Chapel Background & Overview

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.

Motivation for Chapel

Q: Can a single language be…

…as productive as Python?
…as fast as Fortran?
…as portable as C?
…as scalable as MPI?
…as fun as <your favorite language here>?

A: We believe so.

The Challenge

Q: So why don't we have such languages already?

~~**A: Technical challenges?**~~

- while they exist, we don't think this is the main issue…

A: Due to a lack of…

…long-term efforts
…resources
…community will
…co-design between developers and users
…patience

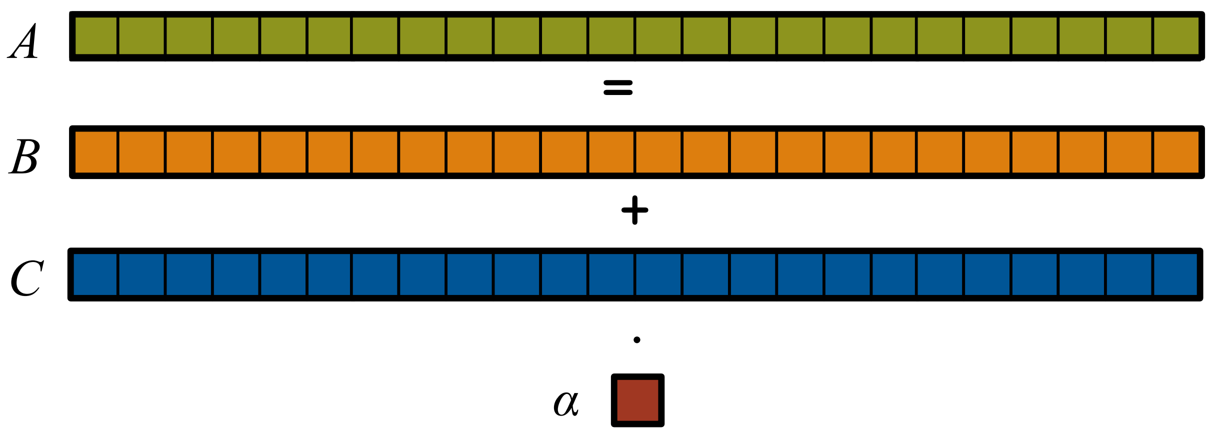
Chapel is our attempt to reverse this trend

STREAM Triad: a trivial parallel computation

Given: m-element vectors A, B, C

Compute: $\forall i \in 1..m,\ A_i = B_i + \alpha \cdot C_i$

In pictures:

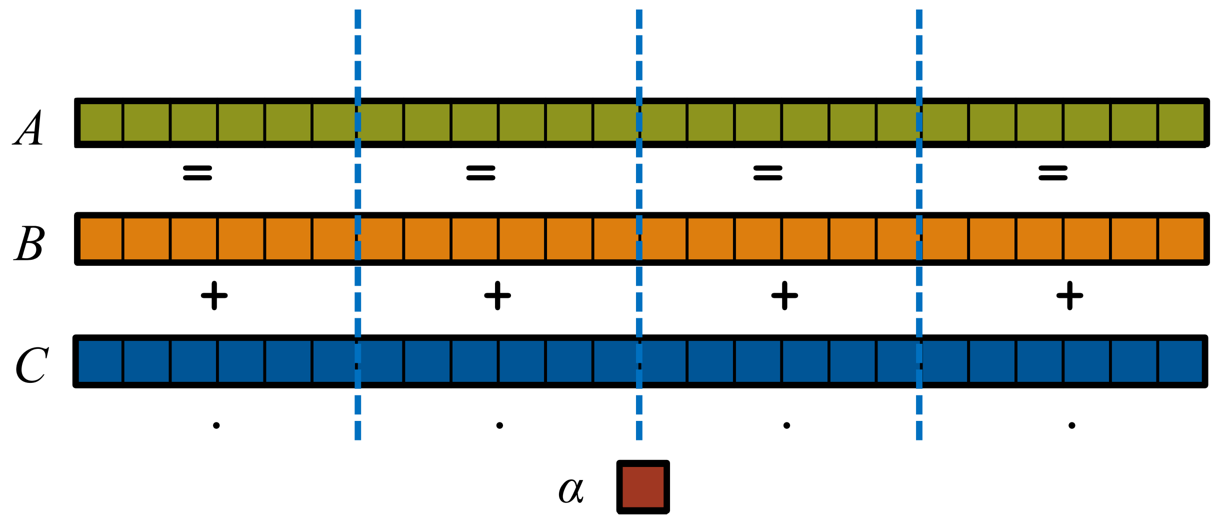


STREAM Triad: a trivial parallel computation

Given: m-element vectors A, B, C

Compute: $\forall i \in 1..m, A_i = B_i + \alpha \cdot C_i$

In pictures, in parallel:

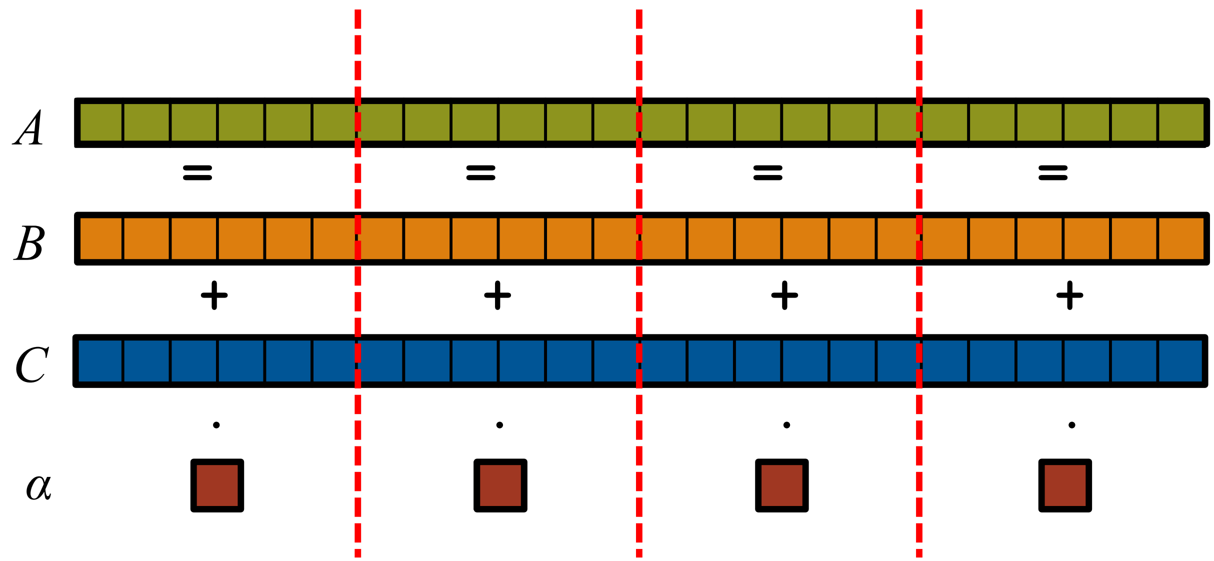


STREAM Triad: a trivial parallel computation

Given: m-element vectors A, B, C

Compute: $\forall i \in 1..m, A_i = B_i + \alpha \cdot C_i$

In pictures, in parallel (distributed memory):

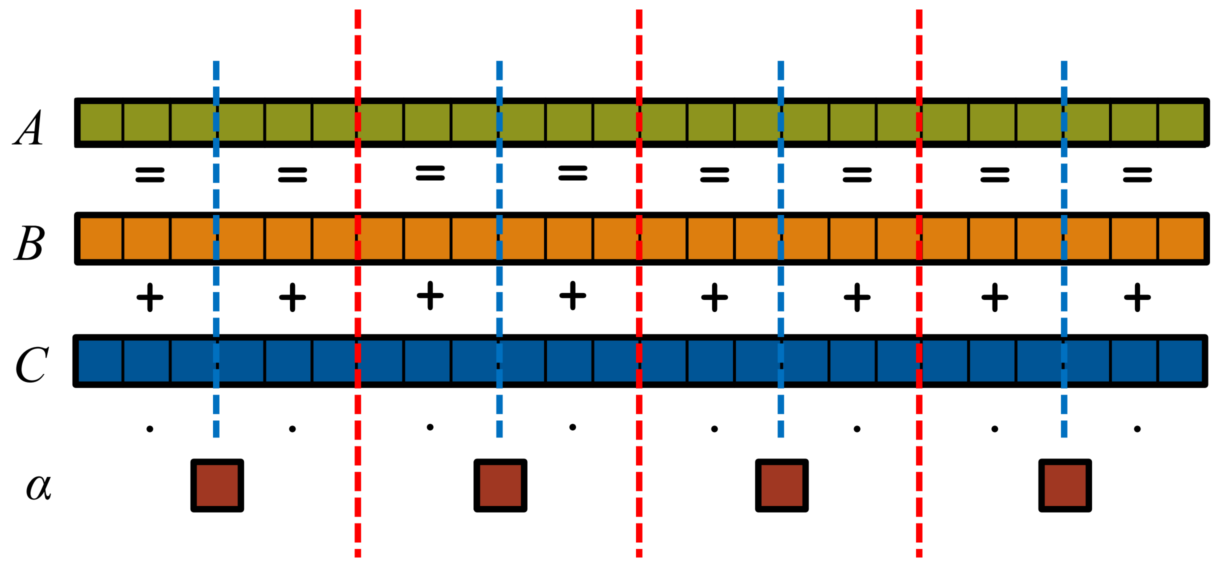


STREAM Triad: a trivial parallel computation

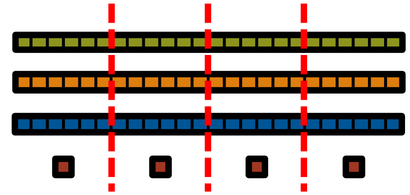
Given: m-element vectors A, B, C

Compute: $\forall i \in 1..m,\ A_i = B_i + \alpha \cdot C_i$

In pictures, in parallel (distributed memory multicore):



STREAM Triad: MPI

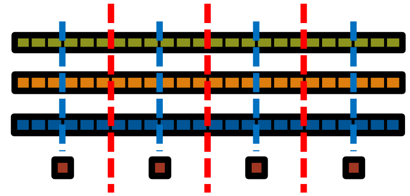


MPI

```c
#include <hpcc.h>



static int VectorSize;
static double *a, *b, *c;

int HPCC_StarStream(HPCC_Params *params) {
  int myRank, commSize;
  int rv, errCount;
  MPI_Comm comm = MPI_COMM_WORLD;

  MPI_Comm_size( comm, &commSize );
  MPI_Comm_rank( comm, &myRank );

  rv = HPCC_Stream( params, 0 == myRank);
  MPI_Reduce( &rv, &errCount, 1, MPI_INT, MPI_SUM,
    0, comm );

  return errCount;
}

int HPCC_Stream(HPCC_Params *params, int doIO) {
  register int j;
  double  scalar;

  VectorSize = HPCC_LocalVectorSize( params, 3,
    sizeof(double), 0 );

  a = HPCC_XMALLOC( double, VectorSize );
  b = HPCC_XMALLOC( double, VectorSize );
  c = HPCC_XMALLOC( double, VectorSize );
```

```c
if (!a || !b || !c) {
  if (c) HPCC_free(c);
  if (b) HPCC_free(b);
  if (a) HPCC_free(a);
  if (doIO) {
    fprintf( outFile, "Failed to allocate memory
  (%d).\n", VectorSize );
    fclose( outFile );
  }
  return 1;
}



for (j=0; j<VectorSize; j++) {
  b[j] = 2.0;
  c[j] = 1.0;
}

scalar = 3.0;



for (j=0; j<VectorSize; j++)
  a[j] = b[j]+scalar*c[j];

HPCC_free(c);
HPCC_free(b);
HPCC_free(a);
```

STREAM Triad: MPI+OpenMP



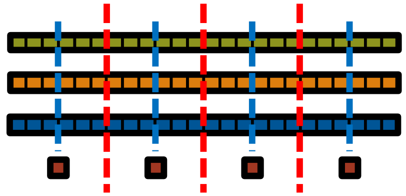
MPI + OpenMP

```c
#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif

static int VectorSize;
static double *a, *b, *c;

int HPCC_StarStream(HPCC_Params *params) {
  int myRank, commSize;
  int rv, errCount;
  MPI_Comm comm = MPI_COMM_WORLD;

  MPI_Comm_size( comm, &commSize );
  MPI_Comm_rank( comm, &myRank );

  rv = HPCC_Stream( params, 0 == myRank);
  MPI_Reduce( &rv, &errCount, 1, MPI_INT, MPI_SUM,
    0, comm );

  return errCount;
}

int HPCC_Stream(HPCC_Params *params, int doIO) {
  register int j;
  double  scalar;

  VectorSize = HPCC_LocalVectorSize( params, 3,
    sizeof(double), 0 );

  a = HPCC_XMALLOC( double, VectorSize );
  b = HPCC_XMALLOC( double, VectorSize );
  c = HPCC_XMALLOC( double, VectorSize );
```

```c
  if (!a || !b || !c) {
    if (c) HPCC_free(c);
    if (b) HPCC_free(b);
    if (a) HPCC_free(a);
    if (doIO) {
      fprintf( outFile, "Failed to allocate memory
    (%d).\n", VectorSize );
      fclose( outFile );
    }
    return 1;
  }

#ifdef _OPENMP
#pragma omp parallel for
#endif
  for (j=0; j<VectorSize; j++) {
    b[j] = 2.0;
    c[j] = 1.0;
  }

  scalar = 3.0;

#ifdef _OPENMP
#pragma omp parallel for
#endif
  for (j=0; j<VectorSize; j++)
    a[j] = b[j]+scalar*c[j];

  HPCC_free(c);
  HPCC_free(b);
  HPCC_free(a);
```

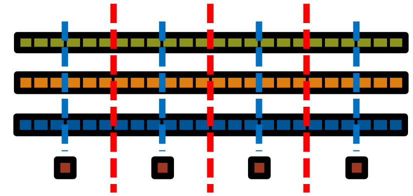
STREAM Triad: MPI+OpenMP vs. CUDA

MPI + OpenMP



```c
#ifdef _OPENMP
#include <omp.h>
#endif

static int VectorSize;
static double *a, *b, *c;

int HPCC_StarStream(HPCC_Params *params) {
  int myRank, commSize;
  int rv, errCount;
  MPI_Comm comm = MPI_COMM_WORLD;

  MPI_Comm_size( comm, &commSize );
  MPI_Comm_rank( comm, &myRank );

  rv = HPCC_Stream( params, 0 == myRank);
  MPI_Reduce( &rv, &errCount, 1, MPI_INT, MPI_SUM, 0, comm );

  return errCount;
}

int HPCC_Stream(HPCC_Params *params, int doIO) {
  register int j;
  double  scalar;

  VectorSize = HPCC_LocalVectorSize( params, 3, sizeof(double), 0 );

  a = HPCC_XMALLOC( double, VectorSize );
  b = HPCC_XMALLOC( double, VectorSize );
  c = HPCC_XMALLOC( double, VectorSize );

  if (!a || !b || !c) {
    if (c) HPCC_free(c);
    if (b) HPCC_free(b);
    if (a) HPCC_free(a);
    if (doIO) {
      fprintf( outFile, "Failed to allocate memory (%d).\n", VectorSize );
      fclose( outFile );
    }
    return 1;
  }

#ifdef _OPENMP
#pragma omp parallel for
#endif
  for (j=0; j<VectorSize; j++) {
    b[j] = 2.0;
    c[j] = 1.0;
  }

  scalar = 3.0;

#ifdef _OPENMP
#pragma omp parallel for
#endif
  for (j=0; j<VectorSize; j++)
    a[j] = b[j]+scalar*c[j];

  HPCC_free(c);
  HPCC_free(b);
  HPCC_free(a);

  return 0;
}
```

CUDA



```c
#define N       2000000

int main() {
  float *d_a, *d_b, *d_c;
  float scalar;

  cudaMalloc((void**)&d_a, sizeof(float)*N);
  cudaMalloc((void**)&d_b, sizeof(float)*N);
  cudaMalloc((void**)&d_c, sizeof(float)*N);

  dim3 dimBlock(128);
  dim3 dimGrid(N/dimBlock.x );
  if( N % dimBlock.x != 0 ) dimGrid

  set_array<<<dimGrid,dimBlock>>>(d_b, .5f, N);
  set_array<<<dimGrid,dimBlock>>>(d_c, .5f, N);

  scalar=3.0f;
  STREAM_Triad<<<dimGrid,dimBlock>>>(d_b, d_c, d_a, scalar,  N);
  cudaThreadSynchronize();

  cudaFree(d_a);
  cudaFree(d_b);
  cudaFree(d_c);
}

__global__ void set_array(float *a,  float value, int len) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if (idx < len) a[idx] = value;
}

__global__ void STREAM_Triad( float *a, float *b, float *c,
                              float scalar, int len) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if (idx < len) c[idx] = a[idx]+scalar*b[idx];
}
```

HPC suffers from too many distinct notations for expressing parallelism and locality

Why so many programming models?

HPC tends to approach programming models bottom-up:

Given a system and its core capabilities…

…provide features that can access the available performance.

- portability, generality, programmability: not strictly necessary.

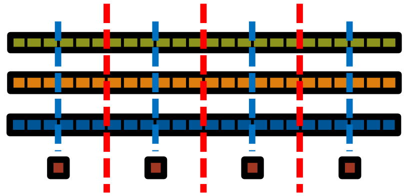
Type of HW Parallelism	Programming Model	Unit of Parallelism
Inter-node	MPI	executable
Intra-node/multicore	OpenMP / pthreads	iteration/task
Instruction-level vectors/threads	pragmas	iteration
GPU/accelerator	CUDA / Open[MP|CL|ACC]	SIMD function/task

benefits: lots of control; decent generality; easy to implement

downsides: lots of user-managed detail; brittle to changes

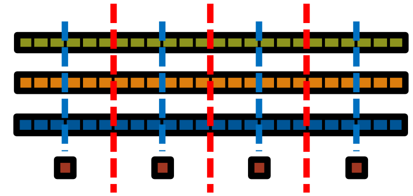
Rewinding a few slides…

MPI + OpenMP



```
#ifdef _OPENMP
#include <omp.h>
#endif

static int VectorSize;
static double *a, *b, *c;

int HPCC_StarStream(HPCC_Params *params) {
  int myRank, commSize;
  int rv, errCount;
  MPI_Comm comm = MPI_COMM_WORLD;

  MPI_Comm_size( comm, &commSize );
  MPI_Comm_rank( comm, &myRank );

  rv = HPCC_Stream( params, 0 == myRank);
  MPI_Reduce( &rv, &errCount, 1, MPI_INT, MPI_SUM, 0, comm );

  return errCount;
}

int HPCC_Stream(HPCC_Params *params, int doIO) {
  register int j;
  double  scalar;

  VectorSize = HPCC_LocalVectorSize( params, 3, sizeof(double), 0 );

  a = HPCC_XMALLOC( double, VectorSize );
  b = HPCC_XMALLOC( double, VectorSize );
  c = HPCC_XMALLOC( double, VectorSize );

  if (!a || !b || !c) {
    if (c) HPCC_free(c);
    if (b) HPCC_free(b);
    if (a) HPCC_free(a);
    if (doIO) {
      fprintf( outFile, "Failed to allocate memory (%d).\n", VectorSize );
      fclose( outFile );
    }
    return 1;
  }

#ifdef _OPENMP
#pragma omp parallel for
#endif
  for (j=0; j<VectorSize; j++) {
    b[j] = 2.0;
    c[j] = 1.0;
  }

  scalar = 3.0;

#ifdef _OPENMP
#pragma omp parallel for
#endif
  for (j=0; j<VectorSize; j++)
    a[j] = b[j]+scalar*c[j];

  HPCC_free(c);
  HPCC_free(b);
  HPCC_free(a);

  return 0;
}
```

CUDA



```
#define N        2000000

int main() {
  float *d_a, *d_b, *d_c;
  float scalar;

  cudaMalloc((void**)&d_a, sizeof(float)*N);
  cudaMalloc((void**)&d_b, sizeof(float)*N);
  cudaMalloc((void**)&d_c, sizeof(float)*N);

  dim3 dimBlock(128);
  dim3 dimGrid(N/dimBlock.x [illegible]
  if( N % dimBlock.x != 0 ) dimGrid [illegible]

  set_array<<<dimGrid,dimBlock>>>(d_b, .5f, N);
  set_array<<<dimGrid,dimBlock>>>(d_c, .5f, N);

  scalar=3.0f;
  STREAM_Triad<<<dimGrid,dimBlock>>>(d_b, d_c, d_a, scalar,  N);
  cudaThreadSynchronize();

  cudaFree(d_a);
  cudaFree(d_b);
  cudaFree(d_c);
}

__global__ void set_array(float *a,  float value, int len) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if (idx < len) a[idx] = value;
}

__global__ void STREAM_Triad( float *a, float *b, float *c,
                              float scalar, int len) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if (idx < len) c[idx] = a[idx]+scalar*b[idx];
}
```

HPC suffers from too many distinct notations for expressing parallelism and locality

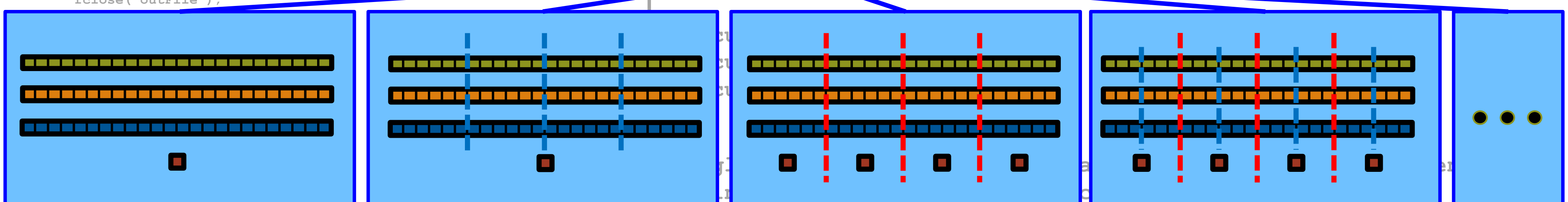
STREAM Triad: Chapel

MPI + OpenMP

Chapel

```
config const m = 1000,
             alpha = 3.0;

const ProblemSpace = {1..m} dmapped …;

var A, B, C: [ProblemSpace] real;

B = 2.0;
C = 1.0;

A = B + alpha * C;
```

the special sauce



Philosophy: Good, *top-down* language design can tease system-specific implementation details away from an algorithm, permitting the compiler, runtime, applied scientist, and HPC expert to each focus on their strengths.

