Benchmark Improvements

Chapel Team, Cray Inc.

Chapel version 1.14

October 6, 2016

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 Improvements and Status
- Computer Language Benchmark Game (CLBG)
- Other Benchmark Improvements

LCALS Improvements and Status

LCALS: Background

- **LCALS: Livermore Compiler Analysis Loop Suite**
  - Loop kernels designed to measure compiler performance
  - Developed by LLNL
  - https://codesign.llnl.gov/LCALS.php

LCALS Code
Richard D. Hornung
LCALS version 1.0
LLNL-CODE-638939
2013

- **Three loop subsets (30 kernels total)**
  - Subset A: Loops representative of application codes
  - Subset B: Simple, basic loops
  - Subset C: Loops extracted from “Livermore Loops coded in C”

- **Each kernel is run for three sizes (Short, Medium, Long)**

- **Each kernel is implemented in a number of “variants”**
  - RAW (traditional C usage), OpenMP, C++ template-based, etc.

LCALS: This Effort

- **All kernels/variants/sizes performance tested nightly**
- **Benchmark run on a 24-core Cray XC30 node**
  - Compare serial performance vs. C++
  - Compare parallel performance vs. C++ with OpenMP
  - Compiled with g++ 6.2.0
- **Working toward matching or beating g++ with OpenMP**
  - Serial variant is on par with g++ for almost all kernels
  - Parallel variant is off by 2x-3x in most cases for Long problem size
    - Best case: PIC_2D nearly 40% faster
    - Worst case: COUPLE over 7x slower
  - Short problem size parallel variant is much faster than C++ version
    - --dataParMinGranularity=1000 option limits parallelism to reasonable level

Compilation/Execution commands:

```
chpl --fast --no-ieee-float
lcals-chpl --dataParMinGranularity=1000
```

```
g++ -Ofast -fopenmp
lcals.exe
```

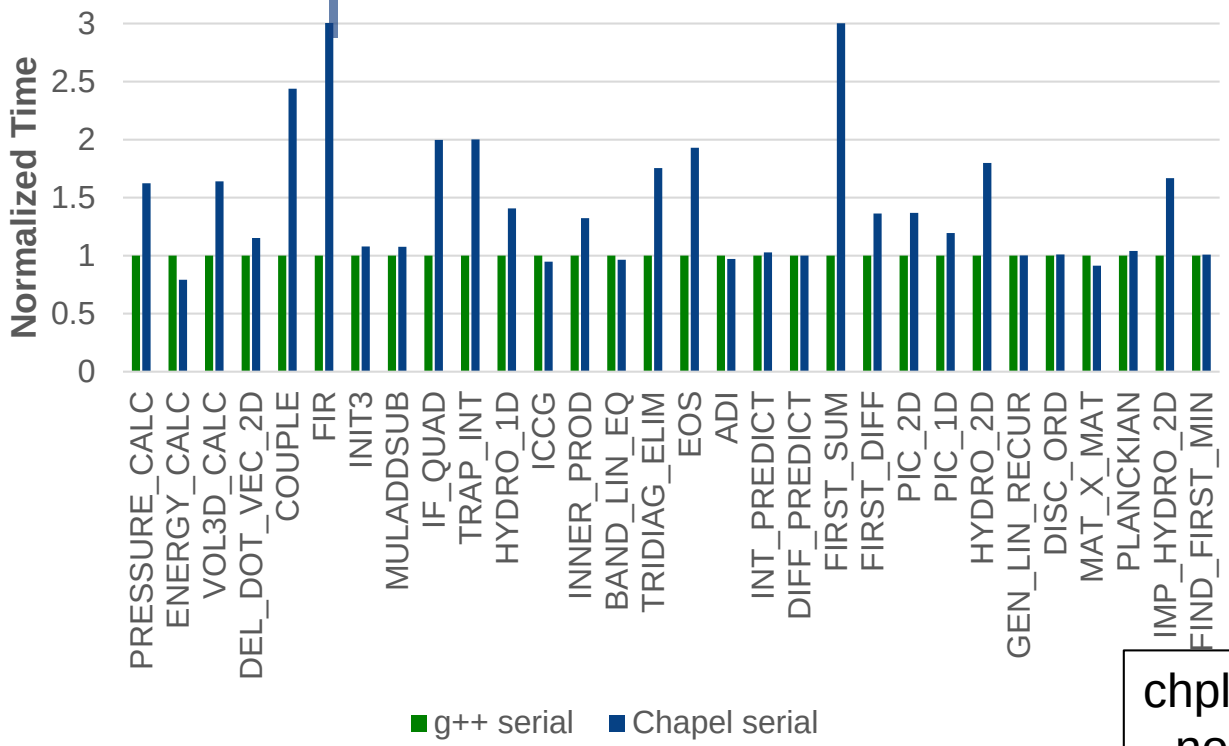
LCALS: Improvements since 1.13

- Removed -sassertNoSlicing config param setting
  - Subsumed by the (default-on) compiler optimization --optimize-array-indexing
- Set dataParMinGranularity to 1000
  - Small and medium loop sizes created too many tasks without this setting
- Replaced several loops over arrays with whole array assignments
- Added an output mode that is easier to parse by scripts
- Enabled performance tracking for all variants, kernels, and loop sizes
  - Now test and track the performance of all 156 variants/kernels/sizes daily
  - Graphs are online at the Chapel Performance Overview website
- Significantly increased PIC_2D performance with improvements to atomics

LCALS Status: Serial Performance v1.13.0

Long problem size

(Similar results for medium and short problem sizes)



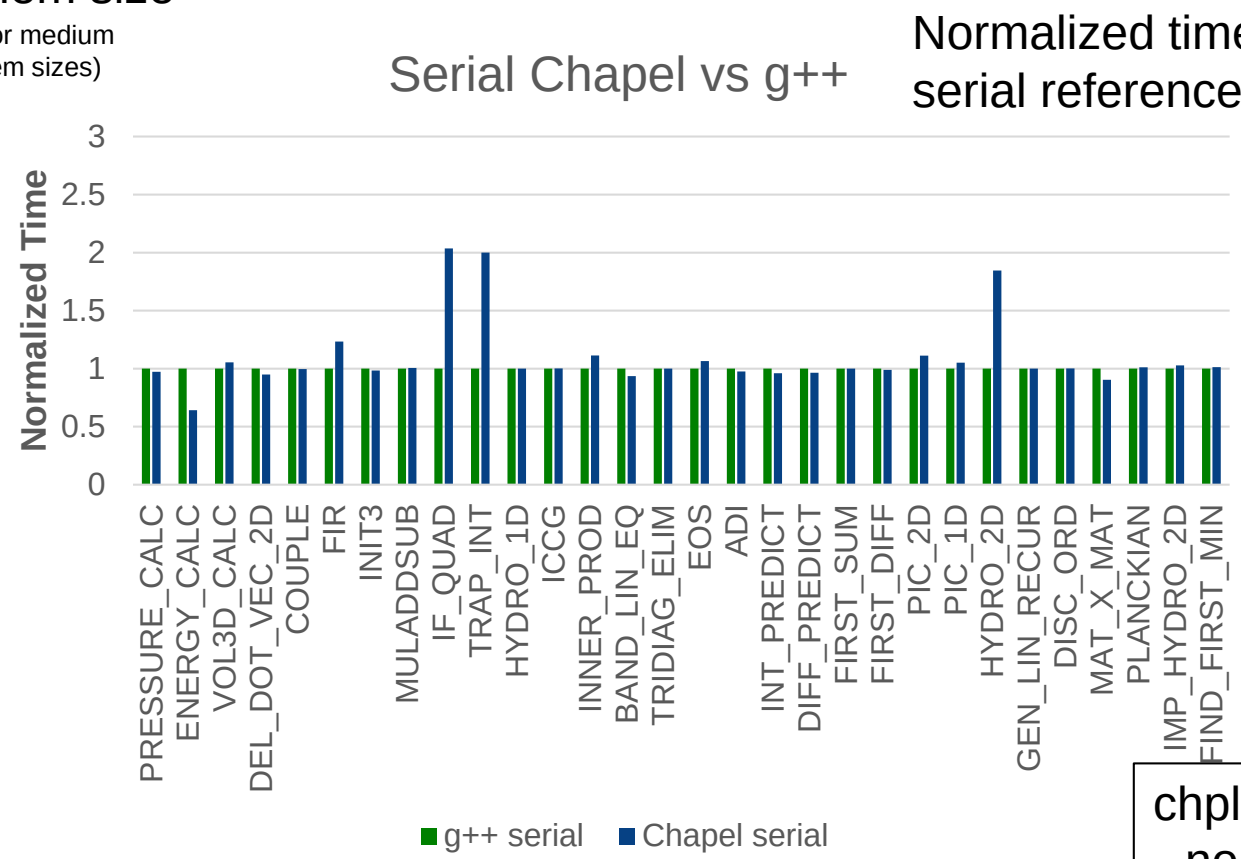
chpl --fast --no-ieee-float

g++ -Ofast -fopenmp

LCALS Status: Serial Performance v1.14.0

Long problem size

(Similar results for medium and short problem sizes)



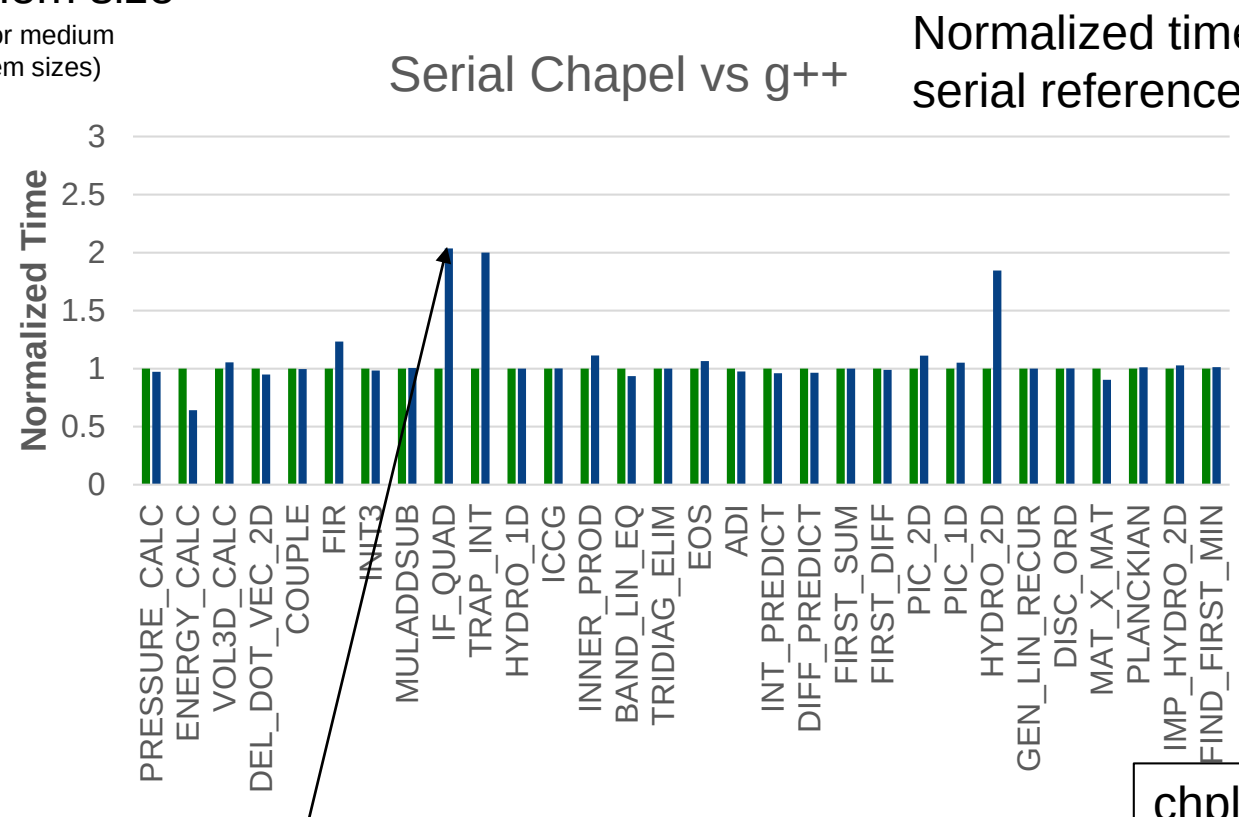
chpl --fast --no-ieee-float

g++ -Ofast -fopenmp

LCALS Status: Serial Performance v1.14.0

Long problem size

(Similar results for medium and short problem sizes)



`--replace-array-accesses-with-ref-temps` optimization fixes IF_QUAD performance

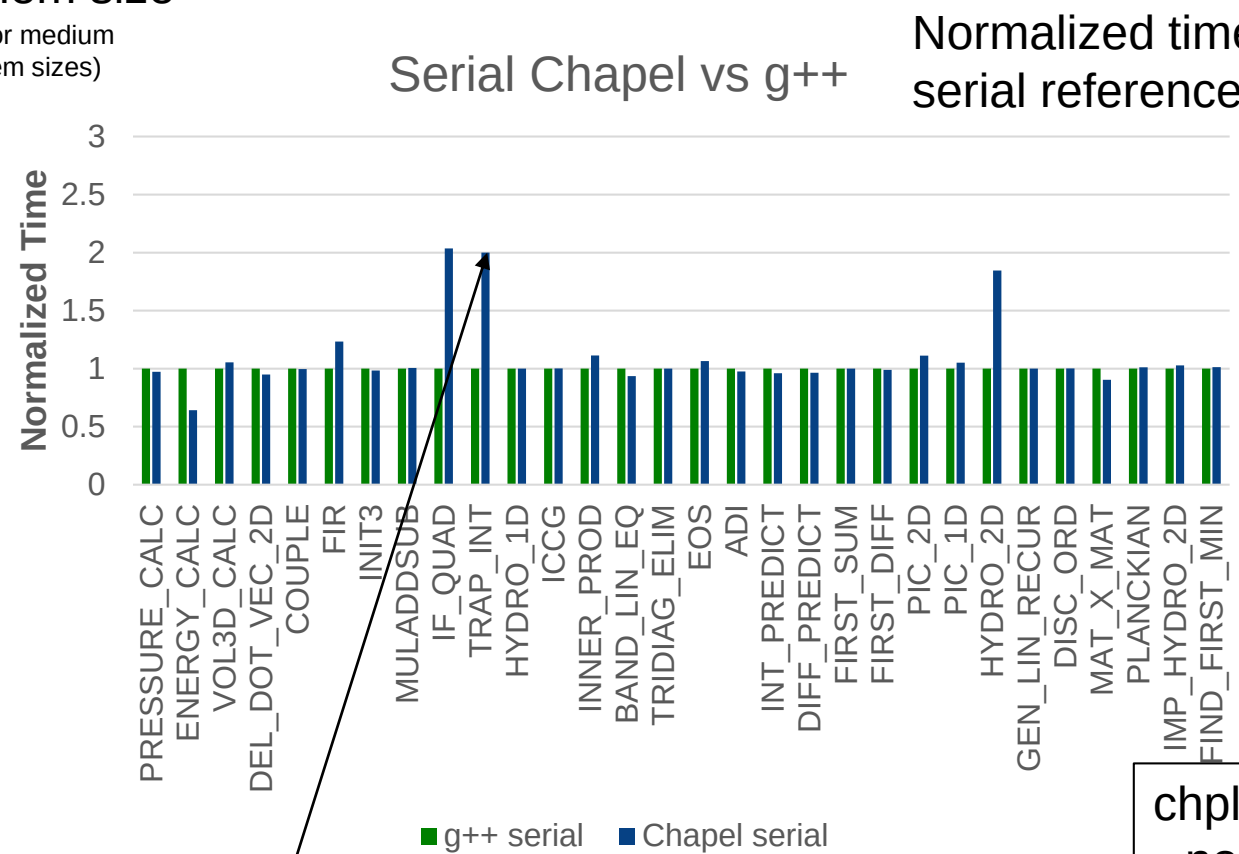
chpl --fast --no-ieee-float

g++ -Ofast -fopenmp

LCALS Status: Serial Performance v1.14.0

Long problem size

(Similar results for medium and short problem sizes)



Mis-translation: The main iteration range of TRAP_INT should use `int(32)` instead of `int(64)`. Fixed on master after 1.14.

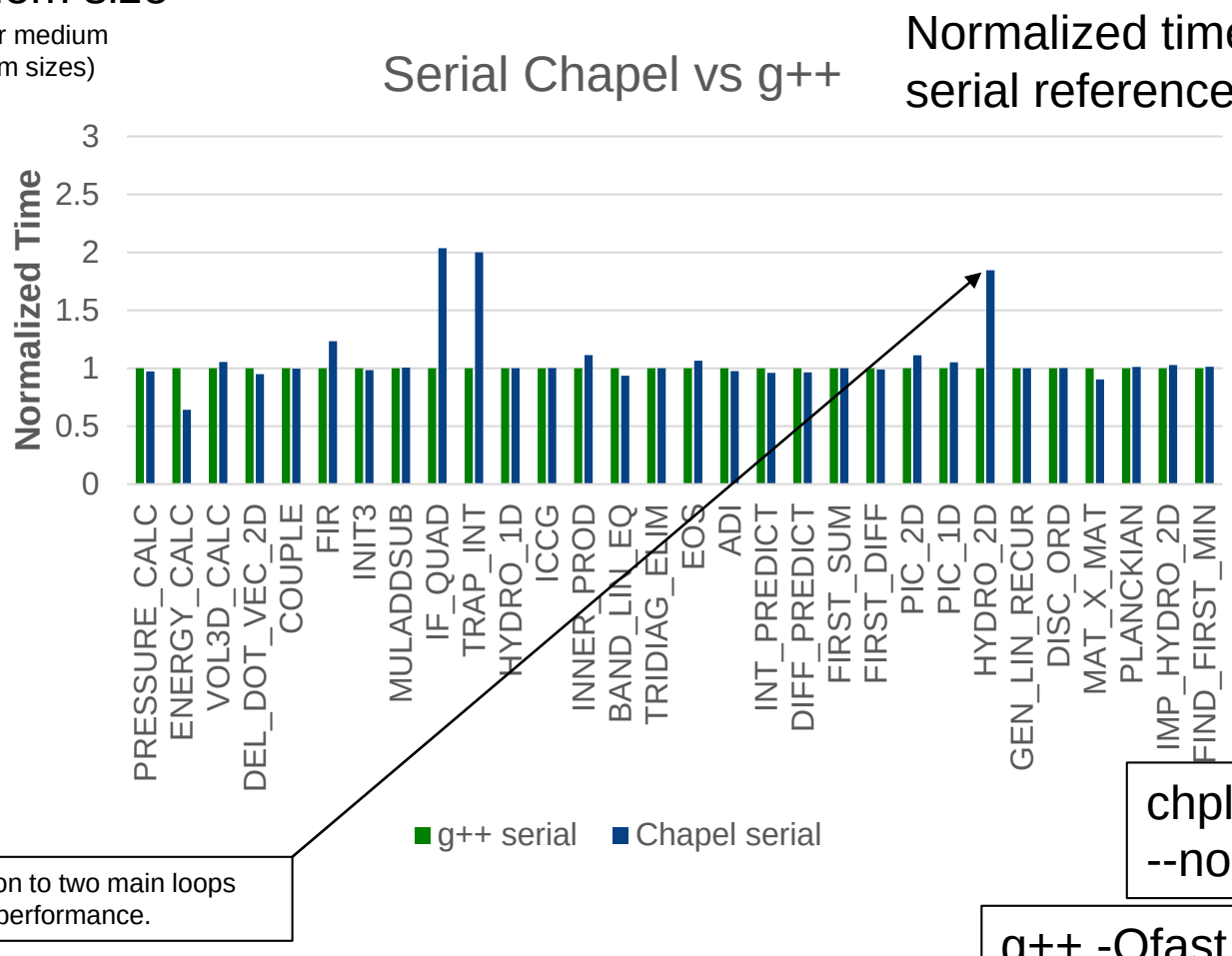
chpl --fast --no-ieee-float

g++ -Ofast -fopenmp

LCALS Status: Serial Performance v1.14.0

Long problem size

(Similar results for medium and short problem sizes)



