Benchmark Improvements

Chapel Team, Cray Inc.
Chapel version 1.15
April 6, 2017

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.

Outline

- LCALS Benchmark Improvements
- ISx Benchmark Improvements
- Computer Language Benchmarks Game (CLBG)

LCALS Benchmark Improvements

LCALS Improvements: Background

- **Livermore Compiler Analysis Loop Suite**
- **Chapel port first released with version 1.13.0**
- **Version 1.14.0 saw significant performance improvements**
  - Serial variant matching reference version
  - Parallel variant still had room for improvement

LCALS Improvements: This Effort

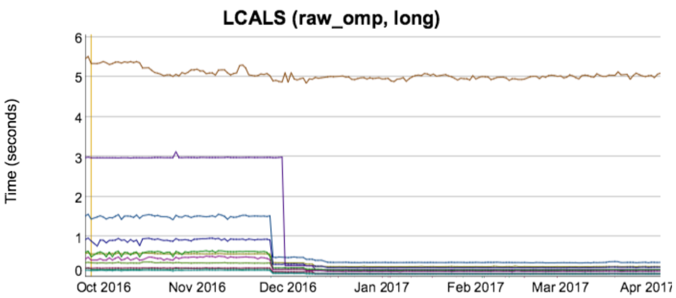
- **Kernel-by-kernel improvements**
  - Made nested loops into loops over 2D domains
  - Improved style of array initializations
  - Changed a 64-bit based range to 32-bit to match the reference version
- **Changed some compiler/execution default options**
  - dropped --data-par-min-granularity=1000
  - added –no-ieee-float
    - both option changes done for a more equal comparison to the reference
- **Added SPMD variant of parallel kernels**
  - Start all tasks
  - Block the kernels up across tasks
  - Execute blocks serially within each task
  - Barrier at the end

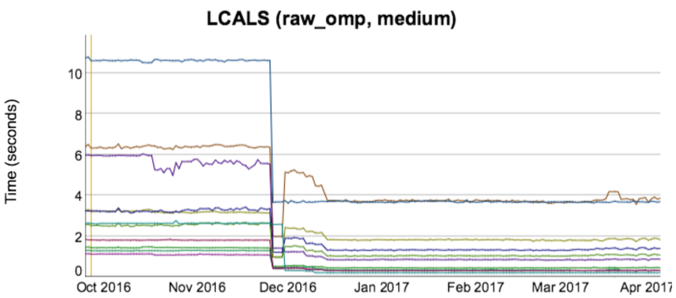
LCALS Improvements: Task Startup Times

- **The amount of work in each kernel is small**
  - Split up across ~24 tasks the amount of work per task is very small
  - Kernels are repeated a large number of times
  - Leads to millions of tasks doing very little work each
- **Task startup times are a large factor in total execution time**
- **Optimization added to the Chapel tasking interface**
  - Enable threads looking for work to more quickly pick up new tasks
  - Dramatically improves task startup times

LCALS Improvements: Impact

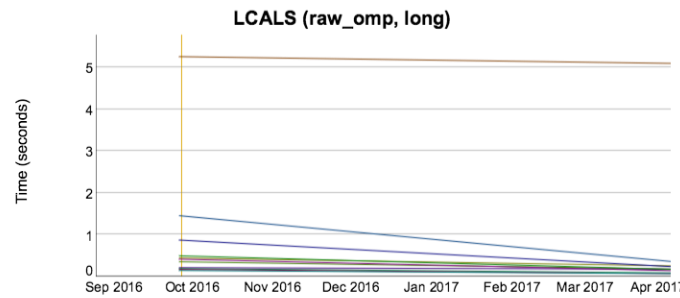
- Improved task startup time increased performance overall

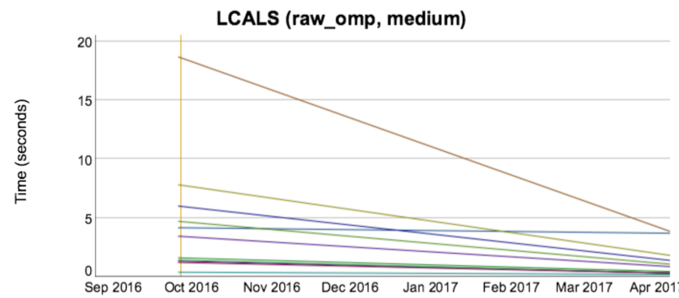


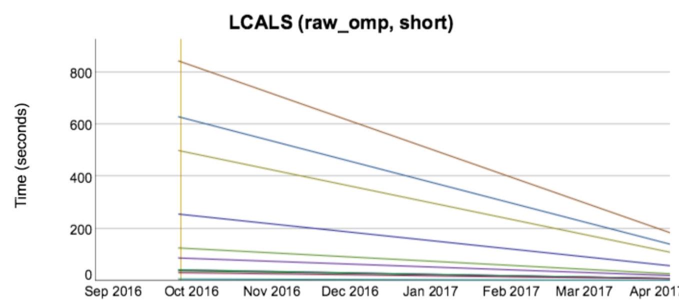


Nightly performance

Release over release performance





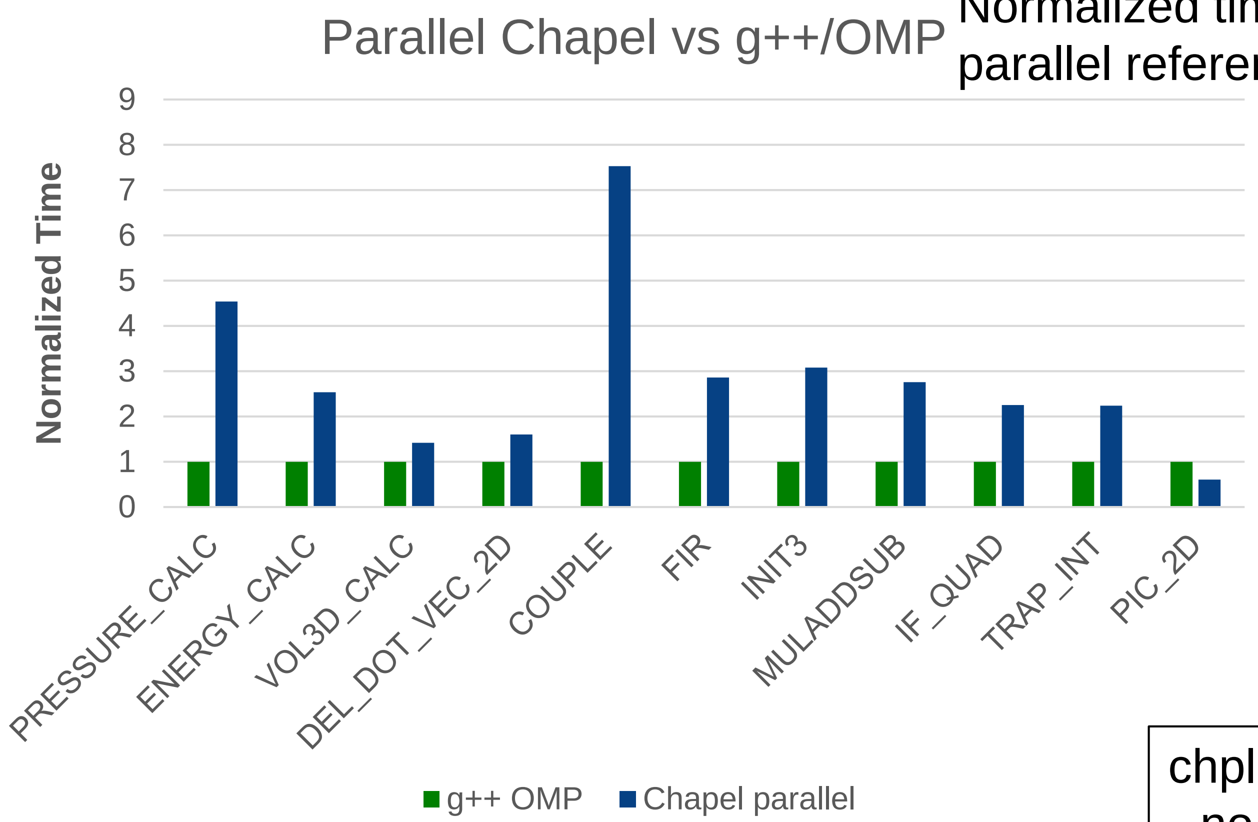


Improvement for short loop size was masked in nightly testing by another change, but is clear in release-over-release testing.

LCALS Status: Impact: Parallel Perf. v1.14.0

Long problem size

Normalized time – parallel reference is 1.0



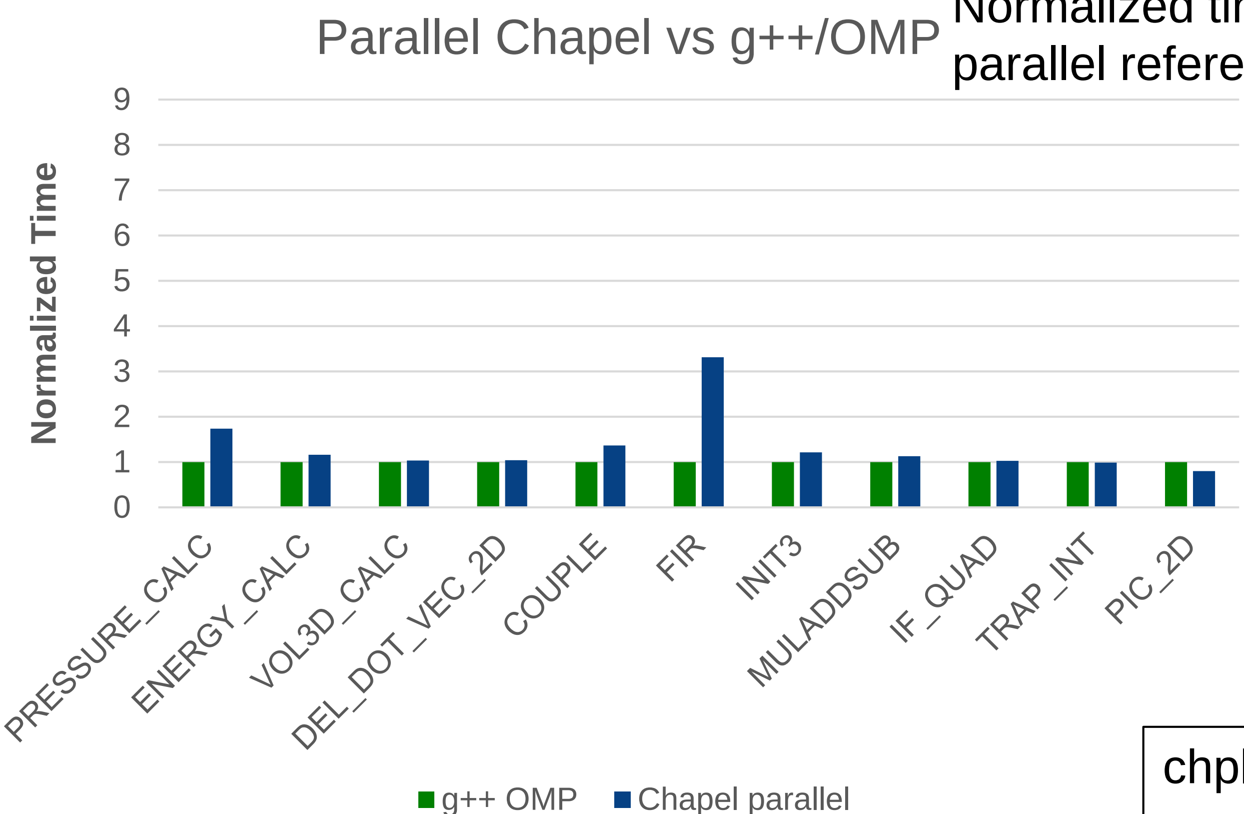
chpl --fast --no-ieee-float

g++ -Ofast -fopenmp

LCALS Status: Impact: Parallel Perf. v1.15.0

Long problem size

Normalized time –
parallel reference is 1.0



chpl --fast
--no-ieee-float

g++ -Ofast -fopenmp

LCALS Improvements: Status and Next Steps

Status:

- The parallel kernels have improved dramatically since version 1.14.0
- A few still lag slightly behind the C+OpenMP reference versions
- But most are effectively matching reference versions

Next Steps:

- There is still some startup overhead to work through
  - Some ideas:
    - start multiple tasks simultaneously
    - improve task counting/sync mechanism

ISx Benchmark Improvements

ISx: Background

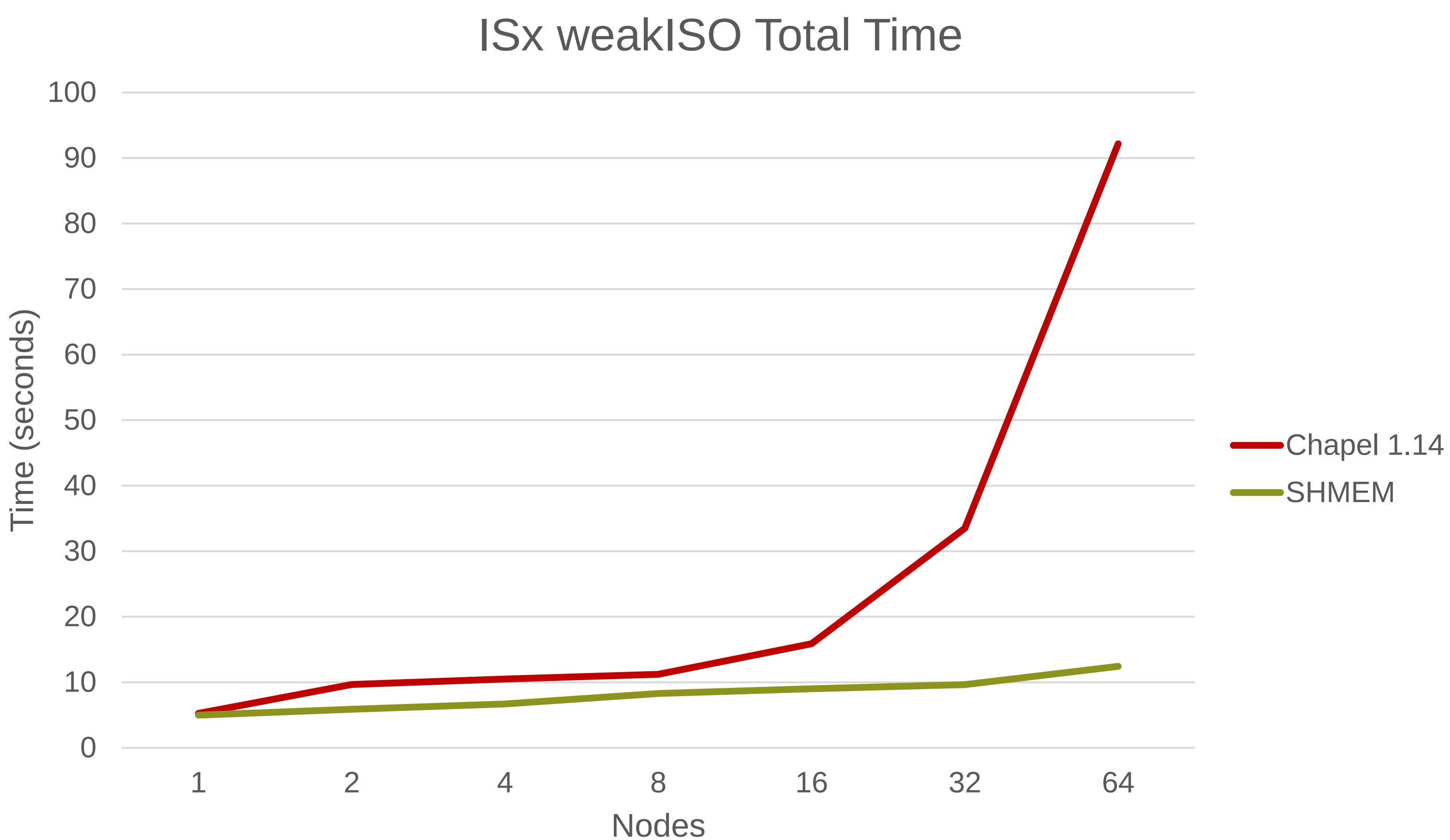
- **Scalable Integer Sort benchmark**
  - Developed at Intel, published at PGAS 2015
  - SPMD-style computation with barriers
  - Punctuated by all-to-all bucket exchange pattern
  - References implemented in SHMEM and MPI

- **Chapel implementation introduced in 1.13 release**
  - Motivation: a common distributed pattern

- **Initial results showed roughly 10x worse performance**

ISx: Background

- **64 nodes on Cray XC**



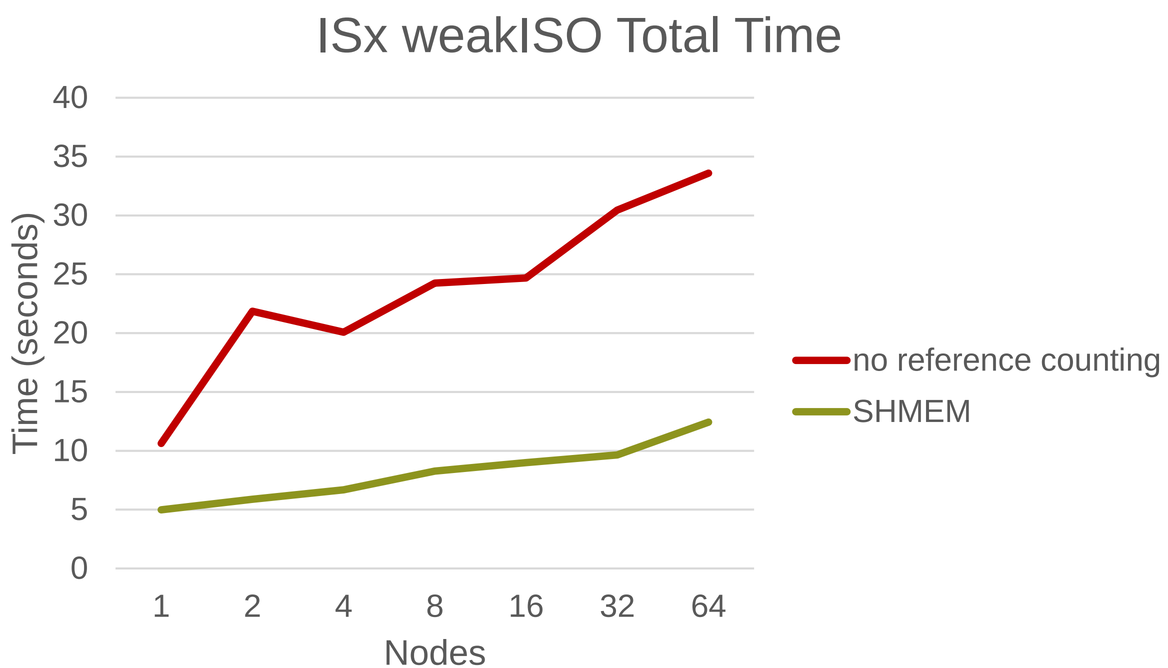
ISx: Background

- **Developed an inelegant optimized version for study**
  - Uses language internals
  - Also called the ‘heroic’ version, referring to user effort
  - Competitive with SHMEM reference

