Chapel

Programmability, Parallelism, and Performance

Brad Chamberlain

Puget Sound Programming Python Meetup (PuPPy)

December 12, 2018

bradc@cray.com

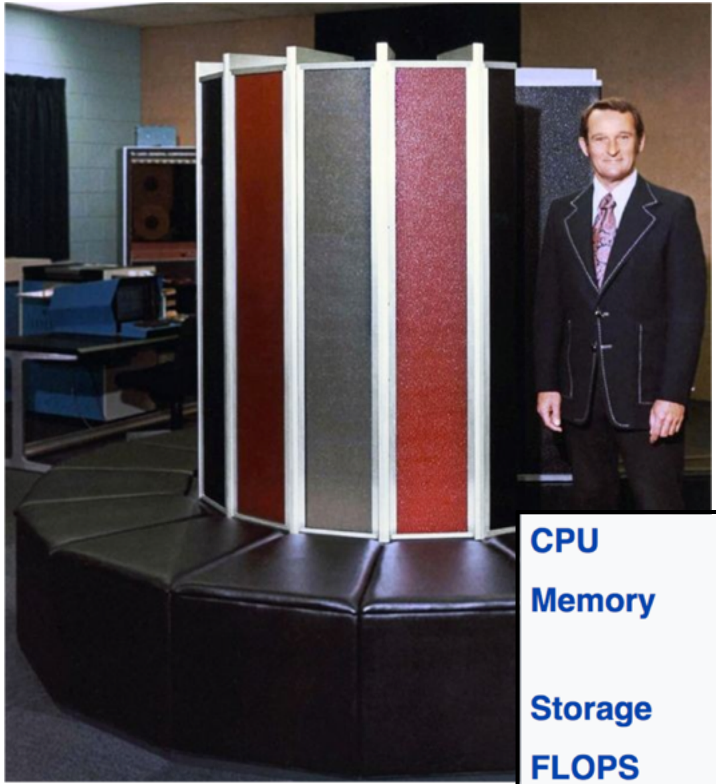
chapel-lang.org

@ChapelLanguage

CRAY

CRAY CHAPEL

Cray-1: A Pioneering Supercomputer (1975)



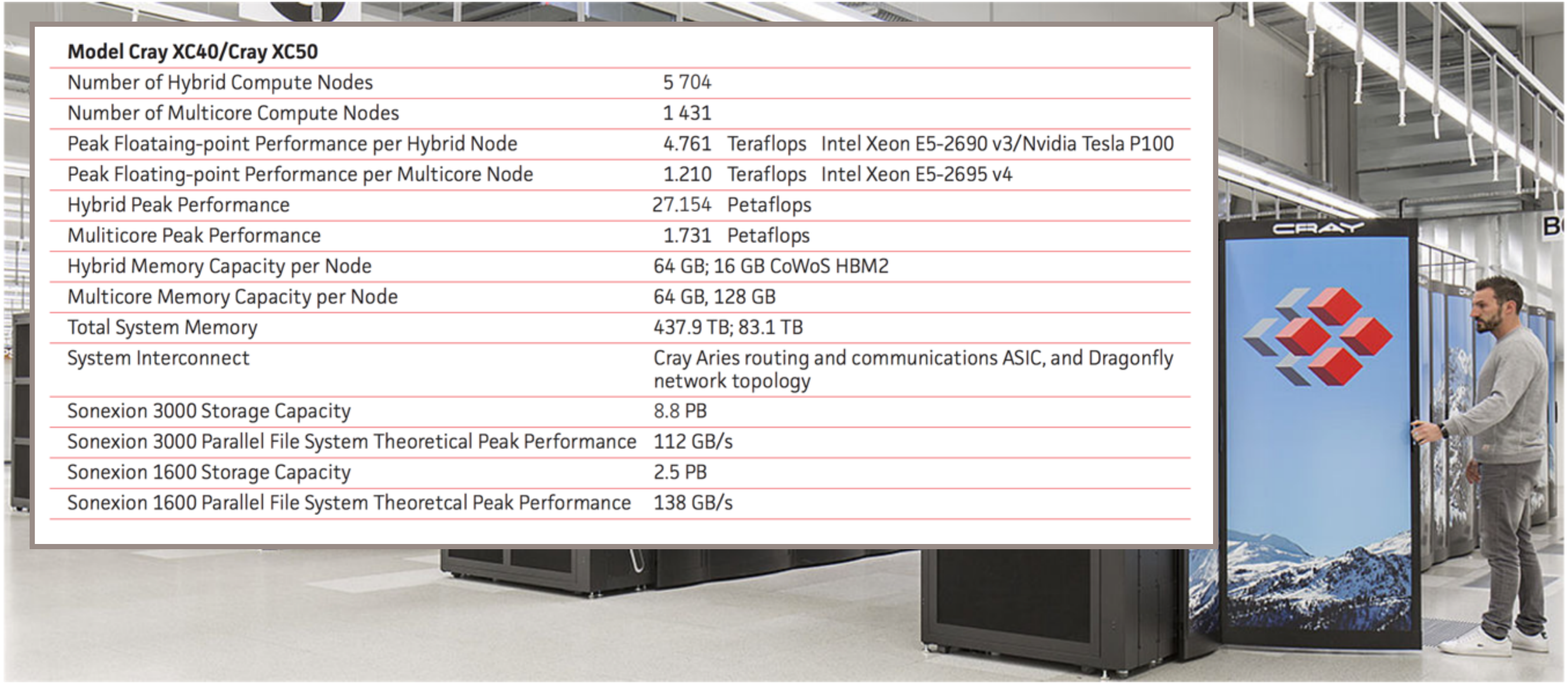
CPU	64-bit processor @ 80 MHz[1]
Memory	8.39 Megabytes (up to 1 048 576 words)[1]
Storage	303 Megabytes (DD19 Unit)[1]
FLOPS	160 MFLOPS

https://en.wikipedia.org/wiki/Cray-1

Piz Daint: One of Today's Most Powerful Supercomputers

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

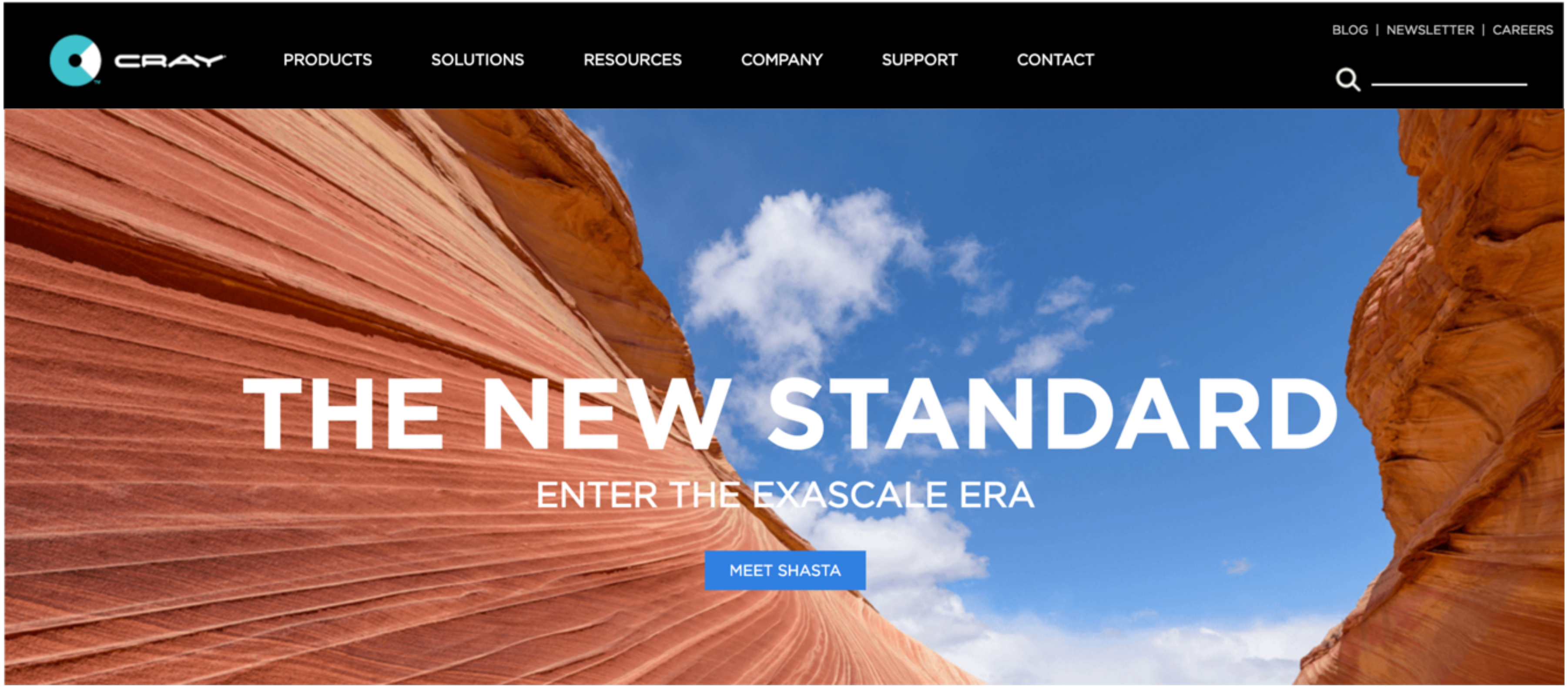
Piz Daint: One of Today's Most Powerful Supercomputers



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	
Mulitcore 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		

