Chapel

Programmability, Parallelism, and Performance

Brad Chamberlain
CSC 2/458, University of Rochester
March 18, 2019


bradc@cray.com

chapel-lang.org

@ChapelLanguage

Who am I?

Education:

- Earned Ph.D. from University of Washington CSE in 2001
  - focused on the ZPL data-parallel array language
- Remain associated with UW CSE as an Affiliate Professor

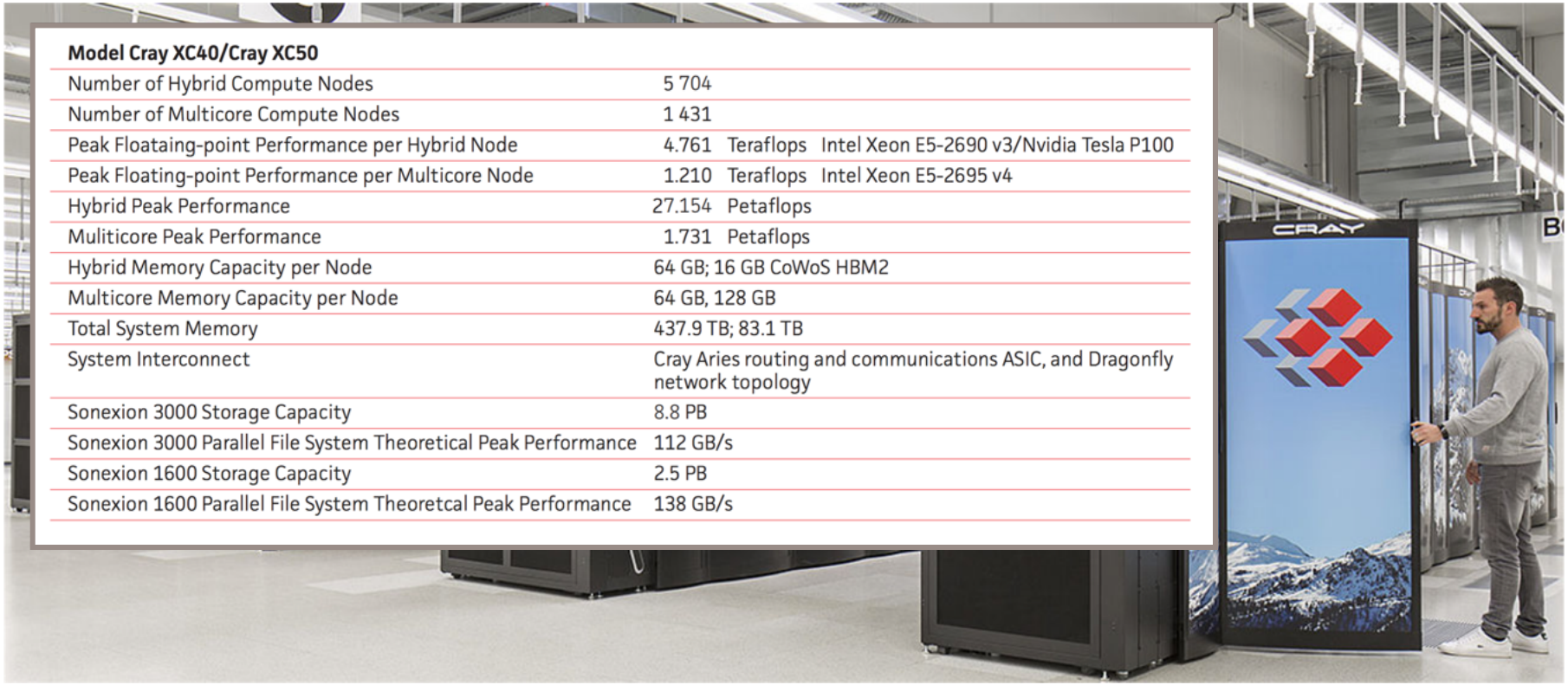
Industry:

- Currently a Principal Engineer at Cray Inc.
- Technical lead / founding member of the Chapel project

Piz Daint: One of Today's Most Powerful Supercomputers

https://www.cscs.ch/computers/piz-daint/

Piz Daint: One of Today's Most Powerful Supercomputers



Model Cray XC40/Cray XC50			
Number of Hybrid Compute Nodes	5 704		
Number of Multicore Compute Nodes	1 431		
Peak Floataing-point Performance per Hybrid Node	4.761	Teraflops	Intel Xeon E5-2690 v3/Nvidia Tesla P100
Peak Floating-point Performance per Multicore Node	1.210	Teraflops	Intel Xeon E5-2695 v4
Hybrid Peak Performance	27.154	Petaflops	
Muliticore Peak Performance	1.731	Petaflops	
Hybrid Memory Capacity per Node	64 GB; 16 GB CoWoS HBM2		
Multicore Memory Capacity per Node	64 GB, 128 GB		
Total System Memory	437.9 TB; 83.1 TB		
System Interconnect	Cray Aries routing and communications ASIC, and Dragonfly network topology		
Sonexion 3000 Storage Capacity	8.8 PB		
Sonexion 3000 Parallel File System Theoretical Peak Performance	112 GB/s		
Sonexion 1600 Storage Capacity	2.5 PB		
Sonexion 1600 Parallel File System Theoretcal Peak Performance	138 GB/s		

https://www.cscs.ch/computers/piz-daint/

Chapel's Origins

DARPA HPCS (2003–2012)

- **HPCS** = High *Productivity* Computing Systems
- **Goal:** Improve HPC productivity by 10x

What is Chapel?

Chapel: A productive 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

Outline

✓ Context for this talk

➢ Chapel and Productivity

- Overview of Chapel Features
- Chapel Results
- Chapel Resources

What does “Productivity” mean to you?

Recent Graduates:

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

Seasoned HPC Programmers:

“that sugary stuff that I don’t need because I ~~was born to suffer~~ want full control to ensure 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.”

Chapel and Productivity

Chapel aims to be as…

…**programmable** as Python

…**fast** as Fortran

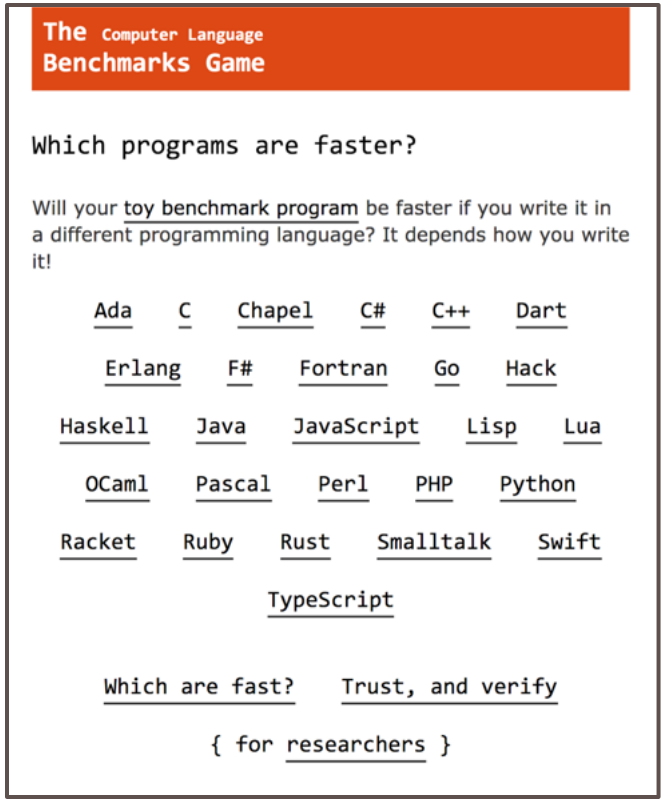
…**scalable** as MPI, SHMEM, or UPC

…**portable** as C

…**flexible** as C++

…**fun** as [your favorite programming language]

Computer Language Benchmarks Game (CLBG)



Website supporting cross-language comparisons

- 10 toy benchmark programs

  x ~27 languages

  x several implementations

  - exercise key computational idioms
  - specific approach prescribed

CLBG: Website

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

×	source	secs	mem	gz	cpu	cpu load
1.0	**Chapel** #2	**1.62**	6,484	423	1.63	99% 1% 1% 2%
1.0	Chapel	1.62	6,488	501	1.63	99% 1% 1% 1%
1.1	Free **Pascal** #3	**1.73**	2,428	530	1.72	0% 2% 100% 1%
1.1	**Rust** #3	**1.74**	4,488	1366	1.74	1% 100% 1% 0%
1.1	Rust	1.74	4,616	1420	1.74	1% 100% 1% 0%
1.1	Rust #2	1.74	4,636	1306	1.74	1% 100% 0% 0%
1.1	**C** gcc	**1.75**	2,728	452	1.74	1% 2% 0% 100%
1.1	**Ada** 2012 GNAT #2	**1.75**	4,312	1068	1.75	1% 0% 100% 0%
1.1	**Swift** #2	**1.76**	8,492	601	1.76	1% 100% 1% 0%
1.1	**Lisp** SBCL #4	**1.79**	20,196	940	1.79	1% 2% 1% 100%
1.2	**C++** g++ #4	**1.89**	4,284	513	1.88	5% 0% 1% 100%
1.3	**Go** #3	**2.04**	8,976	603	2.04	1% 0% 100% 0%
1.3	**PHP** #5	**2.12**	10,664	399	2.11	100% 0% 1% 1%
1.3	PHP #4	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