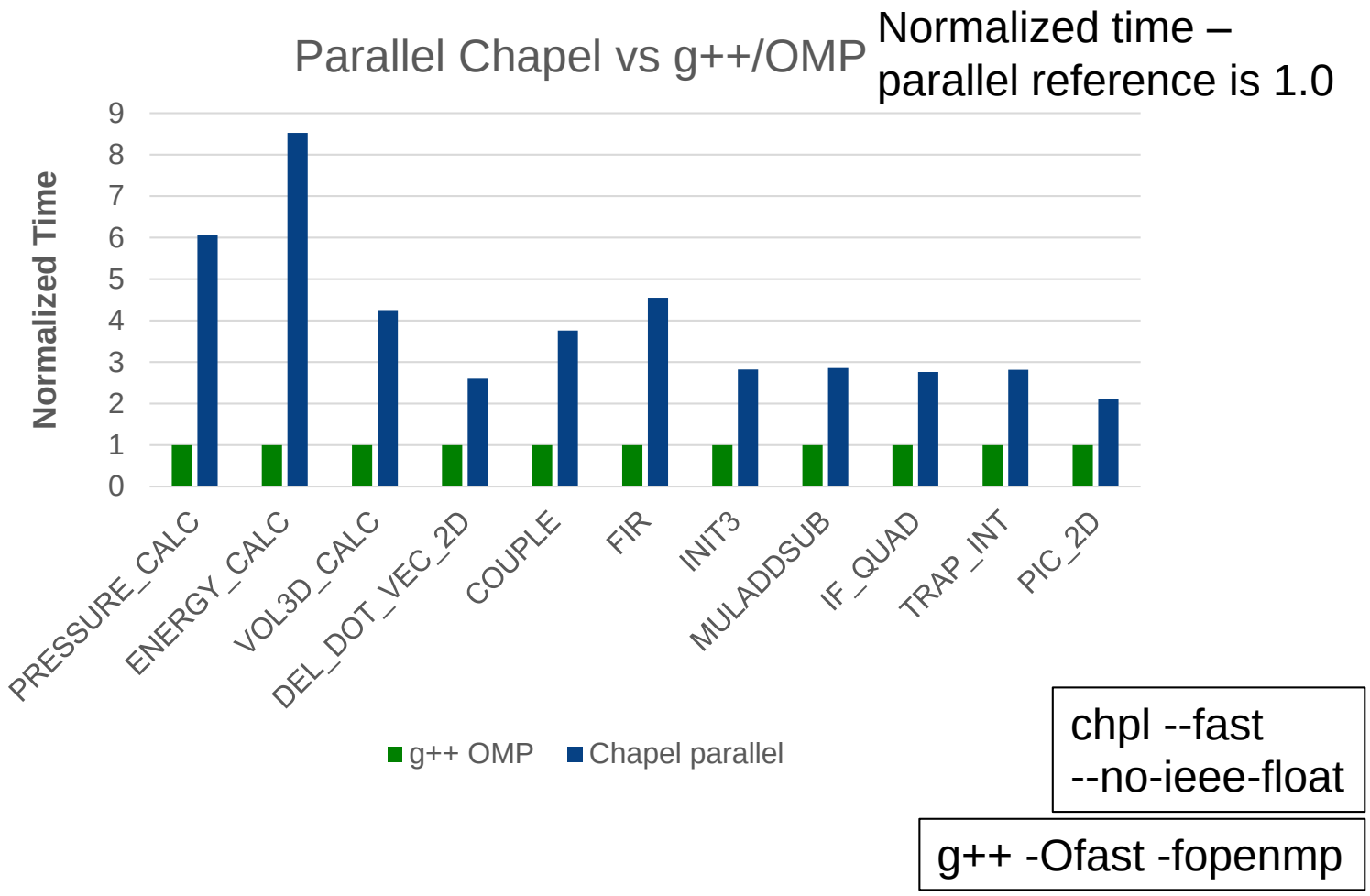
Applying loop fission to two main loops fixes HYDRO_2D performance.

chpl --fast --no-ieee-float

g++ -Ofast -fopenmp

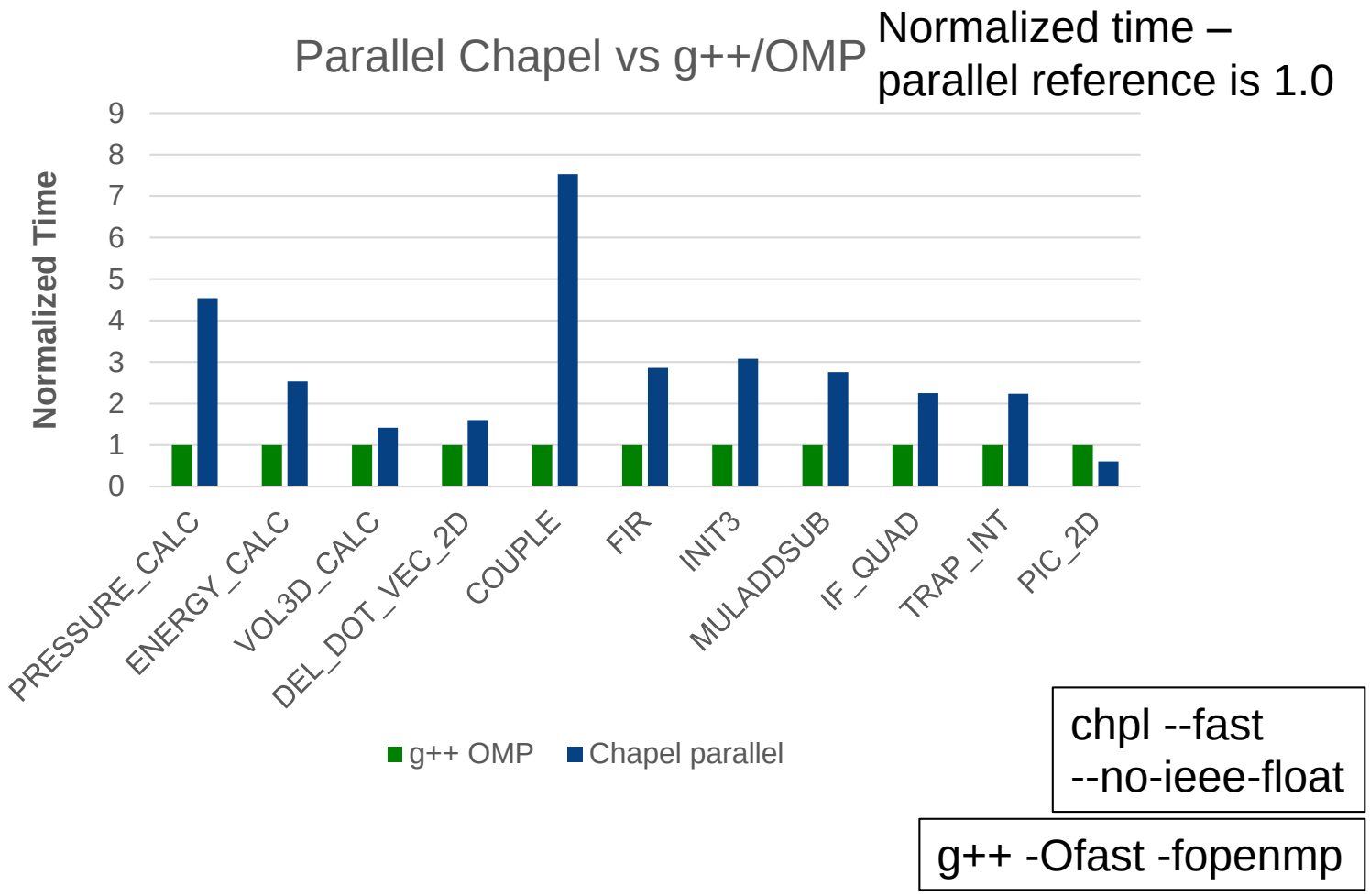
LCALS Status: Parallel Performance v1.13.0

Long problem size



LCALS Status: Parallel Performance v1.14.0

Long problem size

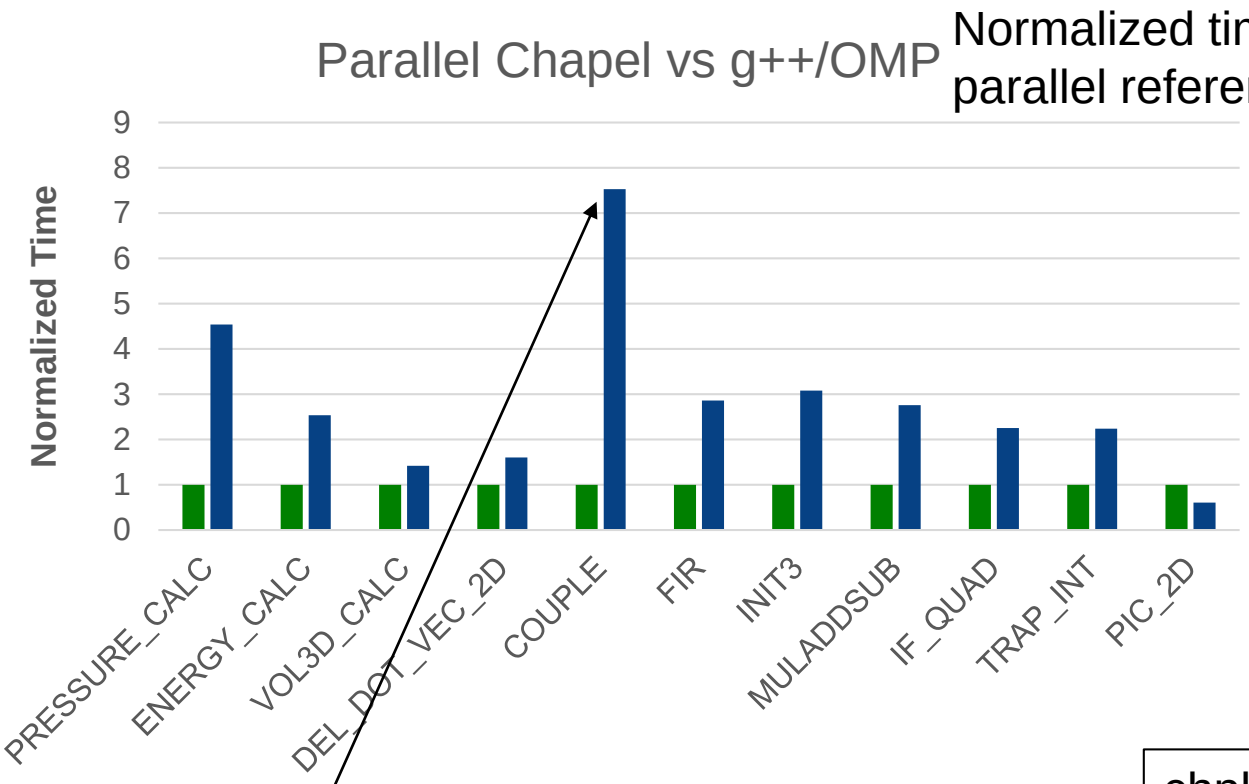


chpl --fast --no-ieee-float

g++ -Ofast -fopenmp

LCALS Status: Parallel Performance v1.14.0

Long problem size



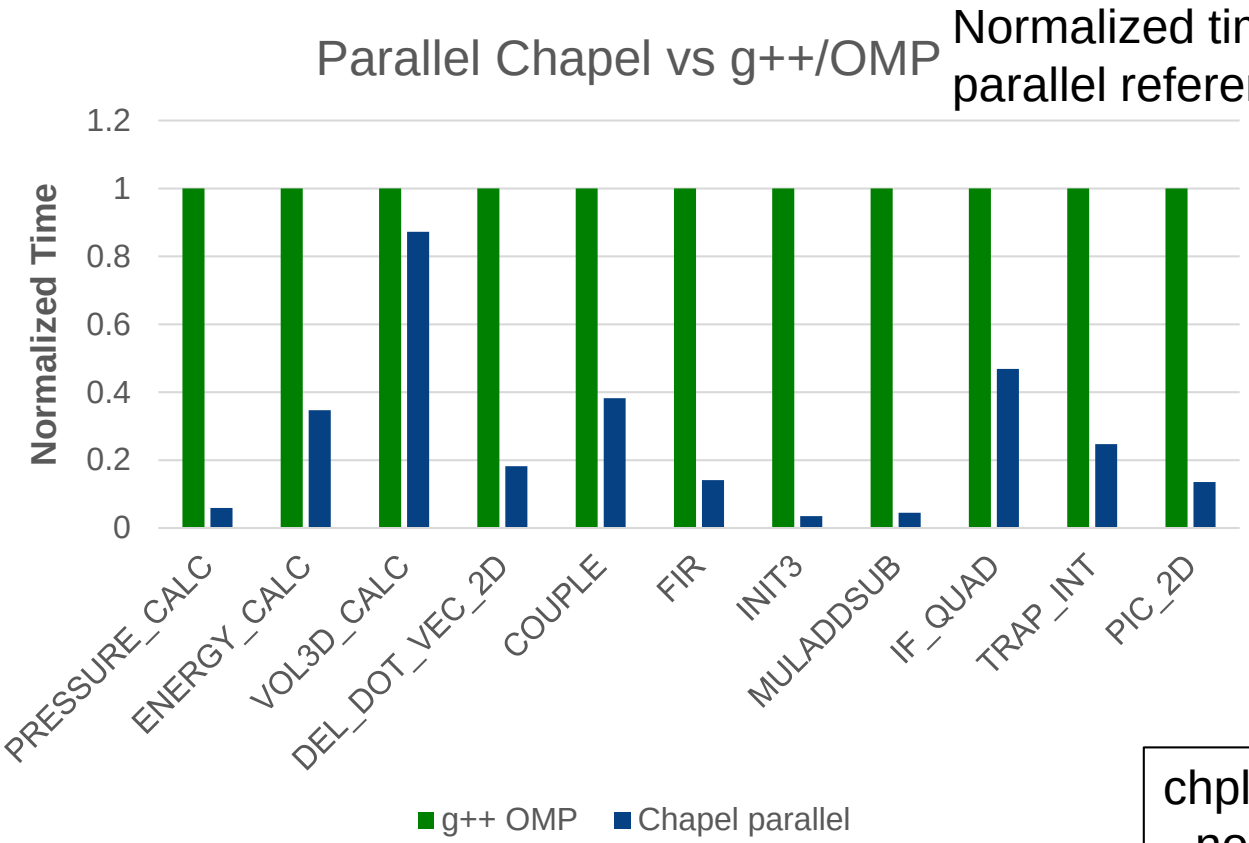
Setting `--dataParMinGranularity=1000` caused COUPLE to not use enough tasks

chpl --fast --no-ieee-float

g++ -Ofast -fopenmp

LCALS Status: Parallel Performance

Short problem size



chpl --fast --no-ieee-float

g++ -Ofast -fopenmp

LCALS: Next Steps

- **Continue optimization effort for parallel kernels**
  - Understand the remaining parallel performance gaps
  - Bring all parallel kernels in line with reference versions
  - Avoid need for minGranularity setting
- **Optimize the serial kernels that are still lagging**
  - IF_QUAD, HYDRO_2D still need some effort
- **Explore more elegant Chapel loop expressions**
  - Make further use of whole-array operations, array slicing, etc.

Computer Language Benchmark Game (CLBG)

CLBG: Background

CLBG: Website supporting cross-language comparisons

- based on 13 simple serial/shared-memory computations:
  - **binary-trees:** memory management stressor
  - **chameneos-redux:** tasking coordination via a shared resource
  - **fannkuch-redux:** compute permutations on a small array
  - **fasta, k-nucleotide, regex-dna, reverse-complement:** string manipulation
  - **mandelbrot:** compute the Mandelbrot set
  - **meteor:** solve a puzzle (program startup is the bottleneck for Chapel)
  - **n-body:** simulate the solar system's largest bodies (wants vectorization)
  - **pidigits:** compute pi (wants GMP or equivalent)
  - **spectral-norm:** compute matrix-vector operations
  - **thread-ring:** pass token between tasks as quickly as possible
- must follow prescribed algorithm (except meteor, which is free-form)
- website's summaries don't include three benchmarks:
  - chameneos-redux, thread-ring, meteor
  - the first two are parallel, which makes them of interest to us
  - program startup time is our bottleneck for the third, so we include it as well

CLBG: Website

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

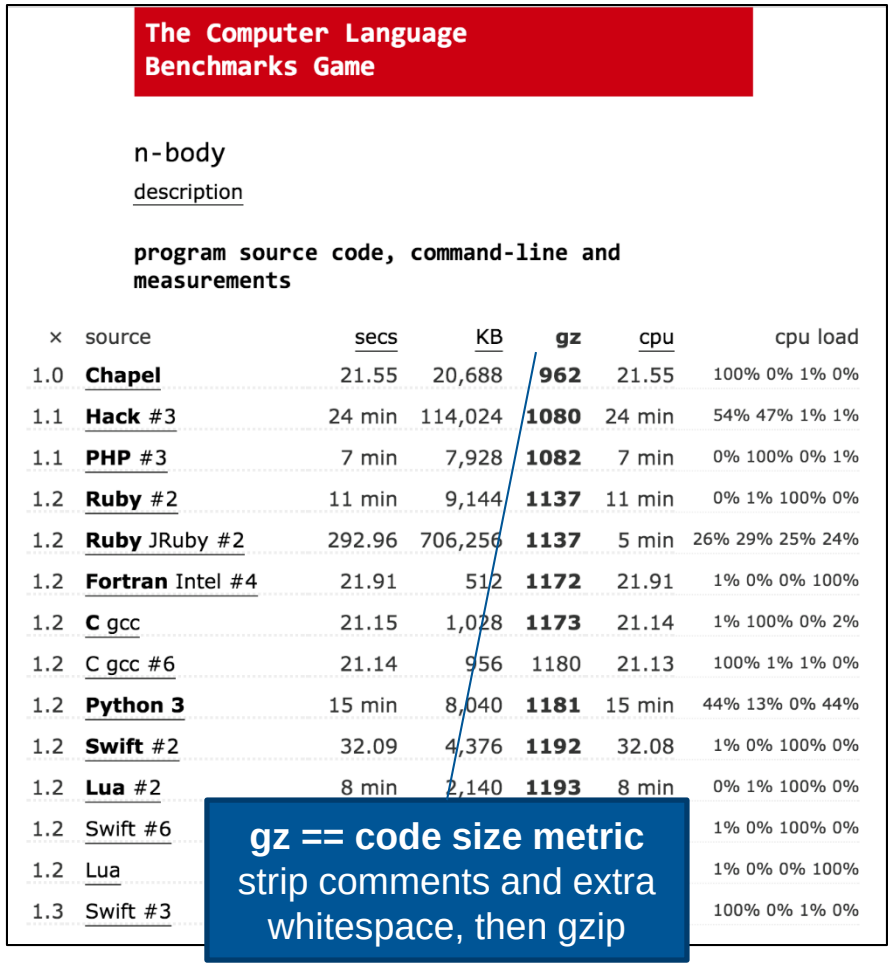
The Computer Language Benchmarks Game

n-body

description

program source code, command-line and measurements

×	source	secs	KB	gz	cpu	cpu load
1.0	**C++** g++ #3	**9.30**	1,712	1763	9.29	100% 1% 1% 0%
1.0	C++ g++ #8	9.37	1,152	1544	9.36	1% 0% 100% 1%
1.0	**C** gcc #4	**9.56**	1,000	1490	9.56	1% 100% 1% 1%
1.0	C++ g++ #7	9.65	940	1545	9.64	100% 1% 1% 1%
1.1	**Fortran** Intel #5	**9.79**	516	1659	9.78	1% 0% 1% 100%
1.2	**Ada** 2005 GNAT #2	**10.99**	1,952	2604	10.99	0% 1% 100% 1%
1.3	C++ g++ #5	11.76	1,728	1749	11.75	1% 100% 1% 0%
1.9	Ada 2005 GNAT #5	18.00	2,028	2436	18.00	0% 0% 100% 1%
2.1	C++ g++ #6	19.20	1,096	1668	19.19	100% 1% 0% 0%
2.1	C++ g++	19.37	1,056	1659	19.36	1% 100% 0% 0%
2.1	Fortran Intel #2	19.84	508	1496	19.83	1% 0% 1% 100%
2.1	Fortran Intel	19.96	512	1389	19.95	0% 1% 0% 100%
2.2	C++ g++ #4	20.20	684	1428	20.19	0% 0% 1% 100%
2.3	C gcc #3	20.97	952	1208	20.96	1% 1% 1% 100%



CLBG: Website

Can also compare languages pair-wise:

The Computer Language Benchmarks Game

Chapel programs versus Swift

all other Chapel programs & measurements

by benchmark task performance

regex-dna

source	secs	KB	gz	cpu	cpu load
Chapel	**9.35**	1,787,668	468	18.46	100% 15% 15% 69%
Swift	103.03	270,796	712	102.95	1% 1% 0% 100%

pidigits

source	secs	KB	gz	cpu	cpu load
Chapel	**1.60**	22,088	501	1.60	99% 4% 2% 1%
Swift	5.02	7,524	1017	5.01	100% 1% 1% 0%

n-body

source	secs	KB	gz	cpu	cpu load
Chapel	**21.55**	20,688	962	21.55	100% 0% 1% 0%
Swift	23.66	4,388	1253	23.65	100% 0% 0% 1%

CLBG: Website

- **site has a sound 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/only views of results
  - it's 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

CLBG: Background

Background:

- we ported these to evaluate Chapel serial/tasking performance
  - many "top" entries are more heroic than typical programmers would write
  - we strived for implementations that balance elegance with speed
- FAQ isn't particularly encouraging of adding new languages:

> **Why don't you include language X?**
> **… my favorite language implementation?**
> **… Microsoft® Windows® ?**
>
> Because I know it will take more time than I choose. Been there; done that.
>
> Measurements of "proggit popular" language implementations like Crystal and Nim and Julia will attract attention and be the basis of yet another successful website (unlike more Fortran or Ada or Pascal or Lisp). So make those repeated measurements at multiple workloads, and publish them and promote them.
>
> If you're interested in something not shown on the benchmarks game website then please take the program source code and the measurement scripts and **publish your own measurements**.

