Tales from the Trenches: Whipping Chapel Performance into Shape

Elliot Ronaghan, Chapel Team, Cray Inc.

CHIUW 2018
May 25 2018
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.
Chapel Performance History

- Poor performance is Chapel’s most common criticism
  - For most of Chapel, this has been an accurate concern/criticism

- Performance was significantly behind MPI
  - And other established HPC technologies (OpenMP, SHMEM, etc.)

- Performance dramatically improved in recent years
  - Many core idioms/benchmarks now on par with MPI+X/SHMEM
A Brief History of Chapel

Chapel’s Infancy: DARPA HPCS (2003–2012)
- Research focus: ~6-7 FTEs
  - distinguish locality from parallelism
  - seamlessly mix data- and task-parallelism
  - support user-defined distributed arrays, parallel iterators

Chapel’s Adolescence: “the five-year push” (2013–2018)
- Development focus: ~13-14 FTEs
  - performance and scalability
  - ecosystem: documentation, libraries, tools, …
  - base language fixes: OOP features, error-handling, strings, …
Plan for this talk

- Highlight optimizations that closed performance gap
  - Array Optimizations
  - Runtime Improvements
  - Communication Reductions

- Showcase benchmark results for key HPC idioms
  - Comparing Then (1.7) vs. Now (1.17) vs Reference

- Interrupt if you want details on specific optimizations
HPC Patterns

**LCALS: Chapel 1.17 vs. Reference**

- Normalized performance vs. locales (x 28 cores)
- Chapel 1.17 outperforms the reference

**STREAM Triad: Chapel 1.17 vs. Reference**

- STREAM performance vs. locales (x 36 cores / locale)
- Chapel 1.17 outperforms the reference

**ISx: Chapel Now vs. Reference**

- ISx time vs. locales (x 36 cores / locale)
- Chapel Now outperforms the reference

**PRK Stencil: Chapel Now vs. Reference**

- PRK stencil performance vs. locales (x 36 cores / locale)
- Chapel Now outperforms the reference

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

**LCALS: Chapel 1.17 vs. Reference**
Local loop kernels

**STREAM Triad: Chapel 1.17 vs. Reference**
Embarrassing/Pleasing Parallelism

**ISx: Chapel Now vs. Reference**
Bucket-Exchange Pattern

**PRK Stencil: Chapel Now vs. Reference**
Stencil Boundary Exchanges

**HPCC RA: Chapel 1.17 vs. Reference**
Global Random Updates

---

Nightly performance tickers online at: [https://chapel-lang.org/perf-nightly.html](https://chapel-lang.org/perf-nightly.html)
Array Optimizations
Array Background

- **Arrays are a fundamental building block in HPC**
  - And a core language feature in Chapel with rich functionality
    - support for arbitrary indexing (vs. fixed 0-based or 1-based)
    - true multi-dimensional arrays
    - built-in parallel iteration
    - support for slices, rank-change, and much, much more

- **Naïve implementations of rich arrays performed poorly**
  - Over 100 times slower for LCALS benchmark
    - LCALS is a collection of simple and local loop kernels
Chapel code for serial pressure_calc kernel:

```chapel
for i in 0..#len do
    bvc[i] = cls * (compression[i] + 1.0);
```
Chapel code:

```chapel
for i in 0..#len do
    bvc[i] = cls * (compression[i] + 1.0);
```

