Chapel

Sam White

CS 598-APK

10-12-18

Overview

- Chapter written by Bradford Chamberlain
  - Excerpted from “Programming Models for Parallel Computation”, edited by Pavan Balaji

- Background and motivation
- Chapel language features
- Performance
- Conclusions

Background

- DARPA HPCS Program (2002 - 2012)
  - HPCS = High Productivity Computing Systems
  - 3 vendors funded to develop new languages for high productivity parallel programming
    - IBM: X10
    - Sun/Oracle: Fortress
    - Cray: Chapel

Motivation

- Goals of the language:
  - General Parallelism: nested data and task parallelism
  - Multithreaded execution: users write tasks, not processes
  - Global-view programming: global data structures, with ability to escape them
  - Multiresolution design: allow high and low level control

Motivation

- Goals of the language:
  - Control over locality: PGAS memory model
  - Data-centric synchronization
  - Clearly define the roles of the user vs the compiler
  - Make HPC easier to adopt
  - Start from scratch

Chapel Language

- Sequential/Base language features:
  - Type inference
  - Generic programming
  - Object-oriented programming
  - Variables can be compile-time (**param**) or run-time (**const**) constants
  - Variables can be set at command-line (**config**)

Chapel: Fibonacci

```
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
…
```

Chapel Language

- Rich support for arrays, domains, tuples, and iterators:
  - Arrays can be multidimensional with user-defined memory layouts
    - **domain**: an index set, can be dense, sparse, associative, or unstructured
  - Supports promotion of scalar expressions to whole array operations

Arrays & Domains

```
const HistSpace: domain(1) = {-3..3},
      MatSpace = {0..#n, 0..#n},
      Rows = {1..n},
      Cols: [Rows] domain(1) = [i in Rows] {1..i};

var Hist: [HistSpace] int,
    Mat: [MatSpace] complex,
    Tri: [i in Rows] [Cols[i]] real;
```

```
forall i in HistSpace do
  Hist[i] = 0;
```

Chapel Language

- Unstructured task parallelism:
  - **begin { … }**: defines an anonymous task
  - **sync { … }**: waits on completion of *all* tasks in scope
  - **sync** variables for finer-grain (full/empty bit) synchronization
    - Can be optimized for single-assignment
  - **atomic** variables for lock-free programming

Chapel Language

- Structured task parallelism:
    - **cobegin { … }**: creates tasks for each statement
        - Implicit synchronization (join) between original task and its children
    - **coforall i in 1..n do**: for-loop with each iteration a separate task, with implicit join at end

Unstructured Task Parallelism

```
cobegin {
  producer();
  consumer();
}

// 'sync' types store full/empty state along with value
var buff$: [0..#buffersize] sync real;

proc producer() {
  var i = 0;
  for … {
    i = (i+1) % buffersize;
    buff$[i] = …;    // reads block until empty, leave full
} }

proc consumer() {
  var i = 0;
  while … {
    i= (i+1) % buffersize;
    …buff$[i]…;    // writes block until full, leave empty
} }
```

Structured Task Parallelism

```
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 -o taskParallel
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
```

Chapel Language

- Data parallelism:
  - **forall i in 1..n do**: a for-loop with an arbitrary number of tasks
    - Mapping of iterations to tasks can be dynamic or user-defined via domains
  - **reduce** and **scan** primitives, can be user-defined

Data Parallelism

```
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);
```

Chapel Language

- PGAS provides global namespace, but difference in local vs remote memory access is critical to performance
- **locales**
  - A type used for reasoning about locality and affinity
    - Can be a node, socket, core, PU, etc.
  - Number of locales specified on command line
  - **on** clause maps a task to a specific locale
    - Can be combined with begin, coforall, etc.