What is Chapel?

Chapel: A productive parallel programming language

- portable
- open-source
- a collaborative effort

Goals:

- Support general parallel programming
  - “any parallel algorithm on any parallel hardware”
- Make parallel programming at scale far more productive

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"~~ want full control to ensure performance"

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 is Portable

- **Chapel is designed to be hardware-independent**

- **The current release requires:**
  - a C/C++ compiler
  - a *NIX environment (Linux, OS X, BSD, Cygwin, …)
  - POSIX threads
  - UDP, MPI, or RDMA (if distributed memory execution is desired)

- **Chapel can run on…**

  …laptops and workstations
  …commodity clusters
  …the cloud
  …HPC systems from Cray and other vendors
  …modern processors like Intel Xeon Phi, GPUs*, etc.

* = academic work only; not yet supported in the official release

Chapel is Open-Source

- **Chapel's development is hosted at GitHub**
  - https://github.com/chapel-lang

- **Chapel is licensed as Apache v2.0 software**

- **Instructions for download + install are online**
  - see http://chapel.cray.com/download.html to get started

The Chapel Team at Cray (May 2016)

14 full-time employees + 2 summer interns + occasional visiting academics
(one of each started after photo taken)

Chapel Community R&D Efforts

(and several others…)

http://chapel.cray.com/collaborations.html

Outline

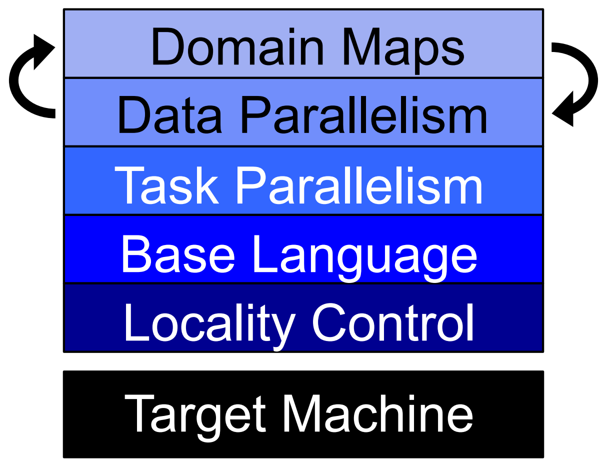
- ✓ Chapel Motivation and Background
- ➢ Chapel in a Nutshell
- Chapel Project: Past, Present, Future

Chapel's Multiresolution Philosophy

Multiresolution Design: **Support multiple tiers of features**

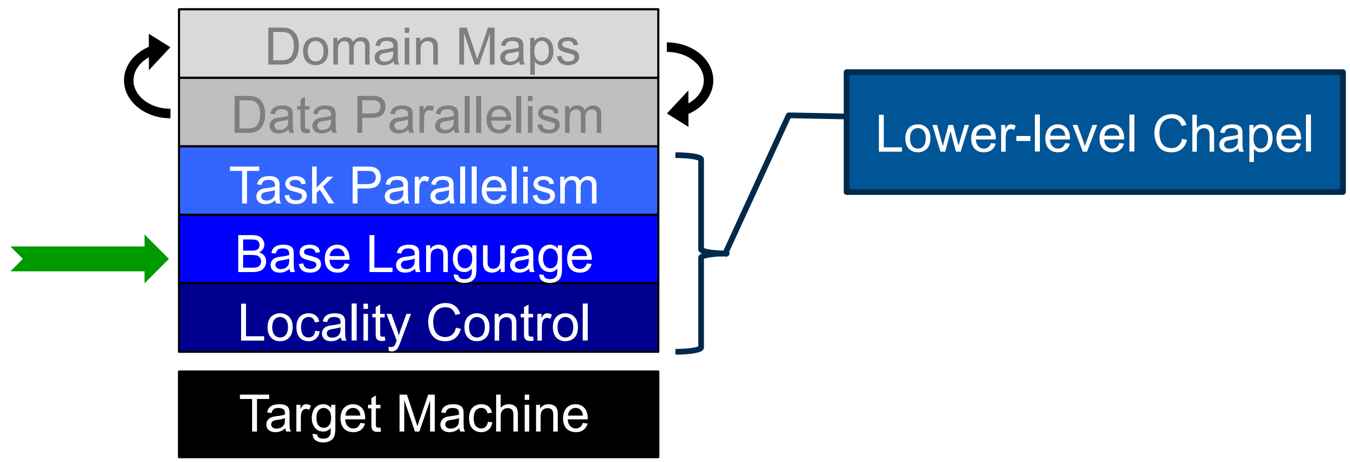
- higher levels for programmability, productivity
- lower levels for greater degrees of control

Chapel language concepts



- build the higher-level concepts in terms of the lower
- permit the user to intermix layers arbitrarily

Base Language



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

Modern iterators

```
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;
  }
}
```

Configuration declarations
(to avoid command-line argument parsing)
`./a.out --n=1000000`

```
config const n = 10;

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

```
0
1
1
2
3
5
8
…
```

Base Language Features, by example

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

```
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

Zippered iteration

```
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

Range types and operators

```
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

tuples

```
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
…
```

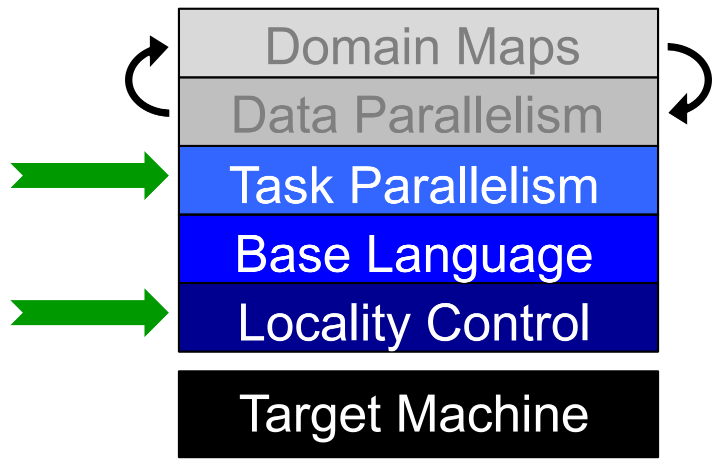
Base Language Features, by example

```
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
…
```

Task Parallelism



Task Parallelism, Locality Control, by example

taskParallel.chpl

```
coforall loc in Locales do
  on loc {
    const numTasks = here.maxTaskPar;
    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
```

Task Parallelism, Locality Control, by example

Abstraction of System Resources

taskParallel.chpl

```
coforall loc in Locales do
  on loc {
    const numTasks = here.maxTaskPar;
    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
```

Task Parallelism, Locality Control, by example

High-Level Task Parallelism

taskParallel.chpl

```
coforall loc in Locales do
  on loc {
    const numTasks = here.maxTaskPar;
    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
```

Task Parallelism, Locality Control, by example

Control of Locality/Affinity

taskParallel.chpl

```
coforall loc in Locales do
  on loc {
    const numTasks = here.maxTaskPar;
    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
```

Task Parallelism, Locality Control, by example

Abstraction of System Resources

taskParallel.chpl

```
coforall loc in Locales do
  on loc {
    const numTasks = here.maxTaskPar;
    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
```

Task Parallelism, Locality Control, by example

High-Level
Task Parallelism

taskParallel.chpl

```
coforall loc in Locales do
  on loc {
    const numTasks = here.maxTaskPar;
    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
```

Task Parallelism, Locality Control, by example

taskParallel.chpl

```
coforall loc in Locales do
  on loc {
    const numTasks = here.maxTaskPar;
    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
```

Not seen here:

Data-centric task coordination via atomic and full/empty vars

Task Parallelism, Locality Control, by example

taskParallel.chpl

```
coforall loc in Locales do
  on loc {
    const numTasks = here.maxTaskPar;
    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
```

Parallelism and Locality: Orthogonal in Chapel

- This is a **parallel**, but local program:

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

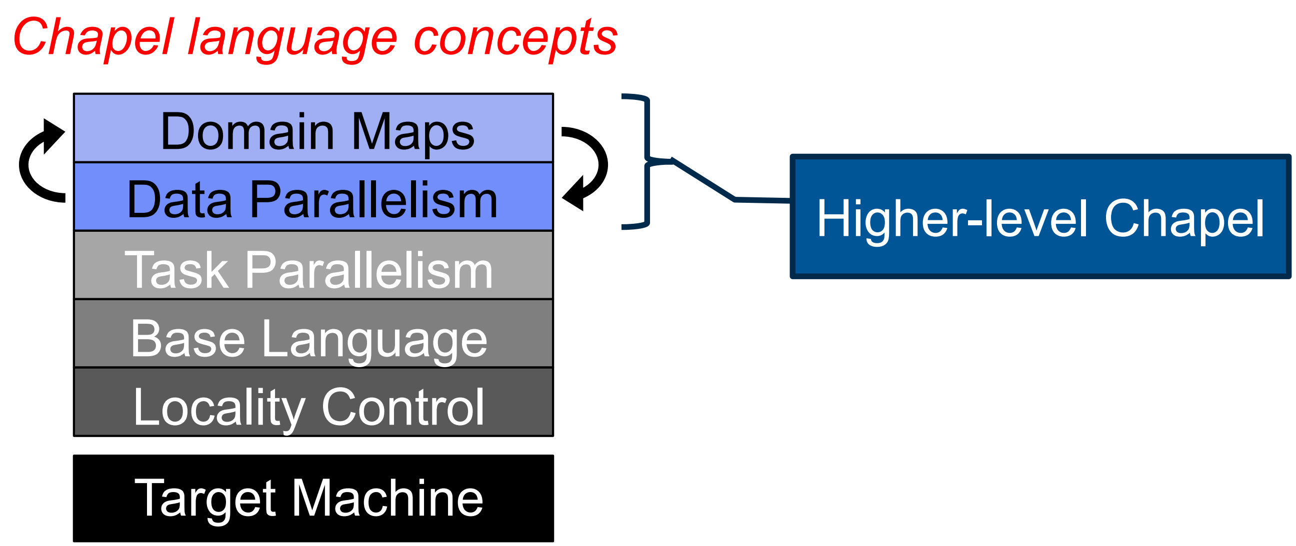
- This is a **distributed**, but serial program:

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

- This is a **distributed parallel** program:

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

Higher-Level Features



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 -o dataParallel
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

Domains (Index Sets)

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 -o dataParallel
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

Arrays

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 -o dataParallel
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

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 -o dataParallel
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

dataParallel.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);
```

Domain Maps
(Map Data Parallelism to the System)

```
prompt> chpl dataParallel.chpl -o dataParallel
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
```

Distributed Data Parallelism, by example

dataParallel.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 -o dataParallel
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
```

Outline

- ✓ Chapel Motivation and Background
- ✓ Chapel in a Nutshell
- ➢ Chapel Project: Past, Present, Future

Chapel's Origins: HPCS

DARPA HPCS: High Productivity Computing Systems

- **Goal:** improve productivity by a factor of 10x
- **Timeframe:** Summer 2002 – Fall 2012
- Cray developed a new system architecture, network, software stack…
  - this became the very successful Cray XC30™ Supercomputer Series

…and a new programming language: Chapel
(at that point, essentially a research prototype)

Chapel's 5-year push

- **Based on positive user response to Chapel under HPCS, Cray undertook a five-year effort to improve it**
  - we're just completing our fourth year

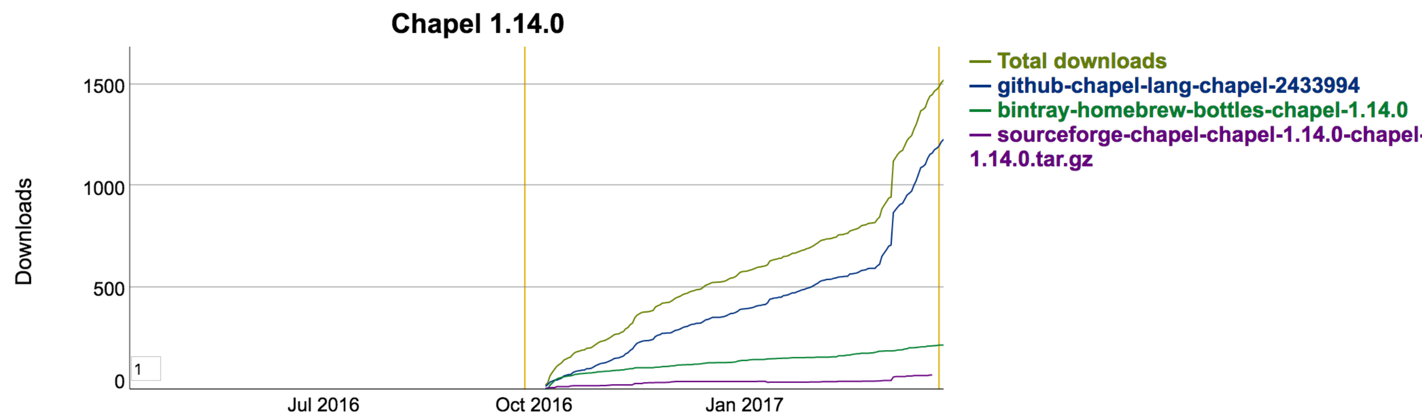
- **Focus Areas:**
  1. Improving **performance** and scaling
  2. **Fixing** immature aspects of the language and implementation
     - e.g., strings, memory management and leaks, OOP, error handling, …
  3. **Porting** to emerging architectures
     - Intel Xeon Phi, accelerators, heterogeneous processors and memories, …
  4. Improving **interoperability**
  5. Growing the Chapel user and developer **community**
     - including non-scientific computing communities
  6. Exploring transition of Chapel **governance** to a neutral, external body

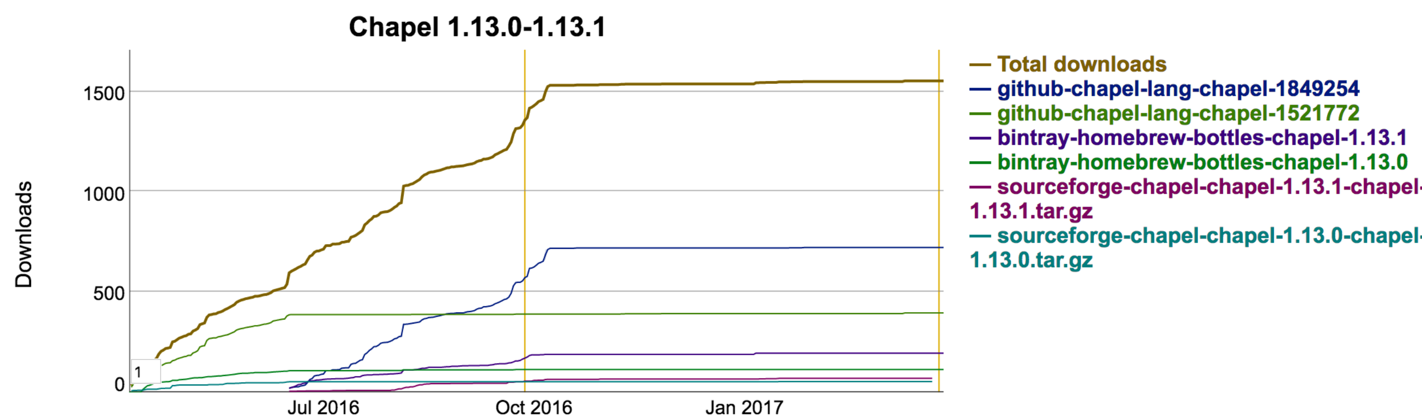
A Year in the Life of Chapel

- **Two major releases per year** (April / October)
  - **~a month later:** detailed release notes
  - **latest release:** Chapel 1.15, released April 6th 2017
    - release notes due to be published this week or next
- **CHIUW:** Chapel Implementers and Users Workshop (~June)
  - (4th annual) CHIUW 2017, June 1-2 at IPDPS (Orlando, FL)
  - talks from members of the broad community + a Chapel code camp
- **SC** (Nov)
  - tutorials, panels, BoFs, posters, educator sessions, exhibits, …
  - annual **CHUG (Chapel Users Group) happy hour**
- **Talks, tutorials, research visits, blog posts, …** (year-round)

Chapel is a Work-in-Progress

- **Currently being picked up by early adopters**
  - ~3000+ downloads per year across two releases





