Chapel
Programmability, Parallelism, and Performance

Brad Chamberlain
Puget Sound Programming Python Meetup (PuPPy)
December 12, 2018

bradc@cray.com
chapel-lang.org
@ChapelLanguage
Cray-1: A Pioneering Supercomputer (1975)

<table>
<thead>
<tr>
<th>CPU</th>
<th>64-bit processor @ 80 MHz[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>8.39 Megabytes (up to 1,048,576 words)[1]</td>
</tr>
<tr>
<td>Storage</td>
<td>303 Megabytes (DD19 Unit)[1]</td>
</tr>
<tr>
<td>FLOPS</td>
<td>160 MFLOPS</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Model Cray XC40/Cray XC50</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Hybrid Compute Nodes</td>
<td>5,704</td>
</tr>
<tr>
<td>Number of Multicore Compute Nodes</td>
<td>1,431</td>
</tr>
<tr>
<td>Peak Floating-point Performance per Hybrid Node</td>
<td>4,761 Teraflops Intel Xeon E5-2690 v3/Nvidia Tesla P100</td>
</tr>
<tr>
<td>Peak Floating-point Performance per Multicore Node</td>
<td>1,210 Teraflops Intel Xeon E5-2695 v4</td>
</tr>
<tr>
<td>Hybrid Peak Performance</td>
<td>27.154 Petaflops</td>
</tr>
<tr>
<td>Multicore Peak Performance</td>
<td>1.731 Petaflops</td>
</tr>
<tr>
<td>Hybrid Memory Capacity per Node</td>
<td>64 GB; 16 GB CoWoS HBM2</td>
</tr>
<tr>
<td>Multicore Memory Capacity per Node</td>
<td>64 GB, 128 GB</td>
</tr>
<tr>
<td>Total System Memory</td>
<td>437.9 TB; 83.1 TB</td>
</tr>
<tr>
<td>System Interconnect</td>
<td>Cray Aries routing and communications ASIC, and Dragonfly network topology</td>
</tr>
<tr>
<td>Sonexion 3000 Storage Capacity</td>
<td>8.8 PB</td>
</tr>
<tr>
<td>Sonexion 3000 Parallel File System Theoretical Peak Performance</td>
<td>112 GB/s</td>
</tr>
<tr>
<td>Sonexion 1600 Storage Capacity</td>
<td>2.5 PB</td>
</tr>
<tr>
<td>Sonexion 1600 Parallel File System Theoretical Peak Performance</td>
<td>138 GB/s</td>
</tr>
</tbody>
</table>

https://www.cscs.ch/computers/piz-daint/
Cray: The Supercomputer Company

ASK YOUR BIGGEST QUESTIONS

SUPERCOMPUTING
Scale your goals with high-performance compute solutions

DATA STORAGE
Get faster insights with simpler storage and data management

BIG DATA ANALYTICS
Think bigger about big data with agile analytics technology

ARTIFICIAL INTELLIGENCE
Create tomorrow with compute tools for AI development

CLOUD
Extend your possibilities with cloud-based supercomputing and storage

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”

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

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

<table>
<thead>
<tr>
<th>Ada</th>
<th>C</th>
<th>Chapel</th>
<th>C#</th>
<th>C++</th>
<th>Dart</th>
</tr>
</thead>
<tbody>
<tr>
<td>Erlang</td>
<td>F#</td>
<td>Fortran</td>
<td>Go</td>
<td>Hack</td>
<td></td>
</tr>
<tr>
<td>Haskell</td>
<td>Java</td>
<td>JavaScript</td>
<td>Lisp</td>
<td>Lua</td>
<td></td>
</tr>
<tr>
<td>OCaml</td>
<td>Pascal</td>
<td>Perl</td>
<td>PHP</td>
<td>Python</td>
<td></td>
</tr>
<tr>
<td>Racket</td>
<td>Ruby</td>
<td>Rust</td>
<td>Smalltalk</td>
<td>Swift</td>
<td></td>
</tr>
<tr>
<td>TypeScript</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Which programs are faster? Trust, and verify