Chapel Language

- Locality control cont’d
  - **dmapped**: domain maps allow mapping data structures across locales
    - Global-view, distributed data structures
    - Enables easily porting applications from shared to distributed memory systems
    - Block, Cylic, BlockCyclic, and user-defined types

Distributed Data Parallelism

```
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);
```

Chapel Language

Local parallel:

```
coforall i in 1..msgs do
  writeln("Hello from task ", i);
```

Distributed serial:

```
writeln("Hello from locale 0!");
on Locales[1] do writeln("Hello from locale 1!");
on Locales[2] do writeln("Hello from locale 2!");
```

Distributed parallel:

```
coforall i in 1..msgs do
  on Locales[i%numLocales] do
    writeln("Hello from task ", i,
            " running on locale ", here.id);
```

Chapel Language

- Comparison to other parallel programming models:
  - Language rather than library
  - Global-view of data
  - Communication is implicit, but can be reasoned about
  - Parallelism and locality are orthogonal
  - How does performance compare?

Chapel Performance

- Not much detail on compiler and runtime optimizations in this overview chapter
  - Chapel’s abstractions enable various optimizations
    - Limited aliasing: **forall** or array programming -> vectorization
    - Prefetching of remote data
    - Aggregation of small messages
  - But can require runtime bounds checking for locality

Chapel Performance

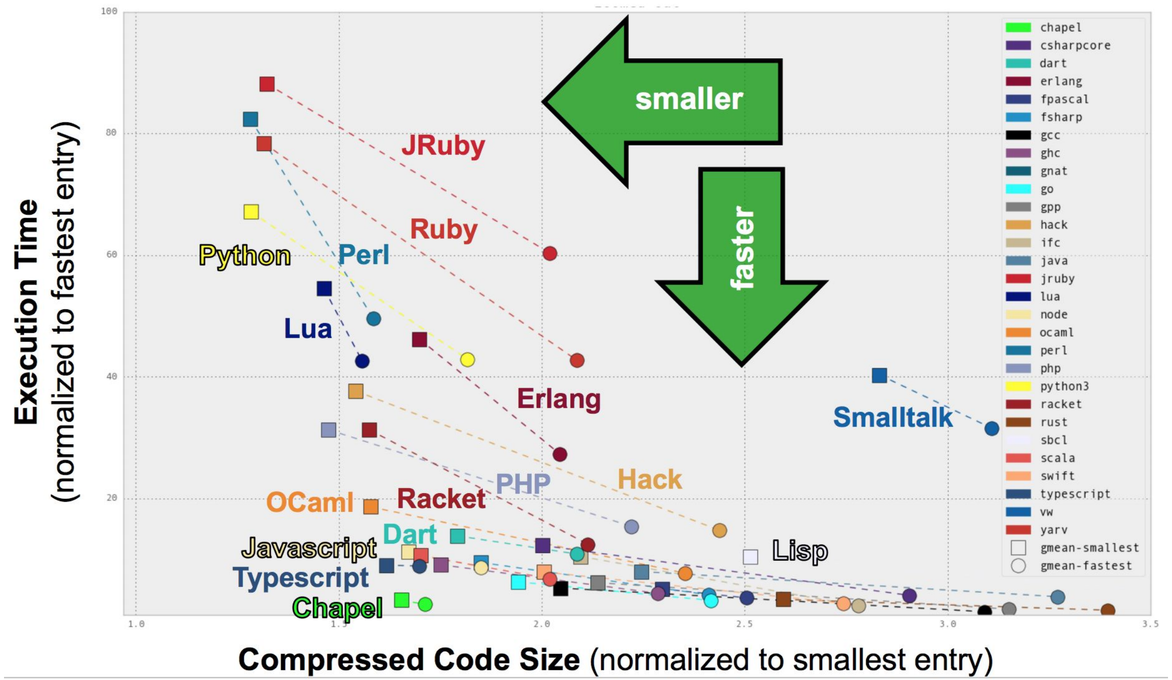
- Chapel compilation
  - Default compiler generates C code, then passed to a native compiler
    - Generates aligned memory allocations
    - Uses restrict and alignment hints
    - Pragmas for vectorization
  - Development of an LLVM Chapel backend underway

