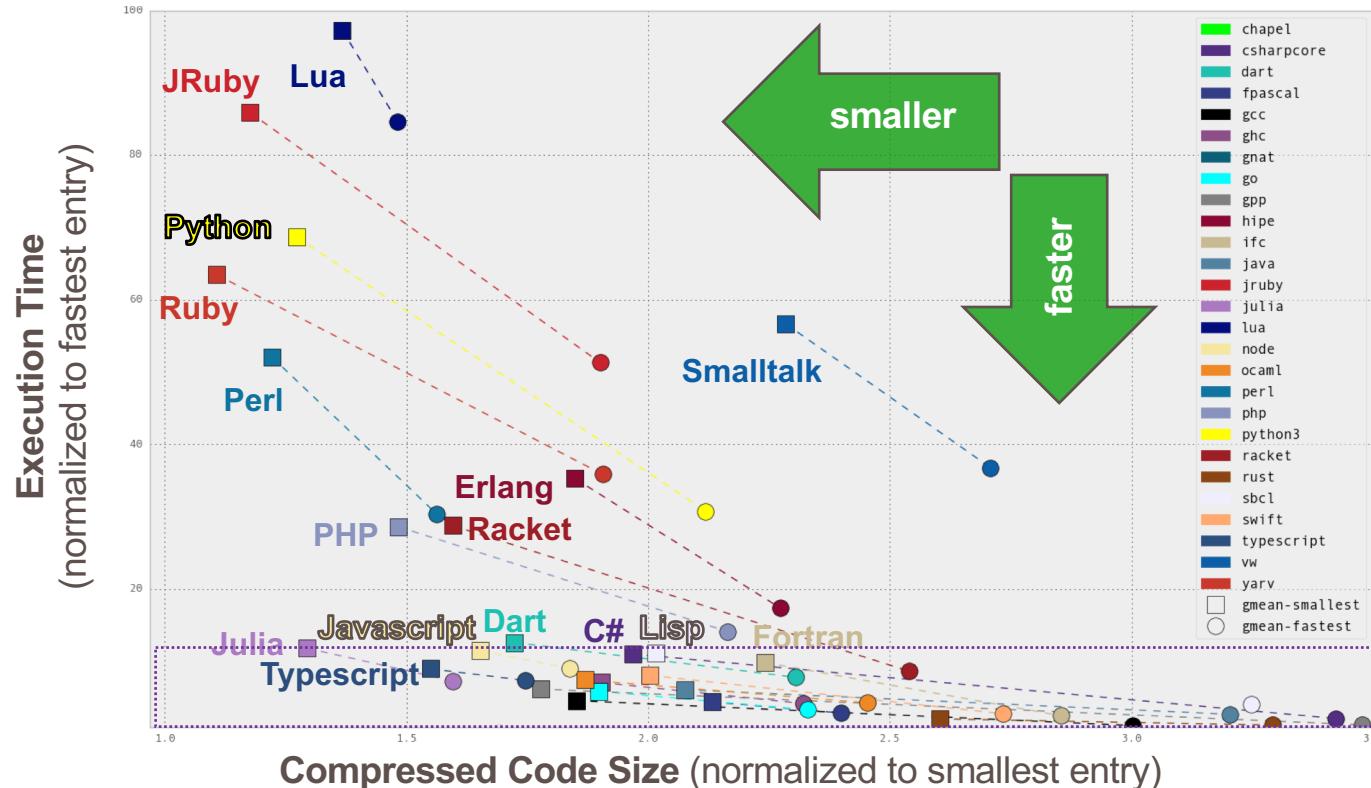
CLBG: Chapel Entries (May 14, 2019)