CLBG: This Effort

Feb 2016: Inquired about submitting a Chapel entry

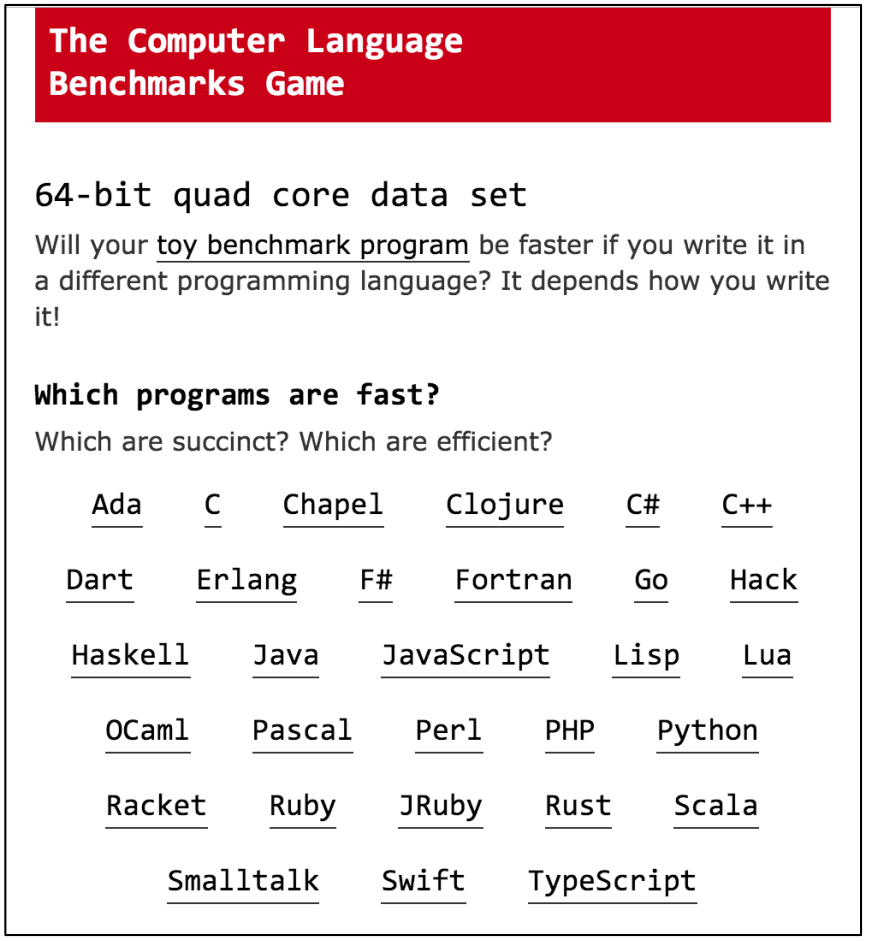
Apr 2016: Got a positive response

May 2016: Submitted first program

Sep 2016: Submitted final program

Listed on the front page:

Oct 2017: Upgraded to 1.14



CLBG: Improvements due to 1.14

1.14 improved many benchmarks with no code changes:

- **thread-ring:** benefitted from qthread native sync variables
  - climbed ~16 slots ⇒ now 5th fastest after Haskell, Go, F#, Scala
  - 1st most compact code followed by Ruby, Racket, Erlang, Ocaml, Python
- **fannkuch-redux:** benefitted from optimized array accesses
  - climbed from ~#22 to #6 in performance
  - ~1.5–2x more compact than most other top entries
- **chameneos-redux:** benefitted from tasking improvements
  - climbed from ~#11 to #8 in terms of performance
- **binary-trees:** benefitted from jemalloc improvements
  - climbed ~2 performance slots as a result
  - still ~5x off from top entries which use explicit memory pools
- **n-body:** saw marginal improvements, but climbed ~17 slots
- **regex-dna, revcomp:** saw marginal improvements, climbed ~3 slots
- **meteor:** saw marginal improvements, climbed ~1 slot
- **fasta:** saw marginal improvements, no change in rank

CLBG: Improvements due to 1.14

1.14 enabled code improvements for other benchmarks:

- **pidigits:** created versions that use the 'bigint' type
  - **pidigits:** uses operators everywhere
  - **pidigits-fast:** uses methods to avoid assigning returned records
- **knucleotide:** needed bug fix due to buggy auto-'use' of 'Sort' in 1.13
  - updated to new 'Sort' interfaces while here
  - also saw performance improvements from optimized array accesses
- **binary-trees:** created an initializer-based implementation
- **mandelbrot:** used complex values and the dynamic domain iterator
- **fasta:** removed a downcast on ascii(); simplified I/O due to a bug fix
- **meteor:** made use of enum.size
- **revcomp:** removed downcasts on ascii()

To date, have only submitted two of the above cases

- trying to avoid maintenance fatigue
- latest versions available in examples/benchmarks/shootout on GitHub

CLBG: Status

Chapel entry highlights (as of Oct 17th):

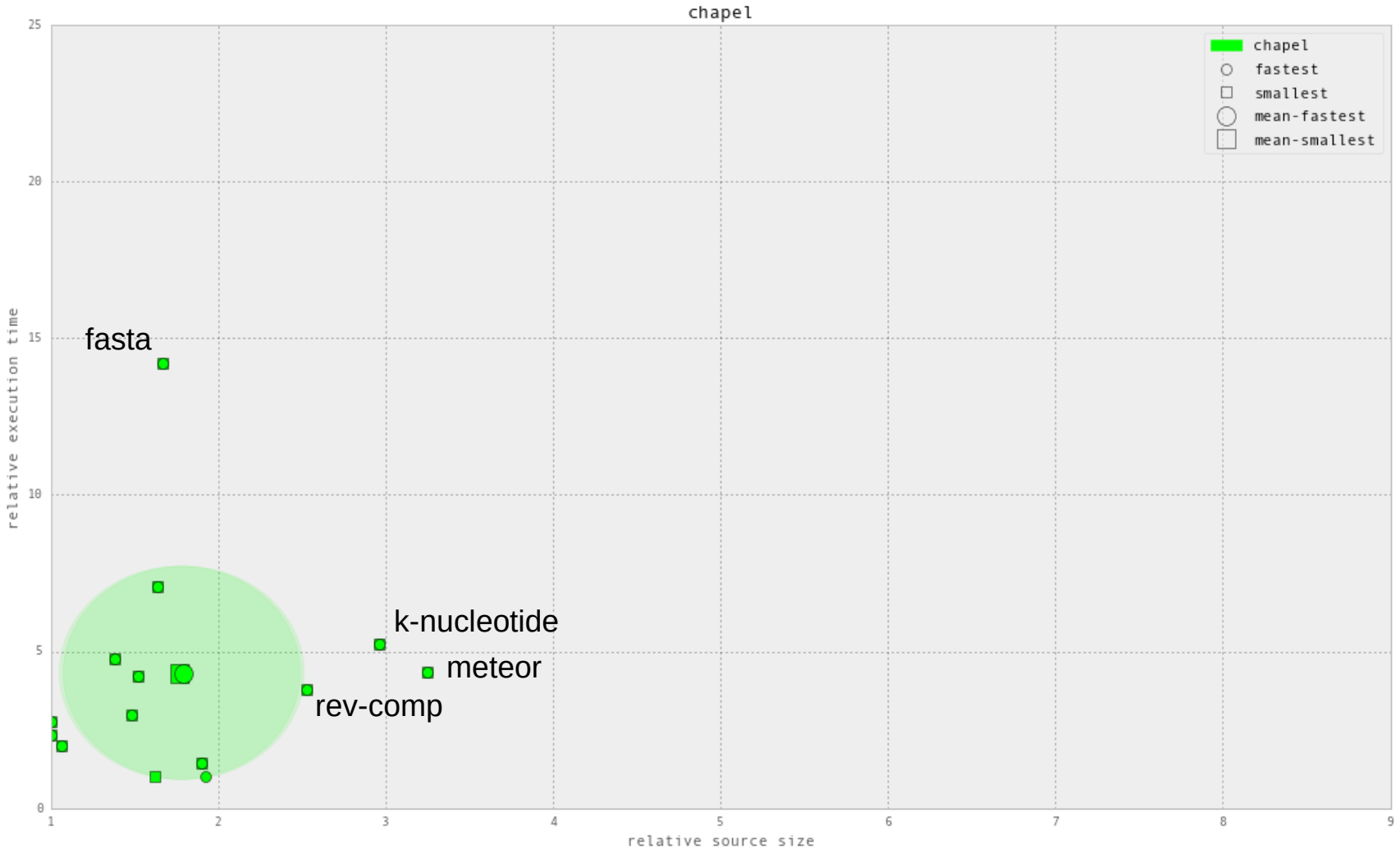
- performance rankings:
  - **top entries:** pidigits
  - **top-5 entries:** meteor-contest, thread-ring
  - **top-10 entries:** fannkuch-redux, chameneos-redux
  - **top-20 entries:** n-body, spectral-norm, binary-trees
- code compactness rankings:
  - **top entries:** n-body, thread-ring
  - **top-5 entries:** spectral-norm, pidigits
  - **top-20 entries:** mandelbrot, regex-dna, chameneos-redux, meteor

CLBG Scatter Plots

CLBG Scatter Plots

- **Made scatter plots to compare performance and code size**
  - created these to help us understand where we're falling short
    - e.g., helps us identify performance outliers
  - compared with the languages of highest interest to our team:
    - **traditional:** C, C++, Fortran, Java
    - **productive:** Python
    - **modern:** Scala, Go, Rust, Swift
  - each program is scaled by the fastest/smallest entries for that axis
    - In *any* language -- not restricted to the subset we're focusing on here
    - e.g., binary trees is scaled by C's time (fastest) and Ruby's size (smallest)
  - data is from the CLBG repository on Oct 18th
- **Notes:**
  - these only characterize the submitted programs
    - i.e., better versions could potentially be written in each language
    - however, this is the data we have to work with
  - not all languages have entries for all programs

CLBG Scatter Plots: Chapel Programs

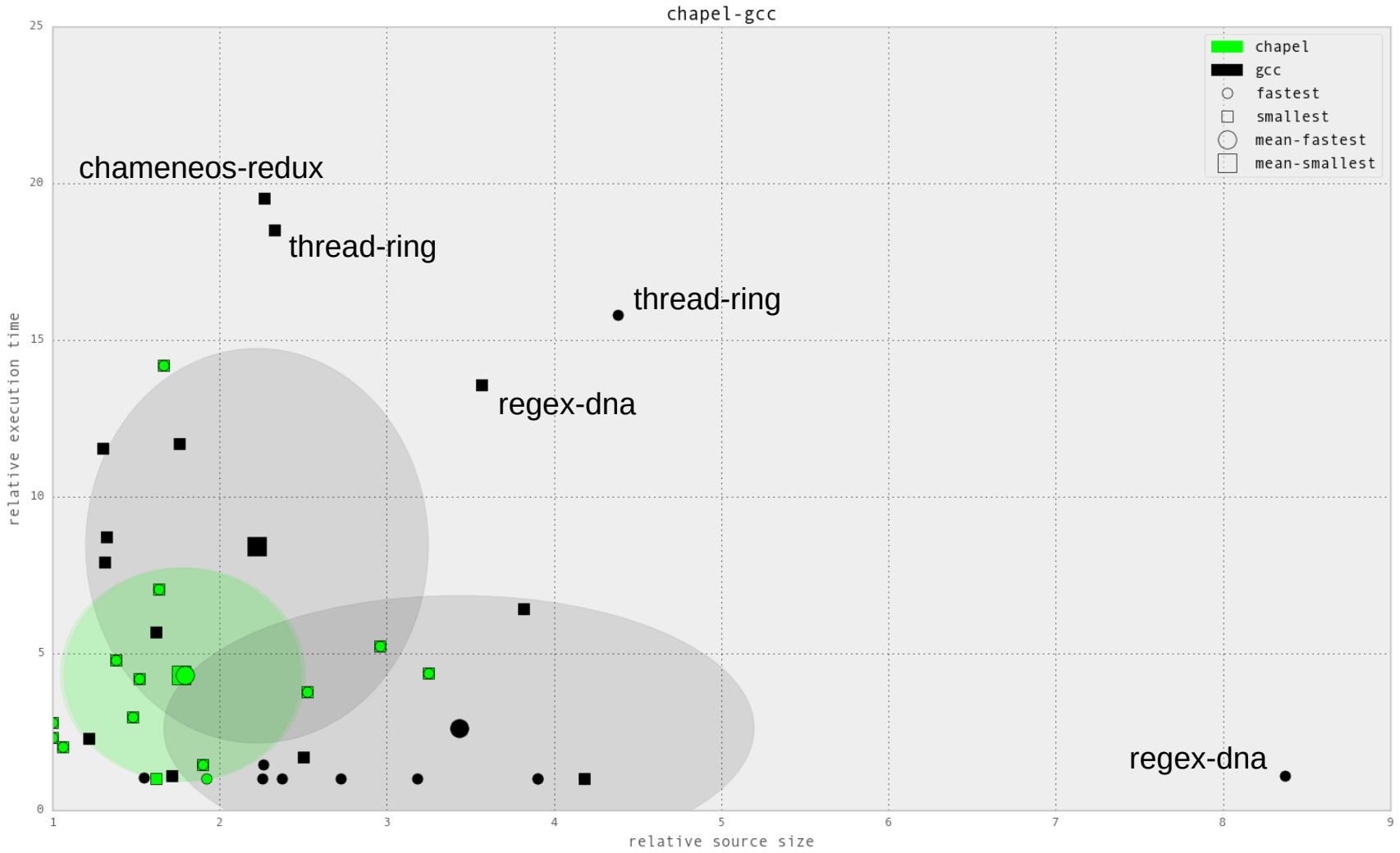


Pairwise Language Comparison Graphs

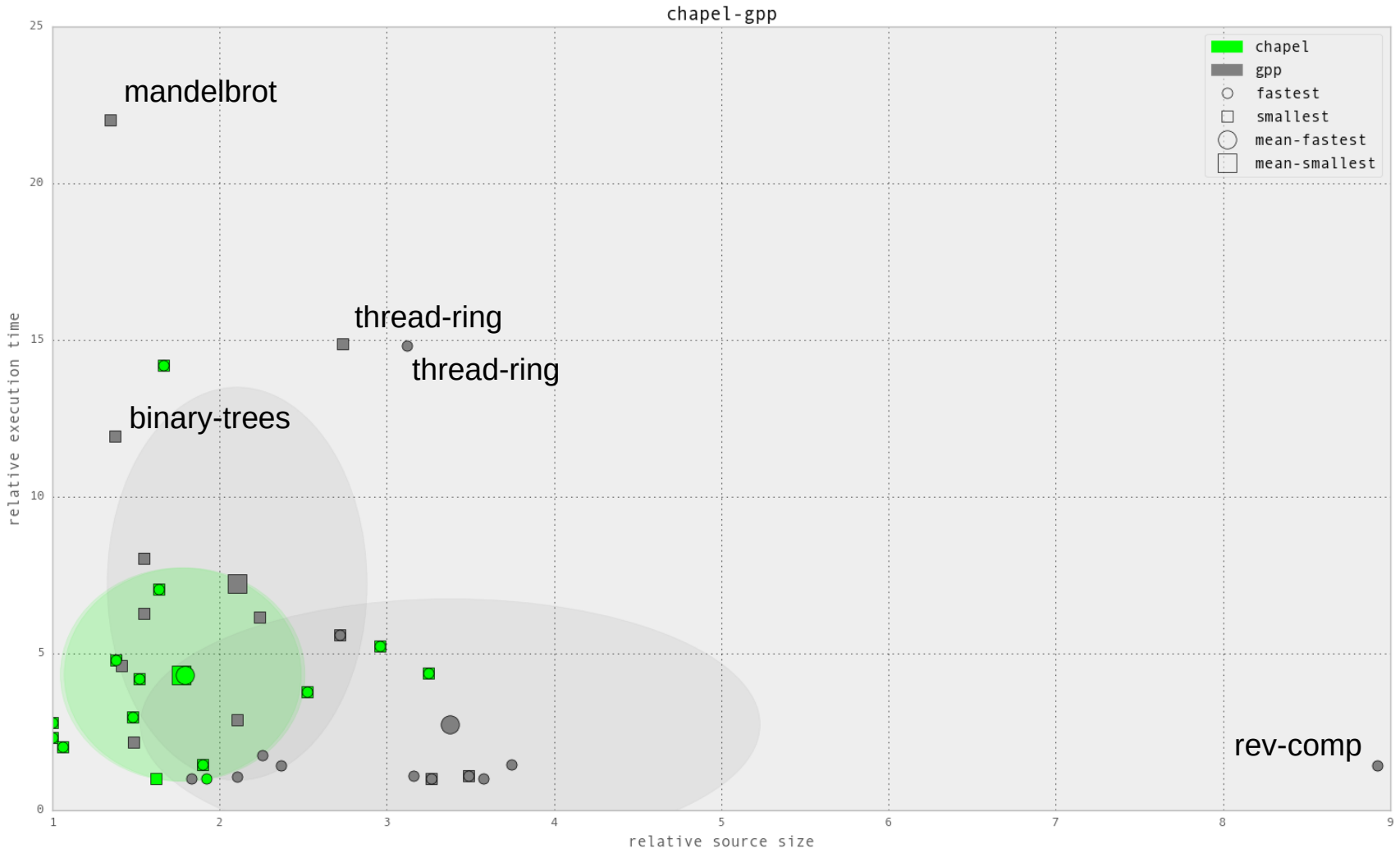
The first series of graphs compares languages pairwise

- For each language, we plot…
  …the fastest version of each benchmark as a circle
  …the smallest version of each benchmark as a square
  …the mean of each set of benchmarks as a larger square/circle
- We also plot an oval at 1σ (a standard deviation away from the mean)
  - this provides an overall "profile" for the language's fastest/smallest entries
- The axis scales are fixed across the graphs
  - in a few cases we zoom out in the subsequent slide to display outliers