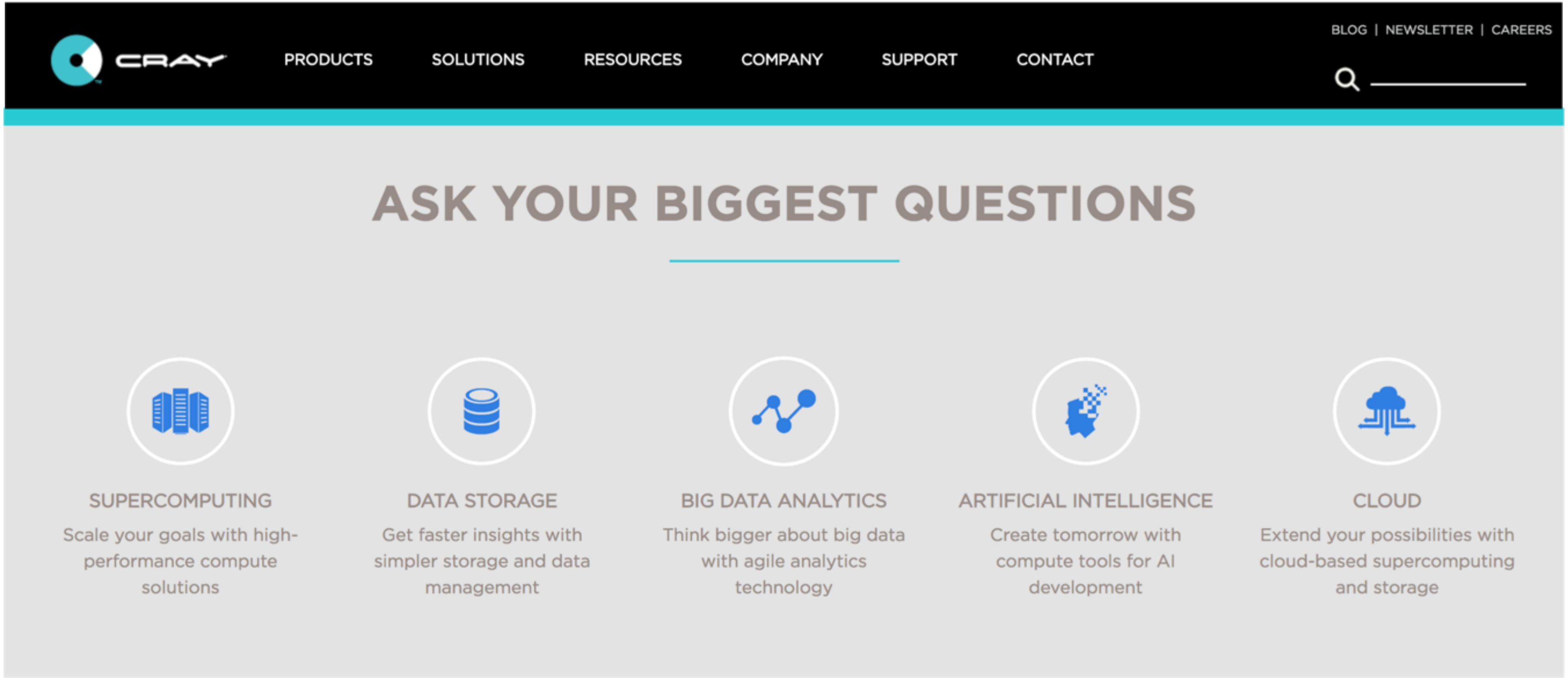
https://www.cscs.ch/computers/piz-daint/

Cray: The Supercomputer Company



https://www.cray.com

Cray: The Supercomputer Company



https://www.cray.com

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

Why might a PuPPy member care about Chapel?

- Chapel is not Python…

    …yet many Python programmers have found it attractive and approachable
- You may want to consider Chapel in order to…

    …get good **performance** without resorting to C

    …easily express **multi-core parallelism** on your laptop / desktop

    …do **distributed programming** on a personal cluster or cloud resource

    …**scale up** from your laptop to the largest supercomputers

    …get **static typing** benefits in a type-inferred language
- Chapel is increasingly interoperable with Python

Outline

✓ Context for this talk

➢ Productivity and Chapel

- Overview of Chapel Features
- Chapel Results and 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)

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

  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

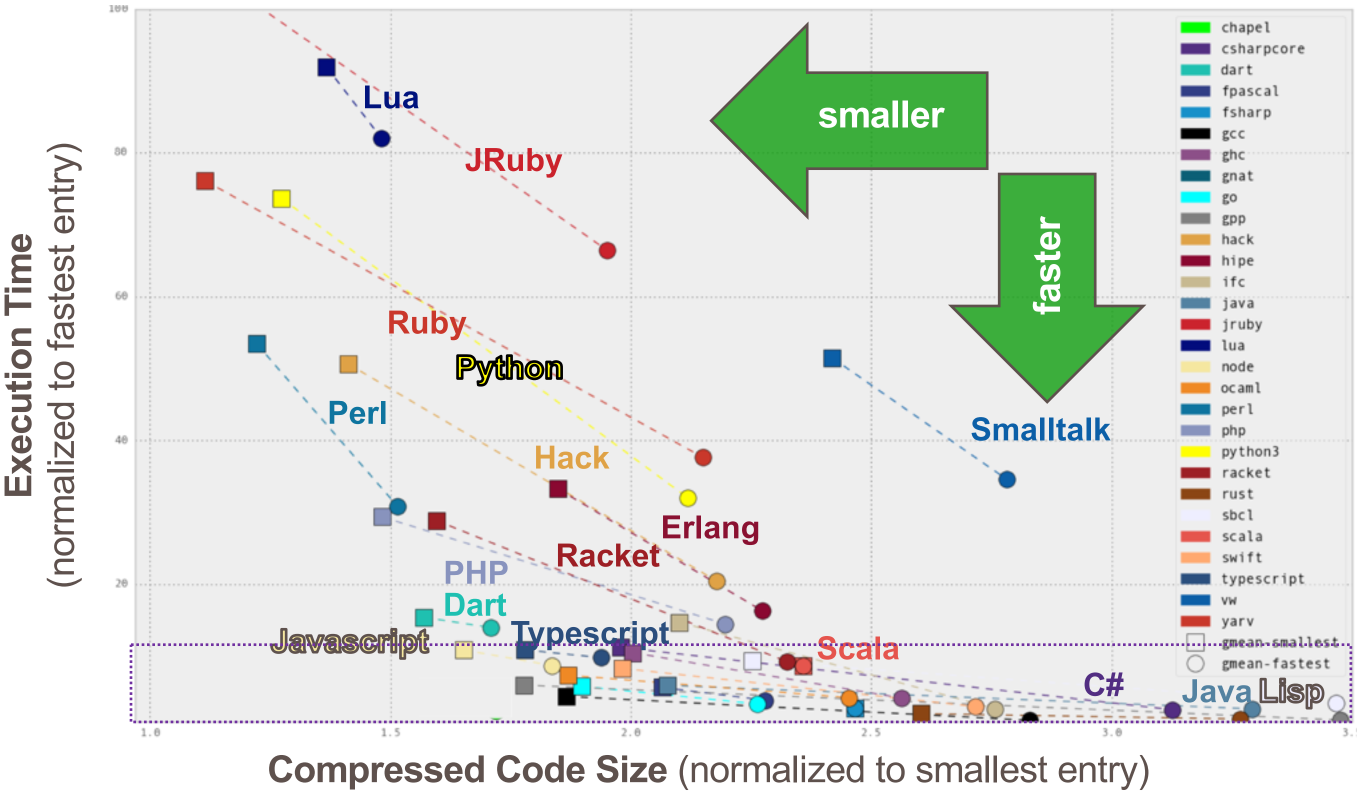
description

program source code, command-line and measurements

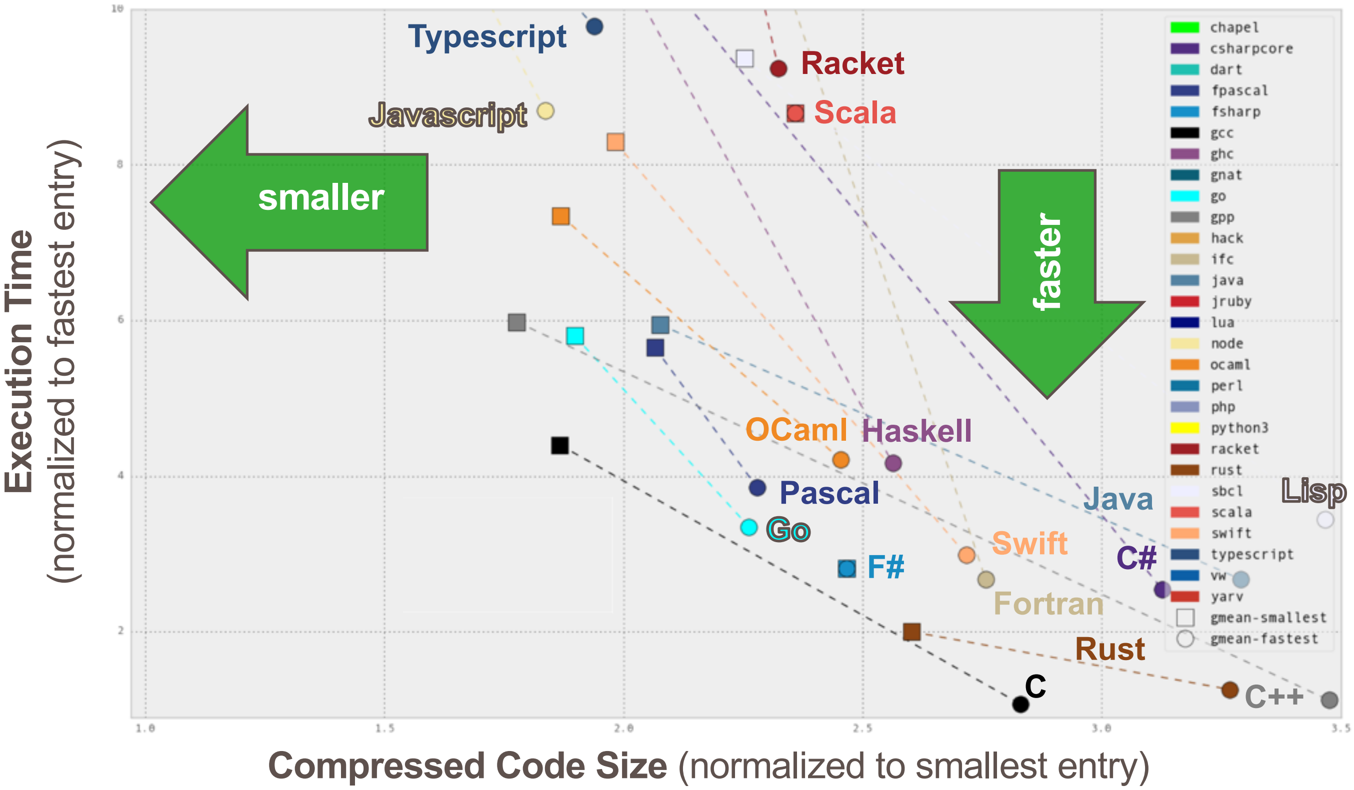
×	source	secs	mem	gz	cpu	cpu load
1.0	**Perl** #4	3.50	7,348	**261**	3.50	100% 1% 1% 1%
1.5	**Python 3** #2	3.51	10,500	**386**	3.50	1% 1% 0% 100%
1.5	**PHP** #4	2.12	10,512	**389**	2.12	100% 0% 0% 2%
1.5	Perl #2	3.83	7,320	389	3.83	2% 1% 100% 1%
1.5	PHP #5	2.12	10,664	399	2.11	100% 0% 1% 1%
1.6	**Chapel** #2	1.62	6,484	**423**	1.63	[illegible]
1.7	**C** gcc	1.75	2,728	**452**	1.74	[illegible]
1.7	**Racket**	27.58	124,156	**453**	27.56	[illegible]
1.8	**OCaml** #5	6.72	19,836	**458**	6.71	[illegible]
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