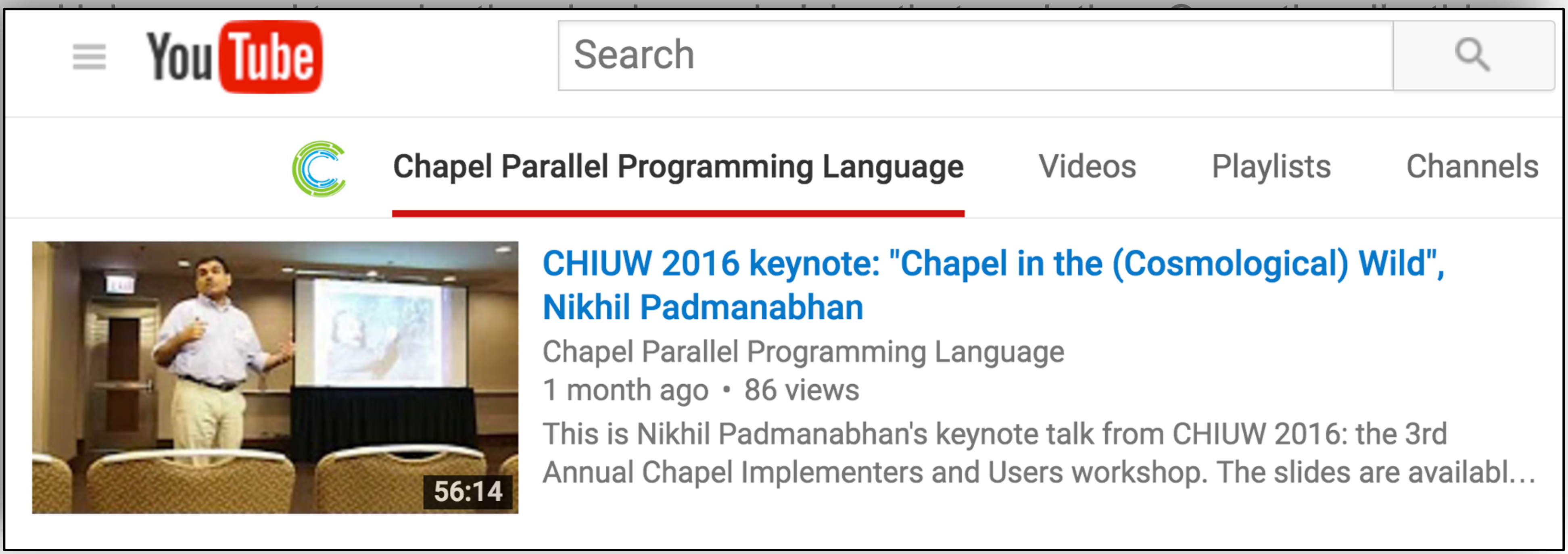
  - Users who try it generally like what they see

A notable early adopter

Chapel in the (Cosmological) Wild 1:00 – 2:00

Nikhil Padmanabhan, *Yale University Professor, Physics & Astronomy*

Abstract: This talk aims to present my personal experiences using Chapel in my research. My research interests are in observational cosmology; more specifically, I use large surveys of galaxies to constrain the evolution of the



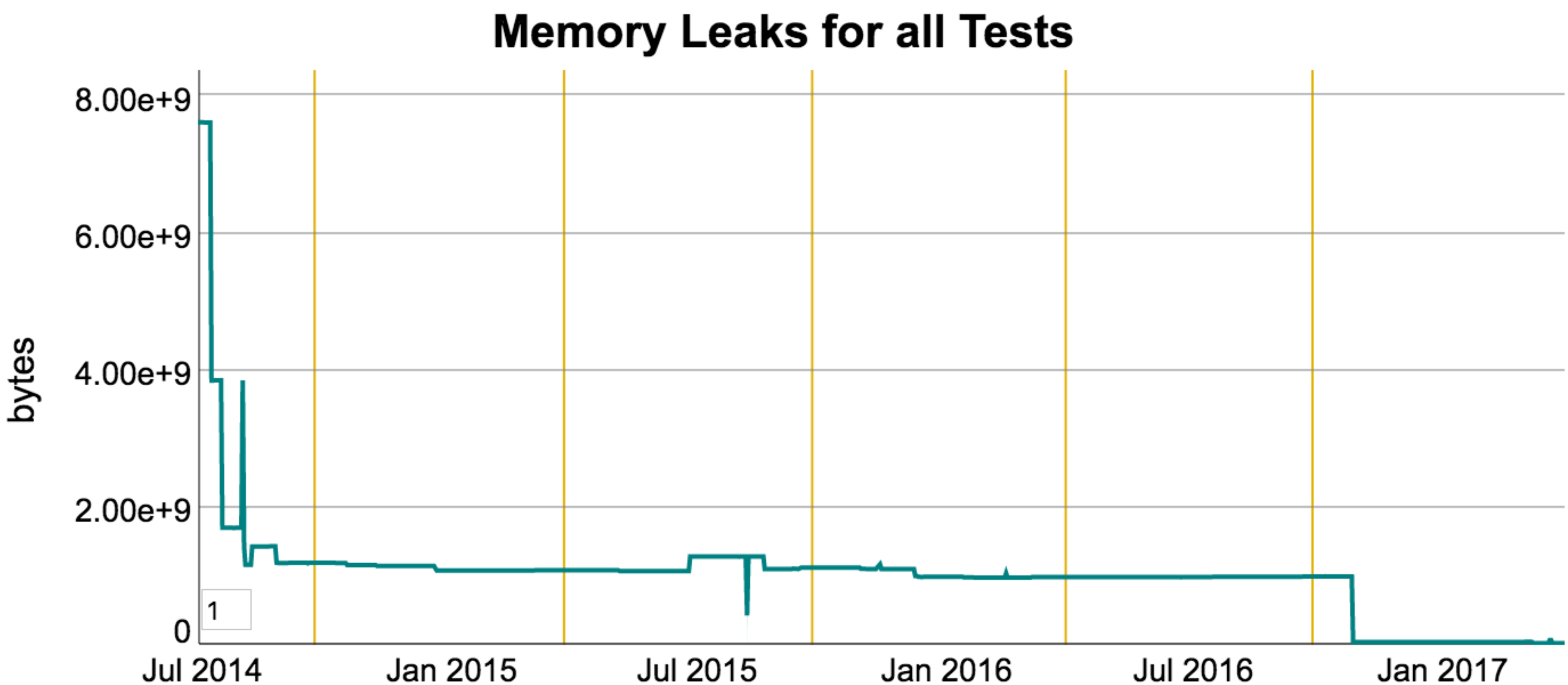
Chapel: Top 3 Historical Barriers to Use

3. Core Language Feature Improvements

- Historical problems that are now much better:
  - strings, memory leaks, memory management
- Areas that have improved, but are still in-progress:
  - initializers (constructor replacement), error-handling

Memory Leak Improvements

- **Effort in recent years has dramatically reduced leaks**
  - most remaining cases are due to user-level leaks in tests themselves



Chapel: Top 3 Traditional Barriers to Use

3. Core Language Feature Improvements

- Historical problems that are now much better:
  - strings, memory leaks, memory management, interoperability, generics
- Areas that have improved, but are still in-progress:
  - initializers (constructor replacement), error-handling

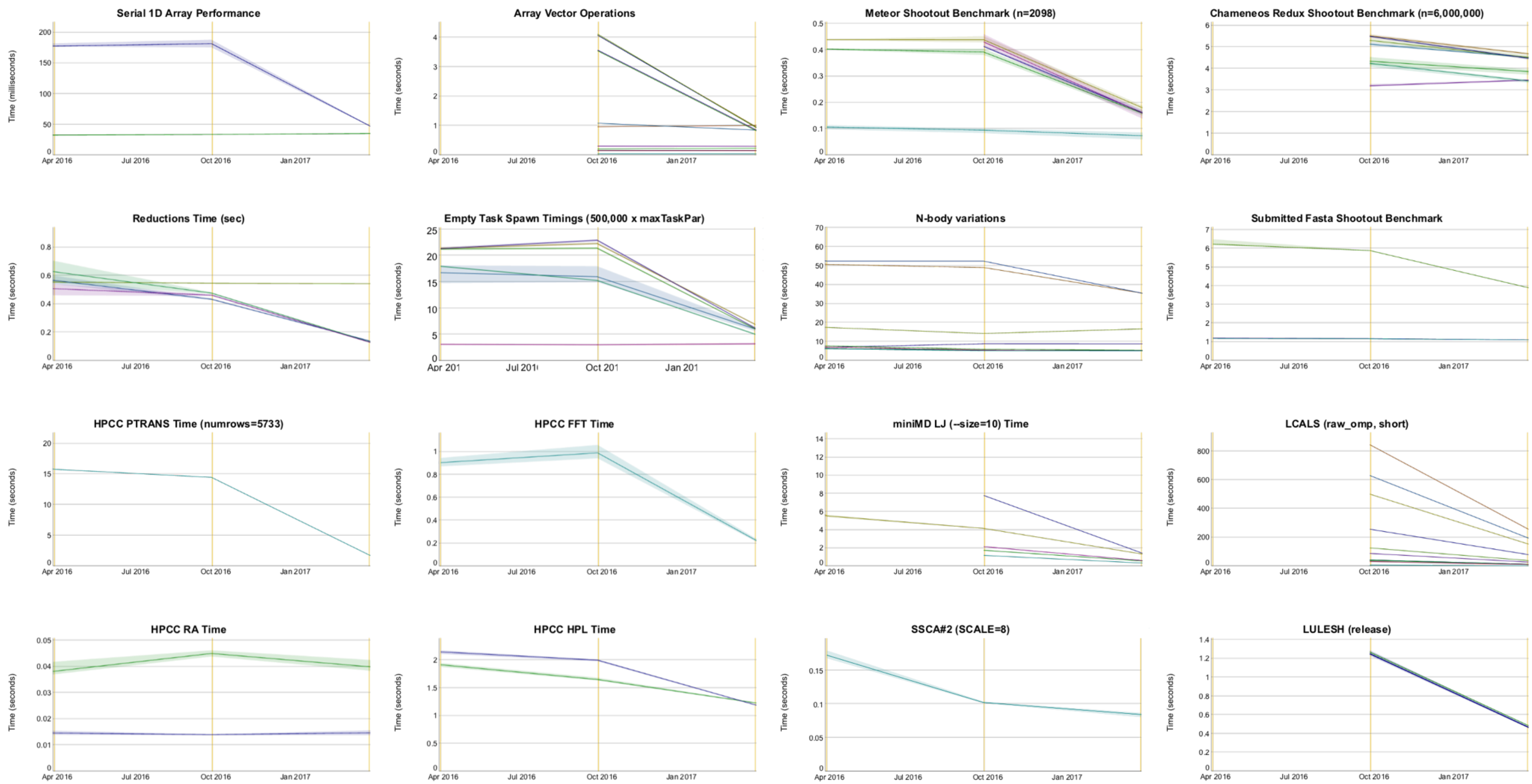
2. Access to Standard Libraries

- Situation has improved significantly over past few years
  - Several core libraries added:
    - BigInteger, BitOps, DateTime, FileSystem, Random, Reflection, Spawn, …
  - As well as access to many standard libraries / technologies:
    - BLAS, Curl, FFTW, Futures, HDFS, LAPACK, LinearAlgebra, MPI, ZMQ, …

1. Performance

- Particularly for old-school HPC users, performance is crucial
- That said, as of this month's release, we're reaching parity more often

Single-Locale Improvements in Execution Time



- Single-locale is increasingly on par with C / C++ / OpenMP

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	[illegible]	[illegible]	[illegible]	[illegible]	32% 38% 37% 39%
1.6	Go	[illegible]	[illegible]	[illegible]	[illegible]	00% 96% 56% 100%
1.6	**Chapel**	[illegible]	[illegible]	[illegible]	[illegible]	0% 100% 100% 100%
1.6	Chapel #2	[illegible]	[illegible]	[illegible]	[illegible]	99% 99% 98% 99%

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

CLBG: Chapel Standings as of Apr 20th

- **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
- **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

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

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, (
  const group2 = [i in 1..popSize2] new Chameneos(i, c

  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, c
  writeln();
}


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

  coforall c in population do          // create a ta
    c.haveMeetings(place, population);

  delete place;
}
```

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

```
proc holdMeetings(population, numMeetings) {
  const place = new MeetingPlace(numMeetings);

  coforall c in population do          // creat
    c.haveMeetings(place, population);

  delete place;
}
```

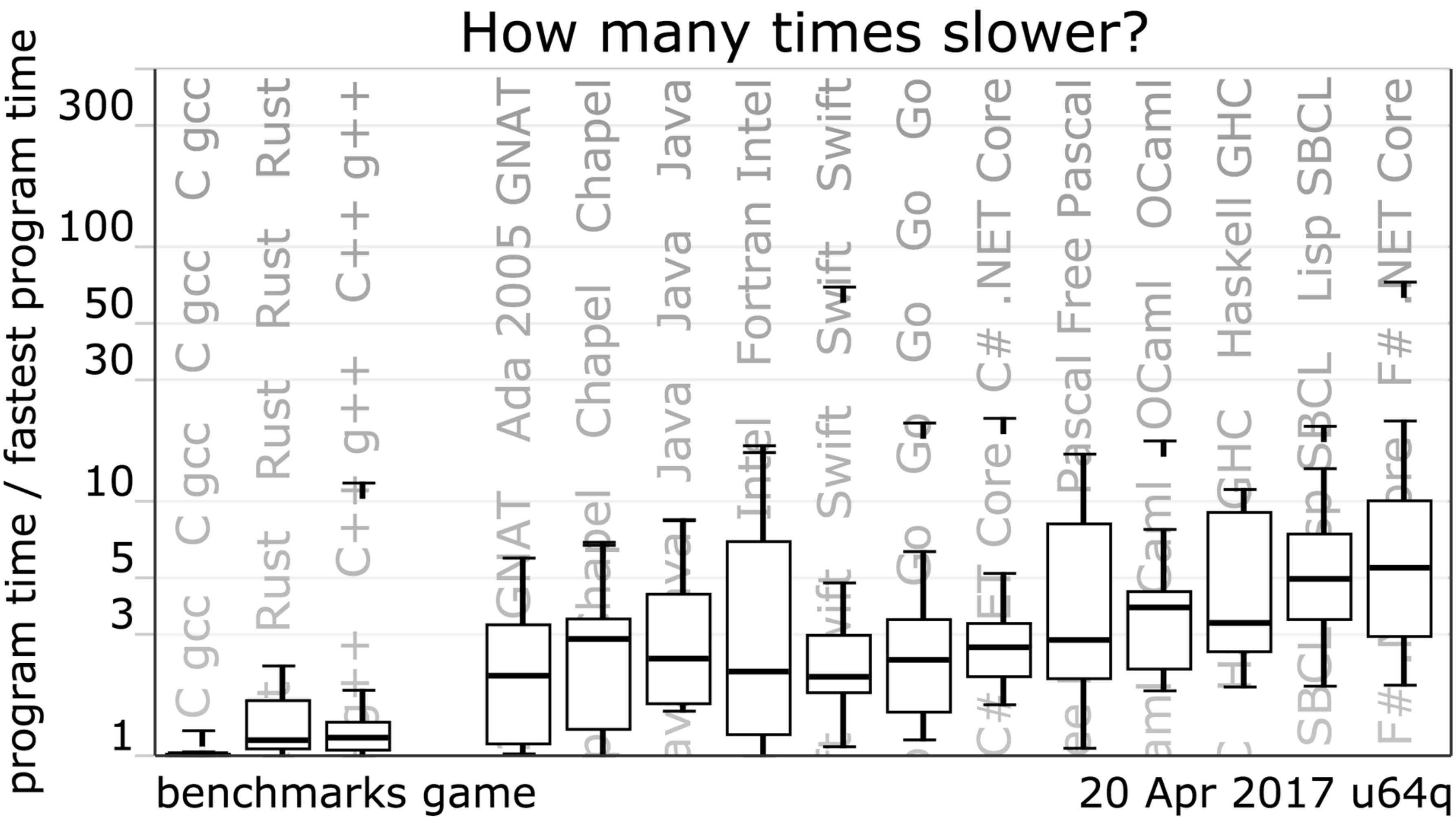
excerpt from 1210 gz Chapel #2 entry

```
void get_affinity(int* is_smp, cpu_set_t* affinity1, cpu_set_t* affinity2)

    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);

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

excerpt from 2863 gz C gcc #5 entry

CLBG: Website

Site summary: relative performance (sorted by geometric mean)