Code generated by 1.7:

```c
end3 = T22;
call_tmp94 = (ret54 != T22);
T23 = call_tmp94;
while (T23) {
    // get (bvc + i)
    ret58 = bvc;
    ret59 = (ret58)->origin;
    ret_11 = &((ret58)->blk);
    ret_x111 = *((ret_11) + 0);
call_tmp95 = (i3 * ret_x111);
call_tmp96 = (ret59 + call_tmp95);
    ret60 = (ret58)->factoredOffs;
call_tmp97 = (call_tmp96 - ret60);
call_tmp98 = (ret58)->data;
call_tmp99 = (call_tmp98 + call_tmp97);
    // get (compression + i)
    ret61 = compression;
    ret62 = (ret61)->origin;
    ret_12 = &((ret61)->blk);
    ret_x112 = *((ret_12) + 0);
call_tmp100 = (i3 * ret_x112);
call_tmp101 = (ret62 + call_tmp100);
    ret63 = (ret61)->factoredOffs;
call_tmp102 = (call_tmp101 - ret63);
call_tmp103 = (ret61)->data;
call_tmp104 = (call_tmp103 + call_tmp102);
    // compute cls * (* (compression + i) + 1.0)
    ret64 = *(call_tmp104);
    call_tmp105 = (ret64 + 1.0);
call_tmp106 = (cls * call_tmp105);
    // store computation
    *(call_tmp99) = call_tmp106;
    // advance index/induction variable
    call_tmp107 = (i3 + 1);
i3 = call_tmp107;
call_tmp108 = (call_tmp107 != end3);
T23 = call_tmp108;
}
```
LCALS Serial Kernel: 1.7 Generated Code

- Chapel code:
  ```chapel
  for i in 0..#len do
    bvc[i] = cls * (compression[i] + 1.0);
  ```

- Code generated by 1.7: While-loop with != operator

```
end3 = T22;
call_tmp94 = (ret54 != T22);
T23 = call_tmp94;
while (T23) {
    // get (bvc + i)
    ret58 = bvc;
    ret59 = (ret58)->origin;
    ret_11 = &((ret58)->blk);
    ret_x111 = *((ret_11) + 0);
    call_tmp95 = (i3 * ret_x111);
    call_tmp96 = (ret59 + call_tmp95);
    ret60 = (ret58)->factoredOffs;
    call_tmp97 = (call_tmp96 - ret60);
    call_tmp98 = (ret58)->data;
    call_tmp99 = (call_tmp98 + call_tmp97);

    // get (compression + i)
    ret61 = compression;
    ret62 = (ret61)->origin;
    ret_12 = &((ret61)->blk);
    ret_x112 = *((ret_12) + 0);
    call_tmp100 = (i3 * ret_x112);
    call_tmp101 = (ret62 + call_tmp100);
    ret63 = (ret61)->factoredOffs;
    call_tmp102 = (call_tmp101 - ret63);
    call_tmp103 = (ret61)->data;
    call_tmp104 = (call_tmp103 + call_tmp102);

    // compute cls * ((compression + i) + 1.0)
    ret64 = *(call_tmp104);
    call_tmp105 = (ret64 + 1.0);
    call_tmp106 = (cls * call_tmp105);

    // store computation
    *(call_tmp99) = call_tmp106;

    // advance index/induction variable
    call_tmp107 = (i3 + 1);
    i3 = call_tmp107;
    call_tmp108 = (call_tmp107 != end3);
    T23 = call_tmp108;
}
```
LCALS Serial Kernel: 1.7 Generated Code

- **Chapel code:**
  ```chapel
  for i in 0..#len do
    bvc[i] = cls * (compression[i] + 1.0);
  ```

- **Code generated by 1.7: Multiply by ‘blk’ offset**

```c
end3 = T22;
call_tmp94 = (ret54 != T22);
T23 = call_tmp94;
while (T23) {
    // get (bvc + i)
    ret58 = bvc;
    ret59 = (ret58)->origin;
    ret_11 =(&(ret58)->blk);
    ret_x111 =(*(ret_11) + 0);
call_tmp95 = (i3 * ret_x111);
call_tmp96 = (ret59 + call_tmp95);
ret60 = (ret58)->factoredOffs;
call_tmp97 = (call_tmp96 - ret60);
call_tmp98 = (ret58)->data;
call_tmp99 = (call_tmp98 + call_tmp97);
    // get (compression + i)
    ret61 = compression;
    ret62 = (ret61)->origin;
    ret_12 = &((ret61)->blk);
    ret_x112 = *(ret_12 + 0);
call_tmp100 = (i3 * ret_x112);
call_tmp101 = (ret62 + call_tmp100);
ret63 = (ret61)->factoredOffs;
call_tmp102 = (call_tmp101 - ret63);
call_tmp103 = (ret61)->data;
call_tmp104 = (call_tmp103 + call_tmp102);
    // compute cls * ((compression + i) + 1.0)
    ret64 = *(call_tmp104);
call_tmp105 = (ret64 + 1.0);
call_tmp106 = (cls * call_tmp105);
    // store computation
    *(call_tmp99) = call_tmp106;
    // advance index/induction variable
call_tmp107 = (i3 + 1);
i3 = call_tmp107;
call_tmp108 = (call_tmp107 != end3);
T23 = call_tmp108;
}
```
LCALS Serial Kernel: 1.7 Generated Code