CRAY



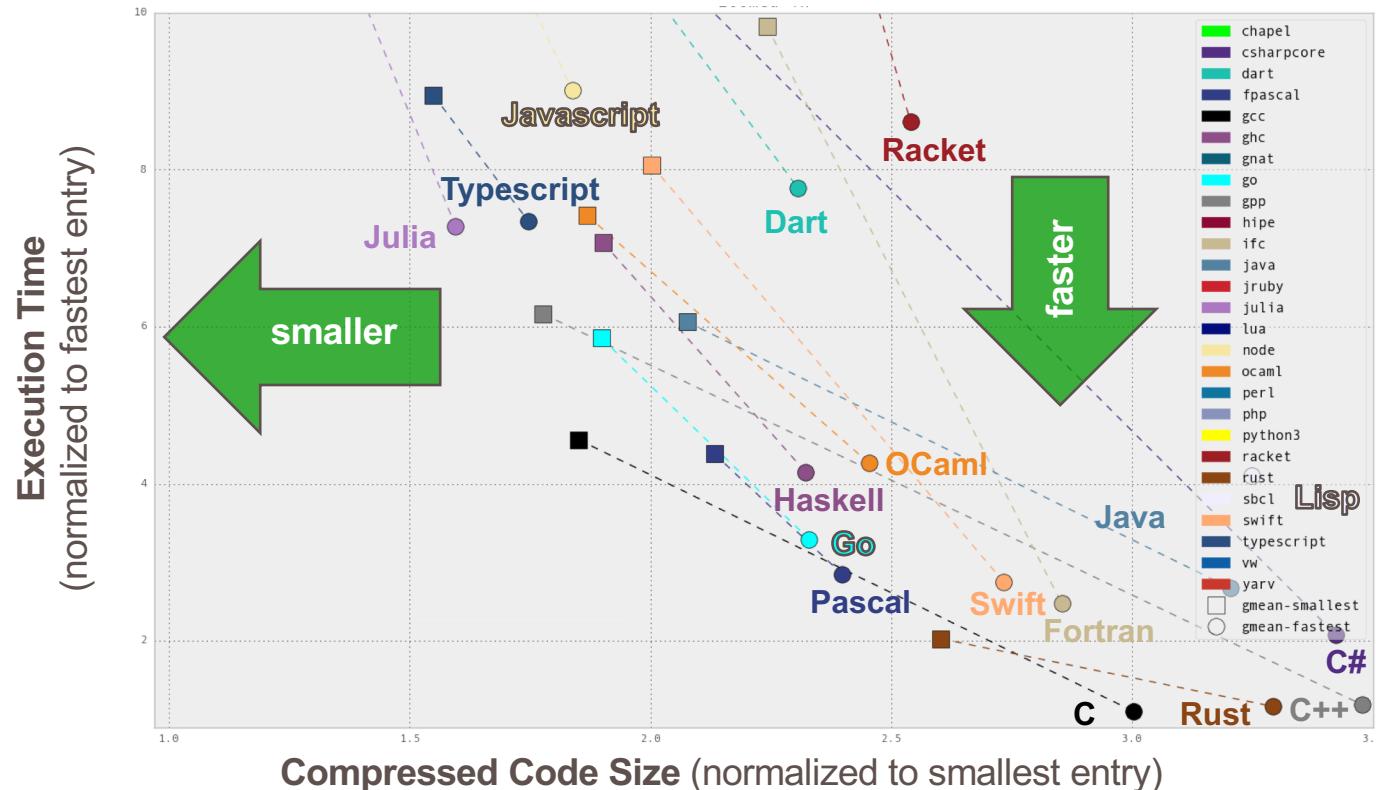
CLBG Cross-Language Summary (May 14, 2019)

CRAY



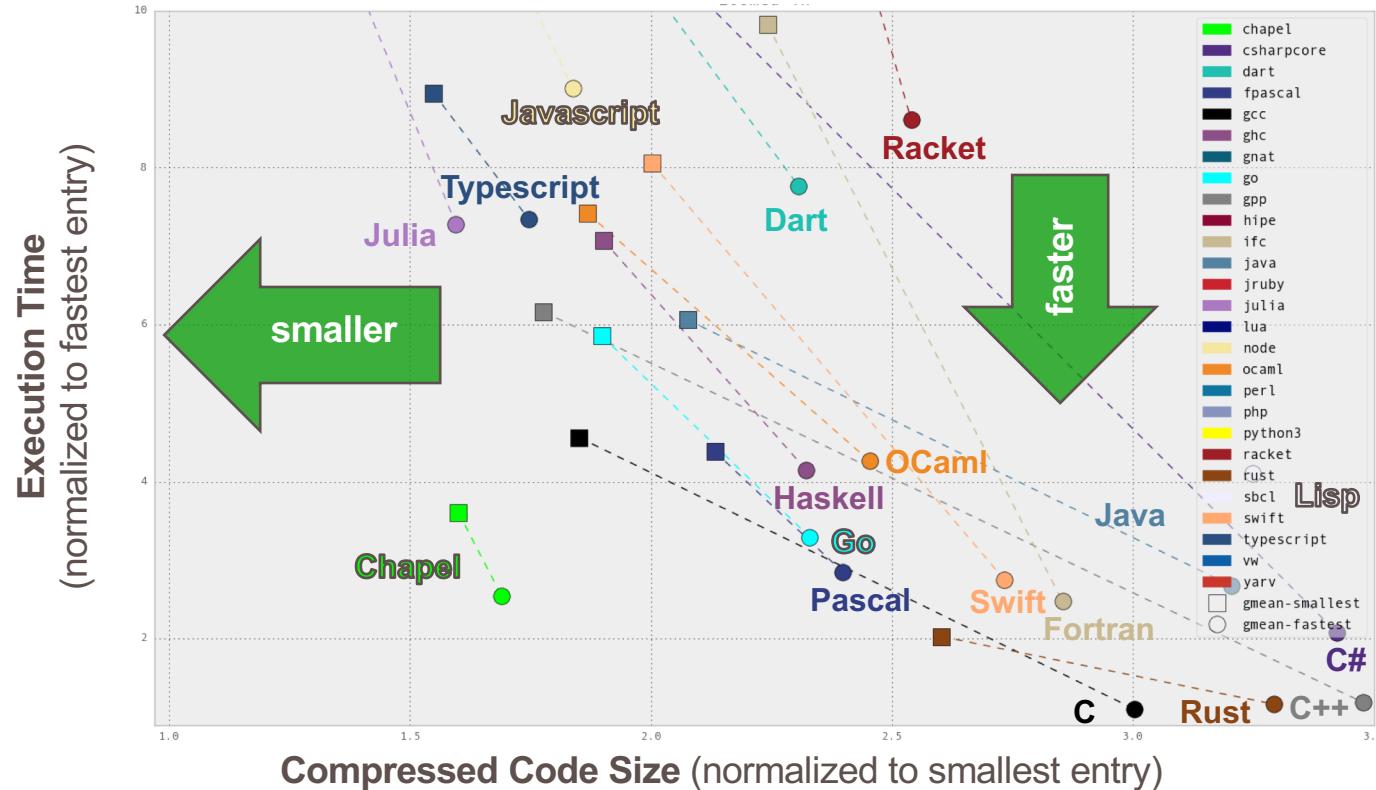
CLBG Cross-Language Summary (May 14, 2019, zoomed)

