What is Chapel?

Chapel: A modern parallel programming language

- portable & scalable
- open-source & collaborative

Goals:

- Support general parallel programming
  - “any parallel algorithm on any parallel hardware”
- Make parallel programming at scale far more productive
What does “Productivity” mean to you?

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

**Seasoned HPC Programmers:**
“that sugary stuff which I don’t need because I require full control to get performance”

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

**Chapel Team:**
“something that lets computational scientists express what they want, without taking away the control that HPC programmers want, implemented in a language as attractive as recent graduates want.”
Why Consider New Languages at all?

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]
Comparing Chapel to Other Languages

Chapel aims to be as...

...**programmable** as Python
...**fast** as Fortran
...**scalable** as MPI, SHMEM, or UPC
...**portable** as C
...**flexible** as C++
...**fun** as [your favorite programming language]
Outline

✓ Context and Motivation

➢ A Brief Tour of Chapel Features
  • Chapel Evaluations
  • Summary and Resources
Chapel Feature Areas

Chapel language concepts

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

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

Lower-level Chapel
Base Language Features, by example

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

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

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

0
1
1
2
3
5
8
...
Base Language Features, by example

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

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

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

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

0
1
1
2
3
5
8...

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

CLU-style iterators

Modern iterators

Iterators
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);

Static type inference for:
• arguments
• return types
• variables
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);
```

Explicit types also supported
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

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

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

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

fib #0 is 0
fib #1 is 1
fib #2 is 1
fib #3 is 2
fib #4 is 3
fib #5 is 5
fib #6 is 8
...```

Zippered iteration
Base Language Features, by example

Range types and operators

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

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

```plaintext
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

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

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

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

Tuples

```
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

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

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

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

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

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

Locales:

0 1 2 3

User’s code starts executing on locale #0
Task Parallelism and Locality, by example

```chpl
const numTasks = here.numPUs();
coforall tid in 1..numTasks do
    writeln("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

```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

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

```bash
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

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

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

```
prompt> chpl taskParallel.chpl
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

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

```
prompt> chpl taskParallel.chpl
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

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

```
prompt> chpl taskParallel.chpl
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

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

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

**Control of Locality/Affinity**

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

```bash
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

```chapel
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
- **task intents**: for specifying how outer-scope variables are passed to tasks
Data Parallelism in Chapel

Chapel language concepts

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

Higher-level Chapel
Data Parallelism, by example

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

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

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

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

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

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

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

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

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

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

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

```
prompt> chpl dataParallel.chpl
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

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

```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

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);
```