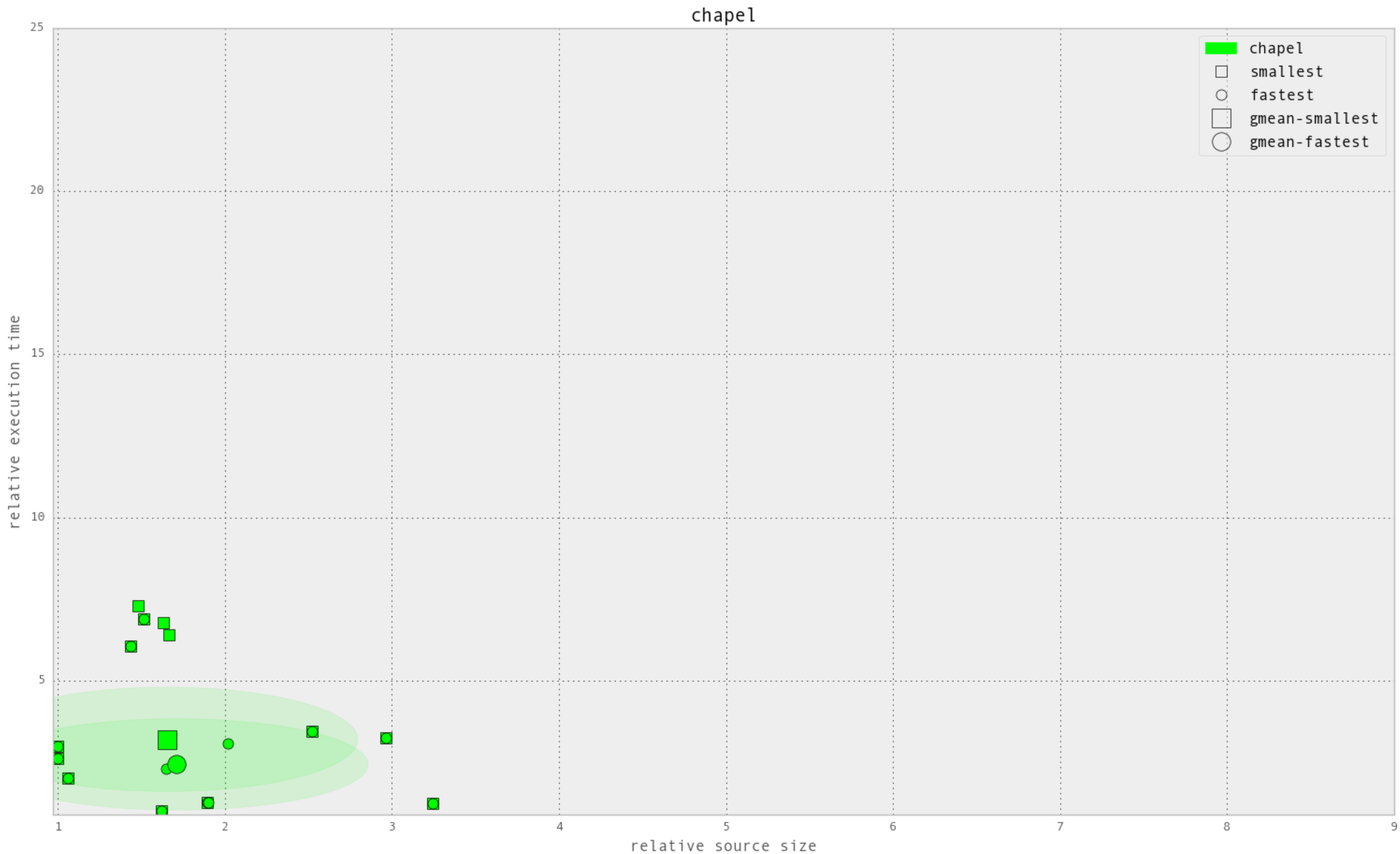
CLBG: Website

- **site has a sound philosophy about too-easy answers**

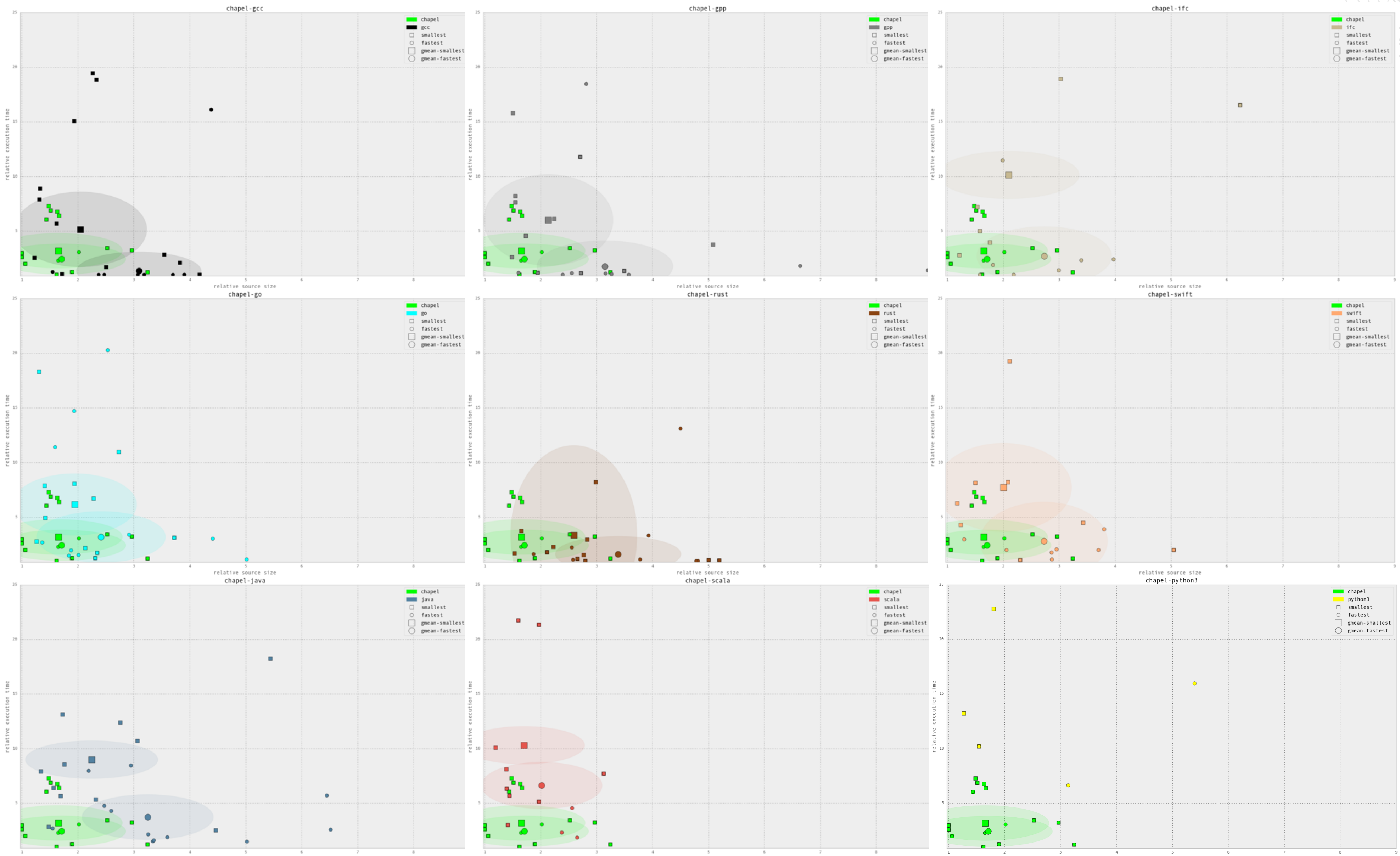
> **We want easy answers**, but easy answers are often incomplete or wrong. You and I know, there's more we should understand:
>
> stories   details   fast?   conclusions

- **yet, most readers probably still jump to conclusions**
  - execution time dominates default (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



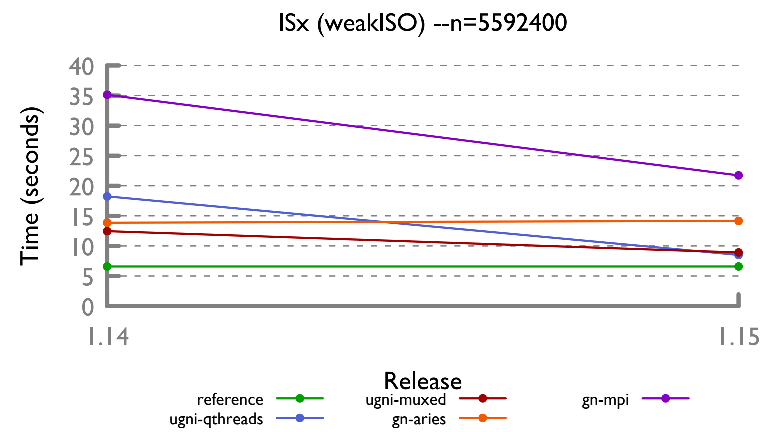
Chapel vs. 9 other languages

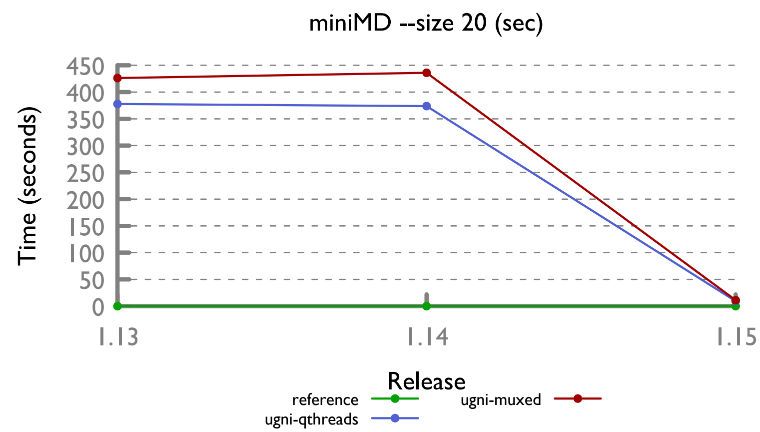


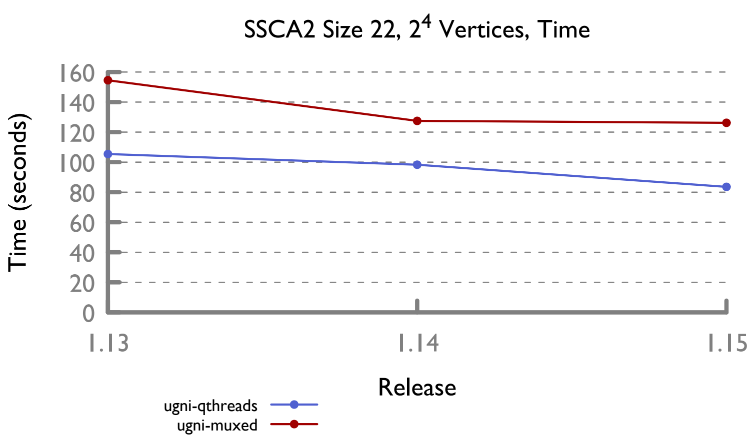
Chapel vs. 9 other languages (zoomed out)

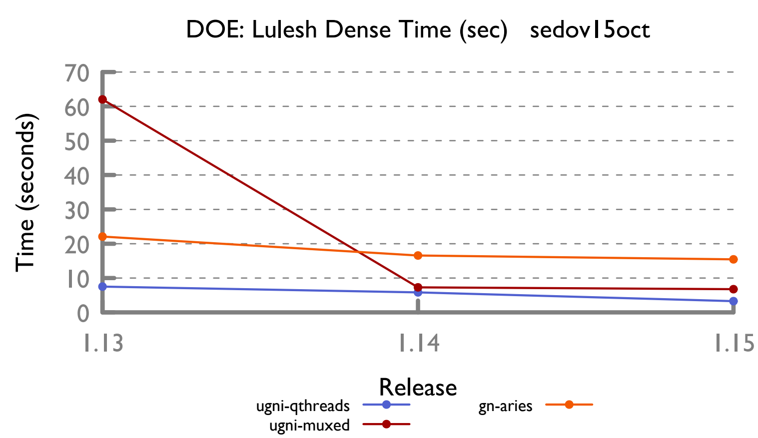
Multi-locale Improvements in Execution Time

- Multi-locale performance is improving significantly as well









3 Key Multi-Locale Communication Benchmarks

STREAM Triad:

- measures embarrassingly / pleasingly parallel computation

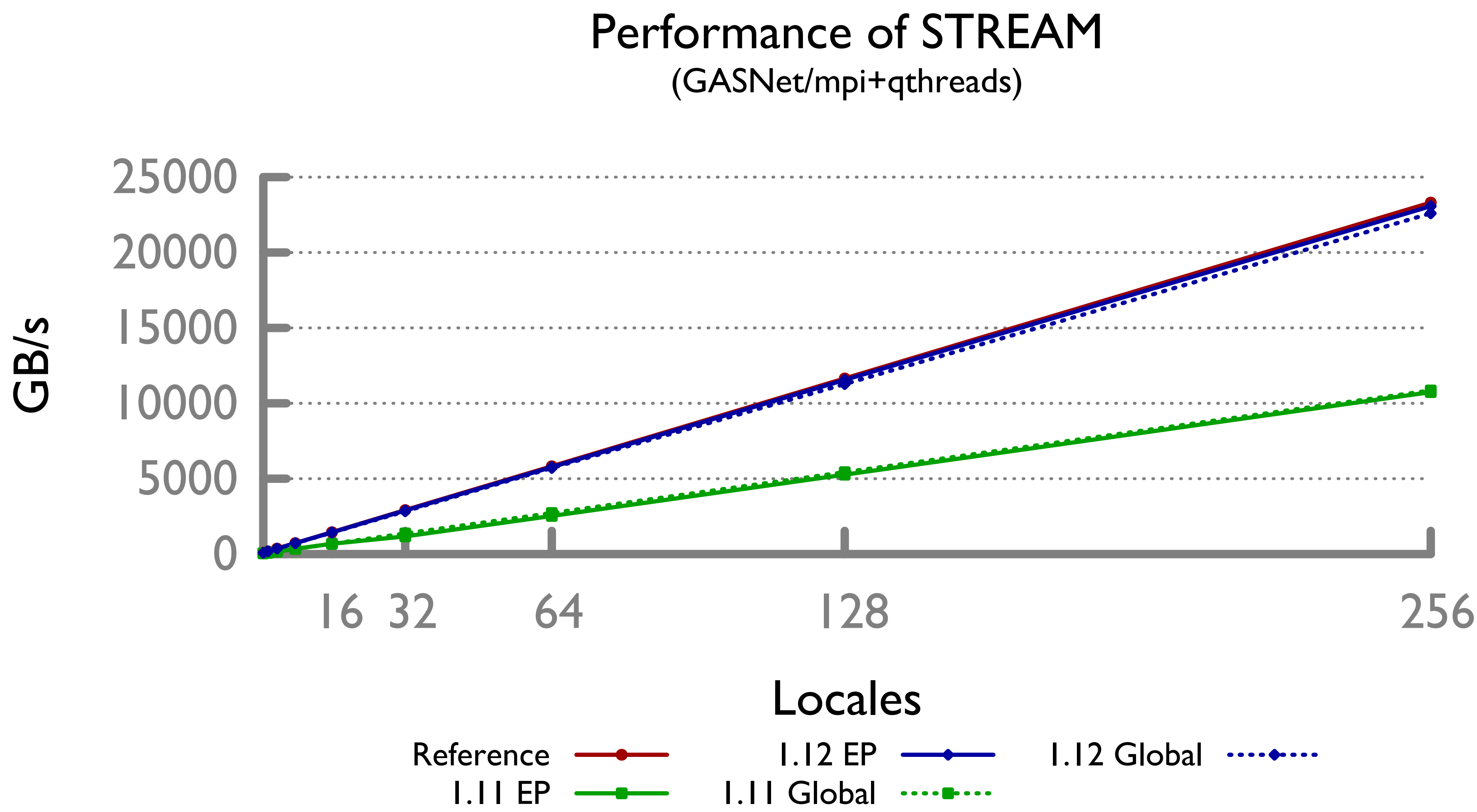
RA:

- measures random updates to a large distributed array

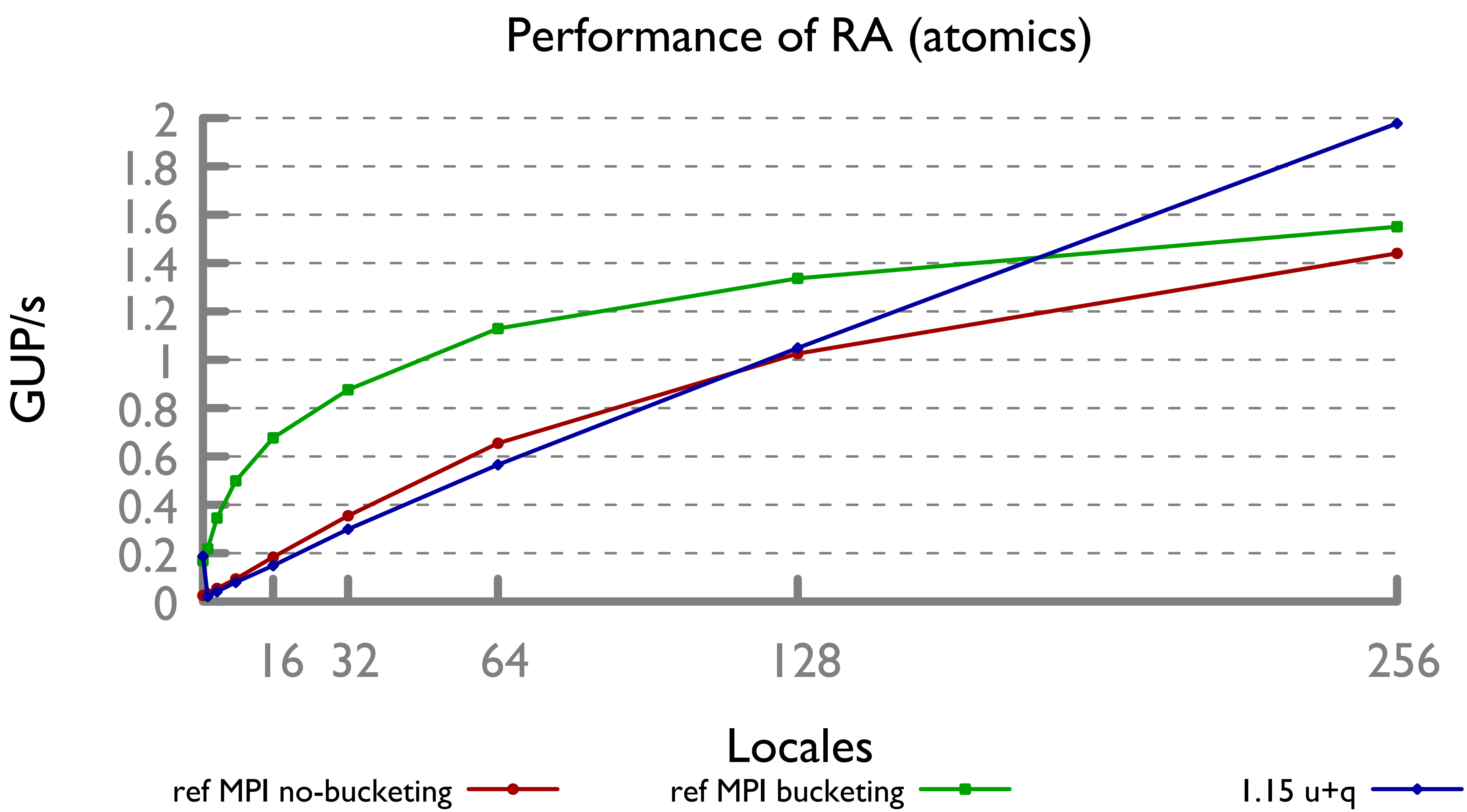
ISx:

- measures bucket-exchange idiom

STREAM Triad: Chapel vs. MPI Scalability

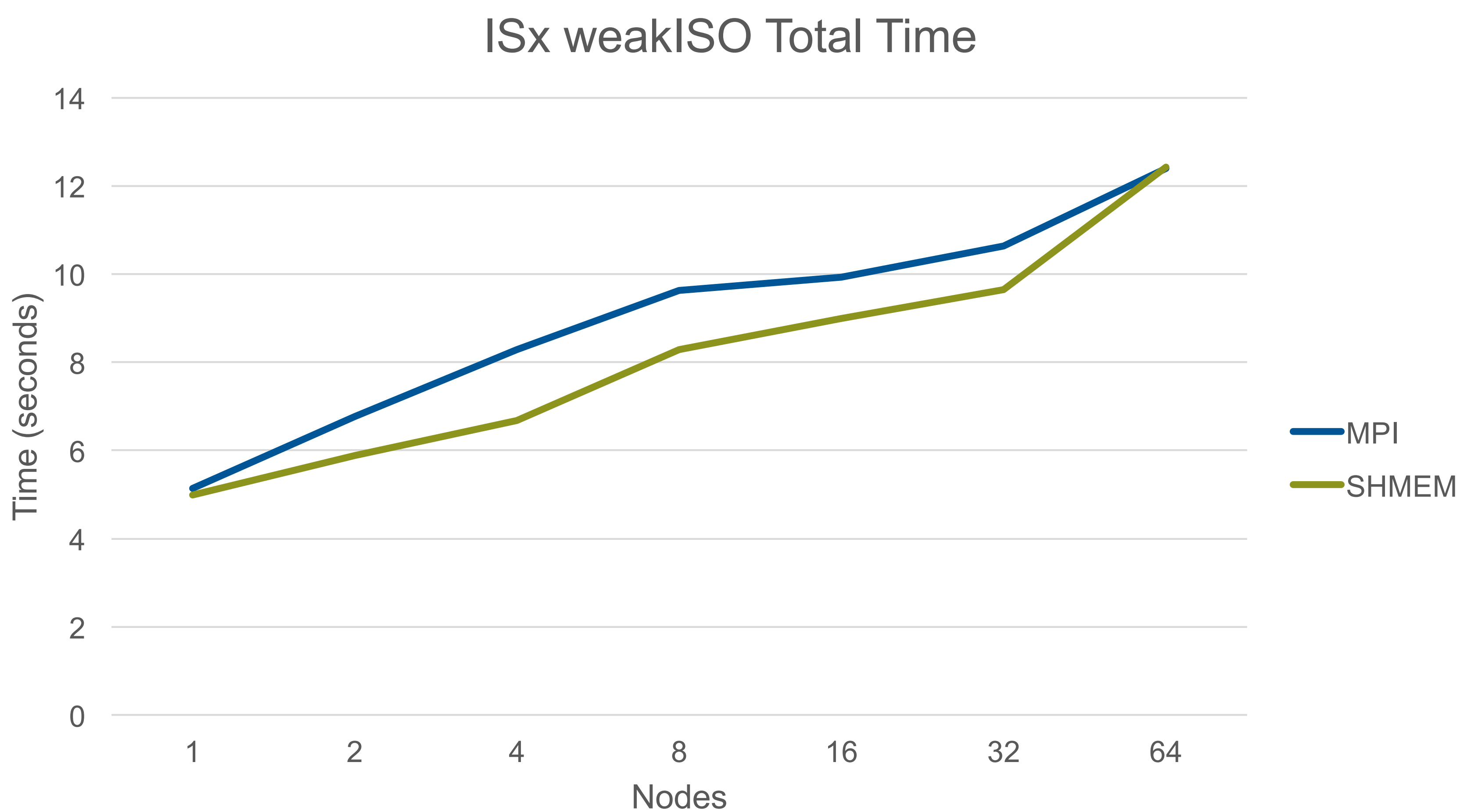


RA: Chapel vs. MPI Scalability



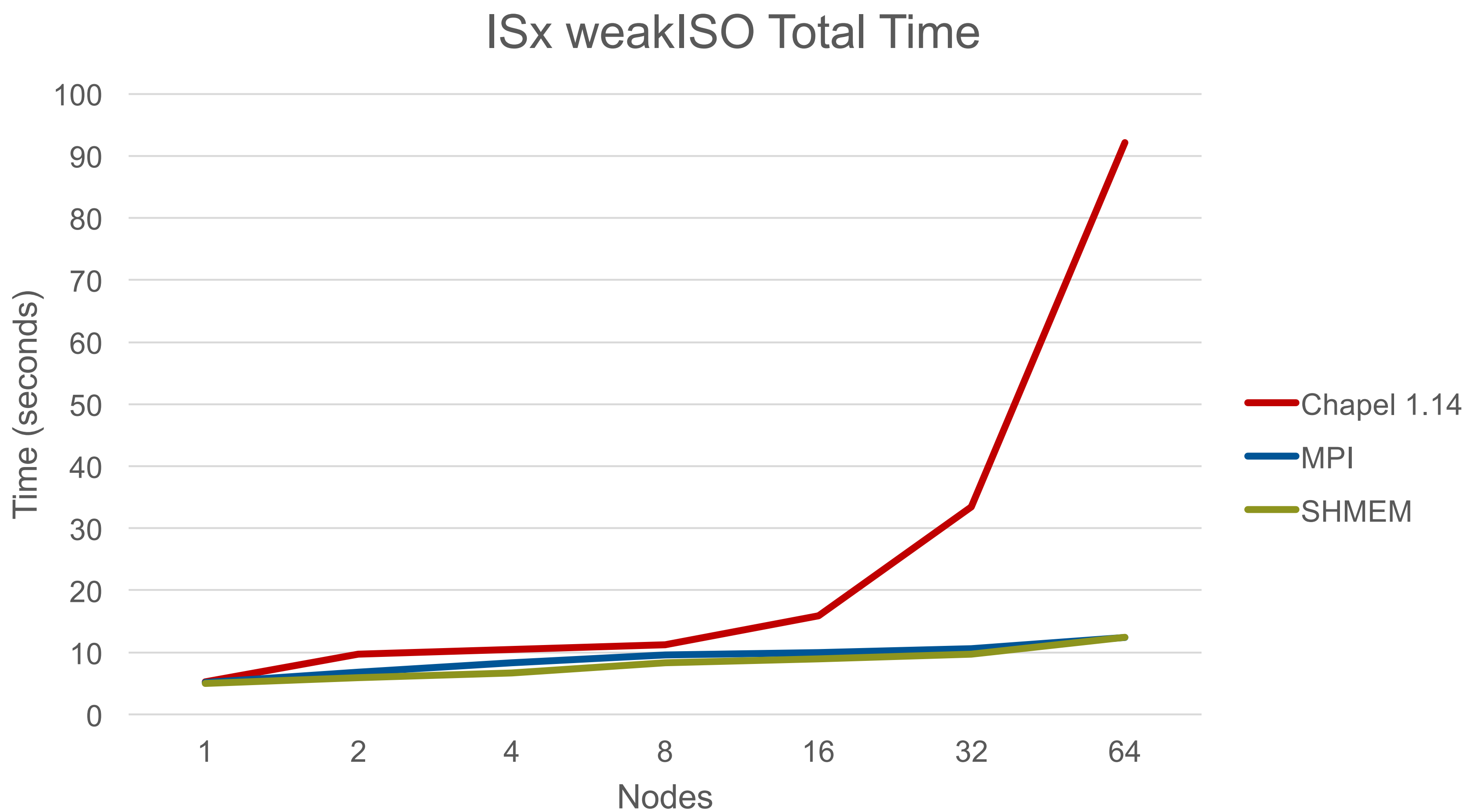
ISx: Performance Summary

- **Gathered on Cray XC with default problem size**
  - reference versions



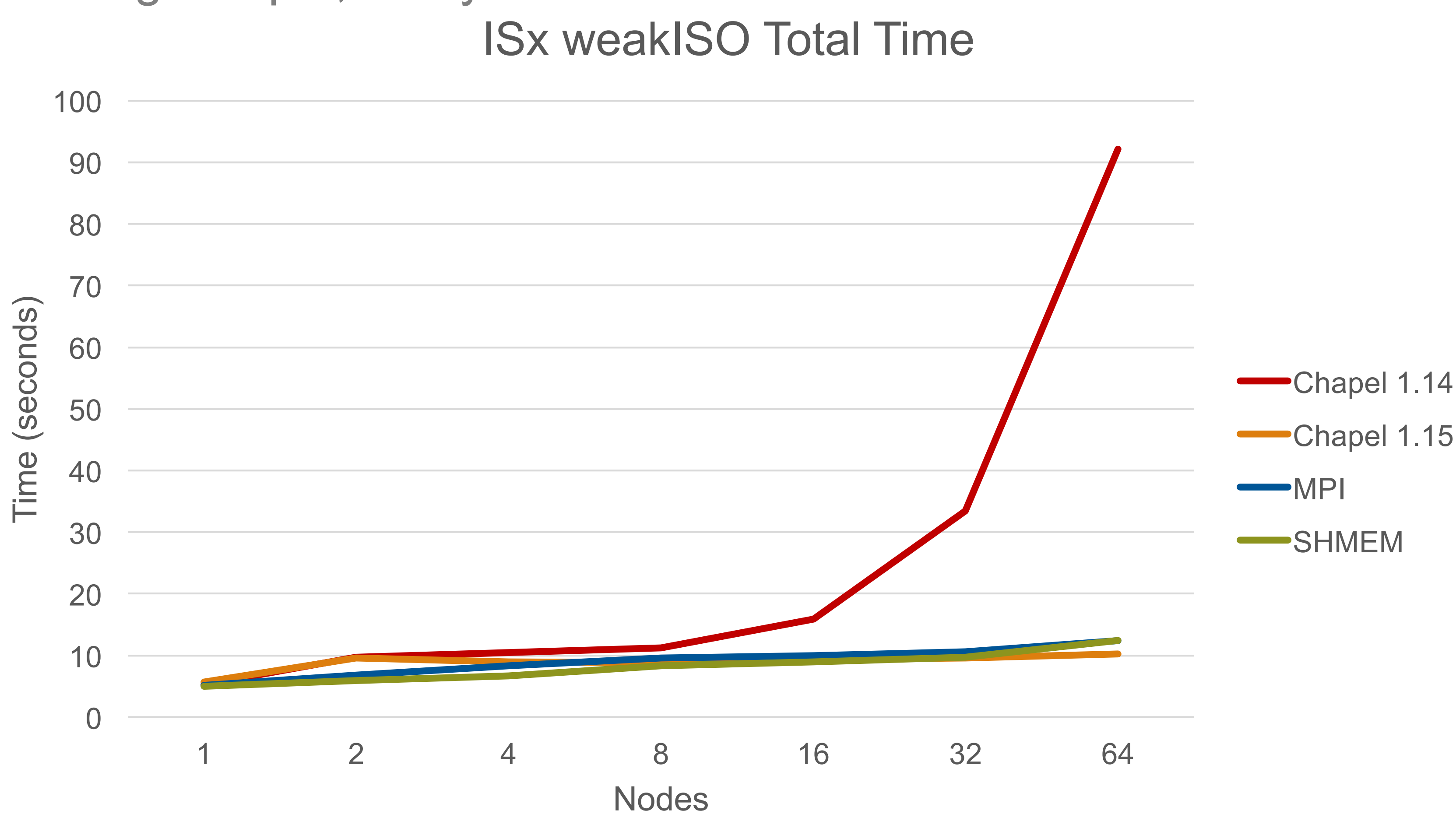
ISx: Performance Summary

- **Gathered on Cray XC with default problem size**
  - adding Chapel, six months ago:



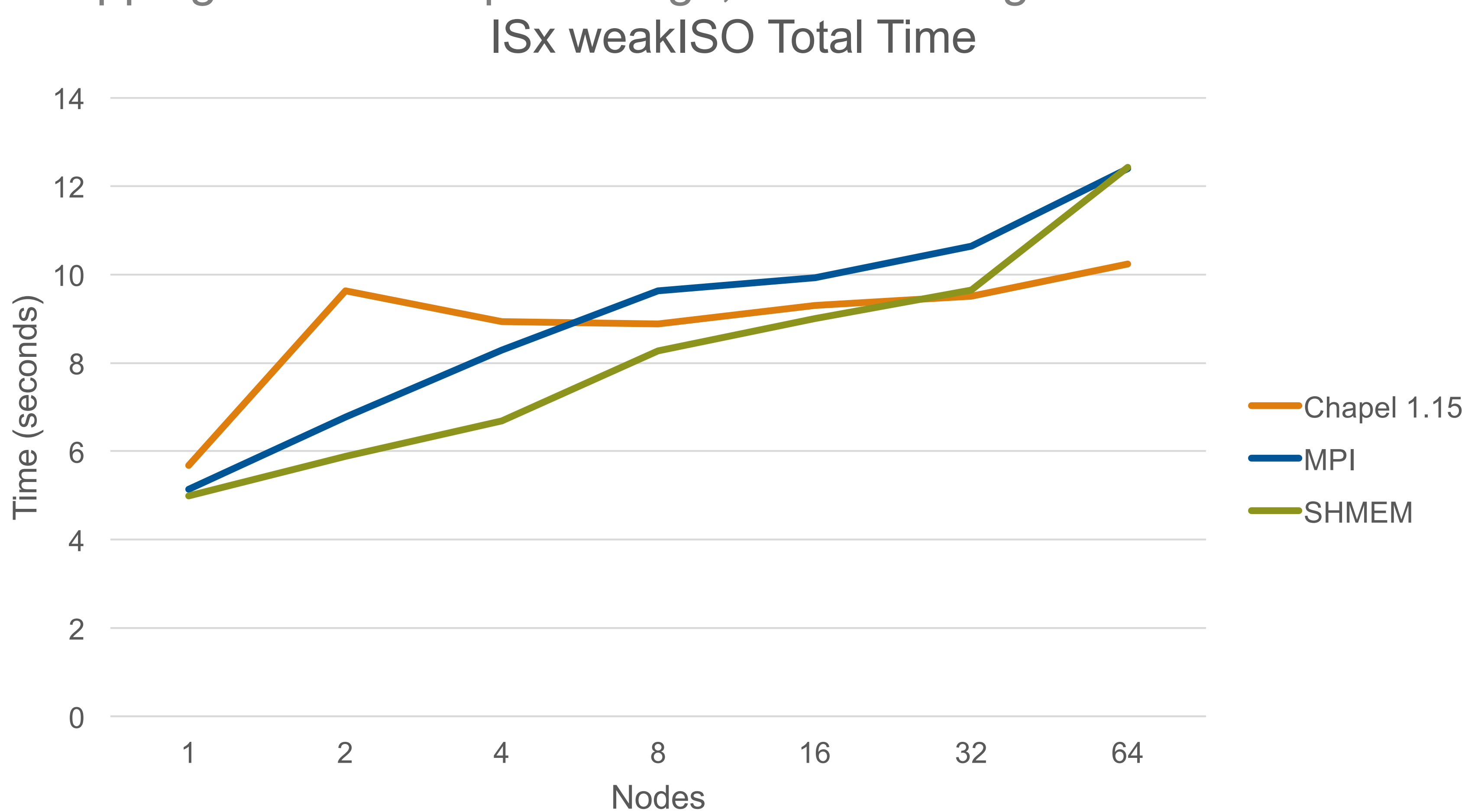
ISx: Performance Summary

- **Gathered on Cray XC with default problem size**
  - adding Chapel, today:



ISx: Performance Summary

- **Gathered on Cray XC with default problem size**
  - dropping the old Chapel timings, and zooming in:



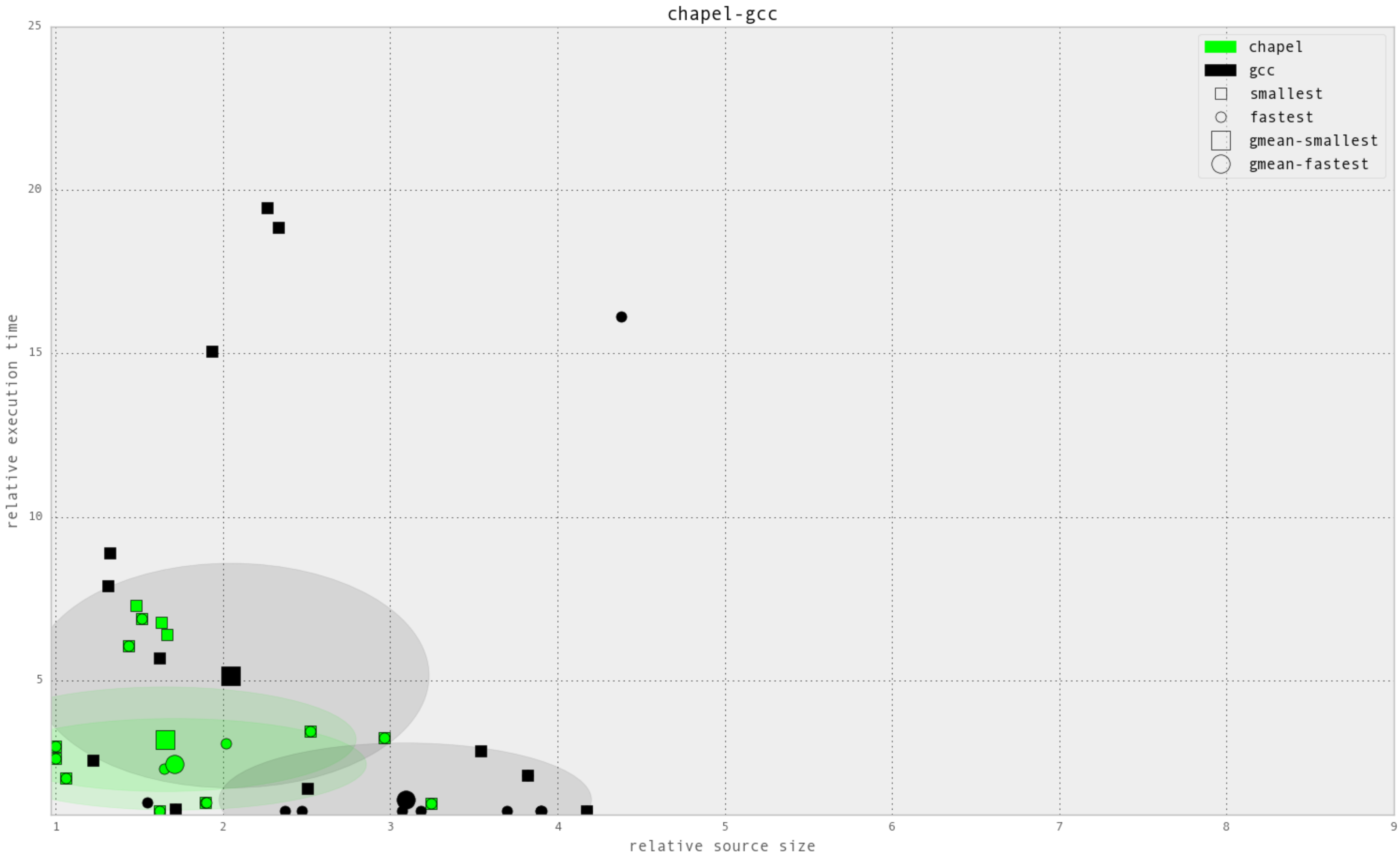
Overview Summary

- **Chapel has nice features for parallelism and locality**
- **Traditional reasons for not using Chapel are falling away**
  - performance specifically is becoming less of a concern with time
- **Aiming for a “version 2.0 release” over the near year or so**
  - intent: no further breaking changes after that point

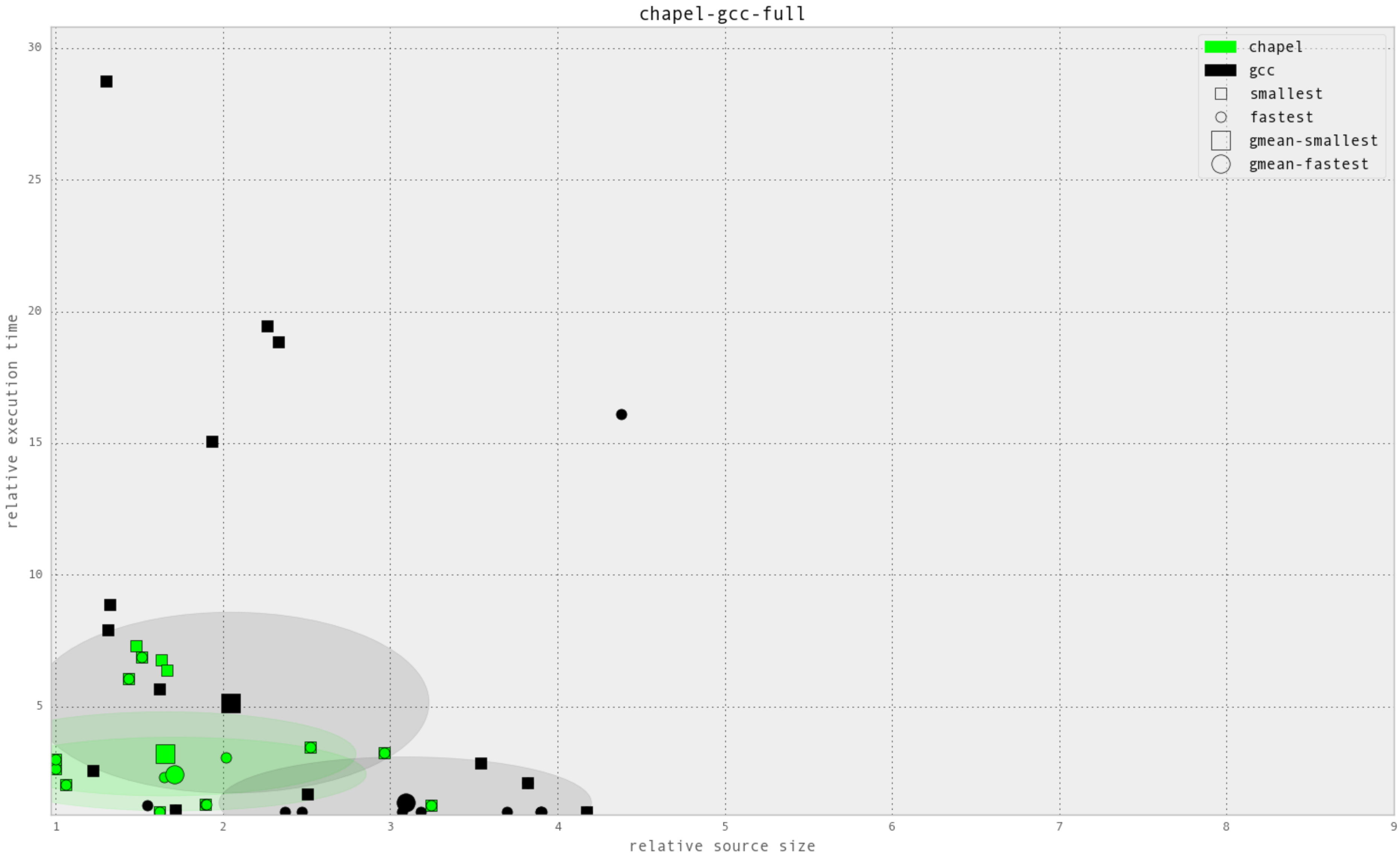
High-level Questions about Chapel?