- **Found several areas for improvement:**
  - Reference counted arrays
  - Sliced assignment between arrays
  - Wide-pointer overhead for serial loops

ISx: This Effort - Reference Counting

- **Reference counting arrays was costly at scale**
  - Lots of on-statements and atomic operations
- **Array memory management work resolved this problem**
  - Eliminated need for reference counting entirely



ISx: This Effort – Sliced Assignment Overhead

- **The following pattern was quite slow at scale:**
  - Core operation in all-to-all exchange step

```
// 'Dest' is a remote 1D array, 'Src' is local
Dest[1..10] = Src[1..10];
```

- **Slicing the remote array involved an on-statement**
  - Used to initialize array metadata
  - In all-to-all step, this might mean numCores^2 on-statements
  - Problem worsens as we scale to more locales

- **on-statement also used for slicing Src's domain**
  - Src was declared over a remote const global domain

ISx: This Effort – Sliced Assignment Overhead

- **Fix #1: ArrayViews**
  - Allows for much cheaper creation of a remote array slice
  - Avoids the on-statement entirely
  - Still able to bulk-transfer the array data (single PUT/GET)

- **Fix #2: Declare Src over local domain**
  - Slicing 'Src' now an entirely local operation
  - Before:

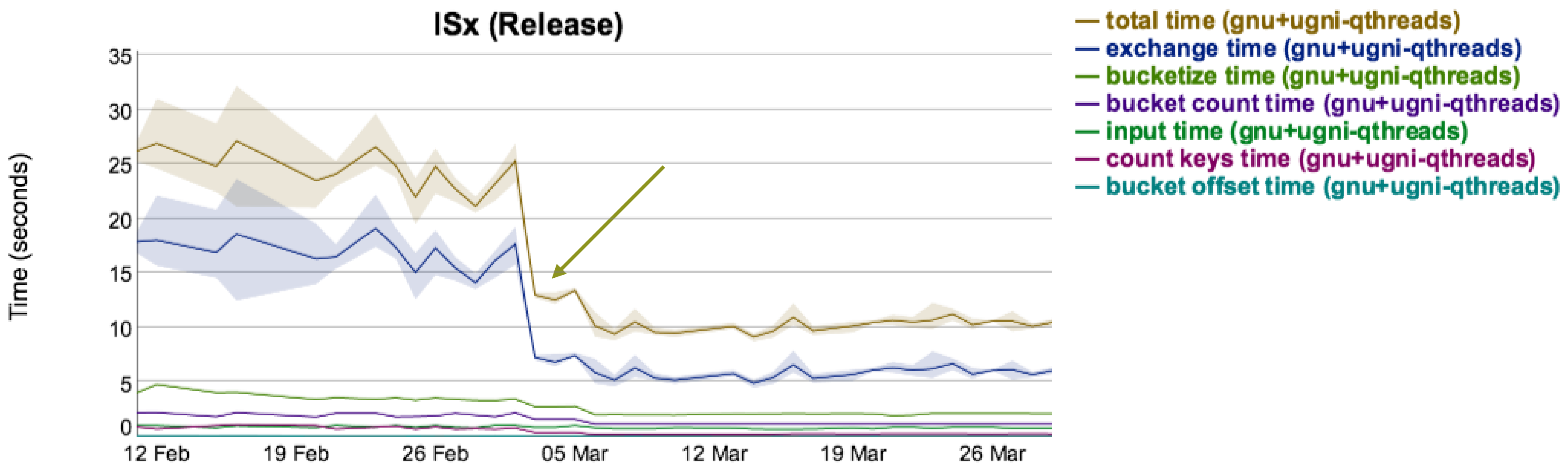
```
var Src : [NumTasksSpace] keyType;
```

  - After:

```
var Src : [0..#numTasks] keyType;
```

ISx: Impact – Sliced Assignment Overhead

- **Result: 2x performance improvement**
  - Also less noise
  - ugni-qthreads, 16 nodes on Cray XC



ISx: This Effort – Wide-pointer overhead

- **Compiler inserts wide-pointers for potentially-remote data**
  - At its simplest, a struct containing a locale ID and pointer
  - Sometimes thwart backend compiler optimizations

- **Can introduce overhead for array accesses**
  - Problematic for subsections like this in ISx:

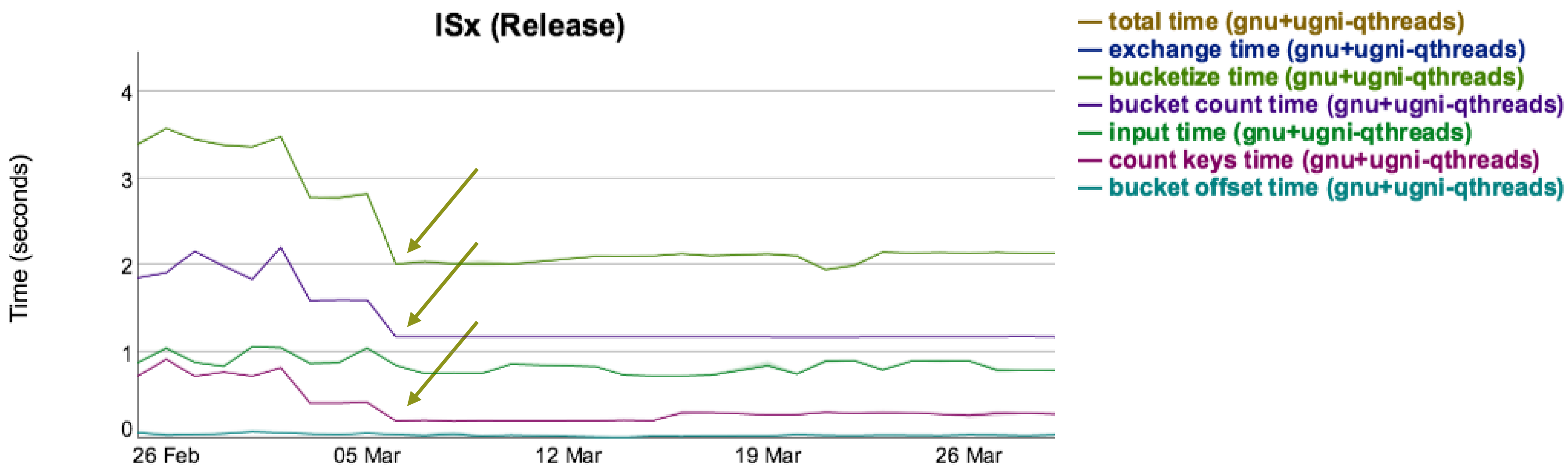
```
for key in myKeys do
  sizes[key/width] += 1;
```

- **1.14 optimized some cases, regressed in 1.15 dev cycle**

- **For 1.15 a similar approach is used, but only for arrays**
  - Disabled with “--[no-]infer-local-fields” compiler flag

ISx: Impact – Wide-pointer overhead

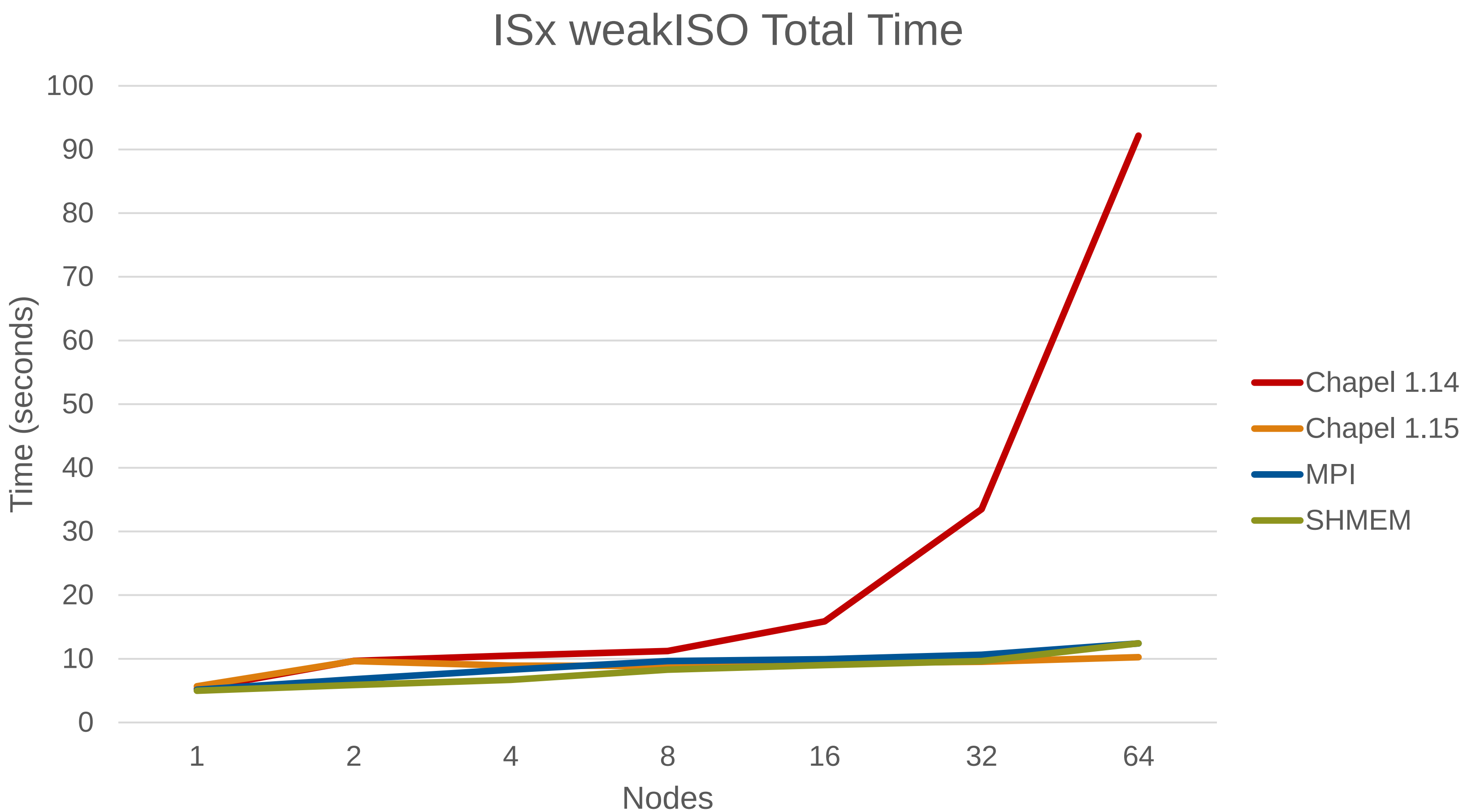
- **Small, but useful, improvements in serial loops**



- **Work remains to match timed subsections of reference**
  - Roughly 20% worse in some cases
  - Note: exchange step, not impacted here, dominates at scale

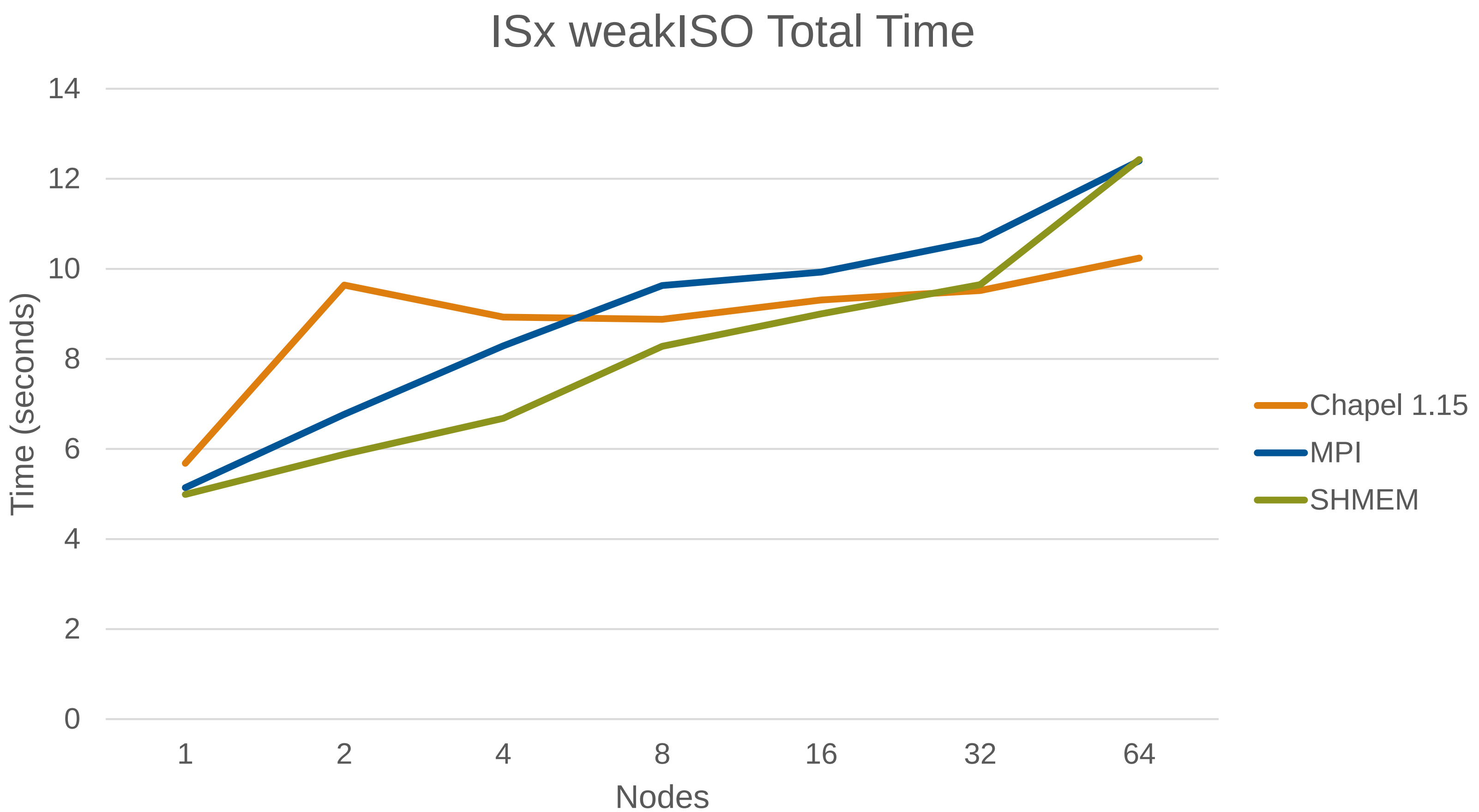
ISx: Performance Summary

- Gathered on Cray XC with default problem size



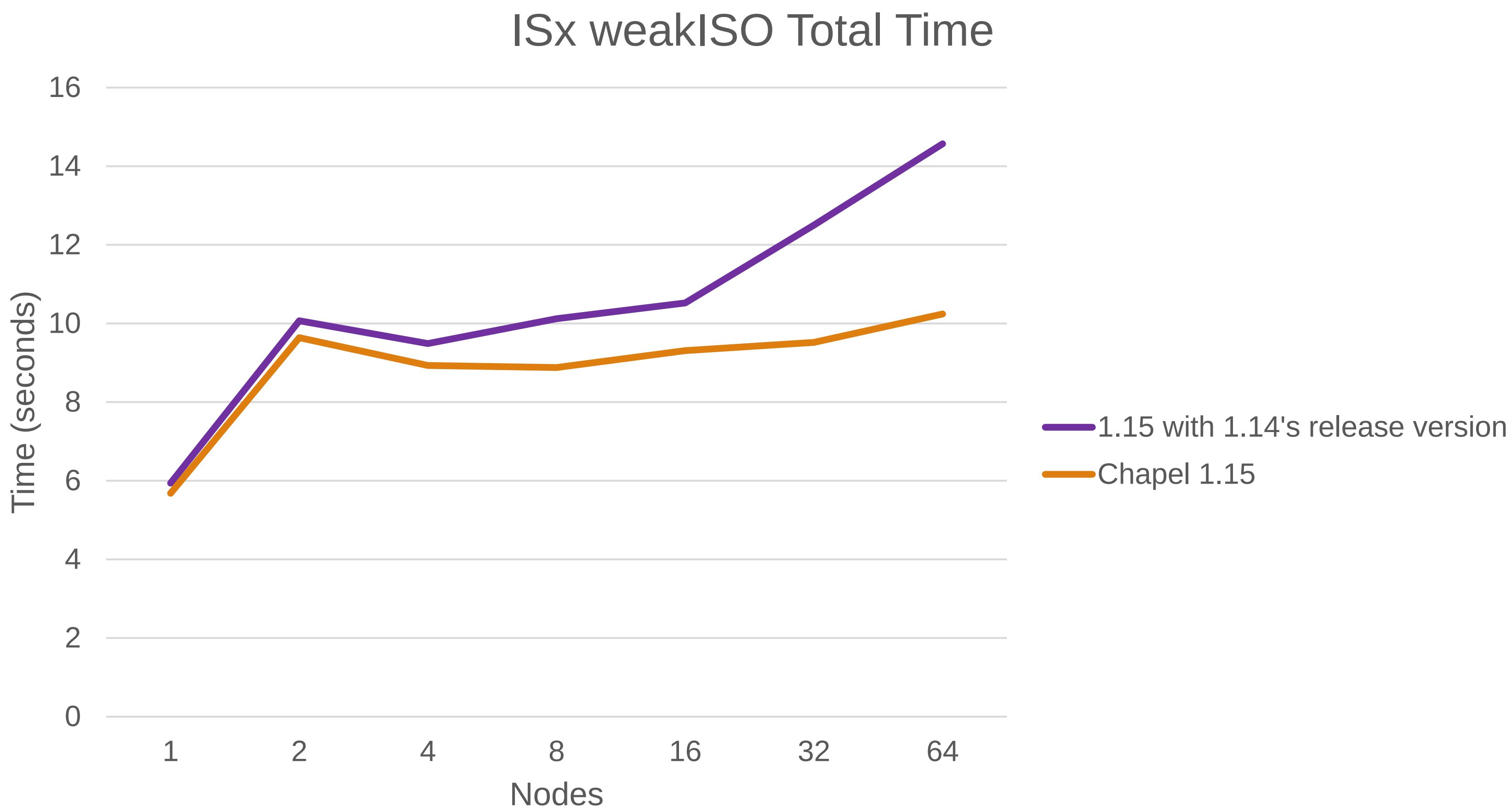
ISx: Performance Summary

- **Gathered on Cray XC with default problem size**
  - Let's zoom in...