Domain Maps (Map Data Parallelism to the System)

```
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

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

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

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

© 2019 Cray Inc.
Chapel Evaluations
Website supporting cross-language comparisons

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

Chapel’s approach to the CLBG:
- striving for elegance over heroism
  - ideally: “Want to learn how program \textit{xyz} works? Read the Chapel version.”
Can sort results by various metrics: execution time, code size, memory use, CPU use:

<table>
<thead>
<tr>
<th>Source</th>
<th>Secs</th>
<th>Mem</th>
<th>Gz</th>
<th>Cpu</th>
<th>Cpu Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapel #2</td>
<td>6.484</td>
<td>423</td>
<td>1.63</td>
<td>99% 1%</td>
<td>1% 2%</td>
</tr>
<tr>
<td>Chapel</td>
<td>6.488</td>
<td>501</td>
<td>1.63</td>
<td>99% 1%</td>
<td>1% 1%</td>
</tr>
<tr>
<td>Free Pascal #3</td>
<td>2,428</td>
<td>530</td>
<td>1.72</td>
<td>0% 2%</td>
<td>100% 1%</td>
</tr>
<tr>
<td>Rust #3</td>
<td>4,488</td>
<td>1,366</td>
<td>1.74</td>
<td>1% 100%</td>
<td>0% 1%</td>
</tr>
<tr>
<td>Rust</td>
<td>4,616</td>
<td>1,420</td>
<td>1.74</td>
<td>1% 100%</td>
<td>0% 1%</td>
</tr>
<tr>
<td>Rust #2</td>
<td>4,636</td>
<td>1,306</td>
<td>1.74</td>
<td>1% 100%</td>
<td>0% 1%</td>
</tr>
<tr>
<td>C gcc</td>
<td>2,728</td>
<td>452</td>
<td>1.74</td>
<td>1% 2%</td>
<td>0% 1%</td>
</tr>
<tr>
<td>Ada 2012 GNAT #2</td>
<td>4,312</td>
<td>1,068</td>
<td>1.75</td>
<td>1% 0%</td>
<td>100% 0%</td>
</tr>
<tr>
<td>Swift #2</td>
<td>8,492</td>
<td>601</td>
<td>1.76</td>
<td>1% 100%</td>
<td>1% 1%</td>
</tr>
<tr>
<td>Lisp SBCL #4</td>
<td>20,196</td>
<td>940</td>
<td>1.79</td>
<td>1% 2%</td>
<td>1% 100%</td>
</tr>
<tr>
<td>C++ g++ #4</td>
<td>4,284</td>
<td>513</td>
<td>1.88</td>
<td>5% 0%</td>
<td>1% 100%</td>
</tr>
<tr>
<td>Go #3</td>
<td>8,976</td>
<td>603</td>
<td>2.04</td>
<td>1% 0%</td>
<td>1% 100%</td>
</tr>
<tr>
<td>PHP #5</td>
<td>10,664</td>
<td>399</td>
<td>2.11</td>
<td>100% 0%</td>
<td>1% 1%</td>
</tr>
<tr>
<td>PHP #4</td>
<td>10,512</td>
<td>389</td>
<td>2.12</td>
<td>100% 0%</td>
<td>2%</td>
</tr>
</tbody>
</table>

gz == code size metric
strip comments and extra whitespace, then gzip
Can also compare languages pair-wise:

- but only sorted by execution speed...

---

<table>
<thead>
<tr>
<th>Chapel versus C++ g++ fastest programs</th>
<th>Chapel versus Python 3 fastest programs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>vs C</strong> vs C++ vs Go vs Java vs Python</td>
<td><strong>vs C</strong> vs C++ vs Go vs Java vs Python</td>
</tr>
<tr>
<td>by faster benchmark performance</td>
<td>by faster benchmark performance</td>
</tr>
<tr>
<td><strong>reverse-complement</strong></td>
<td><strong>mandelbrot</strong></td>
</tr>
<tr>
<td><strong>source</strong></td>
<td><strong>source</strong></td>
</tr>
<tr>
<td><strong>secs</strong></td>
<td><strong>secs</strong></td>
</tr>
<tr>
<td><strong>mem</strong></td>
<td><strong>C++</strong></td>
</tr>
<tr>
<td><strong>giz</strong></td>
<td><strong>cpu</strong></td>
</tr>
<tr>
<td><strong>gpu</strong></td>
<td><strong>cpu load</strong></td>
</tr>
<tr>
<td>Chapel 2.20 1,497,876 707 5.10 96% 42% 58% 38%</td>
<td>Chapel 5.09 36,328 620 20.09 99% 99% 99% 99%</td>
</tr>
<tr>
<td>C++ g++ 2.95 980,472 2280 4.56 50% 41% 16% 50%</td>
<td>Python 3 279.68 49,344 688 1,117.29 100% 100% 100% 100%</td>
</tr>
<tr>
<td><strong>digits</strong></td>
<td><strong>spectral-norm</strong></td>
</tr>
<tr>
<td><strong>source</strong></td>
<td><strong>source</strong></td>
</tr>
<tr>
<td><strong>secs</strong></td>
<td><strong>secs</strong></td>
</tr>
<tr>
<td><strong>mem</strong></td>
<td><strong>mem</strong></td>
</tr>
<tr>
<td><strong>giz</strong></td>
<td><strong>giz</strong></td>
</tr>
<tr>
<td><strong>gpu</strong></td>
<td><strong>gpu</strong></td>
</tr>
<tr>
<td>Chapel 1.62 6,488 501 1.63 99% 1% 1% 1%</td>
<td>Chapel 3.97 5,488 310 15.75 99% 99% 99% 99%</td>
</tr>
<tr>
<td>C++ g++ 1.89 4,284 513 1.88 5% 0% 1% 100%</td>
<td>Python 3 193.86 50,556 443 757.23 99% 98% 99% 99%</td>
</tr>
<tr>
<td><strong>fannkuch-redux</strong></td>
<td><strong>fannkuch-redux</strong></td>
</tr>
<tr>
<td><strong>source</strong></td>
<td><strong>source</strong></td>
</tr>
<tr>
<td><strong>secs</strong></td>
<td><strong>secs</strong></td>
</tr>
<tr>
<td><strong>mem</strong></td>
<td><strong>mem</strong></td>
</tr>
<tr>
<td><strong>giz</strong></td>
<td><strong>giz</strong></td>
</tr>
<tr>
<td><strong>gpu</strong></td>
<td><strong>gpu</strong></td>
</tr>
<tr>
<td>Chapel 12.07 4,556 728 48.05 100% 100% 100% 100%</td>
<td>Chapel 12.07 4,556 728 48.05 100% 100% 100% 100%</td>
</tr>
<tr>
<td>C++ g++ 10.62 2,040 980 41.91 100% 95% 100% 100%</td>
<td>Python 3 547.23 48,052 950 2,162.70 99% 100% 97% 100%</td>
</tr>
</tbody>
</table>
CLBG: Qualitative Code Comparisons

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

excerpt from 1210 gz Chapel entry