● Chapel code:

```chapel
for i in 0..#len do
    bvc[i] = cls * (compression[i] + 1.0);
```

● Code generated by 1.7: 0-shift indexing

```c
end3 = T22;
call_tmp94 = (ret54 != T22);
T23 = call_tmp94;
while (T23) {
    // get (bvc + i)
    ret58 = bvc;
    ret59 = (ret58)->origin;
    ret_11 = &((ret58)->blk);
    ret_x111 = *(*ret_11 + 0);
call_tmp95 = (i3 * ret_x111);
call_tmp96 = (ret59 + call_tmp95);
    ret60 = (ret58)->factoredOffs;
call_tmp97 = (call_tmp96 - ret60);
call_tmp98 = (ret58)->data;
call_tmp99 = (call_tmp98 + call_tmp97);

    // get (compression + i)
    ret61 = compression;
    ret62 = (ret61)->origin;
    ret_12 = &((ret61)->blk);
    ret_x112 = *(*ret_12 + 0);
call_tmp100 = (i3 * ret_x112);
call_tmp101 = (ret62 + call_tmp100);
    ret63 = (ret61)->factoredOffs;
call_tmp102 = (call_tmp101 - ret63);
call_tmp103 = (ret61)->data;
call_tmp104 = (call_tmp103 + call_tmp102);

    // compute cls * (compression + i) + 1.0
    ret64 = *(call_tmp104);
call_tmp105 = (ret64 + 1.0);
call_tmp106 = (cls * call_tmp105);

    // store computation
    *(call_tmp99) = call_tmp106;

    // advance index/induction variable
    call_tmp107 = (i3 + 1);
    i3 = call_tmp107;
call_tmp108 = (call_tmp107 != end3);
T23 = call_tmp108;
}
```
● Chapel code:

```chapel
for i in 0..#len do
    bvc[i] = cls * (compression[i] + 1.0);
```

● Code generated by 1.7: Extra dereferences

```c
end3 = T22;
call_tmp94 = (ret54 != T22);
T23 = call_tmp94;
while (T23) {
    // get (bvc + i)
    ret58 = bvc;
    ret59 = (ret58)->origin;
    ret_11 = &((ret58)->blk);
    ret_x111 = *((ret_11) + 0);
    call_tmp95 = (i3 * ret_x111);
    call_tmp96 = (ret59 + call_tmp95);
    ret60 = (ret58)->factoredOffs;
    call_tmp97 = (call_tmp96 - ret60);
    ret61 = compression;
    ret62 = (ret61)->origin;
    ret_12 = &((ret61)->blk);
    ret_x112 = *((ret_12) + 0);
    call_tmp100 = (i3 * ret_x112);
    call_tmp101 = (ret62 + call_tmp100);
    ret63 = (ret61)->factoredOffs;
    call_tmp102 = (call_tmp101 - ret63);
    call_tmp103 = (ret61)->data;
    call_tmp104 = (call_tmp103 + call_tmp102);
}
```

```
// get (compression + i)
ret64 = *(call_tmp104);
call_tmp105 = (ret64 + 1.0);
call_tmp106 = (cls * call_tmp105);

// store computation
*(call_tmp99) = call_tmp106;

// advance index/induction variable
call_tmp107 = (i3 + 1);
i3 = call_tmp107;
call_tmp108 = (call_tmp107 != end3);
T23 = call_tmp108;
```
Chapel code:

```chapel
for i in 0..#len do
    bvc[i] = cls * (compression[i] + 1.0);
```