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

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

CLBG: Website

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%

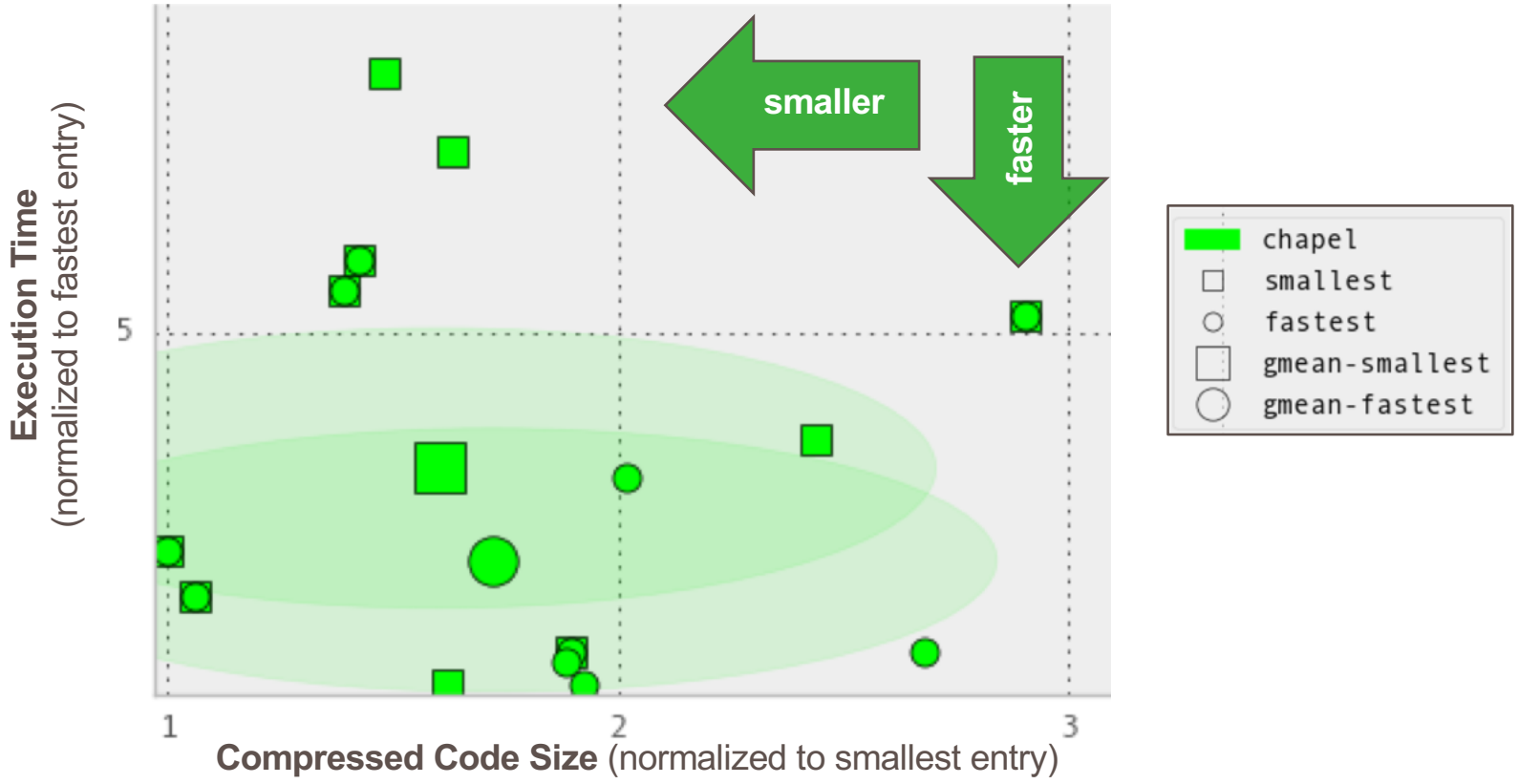
spectral-norm

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%

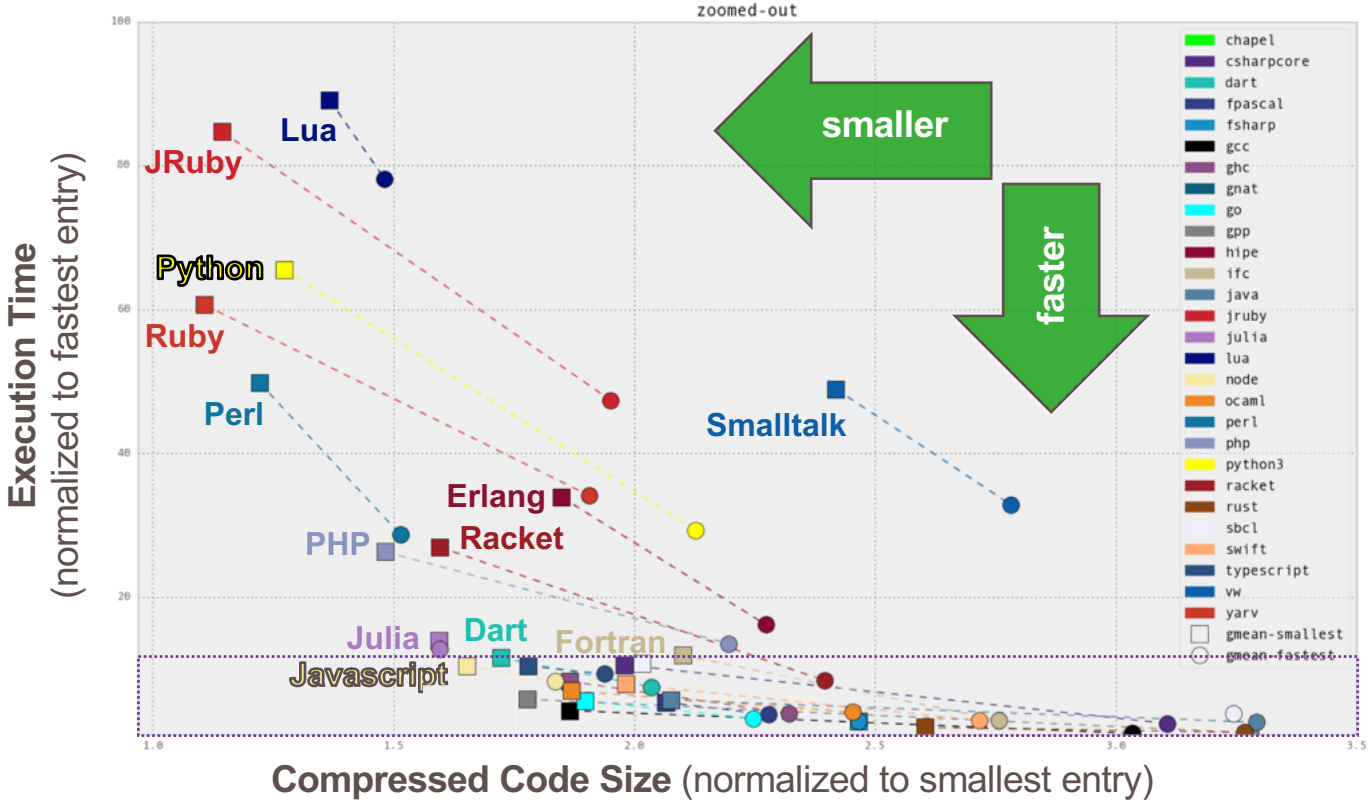
fannkuch-redux

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%

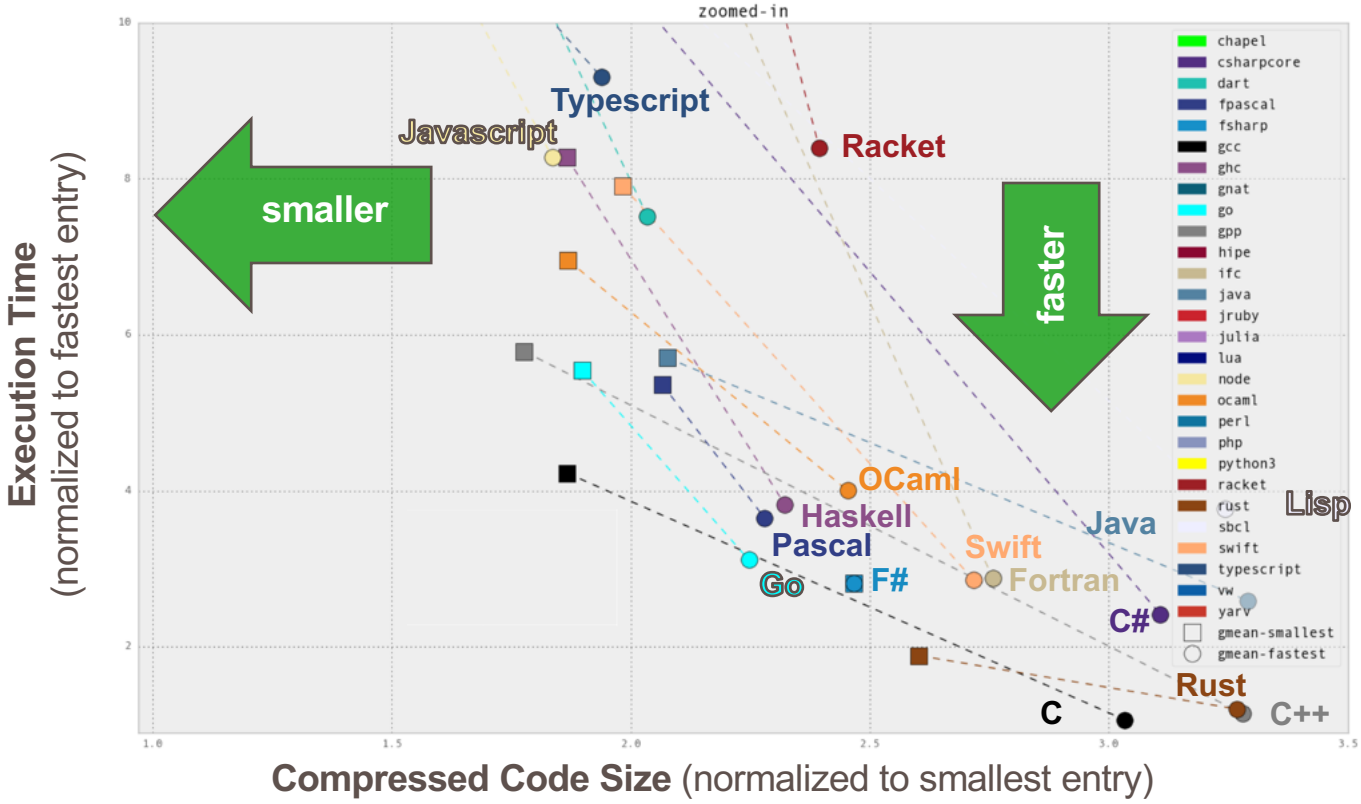
CLBG: Chapel Entries (September 21, 2018)



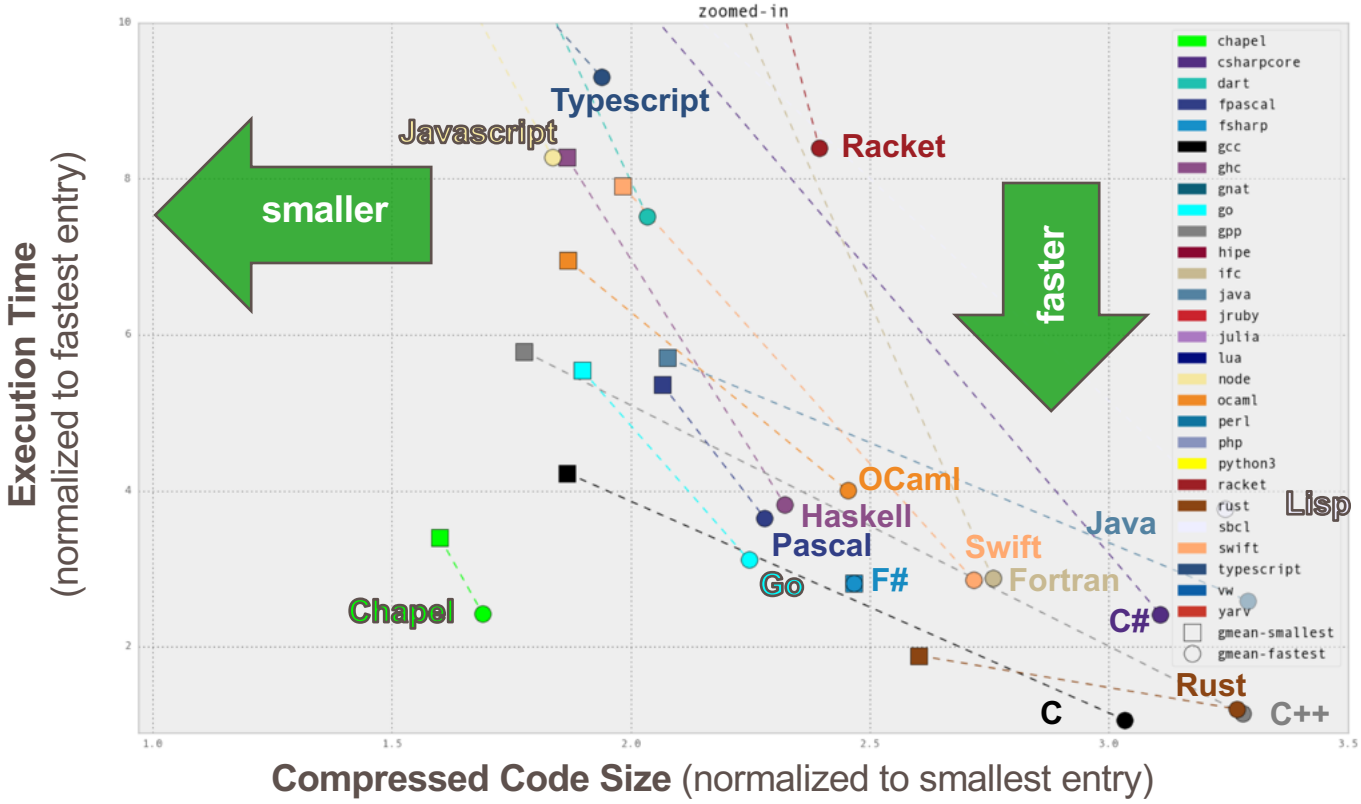
CLBG Cross-Language Summary (Dec 18, 2018)



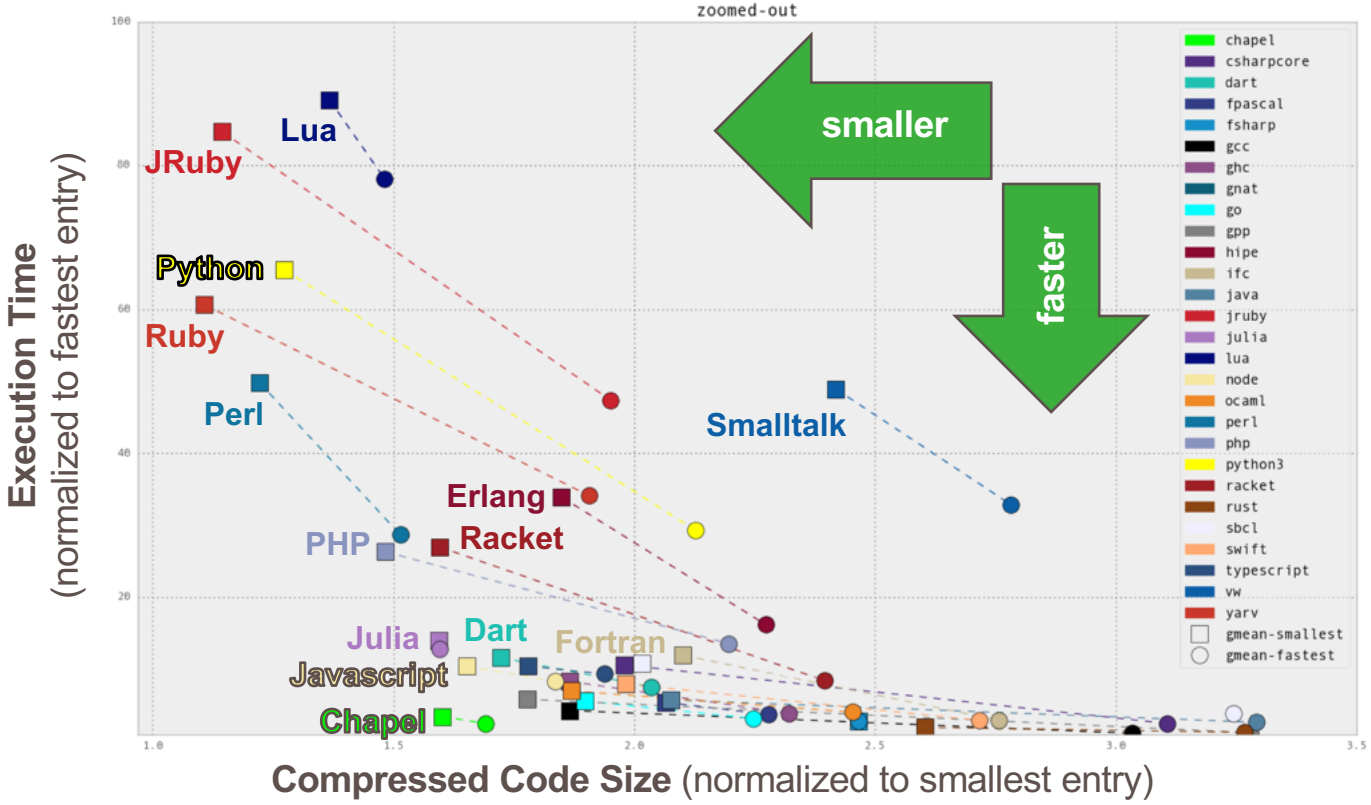
CLBG Cross-Language Summary (Dec 18, 2018, zoomed)



CLBG Cross-Language Summary (Dec 18, 2018, zoomed)



CLBG Cross-Language Summary (Dec 18, 2018)



CLBG: Qualitative Code Comparisons

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;
}
```

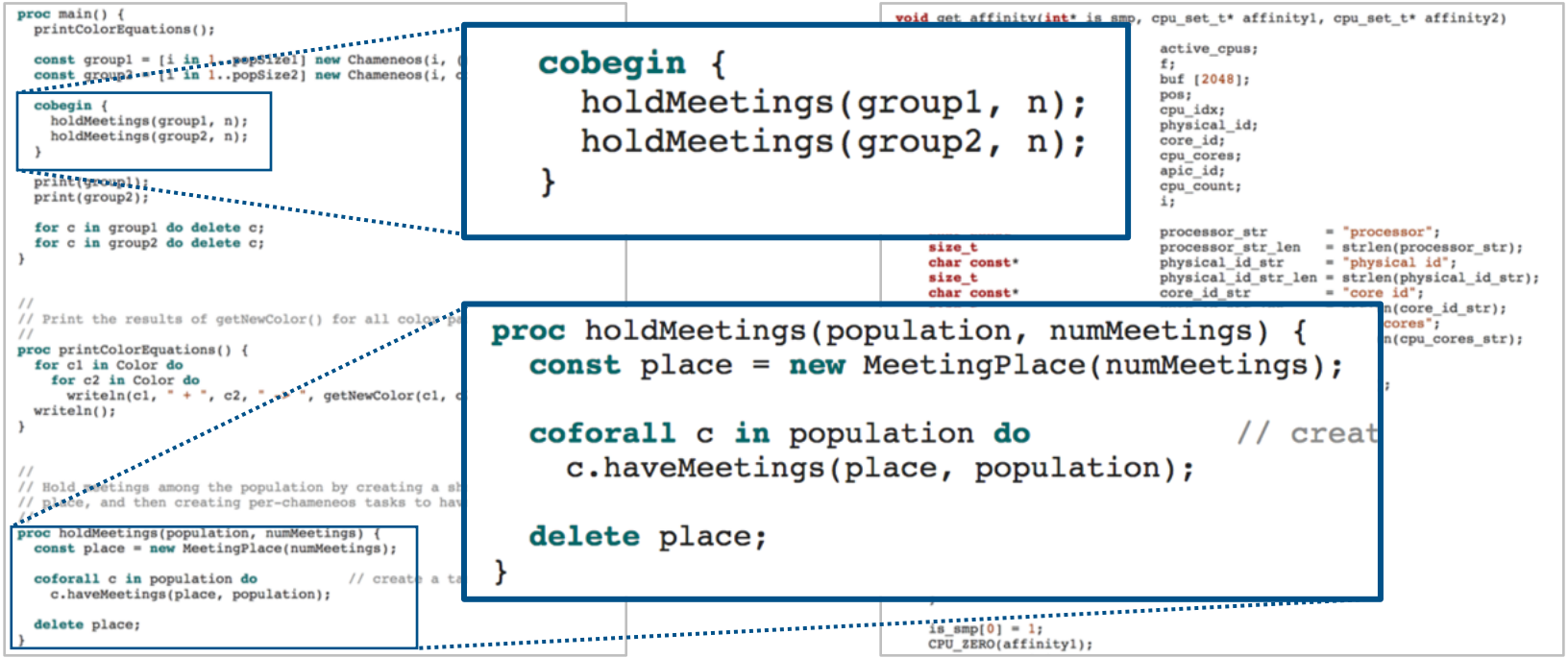
excerpt from 1210 gz Chapel entry

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

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

excerpt from 2863 gz C gcc entry

CLBG: Qualitative Code Comparisons

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



excerpt from 1210 gz Chapel entry

excerpt from 2863 gz C gcc entry

CLBG: Qualitative Code Comparisons

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(co
char const*                     cpu_cores_str        = "cpu core
size_t                          cpu_cores_str_len    = strlen(cp

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;
}
```