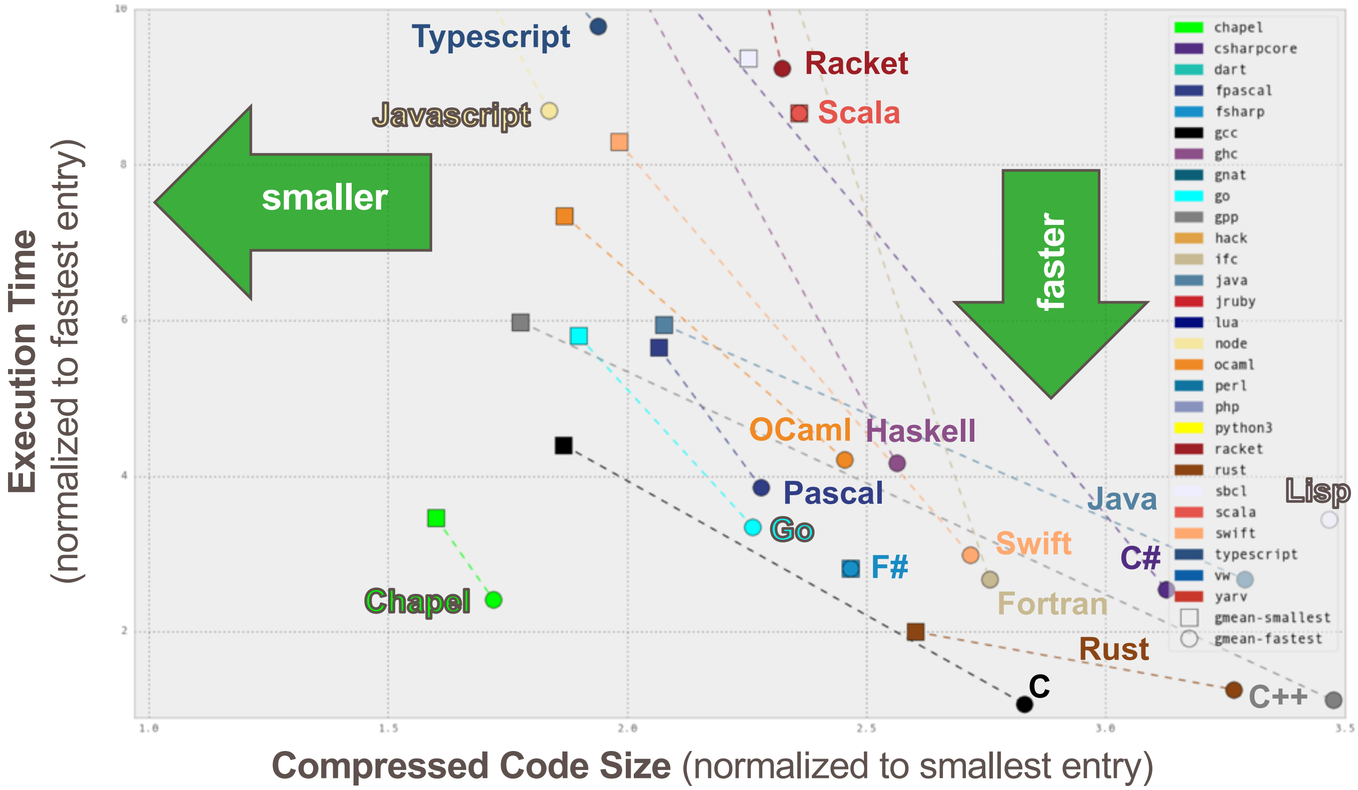
CLBG Cross-Language Summary
(September 21, 2018 standings)



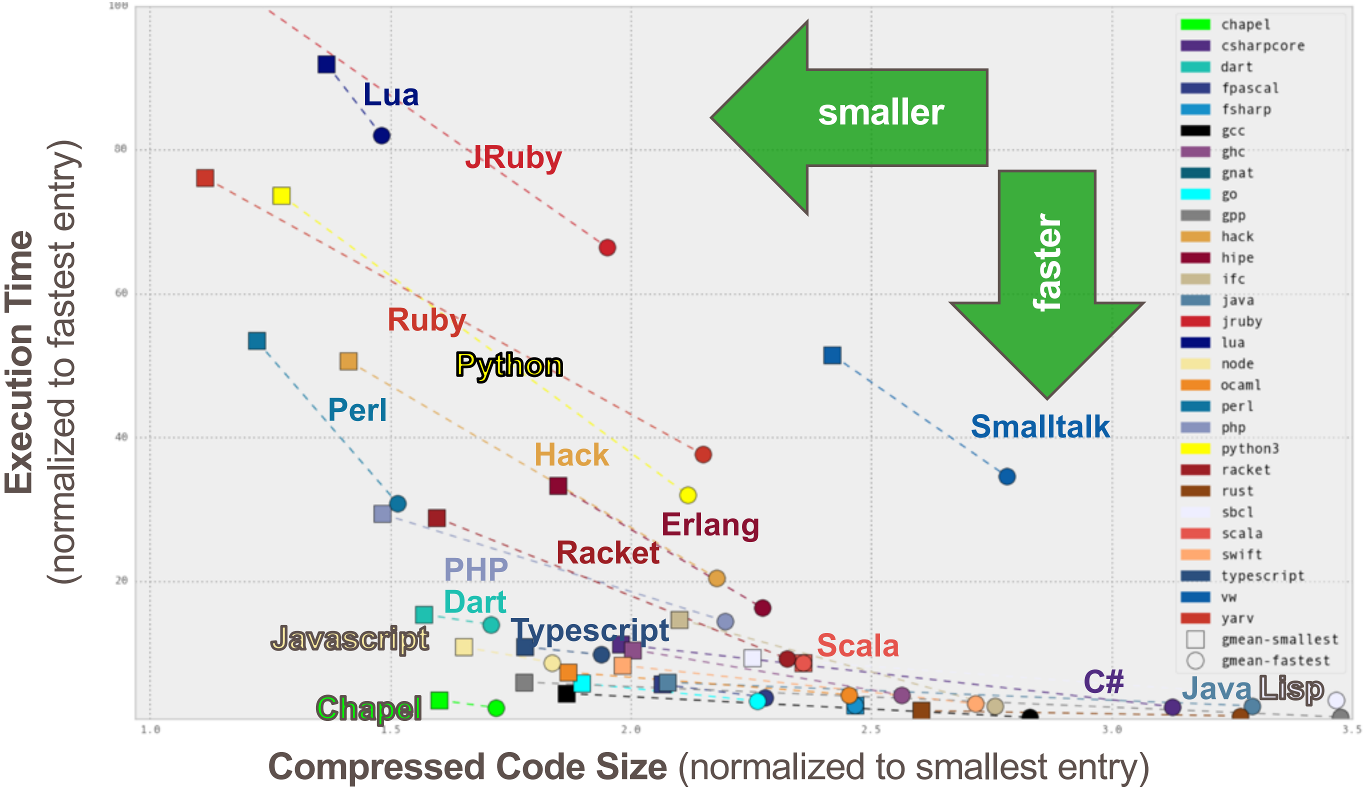
CLBG Cross-Language Summary
(September 21, 2018 standings, zoomed in)



CLBG Cross-Language Summary
(September 21, 2018 standings, zoomed in)



CLBG Cross-Language Summary
(September 21, 2018 standings)



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!)*:

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

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

  coforall c in population do               // creat
    c.haveMeetings(place, population);

  delete place;
}
```

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(cpu

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;
}
```

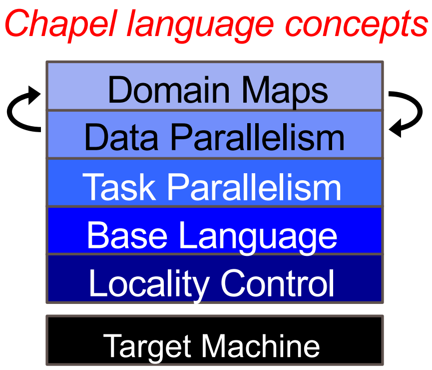
excerpt from 1210 gz Chapel entry

```
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);
```

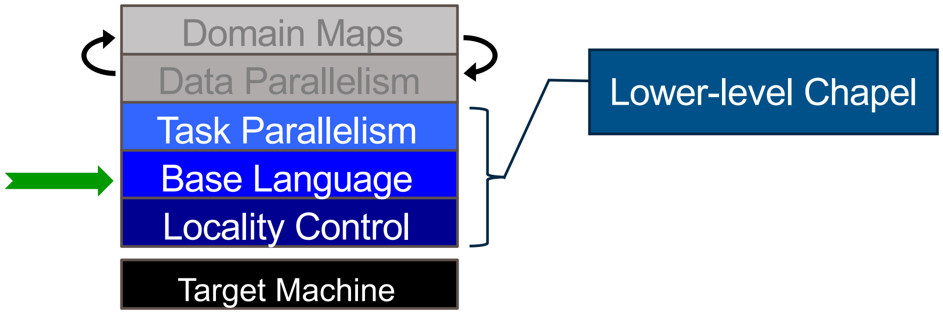
excerpt from 2863 gz C gcc entry

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

Iterators

```
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

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

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