Full-size CLBG Scatter Plots

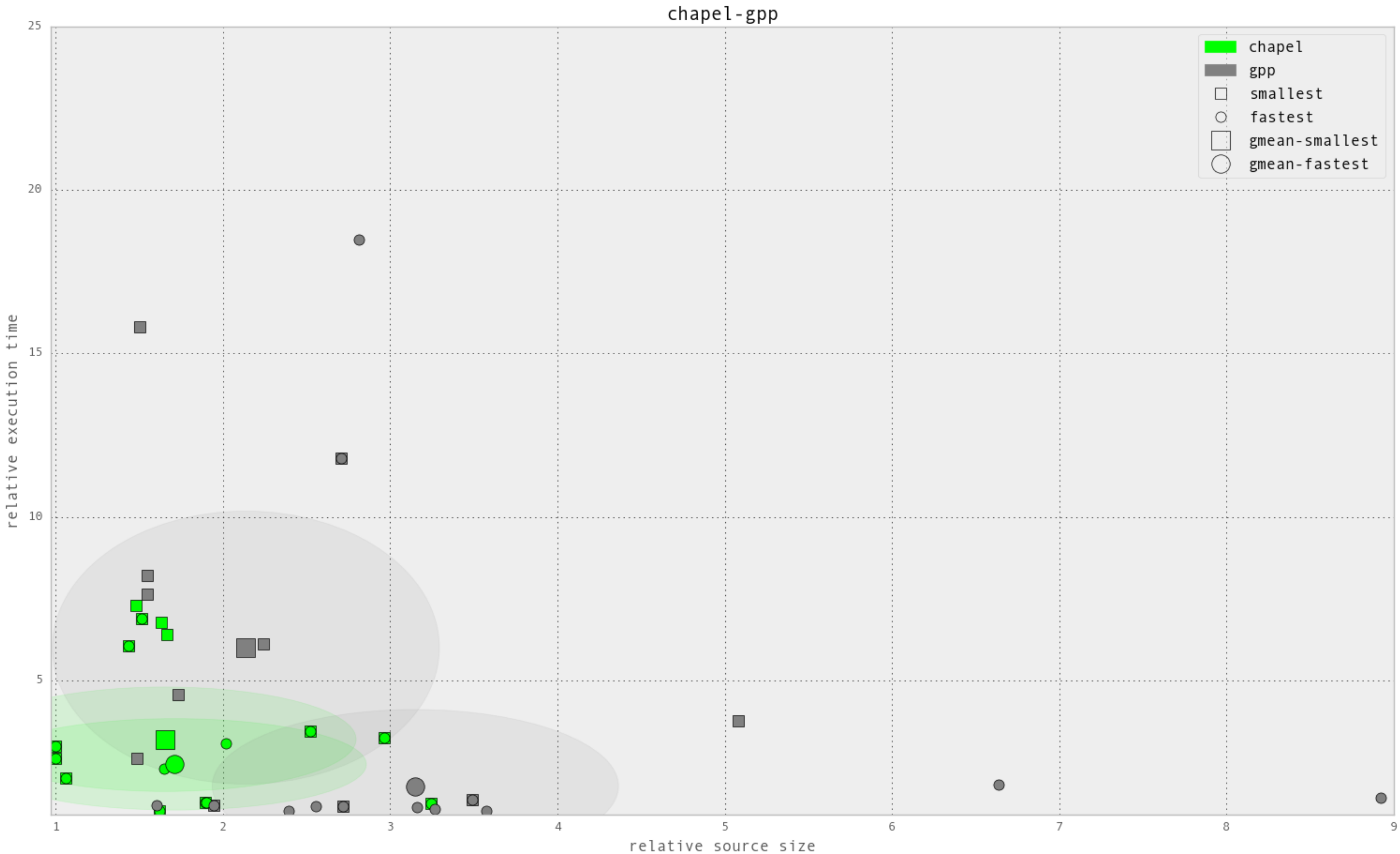
Chapel vs. C


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

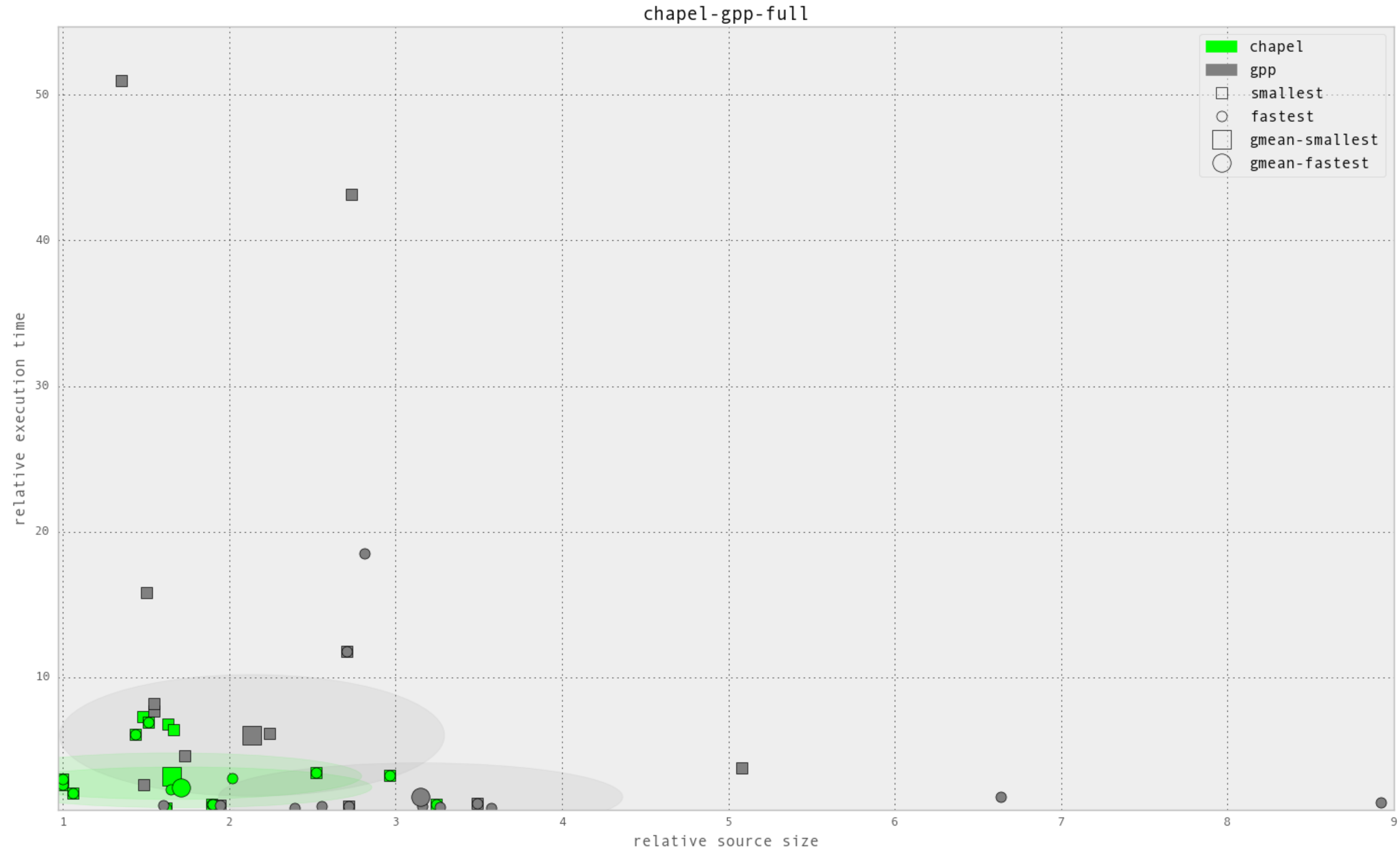
Chapel vs. C (zoomed out)



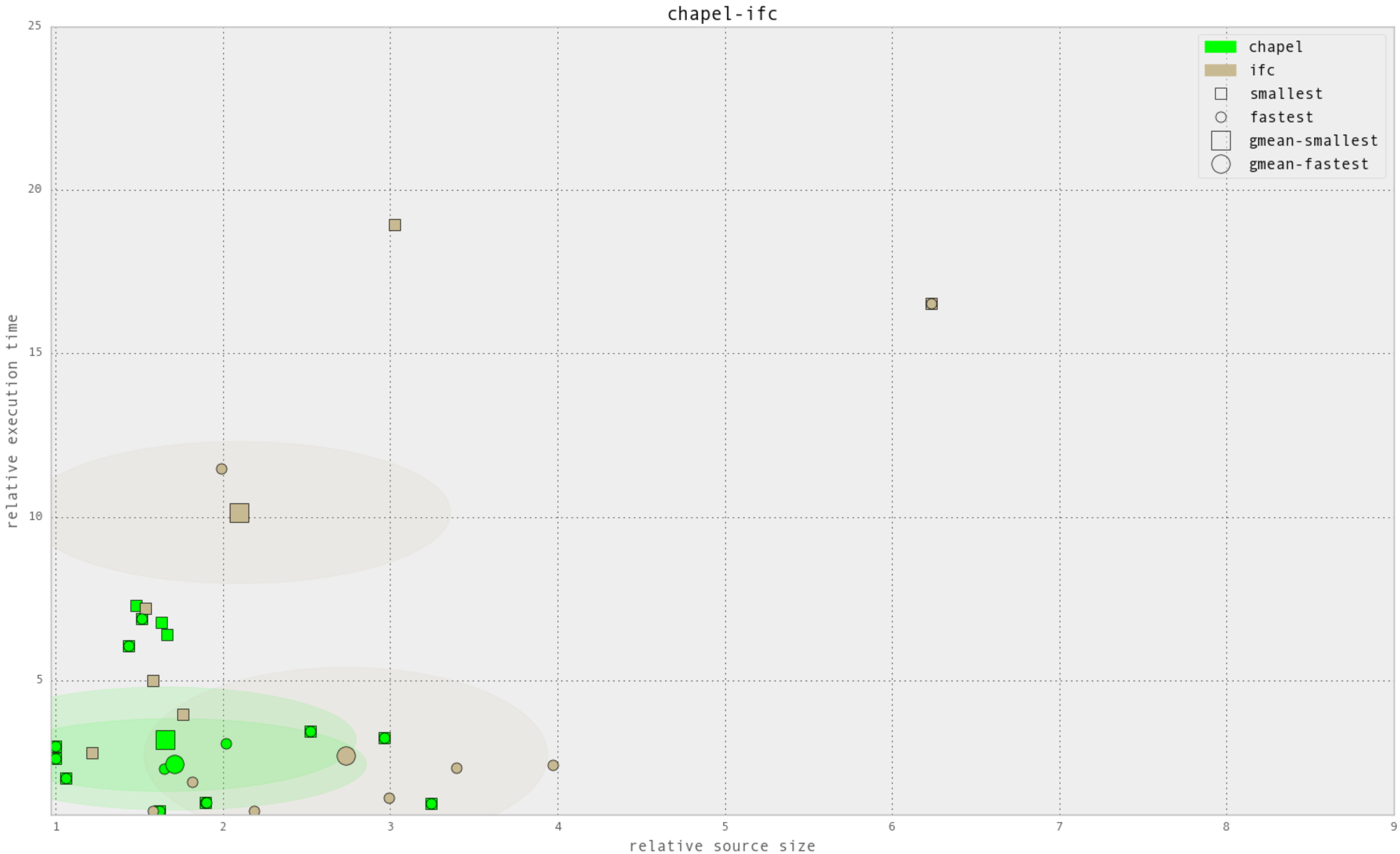
Chapel vs. C++


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

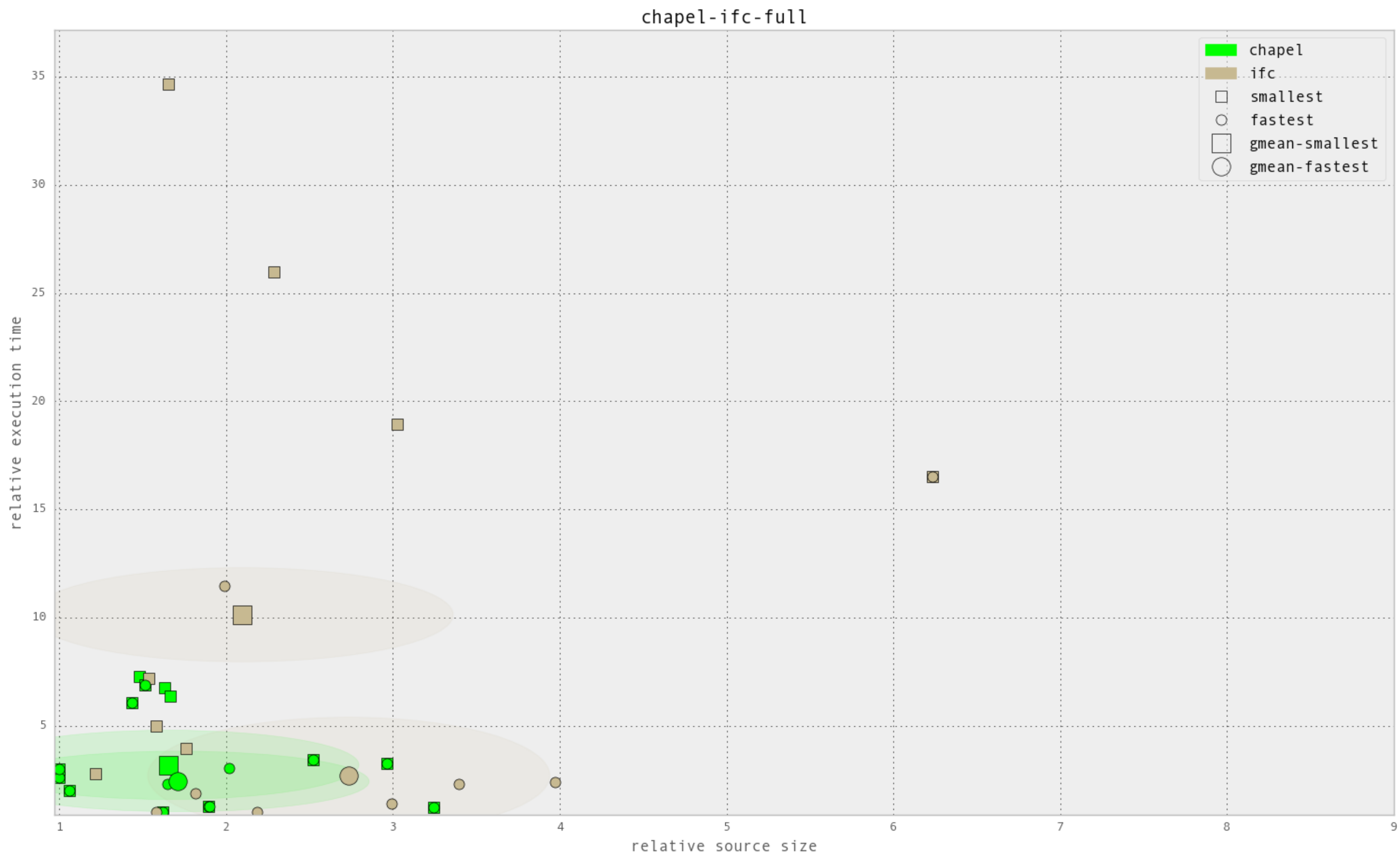
Chapel vs. C++ (zoomed out)



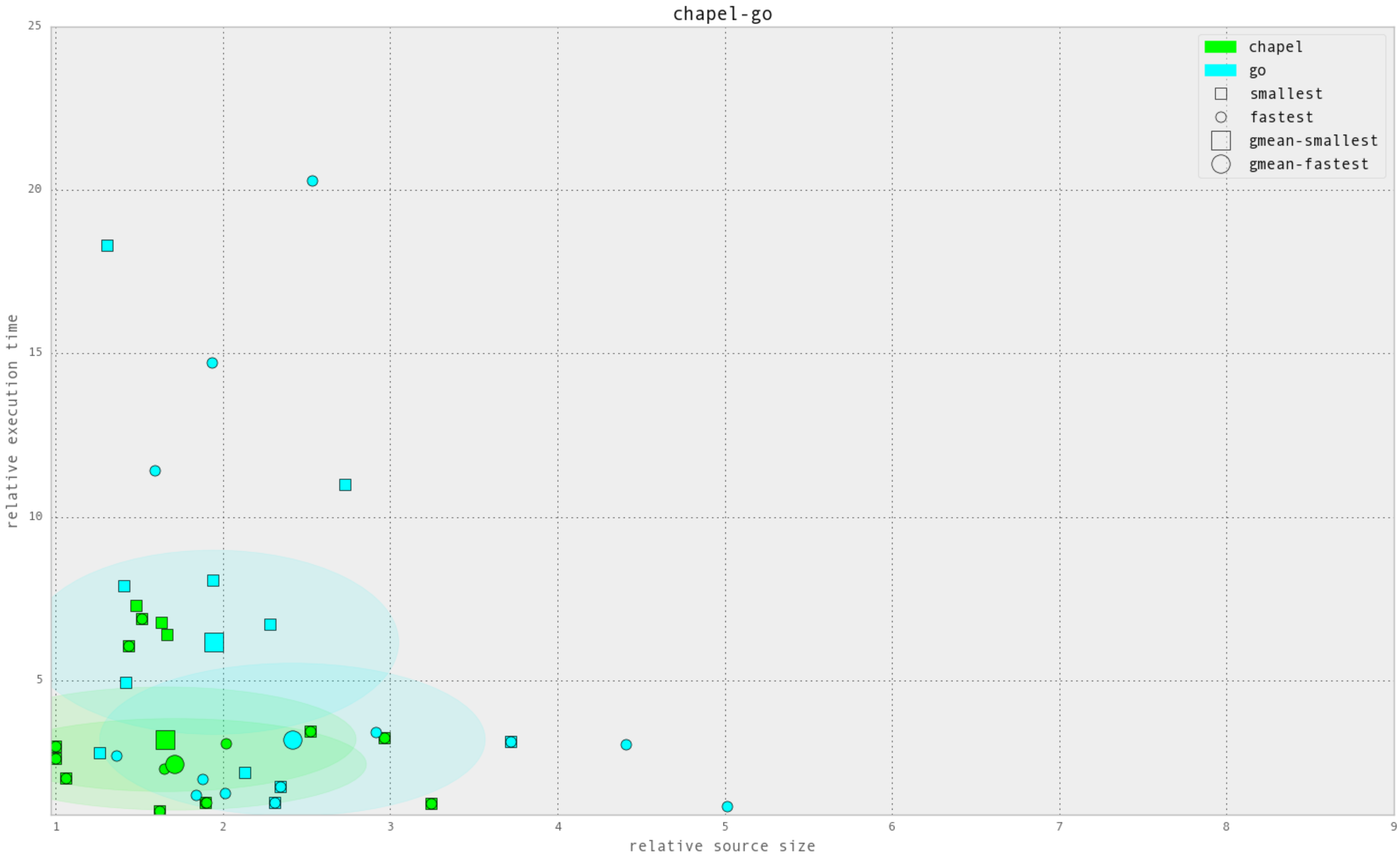
Chapel vs. Fortran



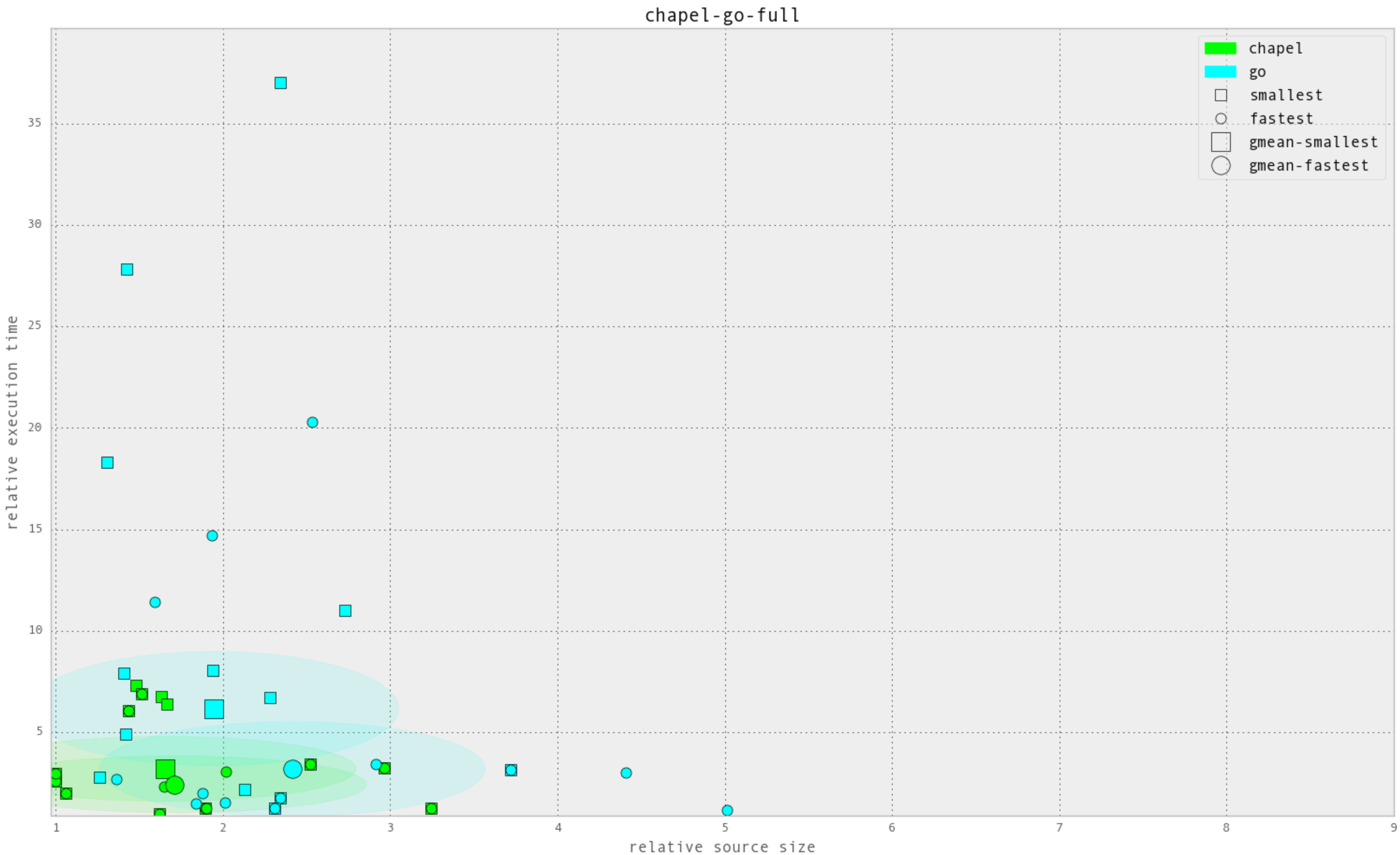
Chapel vs. Fortran (zoomed out)