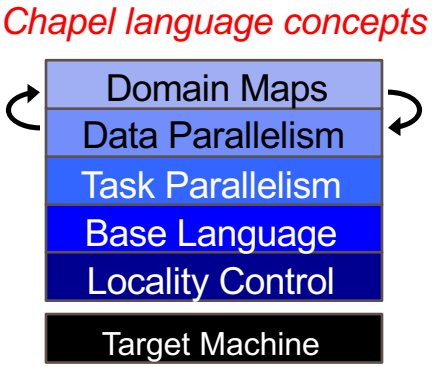
excerpt from 1210 gz Chapel entry

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

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

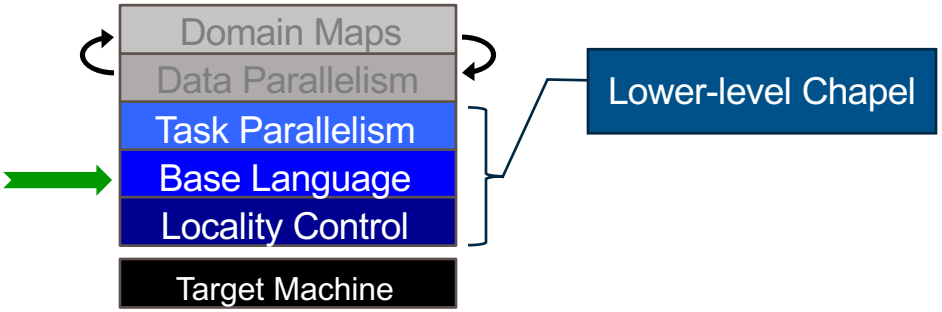
excerpt from 2863 gz C gcc entry

Overview of Chapel Features

Chapel Feature Areas



Base Language



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);
```

```
0
1
1
2
3
5
8
…
```

Base Language Features, by example

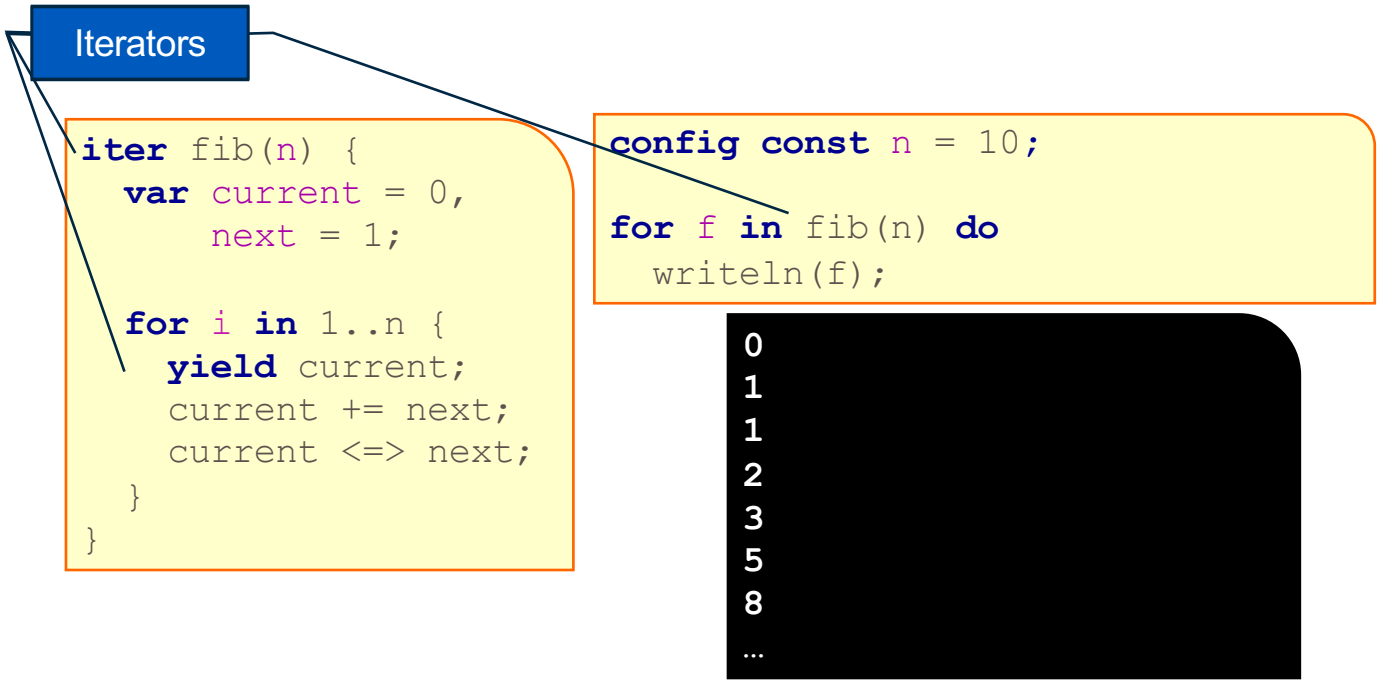
Configuration declarations
(support command-line overrides)
`./fib --n=1000000`

```
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



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


```
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

```
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

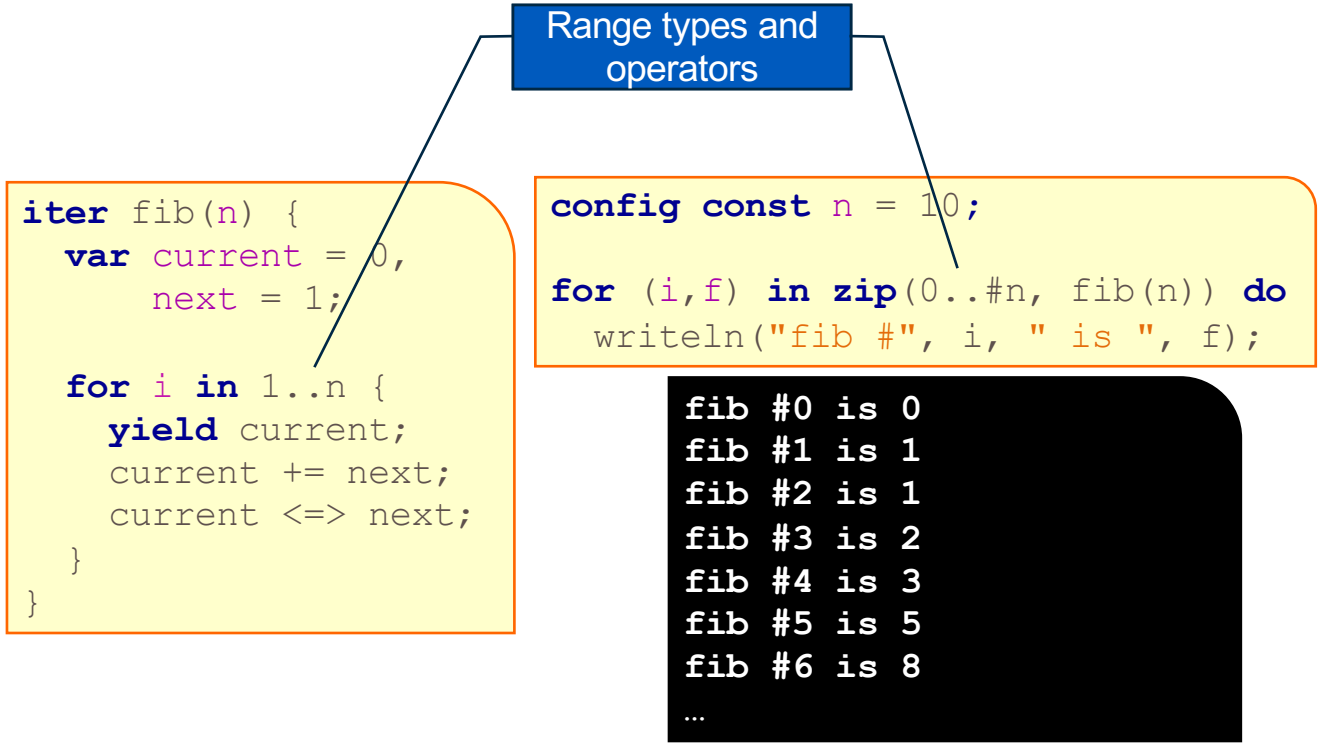
```
iter fib(n) {
  var current = 0,
      next = 1;

  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



Base Language Features, by example

```
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

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

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

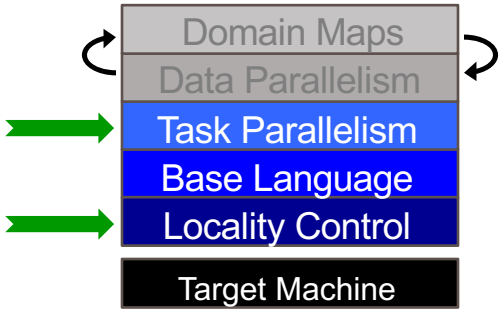
```
config const n = 10;

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

- **Object-oriented features**
- **Generic programming / polymorphism**
- **Procedure overloading / filtering**
- **Arguments:** default values, intents, name-based matching, type queries
- **Compile-time meta-programming**
- **Modules** (namespaces)
- **Managed objects and lifetime checking**
- **Error-handling**
- and more…

Task Parallelism and Locality Control



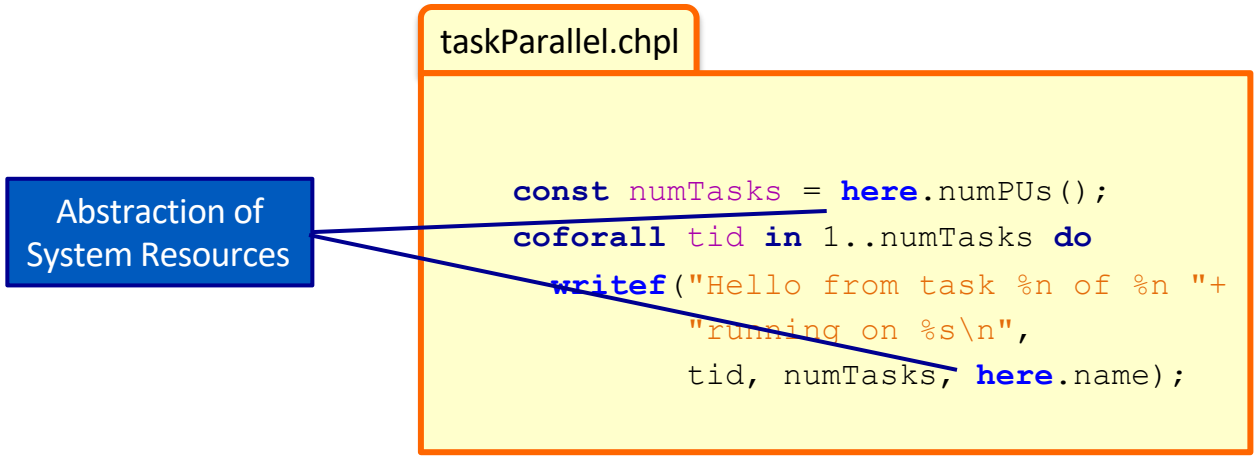
Task Parallelism and Locality, by example

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);
```

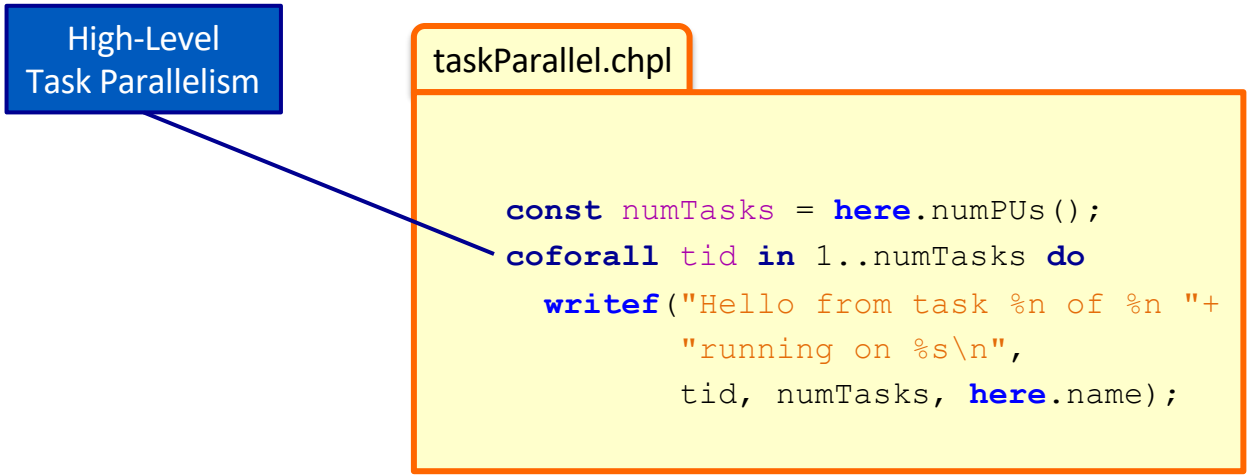
```
prompt> chpl taskParallel.chpl
prompt> ./taskParallel
Hello from task 2 of 2 running on n1032
Hello from task 1 of 2 running on n1032
```

Task Parallelism and Locality, by example




```
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




```
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

This is a shared memory program

Nothing has referred to remote locales, explicitly or implicitly

taskParallel.chpl

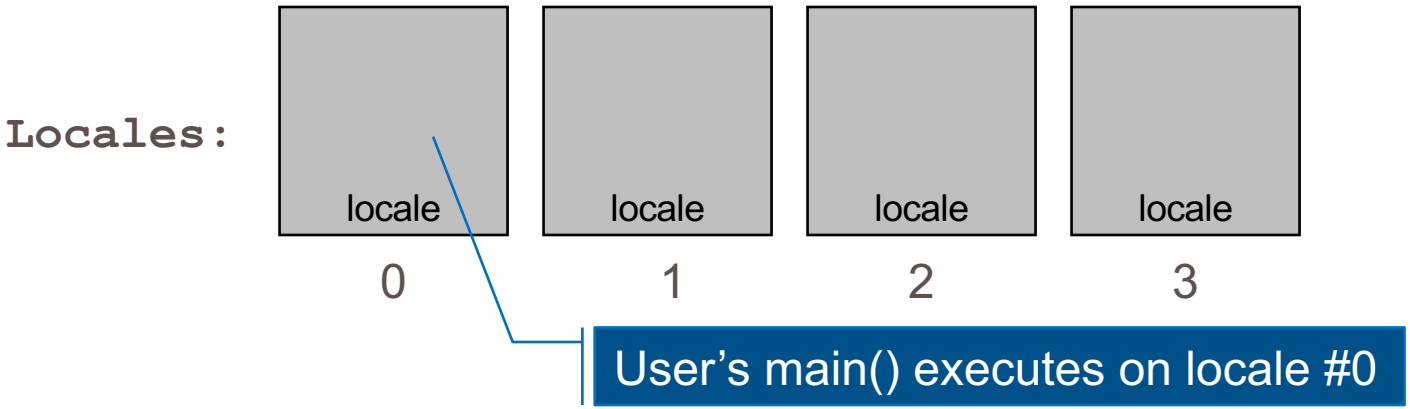
```
const numTasks = here.numPUs();
coforall tid in 1..numTasks do
  writef("Hello from task %n of %n "+
         "running on %s\n",
         tid, numTasks, here.name);
```

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

Locales, briefly

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

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



Task Parallelism and Locality, by example

This is a shared memory program

Nothing has referred 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

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

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

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

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);
  }
```

Control of Locality/Affinity