```
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

Zippered iteration

```
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

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

Tuples

```
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

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

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

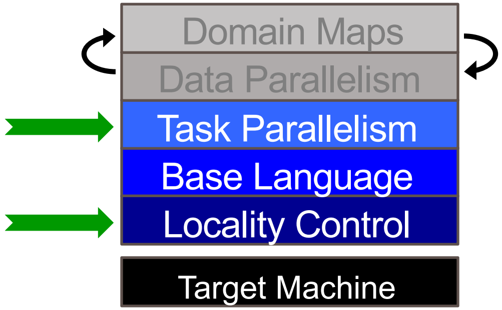
```
config const n = 10;

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…

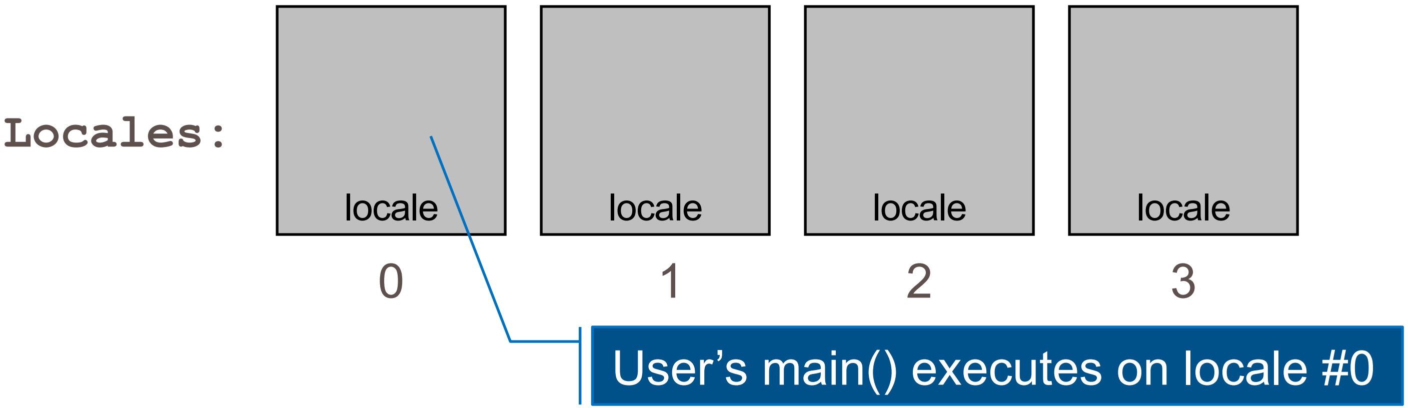
Task Parallelism and Locality Control



Locales, briefly

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

```
> ./myProgram --numLocales=4
```



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

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

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

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

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

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

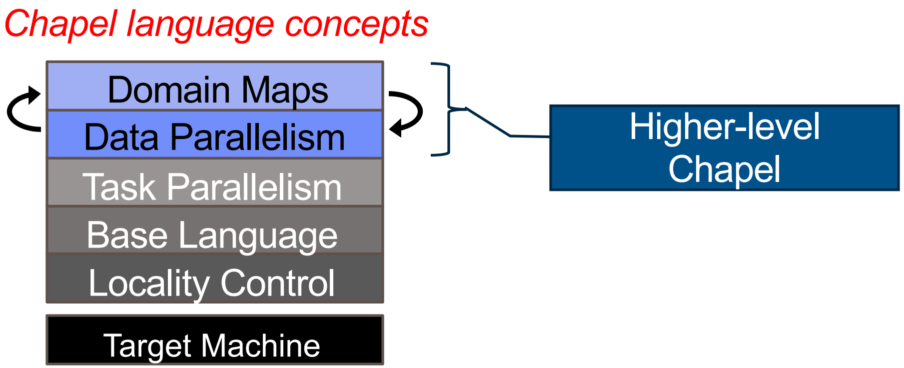
```
coforall loc in Locales do
  on loc {
    const numTasks = here.numPUs();
    coforall tid in 1..numTasks do
      writef("Hello from task %n of %n "+
             "running on %s\n",
             tid, numTasks, here.name);
  }
```

```
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

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

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
```

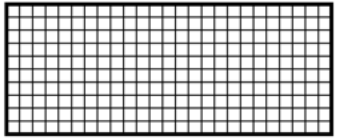
Distributed Data Parallelism, by example

dataParallel.chpl

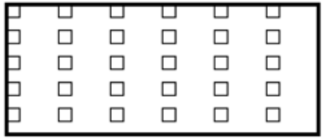
```
use CyclicDist;
config const n = 1000;
var D = {1..n, 1..n}
          dmapped Cyclic(startIdx = (1,1));
var A: [D] real;
forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

```
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
```

