Benchmarks and Performance Optimizations

Chapel Team, Cray Inc.

Chapel version 1.16

October 5, 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

- **PRK Case Study**
  - Bounded Coforall Optimization
  - StencilDist updateFluff() Optimization
  - Dynamic Registration Impact
  - Array Locality Optimization
- **ISx Benchmark Improvements**
  - Record Serialization
  - Task Counting Improvements
- **Computer Language Benchmarks Game Update**
- **Reductions in Memory Leaks**

PRK Case Study

PRK: Background

- **PRK: Parallel Research Kernels**
  - Compact set of parallel apps distilled from real benchmarks
  - 12 kernels, each testing a different parallel idiom
    - stencil, particle-in-cell, matrix transpose, sparse matrices, and more
  - Developed by Intel

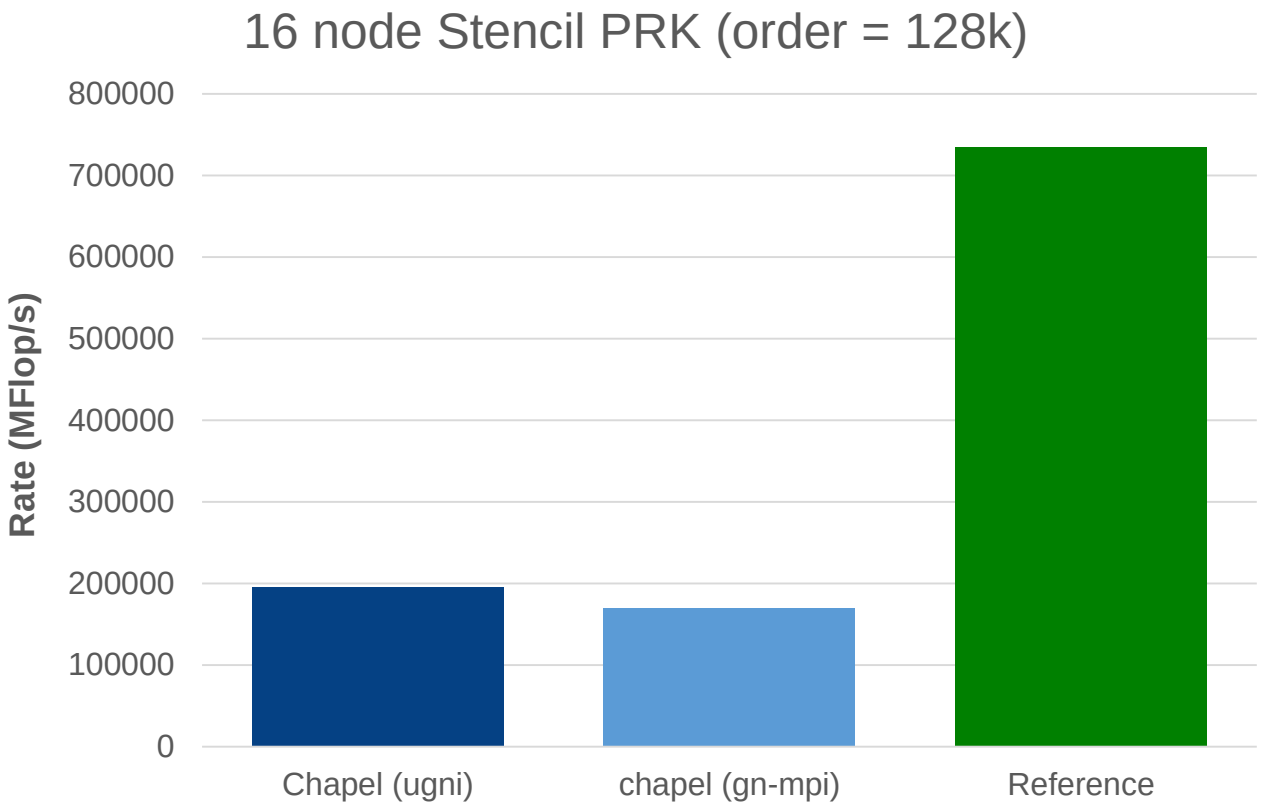
- **Focused on Stencil PRK for this release**
  - Added an “optimized” variant that sacrifices some elegance
    - used array localAccess instead of direct indexing

      `input.localAccess[i, j]` *// instead of input[i,j]*
    - used a local block to squash communication

      `forall` `(i,j)` **`in`** `innerDom` **`do local`** *// instead of **forall** (i, j) **in** innerDom*