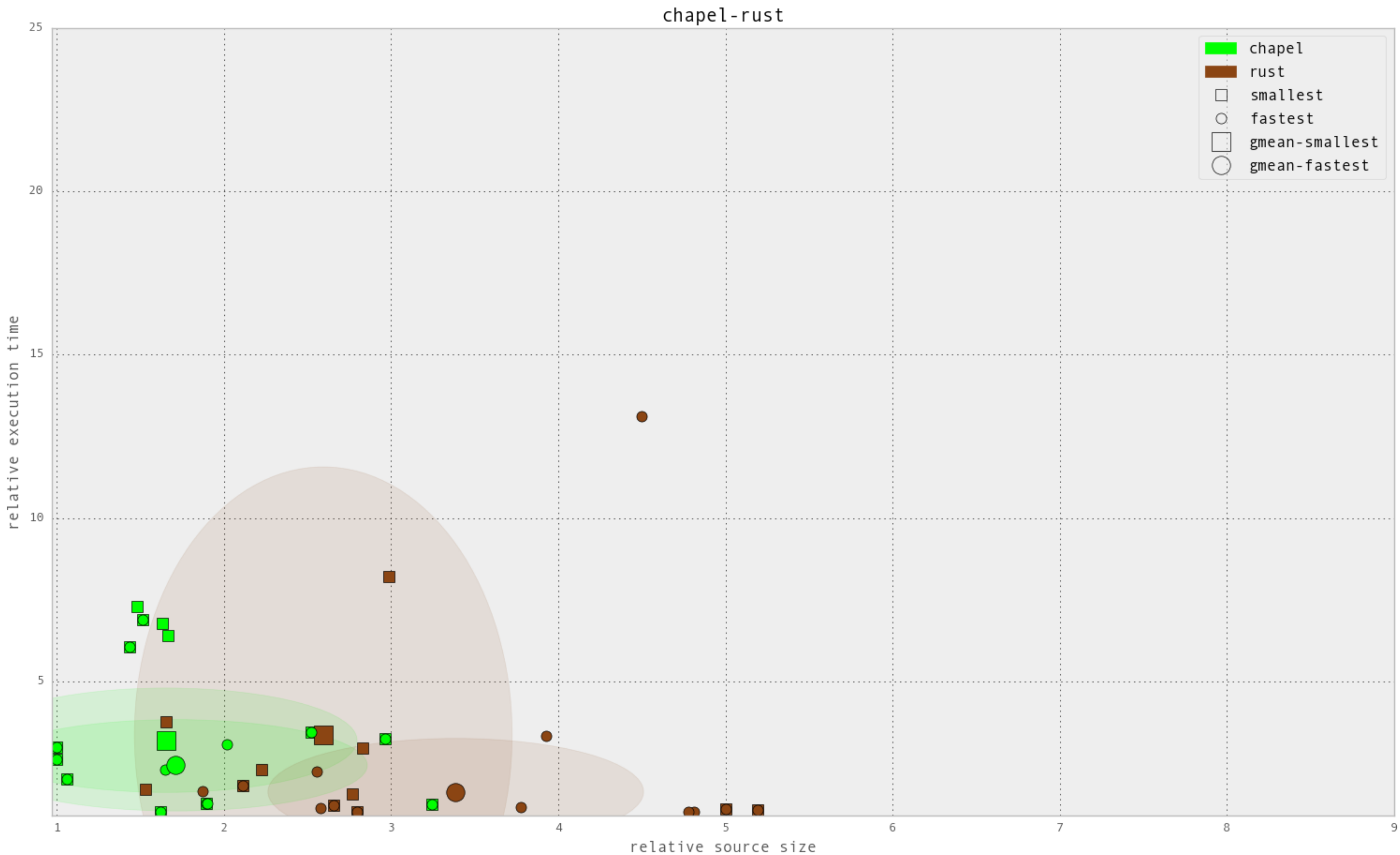
Chapel vs. Go


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

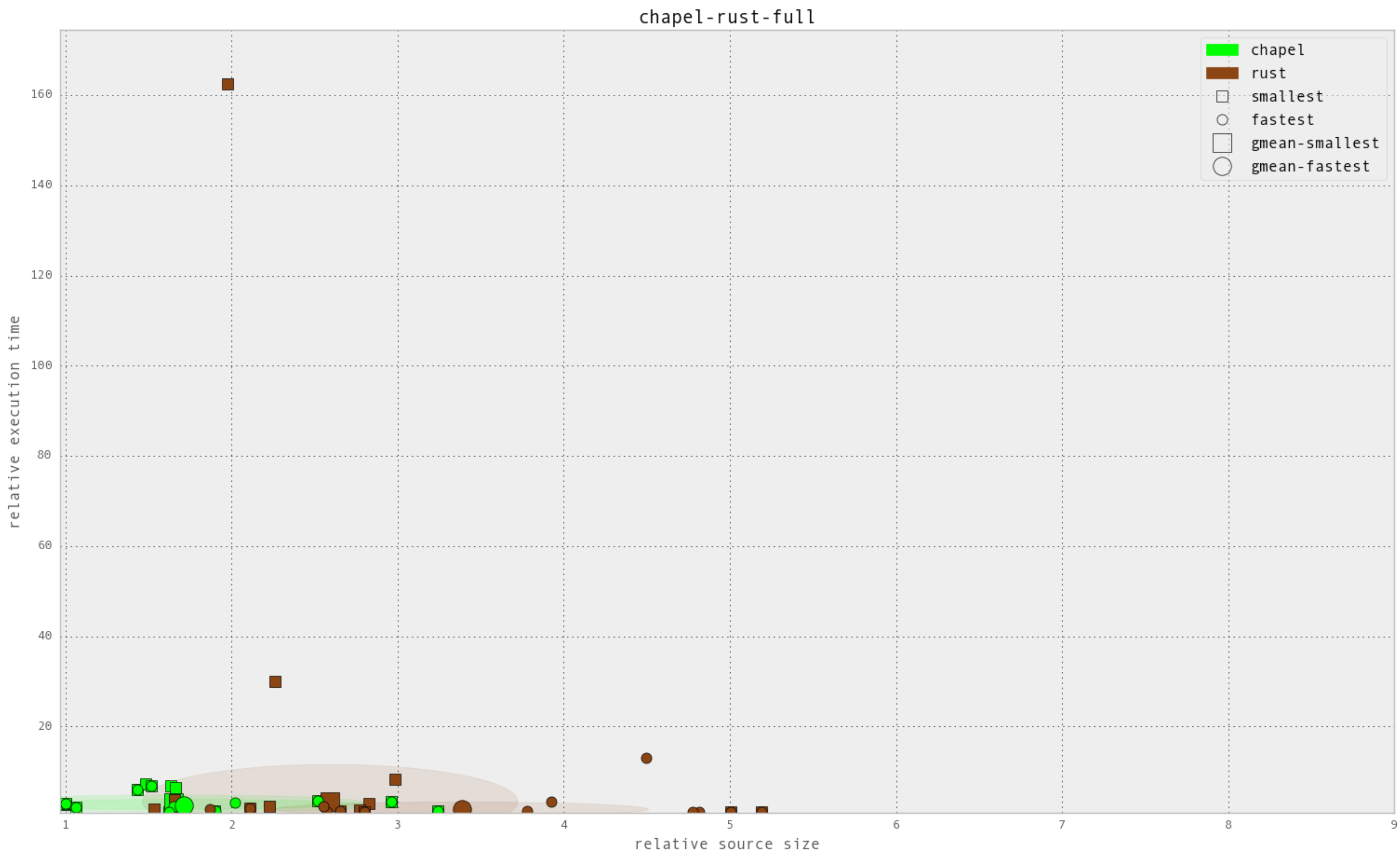
Chapel vs. Go (zoomed out)



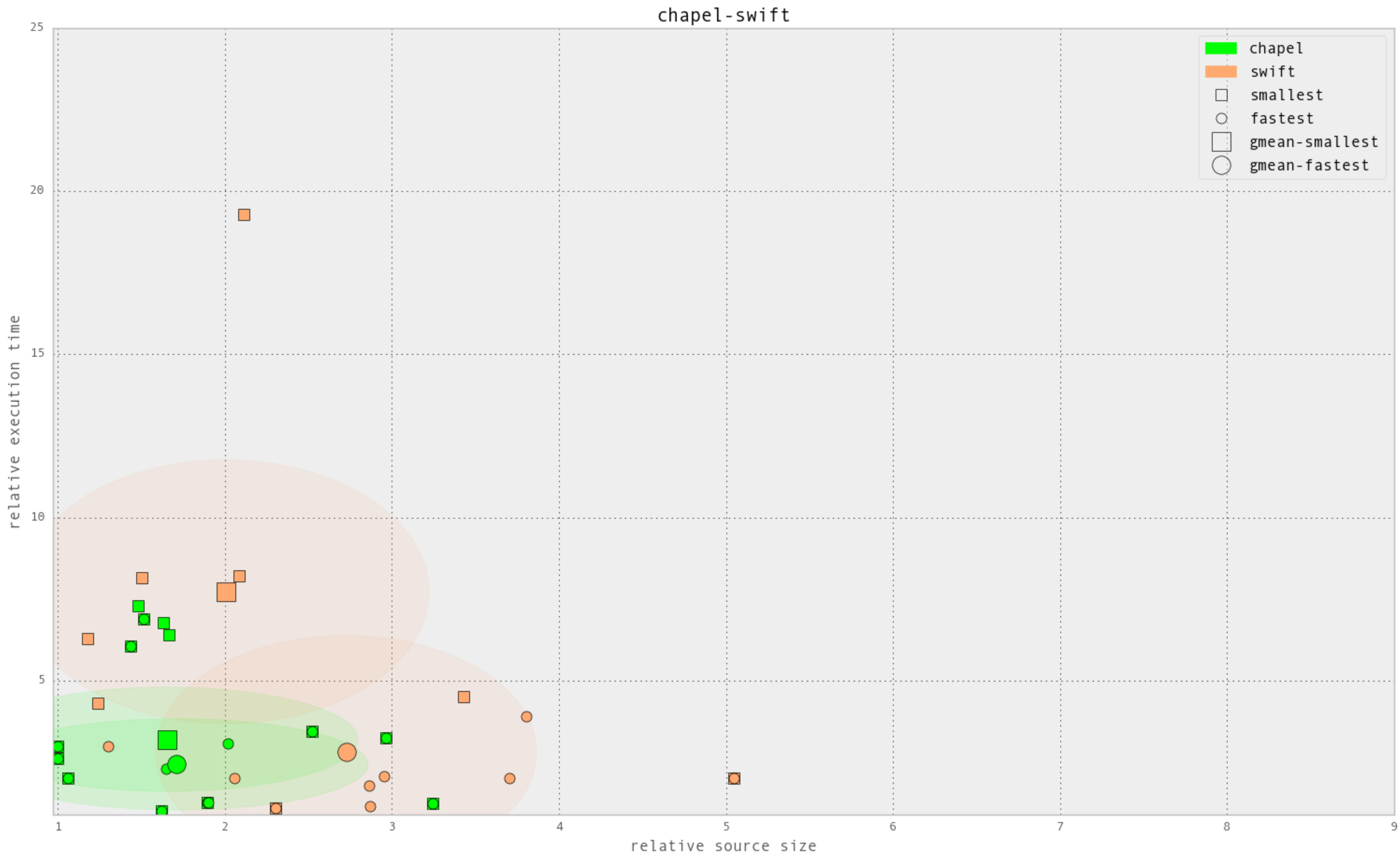
Chapel vs. Rust



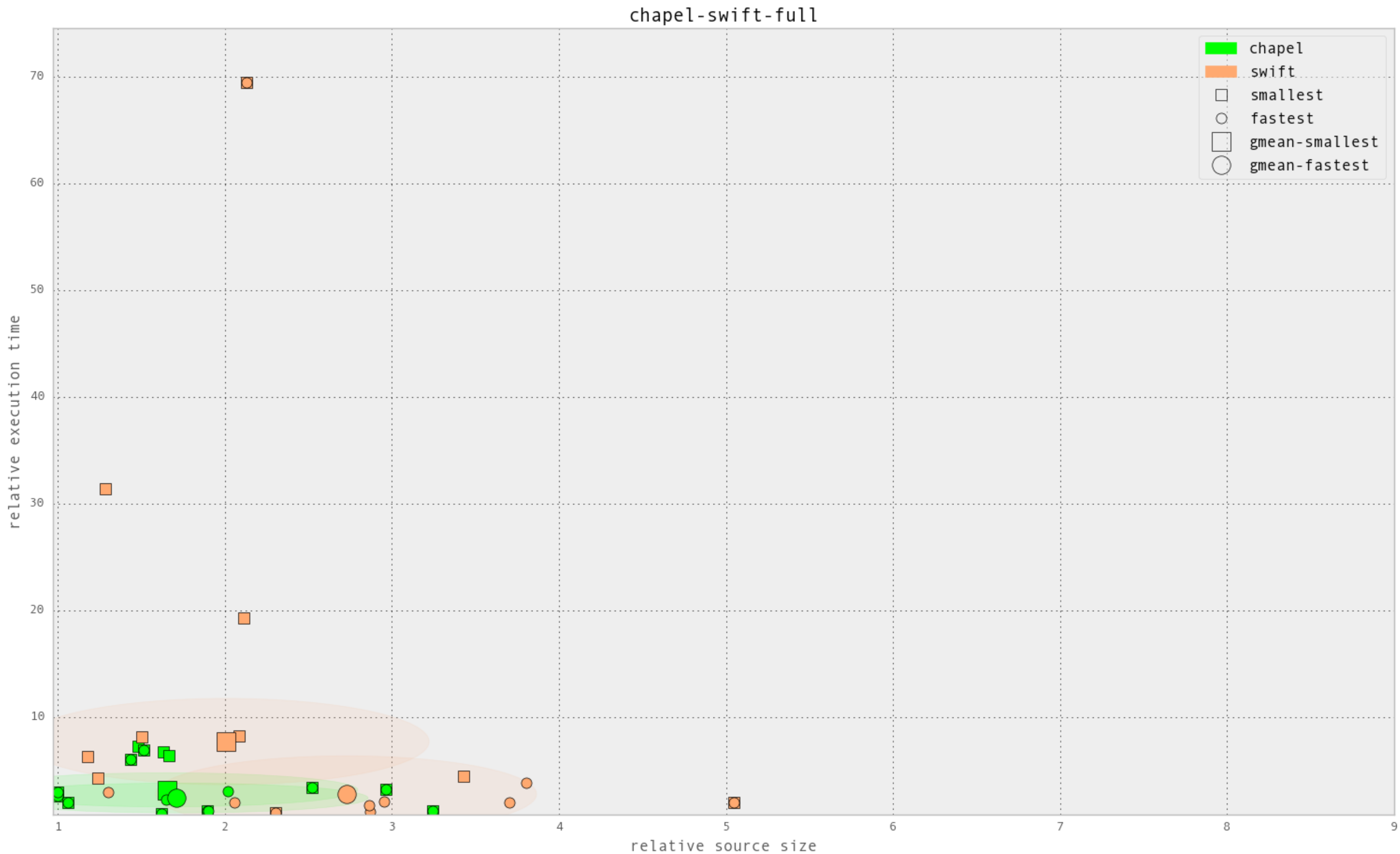
Chapel vs. Rust (zoomed out)