CRAY



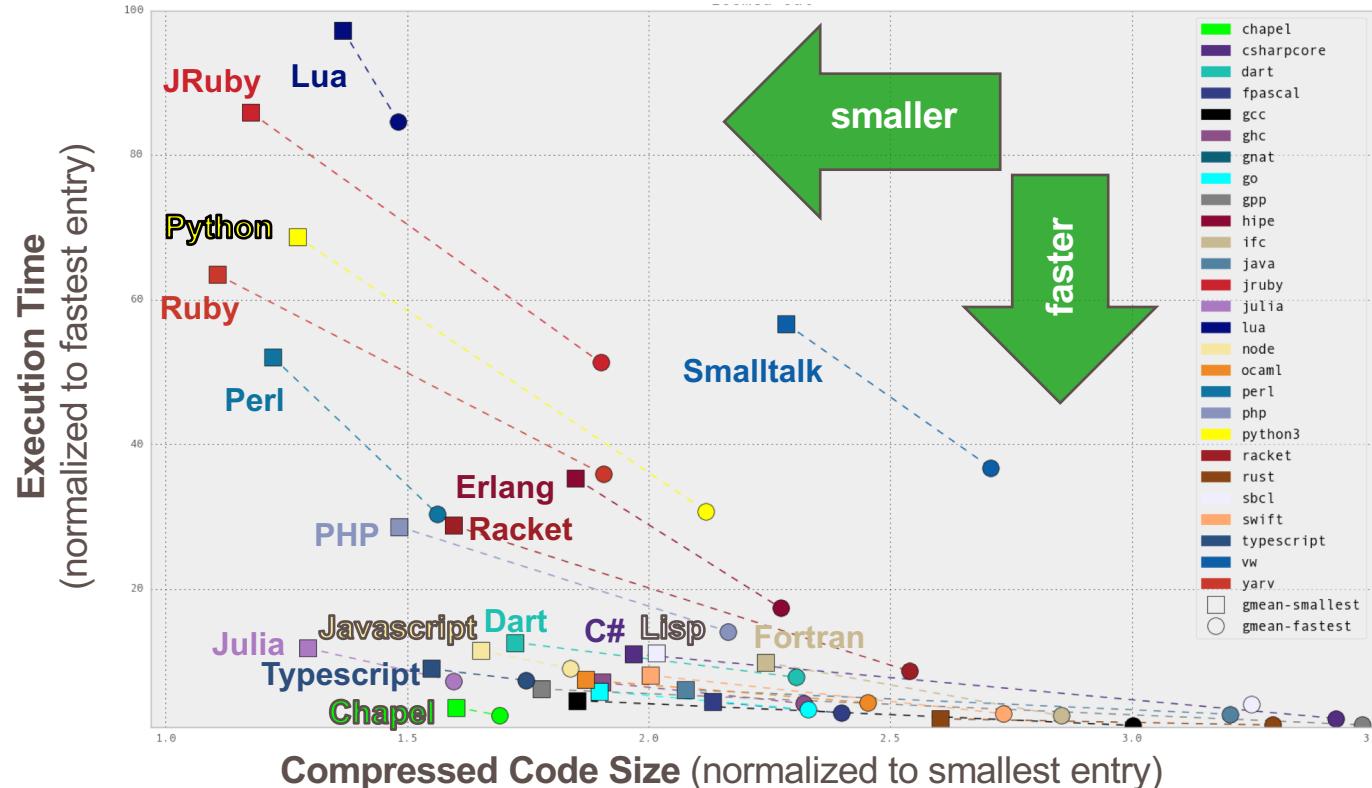
CLBG Cross-Language Summary (May 14, 2019, zoomed)

CRAY



CLBG Cross-Language Summary (May 14, 2019)

CRAY



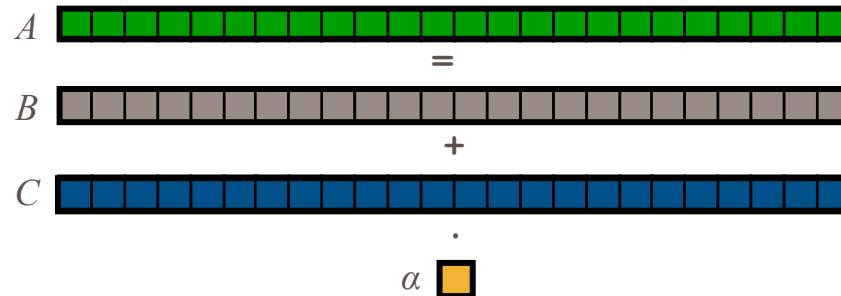
STREAM Triad: a trivial parallel computation

CRAY

Given: m -element vectors A, B, C

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

In pictures:



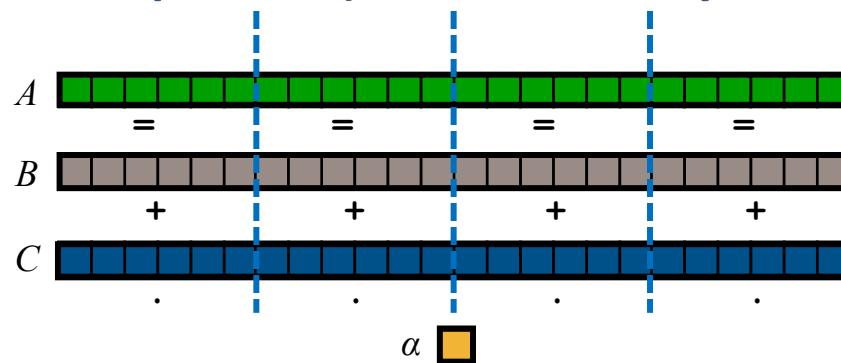
STREAM Triad: a trivial parallel computation

CRAY

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 (shared memory / multicore):



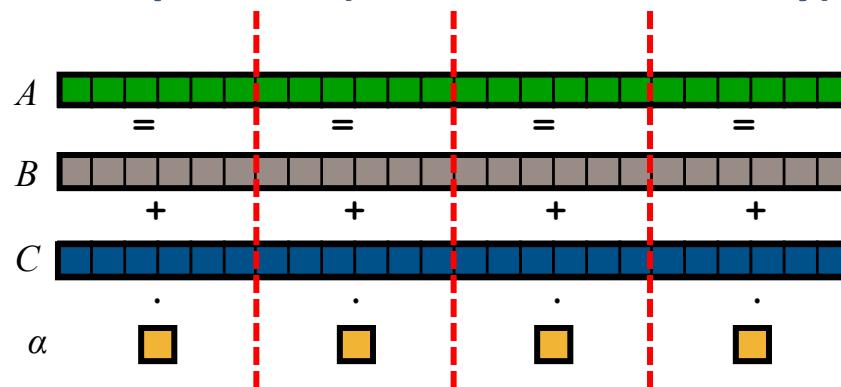
STREAM Triad: a trivial parallel computation

CRAY

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):



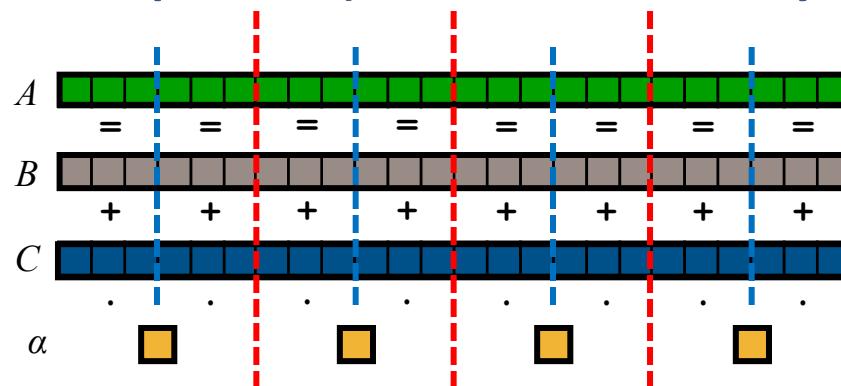
STREAM Triad: a trivial parallel computation

CRAY

Given: m -element vectors A, B, C

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

In pictures, in parallel (distributed memory multicore):



STREAM Triad: C + MPI

CRAY

```
#include <hpcc.h>

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

int HPCC_StarStream(HPCC_Parms *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_Parms *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] = 1.0;
    }
    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);

    return 0;
}
```

STREAM Triad: C + MPI + OpenMP



```
#include <hpcc.h>
#ifndef _OPENMP
#include <omp.h>
#endif

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

int HPCC_StarStream(HPCC_Parms *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_Parms *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] = 1.0;
    }
    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;
}
```

STREAM Triad: Chapel

CRAY

```
#include <hpcc.h>
#ifndef _OPENMP
#include <omp.h>
#endif

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

int HPCC_StarStream(HPCC_Parms *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_IN
```

```
use ...;

config const m = 1000,
        alpha = 3.0;

const ProblemSpace = {1..m} dmapped ...;

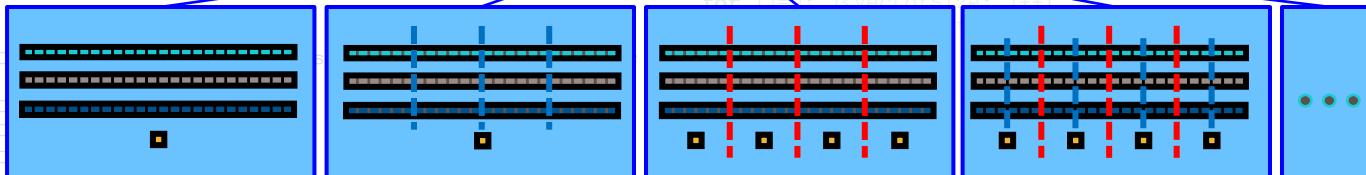
var A, B, C: [ProblemSpace] real;

B = 2.0;
C = 1.0;

A = B + alpha * C;
```