ISx: Performance Summary

- **1.15 with 1.14's release version of source code**
  - 1.14 release version did not declare arrays over local domains



ISx: Status and Next Steps

- **Status:** **Big improvements with minor user-code changes**

- **Next Steps:**
  - Address known performance issues
    - Under-counting tasks bug leading to oversubscription
      - Likely to impact other benchmarks as well
    - Eliminate additional wide-pointer overhead
    - Investigate performance at larger scales
    - Compiler should optimize references to remote const domains

  - Continue to investigate performance results
    - Why does Chapel beat SHMEM and MPI at 64 nodes?
    - Why does Chapel scale poorly from 1-4 nodes, then recover?

Computer Language Benchmarks Game (CLBG)

Computer Language Benchmarks Game (CLBG)

The Computer Language Benchmarks Game

64-bit quad core data set

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

Which programs are fast?

Which are succinct? Which are efficient?

Ada C Chapel Clojure C# C++

Dart Erlang F# Fortran Go Hack

Haskell Java JavaScript Lisp Lua

OCaml Pascal Perl PHP Python

Racket Ruby JRuby Rust Scala

Smalltalk Swift TypeScript

Website that supports cross-language game / comparisons

- 13 toy benchmark programs
- exercises key features like:
  - memory management
  - tasking and synchronization
  - vectorization
  - big integers
  - strings and regular expressions
- specific approach prescribed

Take results w/ grain of salt

- other programs may be different
  - not to mention other programmers
- specific to this platform / OS / …

That said, it's one of the only games in town…

Computer Language Benchmarks Game (CLBG)

The Computer Language Benchmarks Game

64-bit quad core data set

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

Which programs are fast?

Which are succinct? Which are efficient?

Ada	C	Chapel	Clojure	C#	C++
Dart	Erlang	F#	Fortran	Go	Hack
Haskell	Java	JavaScript	Lisp	Lua	
OCaml	Pascal	Perl	PHP	Python	
Racket	Ruby	JRuby	Rust	Scala	
Smalltalk	Swift	TypeScript			

Chapel's approach to CLBG:

- want to know how we compare
- strive for entries that are elegant rather than heroic
  - e.g., "Want to learn how program x works? Check out the Chapel version."

CLBG: Website

Can sort results by execution time, code size, memory or CPU use:

The Computer Language Benchmarks Game

chameneos-redux

description

program source code, command-line and measurements

×	source	secs	mem	gz	cpu	cpu load
1.0	**C** gcc #5	**0.60**	820	2863	2.37	100% 100% 98% 100%
1.2	**C++** g++ #5	**0.70**	3,356	1994	2.65	100% 100% 91% 92%
1.7	**Lisp** SBCL #3	**1.01**	55,604	2907	3.93	97% 96% 99% 99%
2.3	**Chapel** #2	**1.39**	76,564	1210	5.43	99% 99% 98% 99%
3.3	**Rust** #2	**2.01**	56,936	2882	7.81	97% 98% 98% 98%
5.6	C++ g++ #2	3.40	1,880	2016	11.88	100% 51% 100% 100%
6.8	Chapel	4.09	66,584	1199	16.25	100% 100% 100% 100%
8.0	**Java** #4	**4.82**	37,132	1607	16.73	98% 98% 54% 99%
8.5	**Haskell** GHC	**5.15**	8,596	989	9.26	79% 100% 2% 2%
10	Java	6.13	53,760	1770	8.78	42% 45% 41% 16%
10	Haskell GHC #4	6.34	6,908	989	12.67	99% 100% 2% 1%
11	**C#** .NET Core	**6.59**	86,076	1400	22.96	99% 82% 78% 91%
11	**Go**	**6.90**	832	1167	24.19	100% 96% 56% 100%
13	Go #2	7.59	1,384	1408	27.65	91% 99% 99% 78%
13	Java #3	7.94	53,232	1267	26.86	54% 96% 98% 94%

The Computer Language Benchmarks Game

chameneos-redux

description

program source code, command-line and measurements

×	source	secs	mem	gz	cpu	cpu load
1.0	**Erlang**	58.90	28,668	**734**	131.19	62% 60% 51% 53%
1.0	**Erlang** HiPE	59.39	25,784	**734**	131.58	60% 56% 56% 54%
1.1	**Perl** #4	5 min	14,084	**785**	7 min	40% 40% 29% 28%
1.1	**Racket**	5 min	132,120	**791**	5 min	1% 0% 0% 100%
1.1	Racket #2	175.88	116,488	842	175.78	100% 1% 1% 0%
1.2	**Python 3** #2	236.84	7,908	**866**	5 min	24% 48% 27% 45%
1.3	**Ruby**	90.52	9,396	**920**	137.53	35% 35% 35% 34%
1.3	**Ruby** JRuby	48.78	628,968	**928**	112.15	65% 60% 49% 58%
1.3	**Go** #5	11.05	832	**957**	32.48	75% 74% 75% 73%
1.3	**Haskell** GHC #4	6.34	6,908	**989**	12.67	99% 100% 2% 1%
1.3	Haskell GHC	5.15	8,596	989	9.26	79% 100% 2% 2%
1.6	**OCaml** #3					32% 38% 37% 39%
1.6	Go					00% 96% 56% 100%
1.6	**Chapel**					0% 100% 100% 100%
1.6	Chapel #2					99% 99% 98% 99%

gz == code size metric
strip comments and extra whitespace, then gzip

CLBG: Website

Can also compare languages pair-wise:

The Computer Language Benchmarks Game

Chapel programs versus Go

all other Chapel programs & measurements

by benchmark task performance

regex-redux

source	secs	mem	gz	cpu	cpu load
Chapel	**10.02**	1,022,052	477	19.68	99% 72% 14% 12%
Go	29.51	352,804	798	61.51	77% 49% 43% 40%

binary-trees

source	secs	mem	gz	cpu	cpu load
Chapel	**14.32**	324,660	484	44.15	100% 58% 78% 75%
Go	34.77	269,068	654	132.04	95% 97% 95% 95%

fannkuch-redux

source	secs	mem	gz	cpu	cpu load
Chapel	**11.38**	46,056	728	45.18	100% 99% 99% 100%
Go	15.81	1,372	900	62.92	100% 100% 99% 99%

CLBG: Website

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

```
proc main() {
  printColorEquations();

  const group1 = [i in 1..popSize1] new Chameneos(i, ((i-1)%3):Color);
  const group2 = [i in 1..popSize2] new Chameneos(i, colors10[i]);

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

  print(group1);
  print(group2);

  for c in group1 do delete c;
  for c in group2 do delete c;
}


//
// Print the results of getNewColor() for all color pairs.
//
proc printColorEquations() {
  for c1 in Color do
    for c2 in Color do
      writeln(c1, " + ", c2, " -> ", getNewColor(c1, c2));
  writeln();
}


//
// Hold meetings among the population by creating a shared meeting
// place, and then creating per-chameneos tasks to have meetings.
//
proc holdMeetings(population, numMeetings) {
  const place = new MeetingPlace(numMeetings);

  coforall c in population do            // create a task per chameneos
    c.haveMeetings(place, population);

  delete place;
}
```

excerpt from 1210 gz Chapel #2 entry

```
void get_affinity(int* is_smp, cpu_set_t* affinity1, cpu_set_t* affinity2)
{
    cpu_set_t                   active_cpus;
    FILE*                       f;
    char                        buf [2048];
    char const*                 pos;
    int                         cpu_idx;
    int                         physical_id;
    int                         core_id;
    int                         cpu_cores;
    int                         apic_id;
    size_t                      cpu_count;
    size_t                      i;

    char const*                 processor_str       = "processor";
    size_t                      processor_str_len   = strlen(processor_str);
    char const*                 physical_id_str     = "physical id";
    size_t                      physical_id_str_len = strlen(physical_id_str);
    char const*                 core_id_str         = "core id";
    size_t                      core_id_str_len     = strlen(core_id_str);
    char const*                 cpu_cores_str       = "cpu cores";
    size_t                      cpu_cores_str_len   = strlen(cpu_cores_str);

    CPU_ZERO(&active_cpus);
    sched_getaffinity(0, sizeof(active_cpus), &active_cpus);
    cpu_count = 0;
    for (i = 0; i != CPU_SETSIZE; i += 1)
    {
        if (CPU_ISSET(i, &active_cpus))
        {
            cpu_count += 1;
        }
    }

    if (cpu_count == 1)
    {
        is_smp[0] = 0;
        return;
    }

    is_smp[0] = 1;
    CPU_ZERO(affinity1);
```

excerpt from 2863 gz C gcc #5 entry

What's new with the CLBG?

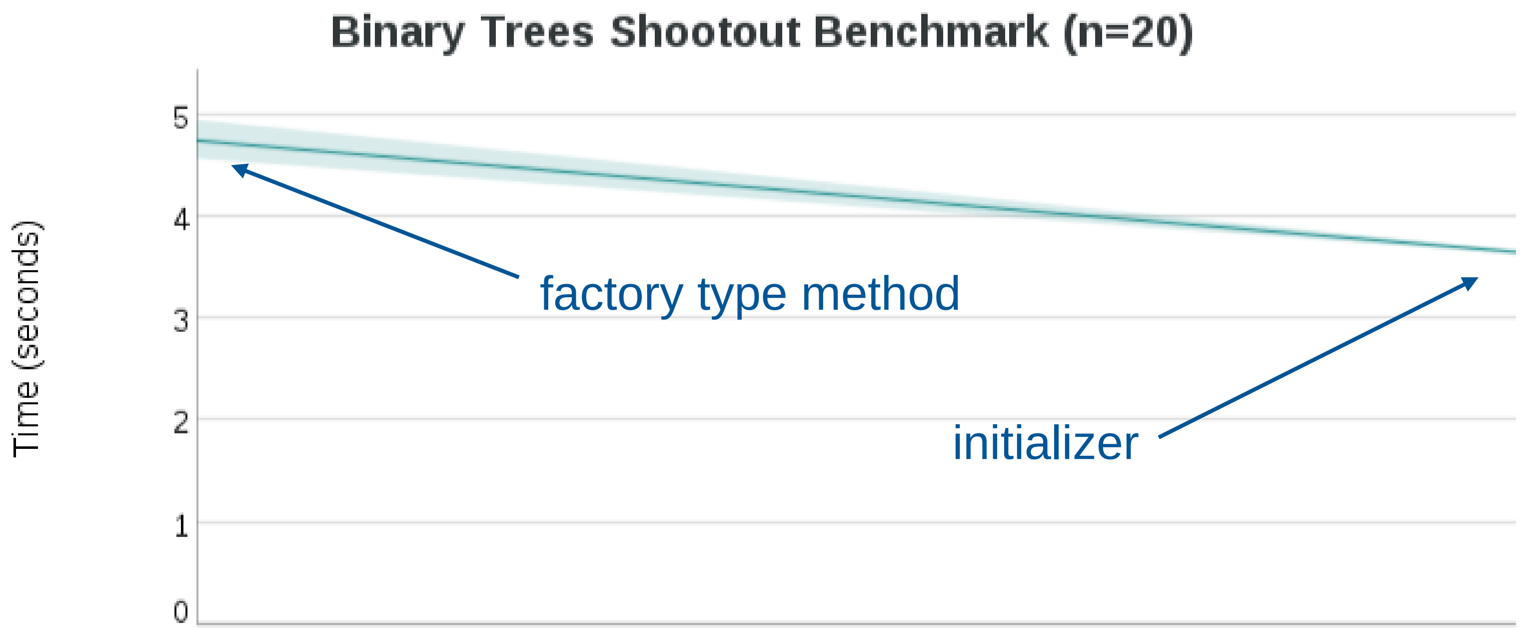
- **Two programs changed their official definitions:**

  binary-trees:
  - improved checksum to avoid false positives at 1/2, 1/4, 1/8 the memory
  - eliminated per-node data field
  - changed what trees are allocated and freed, slightly
  - increased the problem size

  regex:
  - changed the regular expression used
  - renamed the test to regex-redux
  - several versions are not currently passing due to these changes
    - our current standings may be due in part to this

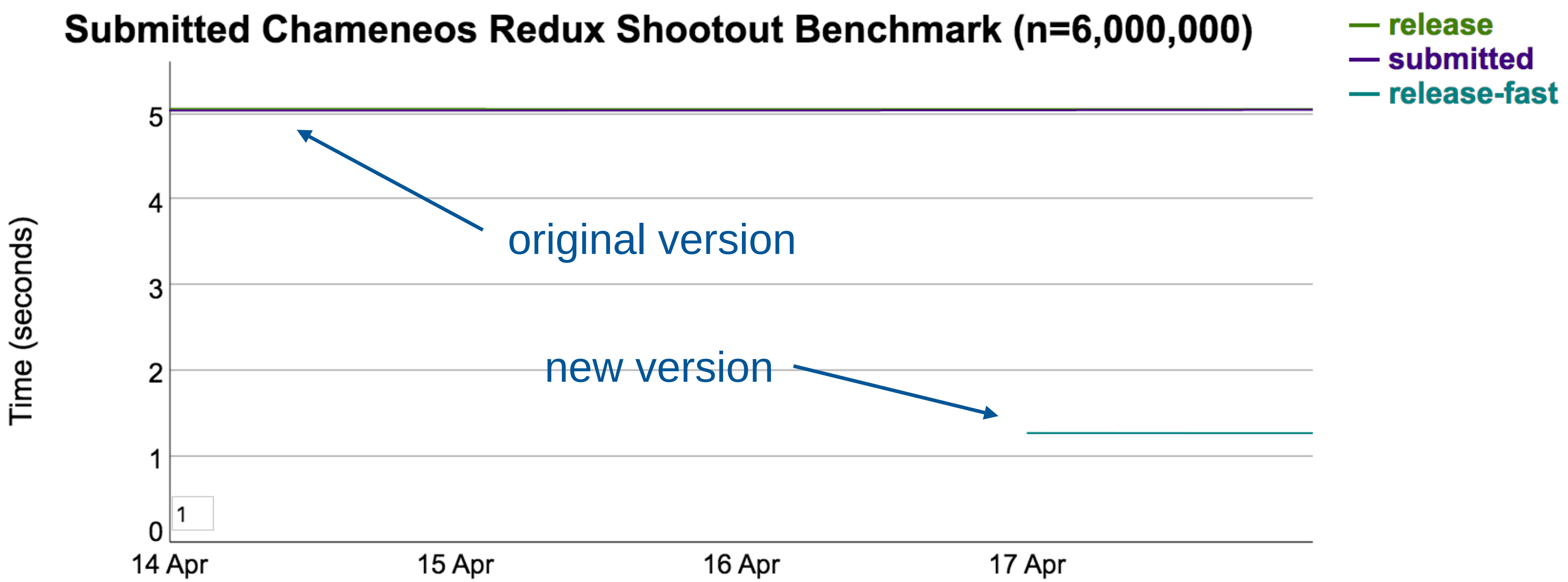
What's new with the Chapel CLBG entries?