{ for researchers }
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>1.62</td>
<td>6,484</td>
<td>423</td>
<td>1.63</td>
<td>99% 1% 1% 2%</td>
</tr>
<tr>
<td>Chapel</td>
<td>1.62</td>
<td>6,488</td>
<td>501</td>
<td>1.63</td>
<td>99% 1% 1% 1%</td>
</tr>
<tr>
<td>Free Pascal #3</td>
<td>1.73</td>
<td>2,428</td>
<td>530</td>
<td>1.72</td>
<td>0% 2% 100% 1%</td>
</tr>
<tr>
<td>Rust #3</td>
<td>1.74</td>
<td>4,488</td>
<td>1366</td>
<td>1.74</td>
<td>1% 100% 1% 0%</td>
</tr>
<tr>
<td>Rust</td>
<td>1.74</td>
<td>4,616</td>
<td>1420</td>
<td>1.74</td>
<td>1% 100% 1% 0%</td>
</tr>
<tr>
<td>Rust #2</td>
<td>1.74</td>
<td>4,636</td>
<td>1306</td>
<td>1.74</td>
<td>1% 100% 1% 0%</td>
</tr>
<tr>
<td>C gcc</td>
<td>1.75</td>
<td>2,728</td>
<td>452</td>
<td>1.74</td>
<td>1% 2% 0% 100%</td>
</tr>
<tr>
<td>Ada 2012 GNAT #2</td>
<td>1.75</td>
<td>4,312</td>
<td>1068</td>
<td>1.75</td>
<td>1% 0% 100% 0%</td>
</tr>
<tr>
<td>Swift #2</td>
<td>1.76</td>
<td>8,492</td>
<td>601</td>
<td>1.76</td>
<td>1% 100% 1% 0%</td>
</tr>
<tr>
<td>Lisp SBCL #4</td>
<td>1.79</td>
<td>20,196</td>
<td>940</td>
<td>1.79</td>
<td>1% 2% 1% 100%</td>
</tr>
<tr>
<td>C++ g++ #4</td>
<td>1.89</td>
<td>4,284</td>
<td>513</td>
<td>1.88</td>
<td>5% 0% 1% 100%</td>
</tr>
<tr>
<td>Go #3</td>
<td>2.04</td>
<td>8,976</td>
<td>603</td>
<td>2.04</td>
<td>1% 0% 100% 0%</td>
</tr>
<tr>
<td>C++ #5</td>
<td>2.12</td>
<td>10,664</td>
<td>399</td>
<td>2.11</td>
<td>100% 0% 1% 1%</td>
</tr>
<tr>
<td>PHP #3</td>
<td>2.12</td>
<td>10,512</td>
<td>389</td>
<td>2.12</td>
<td>100% 0% 2% 0%</td>
</tr>
</tbody>
</table>

**gz == code size metric**

- strip comments and extra whitespace, then gzip
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!):

```chapel
proc main() {
    printColorEquations();
    const group1 = {i in 1..popSize1} new Chamenees(i, ((i-1)%3):Color);
    const group2 = {i in 1..popSize2} new Chamenees(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;
}
```

```c
void get_affinity(int* is_amp, 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 apic_id;
    size_t cpu_count;
    size_t
    char const* processor_str = "processor";
    char const* physical_id_str = "physical id";
    size_t core_id_str = "core id";
    size_t cpu_cores_str = "cpu cores";
    CPU_set_t active_cpus;
    CPU_getaffinity(0, e_sizeof(active_cpus), &active_cpus);
    cpu_count = 0;
    for (i = 0; i < CPU_SETSIZE; i++)
    { if (CPU_ISSET(i, &active_cpus))
        { cpu_count ++ 1; }
    }
    if (cpu_count == 0)
    { is_amp[0] = 0;
      return;
    }
    is_amp[0] = 1;
    CPU_EROE(affinity1);
}
```

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!*):

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!)*:

excerpt from 1210 gz Chapel entry

```chapel
proc main() {
    char const* core_id_str = "core id"
    size_t core_id_str_len = strlen(core_id_str);
    char const* cpu_cores_str = "cpu cores"
    size_t cpu_cores_str_len = strlen(cpu_cores_str);
    CPU_ZERO(&active_cpus);
    sched_getaffinity(0, sizeof(active_cpus), &active_cpus);
    cpu_count = 0;
    for (i = 0; i != CPU_SETSIZE; i ++)
    {
        if (CPU_ISSET(i, &active_cpus))
        {
            cpu_count += 1;
        }
    }
    if (cpu_count == 1)
    {
        is_smp[0] = 0;
        return;
    }
}
```

excerpt from 2863 gz C gcc entry

```c
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 int t equ;
    int s
    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 ++)
    {
        if (CPU_ISSET(i, &active_cpus))
        {
            cpu_count += 1;
        }
    }
    if (cpu_count == 1)
    {
        is_smp[0] = 0;
        return;
    }
    is_smp[0] = 1;
    CPU_ZERO(affinity);
```
Overview of Chapel Features
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