PRK: Background

- **Optimized stencil still lagged behind reference in 1.15**
  - Chapel was 3-4x slower than reference OpenMP+MPI version



Bounded Coforall Optimization

Bounded Remote Coforall: Background

- **Remote coforalls were transformed by the compiler, from:**

```
coforall loc in Locales do on loc { body(); }
```

roughly into:

```
var endCount: atomic int;

for loc in Locales {
  endCount.add(1);                          // note: incrementing counter once per task
  executeOnNB(bodyWrapper, endCount);
}
endCount.waitFor(0);

proc bodyWrapper(endCount) {
  body();
  endCount.sub(1);
}
```

Bounded Remote Coforall: This Effort

- **Minimize end-count manipulation for "bounded" coforalls**
  - "bounded" coforalls have a known trip-count (range/domain/array)

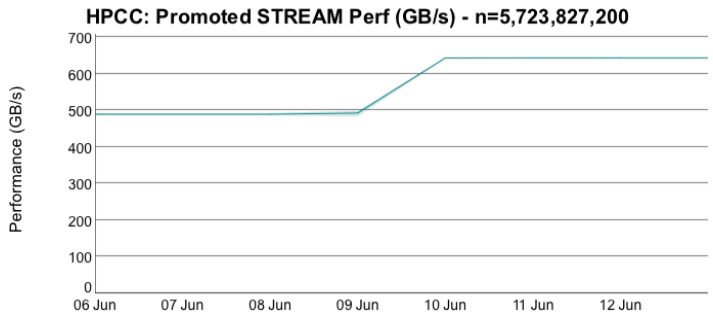
```
coforall loc in Locales do on loc { body(); }
```

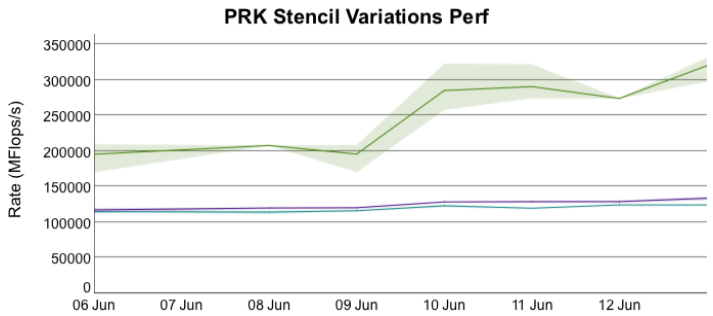
now roughly converted to:

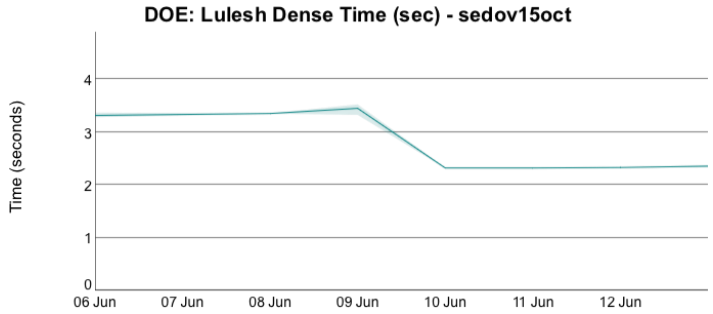
```
var tmpIter = Locales;
var numTasks = tmpIter.size
var endCount: atomic int;

endCount.add(numTasks);                    // single atomic op vs. op per task
for loc in tmpIter {
  executeOnNB(bodyWrapper, endCount);
}
endCount.waitFor(0);

proc bodyWrapper(endCount) { /* same as before */ }
```