- **We've submitted some new versions:**
  binary-trees: used an initializer rather than a factory type method



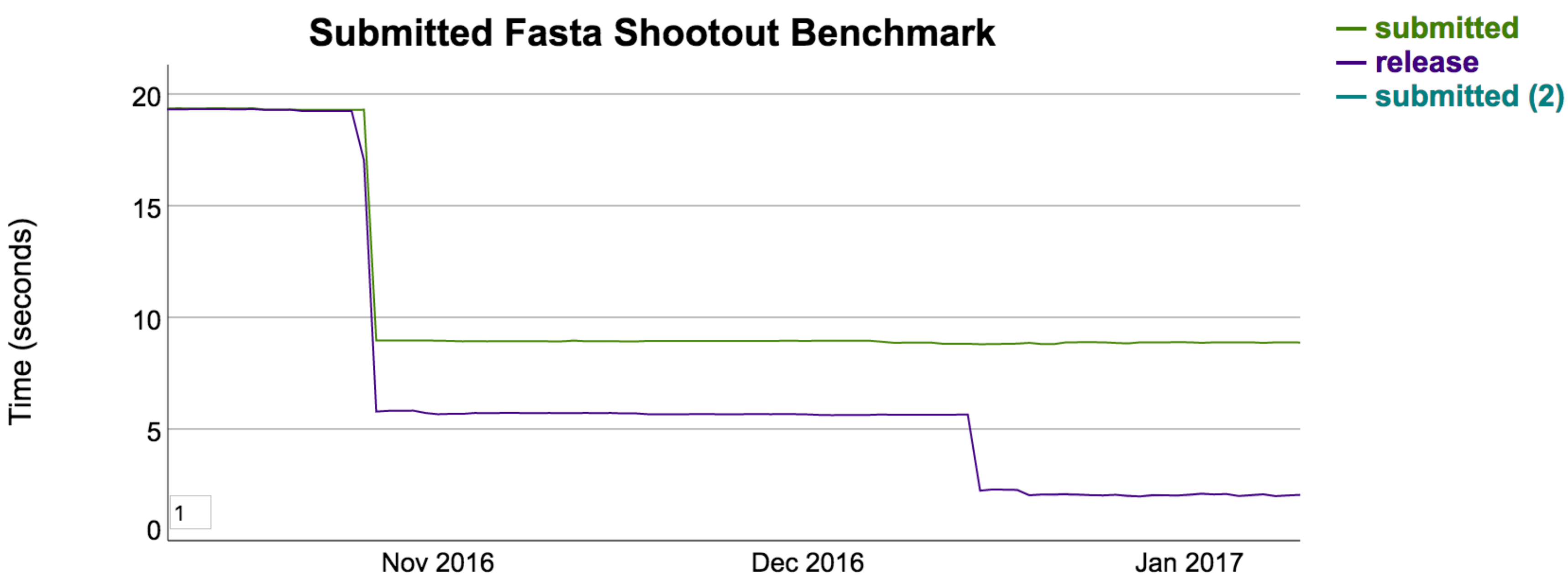
What's new with the Chapel CLBG entries?

- **We've submitted some new versions:**
  - **binary-trees:** used an initializer rather than a factory type method
  - **chameneos-redux:** increased parallelism and tuned a spin-wait



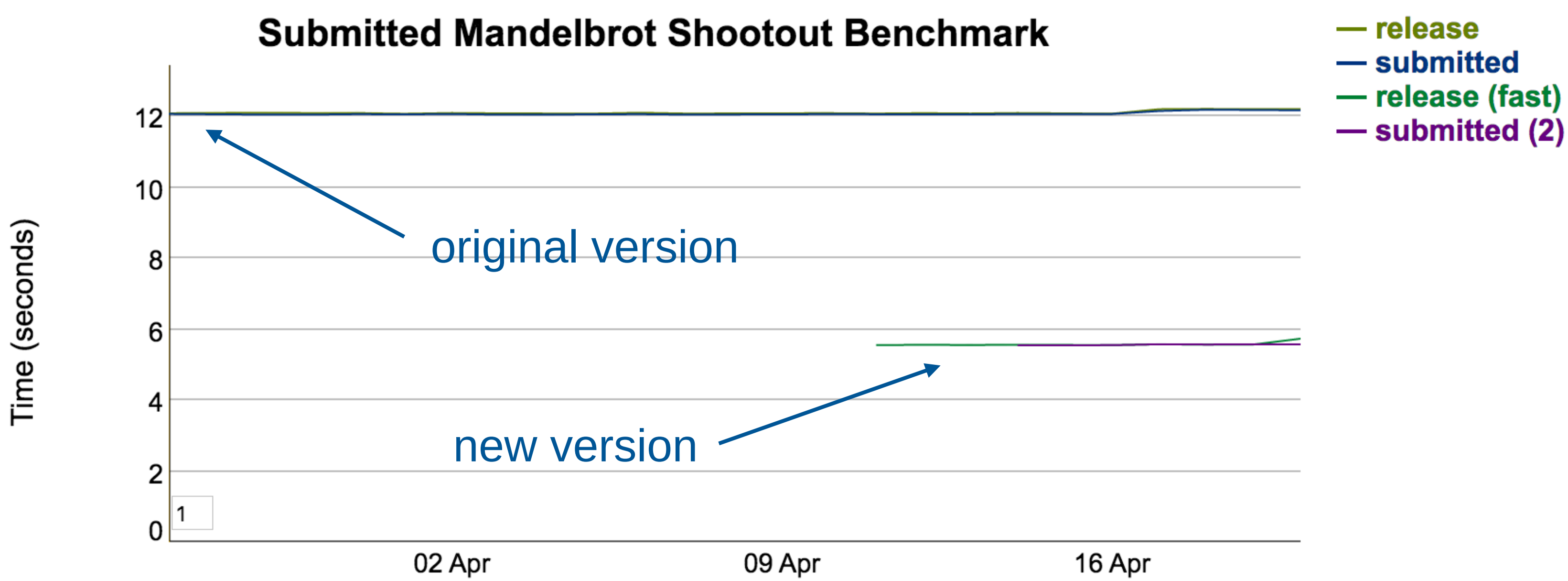
What's new with the Chapel CLBG entries?

- **We've submitted some new versions:**
  - **binary-trees:** used an initializer rather than a factory type method
  - **chameneos-redux:** increased parallelism and tuned a spin-wait
  - **fasta:** implemented a parallel version and tuned for clarity and speed
    - also, changed some 'var' declarations due to const-checking improvements



What's new with the Chapel CLBG entries?

- **We've submitted some new versions:**
  - **binary-trees:** used an initializer rather than a factory type method
  - **chameneos-redux:** increased parallelism and tuned a spin-wait
  - **fasta:** implemented a parallel version and tuned for clarity and speed
    - also, changed some 'var' declarations due to const-checking improvements
  - **mandelbrot:** accelerated by hoisting values and using tuples of values



What's new with the Chapel CLBG entries?

- **We've submitted some new versions:**
  binary-trees: used an initializer rather than a factory type method
  chameneos-redux: increased parallelism and tuned a spin-wait
  fasta: implemented a parallel version and tuned for clarity and speed
    - also, changed some 'var' declarations due to const-checking improvements

  mandelbrot: accelerated by hoisting values and using tuples of values
  meteor-fast: fixed a race condition caused by array memory changes
    - textbook example of an array being used by a 'begin' task

  pidigits: submitted a version that uses 'bigint's
    - currently the #1 fastest version, and also quite elegant

- **Note that some of these changes followed the 1.15 release**
  - As such, not all are found in examples/benchmarks/shootout/ for 1.15

CLBG: Chapel Standings as of Oct 17th

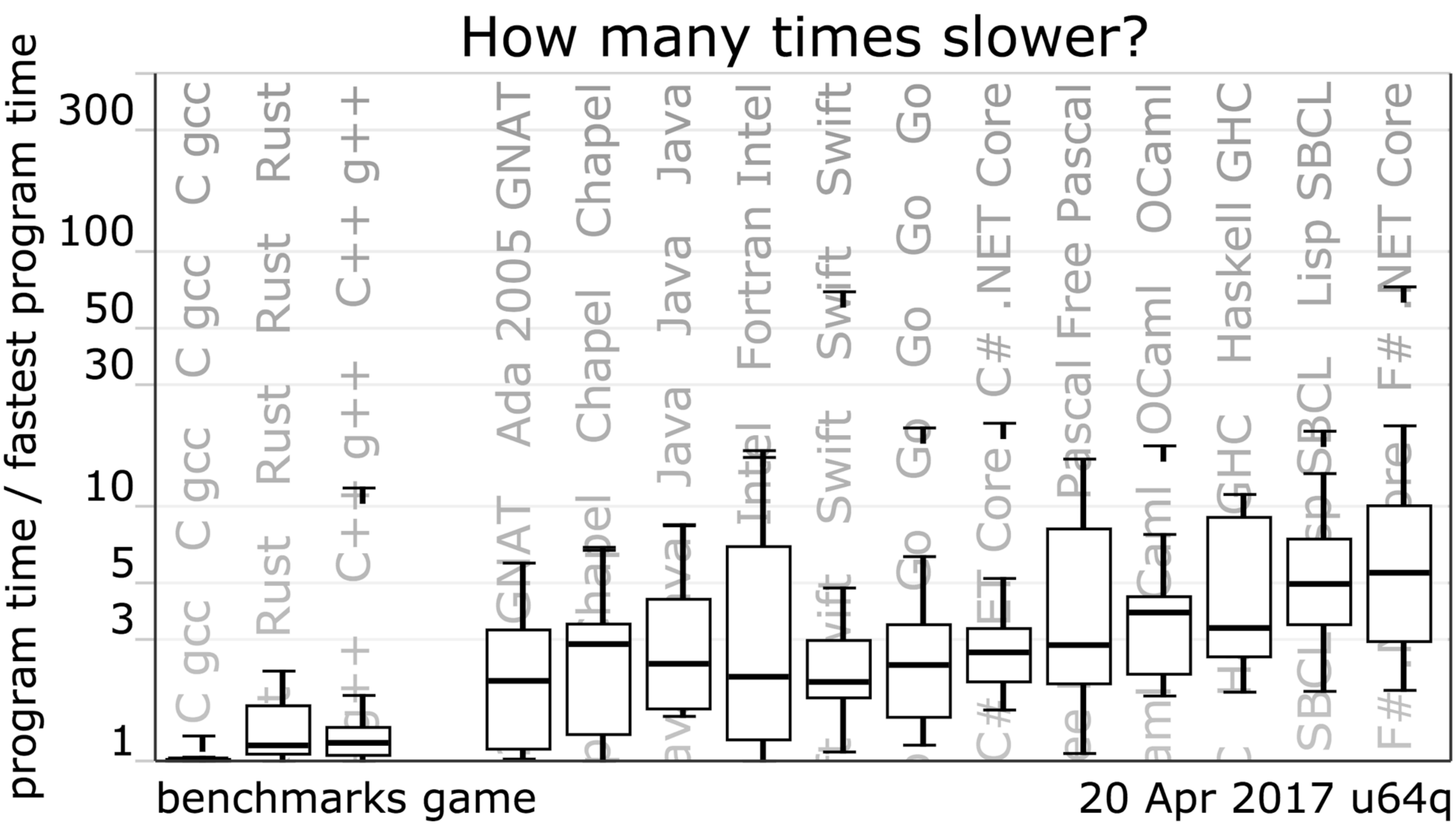
- **8 / 13 programs in top-20 fastest:**
  - one #1 fastest:
    pidigits
  - 2 others in the top-5 fastest:
    meteor-contest
    thread-ring
  - 2 others in the top-10 fastest:
    chameneos-redux
    fannkuch-redux
  - 3 others in the top-20 fastest:
    binary-trees
    n-body
    spectral-norm
- **8 / 13 programs in top-20 smallest:**
  - two #1 smallest:
    n-body
    thread-ring
  - 2 others in the top-5 smallest:
    pidigits
    spectral-norm
  - 4 others in the top-20 smallest:
    chameneos-redux
    mandelbrot
    meteor-contest
    regex-dna

CLBG: Chapel Standings as of Apr 20th

- **12 /13 programs in top-20 fastest:**
  - one #1 fastest:
    pidigits
  - 3 others in the top-5 fastest:
    chameneos-redux
    meteor-contest
    thread-ring
  - 3 others in the top-10 fastest:
    fannkuch-redux
    fasta
    mandelbrot
  - 5 others in the top-20 fastest:
    binary-trees
    k-nucleotide
    n-body
    regex-redux
    spectral-norm
- **8 / 13 programs in top-20 smallest:**
  - two #1 smallest:
    n-body
    thread-ring
  - 2 others in the top-5 smallest:
    pidigits
    spectral-norm
  - 1 other in the top-10 smallest:
    regex-redux
  - 3 others in the top-20 smallest:
    chameneos-redux
    mandelbrot
    meteor-contest

CLBG: Website's fast-faster-fastest graph

Site summary: relative performance (sorted by geometric mean)



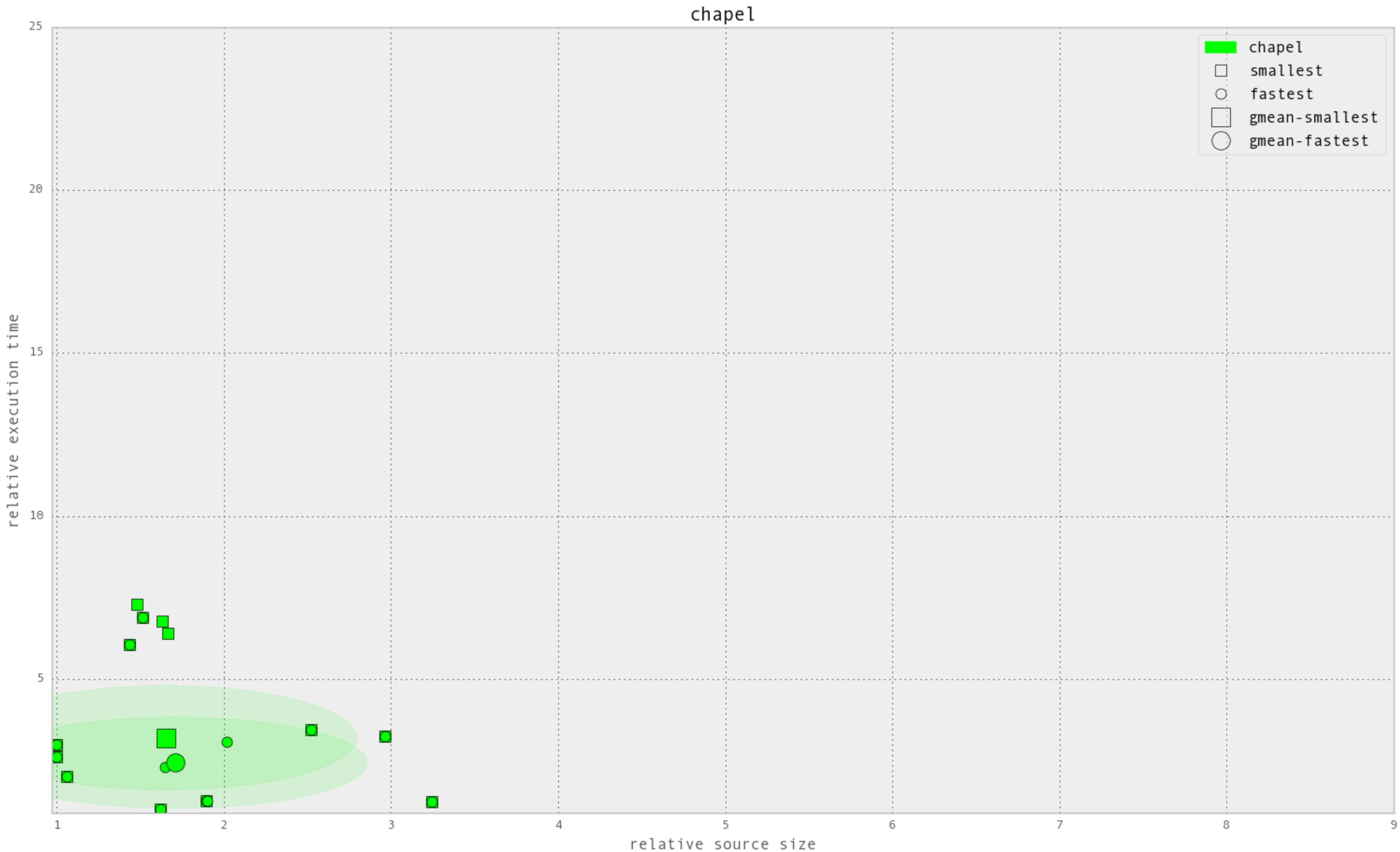
CLBG: Website

- **site has voiced good philosophy about too-easy answers**

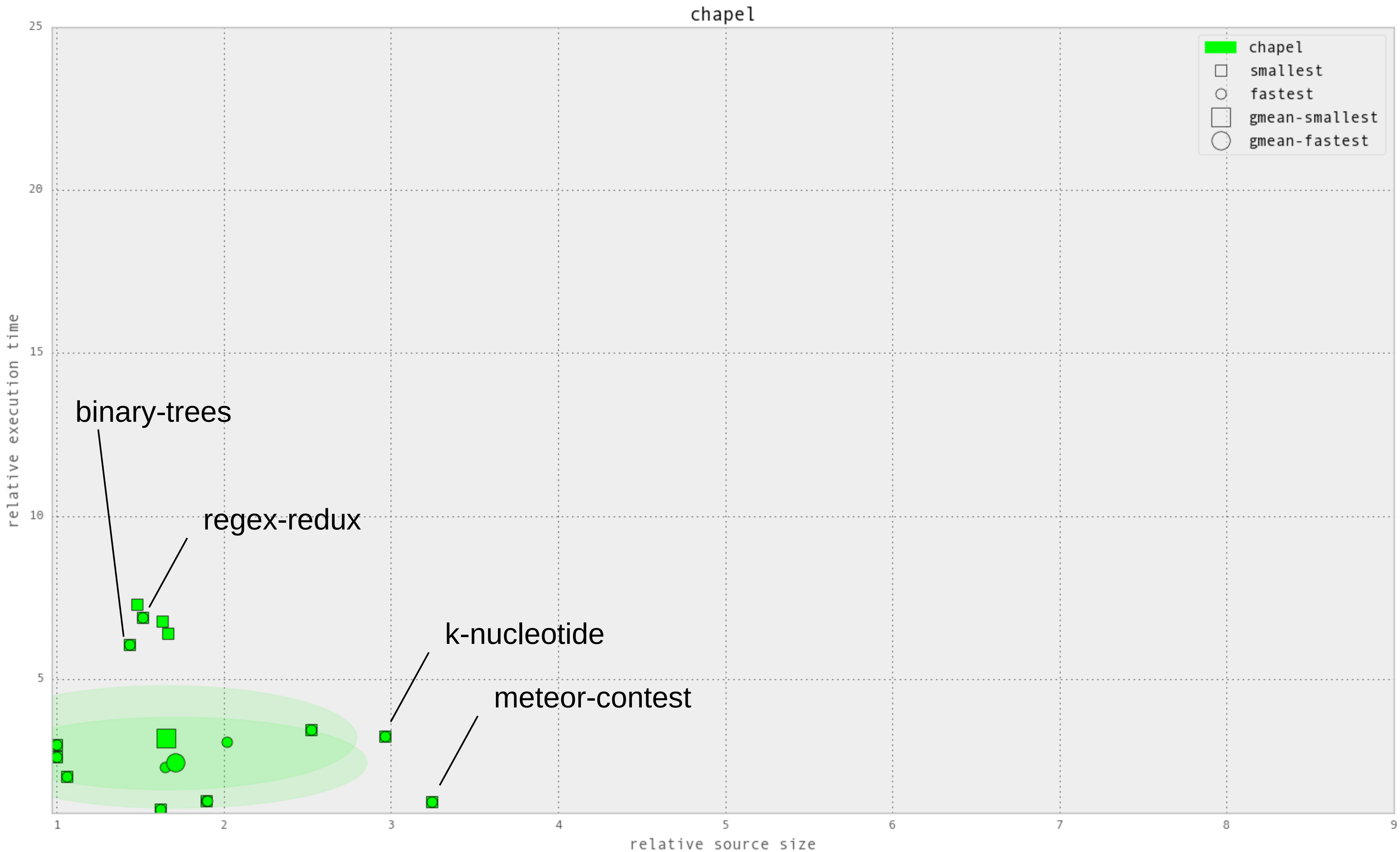
> **We want easy answers**, but easy answers are often incomplete or wrong. You and I know, there's more we should understand:
>
> stories details fast? conclusions