Code generated by 1.7:

```c
end3 = T22;
call tmp94 = (ret54 != T22);
T23 = call tmp94;
while (T23) {
    // get (bvc + i)
    ret58 = bvc;
    ret59 = (ret58)->origin;
    ret_11 = &((ret58)->blk);
    ret_x111 = *((ret_11) + 0);
    call_tmp95 = (i3 * ret_x111);
    call_tmp96 = (ret59 + call_tmp95);
    ret60 = (ret58)->factoredOffs;
    call_tmp97 = (call_tmp96 - ret60);
    call_tmp98 = (ret58)->data;
    call_tmp99 = (call_tmp98 + call_tmp97);
    // get (compression + i)
    ret61 = compression;
    ret62 = (ret61)->origin;
    ret_12 = &((ret61)->blk);
    ret_x112 = *((ret_12) + 0);
    call_tmp100 = (i3 * ret_x112);
    call_tmp101 = (ret62 + call_tmp100);
    ret63 = (ret61)->factoredOffs;
    call_tmp102 = (call_tmp101 - ret63);
    call_tmp103 = (ret61)->data;
    call_tmp104 = (call_tmp103 + call_tmp102);
    // compute cls * (* (compression + i) + 1.0)
    ret64 = *(call_tmp104);
    call_tmp105 = (ret64 + 1.0);
    call_tmp106 = (cls * call_tmp105);
    // store computation
    *(call_tmp99) = call_tmp106;
    // advance index/induction variable
    call_tmp107 = (i3 + 1);
    i3 = call_tmp107;
    call_tmp108 = (call_tmp107 != end3);
    T23 = call_tmp108;
}
```
Array Optimizations

● Implemented a "shifted data" optimization
  ● Eliminates overhead of arbitrary indexing

● Implemented loop-invariant code motion
  ● Eliminates array meta-data references in loop bodies

● Eliminated a multiply in indexing operations
  ● (Required for outermost dimensions of multi-dimensional arrays)

● Dramatically improved generated code for loops
Chapel code:

```chapel
for i in 0..#len do
    bvc[i] = cls * (compression[i] + 1.0);
```

Code generated by 1.7:

```chapel
call tmp94 = (ret54 != T22);
T23 = call_tmp94;
while (T23) {
    // get (bvc + i)
    ret58 = bvc;
    ret59 = (ret58)->origin;
    ret_11 = &((ret58)->blk);
    ret_x111 = *((ret_11) + 0);
    call_tmp95 = (i3 * ret_x111);
    call_tmp96 = (ret59 + call_tmp95);
    ret60 = (ret58)->factoredOffs;
    call_tmp97 = (call_tmp96 - ret60);
    call_tmp98 = (ret58)->data;
    call_tmp99 = (call_tmp98 + call_tmp97);

    // get (compression + i)
    ret61 = compression;
    ret62 = (ret61)->origin;
    ret_12 = &((ret61)->blk);
    ret_x112 = *((ret_12) + 0);
    call_tmp100 = (i3 * ret_x112);
    call_tmp101 = (ret62 + call_tmp100);
    ret63 = (ret61)->factoredOffs;
    call_tmp102 = (call_tmp101 - ret63);
    call_tmp103 = (ret61)->data;
    call_tmp104 = (call_tmp103 + call_tmp102);

    // compute cls * *(compression + i) + 1.0
    ret64 = *(call_tmp104);
    call_tmp105 = (ret64 + 1.0);
    call_tmp106 = (cls * call_tmp105);

    // store computation
    *(call_tmp99) = call_tmp106;

    // advance index/induction variable
    call_tmp107 = (i3 + 1);
    i3 = call_tmp107;
    call_tmp108 = (call_tmp107 != end3);
    T23 = call_tmp108;
}
```
LCALS Serial Kernel: 1.17 Generated Code