Chapel Performance

- No performance results here, but provided elsewhere in various papers and presentations
  - HPC benchmarks: CLBG, HPCC, Intel PRK, and various DOE mini-apps
  - Performance optimization is work in progress, with most progress in the past ~5 years

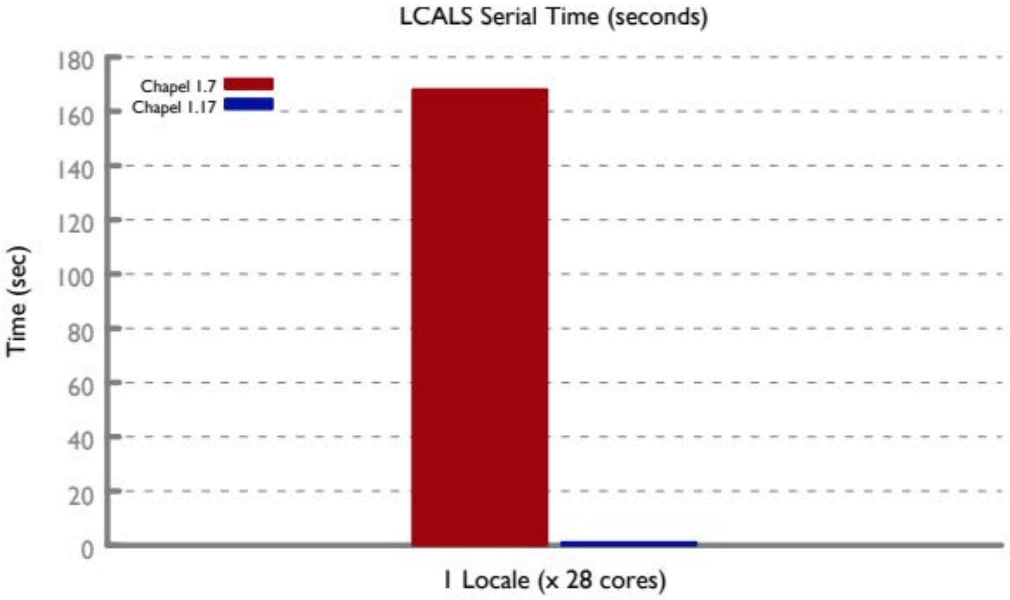
Chapel Performance

- Cross-Language Benchmark Games (10/2017)