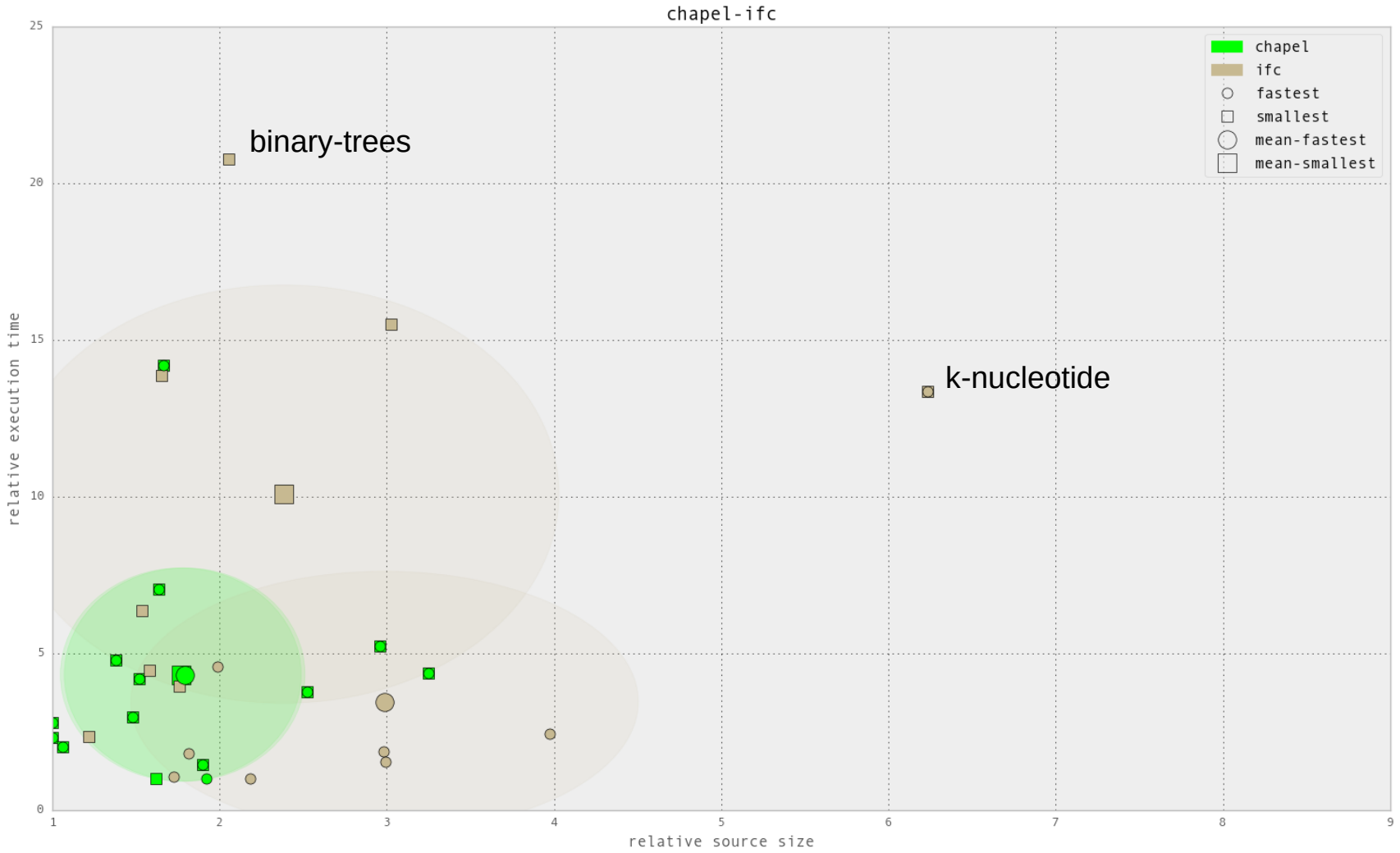
CLBG Scatter Plots: Chapel vs. C



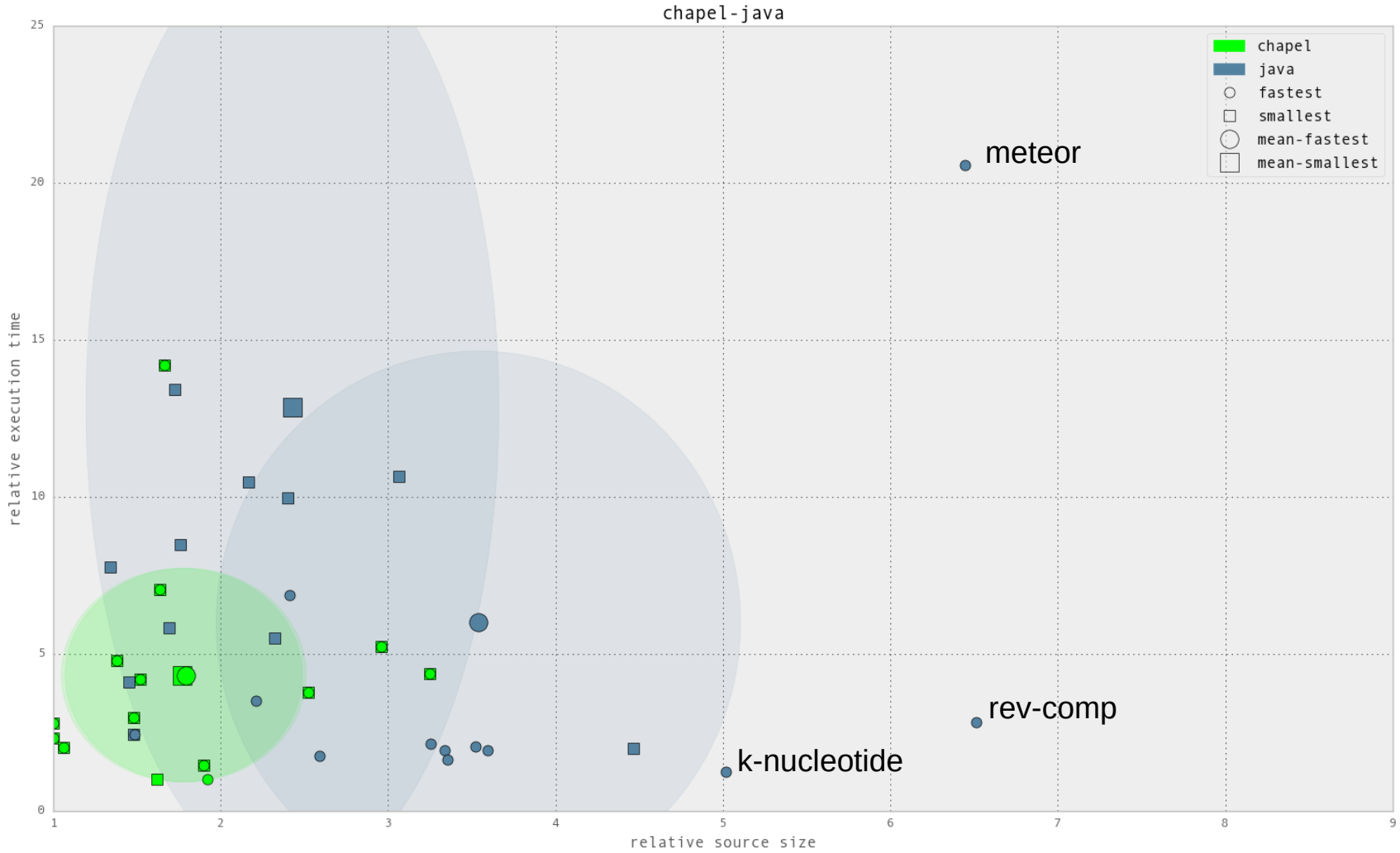
CLBG Scatter Plots: Chapel vs. C++



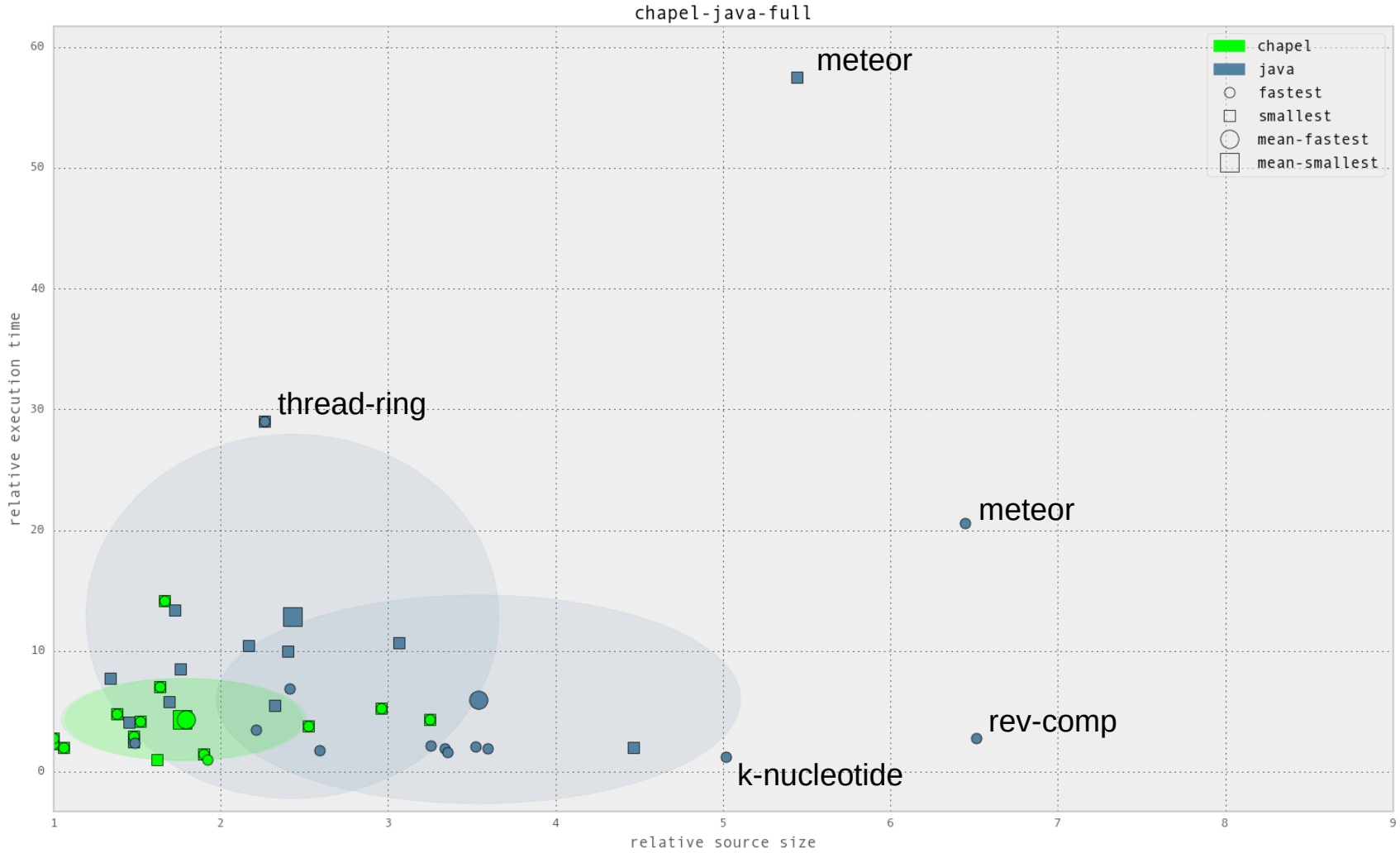
CLBG Scatter Plots: Chapel vs. Fortran



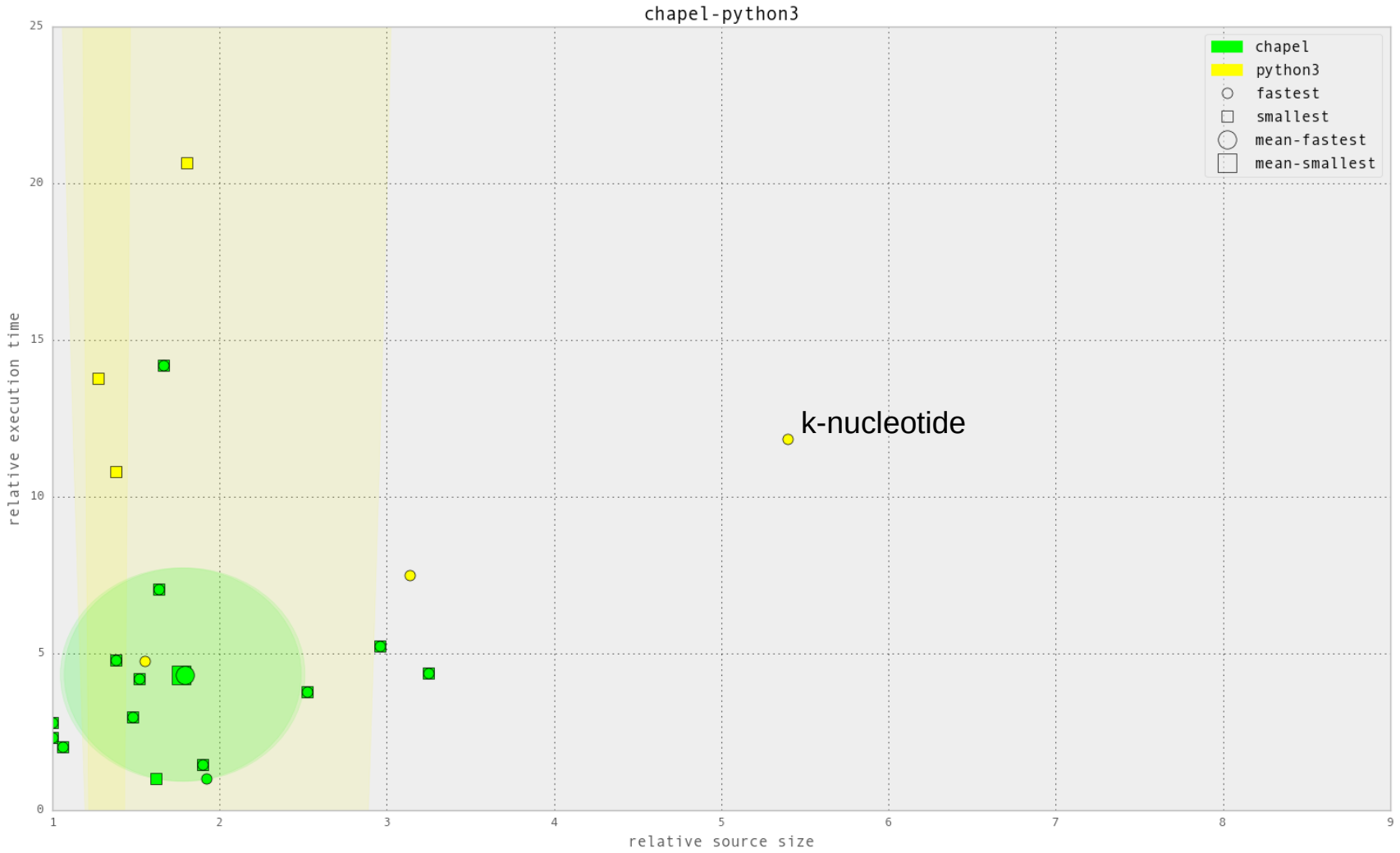
CLBG Scatter Plots: Chapel vs. Java



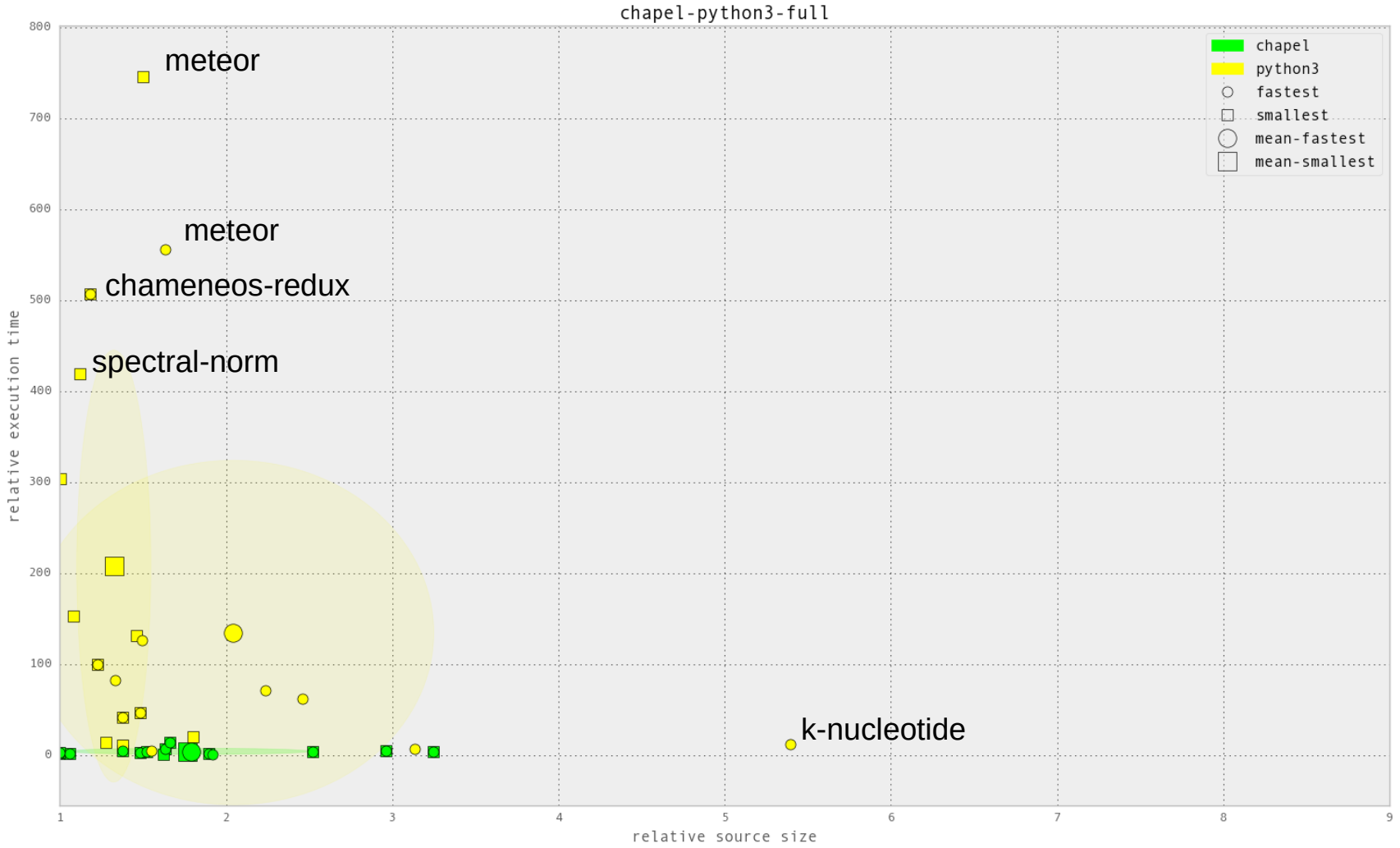
CLBG Scatter Plots: Chapel vs. Java (zoom)



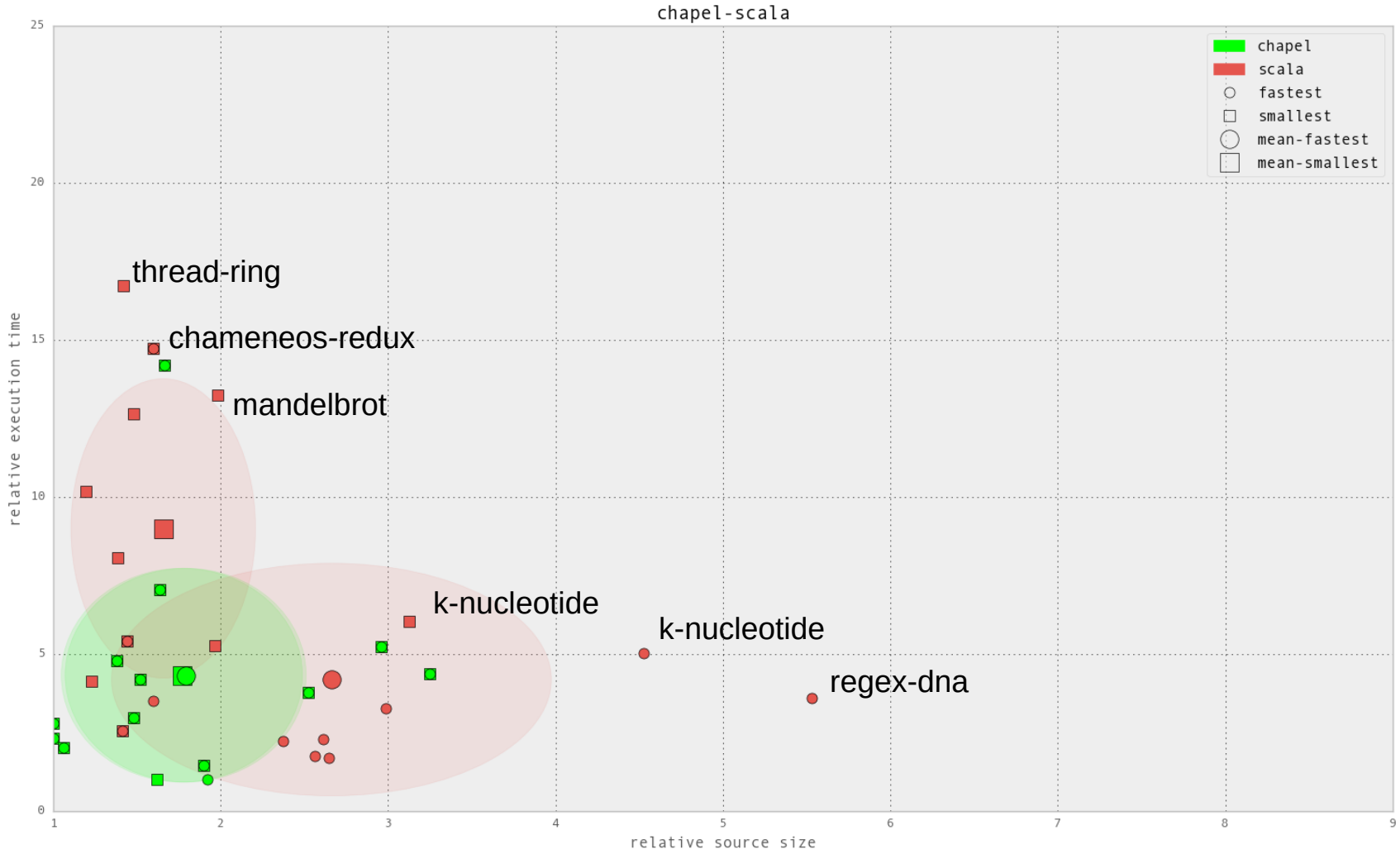
CLBG Scatter Plots: Chapel vs. Python



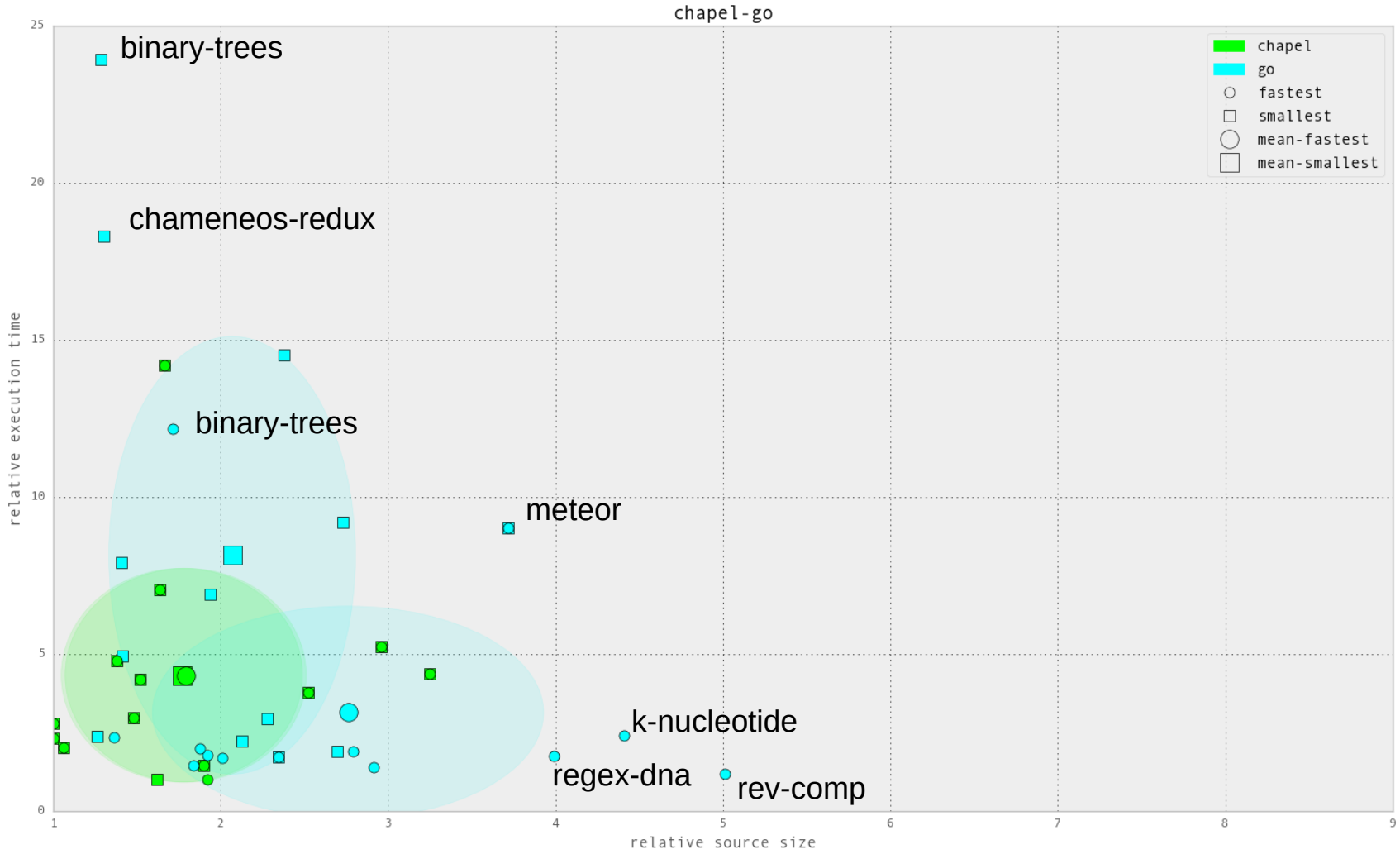
CLBG Fastest Entries: Chapel vs. Python (zoom)