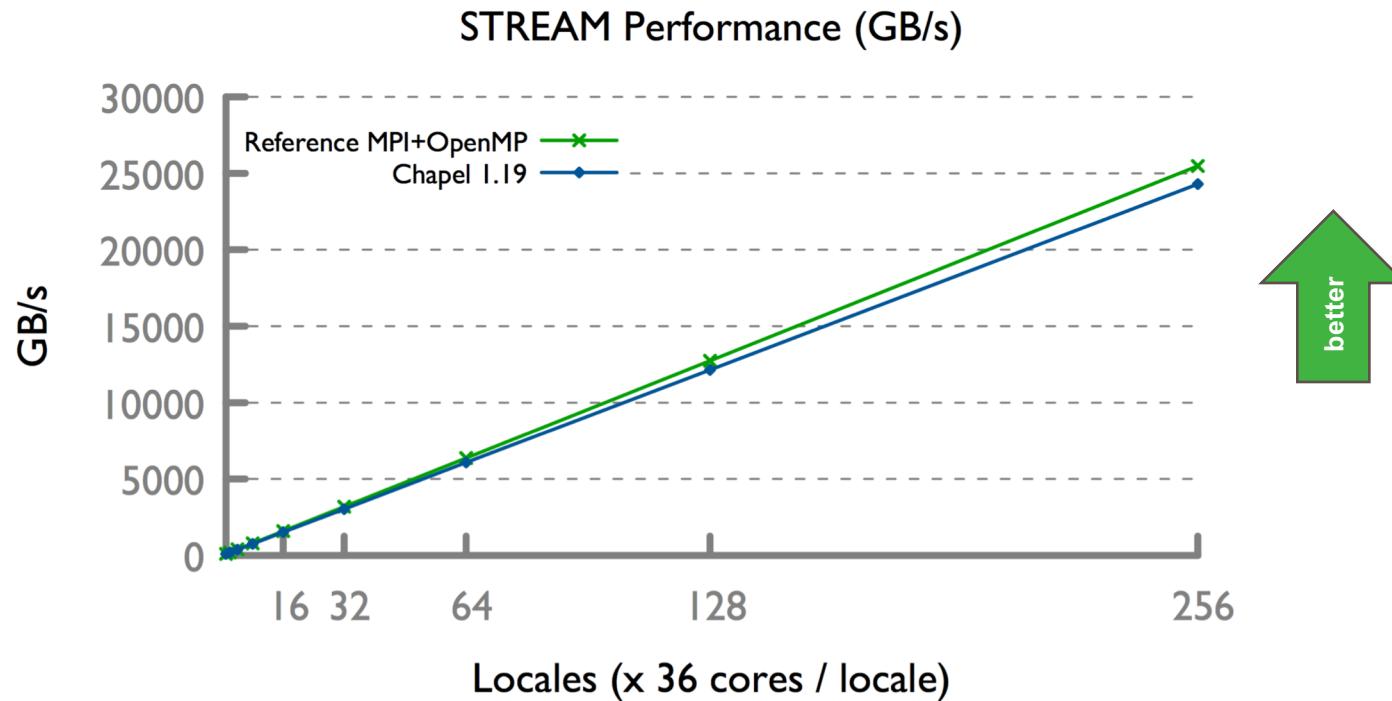
The special sauce:
How should this index set—and any arrays and computations over it—be mapped to the system?

```
int HPCC_Stream(HPCC_Parms *params, int doIO) {
    register int j;
    double scalar;
    VectorSize = HPCC_XMAXLOC;
    a = HPCC_XMALLOC();
    b = HPCC_XMALLOC();
    c = HPCC_XMALLOC();
    #pragma offload target(mpi) a
    #endif
    for (j=0; j < VectorSize; j++) {
        a[j] = b[j] + c[j];
    }
}
```



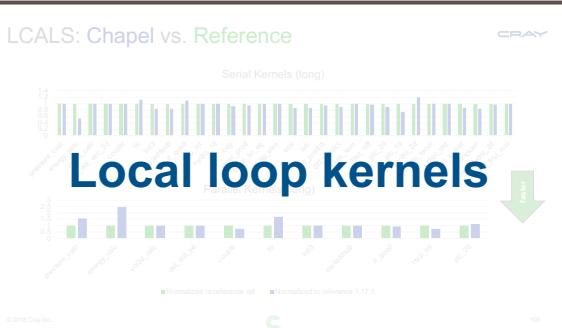
HPCC STREAM Triad: Chapel vs. C+MPI+OpenMP