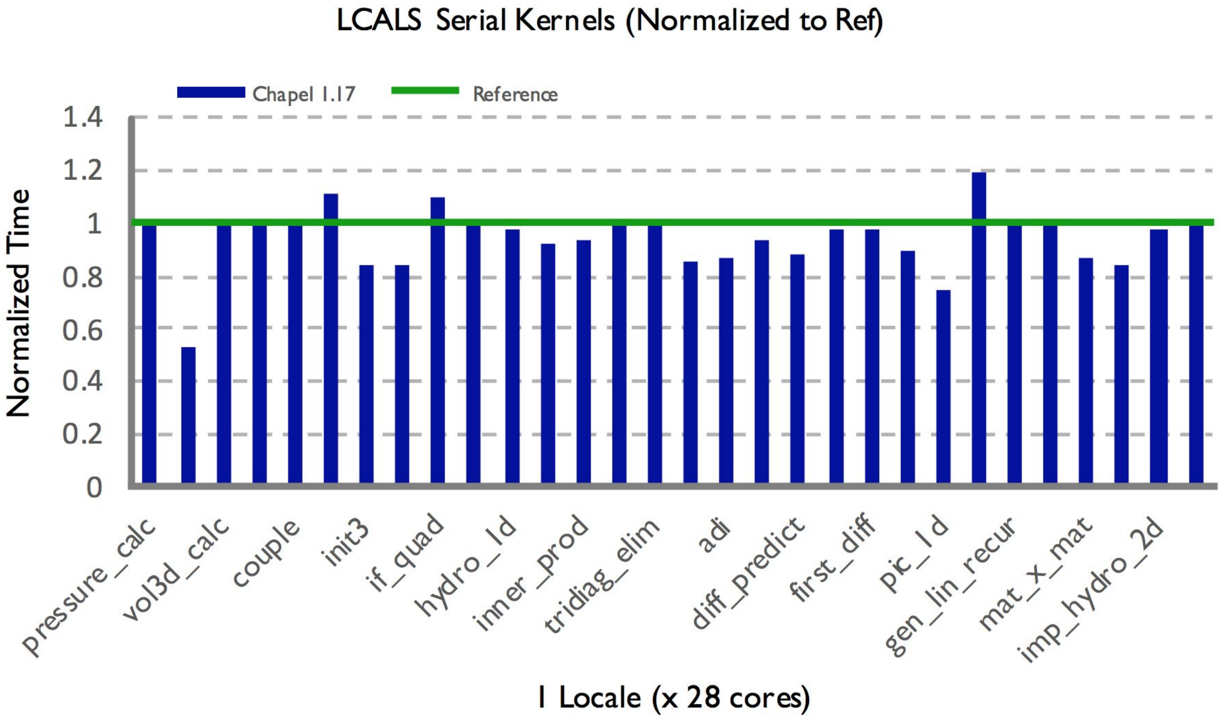


Chapel Performance

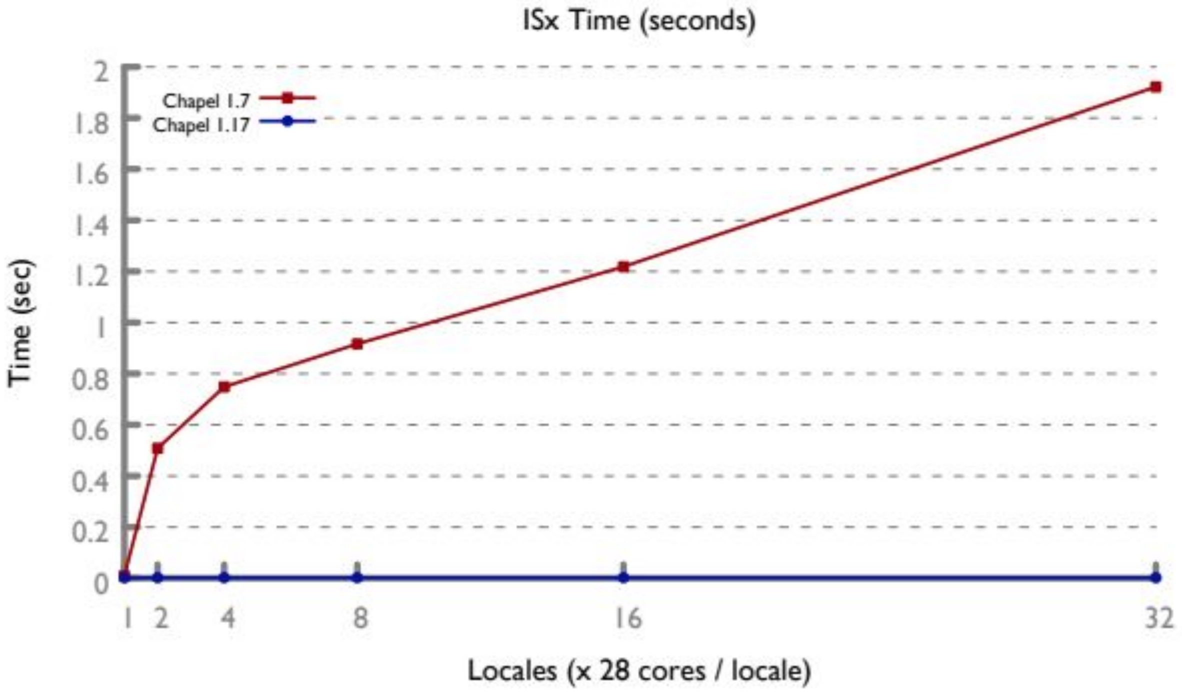
- Chapel v1.7 (2013) vs v1.17 (2018)