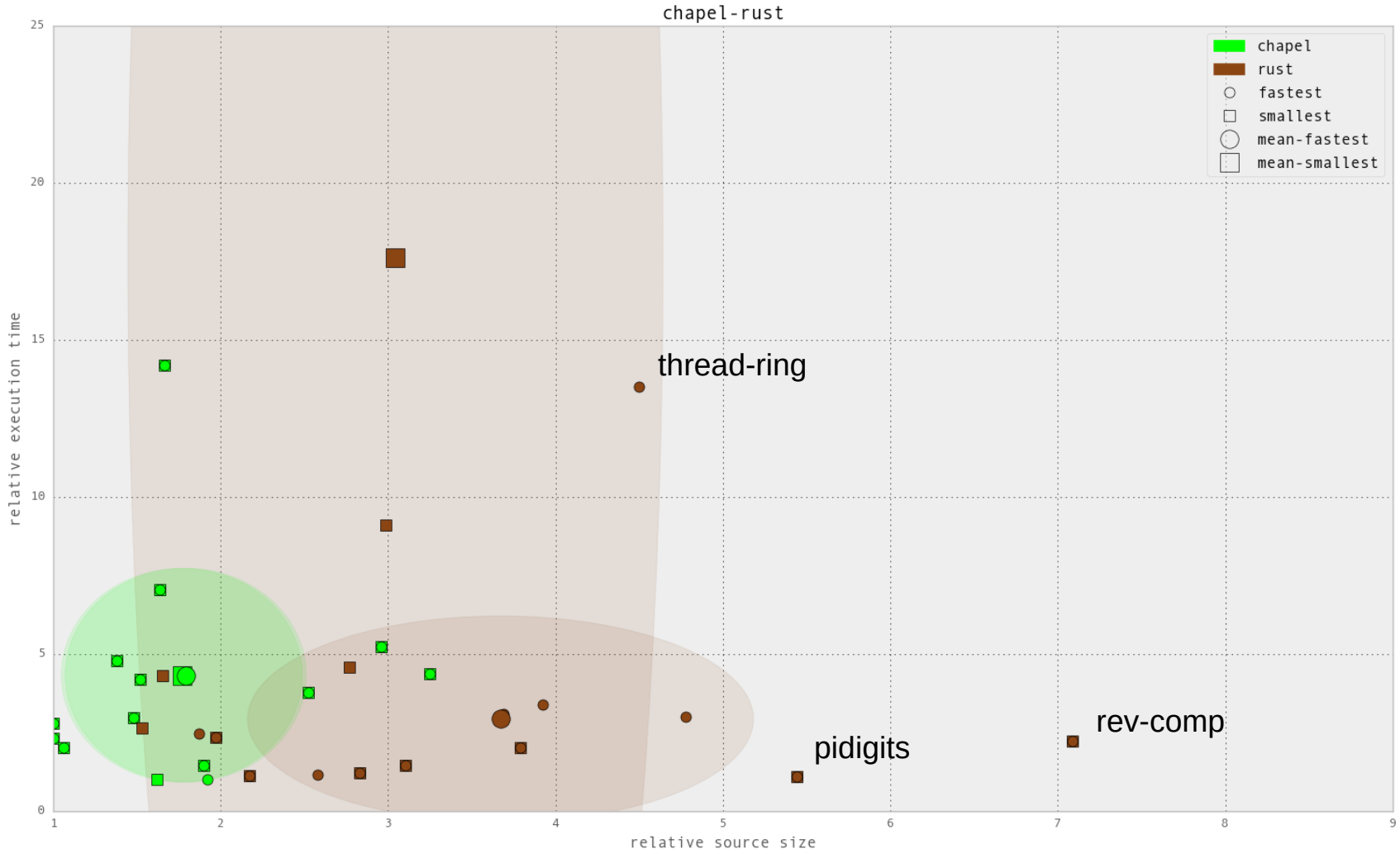
CLBG Scatter Plots: Chapel vs. Scala



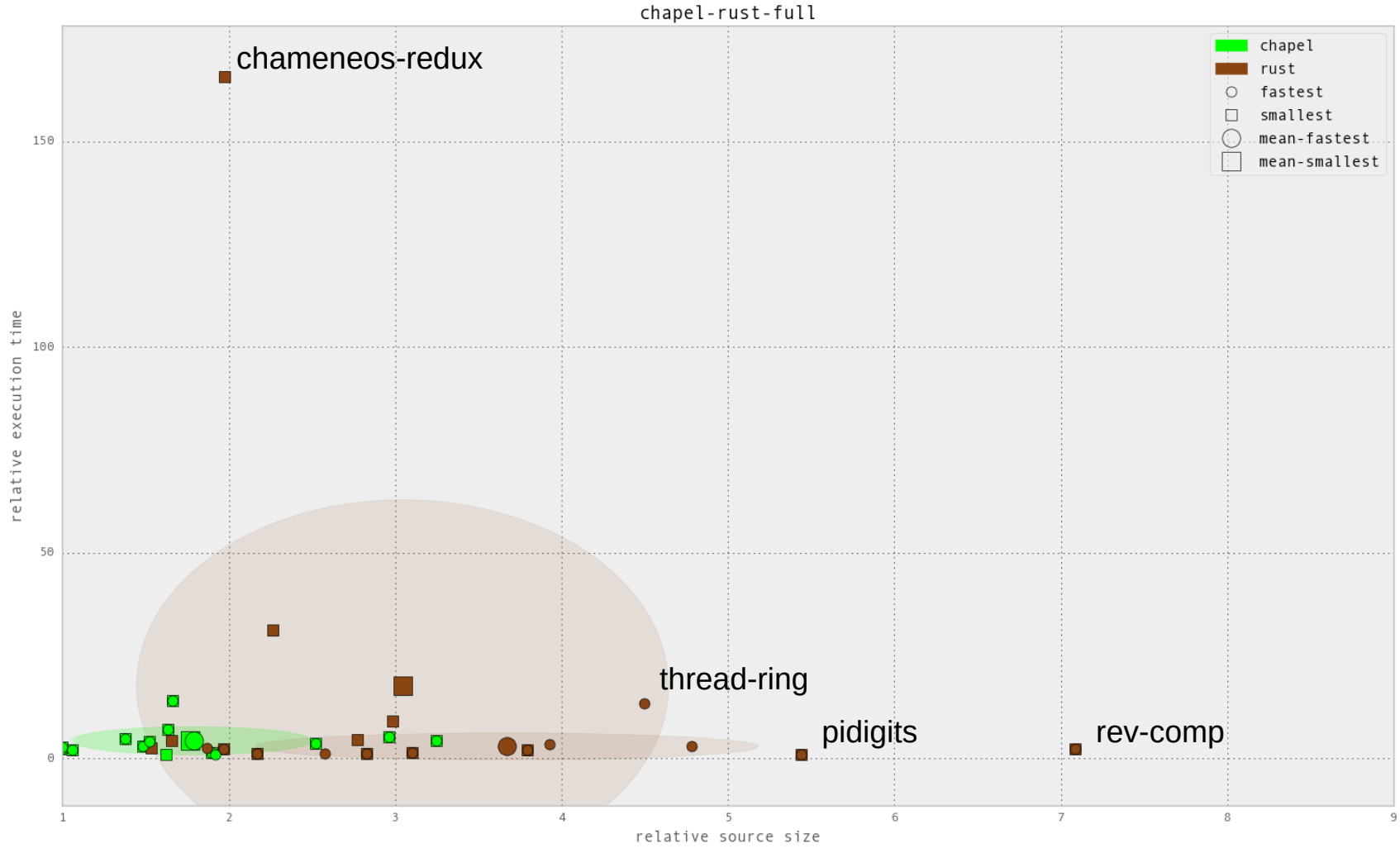
CLBG Scatter Plots: Chapel vs. Go



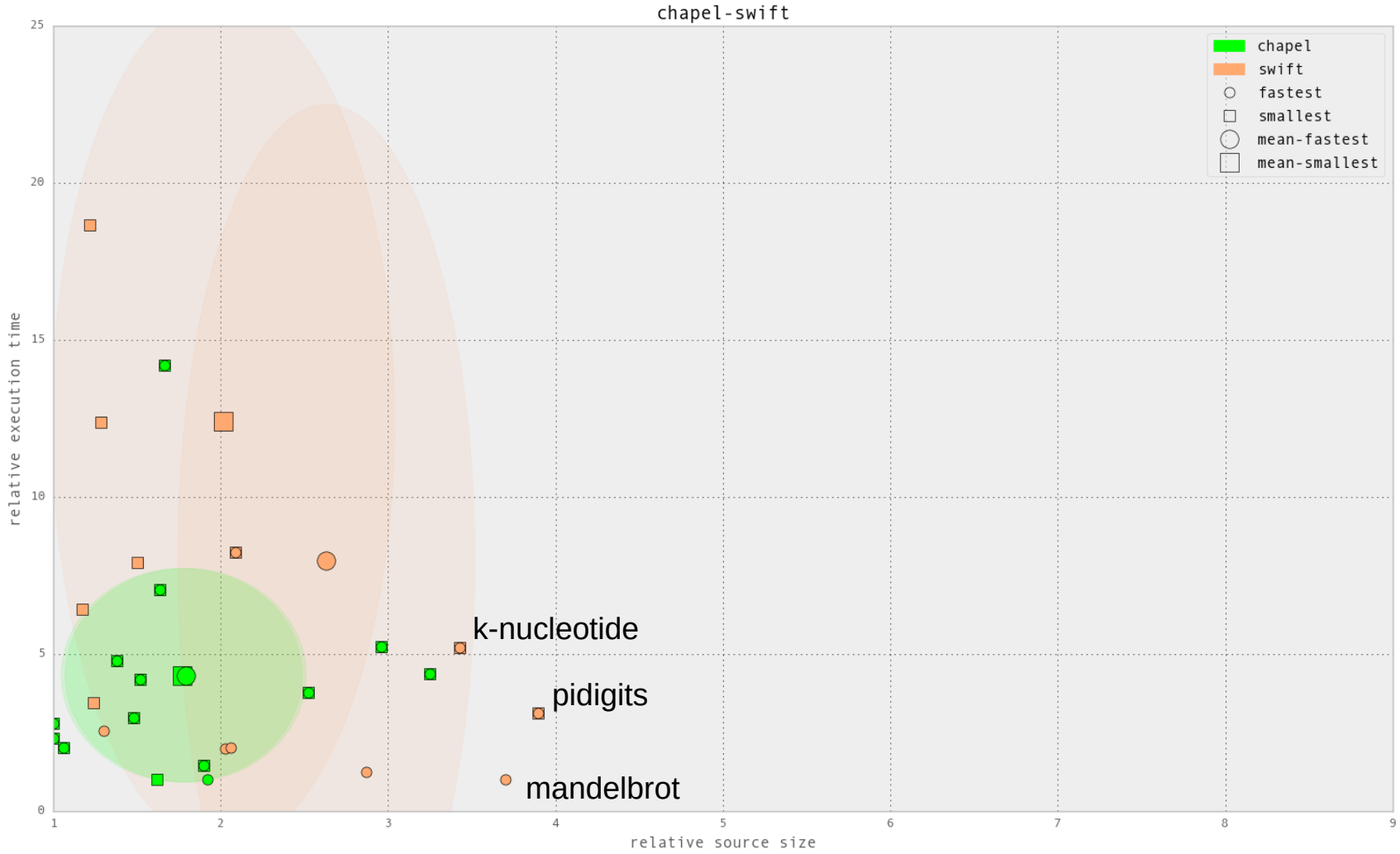
CLBG Scatter Plots: Chapel vs. Rust



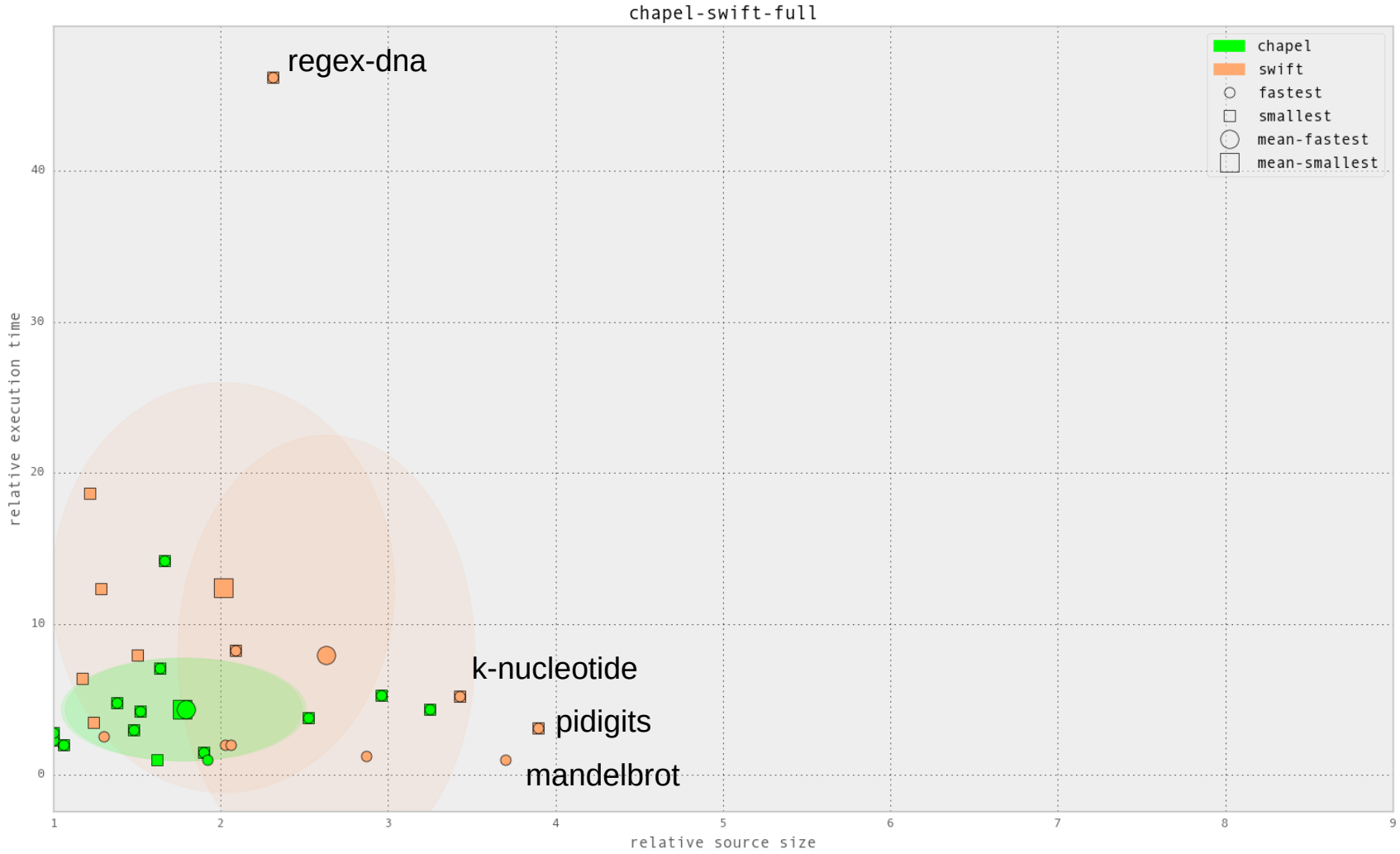
CLBG Scatter Plots: Chapel vs. Rust (zoom)



CLBG Scatter Plots: Chapel vs. Swift



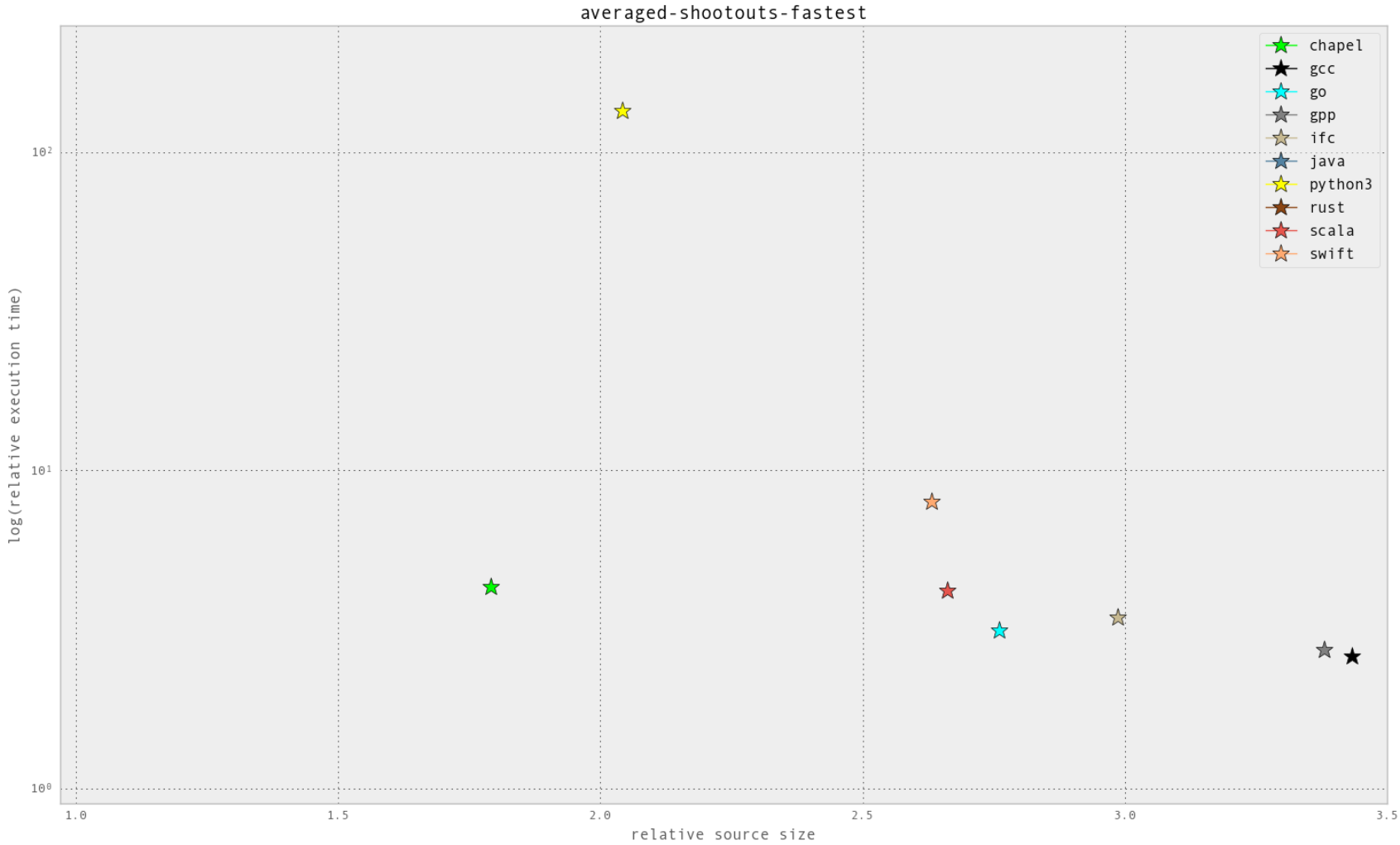
CLBG Scatter Plots: Chapel vs. Swift (zoom)



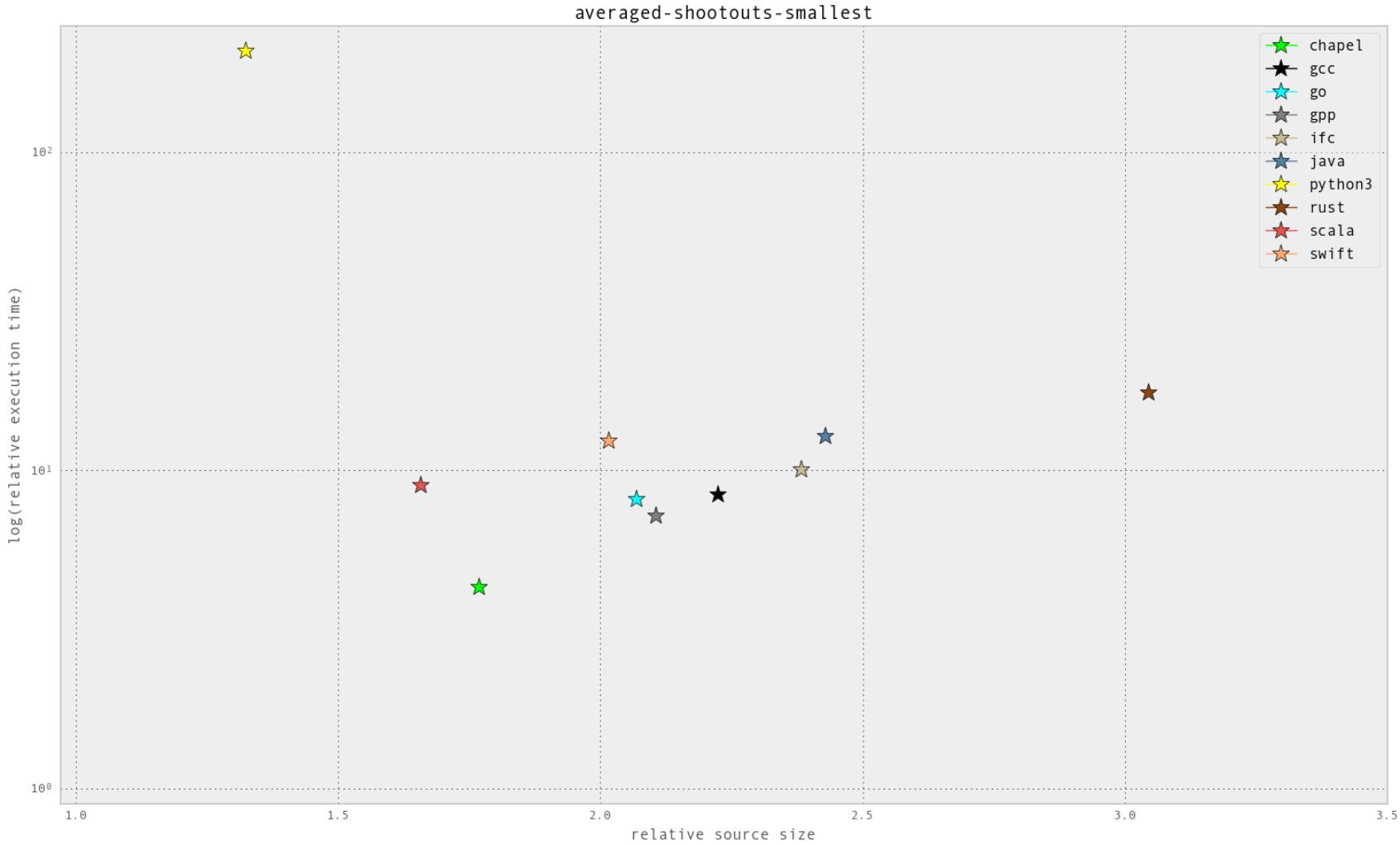
Language Summary Plots

- **The following two graphs plot the means for all languages**
  first: across the set of fastest entries
  then: across the set of most compact entries
- **Note that the y-axis is logarithmic**
  - otherwise, Python's inclusion flattens all data along the x-axis