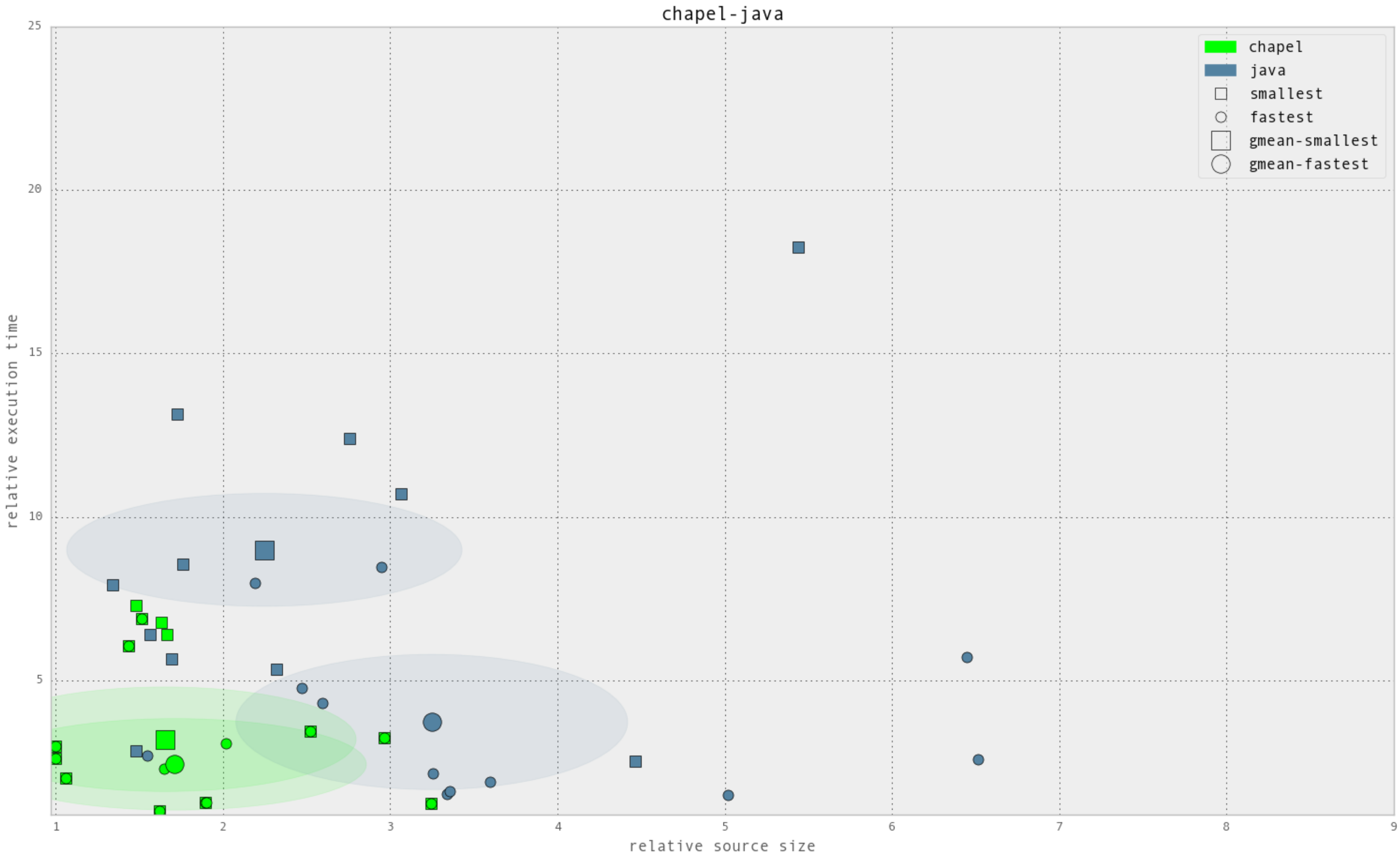
Chapel vs. Swift



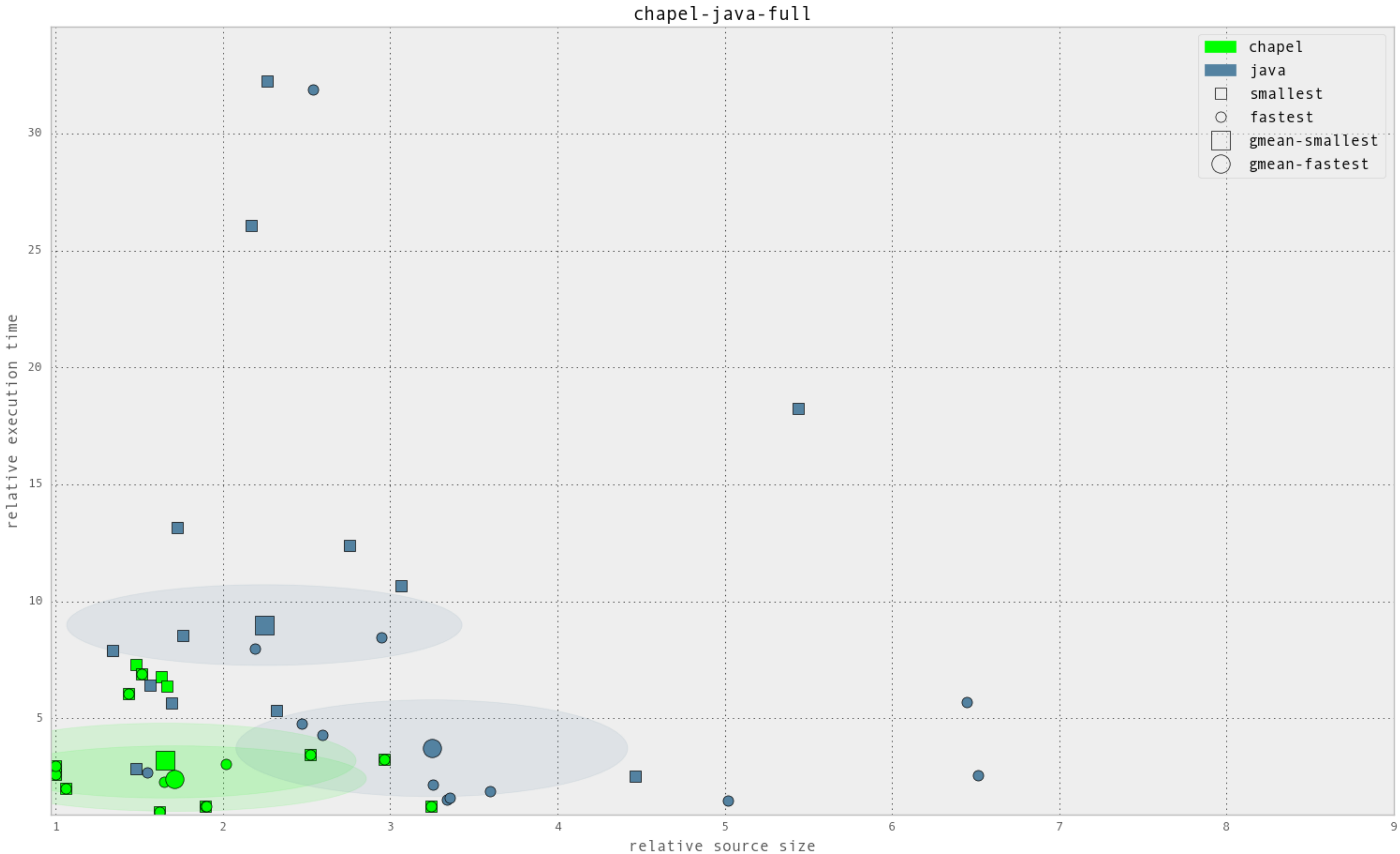
Chapel vs. Swift (zoomed out)



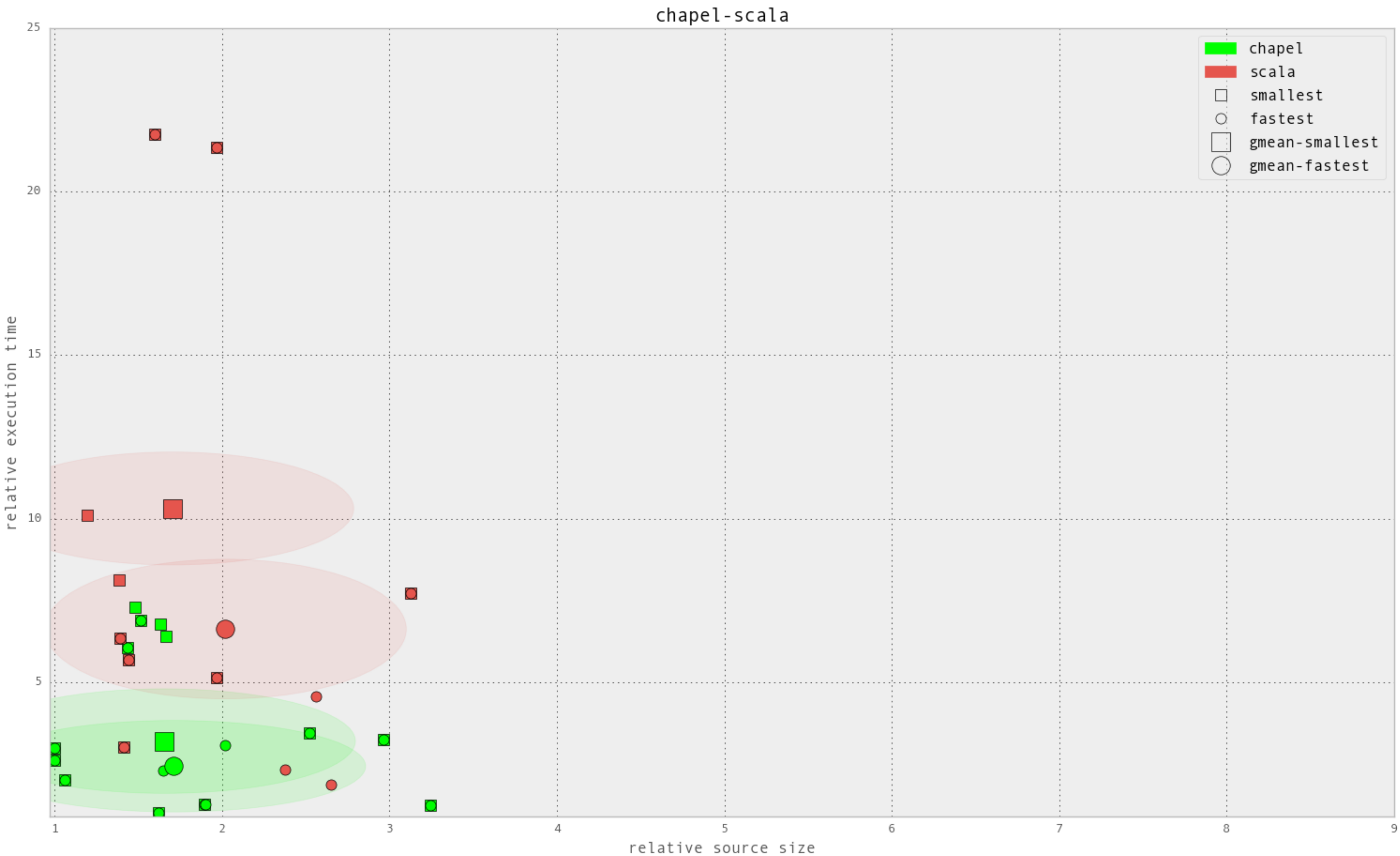
Chapel vs. Java



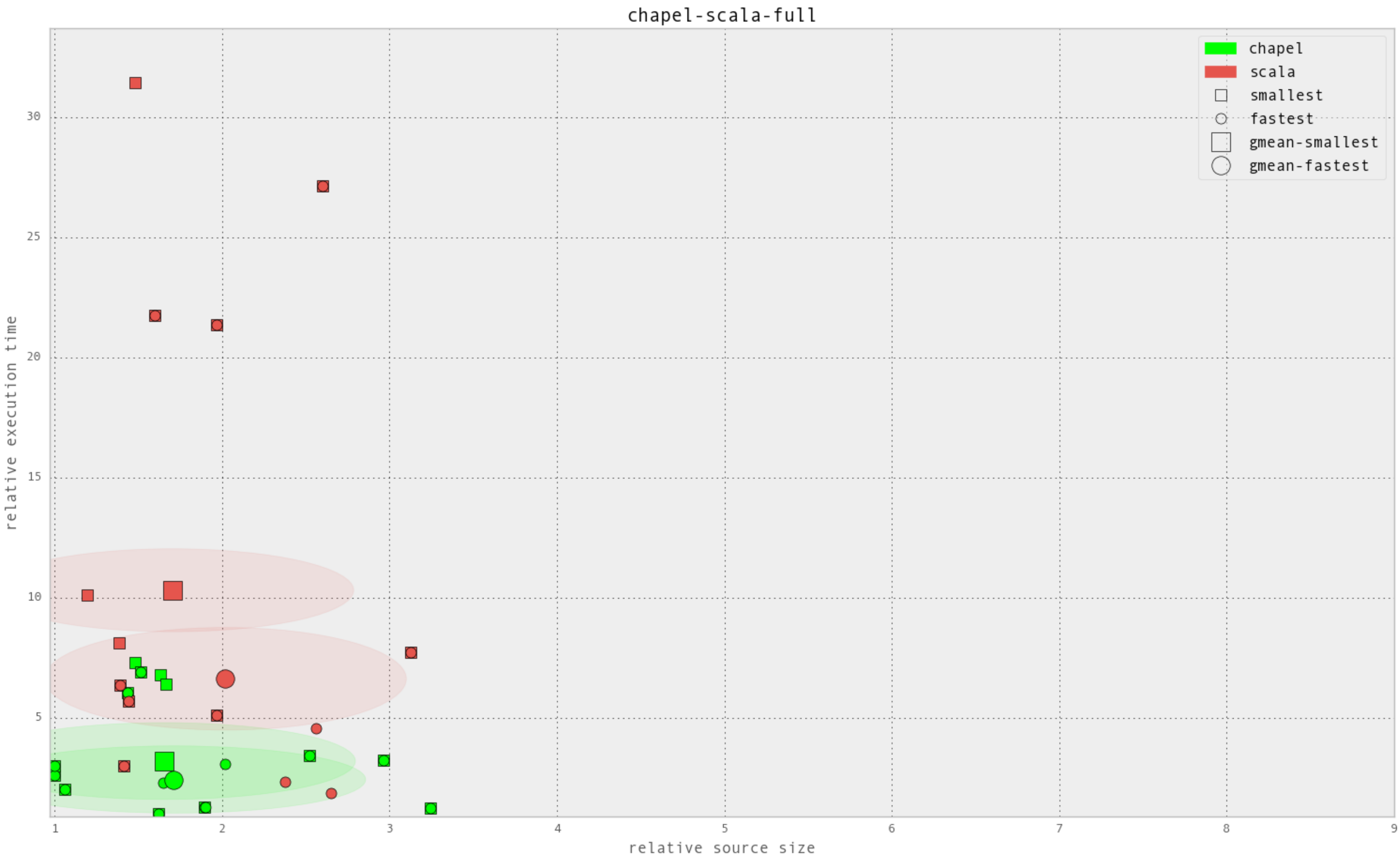
Chapel vs. Java (zoomed out)



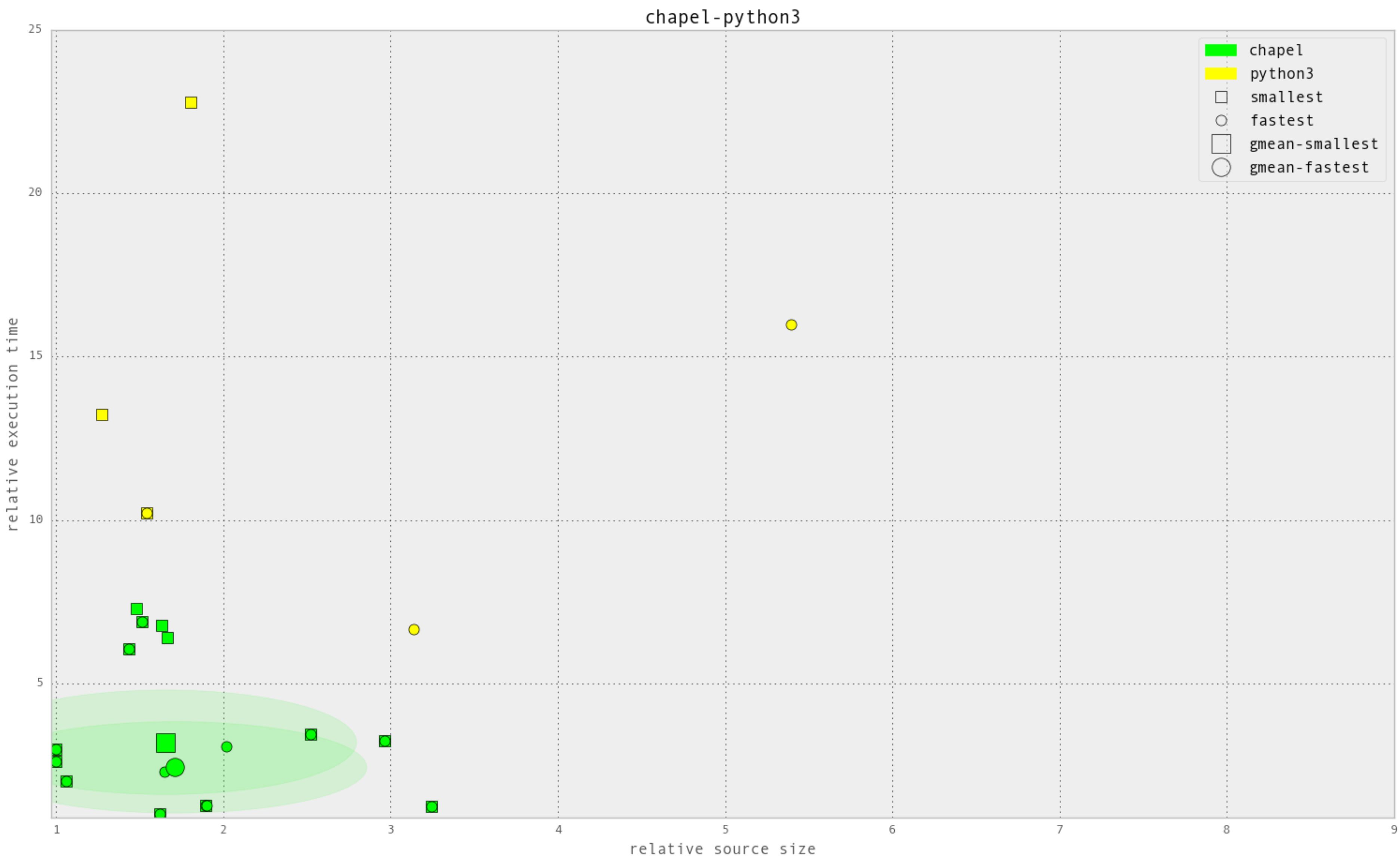
Chapel vs. Scala



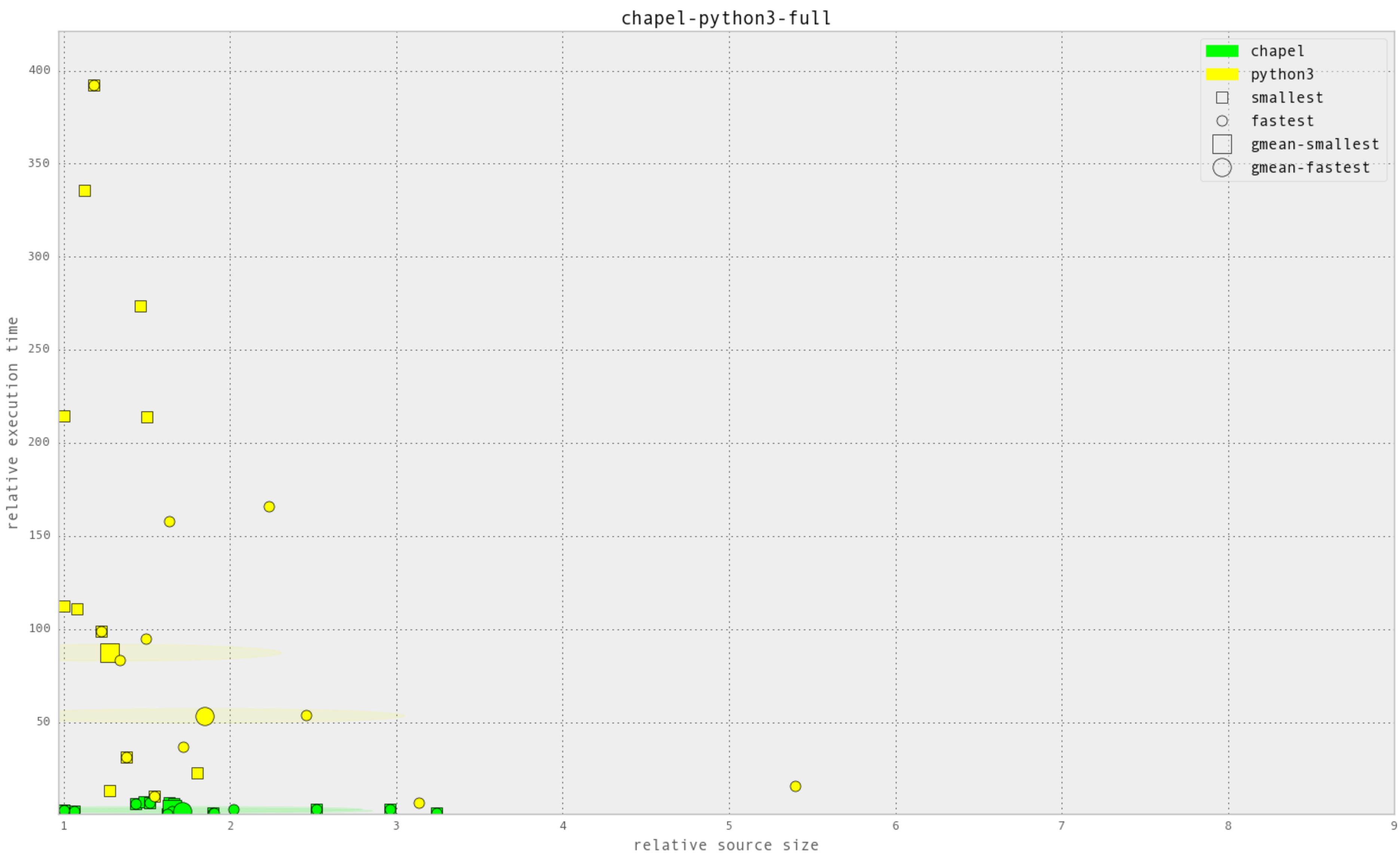
Chapel vs. Scala (zoomed out)



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.