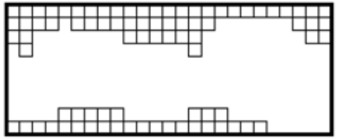
Chapel Has Several Domain / Array Types



dense

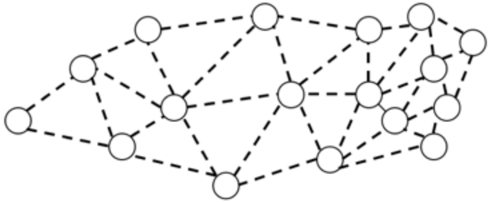


strided



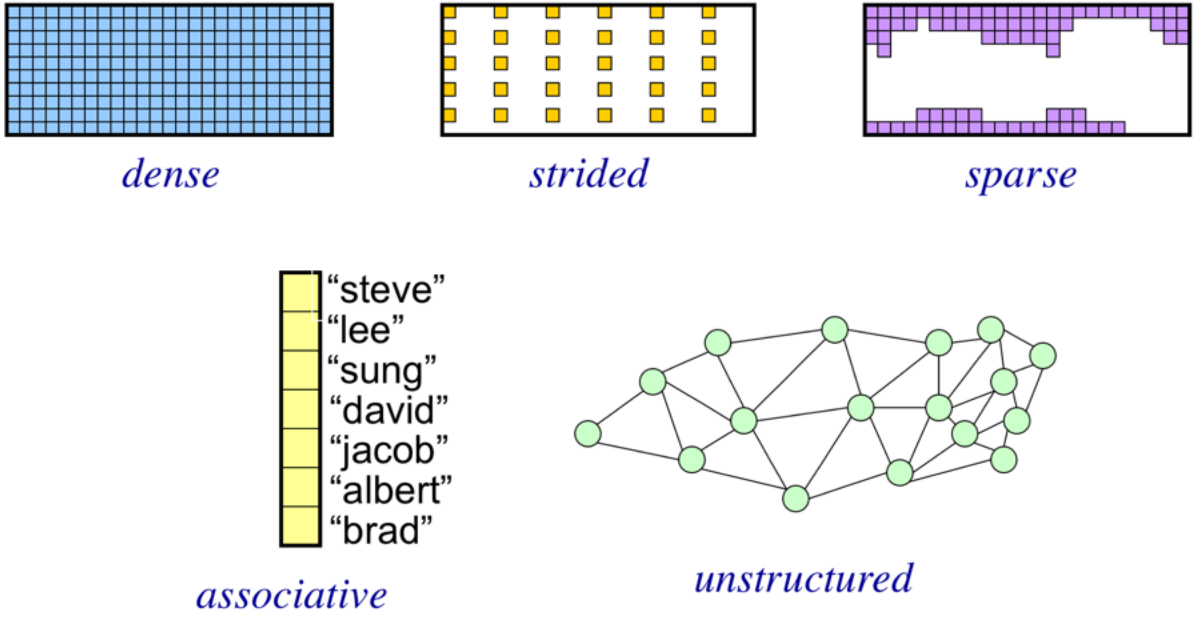
sparse

associative

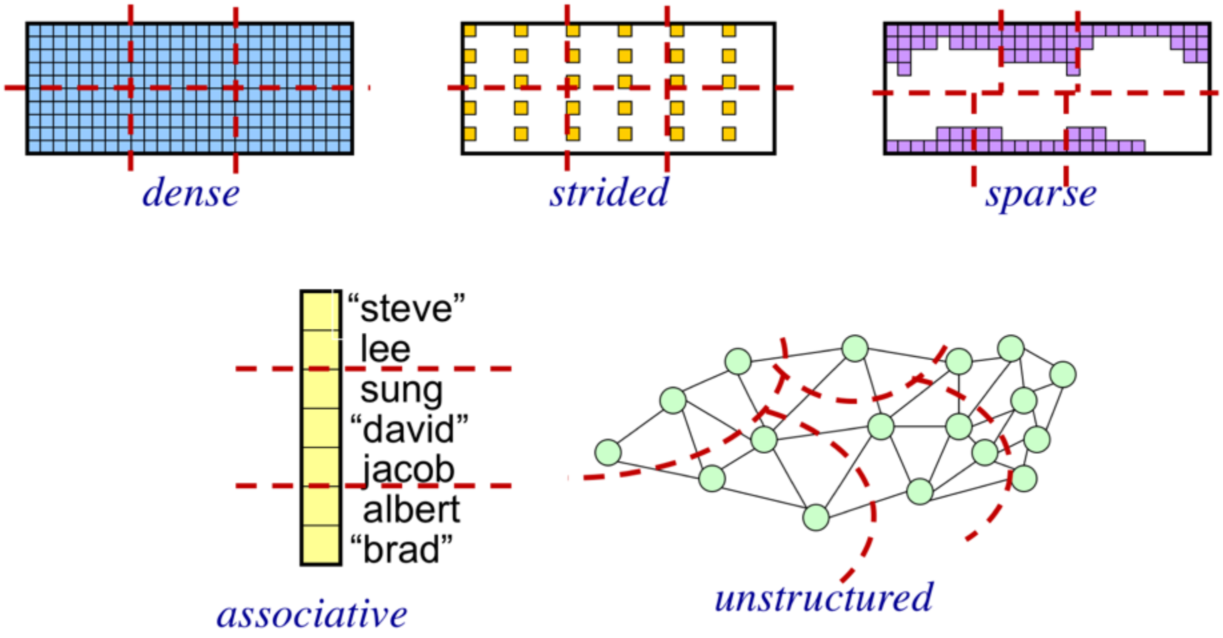


unstructured

Chapel Has Several Domain / Array Types



Chapel Has Several Domain / Array Types

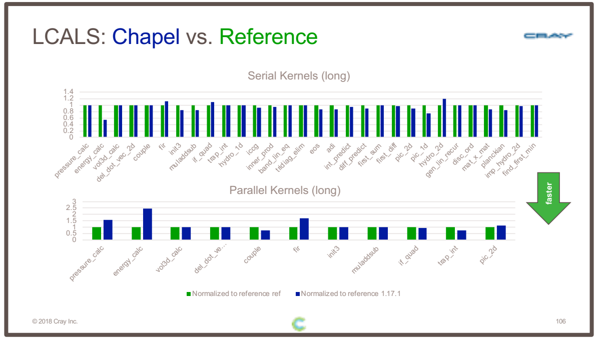


Other Data Parallel Features

- **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**

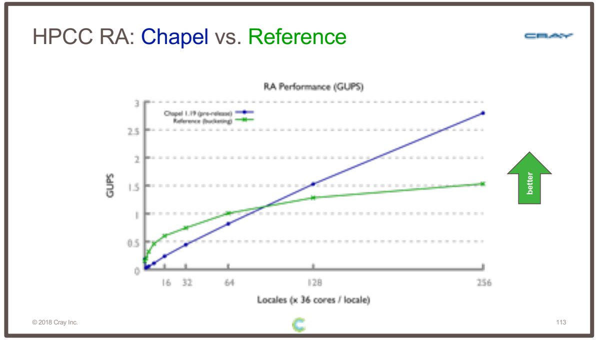
Chapel Results and Resources

HPC Patterns: Chapel vs. Reference



LCALS

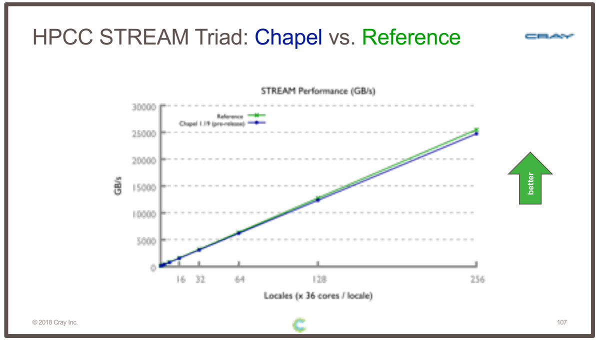
HPCC RA

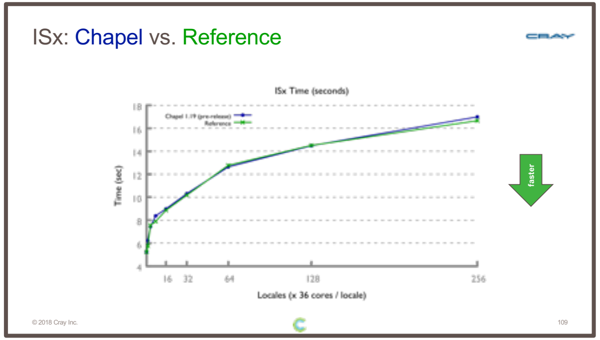


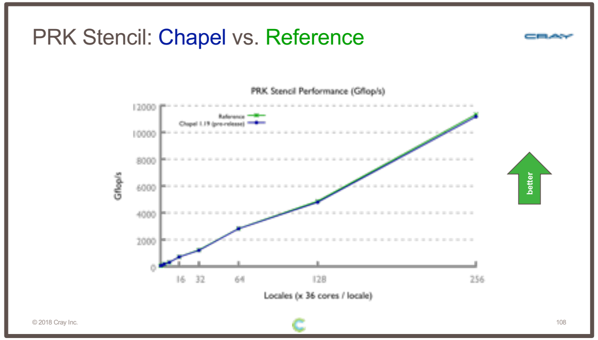
STREAM Triad

ISx

PRK Stencil

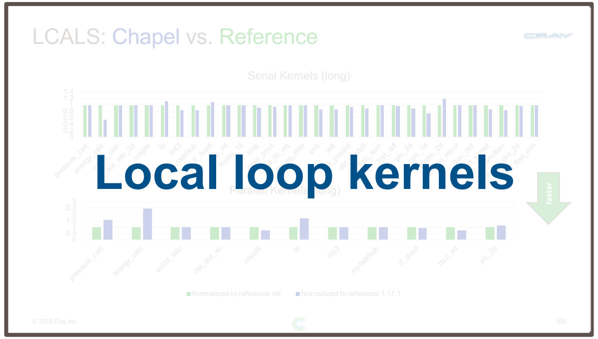






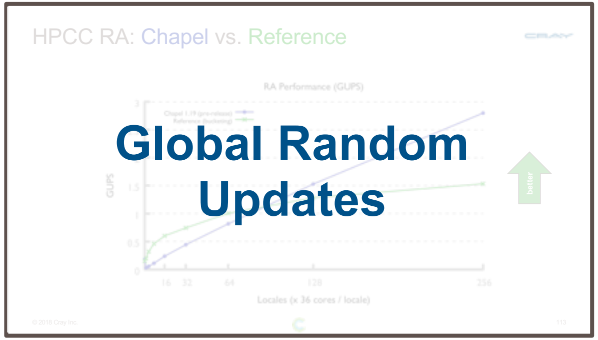
Nightly performance tickers online at:
https://chapel-lang.org/perf-nightly.html

HPC Patterns: Chapel vs. Reference



LCALS

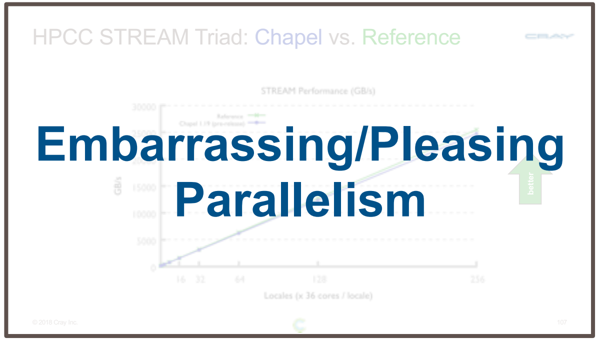
HPCC RA

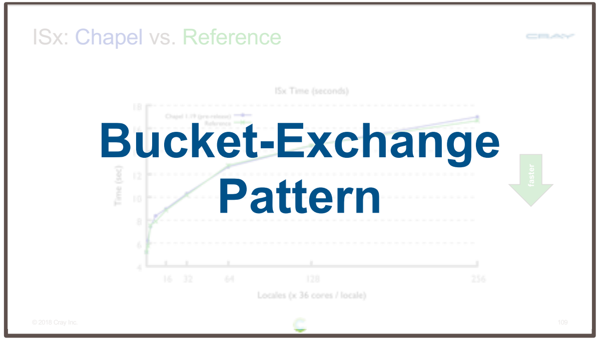


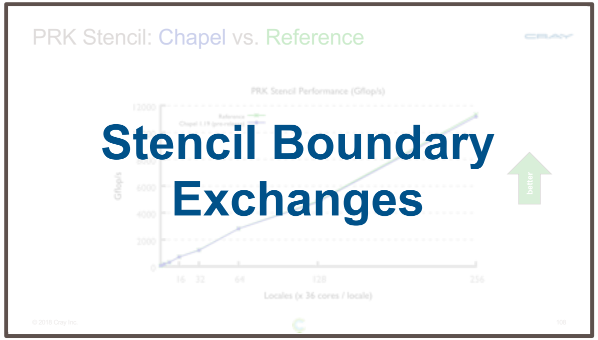
STREAM Triad

ISx

PRK Stencil

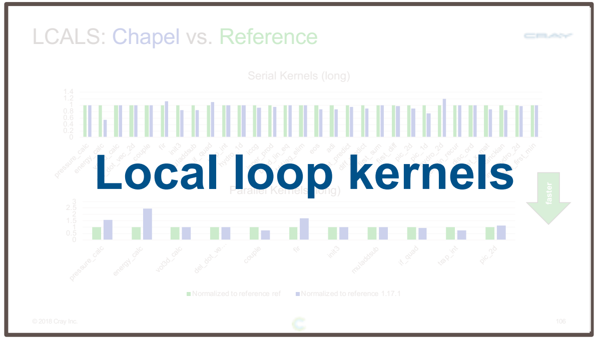






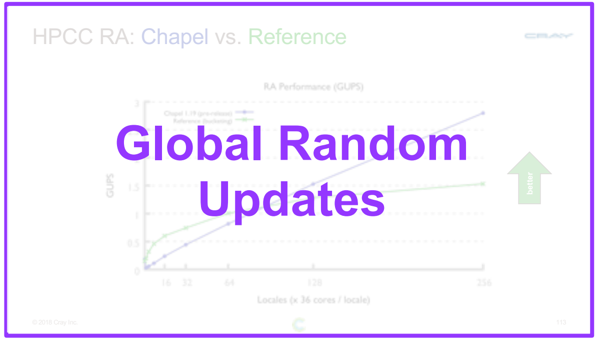
Nightly performance tickers online at:
https://chapel-lang.org/perf-nightly.html