CLBG Fastest Codes: Averages (log scale perf)



CLBG Smallest Codes: Averages (log scale perf)

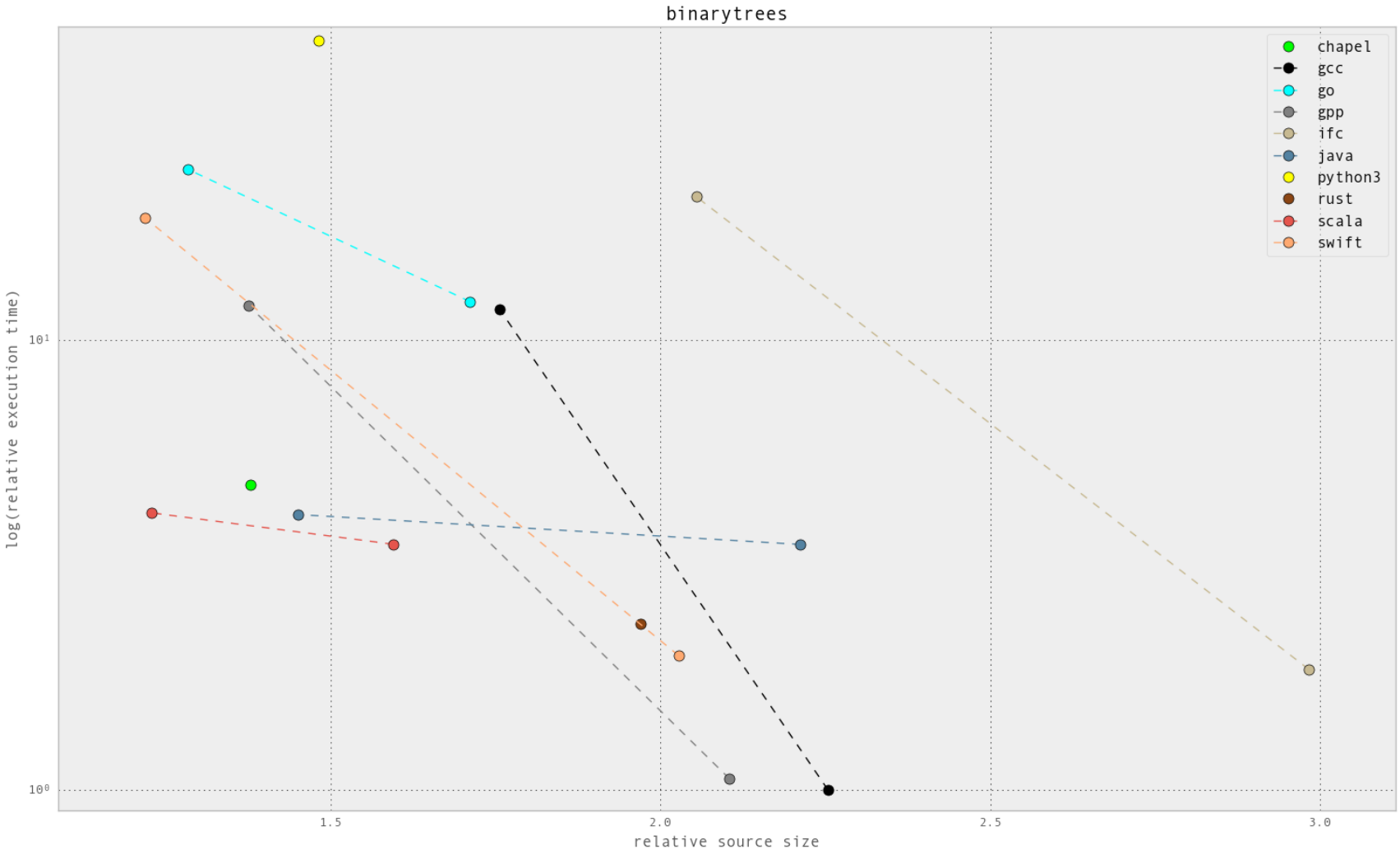


Per-Benchmark Comparison Graphs

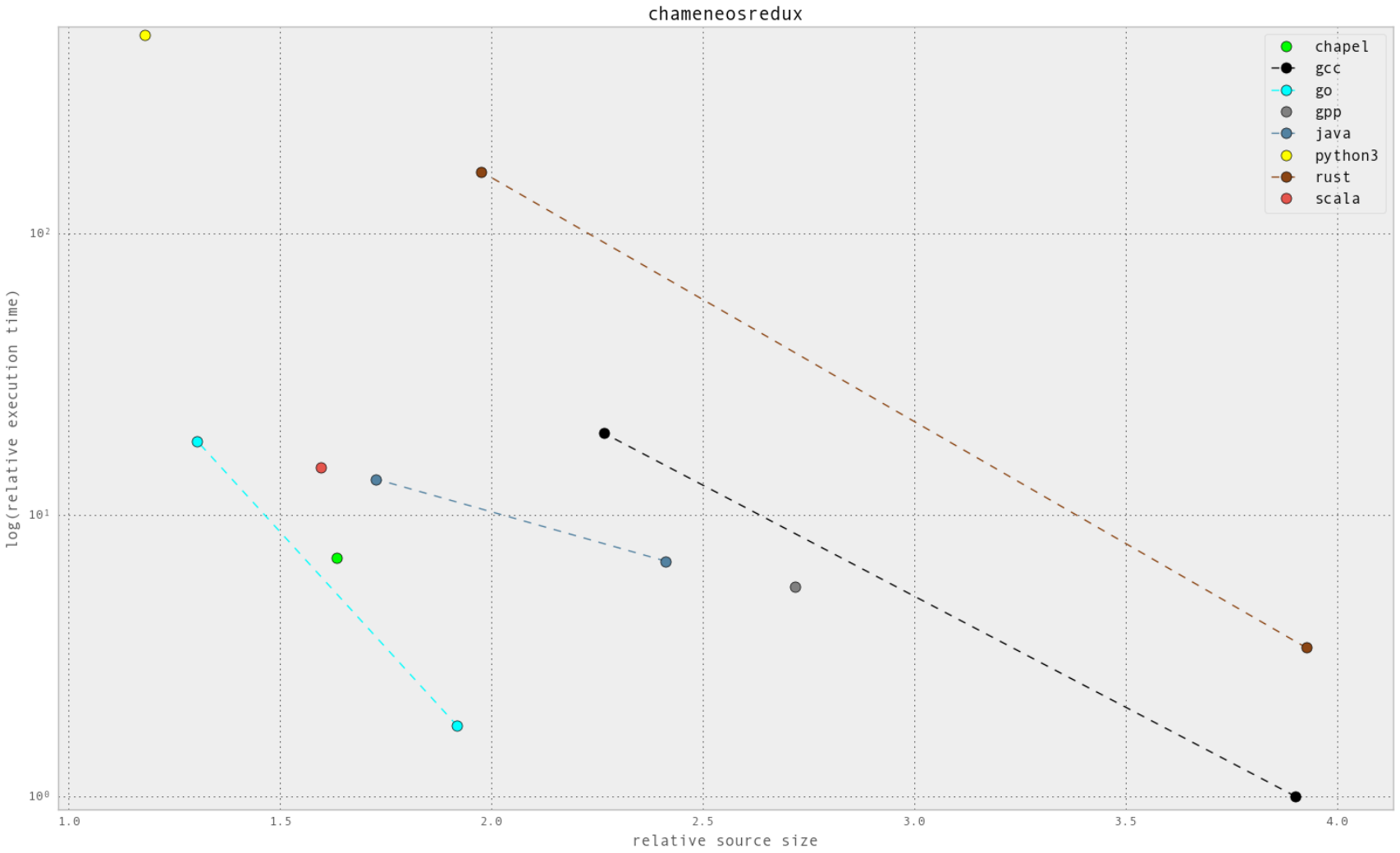
The next graphs compare benchmarks across languages

- For each language, we plot its fastest and most compact version
- We connect these versions with a line to help the eye associate them
  - note that this line doesn't imply a bound or a constraint, just a visual link
- Note that the y-axis is logarithmic