```chapel
proc main()
{
    printColorEquations();
    const group1 = [{i in 1...popSize1} new Chameneoes(i, {((i-1)modColor); }]
    const group2 = [{i in 1...popSize2} new Chameneoes(i, colors10[i]);}
    cобegins
        holdMeetings(group1, n);
        holdMeetings(group2, n);
    cобegin
    print(group1);
    print(group2);
    for c in group1 do delete c;
    for c in group2 do delete c;
}
```

excerpt from 2863 gz C gcc entry

```c
void get_affinity(int* is_mp, cpu_set_t* affinity1, cpu_set_t* affinity2)
{
    CPU_set_t active_cpus;
    FILE* f;
    char
    char const* pos;
    int cpu_idx;
    physical_id;
    int core_id;
    int cpu_cores;
    int alic_id;
    size_t cpu_count;
    size_t
    size_t
    size_t
    size_t
    size_t
    processor_str = "processor";
    processor_id = "physical";
    core_id = "core_id";
    cpu_cores = "cpu_cores";
    cpu_count = 0;
    for (i = 0; i != CPU_SETSIZE; i++)
        if (CPU_ISSET(i, &active_cpus))
            cpu_count++;
    if (cpu_count == 1)
        is_mp[0] = 0;
    return;
    CPU_TEST(affinity1);
```
CLBG: Qualitative Code Comparisons

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

```
proc main()
    printColorEquations();
    const group1 = [i in ...];
    new Chamenece(i, c);
    cobegin
        cobegin
            holdMeetings(group1, n);
            holdMeetings(group2, n);
        endcobegin
    endcobegin
    print(group2);

proc holdMeetings(population, numMeetings)
    const place = new MeetingPlace(numMeetings);
    coforall c in population do
        c.haveMeetings(place, population);
    delete place;
```

excerpt from 1210 gz Chapel entry

```
void set_affinity(int* is affinity, int* affinity1, int* affinity2)
    active_cpus;
    f;
    buf [048];
    pos;
    cpu_idx;
   ysical_id;
    core_id;
    cpu_cores;
    epic_id;
    cpu_count;
    i;
```

excerpt from 2863 gz C gcc entry
CLBG: Qualitative Code Comparisons

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