HPC Patterns: Chapel vs. Reference



LCALS

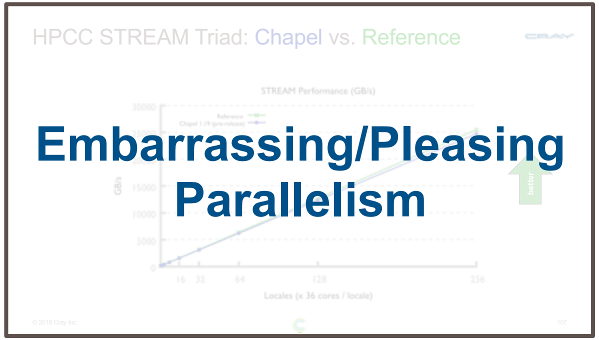
HPCC RA

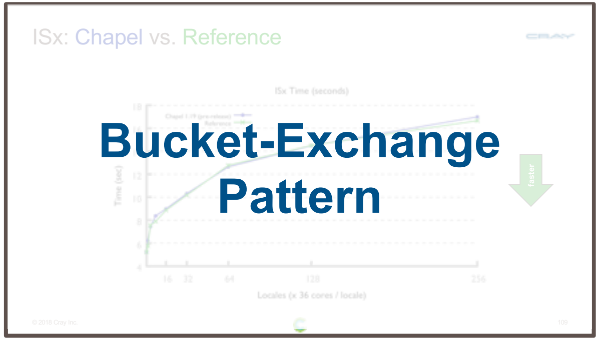


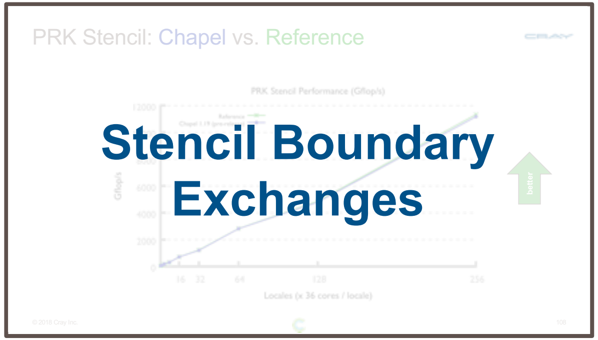
STREAM Triad

ISx

PRK Stencil

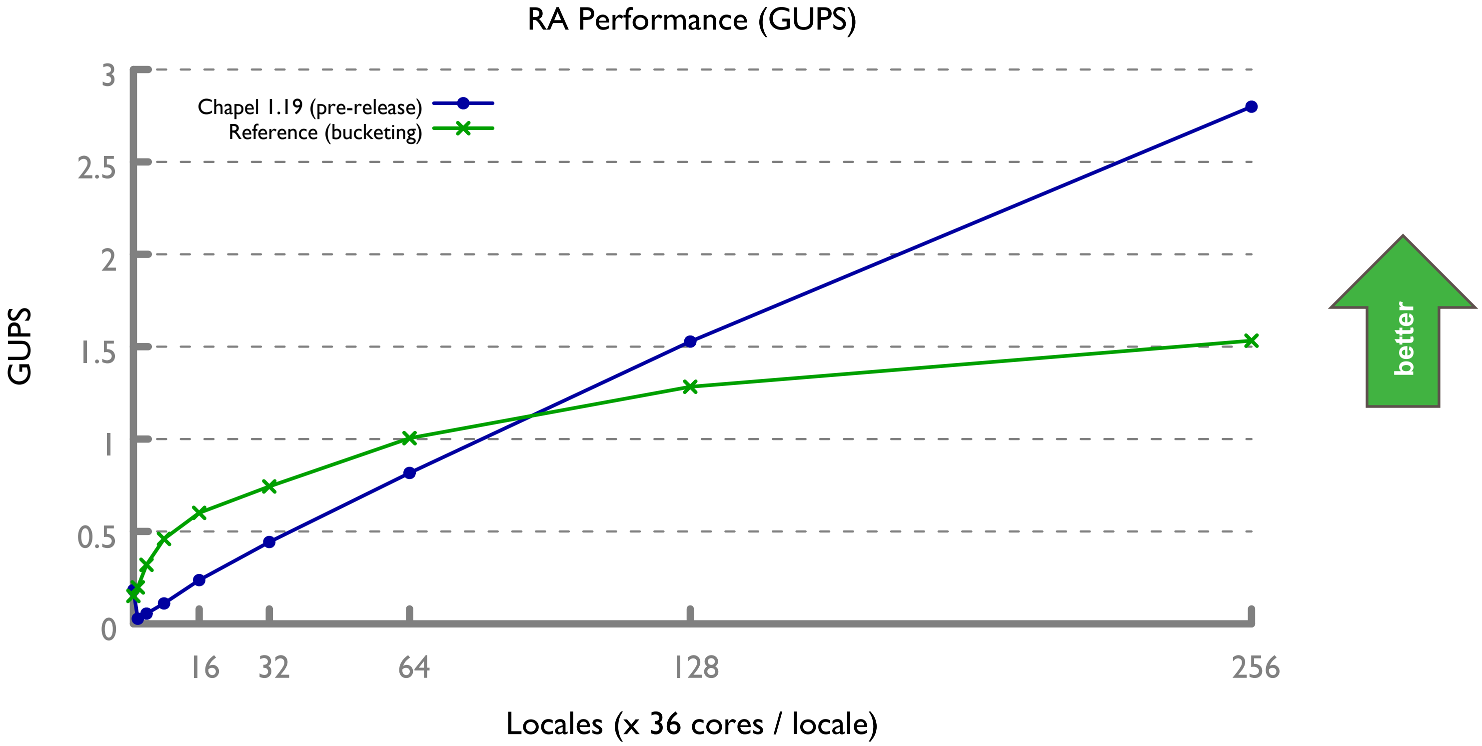






Nightly performance tickers online at:
https://chapel-lang.org/perf-nightly.html

HPCC RA: Chapel vs. Reference



HPCC Random Access Kernel: MPI

```
/* 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 Random Access Kernel: MPI

```
/* 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;
 *    }
 */
```

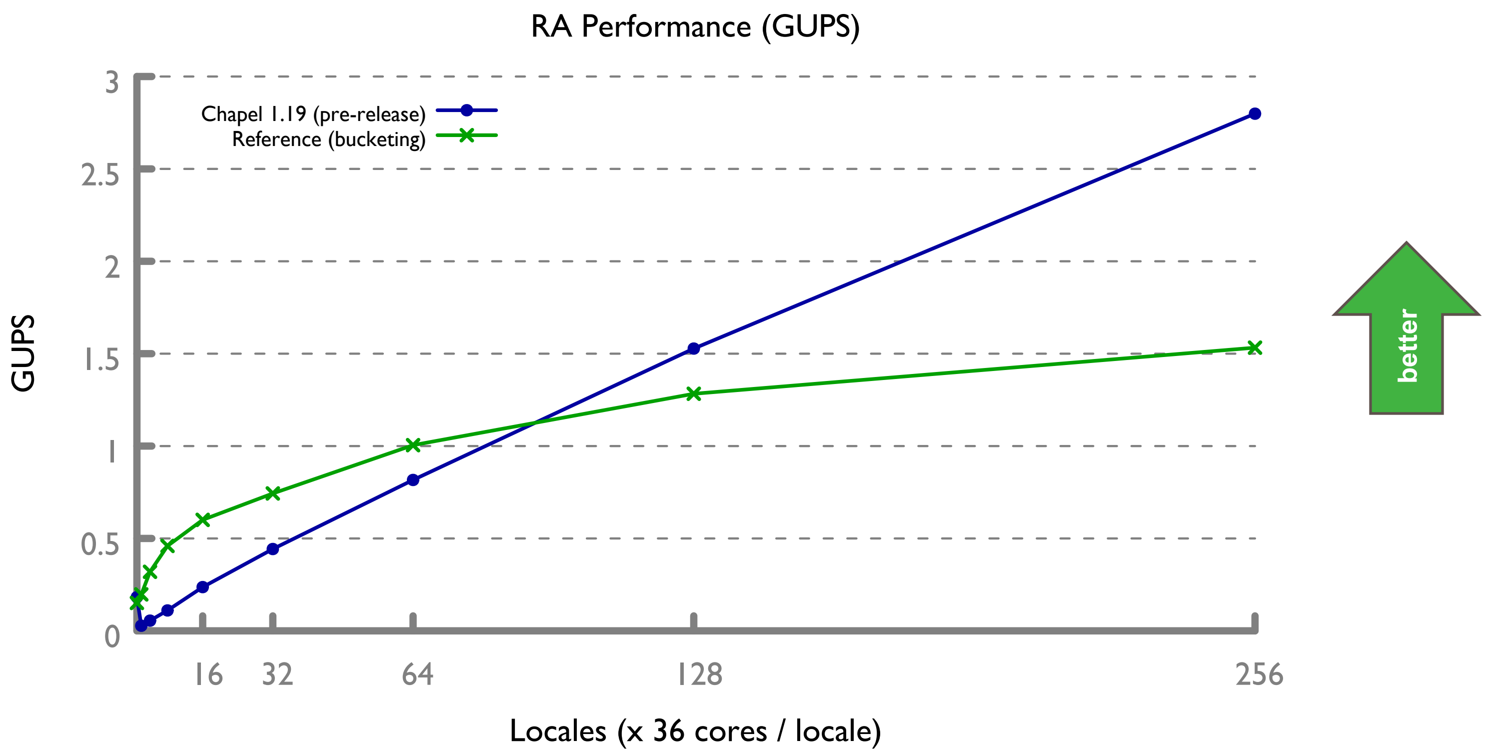
Chapel Kernel

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

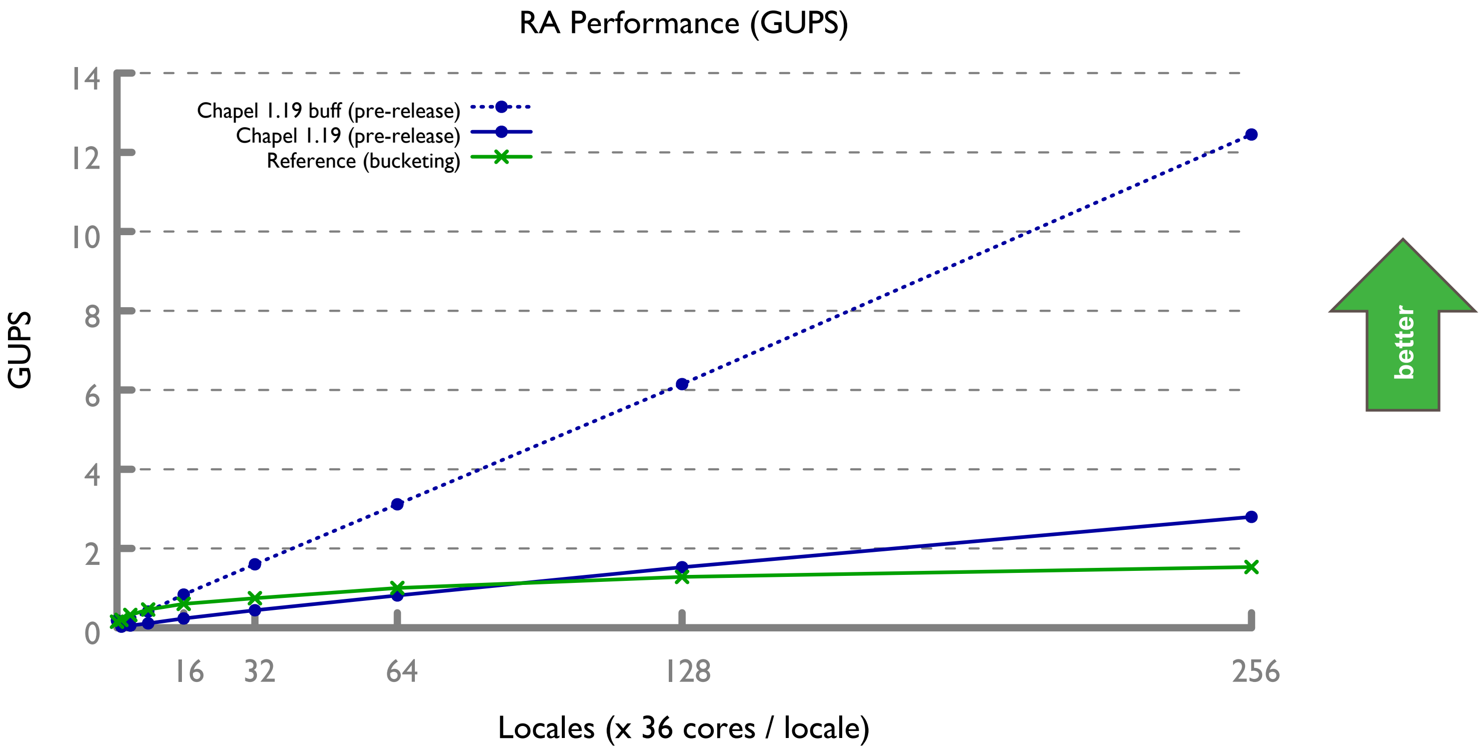
MPI Comment