```
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

taskParallel.chpl

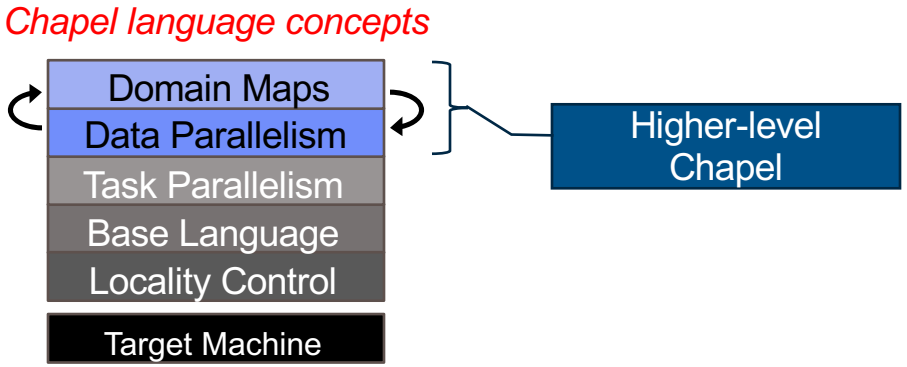
```
coforall loc in Locales do
  on loc {
    const numTasks = here.numPUs();
    coforall tid in 1..numTasks do
      writef("Hello from task %n of %n "+
             "running on %s\n",
             tid, numTasks, here.name);
  }
```

```
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

- **Atomic / Synchronized variables:** for sharing data & coordination
- **begin / cobegin statements:** other ways of creating tasks

Data Parallelism in Chapel



Data Parallelism, by example

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

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

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

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

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
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

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
```

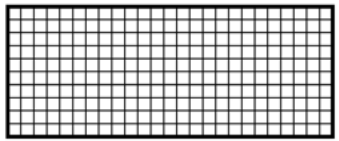

Distributed Data Parallelism, by example

dataParallel.chpl

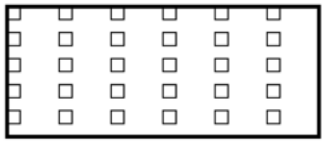
```
use CyclicDist;
config const n = 1000;
var D = {1..n, 1..n}
        dmapped Cyclic(startIdx = (1,1));
var A: [D] real;
forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

```
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
```

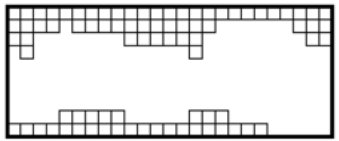

Chapel Has Several Domain / Array Types



dense

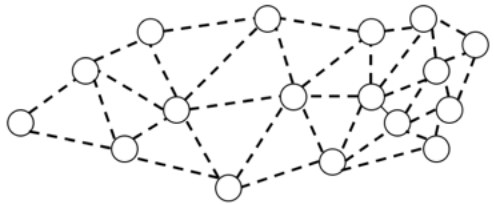


strided



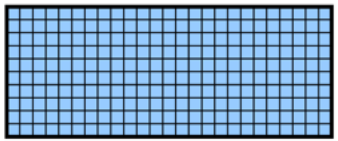
sparse

associative

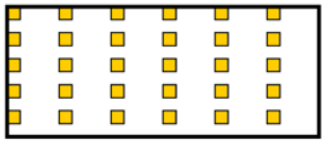


unstructured

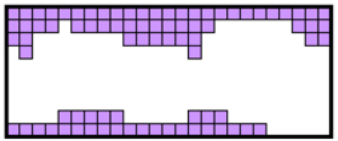
Chapel Has Several Domain / Array Types



dense

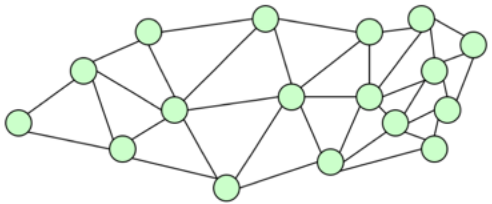


strided



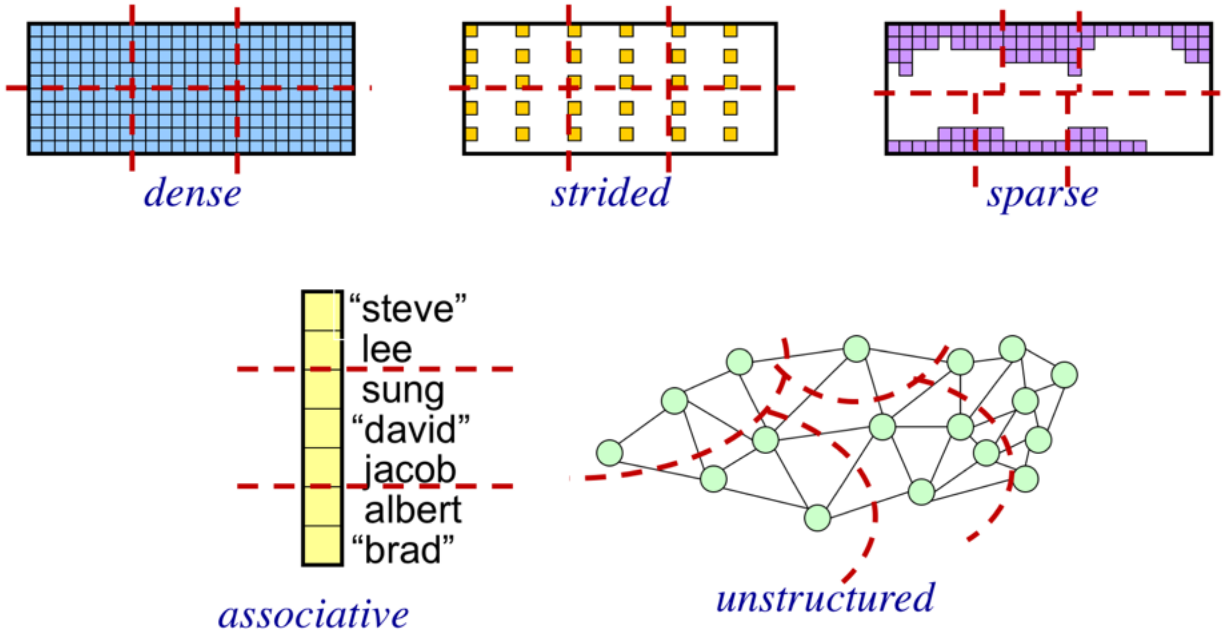
sparse

associative



unstructured

Chapel Has Several Domain / Array Types

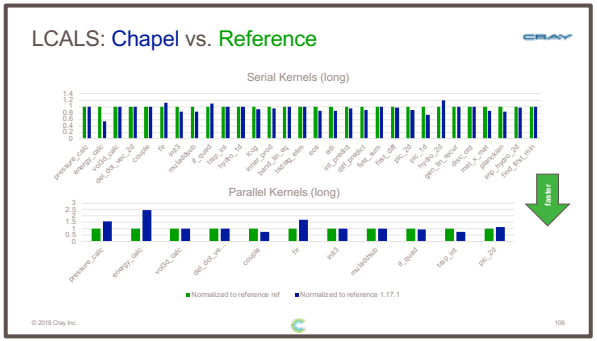


Other Data Parallel Features

- **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:** compute parallel prefix operations
- **Several Flavors of Domains and Arrays**

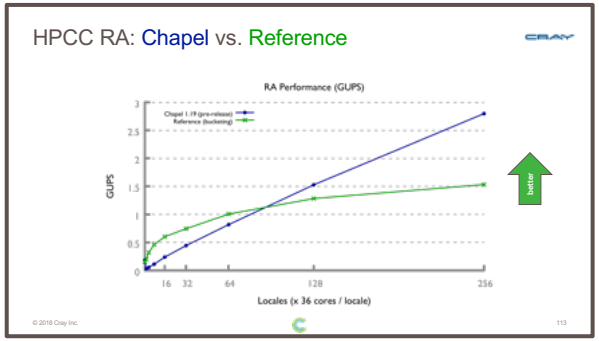
Chapel Results

HPC Patterns: Chapel vs. Reference



LCALS

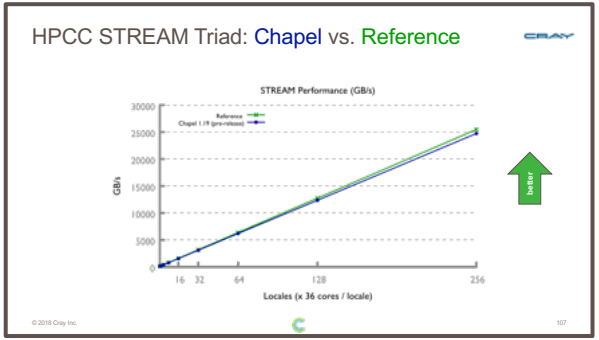
HPCC RA

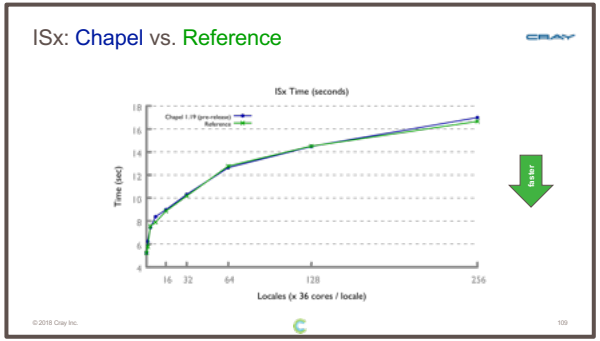


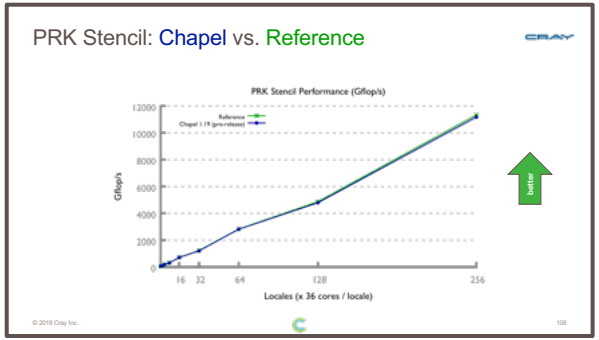
STREAM Triad

ISx

PRK Stencil

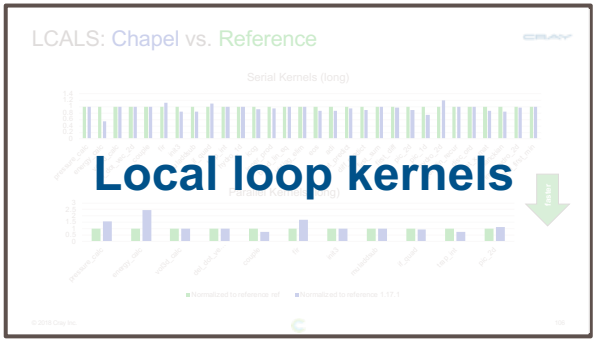






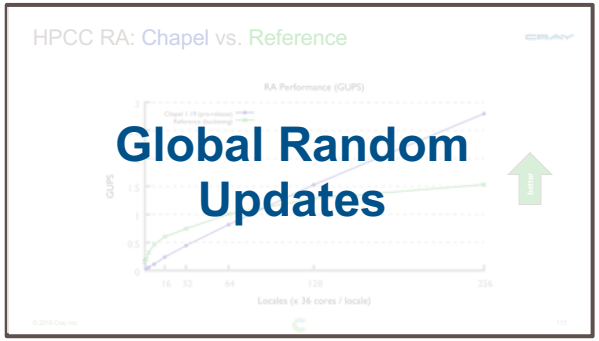
Nightly performance tickers online at:
https://chapel-lang.org/perf-nightly.html

HPC Patterns: Chapel vs. Reference



LCALS

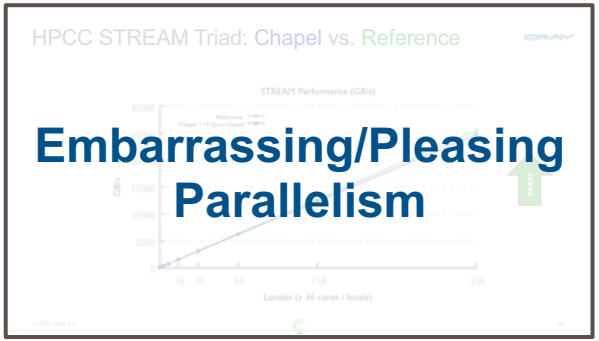
HPCC RA

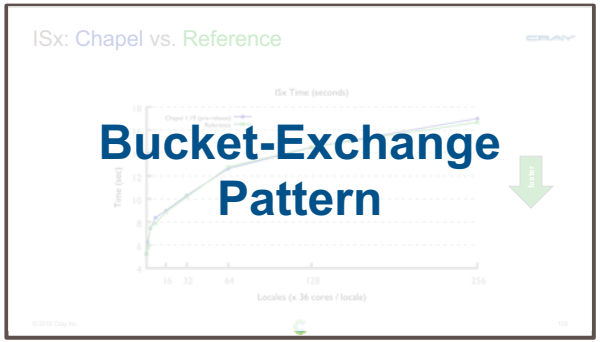


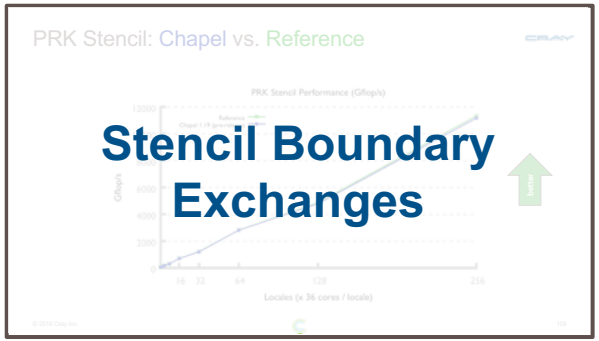
STREAM Triad

ISx

PRK Stencil

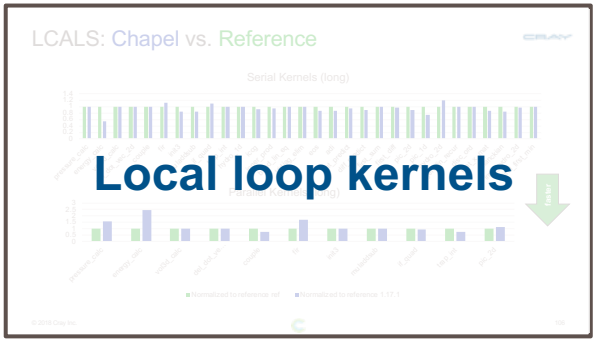






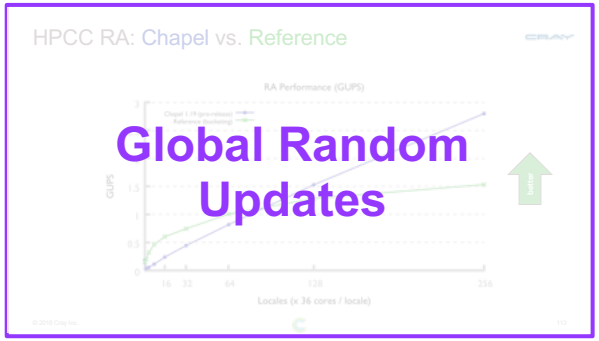
Nightly performance tickers online at:
https://chapel-lang.org/perf-nightly.html

HPC Patterns: Chapel vs. Reference



LCALS

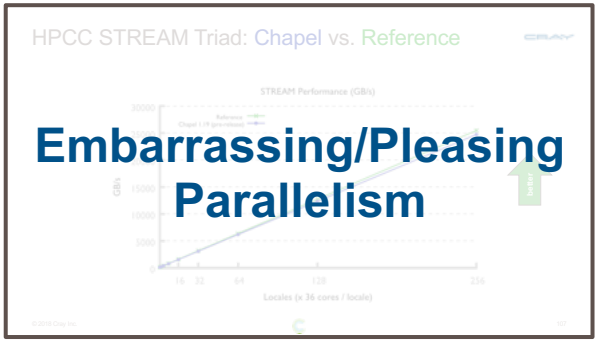
HPCC RA

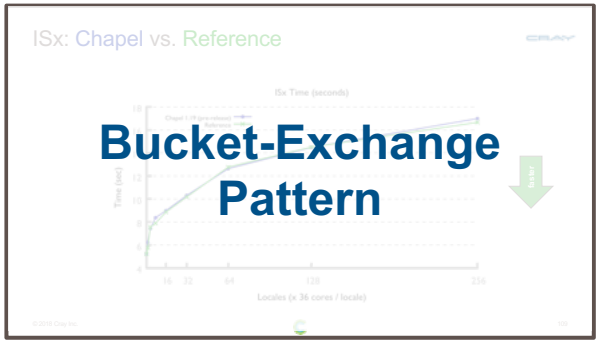


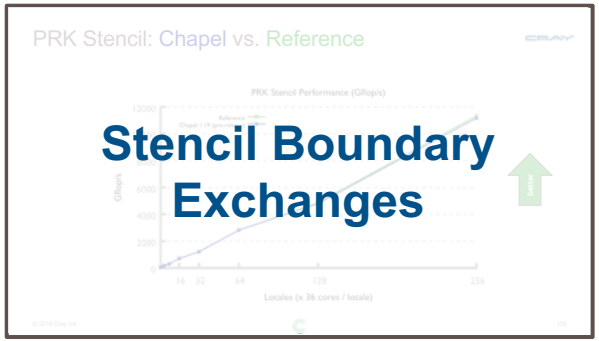
STREAM Triad

ISx

PRK Stencil



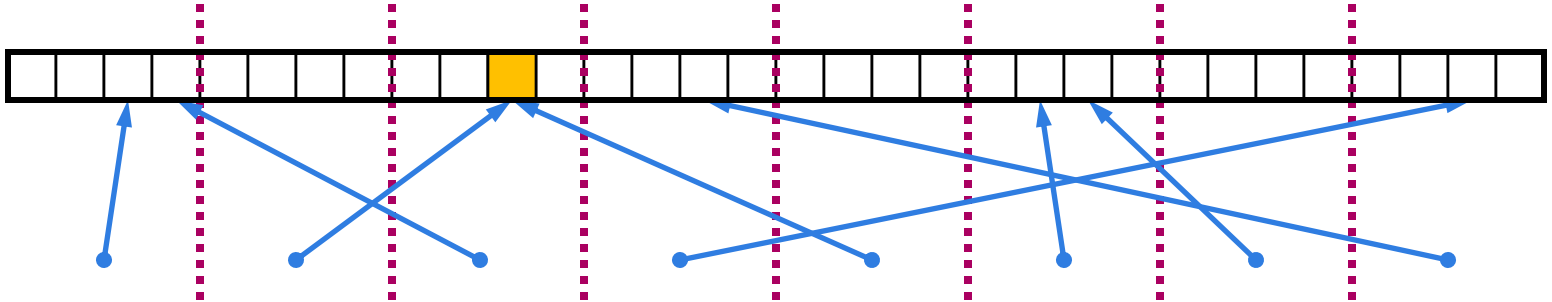




Nightly performance tickers online at:
https://chapel-lang.org/perf-nightly.html

Case Study: HPCC Random Access (RA)

Data Structure: distributed table



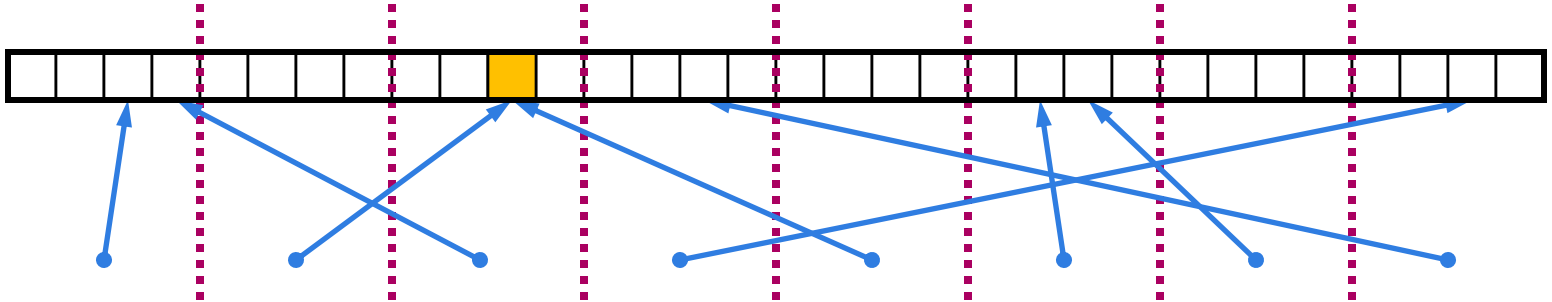
Computation: update random table locations in parallel

Two variations:

- **lossless:** don't allow any updates to be lost
- **lossy:** permit some fraction of updates to be lost

Case Study: HPCC Random Access (RA)

Data Structure: distributed table



Computation: update random table locations in parallel

Two variations:

- **lossless:** don't allow any updates to be lost
- **lossy:** permit some fraction of updates to be lost

HPCC RA: MPI kernel