CRAY



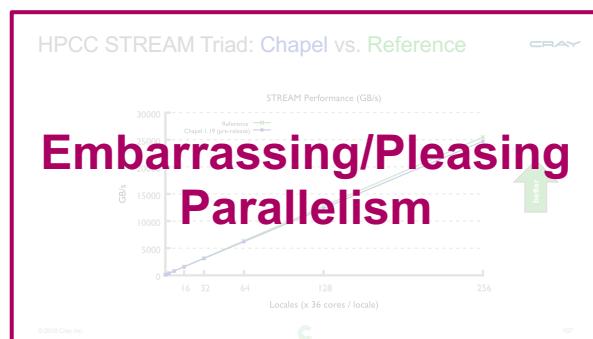
HPC Patterns: Chapel vs. Reference

CRAY



LCALS

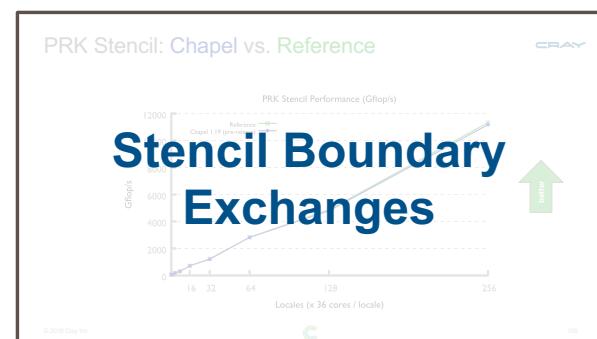
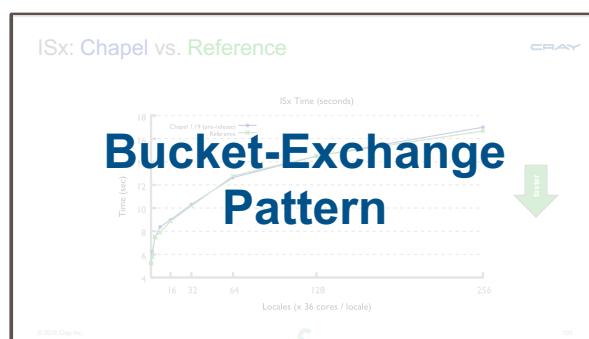
HPCC RA



STREAM
Triad

ISx

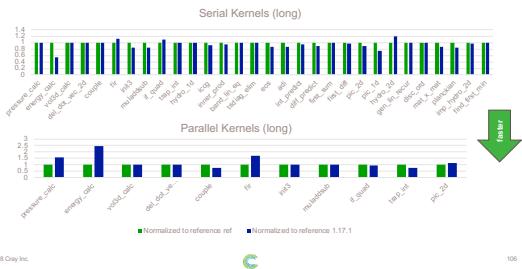
PRK
Stencil



HPC Patterns: Chapel vs. Reference

CRAY

LCALS: Chapel vs. Reference



LCALS

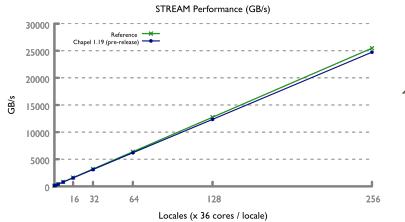
HPCC RA

STREAM
Triad

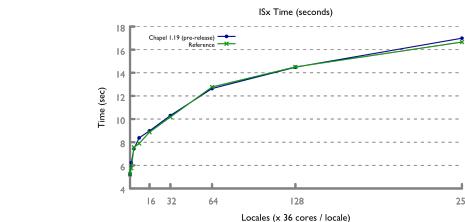
ISx

PRK
Stencil

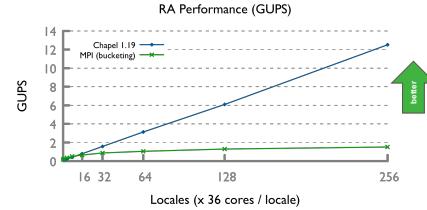
HPCC STREAM Triad: Chapel vs. Reference



ISx: Chapel vs. Reference



HPCC RA: Chapel vs. C+MPI



Summary and Resources



Summarizing this Talk

CRAY

Chapel cleanly and orthogonally supports...

...expression of parallelism and locality

...specifying how to map computations to the system

Chapel is powerful:

- supports succinct, straightforward code
- can result in performance that competes with (or beats) C+MPI+OpenMP

Chapel Central

CRAY

<https://chapel-lang.org>

- download Chapel
- presentations
- papers
- resources
- documentation



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
- **productive:** 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:

```
use CyclicDist;           // use the Cyclic distribution library
config const n = 100;      // use --n=<val> when executing to override this default
forall i in {1..n} dmapped Cyclic(startIdx=1) do
    writeln("Hello from iteration ", i, " of ", n, " running on node ", here.id);
```

What's Hot?

- Chapel 1.17 is now available—[download](#) a copy or browse its [release notes](#)
- The [advance program](#) for CHI UW 2018 is now available—hope to see you there!
- Chapel is proud to be a [Rails Girls Summer of Code 2018 organization](#)
- Watch talks from [ACCU 2017](#), [CHI UW 2017](#), and [ATPESC 2016](#) on [YouTube](#)
- [Browse slides](#) from SIAM PP18, NWCPP, SeaLang, SC17, and other recent talks
- Also see: [What's New?](#)



Chapel Online Documentation

CRAY

<https://chapel-lang.org/docs>: ~200 pages, including primer examples

The screenshot displays the Chapel Online Documentation website, version 1.17. The main navigation bar includes links for "Docs", "Chapel Documentation", "version 1.17 ▾", "Search docs", and "View page source". The sidebar contains links for "COMPILING AND RUNNING CHAPEL" (Quickstart Instructions, Using Chapel, Platform-Specific Notes, Technical Notes, Tools) and "WRITING CHAPEL PROGRAMS" (Quick Reference, Hello World Variants, Primers, Language Specification, Built-in Types and Functions, Standard Modules, Package Modules, Standard Layouts and Distributions, Chapel Users Guide (WIP)). The main content area shows the "Chapel Documentation" page, which includes sections for "Compiling and Running Chapel" (with a list of sub-topics like Quickstart Instructions, Using Chapel, etc.) and "Writing Chapel Programs" (with a list of sub-topics like Quick Reference, Hello World Variants, Primers, Language Specification, etc.). Below these are "Language History" (Chapel Evolution, Archived Language Specifications) and "Platform-Specific Notes" (Technical Notes). A second panel shows the "Using Chapel" page, featuring a "Contents:" sidebar with links to "Using Chapel" sub-topics such as Chapel Prerequisites, Setting up Your Environment for Chapel, Building Chapel, Compiling Chapel Programs, Chapel Man Page, Executing Chapel Programs, Multilocale Chapel Execution, Chapel Launchers, Chapel Tasks, Debugging Chapel Programs, and Reporting Chapel Issues. The third panel shows the "Task Parallelism" page under "Primer" documentation, with sections for "Begin Statements" and "Cobegin Statements", each containing code snippets and descriptions. The footer of the site includes the Cray logo and the text "© Copyright 2018, Cray Inc."