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

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

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

Iterators

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

```clu
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;
    }
}

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

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

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

Static Type Inference for:
- arguments
- return types
- variables

Explicit types also supported

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

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

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

```c
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;
  }
}

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

```
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

```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);
```

```
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
...
```

Tuples
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

Locales:

0 1 2 3

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

```chpl
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
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

```
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

```
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

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

```
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
Hello from task 2 of 2 running on n1032
Hello from task 1 of 2 running on n1032
Task Parallelism and Locality, by example

```chapel
coforall loc in Locales do
    on loc {
        const numTasks = here.numPUs();
        coforall tid in 1..numTasks do
            printf("Hello from task %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

```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);
    }
```

```
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
      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);
  }
```

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

doi:10.1007/978-3-319-66001-6_5

Listing:

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

```
prompt> chpl taskParallel.chpl
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

Chapel language concepts

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

Higher-level 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

```
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

Data-Parallel Forall Loops

```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

```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

```
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)
Distributed Data Parallelism, by example

```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

- **Dense**
- **Strided**
- **Sparse**
- **Associative**
- **Unstructured**
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: Chapel vs. Reference**

Serial Kernels (long)

Parallel Kernels (long)

**STREAM Triad**

**HPCC RA**

**ISx**

**PRK Stencil**

**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)
HPC Patterns: Chapel vs. Reference

LCALS: Chapel vs. Reference
- Local loop kernels

HPCC STREAM Triad: Chapel vs. Reference
- STREAM Triad

HPCC RA: Chapel vs. Reference
- Global Random Updates
- PRK Stencil

HPC STREAM Triad: Chapel vs. Reference
- Embarrassing/Pleasing Parallelism

ISx: Chapel vs. Reference
- Bucket-Exchange Pattern

PRK Stencil: Chapel vs. Reference
- Stencil Boundary Exchanges

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

- **Local loop kernels**
- **STREAM Triad**
- **Embarrassing/Pleasing Parallelism**

- **HPCC RA**
- **ISx**
- **Bucket-Exchange Pattern**

- **PRK Stencil**
- **Stencil Boundary Exchanges**

Nightly performance tickers online at: [https://chapel-lang.org/perf-nightly.html](https://chapel-lang.org/perf-nightly.html)
HPCC RA: Chapel vs. Reference

RA Performance (GUPS)

Chapel 1.19 (pre-release) vs. Reference (bucketing)