```
/* Perform updates to main table.  The scalar equivalent is:
 *
 *   for (i=0; i<NUPDATE; i++) {
 *     Ran = (Ran << 1) ^ (((s64Int) Ran < 0) ? POLY : 0);
 *     Table[Ran & (TABSIZE-1)] ^= Ran;
 *   }
 */

MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64,
          MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &inreq);
 while (i < SendCnt) {
   /* receive messages */
   do {
     MPI_Test(&inreq, &have_done, &status);
     if (have_done) {
       if (status.MPI_TAG == UPDATE_TAG) {
         MPI_Get_count(&status, tparams.dtype64, &recvUpdates);
         bufferBase = 0;
         for (j=0; j < recvUpdates; j ++) {
           inmsg = LocalRecvBuffer[bufferBase+j];
           LocalOffset = (inmsg & (tparams.TableSize - 1)) -
                         tparams.GlobalStartMyProc;
           HPCC_Table[LocalOffset] ^= inmsg;
         }
       } else if (status.MPI_TAG == FINISHED_TAG) {
         NumberReceiving--;
       } else
         MPI_Abort( MPI_COMM_WORLD, -1 );
       MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64,
                 MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &inreq);
     }
   } while (have_done && NumberReceiving > 0);
   if (pendingUpdates < maxPendingUpdates) {
     Ran = (Ran << 1) ^ ((s64Int) Ran < ZERO64B ? POLY : ZERO64B);
     GlobalOffset = Ran & (tparams.TableSize-1);
     if ( GlobalOffset < tparams.Top)
       WhichPe = ( GlobalOffset / (tparams.MinLocalTableSize + 1) );
     else
       WhichPe = ( (GlobalOffset - tparams.Remainder) /
                 tparams.MinLocalTableSize );
     if (WhichPe == tparams.MyProc) {
       LocalOffset = (Ran & (tparams.TableSize - 1)) -
                     tparams.GlobalStartMyProc;
       HPCC_Table[LocalOffset] ^= Ran;
     } else {
       HPCC_InsertUpdate(Ran, WhichPe, Buckets);
       pendingUpdates++;
     }
     i++;
   }
   else {
     MPI_Test(&outreq, &have_done, MPI_STATUS_IGNORE);
     if (have_done) {
       outreq = MPI_REQUEST_NULL;
       pe = HPCC_GetUpdates(Buckets, LocalSendBuffer, localBufferSize,
                            &peUpdates);
       MPI_Isend(&LocalSendBuffer, peUpdates, tparams.dtype64, (int)pe,
                 UPDATE_TAG, MPI_COMM_WORLD, &outreq);
       pendingUpdates -= peUpdates;
     }
   }
}
/* send remaining updates in buckets */
while (pendingUpdates > 0) {
  /* receive messages */
  do {
    MPI_Test(&inreq, &have_done, &status);
    if (have_done) {
      if (status.MPI_TAG == UPDATE_TAG) {
        MPI_Get_count(&status, tparams.dtype64, &recvUpdates);
        bufferBase = 0;
        for (j=0; j < recvUpdates; j ++) {
          inmsg = LocalRecvBuffer[bufferBase+j];
          LocalOffset = (inmsg & (tparams.TableSize - 1)) -
                        tparams.GlobalStartMyProc;
          HPCC_Table[LocalOffset] ^= inmsg;
        }
      } else if (status.MPI_TAG == FINISHED_TAG) {
        /* we got a done message.  Thanks for playing... */
        NumberReceiving--;
      } else {
        MPI_Abort( MPI_COMM_WORLD, -1 );
      }
      MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64,
                MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &inreq);
    }
  } while (have_done && NumberReceiving > 0);
  MPI_Test(&outreq, &have_done, MPI_STATUS_IGNORE);
  if (have_done) {
    outreq = MPI_REQUEST_NULL;
    pe = HPCC_GetUpdates(Buckets, LocalSendBuffer, localBufferSize,
                         &peUpdates);
    MPI_Isend(&LocalSendBuffer, peUpdates, tparams.dtype64, (int)pe,
              UPDATE_TAG, MPI_COMM_WORLD, &outreq);
    pendingUpdates -= peUpdates;
  }
}
/* send our done messages */
for (proc_count = 0 ; proc_count < tparams.NumProcs ; ++proc_count) {
  if (proc_count == tparams.MyProc) { tparams.finish_req[tparams.MyProc] =
                                      MPI_REQUEST_NULL; continue; }
  /* send garbage - who cares, no one will look at it */
  MPI_Isend(&Ran, 0, tparams.dtype64, proc_count, FINISHED_TAG,
            MPI_COMM_WORLD, tparams.finish_req + proc_count);
}
/* Finish everyone else up... */
while (NumberReceiving > 0) {
  MPI_Wait(&inreq, &status);
  if (status.MPI_TAG == UPDATE_TAG) {
    MPI_Get_count(&status, tparams.dtype64, &recvUpdates);
    bufferBase = 0;
    for (j=0; j < recvUpdates; j ++) {
      inmsg = LocalRecvBuffer[bufferBase+j];
      LocalOffset = (inmsg & (tparams.TableSize - 1)) -
                    tparams.GlobalStartMyProc;
      HPCC_Table[LocalOffset] ^= inmsg;
    }
  } else if (status.MPI_TAG == FINISHED_TAG) {
    /* we got a done message.  Thanks for playing... */
    NumberReceiving--;
  } else {
    MPI_Abort( MPI_COMM_WORLD, -1 );
  }
  MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64,
            MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &inreq);
}

MPI_Waitall( tparams.NumProcs, tparams.finish_req, tparams.finish_statuses);
```

HPCC RA: MPI kernel comment vs. Chapel

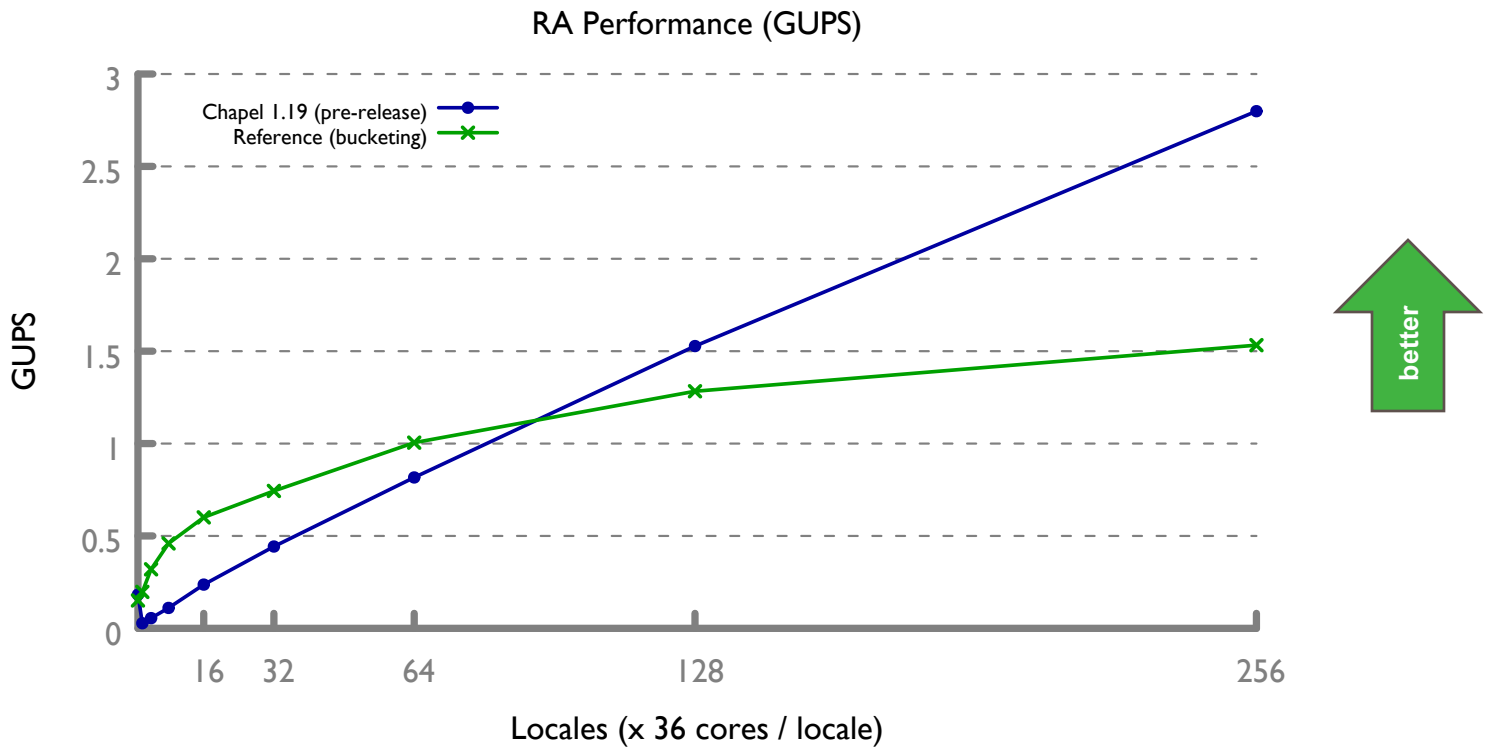
Chapel Kernel

```
forall (_, r) in zip(Updates, RAStream()) do
  T[r & indexMask].xor(r);
```

MPI Comment

```
/* Perform updates to main table. The scalar equivalent is:
 *
 *      for (i=0; i<NUPDATE; i++) {
 *        Ran = (Ran << 1) ^ (((s64Int) Ran < 0) ? POLY : 0);
 *        Table[Ran & (TABSIZE-1)] ^= Ran;
 *      }
 */
```

HPCC RA: Chapel vs. Reference



HPCC RA: Chapel translation

- Given the Chapel code:

```
forall (_, r) in zip(Updates, RAStream()) do
  T[r & indexMask].xor(r);
```

- An *approximate* translation of this code is:

```
coforall tid in 0..#nTasks do on … do          // create a number of distributed tasks
  for r in chunk(RAStream(), tid, nTasks) do   // loop over each task's iterations…
    T[r & indexMask].xor(r);                   // …computing each atomic op serially
```

HPCC RA: Chapel translation

- Given the Chapel code:

```
forall (_, r) in zip(Updates, RAStream()) do
  T[r & indexMask].xor(r);
```

- An *approximate* translation of this code is:

```
coforall tid in 0..#nTasks do on … do          // create a number of distributed tasks
  for r in chunk(RAStream(), tid, nTasks) do   // loop over each task's iterations…
    T[r & indexMask].xor(r);                   // …computing each atomic op serially
```

Note an opportunity for optimization:

- forall-loops imply iterations can execute simultaneously / in any order
- T[] is obviously not read again within this loop's body
- therefore, there's no need to serially execute each atomic op

HPCC RA: Chapel translation, optimized

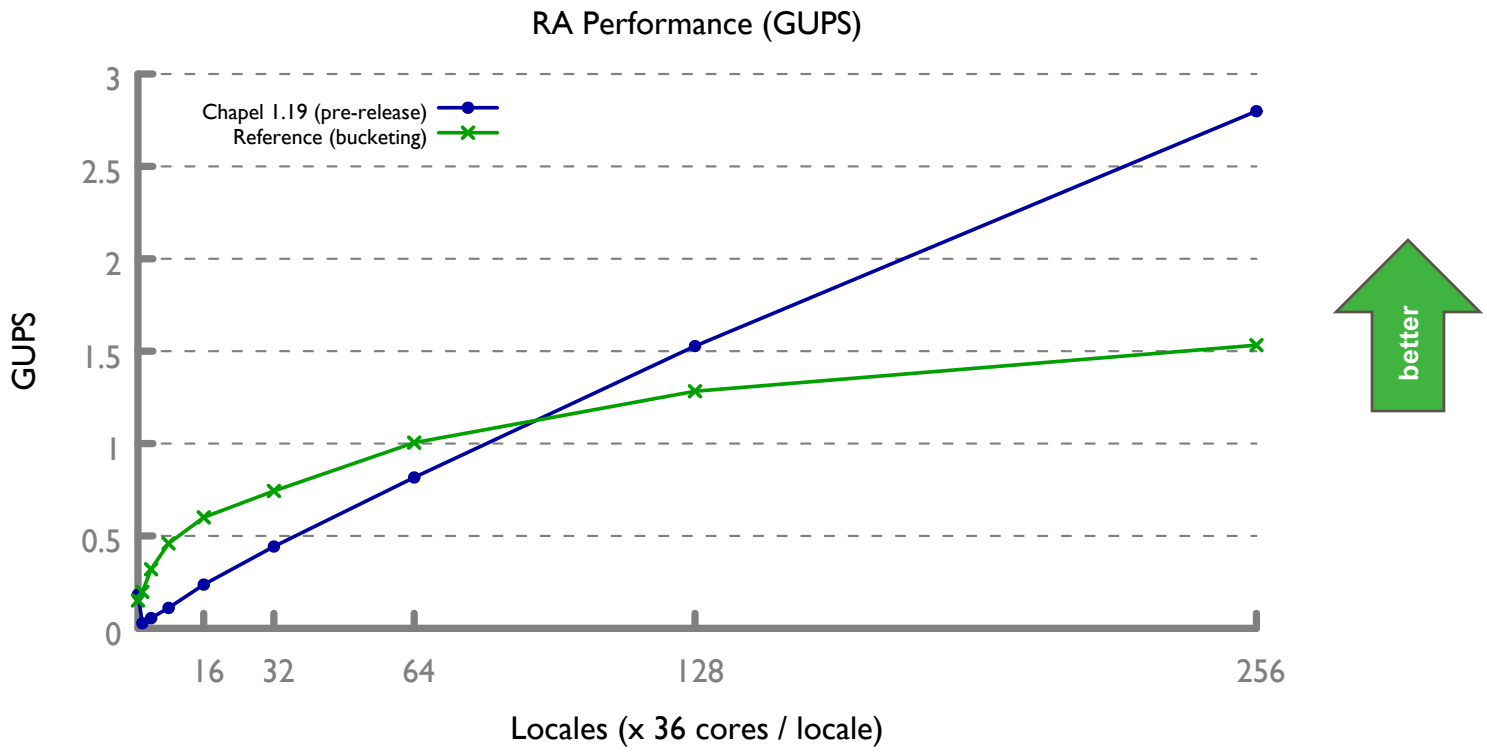
- Given the Chapel code:

```
forall (_, r) in zip(Updates, RAStream()) do
  T[r & indexMask].xor(r);
```

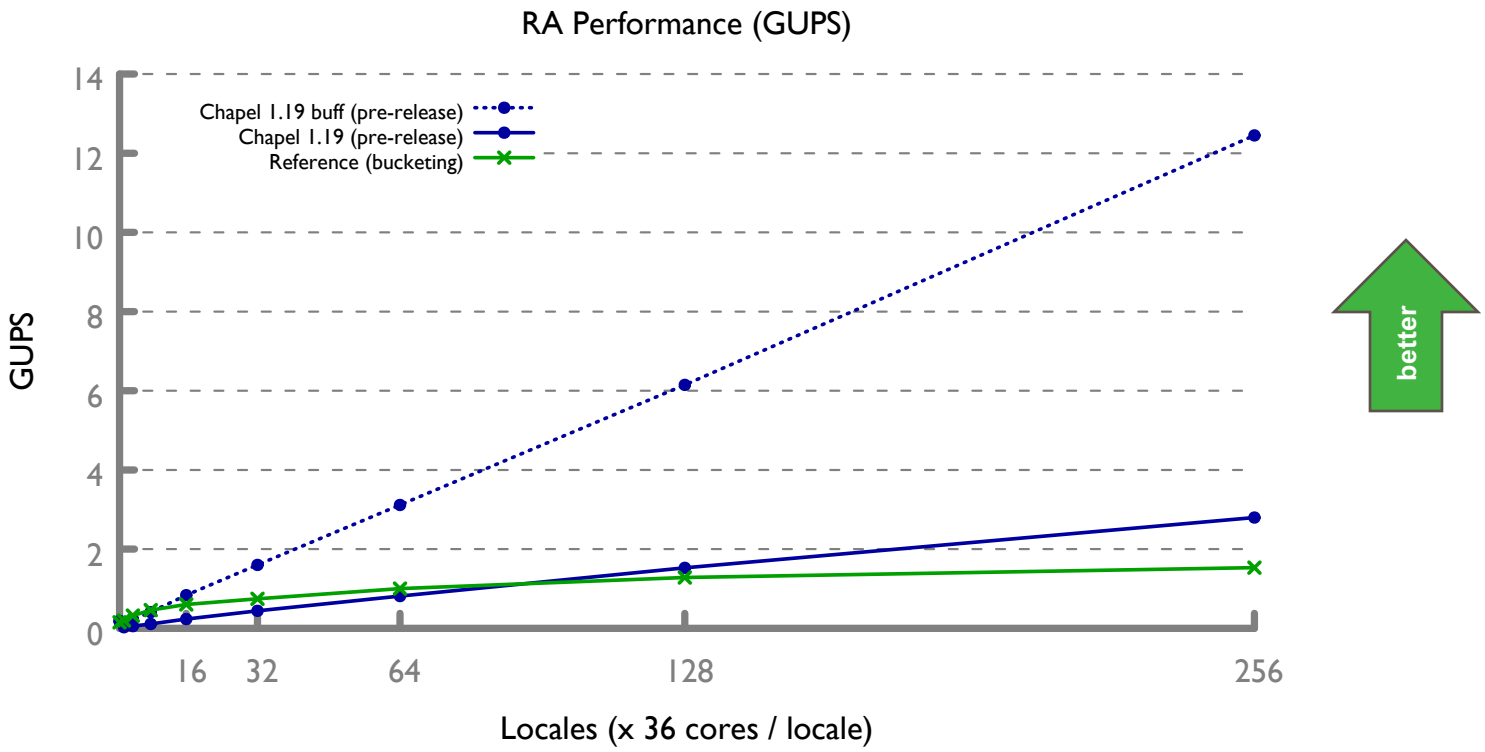
- An *approximate* translation of this code, when optimized, is:

```
coforall tid in 0..#nTasks do on … do          // create a number of distributed tasks
  for r in chunk(RAStream(), tid, nTasks) do   // loop over each task's iterations…
    T[r & indexMask].xor_async(r);             // …computing each atomic op asynchronously
  // tasks wait for asynchronous atomics to complete before terminating
```

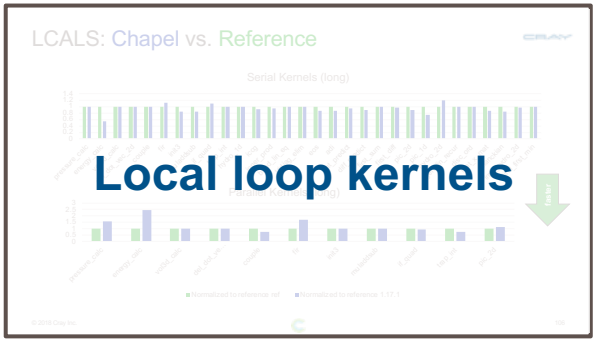
HPCC RA: Chapel vs. Reference



HPCC RA: Chapel (w/ async. atomics) vs. Reference

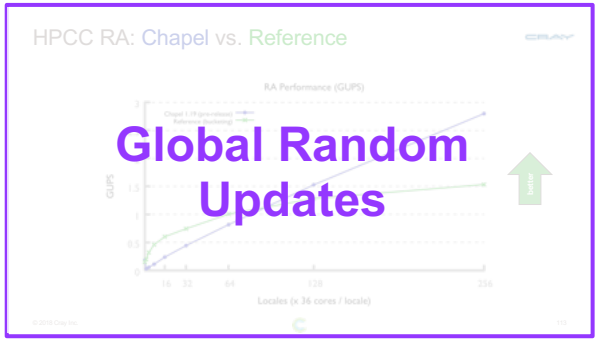


HPC Patterns: Chapel vs. Reference



LCALS

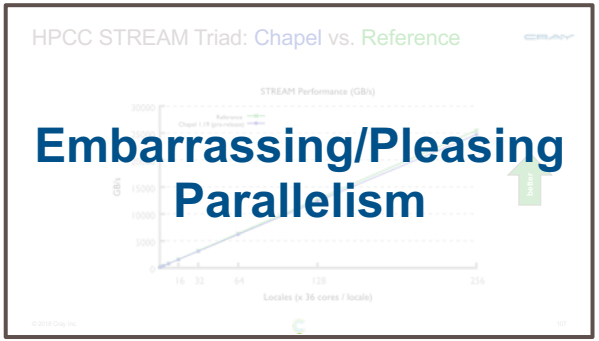
HPCC RA

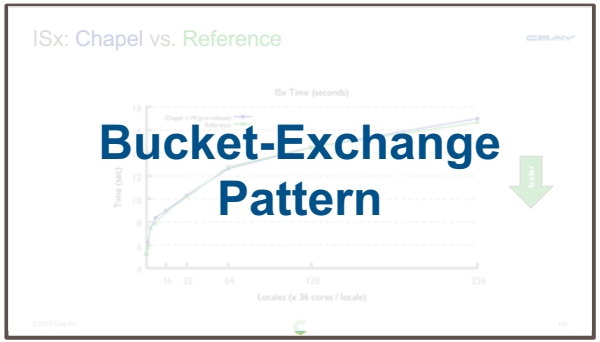


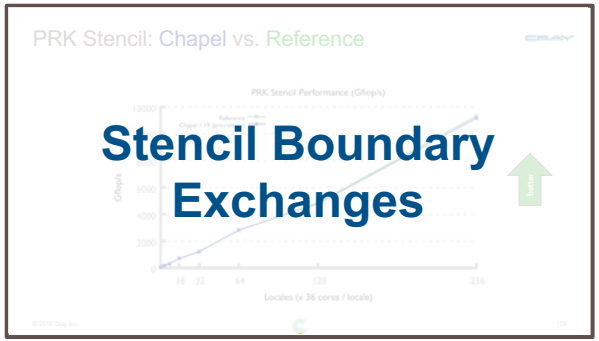
STREAM Triad

ISx

PRK Stencil

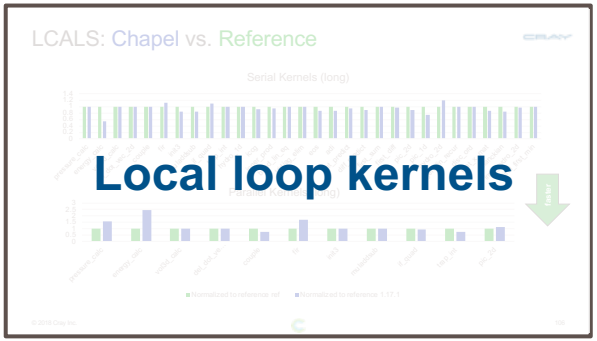






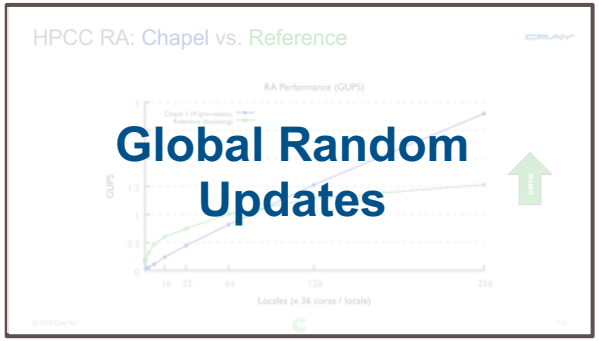
Nightly performance tickers online at:
https://chapel-lang.org/perf-nightly.html

HPC Patterns: Chapel vs. Reference



LCALS

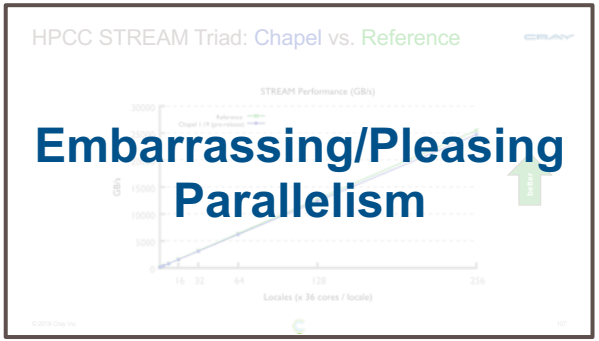
HPCC RA

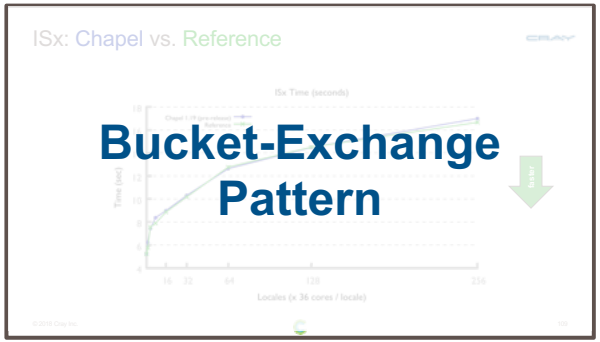


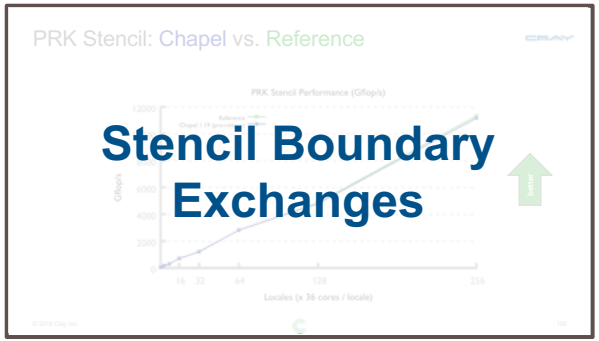
STREAM Triad

ISx

PRK Stencil

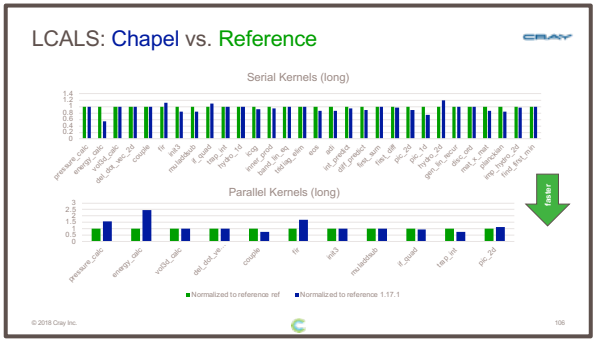




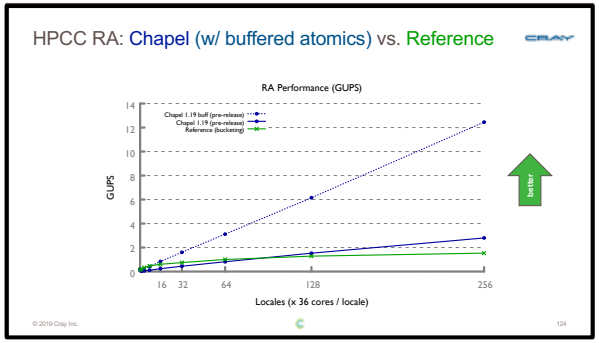


Nightly performance tickers online at:
https://chapel-lang.org/perf-nightly.html

HPC Patterns: Chapel vs. Reference

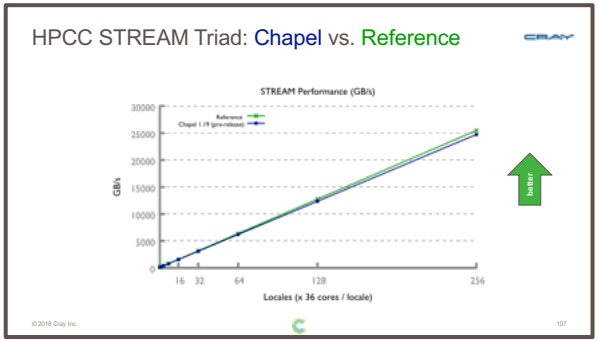


LCALS

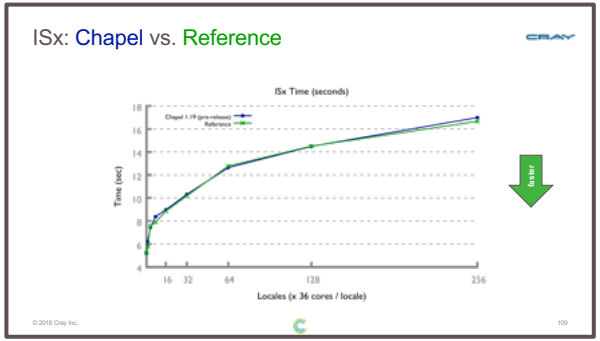


HPCC RA

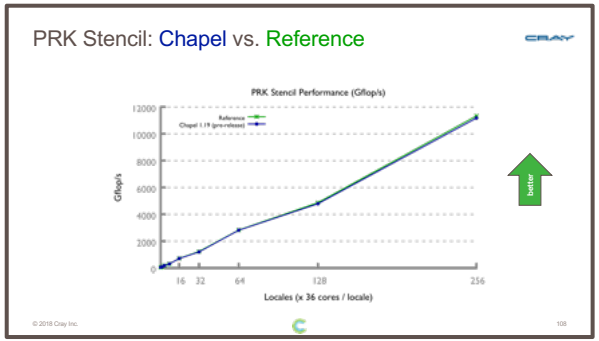
STREAM Triad



ISx



PRK Stencil



Nightly performance tickers online at:
https://chapel-lang.org/perf-nightly.html

The Chapel Team at Cray (May 2018)

Chapel Community Partners

(and several others…)

https://chapel-lang.org/collaborations.html

Typical arguments against languages for HPC

- “It’s too difficult for new languages to get adopted”
- “We’re too small of a community to be able to support a language”
- “HPC programmers are happy with current programming methods”
- “HPC is so performance-oriented that productivity doesn’t matter”
- “It’s challenging to get performance from parallel languages”

I think there are counterarguments to each of these, the overarching one being:
“Scalable parallel programming is deserving of first-class language support”

Why Consider New Languages at all?

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, CHIUW 2018 keynote: *Why Languages Matter More Than Ever*]

A Brief History of Chapel: Infancy

Chapel's infancy: DARPA HPCS (2003–2012)

- **Goal:** Improve HPC productivity by 10x
- ~6 FTEs
- Research focus:
  - distinguish locality from parallelism
  - seamlessly mix data- and task-parallelism
  - support user-defined distributed arrays, parallel iterators
- CUG 2013 paper captured post-HPCS project status:

  The State of the Chapel Union
  Chamberlain, Choi, Dumler, Hildebrandt, Iten, Litvinov, Titus

A Brief History of Chapel: Adolescence

Chapel's adolescence: "the five-year push" (2013–2018)

- **Goal:** evolve the HPCS research prototype towards production
- ~13–14 FTEs
- Development focus:
  - address weak points in HPCS prototype
  - support and nurture the Chapel community
- CUG 2018 talk & paper reported on progress during this time

  Chapel Comes of Age: Making Scalable Programming Productive,
  Chamberlain, Ronaghan, Albrecht, Duncan, Ferguson, Harshbarger, Iten, Keaton, Litvinov, Sahabu, and Titus

A Brief History of Chapel: What's Next?

Chapel's college years: plans for 2019-2021

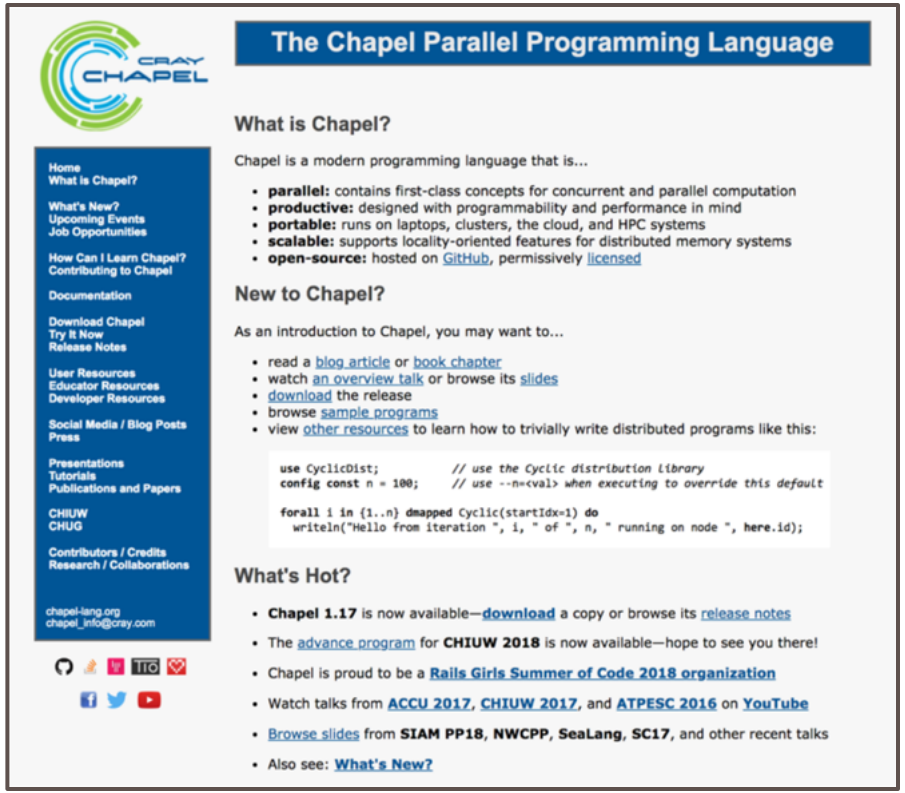
- **Goal:** evolve the HPCS research prototype towards production
- ~13–16 FTEs
- Development focus:
  - Stabilize Core Language
  - Interoperability / Usability Improvements
  - Portability Improvements (Cray Shasta, GPUs)
  - Data Ingestion
  - Chapel AI

Chapel Resources

Chapel Central

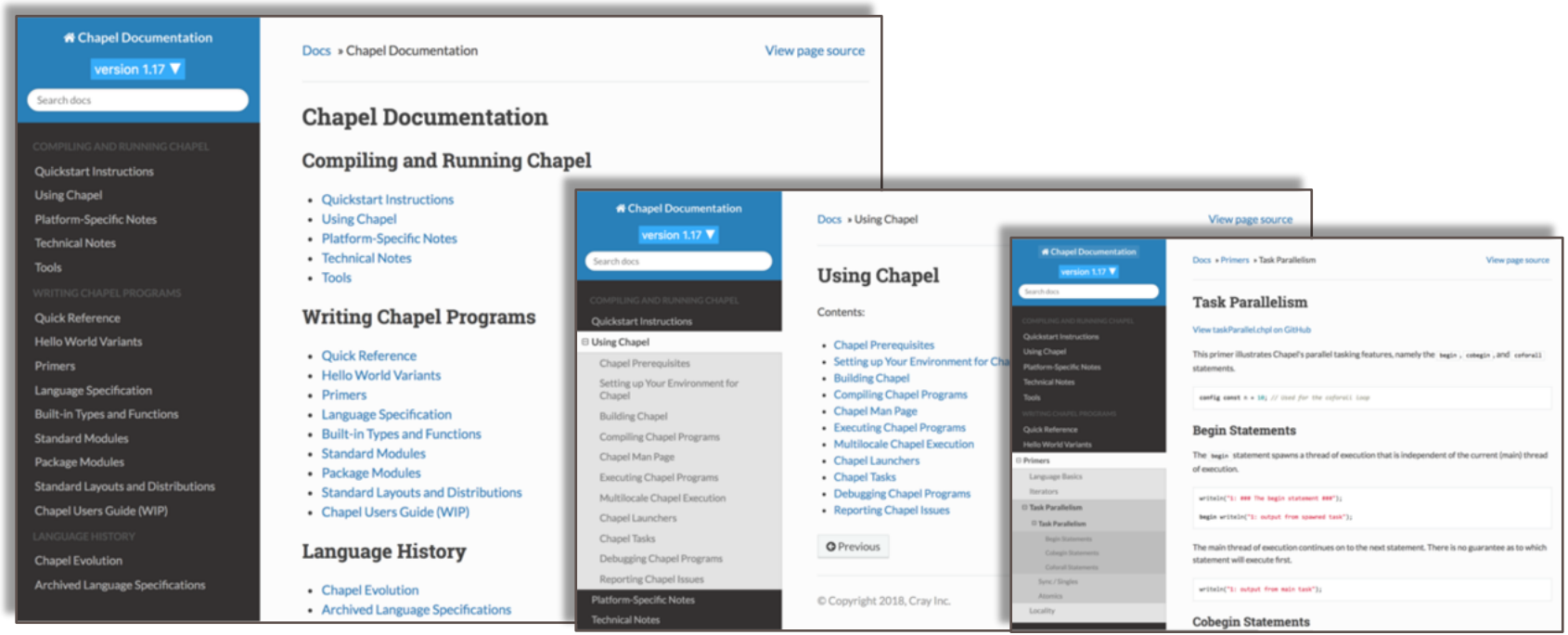
https://chapel-lang.org

- downloads
- presentations
- papers
- resources
- documentation

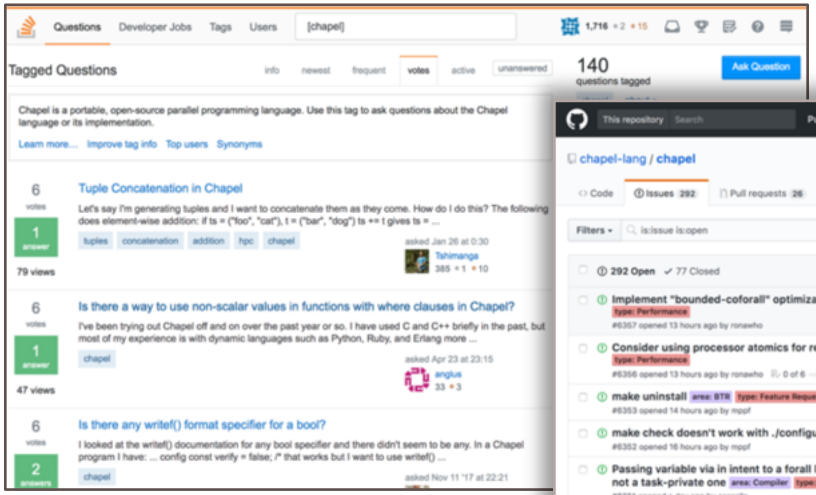


Chapel Online Documentation

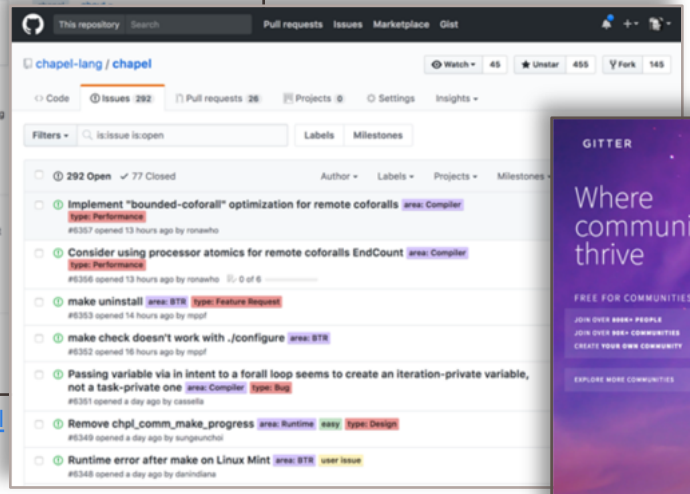
https://chapel-lang.org/docs: ~200 pages, including primer examples



Chapel Community



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



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

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

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

Chapel Social Media (no account required)

http://twitter.com/ChapelLanguage

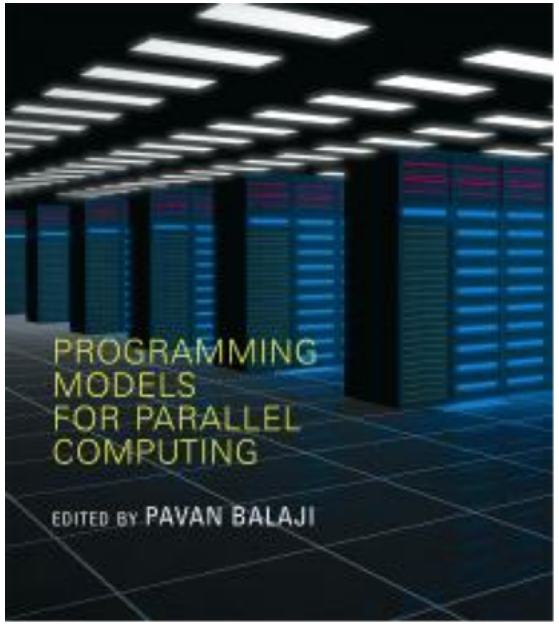
http://facebook.com/ChapelLanguage

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

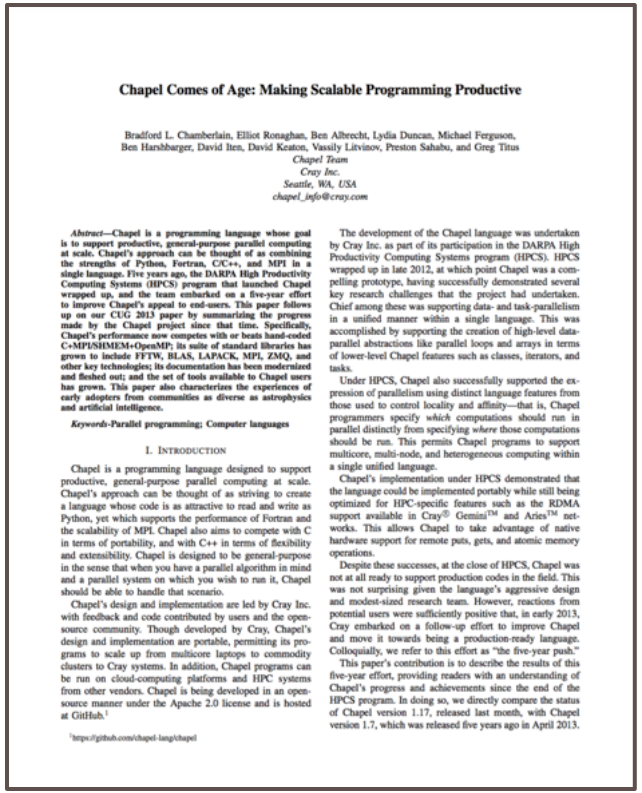
Suggested Reading: Chapel history and overview

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 Harshbarger, 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. Five years ago, the DARPA High Productivity Computing Systems (HPCS) program that launched Chapel wrapped up, and the team embarked on a five-year effort to improve Chapel's appeal to end-users. This paper follows up on our CUG 2013 paper by summarizing the progress made by the Chapel project since that time. Specifically, Chapel's performance now competes with or beats hand-coded C+MPI/SHMEM+OpenMP; its suite of standard libraries has grown to include FFTW, BLAS, 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 early adopters 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 a parallel system on which you wish 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

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 unified manner within a single language. This was accomplished by supporting the creation of high-level data-parallel abstractions like 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 distinctly from specifying *where* those computations should be run. This permits Chapel programs to support multicore, 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-sized 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 effort as "the 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.

[paper](#) and [slides](#) available at chapel-lang.org

Summary and Wrap-up

Chapel offers a unique combination of productivity, performance, and parallelism

Scalable Parallel Computing is deserving of first-class language features

We're interested in identifying and working with the next generation of Chapel users, and are interested in your thoughts and feedback

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-/

Backup Slides: PGAS & Chapel

Partitioned Global Address Space (PGAS) Languages
(Or more accurately: partitioned global namespace languages)

- **abstract concept:**
  - support a shared namespace on distributed memory
    - permit parallel tasks to access remote variables by naming them
  - establish a strong sense of ownership
    - every variable has a well-defined location
    - local variables are cheaper to access than remote ones
- **traditional PGAS languages have been SPMD in nature**
  - best-known examples: Fortran 2008's co-arrays, Unified Parallel C (UPC)

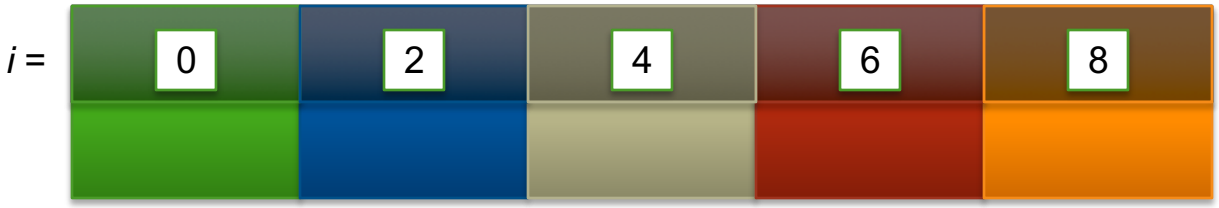
SPMD PGAS Languages (using a pseudo-language, not Chapel)