```
/* 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

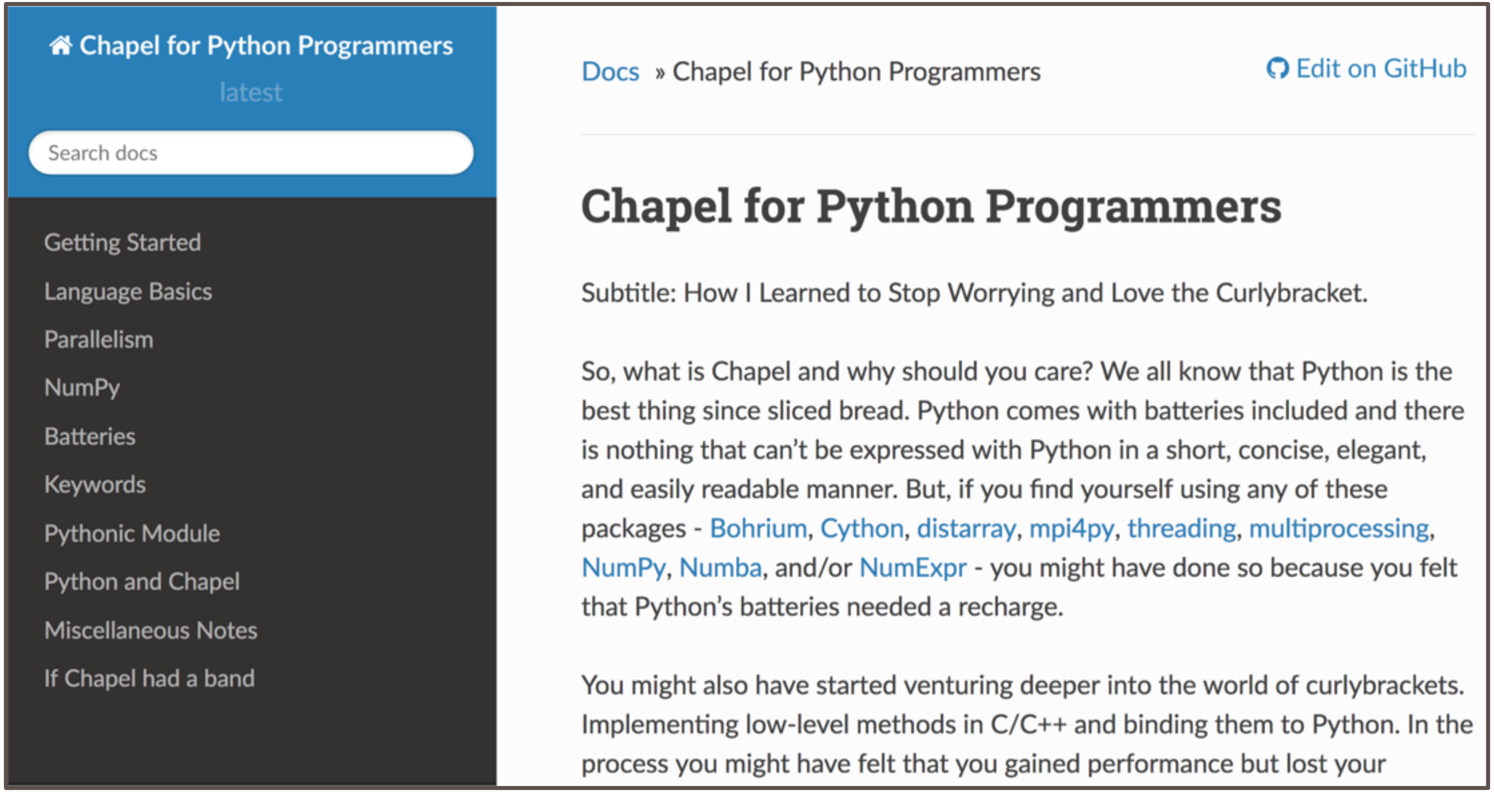


HPCC RA: Chapel vs. Reference (w/ buffered atomics)



Chapel for Python Programmers

Developed by Simon Lund



Chapel for Python Programmers

latest

Search docs

- Getting Started
- Language Basics
- Parallelism
- NumPy
- Batteries
- Keywords
- Pythonic Module
- Python and Chapel
- Miscellaneous Notes
- If Chapel had a band

Docs » Chapel for Python Programmers

Edit on GitHub

Chapel for Python Programmers

Subtitle: How I Learned to Stop Worrying and Love the Curlybracket.

So, what is Chapel and why should you care? We all know that Python is the best thing since sliced bread. Python comes with batteries included and there is nothing that can't be expressed with Python in a short, concise, elegant, and easily readable manner. But, if you find yourself using any of these packages - Bohrium, Cython, distarray, mpi4py, threading, multiprocessing, NumPy, Numba, and/or NumExpr - you might have done so because you felt that Python's batteries needed a recharge.

You might also have started venturing deeper into the world of curlybrackets. Implementing low-level methods in C/C++ and binding them to Python. In the process you might have felt that you gained performance but lost your

https://chapel-for-python-programmers.readthedocs.io/

Python ↔ Chapel Interoperability

- We’ve recently added support for calling from Python to Chapel
  - Exposes Chapel libraries as Python modules
  - Uses compiler-generated Cython files under the hood
- Users have extended this to write Chapel cells within Jupyter, calling from Python
- Work remains to support additional types and usage patterns

- For more information, see:

  https://chapel-lang.org/docs/technotes/libraries.html#using-your-library-in-python

The Chapel Team at Cray (May 2018)

Chapel is Currently Hiring

- Our team has two positions open at present
- An ideal candidate would have experience in:
  - parallel, concurrent, and/or distributed computing
  - compilers
- But more important are software developers…

  …with a passion for creating a great parallel language

  …who are fearless in tackling the related technical and social challenges

Chapel Community Partners

CRAY

HAVERFORD COLLEGE

AMD

THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC

WESTERN
WASHINGTON UNIVERSITY

東京大学
THE UNIVERSITY OF TOKYO

THE UNIVERSITY OF ARIZONA

RICE

UNIVERSITY OF MARYLAND

BERKELEY LAB
Lawrence Berkeley National Laboratory

Lawrence Livermore National Laboratory

Sandia National Laboratories

DEPARTMENT OF DEFENSE
UNITED STATES OF AMERICA

Yale

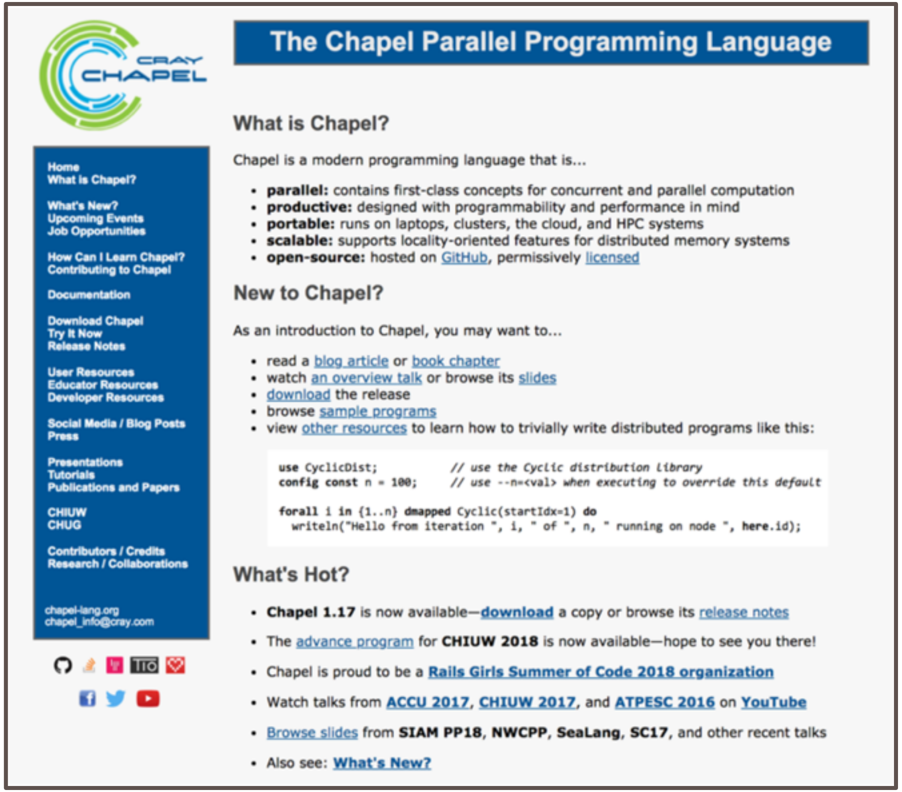
(and several others…)