Locales (x 36 cores / locale)
/* Perform updates to main table. The scalar equivalent is:
 * for (i = 0; i < UPDATE; i++) {
 *   if (i < SendCnt) {
 *     if (GlobalOffset[i] <= NumberReceiving) {  
 *       while (have_done && NumberReceiving > 0) {
 *         if (pendingUpdates <= maxPendingUpdates) {
 *           Ran = (Ran << 1) ^ ((s64Int) Ran < 0) ? POLY : 0;
 *           HPCC_InsertUpdate(Ran, WhichPw, Buckets);
 *           pendingUpdates++;
 *         }
 *         i++;
 *       }
 *       else {  
 *         HPCC_Table[localOffset] ^= Ran;
 *         MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64, 
 *                   HPCC_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, inreq);
 *         /* receive messages */
 *         do {  
 *           if (have_done) |
 *             HPCC_Incr(HPCC_Table, tparams DIAG, irecvUpdates);  
 *           bufferBase = 0;
 *           for (j = 0; j < proc_count; j++) {
 *             imag = LocalRecvBuffer[bufferBase++];
 *             GlobalOffset = (imag & (tparams.TableSize - 1)) -
 *               tparams.GlobalStartMyProc;
 *             HPCC_Table[localOffset] = imag;
 *           }  
 *           else if (status.MPI_TAG == FINISHED_TAG) {  
 *             NumberReceiving = 0;
 *           }  
 *           else  
 *             MPI_Abort(MPI_COMM_WORLD, -1);  
 *         }  
 *         while (have_done && NumberReceiving > 0);
 *         if (pendingUpdates < maxPendingUpdates) |
 *           Ran = (Ran << 1) ^ ((s64Int) Ran < 0) ? POLY : 0;
 *           GlobalOffset = Ran & (tparams.TableSize-1);  
 *         if (GlobalOffset < tparams.Top)  
 *           WhichPw = (GlobalOffset / (tparams.MinLocalTableSize + 1) -
 *             tparams.GlobalStartMyProc;  
 *           else  
 *             WhichPw = (GlobalOffset - tparams.Remainder) / 
 *               tparams.MinLocalTableSize;  
 *         if (WhichPw == tparams.MyProc)  
 *           (LocalOffset = (Ran & (tparams.TableSize - 1)) -
 *             tparams.GlobalStartMyProc);  
 *         HPCC_Table[localOffset] ^= Ran;
 *         MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64, 
 *                   HPCC_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, inreq);
 *         /* receive messages */
 *         do {  
 *           if (have_done) |
 *             HPCC_Incr(HPCC_Table, tparams DIAG, irecvUpdates);  
 *           bufferBase = 0;
 *           for (j = 0; j < proc_count; j++) {
 *             imag = LocalRecvBuffer[bufferBase++];
 *             GlobalOffset = (imag & (tparams.TableSize - 1)) -
 *               tparams.GlobalStartMyProc;
 *             HPCC_Table[localOffset] = imag;
 *           }  
 *           else if (status.MPI_TAG == FINISHED_TAG) {  
 *             NumberReceiving = 0;
 *           }  
 *           else  
 *             MPI_Abort(MPI_COMM_WORLD, -1);  
 *         }  
 *         while (have_done && NumberReceiving > 0);
 *         if (status.MPI_TAG == UPDATE_TAG)  
 *           MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64, 
 *                      HPCC_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, inreq);
 *         else {  
 *           HPCC_Incr(HPCC_Table, tparams DIAG, irecvUpdates);  
 *         }  
 *         status.MPI_TAG = MPI_TAG_NULL;  
 *         bufferBase = 0;
 *         for (j = 0; j < proc_count; j++) {
 *             imag = LocalRecvBuffer[bufferBase++];
 *             GlobalOffset = (imag & (tparams.TableSize - 1)) -
 *               tparams.GlobalStartMyProc;
 *             HPCC_Table[localOffset] = imag;
 *         }  
 *         else if (status.MPI_TAG == FINISHED_TAG) {  
 *             NumberReceiving = 0;
 *           }  
 *           else  
 *             MPI_Abort(MPI_COMM_WORLD, -1);  
 *         }  
 *         while (have_done && NumberReceiving > 0);  
 *       }  
 *       i++;
 *     }
 *   }  
 *   else  
 *     if (MPI_Test(inreq, sh awe_done, MPI_STATUS_IGNORE);  
 *       if (have_done) |
 *         outreq = MPI_REQUEST_NULL;
 *         proc_count = tparams.NumProcs;
 *         proc_count = proc_count + proc_count;
 *         if (proc_count > tparams.NumProcs)  
 *           tparams финиш req tparams.MyProc =  
 *             MPI_REQUEST_NULL; continue;  
 *       */
 *       send done messages */
 *       for (proc_count = 0; proc_count < tparams.NumProcs; proc_count) {  
 *         if (proc_count == tparams.NumProcs)  
 *           tparams финиш req tparams.MyProc =  
 *             MPI_REQUEST_NULL; continue;  
 *       */
 *       send garbage = who cares, no one will look at it */
 *       MPI_Isend(&LocalSendBuffer, peUpdates, tparams.dtype64, (int)pe,  
 *                 UPDATE_TAG, MPI_COMM_WORLD, outreq);
 *       pendingUpdates = peUpdates;
 *     }  
 *   }  
 * }  
 * while (have_done && NumberReceiving > 0) {  
 *   if (status.MPI_TAG == UPDATE_TAG)  
 *     MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64, 
 *               HPCC_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, inreq);
 *   else {  
 *     HPCC_Incr(HPCC_Table, tparams DIAG, irecvUpdates);  
 *   }  
 *   bufferBase = 0;
 *   for (j = 0; j < proc_count; j++) {
 *       imag = LocalRecvBuffer[bufferBase++];
 *       GlobalOffset = (imag & (tparams.TableSize - 1)) -
 *         tparams.GlobalStartMyProc;
 *       HPCC_Table[localOffset] = imag;
 *   }  
 *   else if (status.MPI_TAG == FINISHED_TAG) {  
 *     NumberReceiving = 0;
 *   }  
 *   else  
 *     MPI_Abort(MPI_COMM_WORLD, -1);  
 * }  
 * while (NumberReceiving > 0) {  
 *   if (status.MPI_TAG == UPDATE_TAG)  
 *     MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64, 
 *               HPCC_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, inreq);
 *   else {  
 *     HPCC_Incr(HPCC_Table, tparams DIAG, irecvUpdates);  
 *   }  
 *   bufferBase = 0;
 *   for (j = 0; j < proc_count; j++) {
 *       imag = LocalRecvBuffer[bufferBase++];
 *       GlobalOffset = (imag & (tparams.TableSize - 1)) -
 *         tparams.GlobalStartMyProc;
 *       HPCC_Table[localOffset] = imag;
 *   }  
 *   else if (status.MPI_TAG == FINISHED_TAG) {  
 *     NumberReceiving = 0;
 *   }  
 *   else  
 *     MPI_Abort(MPI_COMM_WORLD, -1);  
 * }  
 * MPI_Waitall(tparams.NumProcs, tparams финиш req, tparams финиш статусы);
/* 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;
 * }
 */