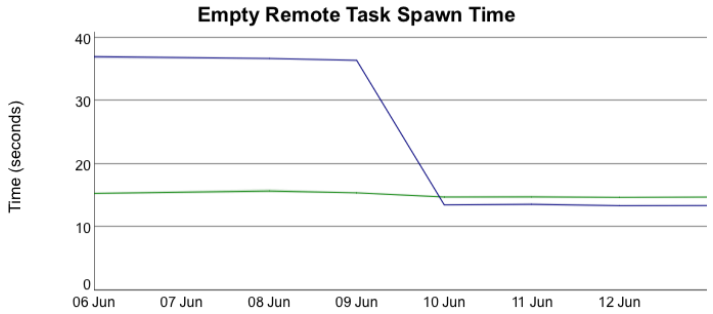
Bounded Remote Coforall: Impact

- Improved performance for several multi-locale benchmarks



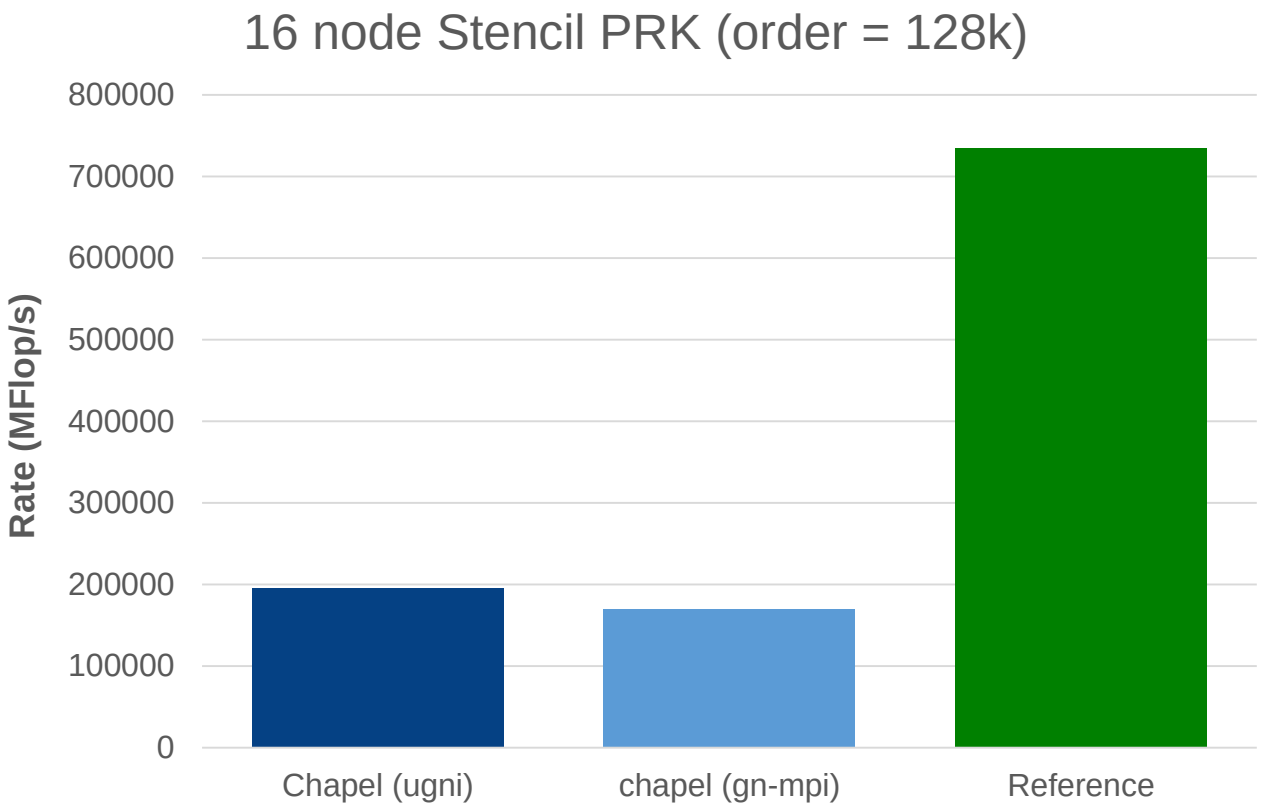






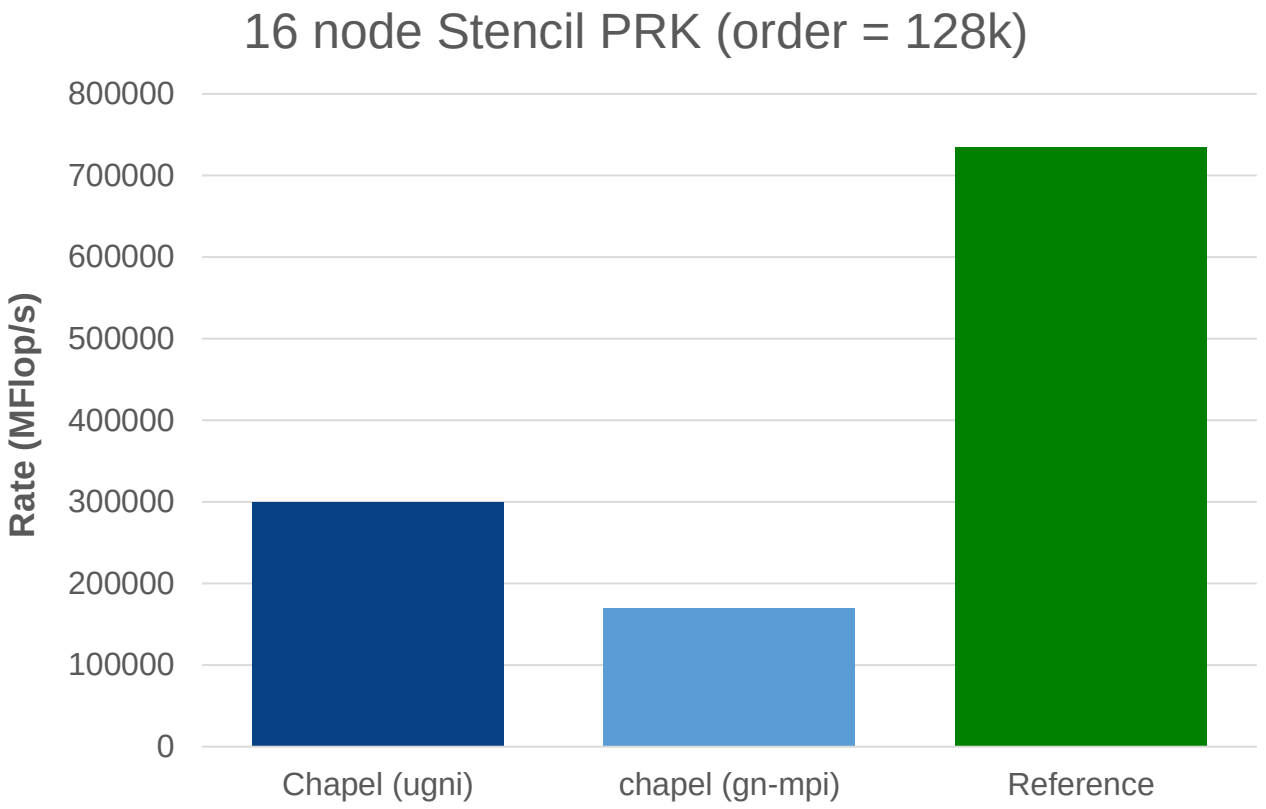
Bounded Remote Coforall: Stencil Impact

- **Previously Chapel was 3-4x slower than reference**



Bounded Remote Coforall: Stencil Impact

- **Previously Chapel was 3-4x slower than reference**
  - ugni version now ~1.5x faster than before