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

Chapel Central

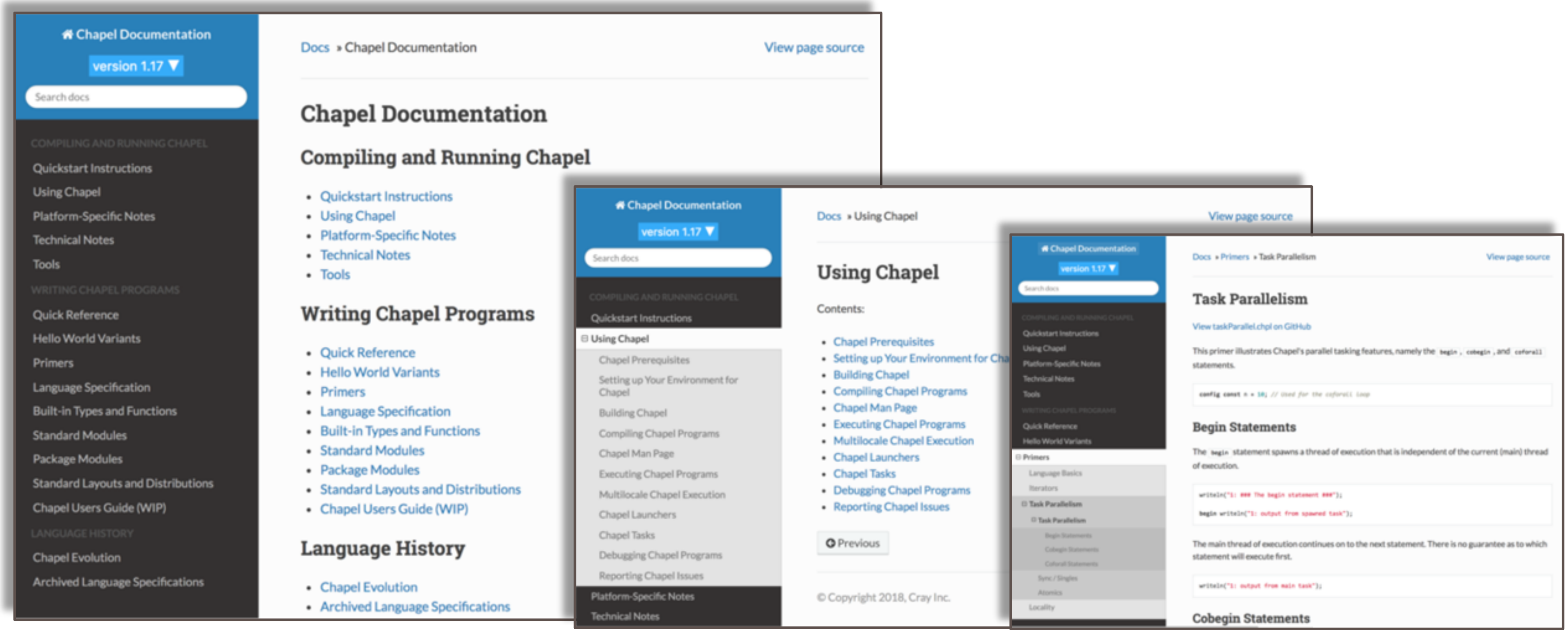
https://chapel-lang.org

- downloads
- presentations
- papers
- resources
- documentation



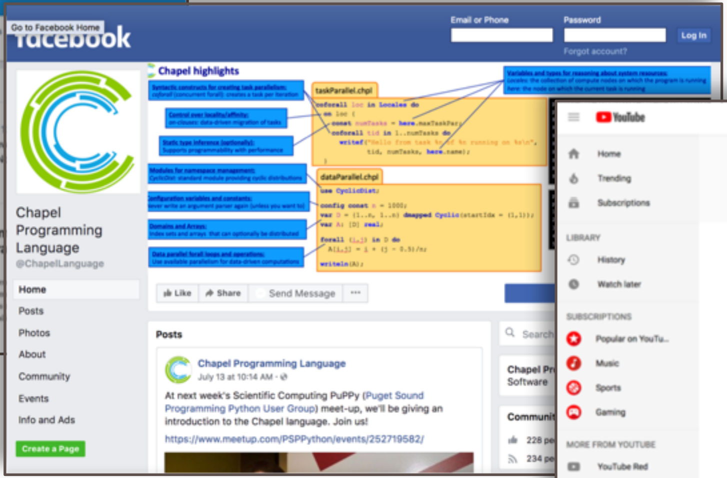
Chapel Online Documentation

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

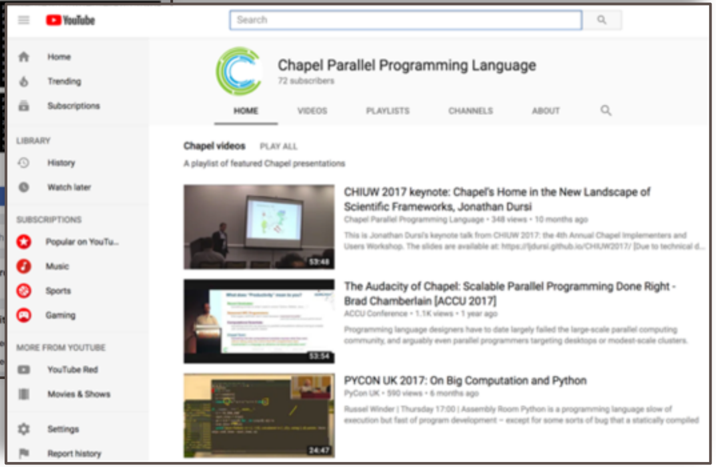


Chapel Social Media (no account required)

http://twitter.com/ChapelLanguage

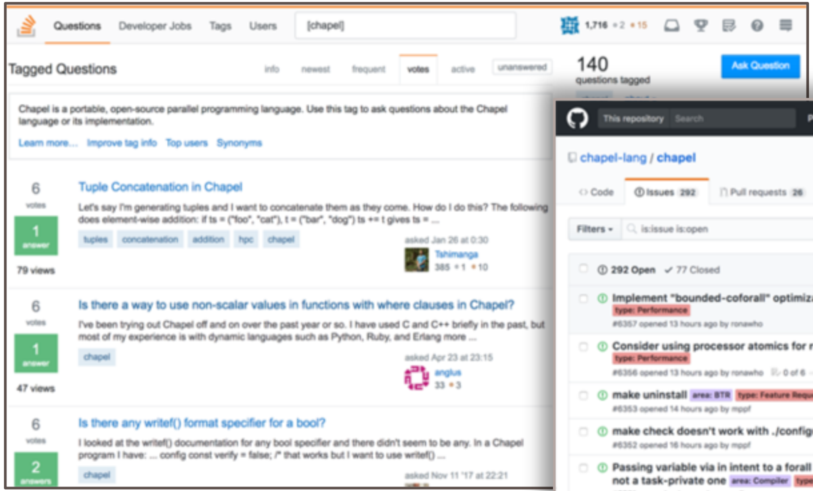


http://facebook.com/ChapelLanguage

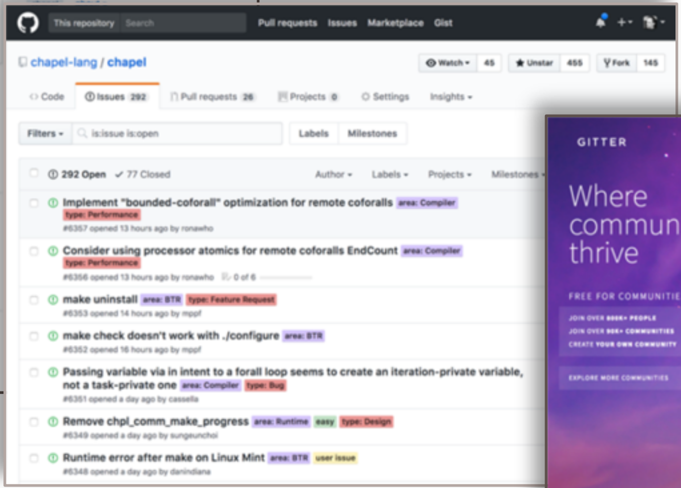


https://www.youtube.com/channel/UCHmm27bYjhknK5mU7ZzPGsQ/

Chapel Community



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



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

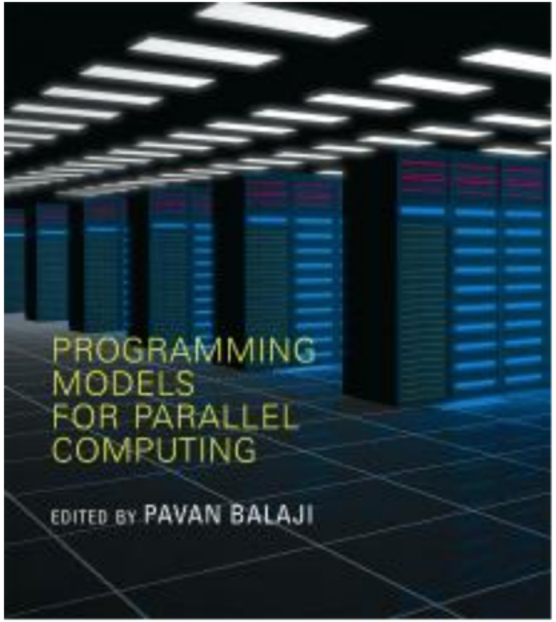
https://gitter.im/chapel-lang/chapel

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

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]

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.

[1] https://github.com/chapel-lang/chapel

paper and slides available at chapel-lang.org

Summary and Wrap-up

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

Chapel may be attractive to Python programmers seeking performance, parallelism, scalability, and/or static typing

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

We are hiring!

I'll be available afterwards for questions, discussion, demos, etc.

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