Chapel Community

CRAY

Questions Developer Jobs Tags Users [chapel]

Tagged Questions info newest frequent votes active unanswered

140 questions tagged Ask Question

Chapel is a portable, open-source parallel programming language. Use this tag to ask questions about the Chapel language or its implementation.

Learn more... Improve tag info Top users Synonyms

Tuple Concatenation in Chapel
Let's say I'm generating tuples and I want to concatenate them as they come. How do I do this? The following does element-wise addition: if `ts = ("foo", "cat"), t = ("bar", "dog") ts += t` gives `ts = ...`
tuples concatenation addition hpc chapel asked Jan 26 at 0:30 Tahmangia 385 1 10

Is there a way to use non-scalar values in functions with where clauses in Chapel?
I've been trying out Chapel off and on over the past year or so. I have used C and C++ briefly in the past, but most of my experience is with dynamic languages such as Python, Ruby, and Erlang more ...
chapel angular 33 3 asked Apr 23 at 23:15

Is there any `writeln()` format specifier for a bool?
I looked at the `writeln()` documentation for any bool specifier and there didn't seem to be any. In a Chapel program I have: `... config const verify = false; /* that works but I want to use writeln() ...`
chapel asked Nov 11 '17 at 22:21

<https://stackoverflow.com/questions/tagged/chapel>

This repository Search Pull requests Issues Marketplace Gist

chapel-lang / chapel

Code Issues 292 Pull requests 26 Projects 0 Settings Insights

Filters IsIssue:open Labels Milestones

292 Open 77 Closed

- Implement "bounded-coforall" optimization for remote coforalls area: Compiler type: Performance #6357 opened 13 hours ago by ronawho
- Consider using processor atomics for remote coforalls EndCount area: Compiler type: Performance #6356 opened 13 hours ago by ronawho 0 of 6
- make uninstall area: BTR type: Feature Request #6353 opened 14 hours ago by mppf
- make check doesn't work with ./configure area: BTR #6362 opened 16 hours ago by mppf
- Passing variable via intent to a forall loop seems to create an iteration-private variable, not a task-private one area: Compiler type: Bug #6351 opened a day ago by cassella
- Remove chpl_comm_make_progress area: Runtime easy type: Design #6349 opened a day ago by sungunchoi
- Runtime error after make on Linux Mint area: BTR user issue #6348 opened a day ago by denindiana

<https://github.com/chapel-lang/chapel/issues>

GITTER

chapel-lang/chapel Chapel programming language | Peak developer hours are 0600-1700 PT

Brian Dolan @buddha314 what is the syntax for making a copy (not a reference) to an array? May 09 14:34

Michael Ferguson @mppf like in a new variable? May 09 14:40

```
var A[1..10] int;
var B = A; // makes a copy of A
ref C = A; // refers to A
```

Brian Dolan @buddha314 oh, got it, thanks! May 09 14:41

Michael Ferguson @mppf proc f(xr) { /* xr refers to the actual argument */ }
var g(in xr) { /* xr is a copy of the actual argument */
 var A[1..10] int;
 f(A);
 g(A);

Brian Dolan @buddha314 isn't there a proc f(ref arr) {} as well? May 09 14:43

Michael Ferguson @mppf yes. The default intent for array is 'ref' or 'const ref' depending on if the function body modifies it. So that's effectively the default. May 09 14:55

Brian Dolan @buddha314 thanks! May 09 14:55

<https://gitter.im/chapel-lang/chapel>

read-only mailing list: chapel-announce@lists.sourceforge.net (~15 mails / year)

Chapel Social Media (no account required)

CRAY

[http://twitter.com/ChapelLanguage](https://twitter.com/ChapelLanguage)

[http://facebook.com/ChapelLanguage](https://facebook.com/ChapelLanguage)

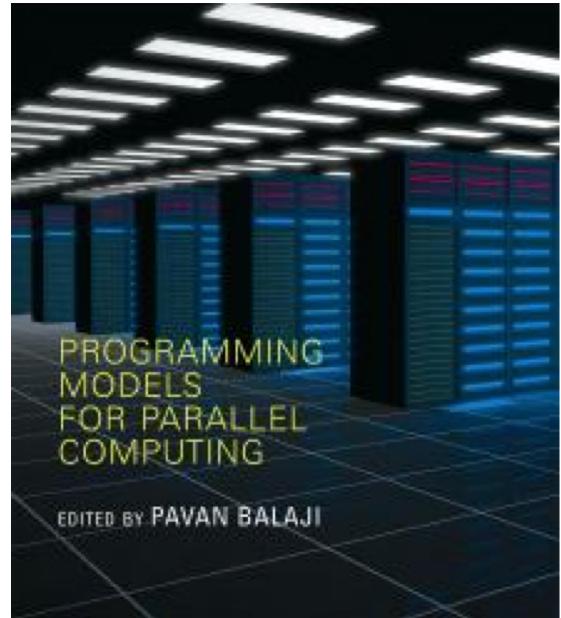
<https://www.youtube.com/channel/UCHmm27bYjhknK5mU7ZzPGsQ/>