- **yet, most readers probably still jump to conclusions**
  - execution time dominates default (or only) views of results
  - it's simply human nature
- **we're interested in elegance as well as performance**
  - elegance is obviously in the eye of the beholder
    - we compare source codes manually
    - but then use CLBG's code size metric as a quantitative stand-in
  - want to be able to compare both axes simultaneously
  - to that end, we used scatter plots to compare implementations

Chapel entries



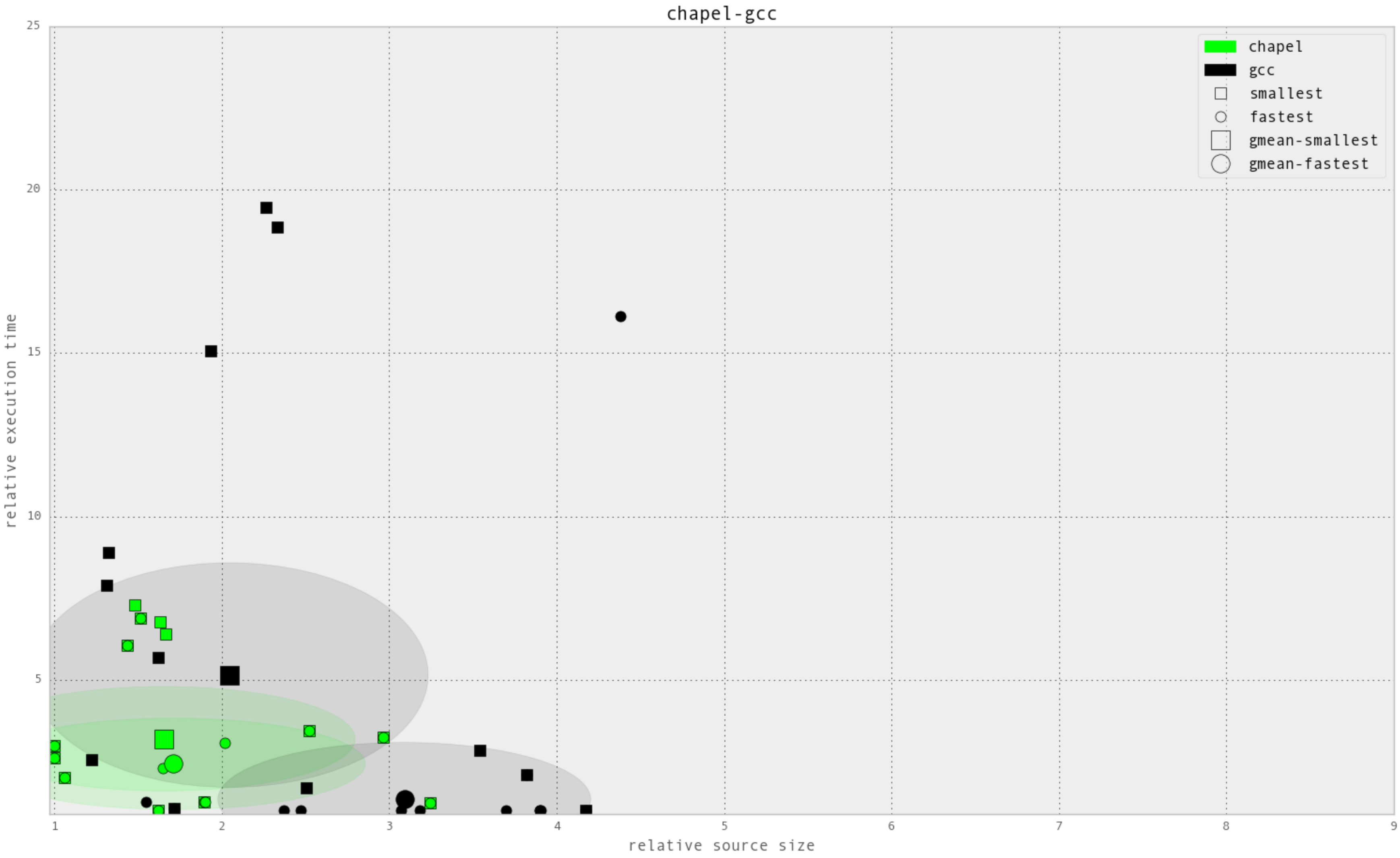
Chapel entries (noting outliers)



Chapel vs. 9 other languages

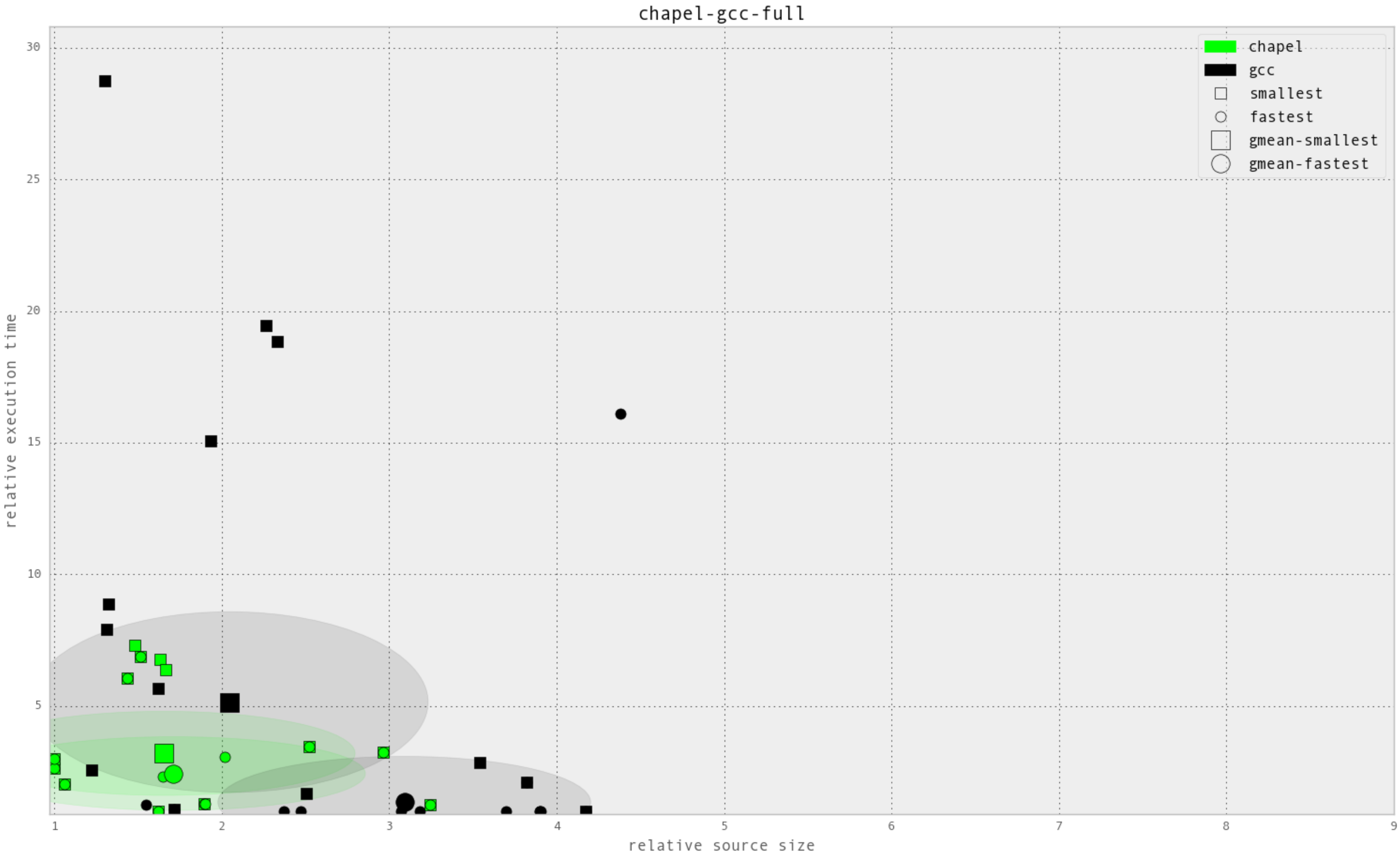
Chapel vs. 9 other languages (zoomed out)

Chapel vs. C

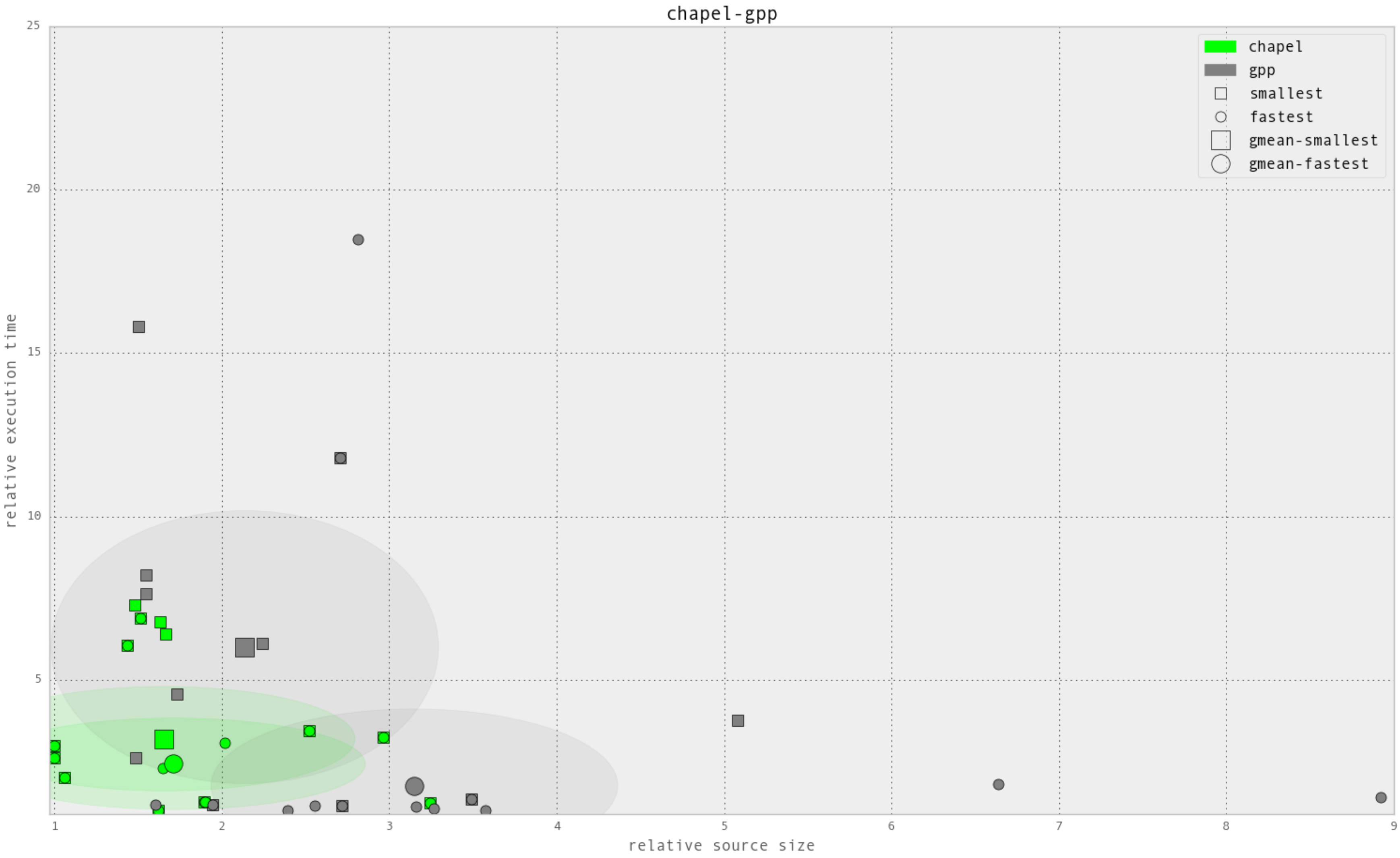


chapel-gcc
chapel
gcc
smallest
fastest
gmean-smallest
gmean-fastest
relative execution time
relative source size
25
20
15
10
5
1
2
3
4
5
6
7
8
9

Chapel vs. C (zoomed out)