```
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;
}
```

```
void get_affinity(int* is_smp, cpu_set_t* affinity1, cpu_set_t* affinity2)
{
    cpu_set_t active_cpus;
    FILE* f;
    char char const* pos;
    int cpu_idx;
    int physical_id;
    int core_id;
    int cpu_cores;
    int api_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;
}
```

*excerpt from 1210 gz Chapel entry*

*excerpt from 2863 gz C gcc entry*
CLBG: Chapel Entries (May 14, 2019)

- Execution Time (normalized to fastest entry)
- Compressed Code Size (normalized to smallest entry)

- smaller
- faster

- chapel
- smallest
- fastest
- gmean-smallest
- gmean-fastest
CLBG Cross-Language Summary (May 14, 2019, zoomed)
CLBG Cross-Language Summary (May 14, 2019, zoomed)
CLBG Cross-Language Summary (May 14, 2019)
STREAM Triad: a trivial parallel computation

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

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

**In pictures:**

\[
\begin{align*}
A & \quad = \\
B & \quad + \\
C & \quad \cdot \\
\alpha & \quad \text{scaling factor}
\end{align*}
\]
STREAM Triad: a trivial parallel computation

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

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

In pictures, in parallel (shared memory / multicore):
STREAM Triad: a trivial parallel computation

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

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

In pictures, in parallel (distributed memory):

- $A$
- $B$
- $C$
- $\alpha$
STREAM Triad: a trivial parallel computation

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

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

In pictures, in parallel (distributed memory multicore):

![Diagram showing the STREAM Triad computation](image)
#include <hpcc.h>

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

int HPCC_StarStream(HPCC_Params *params) {
    int myRank, commSize;
    int rv, errCount;
    MPI_Comm comm = MPI_COMM_WORLD;

    MPI_Comm_size( comm, &commSize );
    MPI_Comm_rank( comm, &myRank );

    rv = HPCC_Stream( params, 0 == myRank);
    MPI_Reduce( &rv, &errCount, 1, MPI_INT, MPI_SUM, 0, comm );

    return errCount;
}

int HPCC_Stream(HPCC_Params *params, int doIO) {
    register int j;
    double scalar;
    VectorSize = HPCC_LocalVectorSize( params, 3, sizeof(double), 0 );

    a = HPCC_XMALLOC( double, VectorSize );
    b = HPCC_XMALLOC( double, VectorSize );
    c = HPCC_XMALLOC( double, VectorSize );

    if (!a || !b || !c) {
        if (c) HPCC_free(c);
        if (b) HPCC_free(b);
        if (a) HPCC_free(a);
        if (doIO) {
            fprintf(outFile, "Failed to allocate memory (%d).\n", VectorSize );
            fclose(outFile );
        }
        return 1;
    }

    #ifdef _OPENMP
    #pragma omp parallel for
    #endif
    for (j=0; j<vectorSize; j++) {
        b[j] = 2.0;
        c[j] = 1.0;
        scalar = 3.0;
    } 

    #ifdef _OPENMP
    #pragma omp parallel for
    #endif
    for (j=0; j<vectorSize; j++)
        a[j] = b[j]+scalar*c[j];

    HPCC_free(c);
    HPCC_free(b);
    HPCC_free(a);

    return 0;
}
```
#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif

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

int HPCC_SparkStream(HPCC_Params *params) {
    int myRank, commSize;
    int rv, errCount;
    MPI_Comm comm = MPI_COMM_WORLD;
    MPI_Comm_size( comm, &commSize );
    MPI_Comm_rank( comm, &myRank );
    rv = HPCC_Stream( params, 0 == myRank );
    MPI_Reduce( &rv, &errCount, 1, MPI_INT, MPI_SUM, 0, comm );

    return errCount;
}

int HPCC_Stream(HPCC_Params *params, int doIO) {
    register int j;
    double scalar;
    VectorSize = HPCC_LocalVectorSize( params, 3, sizeof(double), 0 );
    