Suggested Reading: Chapel history and overview

CRAY

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 also available [online](#)



Suggested Reading: Recent Progress (CUG 2018)

Chapel Comes of Age: Making Scalable Programming Productive

Bradford L. Chamberlain, Elliot Ronaghan, Ben Albrecht, Lydia Duncan, Michael Ferguson,
Ben Hershberger, David Iten, David Keaton, Vassily Litvinov, Preston Sahabu, and Greg Titus
Chapel Team
Cray Inc.
Seattle, WA, USA
chapel_info@cray.com

Abstract—Chapel is a programming language whose goal is to support productive, general-purpose parallel computing at scale. Chapel's approach can be thought of as combining the strengths of Python, Fortran, C/C++, and MPI in a single language. Over years, the DARPA High Productivity Computing Systems (HPCS) program that launched Chapel wrapped up, and the team embarked on a five-year effort to move on our own to apply it to end-users. This paper follows on from our CUG 2016 paper summarizing the progress made by the Chapel project since that time. Specifically, Chapel's performance now competes with or beats hand-coded GPU/FIRETEAM, MPI, LAPACK, MPI+ZMQ, and other key technologies; its documentation has been modernized and fleshed out; and the set of tools available to Chapel users has grown. This paper also characterizes the experiences of contributors from communities as diverse as astrophysics and artificial intelligence.

Keywords—Parallel programming; Computer languages

I. INTRODUCTION

Chapel is a programming language designed to support productive, general-purpose parallel computing at scale. Chapel's approach can be thought of as striving to create a language whose code is as attractive to read and write as Python, yet which supports the performance of Fortran and the scalability of MPI. Chapel also aims to compete with C in terms of portability, and with C++ in terms of flexibility and extensibility. Chapel is designed to be general-purpose in the sense that when you have a parallel algorithm in mind and want to specify exactly how to run it, Chapel should be able to handle that scenario.

Chapel's design and implementation are led by Cray Inc., with feedback and code contributed by users and the open-source community. Though developed by Cray, Chapel's design and implementation are portable, permitting its programs to scale up from multicore laptops to commodity clusters to Cray systems. In addition, Chapel programs can be run on cloud-computing platforms and HPC systems from other vendors. Chapel is being developed in an open-source manner under the Apache 2.0 license and is hosted at GitHub.¹

¹<https://github.com/chapel-lang/chapel>

paper and slides available at chapel-lang.org



The development of the Chapel language was undertaken by Cray Inc. as part of its participation in the DARPA High Productivity Computing Systems program (HPCS). HPCS wrapped up in late 2012, at which point Chapel was a compelling prototype, having successfully demonstrated several key research challenges that the project had undertaken. Chief among these was supporting data- and task-parallelism in a single unified language, the Chapel language. This was accomplished by supporting the creation of highly nested parallel abstractions such as parallel loops and arrays in terms of lower-level Chapel features such as classes, iterators, and tasks.

Under HPCS, Chapel also successfully supported the expression of parallelism using distinct language features from those used to control locality and affinity—that is, Chapel programmers specify which computations should run in parallel and from specifying where those computations should be run. This allows Chapel programs to support multicores, multi-node, and heterogeneous computing within a single unified language.

Chapel's implementation under HPCS demonstrated that the language could be implemented portably while still being optimized for HPC-specific features such as the RDMA support available in Cray® Gemini™ and Aries™ networks. This allows Chapel to take advantage of native hardware support for remote puts, gets, and atomic memory operations.

Despite these successes, at the close of HPCS, Chapel was not at all ready to support production codes in the field. This was not surprising given the language's aggressive design and modest-size research team. However, reactions from potential users were sufficiently positive that, in early 2013, Cray embarked on a follow-up effort to improve Chapel and move it towards being a production-ready language. Colloquially, we refer to this as "the second five-year push." This paper's contribution is to describe the results of this five-year effort, providing readers with an understanding of Chapel's progress and achievements since the end of the HPCS program. In doing so, we directly compare the status of Chapel version 1.17, released last month, with Chapel version 1.7, which was released five years ago in April 2013.

**Chapel Comes of Age:
Productive Parallelism at Scale** 
CUG 2018
Brad Chamberlain, Chapel Team, Cray Inc.

SAFE HARBOR STATEMENT

This presentation may contain forward-looking statements that are based on our current expectations. Forward looking statements may include statements about our financial guidance and expected operating results, our opportunities and future potential, our product development and new product introduction plans, our ability to expand and penetrate our addressable markets and other statements that are not historical facts.

These statements are only predictions and actual results may materially vary from those projected. Please refer to Cray's documents filed with the SEC from time to time concerning factors that could affect the Company and these forward-looking statements.



THANK YOU

QUESTIONS?



bradc@cray.com



@ChapelLanguage



chapel-lang.org



cray.com



@cray_inc



linkedin.com/company/cray-inc-/