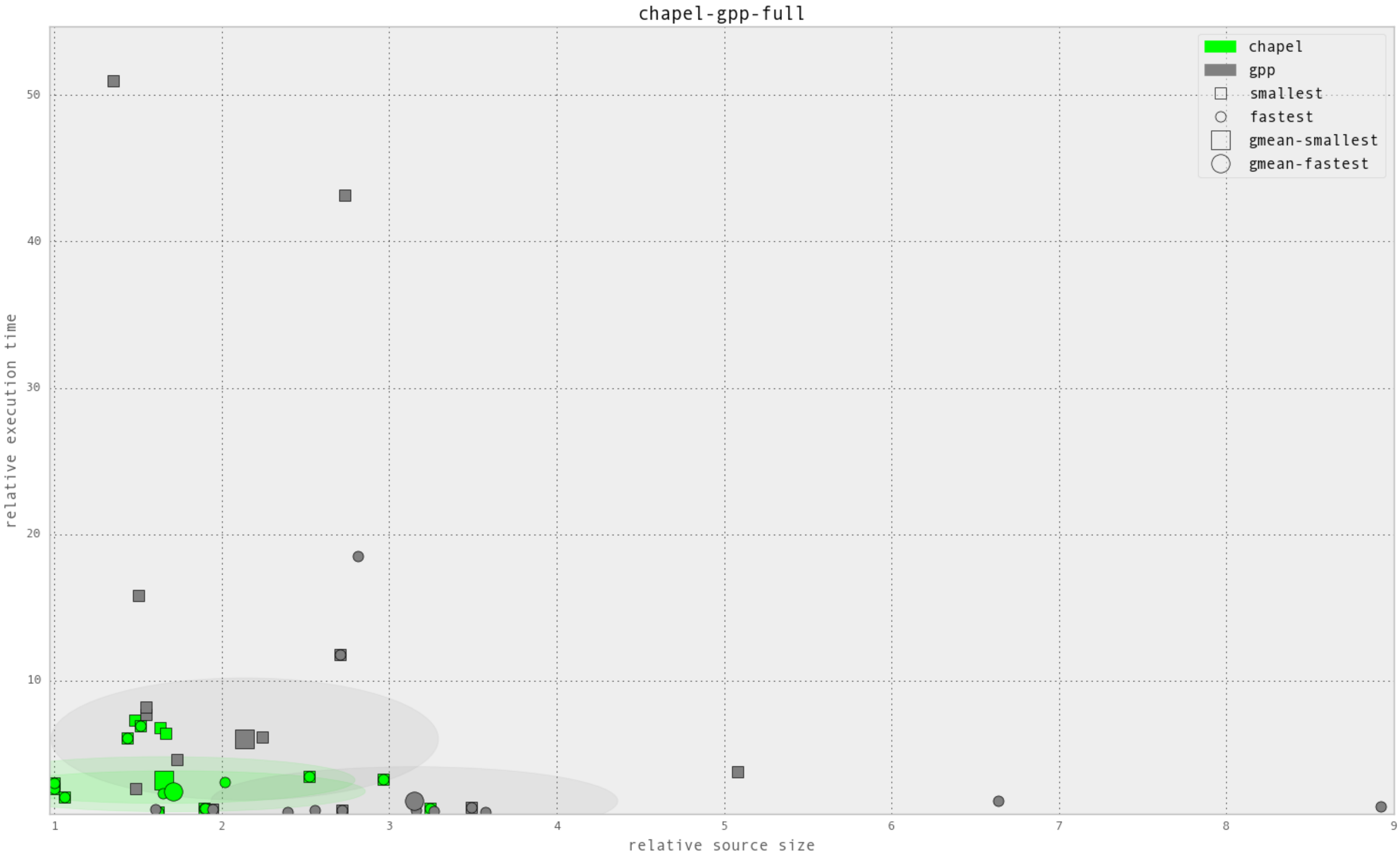


Chapel vs. C++

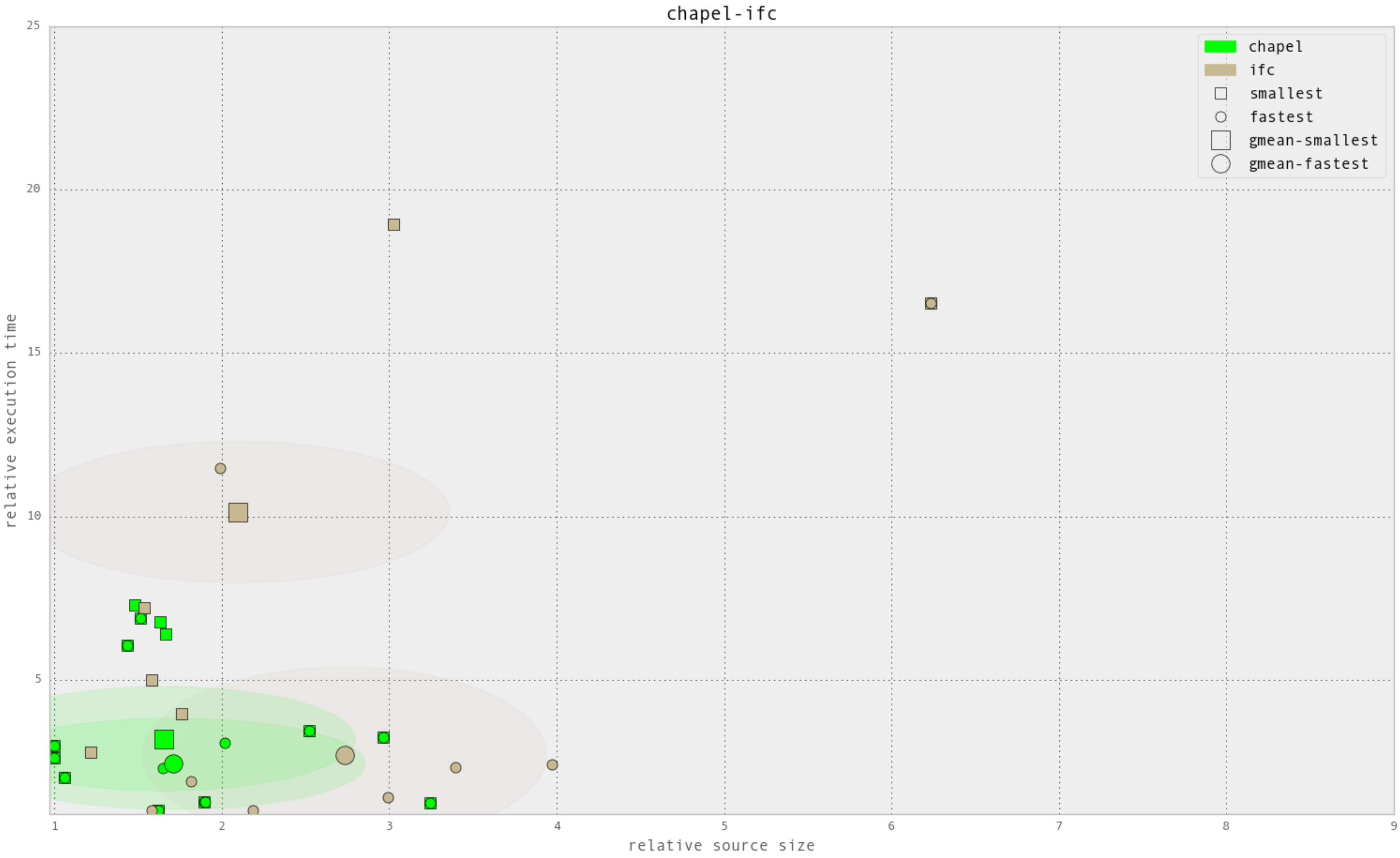


chapel-gpp
chapel
gpp
smallest
fastest
gmean-smallest
gmean-fastest
relative execution time
relative source size
25
20
15
10
5
1
2
3
4
5
6
7
8
9

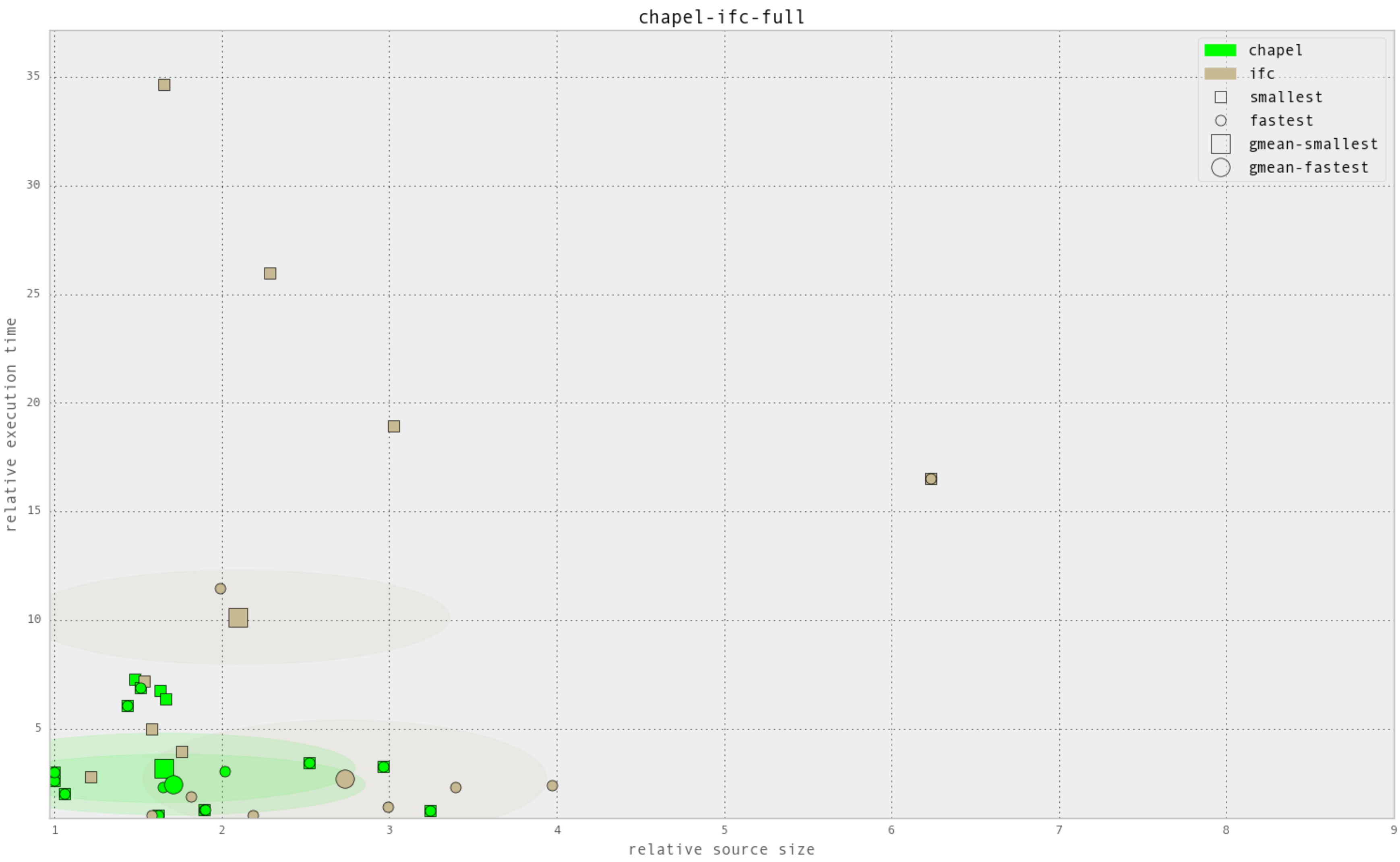
Chapel vs. C++ (zoomed out)



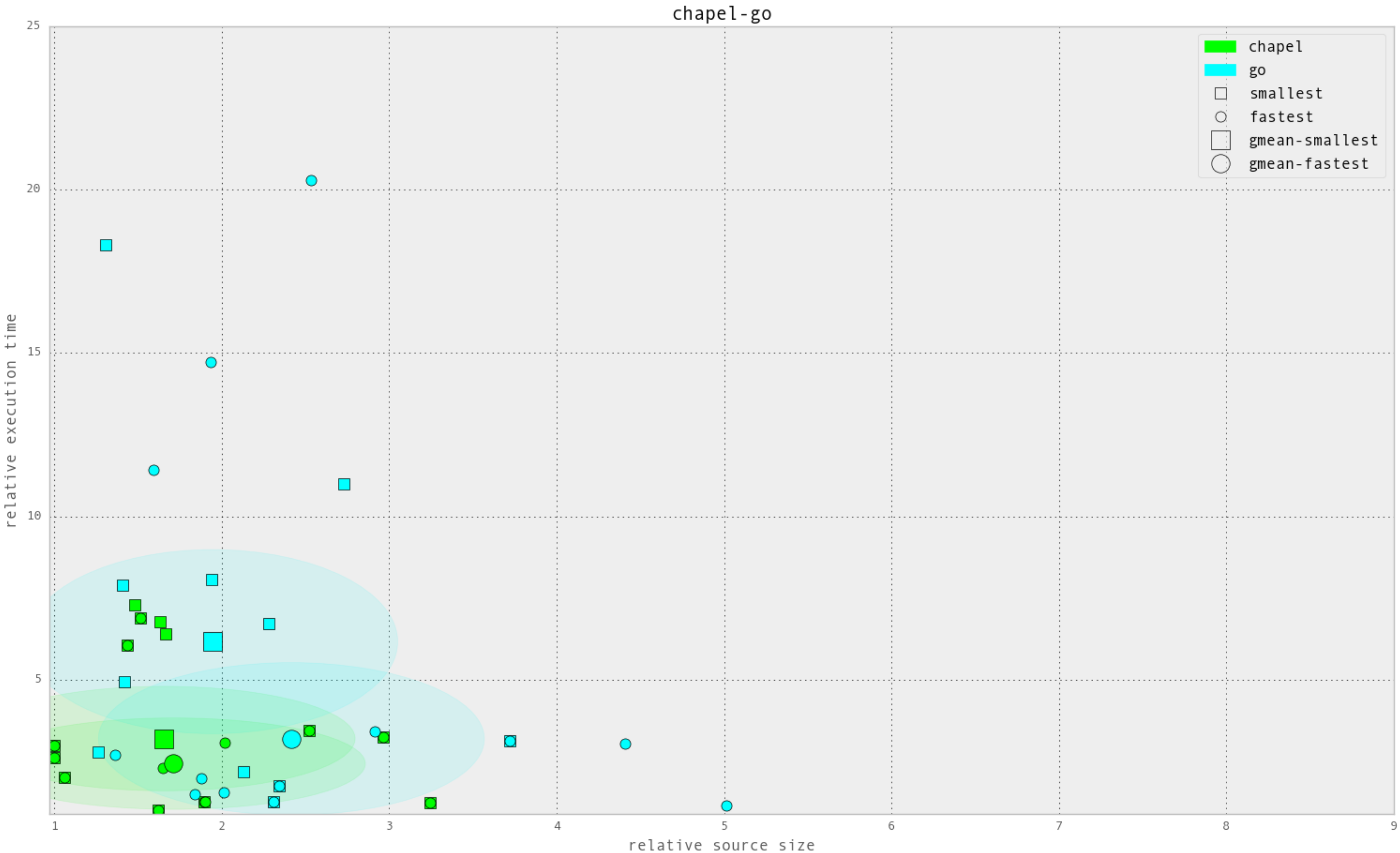
Chapel vs. Fortran



Chapel vs. Fortran (zoomed out)

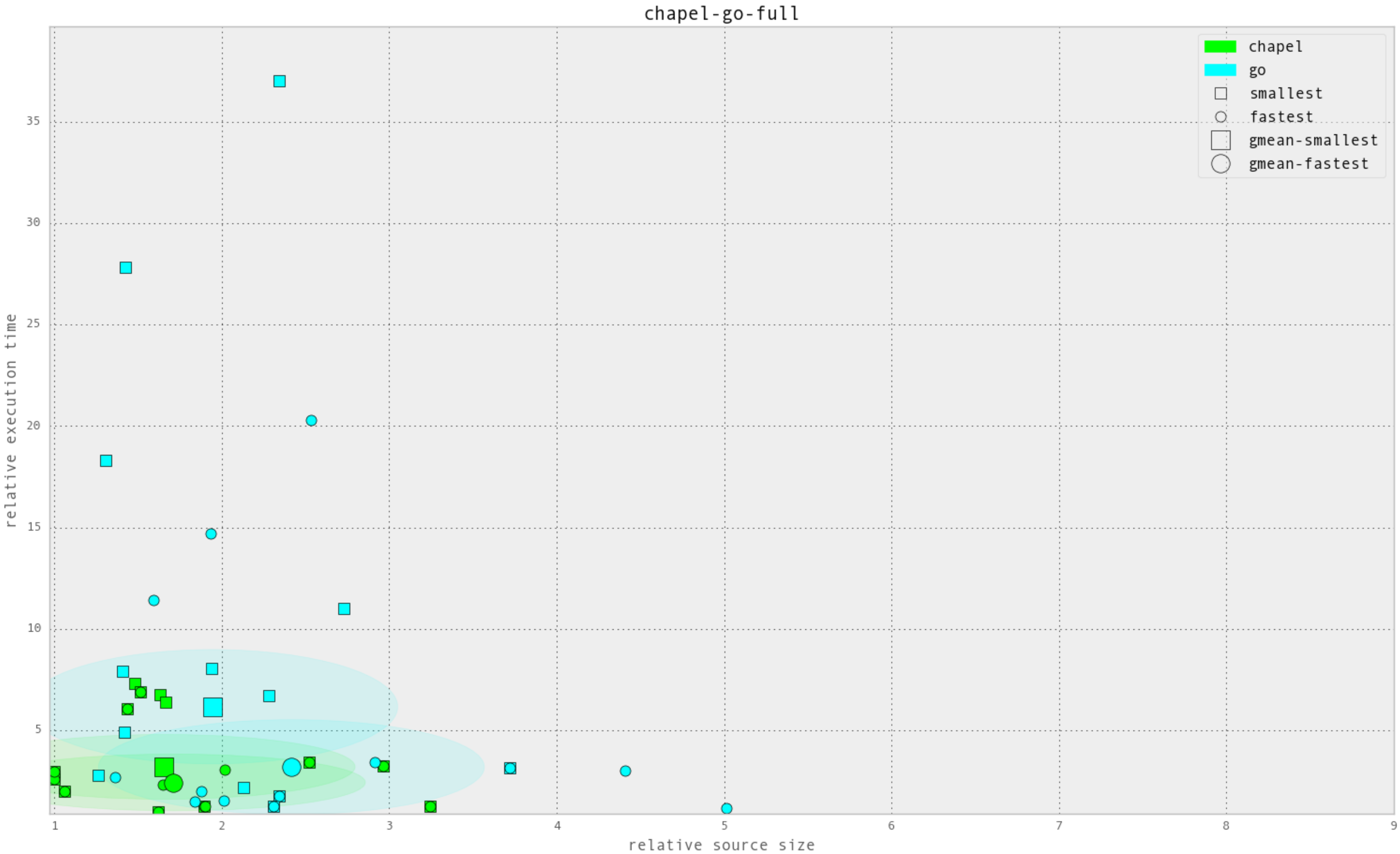


Chapel vs. Go

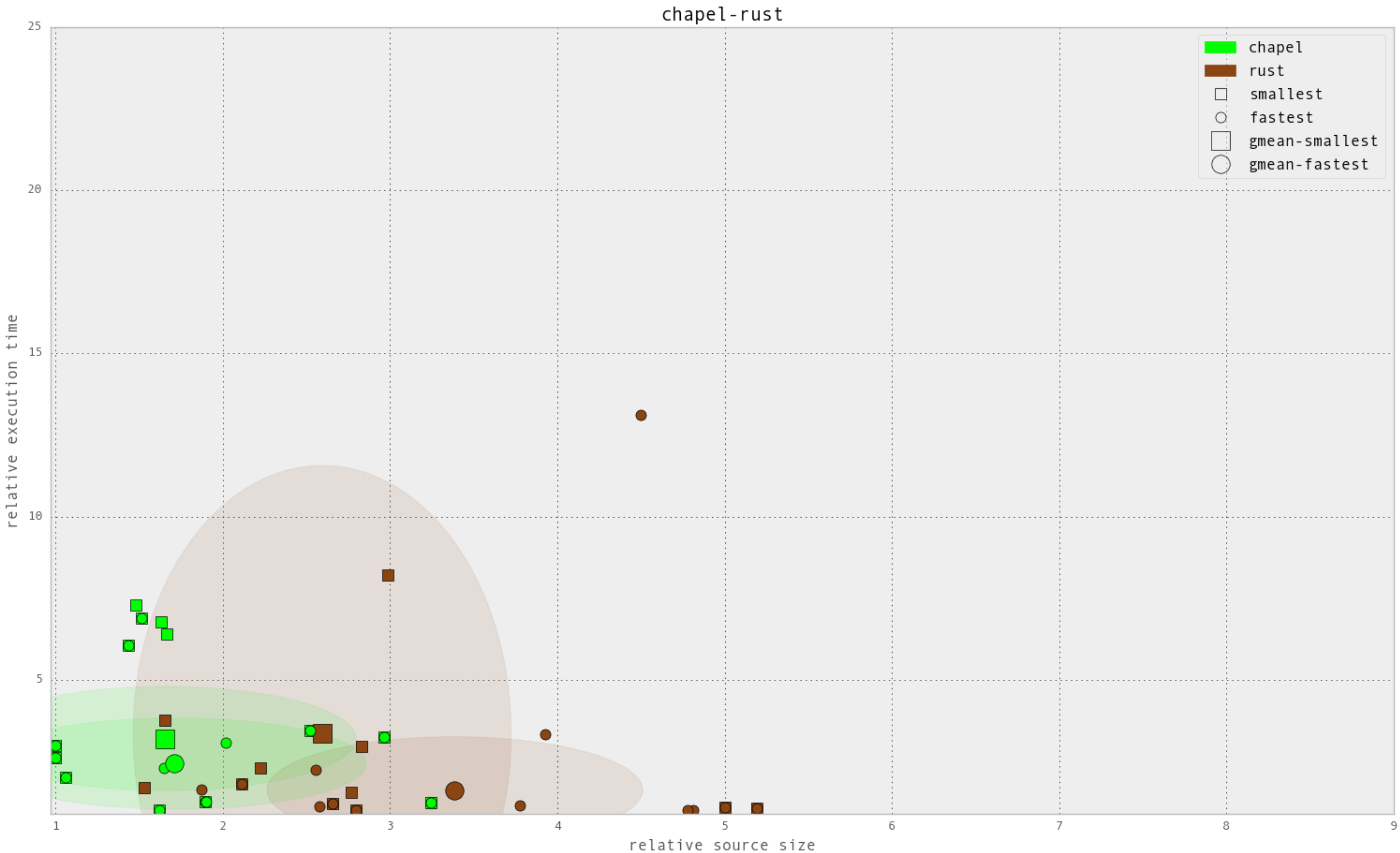


chapel-go
chapel
go
smallest
fastest
gmean-smallest
gmean-fastest
relative execution time
relative source size
25
20
15
10
5
1
2
3
4
5
6
7
8
9