- Chapel code:
  ```chapel
  for i in 0..#len do
    bvc[i] = cls * (compression[i] + 1.0);
  ```

- Code generated by 1.17:
  ```c
  bvc_p = (bvc_inst)->shiftedData; // Shifted pointer (hoisted)
  compression_p = (compression_inst)->shiftedData;

  for (i = 0; i <= len-1; i += 1) {
    bvc_p_i = (bvc_p + i); // Direct index offset (no mult)
    compression_p_i = (compression_p + i);
    *(bvc_p_i) = cls * ((*compression_p_i) + 1.0);
  }
  ```
LCAL Serial Kernel: Chapel Then vs. Now

**LCALS Serial Time (seconds)**

Chapel 1.7
Chapel 1.17

**Time (sec)**
0 20 40 60 80 100 120 140 160 180

**Locale (x 28 cores)**

Chapel 1.7 is 1.17 faster than Chapel 1.7.
LCALS Serial Kernels: Chapel Now vs. Ref

Normalized Time

Chapel 1.17 vs. Reference

faster

1 Locale (x 28 cores)
Runtime Optimizations
Runtime Background

- Chapel’s runtime responsible for low-level operations:
  - Memory allocation
  - Task spawning
  - Topology discovery
  - Communication

- Default runtime in 1.7 was functional/portable, but slow
  - Some faster layers available, but were highly experimental
Chapel Runtime: Then

- Memory allocations satisfied by the system allocator
  - Most system allocations have poor parallel allocation performance

- Chapel tasks were mapped directly to system pthreads
  - Incurs high task spawning and task switching times
  - No regard for NUMA affinity between consecutive parallel loops

- Communication done with GASNet over MPI substrate
  - Did not utilize underlying network capabilities (RDMA, AMO, etc.)
Chapel Runtime: Now

- Memory allocations satisfied by concurrent allocator
  - Parallel and highly scalable allocator built on top of jemalloc

- Tasks are mapped to lightweight user-level qthreads
  - Extremely fast task creation and spawning in user-space
  - Affinity awareness via hwloc (consecutive foralls run same tasks)

- Communication on Crays mapped to uGNI
  - Optimized RDMA operations (gets, puts, AMs, network atomics)
  - Scalability performance with limited software overhead
HPCC STREAM Triad: Chapel Then

STREAM Performance (GB/s)

Locales (x 28 cores / locale)

Chapel 1.7

better
HPCC STREAM Triad: Chapel Then vs. Now

STREAM Performance (GB/s)

Locales (x 28 cores / locale)

GB/s

0 500 1000 1500 2000 2500 3000

1 2 4 8 16 32

Chapel 1.17
Chapel 1.7

better
HPCC STREAM Triad: Chapel Now vs. Ref

STREAM Performance (GB/s)

Locales (x 36 cores / locale)

GB/s

Reference
Chapel 1.17

caller

Copyright 2018 Cray Inc.
HPCC RA: Chapel Then vs. Now

RA Performance (GUPS)

Locales (x 28 cores / locale)

GUPS

Chapel 1.17
Chapel 1.7

better
HPCC RA: Chapel Now vs. Ref

RA Performance (GUPS)

- Reference (bucketing)
- Reference (no bucketing)
- Chapel 1.17

Locales (x 36 cores / locale)

GUPS

better
Communication Reductions
Communication Background

- Chapel compiler is responsible for communication
  - Must insert runtime calls for “wide” (possibly remote) data
  - Call includes conditional for short-circuit or actual network comm
    - conditional branch and function call thwart backend optimizations
Chapel Communication: Then

- Everything was initially widened
  - And then attempted to narrow some calls

- Array assignments were performed element at a time
  - With meta-data access, resulted in ~5*#elements comm events

- In general, little done to optimization communication
Chapel Communication: **Now**

- Only widen what we can’t prove is local
  - Dramatic decrease in locality checks required at runtime