a = HPCC_XMALLOC( double, VectorSize );
b = HPCC_XMALLOC( double, VectorSize );
c = HPCC_XMALLOC( double, VectorSize );

    if (!a || !b || !c) {
        if (c) HPCC_free(c);
        if (b) HPCC_free(b);
        if (a) HPCC_free(a);
        if (doIO) {
            fprintf( outFile, "Failed to allocate memory (%d).\n", VectorSize );
            fclose( outFile );
        }
        return 1;
    }
#ifdef _OPENMP
    #pragma omp parallel for
#endif
    for (j=0; j<VectorSize; j++) {
        b[j] = 2.0;
        c[j] = 1.0;
    }
    scalar = 3.0;
#ifdef _OPENMP
    #pragma omp parallel for
#endif
    for (j=0; j<VectorSize; j++)
        a[j] = b[j]+scalar*c[j];
    HPCC_free(c);
    HPCC_free(b);
    HPCC_free(a);
    return 0;
}
STREAM Triad: Chapel

```c
#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif

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

int HPCC_StarStream(HPCC_Params *params) {
  int myRank, commSize;
  int rv, errCount;

  MPI_Comm comm = MPI_COMM_WORLD;
  MPI_Comm_size( comm, &commSize );
  MPI_Comm_rank( comm, &myRank );

  rv = HPCC_Stream( params, 0 == myRank );
  MPI_Reduce( &rv, &errCount, 1, MPI_INT, MPI_SUM, 0, comm );

  return errCount;
}

int HPCC_Stream(HPCC_Params *params, int doIO) {
  register int j;
  double scalar;

  VectorSize = HPCC_LocalVectorSize( params, 3, sizeof(double), 0 );
  a = HPCC_XMALLOC( double, VectorSize );
  b = HPCC_XMALLOC( double, VectorSize );
  c = HPCC_XMALLOC( double, VectorSize );

  if (!a || !b || !c) {
    if (c) HPCC_free(c);
    if (b) HPCC_free(b);
    if (a) HPCC_free(a);
    if (doIO) {
      fprintf(outFile, "Failed to allocate memory (%d).\n", VectorSize);
      fclose(outFile);
    }
    return 1;
  }

  #ifdef _OPENMP
  #pragma omp parallel for
  #endif
  for (j=0; j<VectorSize; j++) {
    b[j] = 2.0;
    c[j] = 1.0;
  }

  scalar = 3.0;
  #ifdef _OPENMP
  #pragma omp parallel for
  #endif
  for (j=0; j<VectorSize; j++)
    a[j] = b[j] + scalar * c[j];

  HPCC_free(c);
  HPCC_free(b);
  HPCC_free(a);

  return 0;
}
```

The special sauce: How should this index set—and any arrays and computations over it—be mapped to the system?
HPCC STREAM Triad: Chapel vs. C+MPI+OpenMP
HPC Patterns: Chapel vs. Reference

LCALS: Chapel vs. Reference
Local loop kernels

HPCC RA: Chapel vs. C+MPI
Global Random Updates

HPCC STREAM Triad: Chapel vs. Reference
Embarrassing/Pleasing Parallelism

STREAM Triad

ISx: Chapel vs. Reference
Bucket-Exchange Pattern

PRK Stencil: Chapel vs. Reference
Stencil Boundary Exchanges

Local loop kernels

Nightly performance tickers online at: https://chapel-lang.org/perf-nightly.html
HPC Patterns: Chapel vs. Reference

**LCALS: Chapel vs. Reference**

** STREAM Triad **

** HPCC RA **

** HPCC STREAM Triad: Chapel vs. Reference **

** ISx: Chapel vs. Reference **

** PRK Stencil: Chapel vs. Reference **

Nightly performance tickers online at: [https://chapel-lang.org/perf-nightly.html](https://chapel-lang.org/perf-nightly.html)
Summarizing this Talk

Chapel cleanly and orthogonally supports…
  …expression of parallelism and locality
  …specifying how to map computations to the system