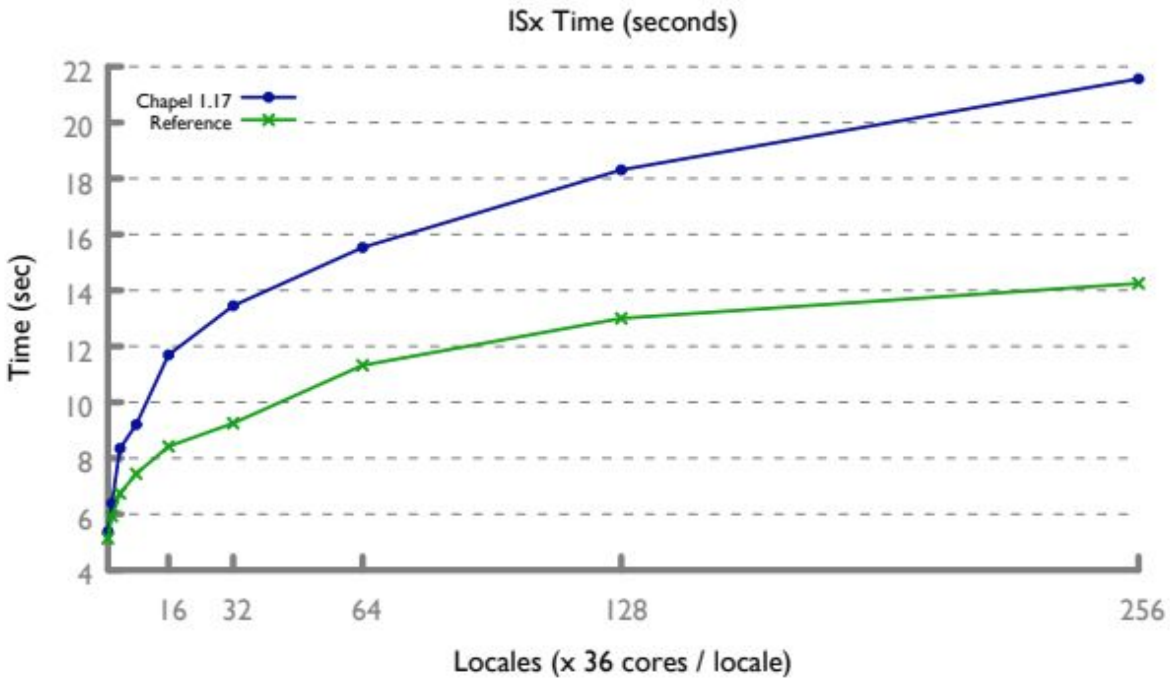
Chapel Performance



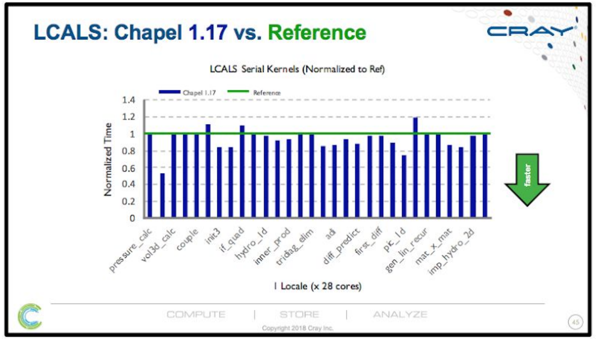
Chapel Performance



Chapel Performance

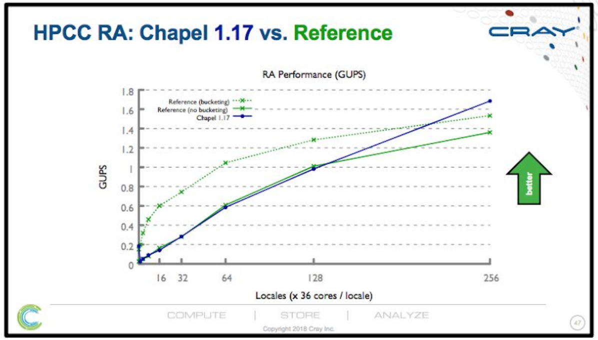


Chapel Performance



LCALS

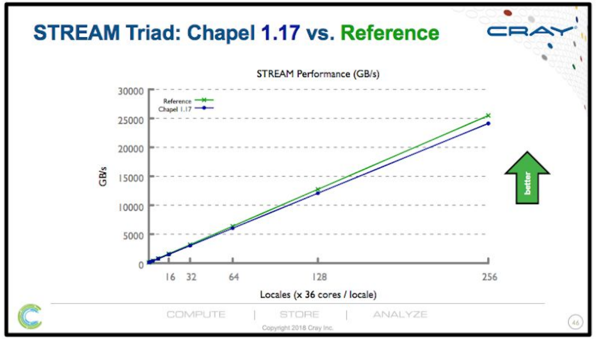
HPCC RA

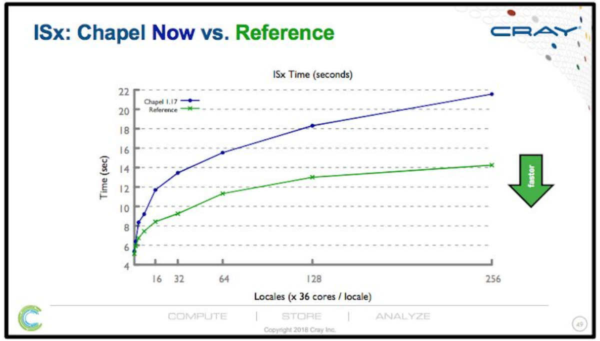


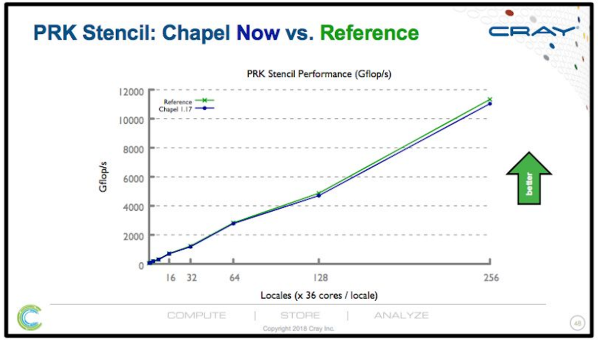
STREAM Triad

ISx

PRK Stencil







LULESH in Chapel

This is the only representation-dependent code.
It specifies:

- data structure choices:
  - structured vs. unstructured mesh
  - local vs. distributed data
  - sparse vs. dense materials arrays
- a few supporting iterators

Domain maps insulate the rest of the application ("the science") from these choices

Conclusions

- Chapel is proposed as a new productive parallel programming language
  - This book chapter focuses on its abstractions
  - See Chapel publications page for ongoing work on performance optimization:
    - https://chapel-lang.org/papers.html

Questions?

References

- Paper:
  - https://chapel-lang.org/publications/PMfPC-Chapel.pdf
- Figures:
  - https://chapel-lang.org/presentations/ChapelForATPESC2016-presented.pdf
  - https://chapel-lang.org/publications/ChapelForCUG2018.pdf