- **Array assignments are performed as 1 operation**
  - With meta-data hoisted, results in ~3 comm events
    - 2 small comms for meta-data, one large comm for all elements

- **Many other communication optimizations implemented**
  - Improved remote-value-forwarding, new distributions, LICM, fast-on optimization, and much more
PRK Stencil: Chapel Then

PRK Stencil Performance (Gflop/s)

Locales (x 28 cores / locale)

Gflop/s

Chapel 1.7

better
PRK Stencil: Chapel Then vs. Now

PRK Stencil Performance (Gflop/s)

Locales (x 28 cores / locale)

Gflop/s

Chapel 1.17
Chapel 1.7

better
PRK Stencil: Chapel Now vs. Ref

PRK Stencil Performance (Gflop/s)

Locales (x 36 cores / locale)

Gflop/s

Reference
Chapel 1.17
ISx: Chapel Then vs. Now

![Graph showing ISx Time (seconds) vs. Locales (x 28 cores / locale)]

- Chapel 1.7
- Chapel 1.17

Time (sec)
- ISx Time (seconds)
- Faster
ISx: Chapel Now vs. Ref

![Graph showing ISx Time (seconds) vs. Locales (x 36 cores / locale)]

- **Chapel 1.17**
- **Reference**

**faster**
Performance Summary
Performance Summary

Array Optimizations:
- shifted data optimization (eliminates arbitrary indexing overhead)
- loop-invariant code motion (eliminates meta-data overhead)
- eliminated multiply in indexing for 1D (and innermost dim of 2D+) arrays

Runtime Improvements:
- scalable parallel memory allocator
- tasks mapped to affinity aware user-level threads
- native/optimized comm with RDMA and limited software overhead

Optimized Communication:
- compiler locality analysis improvements
- bulk array assignments
- remote-value-forwarding, new distributions, fast-ons, …
Performance Summary

● Possible to write Chapel code that competes with MPI
  ● And other established HPC technologies (OpenMP, SHMEM, etc.)

● Still possible to fall off a performance cliff
  ● In some cases due to lack of user-education
  ● In other cases, we still have optimizations to implement

● We believe Chapel has demonstrated it can perform
  ● Let us know if you see otherwise
Performance Summary

**LCALS: Chapel 1.17 vs. Reference**

**STREAM Triad**

**HPCC RA**

**PRK Stencil**

**ISx**

Nightly performance tickers online at:
https://chapel-lang.org/perf-nightly.html
Crossing the Stream of Adoption

Research Prototype

Adopted in Production

Next DOE app

Next weather / climate model

What are the next stepping stones?

Where can Chapel help your workflow’s productivity?

MiniMD

ISx

CoMD

PRK Stencil

CLBG

RA

LULESH

Stream

LCALS

Codes from startups

Time-to-science academic codes

[your production app here]
Next Steps
Next Steps

● **Improve ISx performance**
  ● Achieve performance parity with reference version
    ● (Next slide contains results from prototype branch)

● **Perform scalability testing at higher locale counts**
  ● Have done preliminary testing on 1,024 locales on NERSC Edison
    ● PRK Stencil ~10% off from reference at 1,024 nodes
    ● Plan to address with more scalable tree-based remote task-spawn

● **Focus optimization efforts on user applications**
ISx: Chapel Now vs. Train Ride Branch vs. Ref

- **Chapel 1.17**
- **Chapel Branch**
- **Reference**

**ISx Time (seconds)**

- **Time (sec)**: 4, 6, 8, 10, 12, 14, 16, 18, 20, 22
- **Locales (x 36 cores / locale)**: 16, 32, 64, 128, 256

The Chapel Branch is faster than the reference.
Questions?
Performance Summary

LCALS: Chapel 1.17 vs. Reference

STREAM Triad

HPCC RA: Chapel 1.17 vs. Reference

PRK Stencil

ISx: Chapel Now vs. Reference

PRK Stencil: Chapel Now vs. Reference

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