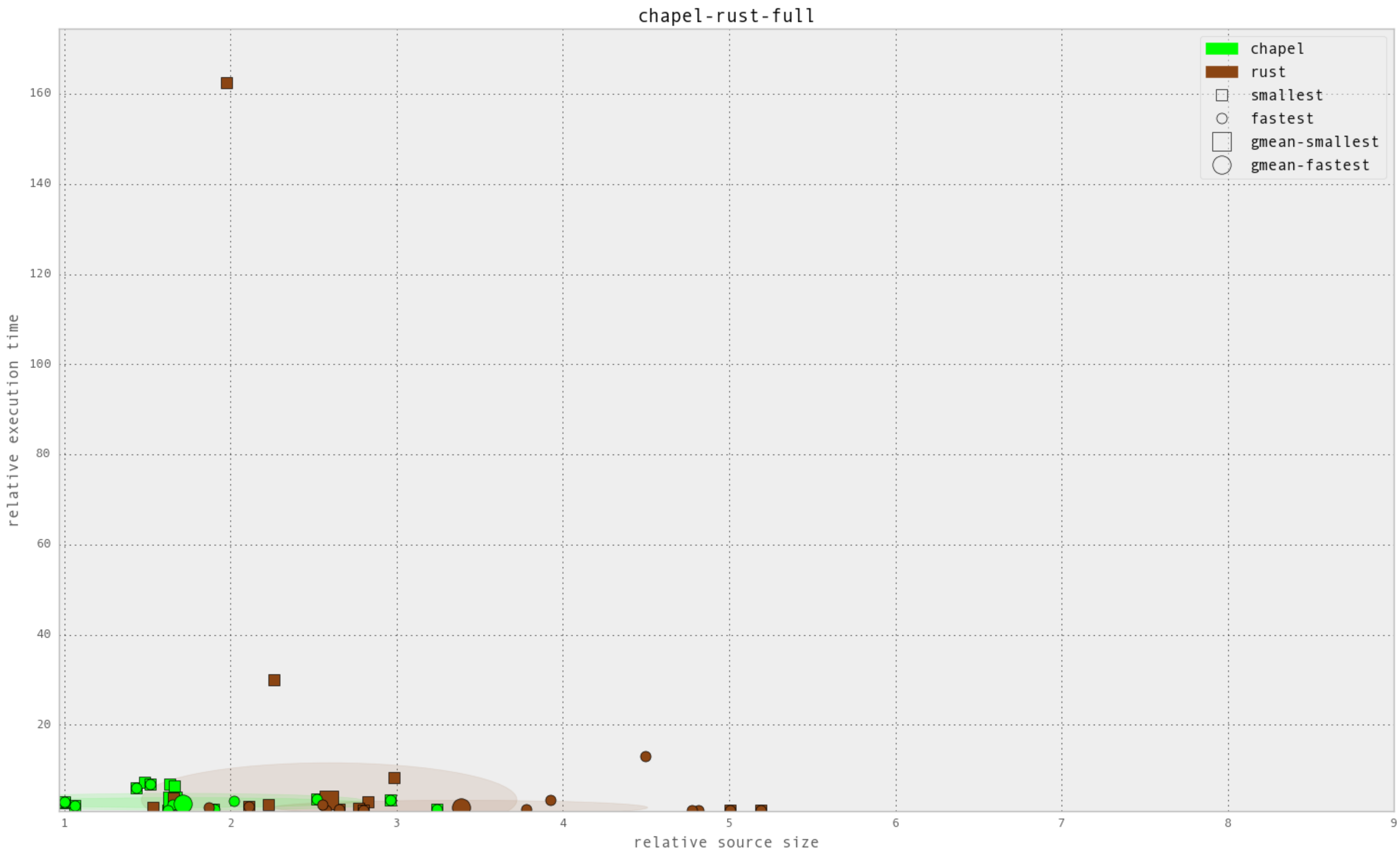
Chapel vs. Go (zoomed out)



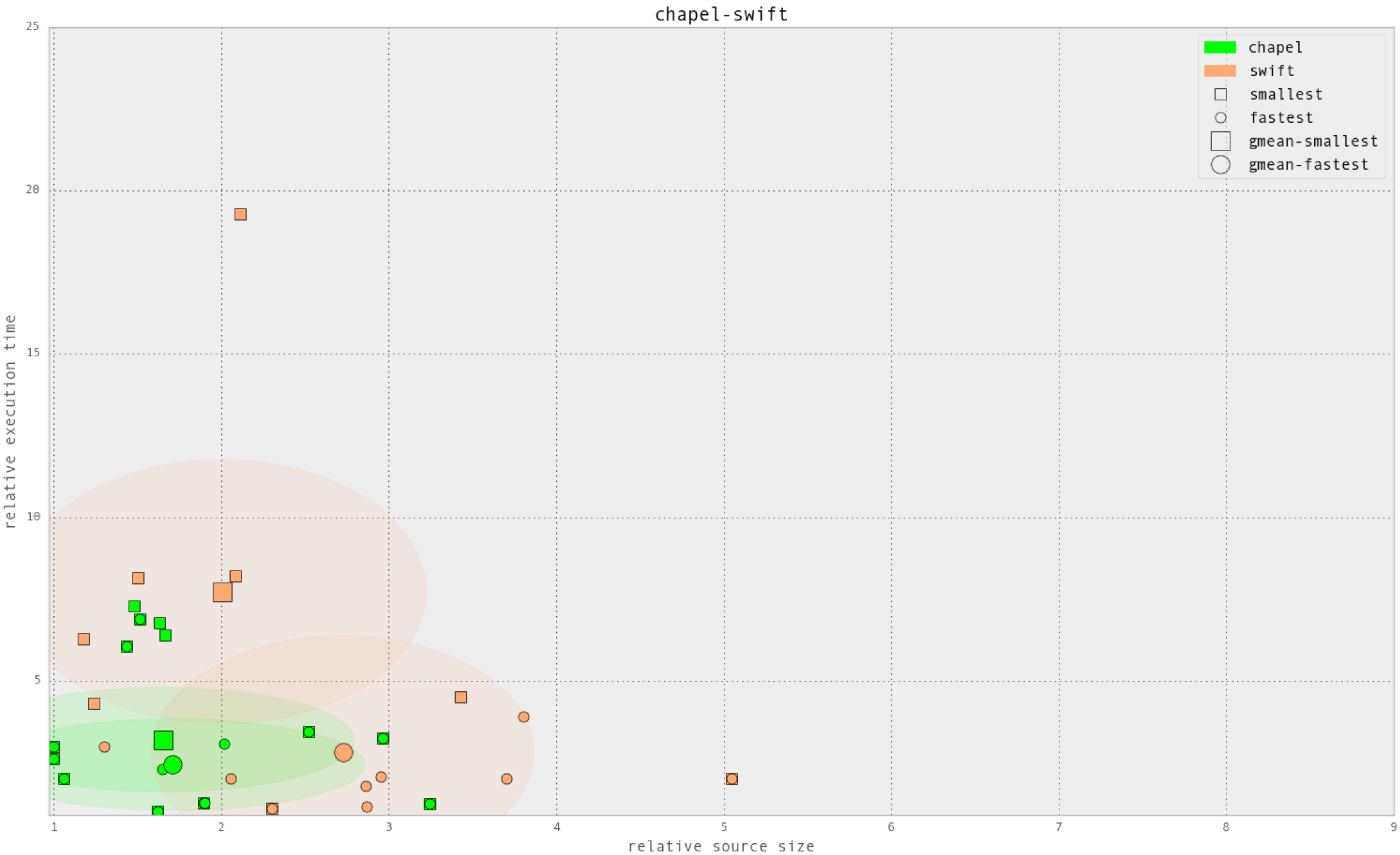
Chapel vs. Rust



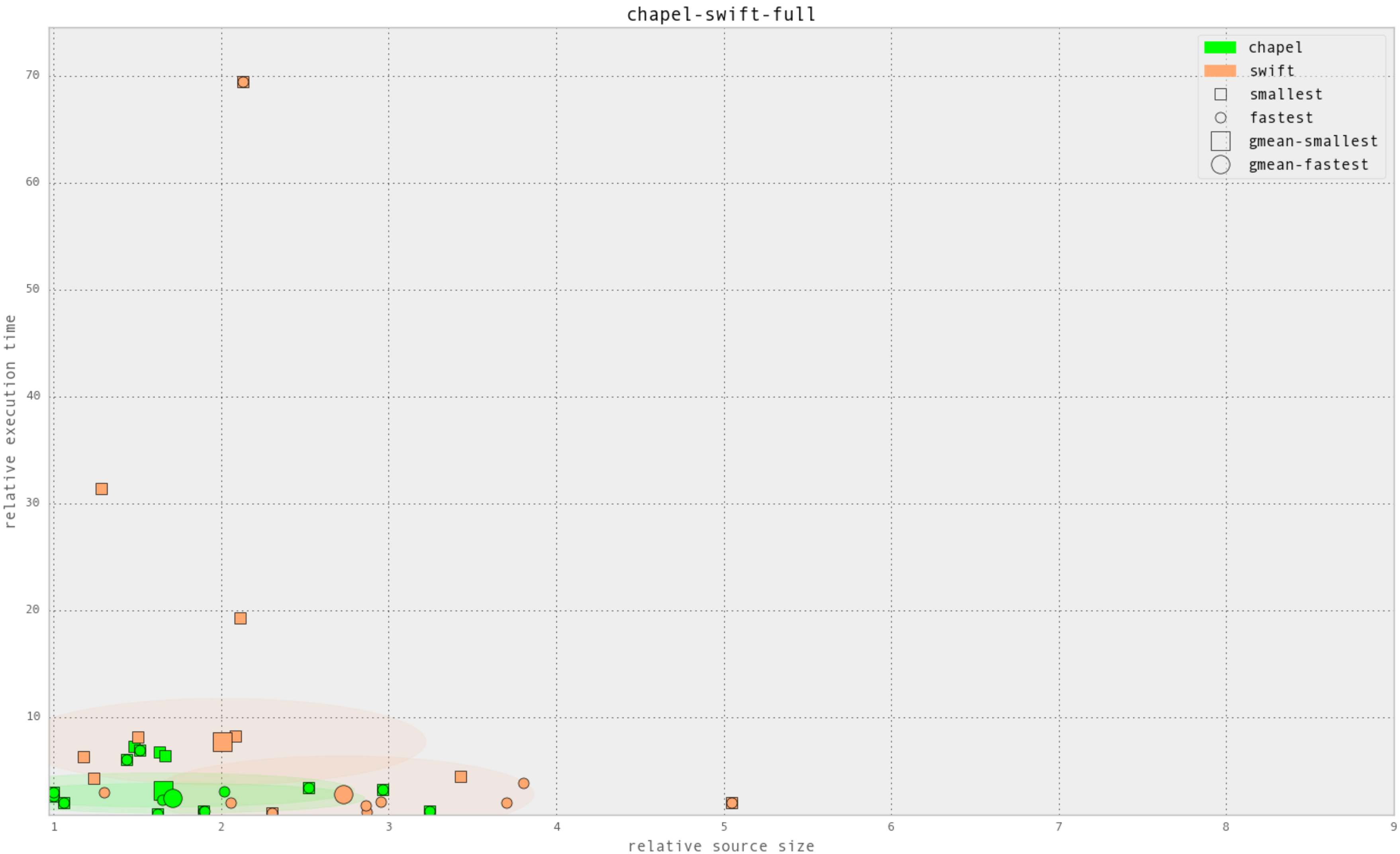
Chapel vs. Rust (zoomed out)



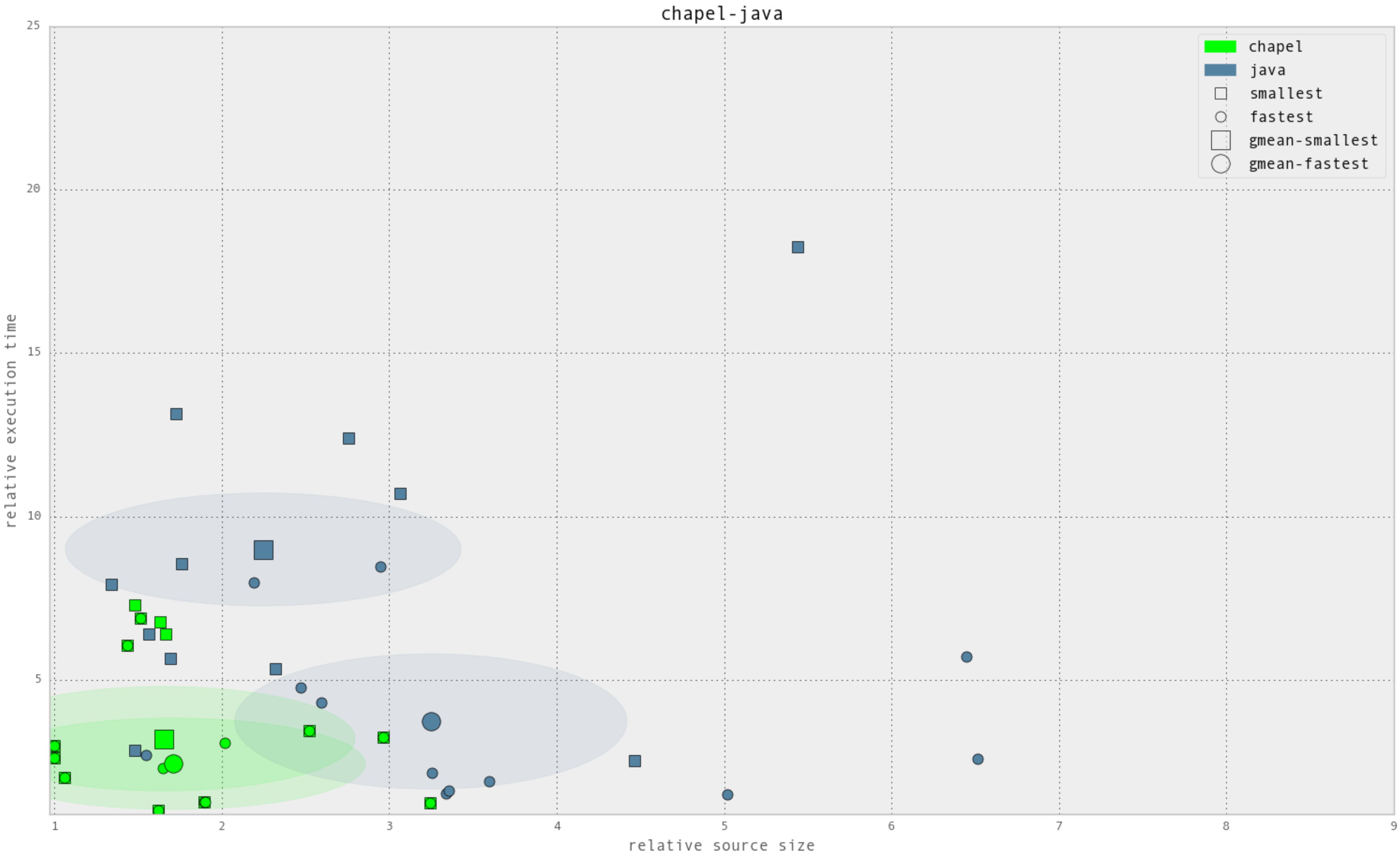
Chapel vs. Swift



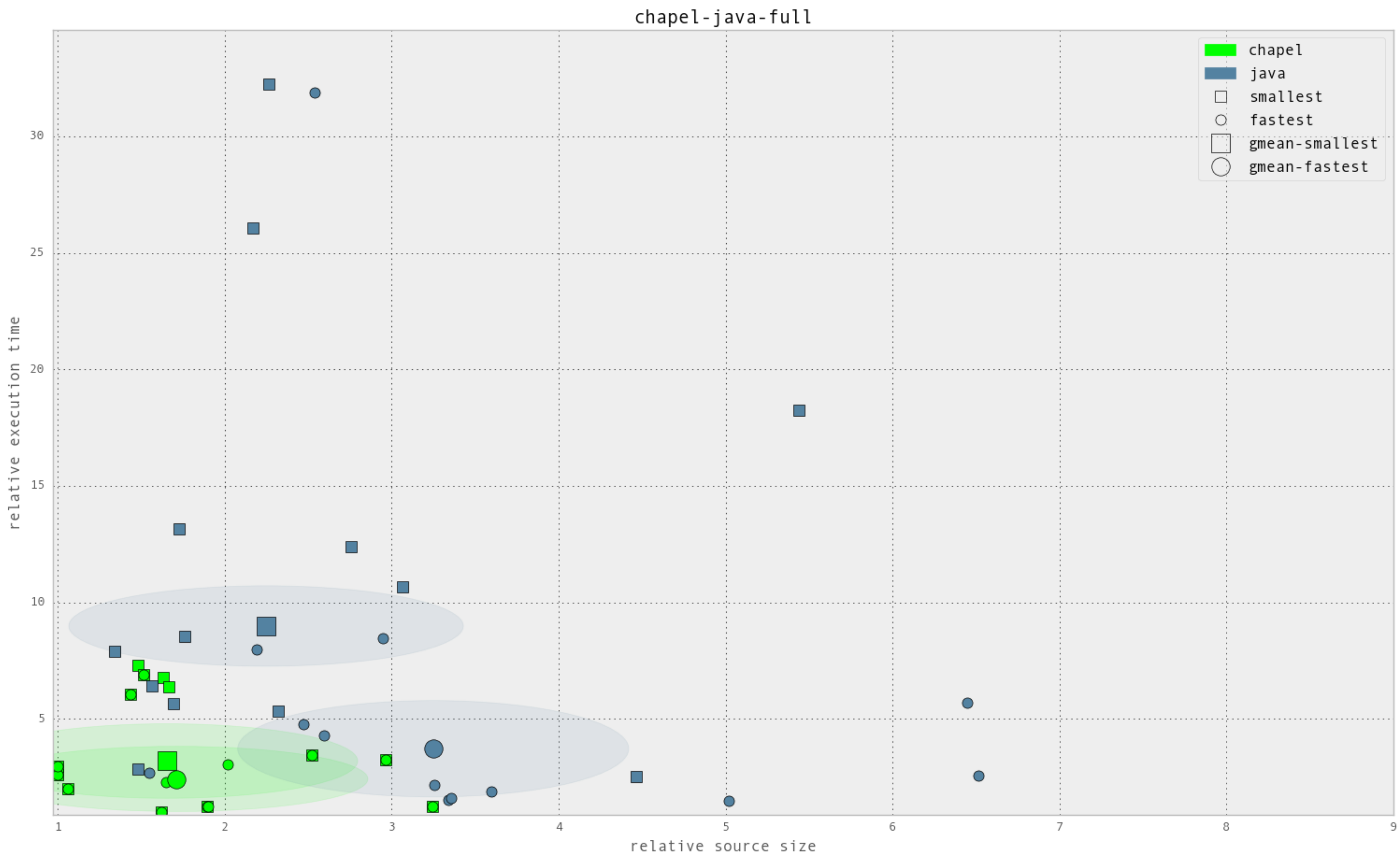
Chapel vs. Swift (zoomed out)