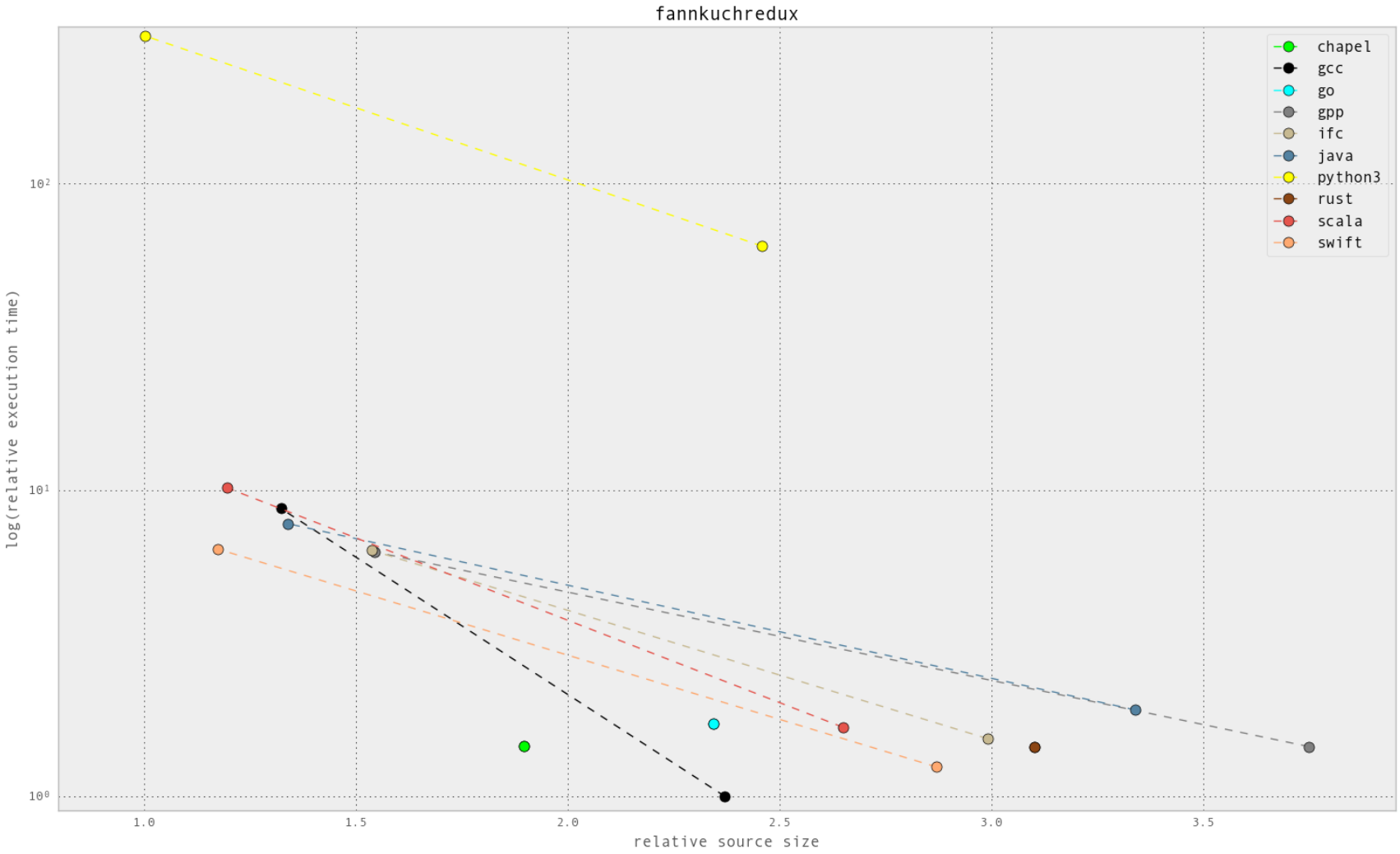
Binary-Trees: Language Comparisons



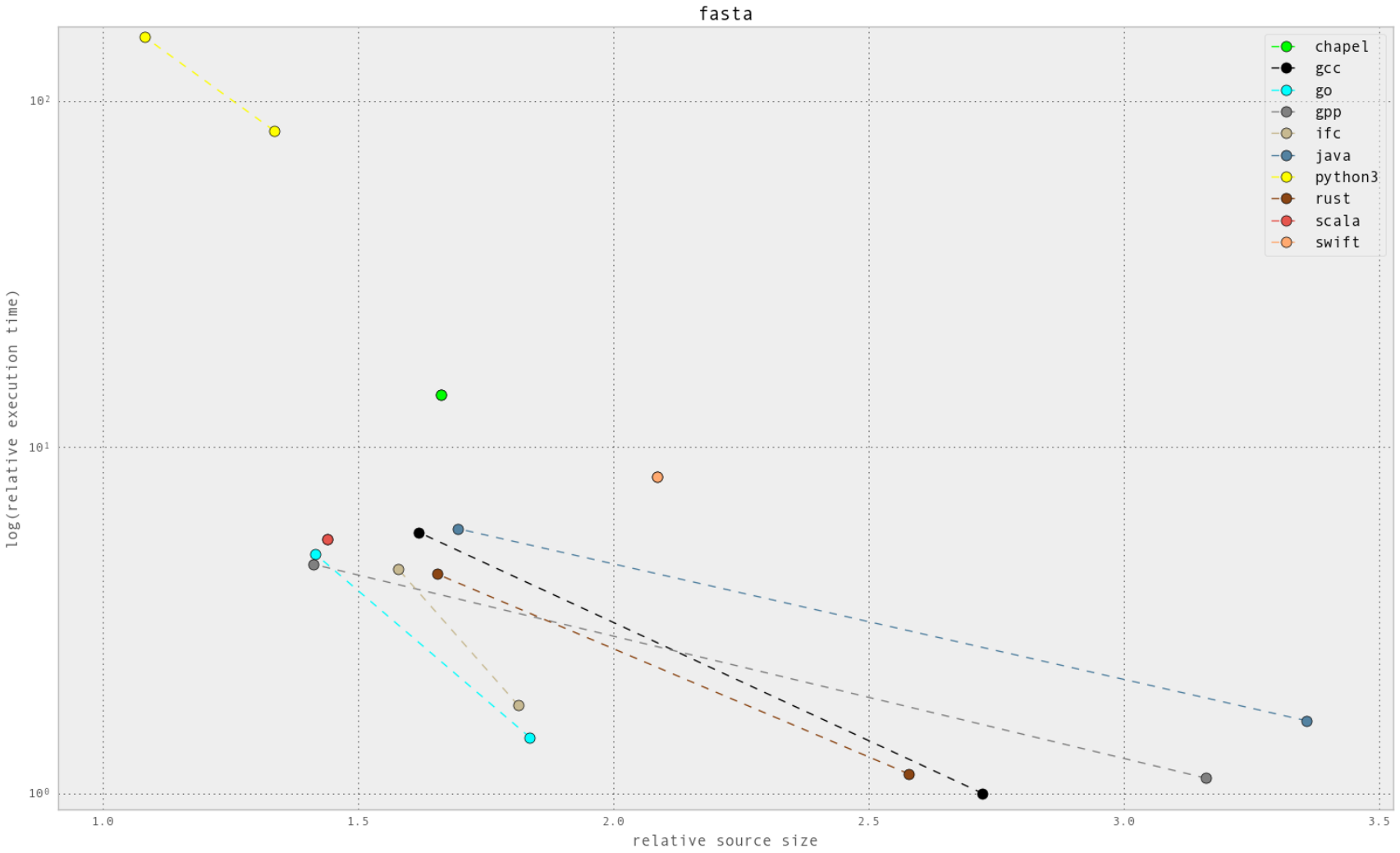
Chameneos-redux: Language Comparisons



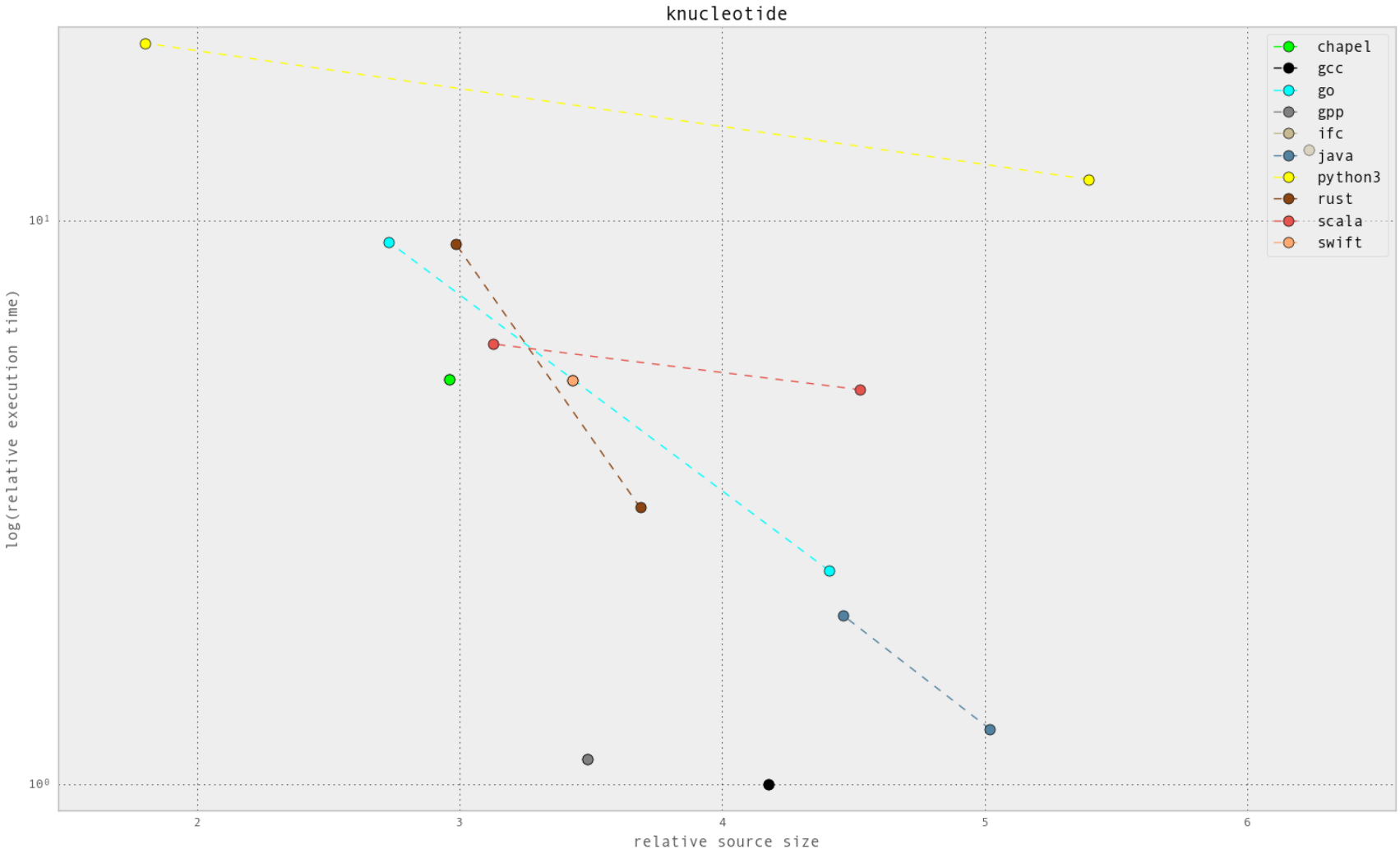
Fannkuch-redux: Language Comparisons



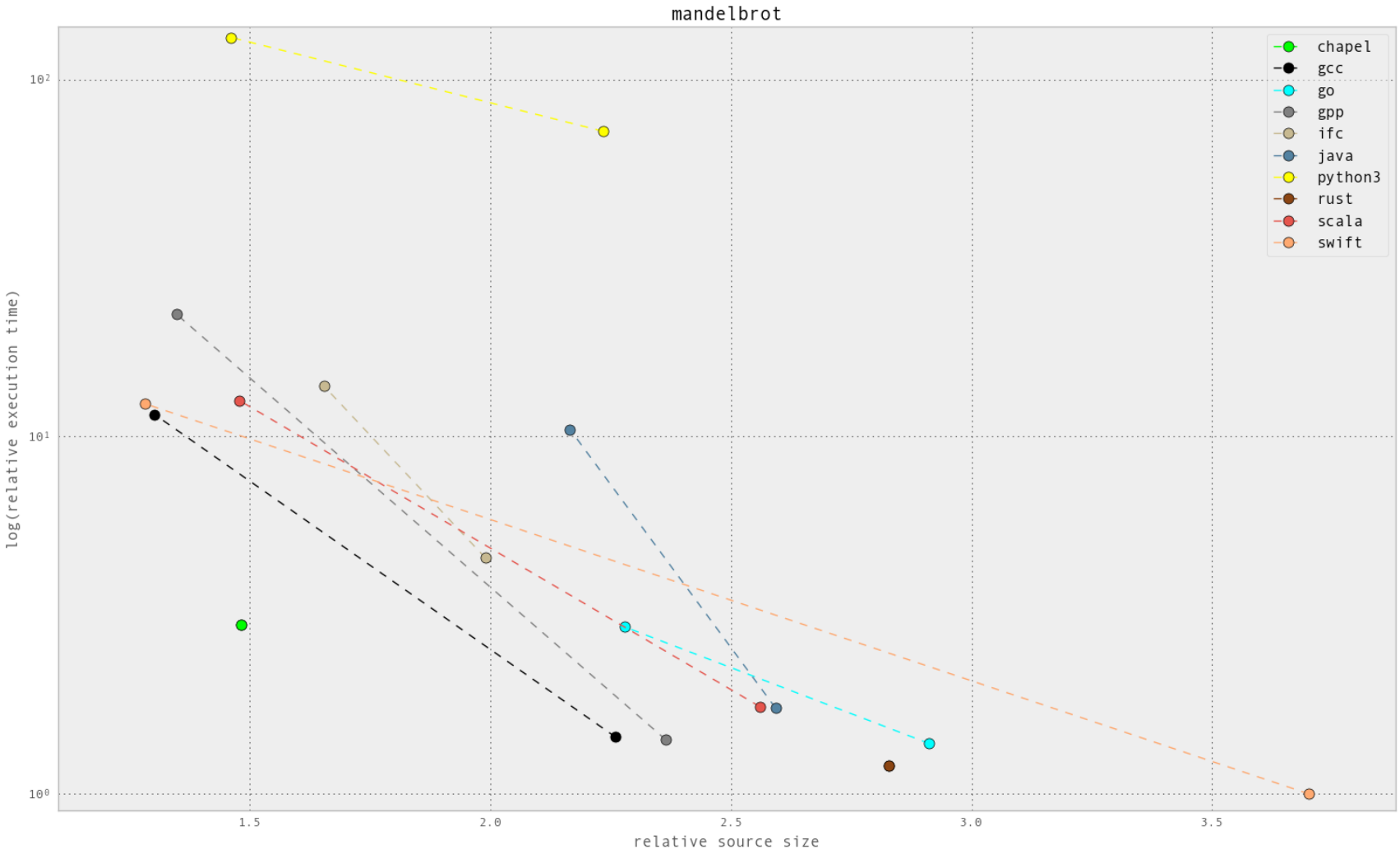
Fasta: Language Comparisons



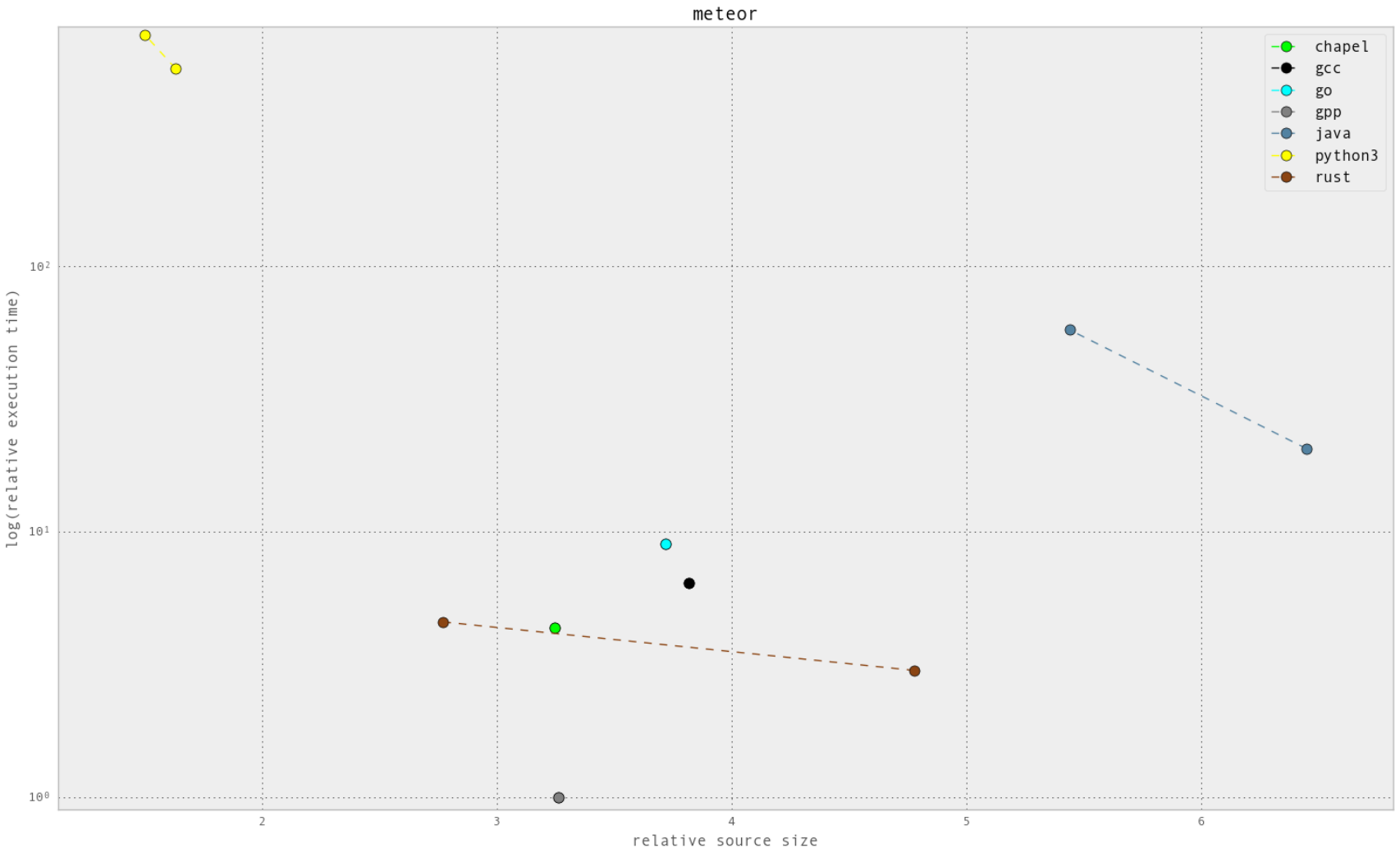
K-Nucleotide: Language Comparisons



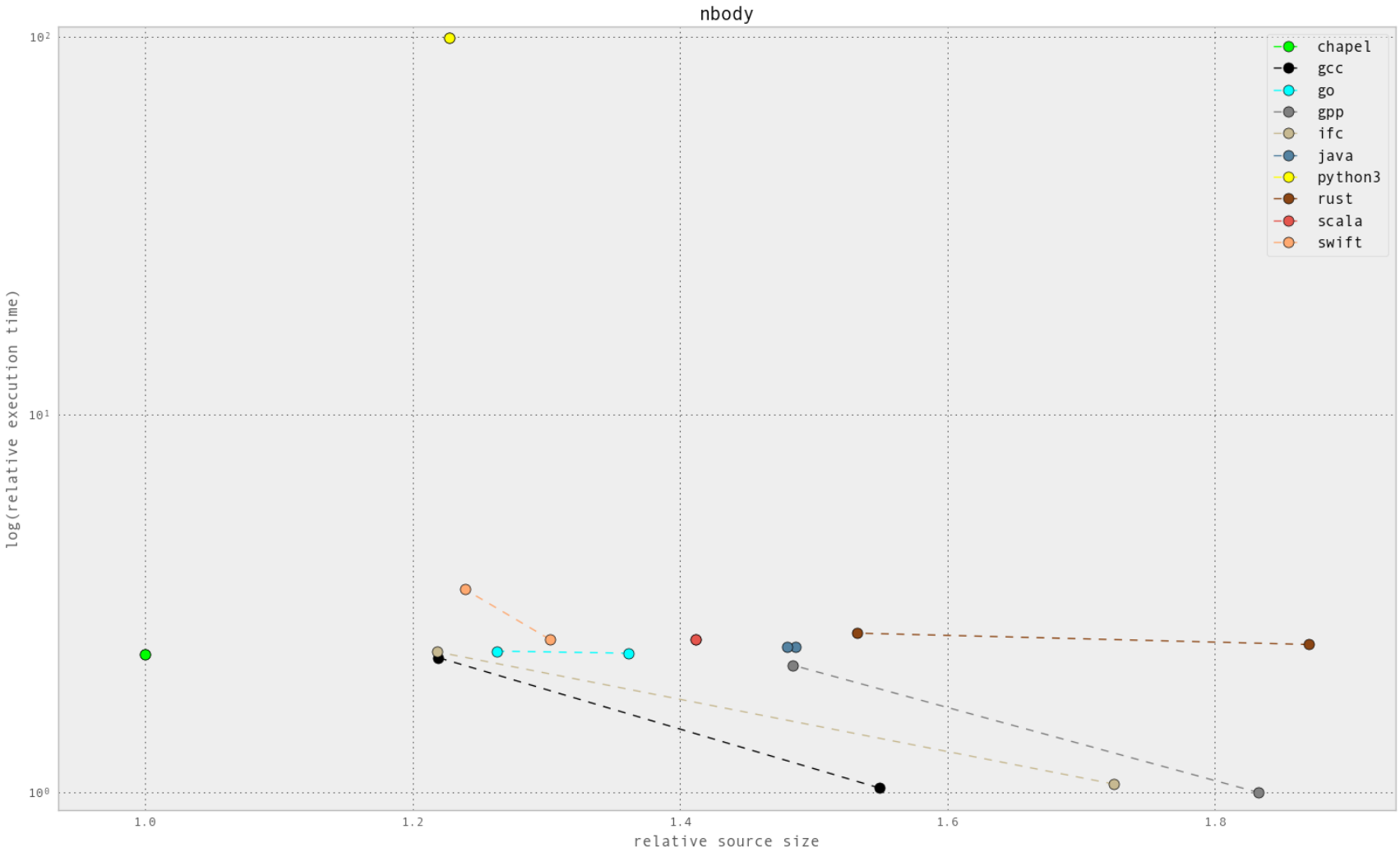
Mandelbrot: Language Comparisons



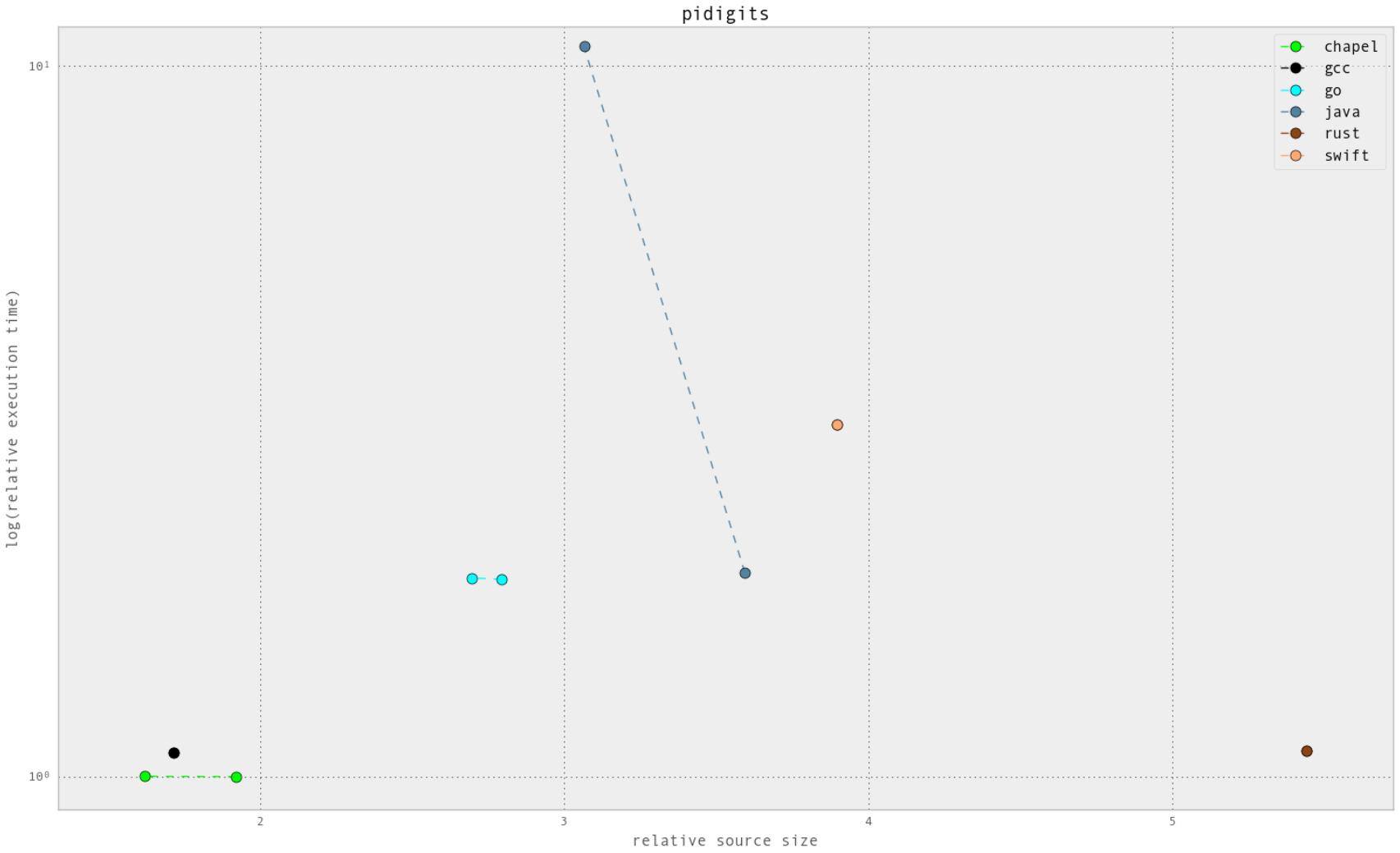
Meteor: Language Comparisons



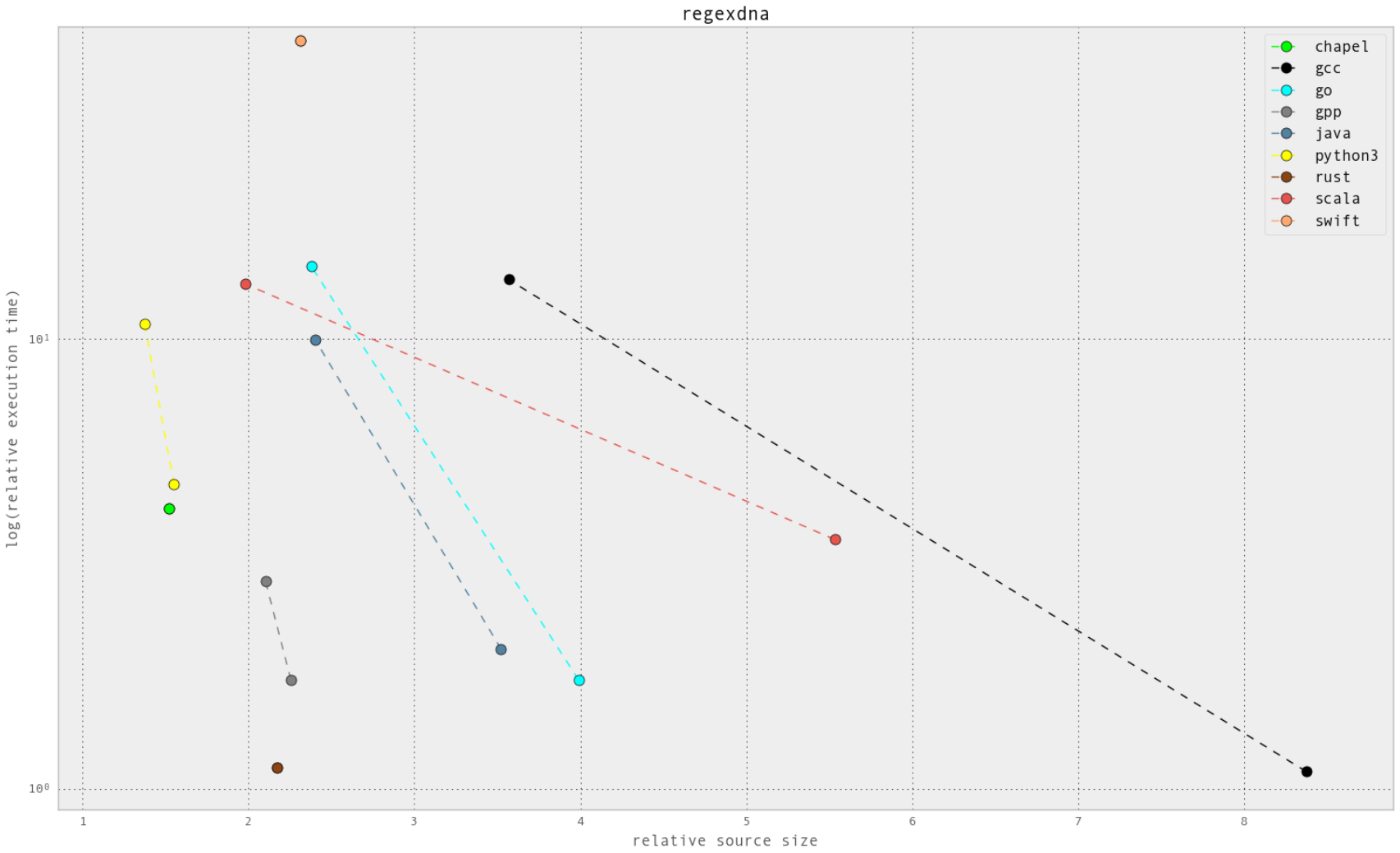
N-Body: Language Comparisons



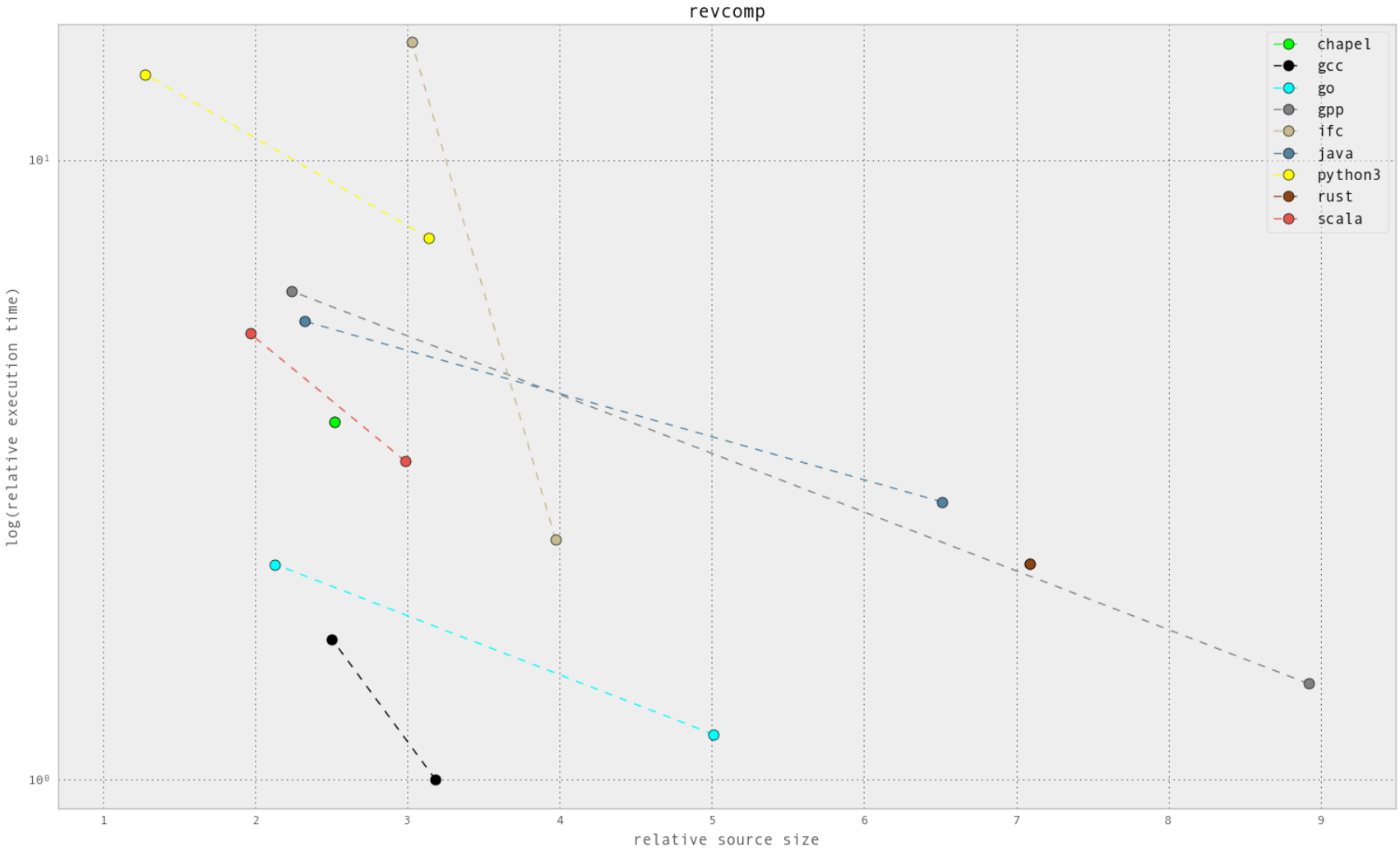
Pidigits: Language Comparisons



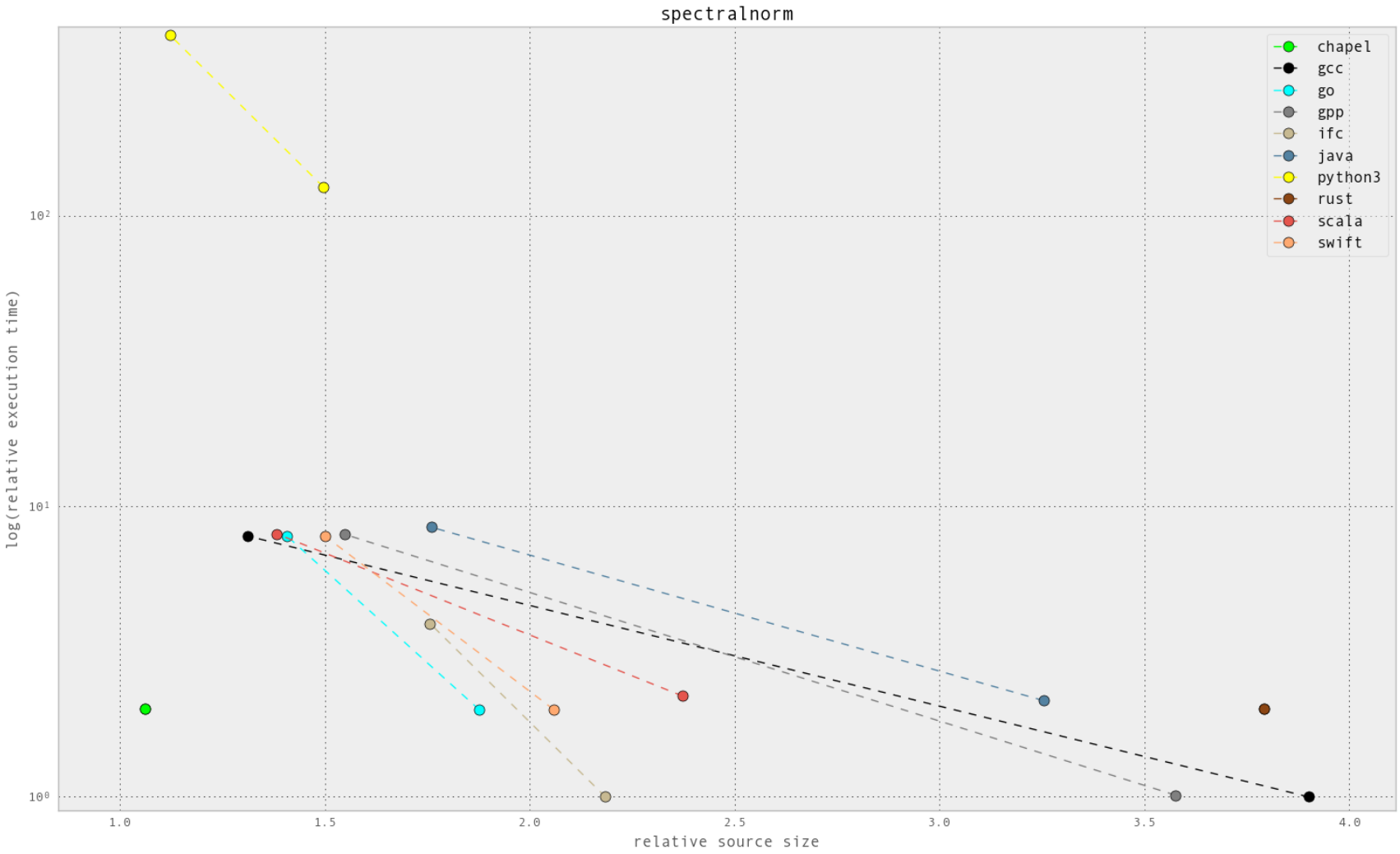
Regexdna: Language Comparisons



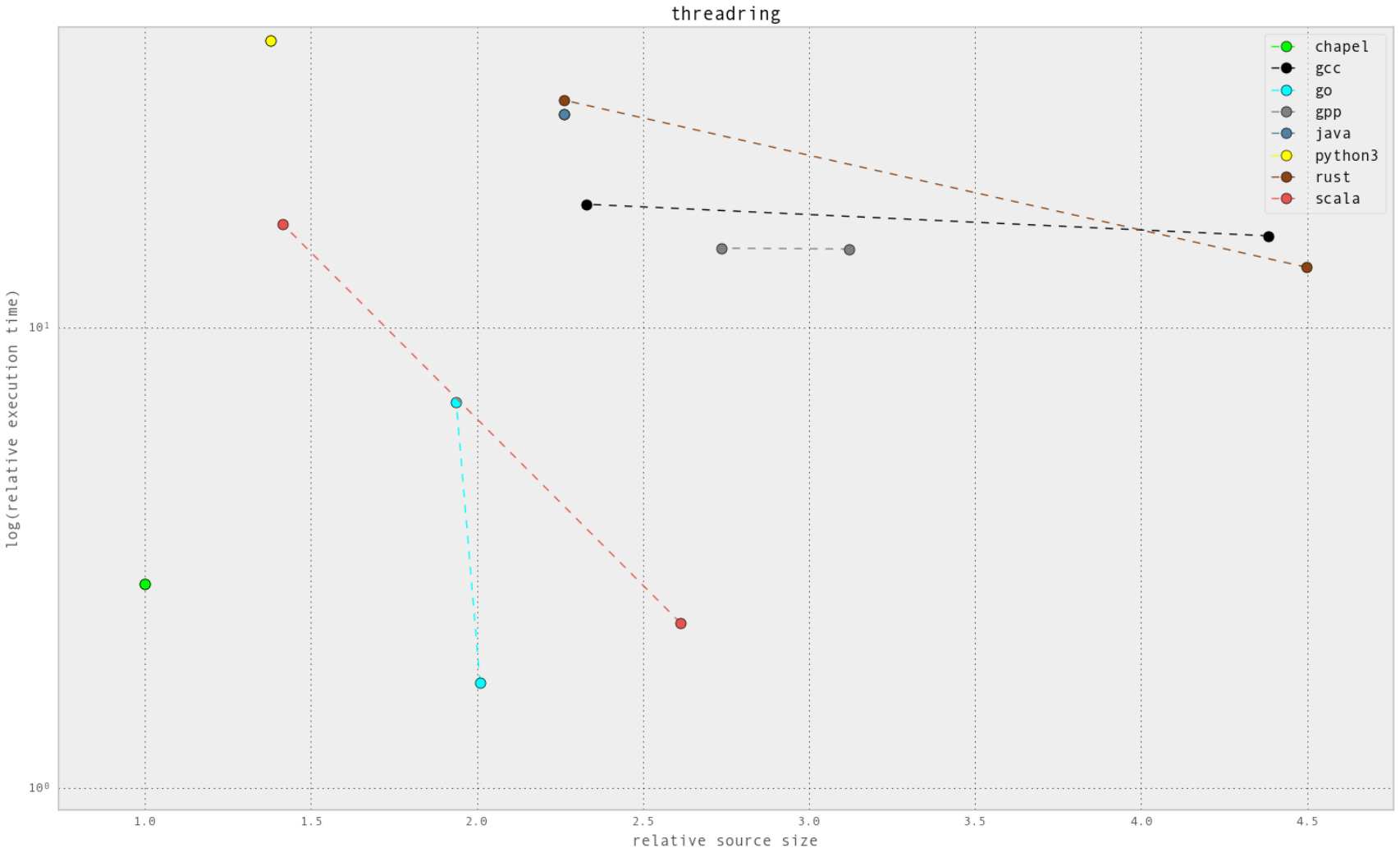
Reverse-Complement: Language Comparisons



Spectral-Norm: Language Comparisons



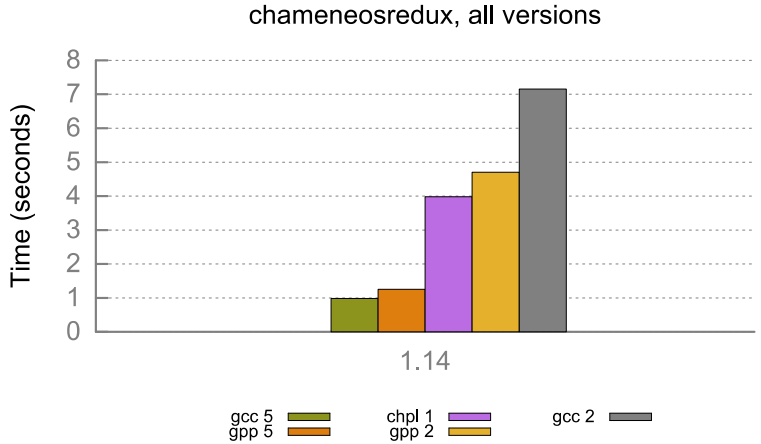
Thread-Ring: Language Comparisons



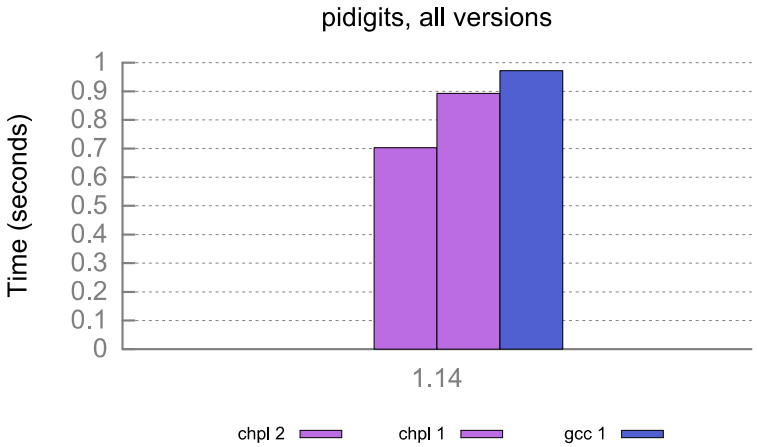
CLBG Comparison Graphs

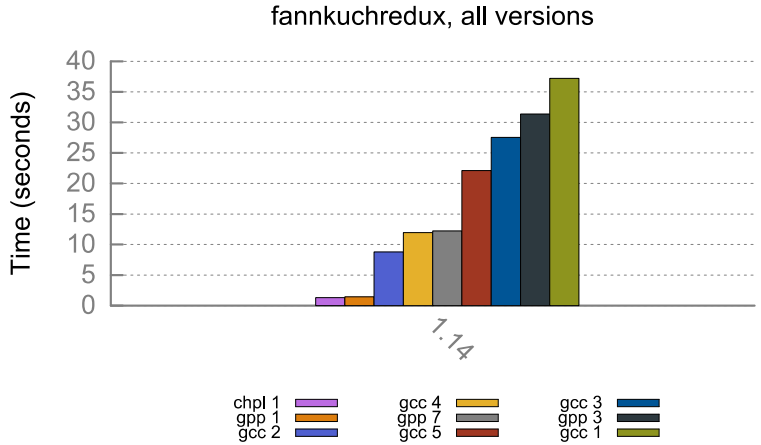
CLBG: Comparisons to C/C++

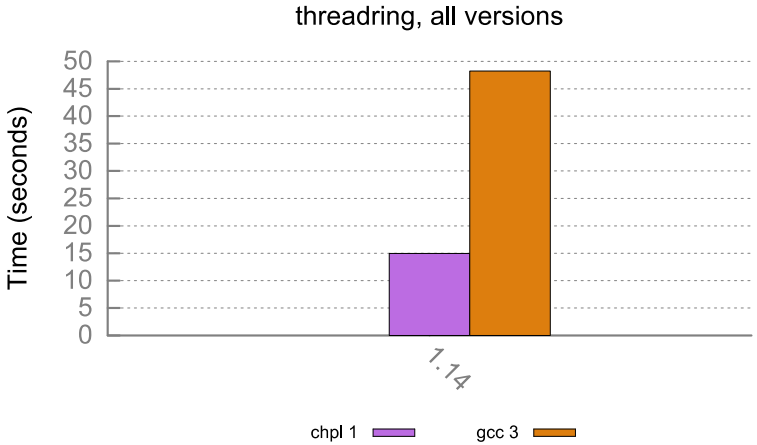
- **The following graphs compare Chapel to C/C++ versions**
  - an update to what we did for the 1.11 release notes
  - run using our team's systems:
    - 2 x 12-core Intel Xeon
    - gcc/g++
  - reflects more recent HW/compiler than the official CLBG system
  - yet imperfect:
    - some C/C++ entries are tuned specifically for the CLBG system
    - others rely on libraries that are not installed on our system