Chapel is powerful:
  • supports succinct, straightforward code
  • can result in performance that competes with (or beats) C+MPI+OpenMP
Chapel Central

https://chapel-lang.org

- download Chapel
- 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
Chapel Comes of Age: Making Scalable Programming Productive

Bradford L. Chamberlain, Elliot Roncahan, Ben Albers, Lydia Duncan, Michael Ferguson, Ben Harchergerg, David Jin, David Kazan, Vassilis Litsifres, Praveen Sadasiva, and Greg Tino

Chapel Inc.
Seattle, WA, USA
chapel.info@cray.com

Abstract—Chapel is a programming language whose goal is to support productive, parallel computing at scale. Chapel’s approach can be thought of as achieving the strengths of Fortran, Python, C++, and MPI in a single language. Five years ago, the DARPA High Productivity Computing Systems (HPCS) program launched Chapel’s development with the intent of using Chapel to improve Chapel’s adoption in end-users. This paper follows up on our CUG 2018 paper by summarizing the progress made by the Chapel project since that time. Specifically, Chapel’s performance was compared with a batch-hand-coded Fortran 90 program for a number of HPC benchmarks. The results showed that Chapel was competitive with Fortran 90 for the benchmarks tested. The development of Chapel continues to grow, to include PETSc, BLAS, LAPACK, MPI, ZMQ, and other key technologies. Its development has been enhanced and codified into the set of tools available at Chapel-lang.org. This paper also chronicles the experience of paying a software defined environment as diverse as autotomation and artificial intelligence.

Keywords—Parallel programming; Computer language

1. INTRODUCTION

Chapel is a programming language designed to support productive, general-purpose parallel computing at scale. Chapel’s approach can be thought of as achieving the strengths of Fortran, Python, C++, and MPI in a single language. Five years ago, the DARPA High Productivity Computing Systems (HPCS) program launched Chapel’s development with the intent of using Chapel to improve Chapel’s adoption in end-users. This paper follows up on our CUG 2018 paper by summarizing the progress made by the Chapel project since that time. Specifically, Chapel’s performance was compared with a batch-hand-coded Fortran 90 program for a number of HPC benchmarks. The results showed that Chapel was competitive with Fortran 90 for the benchmarks tested. The development of Chapel continues to grow, to include PETSc, BLAS, LAPACK, MPI, ZMQ, and other key technologies. Its development has been enhanced and codified into the set of tools available at Chapel-lang.org. This paper also chronicles the experience of paying a software defined environment as diverse as autotomation and artificial intelligence.

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 2013, at which point Chapel was a competing prototype, having successfully demonstrated several key concepts that were intended to improve Chapel’s adoption in end-users. This paper follows up on our CUG 2018 paper by summarizing the progress made by the Chapel project since that time. Specifically, Chapel’s performance was compared with a batch-hand-coded Fortran 90 program for a number of HPC benchmarks. The results showed that Chapel was competitive with Fortran 90 for the benchmarks tested. The development of Chapel continues to grow, to include PETSc, BLAS, LAPACK, MPI, ZMQ, and other key technologies. Its development has been enhanced and codified into the set of tools available at Chapel-lang.org. This paper also chronicles the experience of paying a software defined environment as diverse as autotomation and artificial intelligence.

Chapel’s implementation under HPCS demonstrated that the language could be implemented parallel while still being optimized for HPC-specific features such as the KRAKEN support available in Cray®洸鳍™ and Aciem™ systems. This allowed Chapel to take advantage of native hardware support for remote ports, gpus, and atomic memory operations. Despite these successes, at the close of HPCS, Chapel was not in all ready to support production codes in the field. This was not surprising given the language’s aggressive design and the need to find a balance between implementation cost and thoroughly tested features. The potential users of Chapel 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. Traditionally, we refer to this effort as “the two-year push.”

This paper’s contribution is to describe the results of this two-year effort, providing answers 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 1.15, which was released five years ago in April 2013.

paper and slides available at chapel-lang.org
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-