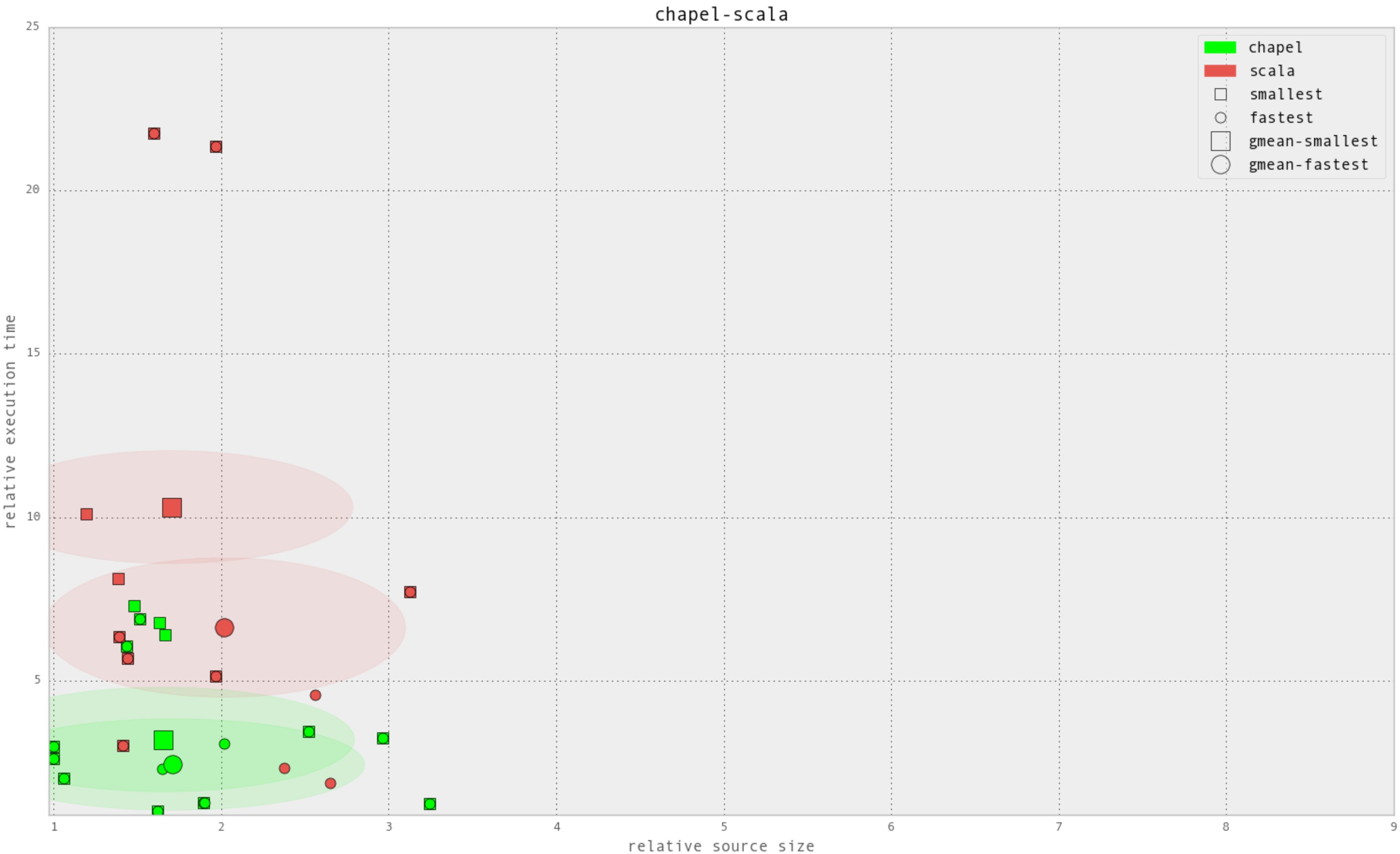
Chapel vs. Java



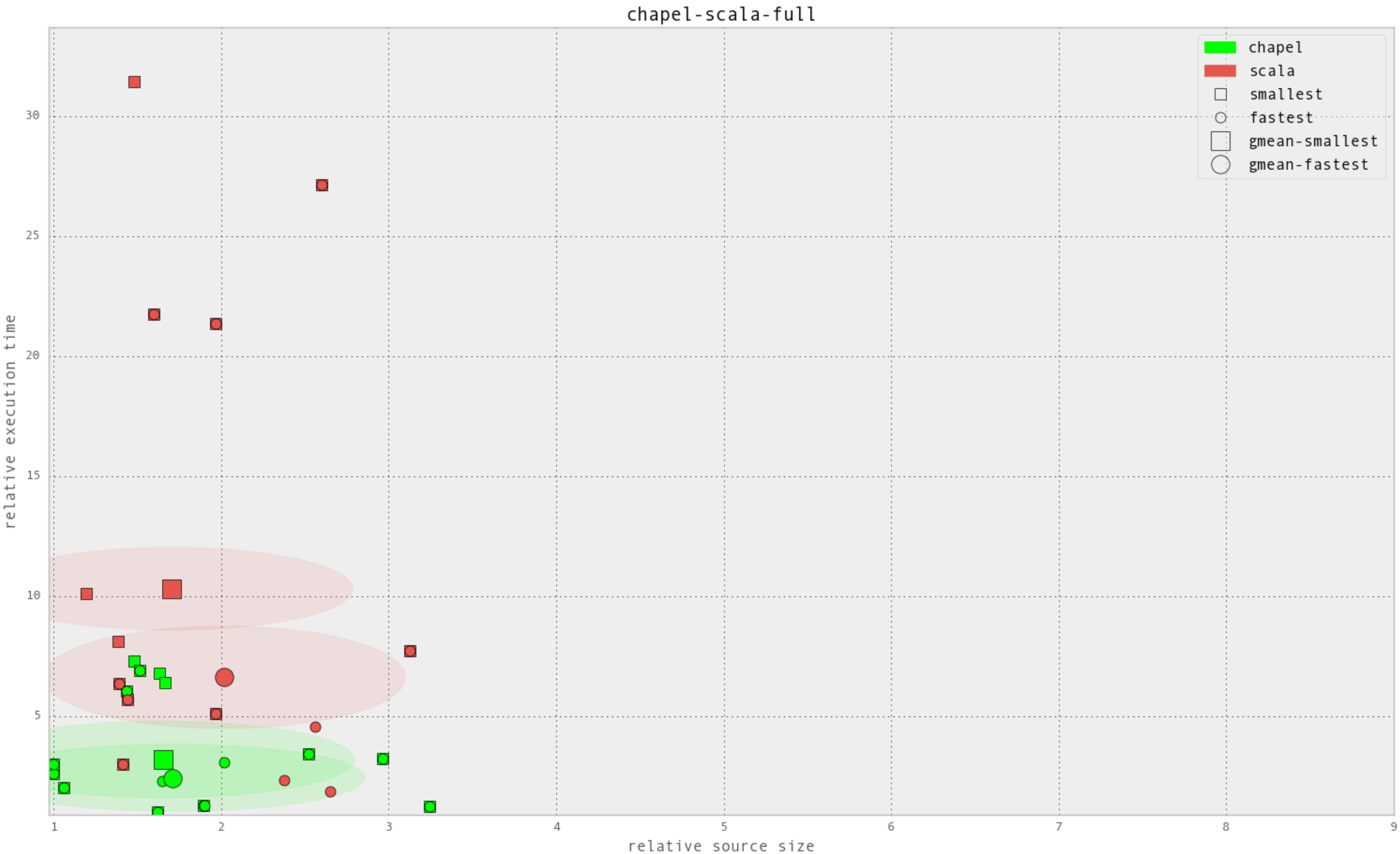
Chapel vs. Java (zoomed out)



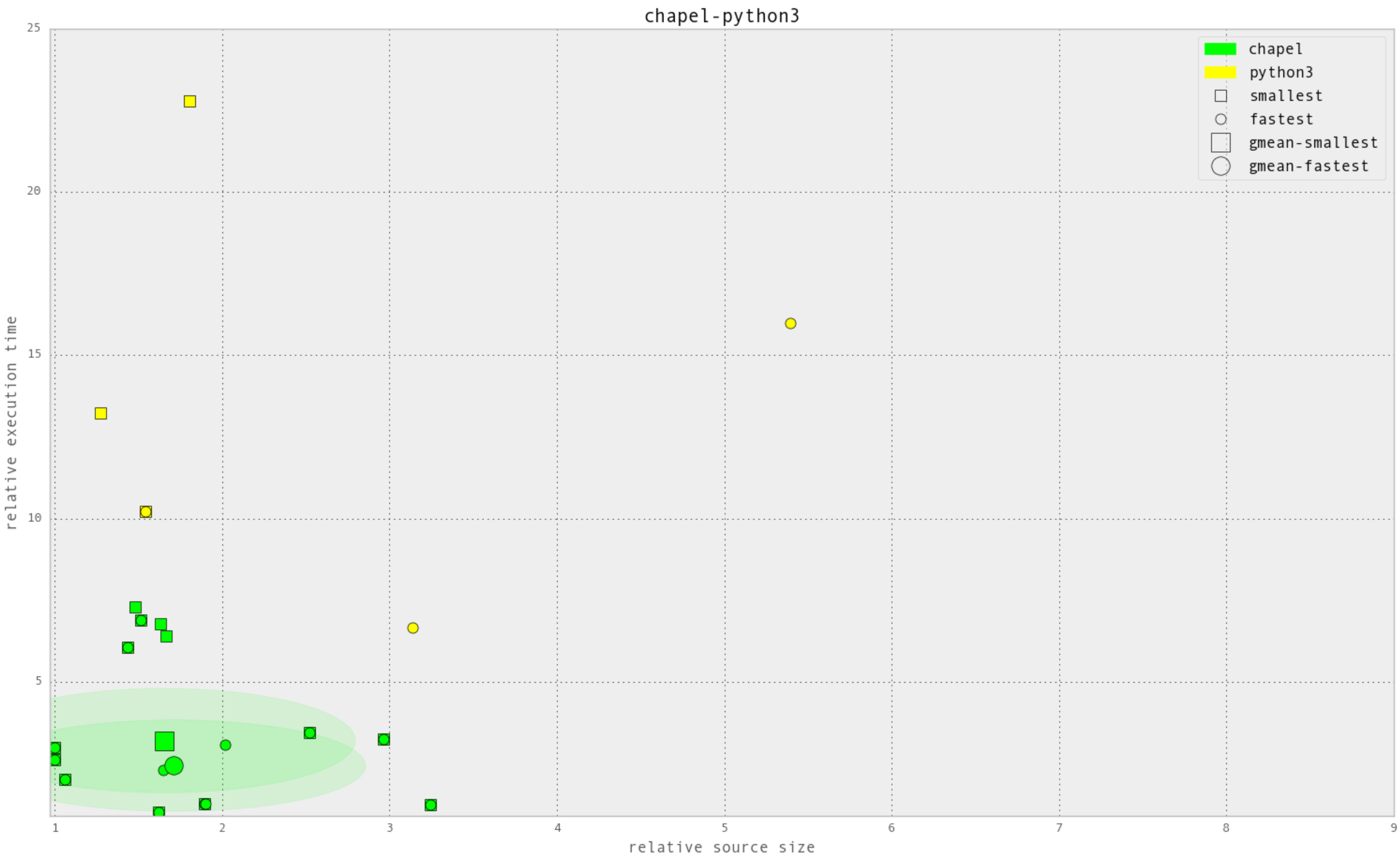
Chapel vs. Scala



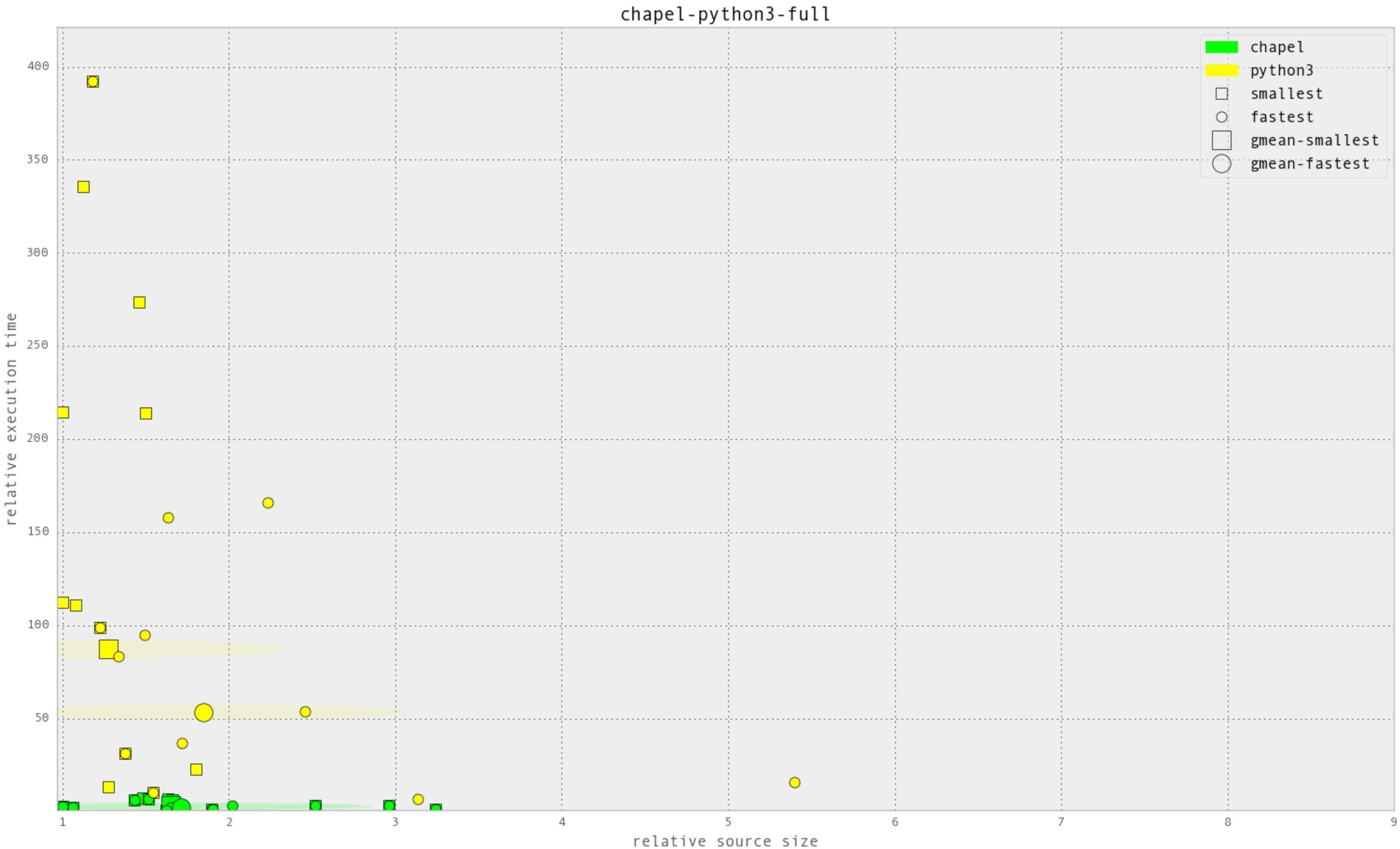
Chapel vs. Scala (zoomed out)



Chapel vs. Python



Chapel vs. Python (zoomed out)



Legal Disclaimer

Information in this document is provided in connection with Cray Inc. products. No license, express or implied, to any intellectual property rights is granted by this document.

Cray Inc. may make changes to specifications and product descriptions at any time, without notice.

All products, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

Cray hardware and software products may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Cray uses codenames internally to identify products that are in development and not yet publically announced for release. Customers and other third parties are not authorized by Cray Inc. to use codenames in advertising, promotion or marketing and any use of Cray Inc. internal codenames is at the sole risk of the user.

Performance tests and ratings are measured using specific systems and/or components and reflect the approximate performance of Cray Inc. products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.

The following are trademarks of Cray Inc. and are registered in the United States and other countries: CRAY and design, SONEXION, and URIKA. The following are trademarks of Cray Inc.: ACE, APPRENTICE2, CHAPEL, CLUSTER CONNECT, CRAYPAT, CRAYPORT, ECOPHLEX, LIBSCI, NODEKARE, THREADSTORM. The following system family marks, and associated model number marks, are trademarks of Cray Inc.: CS, CX, XC, XE, XK, XMT, and XT. The registered trademark LINUX is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Other trademarks used in this document are the property of their respective owners.

CRAY CHAPEL

CRAY

THE SUPERCOMPUTER COMPANY