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

RA Performance (GUPS)

- Chapel 1.19 (pre-release)
- Reference (bucketing)

Locales (x 36 cores / locale)

GUPS

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

Subitle: 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](https://chapel-lang.org/docs/technotes/libraries.html#using-your-library-in-python)
The Chapel Team at Cray (May 2018)

~12 full-time employees + ~2 summer interns
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

(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
Chapel Comes of Age: Making Scalable Programming Productive

Bradford L. Chamberlain, Elliot Romanuk, Ben Abouzari, Lydia Duncan, Michael Ferguson, Ben Harmsheger, David Jem, David Keaton, Vasily Litvinov, Praveen Sabathia, and Greg Tino
Cray 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 combining the strengths of Fortran, Python, C/C++, and MPI in a single language. Five years ago, the DARPA High Productivity Computing Systems (HPCS) program that launched Chapel came to an end, with the objective of improving Chapel’s appeal to end-users. This paper follows up on the 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 MPI, Chapel developers can now use Python to specify algorithms, Chapel grows to include FTVN, BLAS/LAPACK, MPS, ZMQ, and other key technologies. Its documentation has been enhanced and restructured, and the set of tools available to Chapel users has grown. This paper also chronicles the experiences of several research communities as diverse as neurobiologists, biologists, and artificial intelligence.

Keywords: Parallel programming; Computer language

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 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 is parallel, permitting it to run on a wide variety of platforms and hardware. We have contributed to the open-source community by releasing our code under the Apache 2.0 license and are hosted at GitHub.

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 2013, at which point Chapel was a compelling prototype, having successfully demonstrated several novel features, such as Python-like specification of algorithms in Chapel and the introduction of new parallel programming constructs (such as `struct` and `field`). Chapel 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 and task parallel abstractions like parallel loops and arrays in terms of lower-level Chapel features such as classes, instances, and tasks.

Under HPCS, Chapel also significantly supported the integration of parallelism using distinct language features from those used to control locality and affinity—data is Chapel programmers specify which computations should run in parallel while specifying where those computations should be run. This permits Chapel programs to support multithreaded, 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 pairs, gpus, and atomic memory operations.

Despite these successes, at the close of HPCS, Chapel was not in an estate ready to support production codes in the field. This was not surprising given the language’s aggressive design and implementation. Chapel users requested an API that would allow them to leverage the familiar functions, data types, and potential users were sufficiently confident that, in early 2015, Cray embarked on a follow-up effort to improve Chapel and move it towards being a production-ready language. Consequently, 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 makers 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 1.5, which was released five years ago in April 2013.

© 2018 Cray Inc.
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

cray.com
@cray_inc
linkedin.com/company/cray-inc/-