```
shared int i(*);          // declare a shared variable i
```

i =

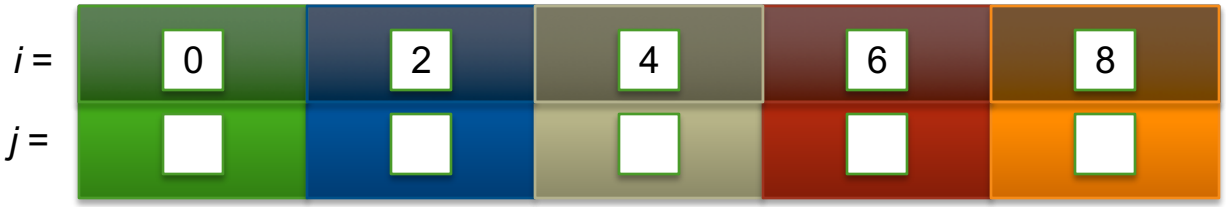
SPMD PGAS Languages (using a pseudo-language, not Chapel)

```
shared int i(*);          // declare a shared variable i
function main() {
  i = 2*this_image();     // each image initializes its copy
```



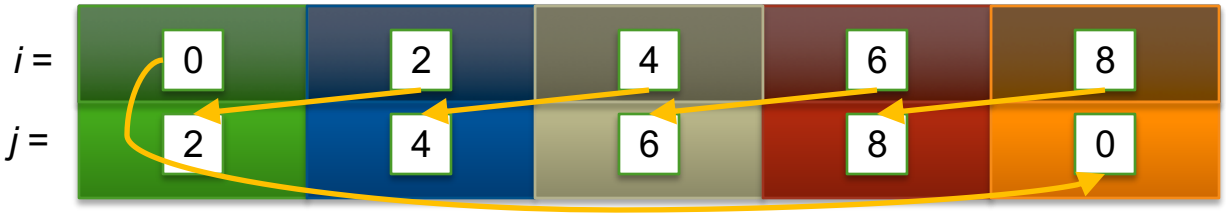
SPMD PGAS Languages (using a pseudo-language, not Chapel)

```
shared int i(*);          // declare a shared variable i
function main() {
  i = 2*this_image();     // each image initializes its copy

  private int j;          // declare a private variable j
```



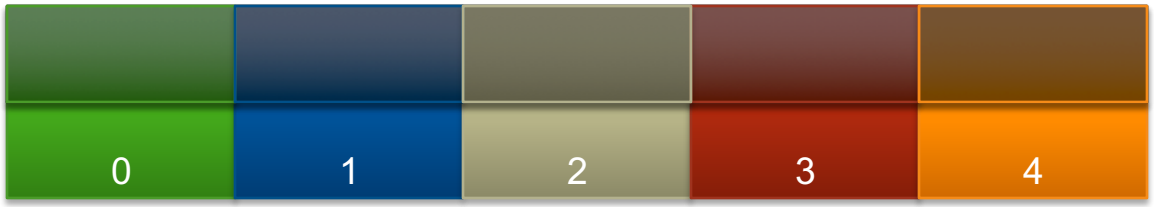
SPMD PGAS Languages (using a pseudo-language, not Chapel)

```
shared int i(*);          // declare a shared variable i
function main() {
  i = 2*this_image();     // each image initializes its copy
  barrier();
  private int j;          // declare a private variable j
  j = i( (this_image()+1) % num_images() );
    // ^^ access our neighbor's copy of i; compiler and runtime implement the communication
    // Q: How did we know our neighbor had an i?
    // A: Because it's SPMD – we're all running the same program so if we have an i, so do they.
```



Chapel and PGAS

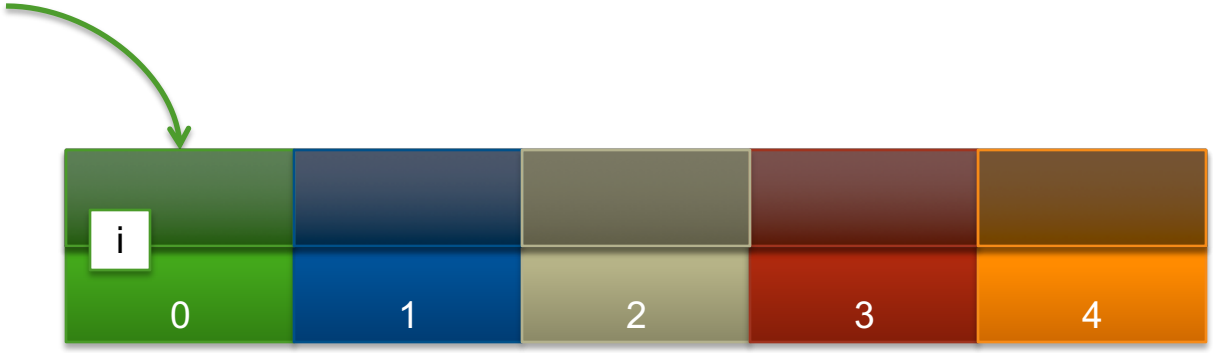
- **Chapel is PGAS, but unlike most, it's not inherently SPMD**
  - never think about "the other copies of the program"
  - "global name/address space" comes from lexical scoping
    - as in traditional languages, each declaration yields one variable
    - variables are stored on the locale where the task declaring it is executing



Locales (think: "compute nodes")

Chapel: Scoping and Locality

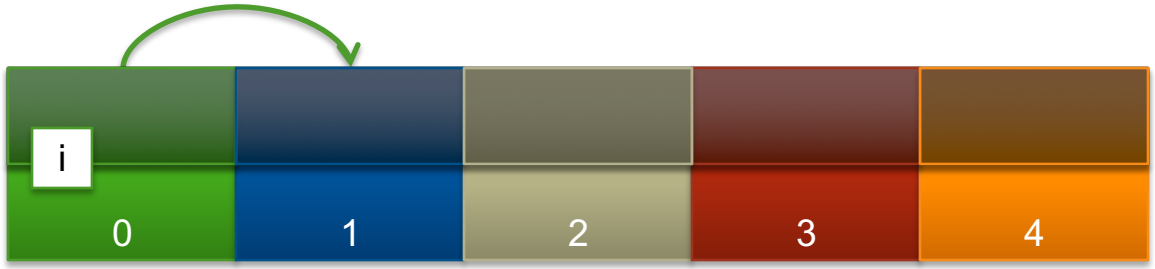
```
var i: int;
```



Locales (think: “compute nodes”)

Chapel: Scoping and Locality

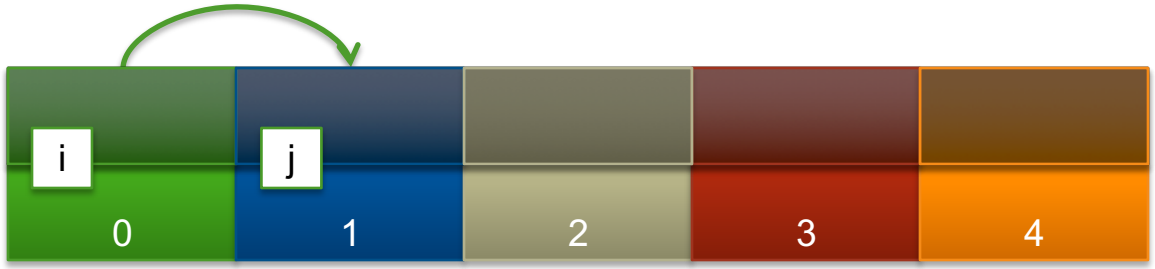
```
var i: int;
on Locales[1] {
```



Locales (think: “compute nodes”)

Chapel: Scoping and Locality

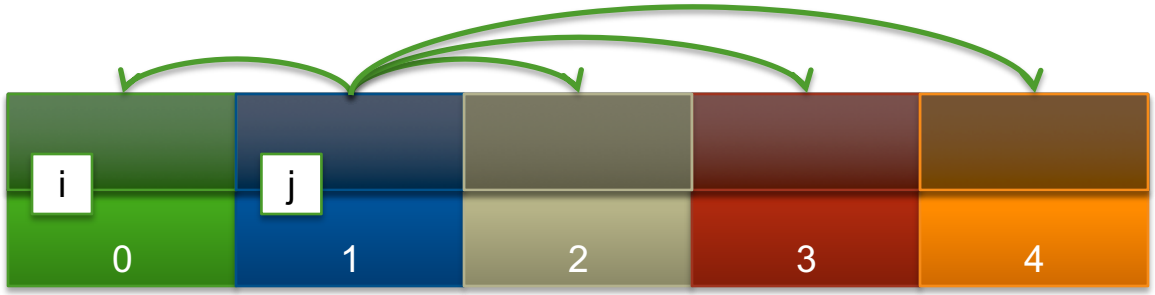
```
var i: int;
on Locales[1] {
  var j: int;
```



Locales (think: “compute nodes”)

Chapel: Scoping and Locality

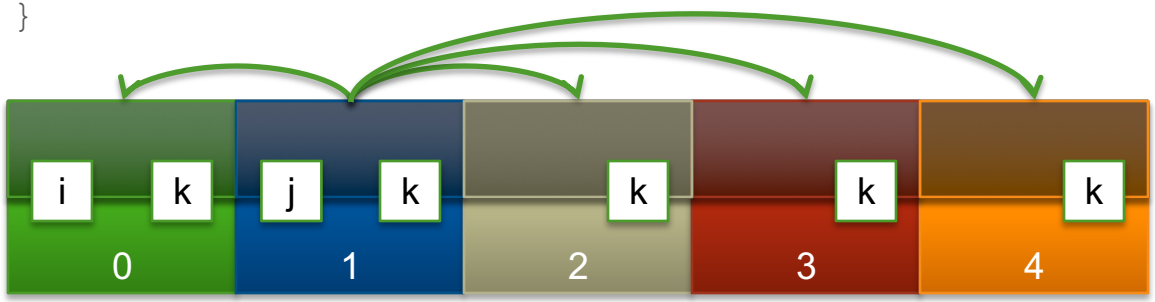
```
var i: int;
on Locales[1] {
  var j: int;
  coforall loc in Locales {
    on loc {
```



Locales (think: “compute nodes”)

Chapel: Scoping and Locality

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



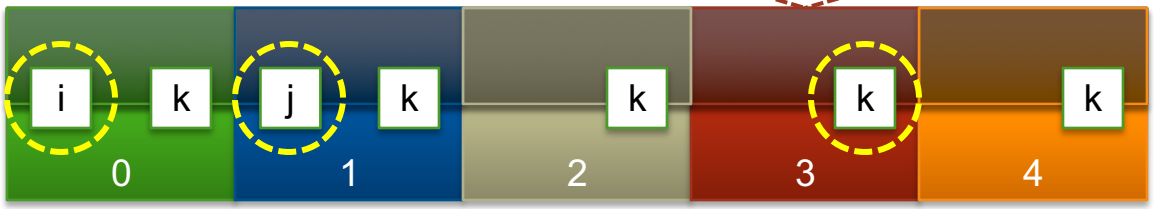
Locales (think: “compute nodes”)

Chapel: Scoping and Locality

```
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

`k = 2*i + j;`

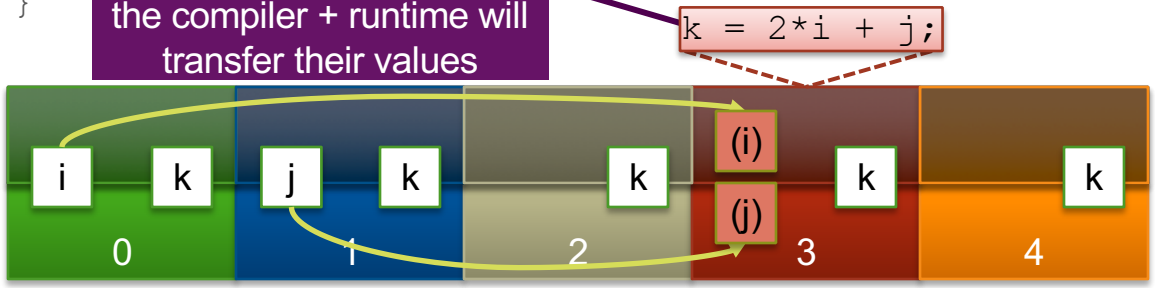


Locales (think: "compute nodes")

Chapel: Scoping and Locality

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

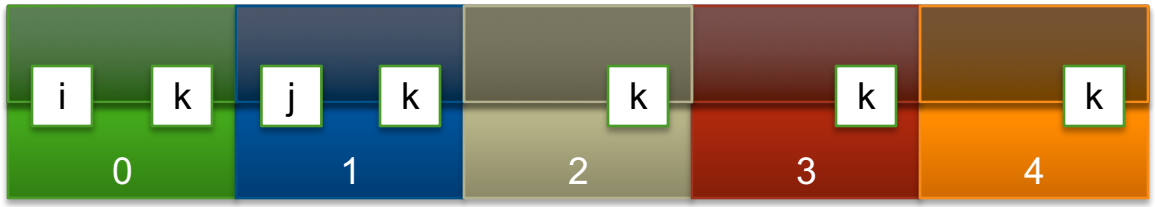
here, *i* and *j* are remote, so the compiler + runtime will transfer their values



Locales (think: "compute nodes")

Chapel: Locality queries

```
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")