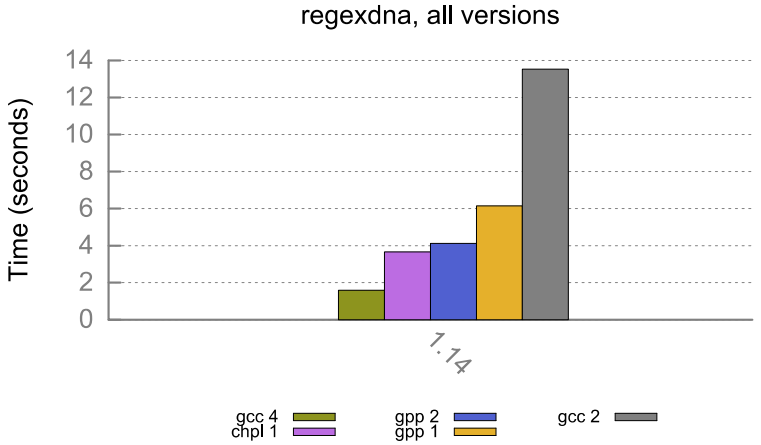


CLBG: Comparisons to C/C++

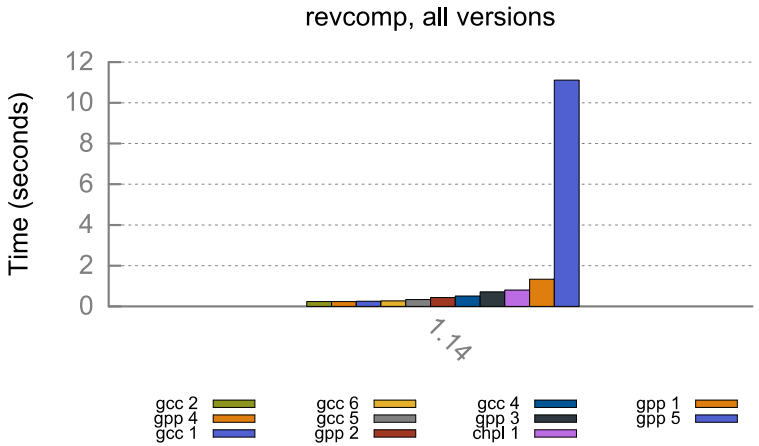


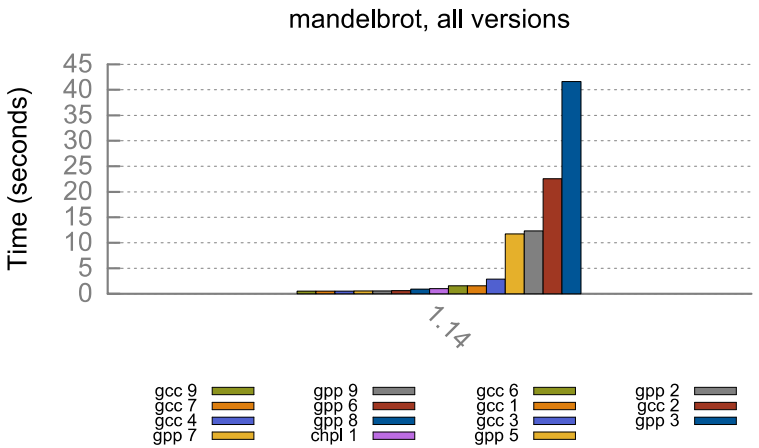


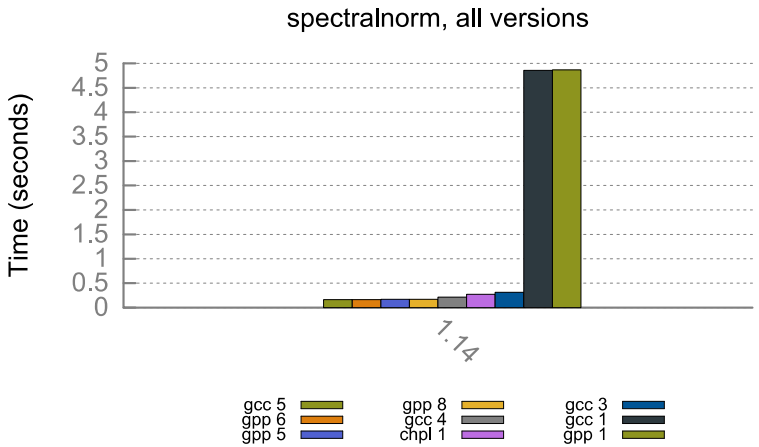


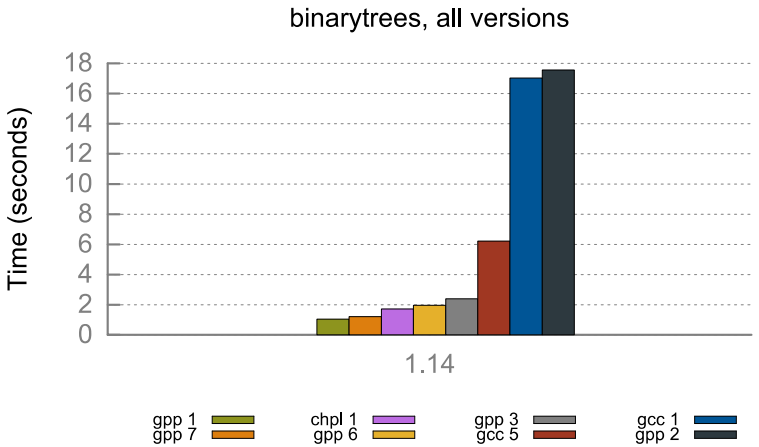


CLBG: Comparisons to C/C++

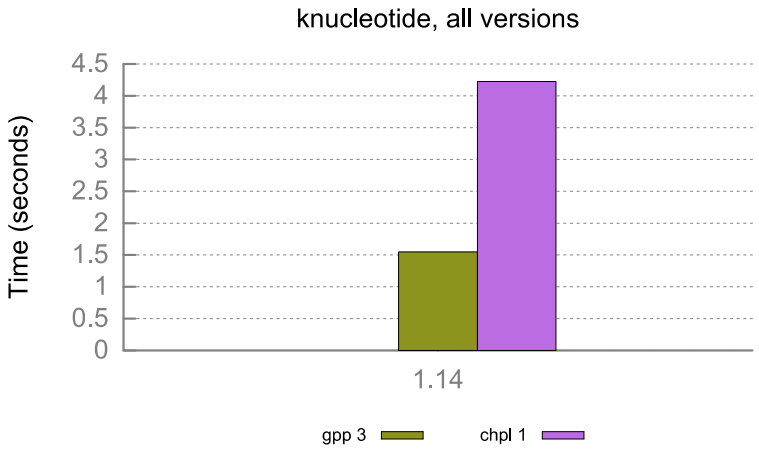


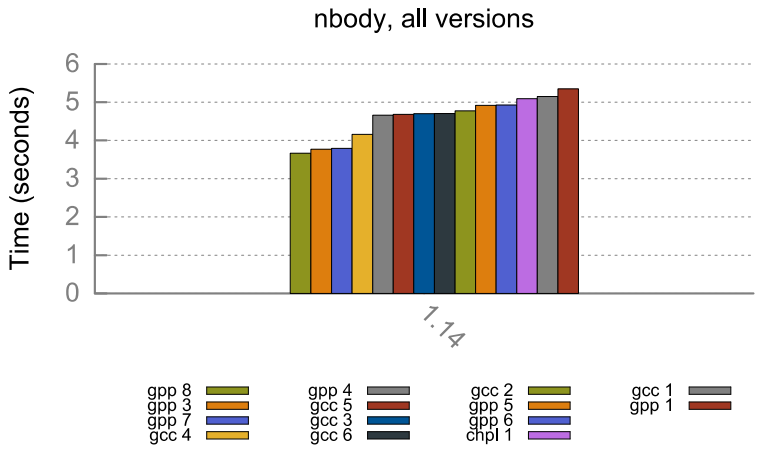


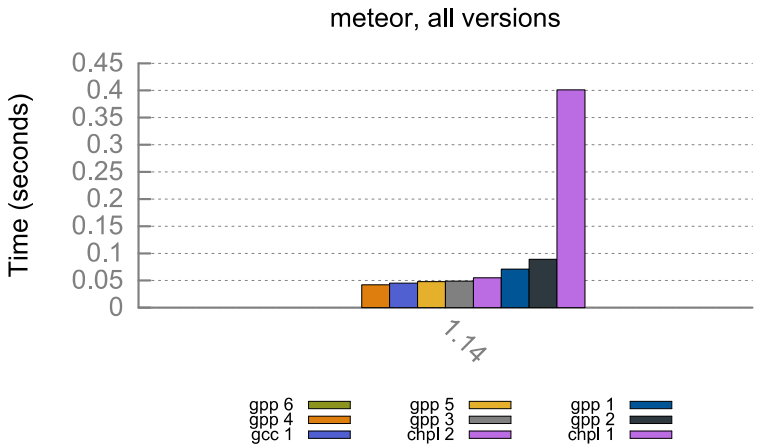


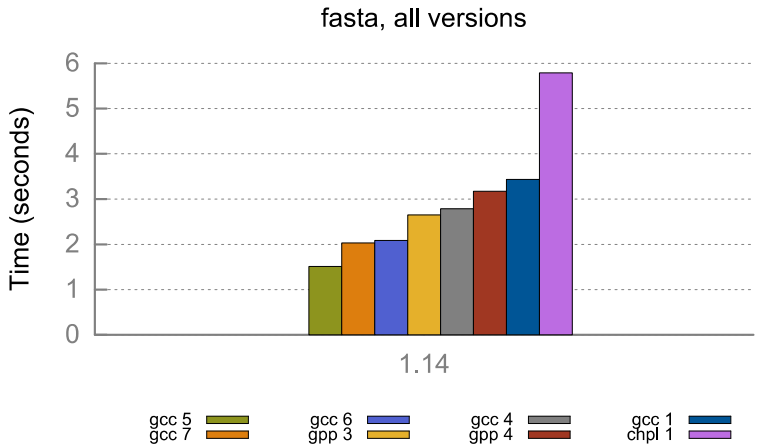


CLBG: Comparisons to C/C++









CLBG: Future Work

- **Continue improving Chapel and our entries:**
  - **fasta:** parallelize and optimize our current version
  - **string benchmarks:** improve string operations and performance
  - **pidigits:** optimize “assign returned record” idioms
  - **meteor:** reduce startup time in qthreads/hwloc
  - **n-body:** enable vectorization
  - **k-nucleotide:** improve associative domain performance and features
  - **mandelbrot:** consider adding ‘unroll’ keyword to for loops
  - continue to study outliers and work on improving them
- **Publish studies that dive beyond the superficial**
  - e.g., show heroism of top versions, compare with more typical ones
- **Consider submitting more heroic Chapel versions**
- **Don’t lose sight of multi-locale performance work**
  - encourage HPC community to establish a CLBG equivalent

Other Benchmark Improvements

Other Benchmark Improvements

- **Switched ISx to use the low-level PCG interface**
  - results in identical data sets as the reference version

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

THE SUPERCOMPUTER COMPANY