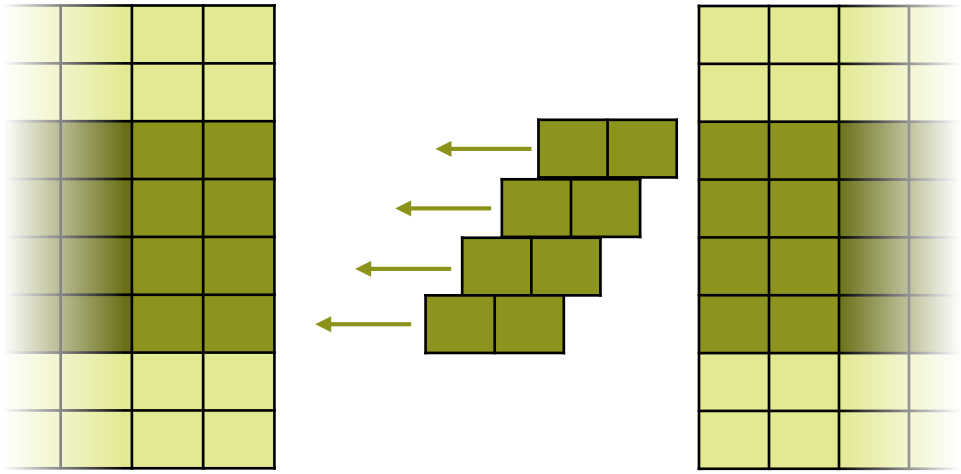


StencilDist updateFluff() Optimization

updateFluff: Background

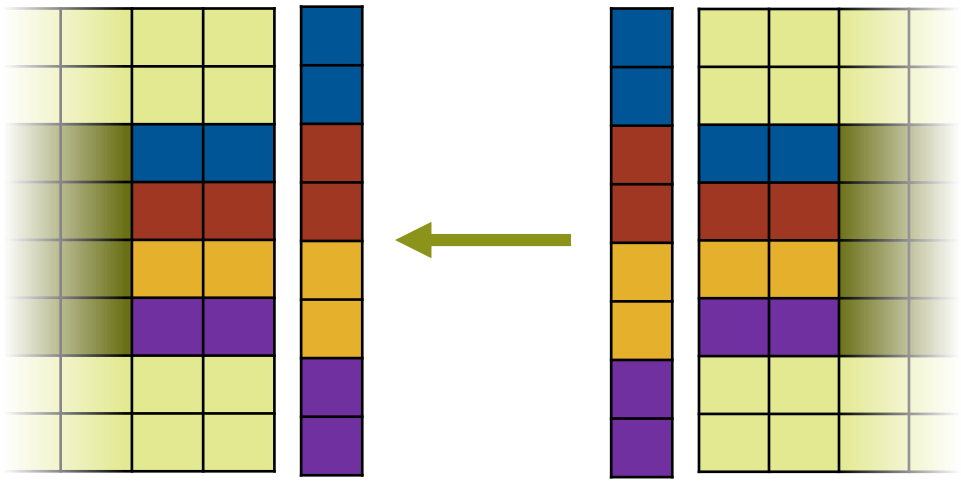
- **StencilDist: Block-like dist. for stencil computations**
  - Uses local cache of elements when read
  - 'updateFluff' method exchanges data with neighbors
  - Initially written for miniMD, now used by Stencil PRK and NPB MG

- **updateFluff naively exchanged elements**
  - Only bulk-transferred contiguous chunks at a time



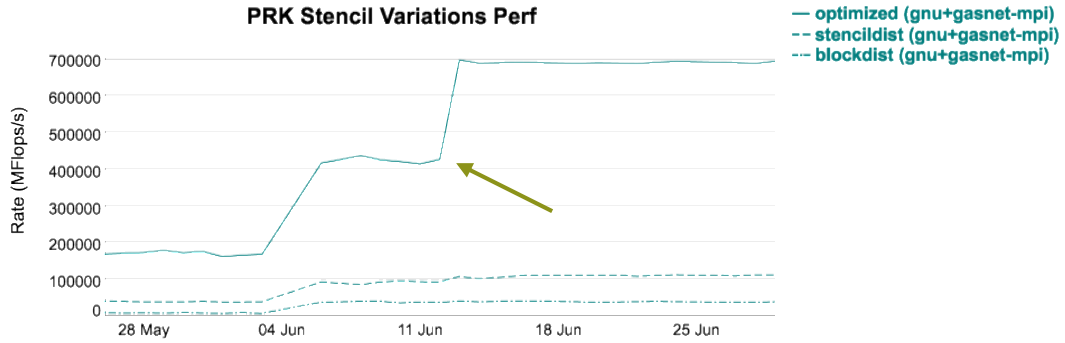
updateFluff: This Effort

- **Pack regions and perform one transfer**
  - Requires additional memory for buffers
  - Unpack on destination locale
  - Controlled by *'stencilDistAllowPackedUpdateFluff'* config param

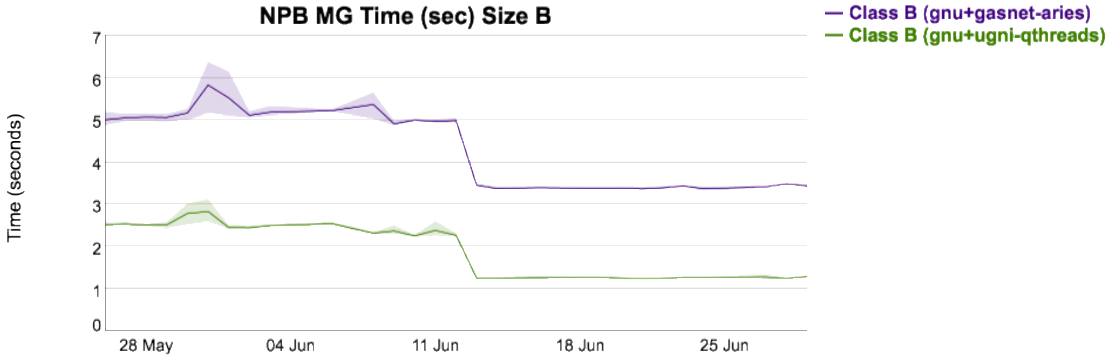


updateFluff: Impact

- Improved Stencil PRK under gasnet-mpi



- Also improved NPB MG



updateFluff: Status and Next Steps

Status:

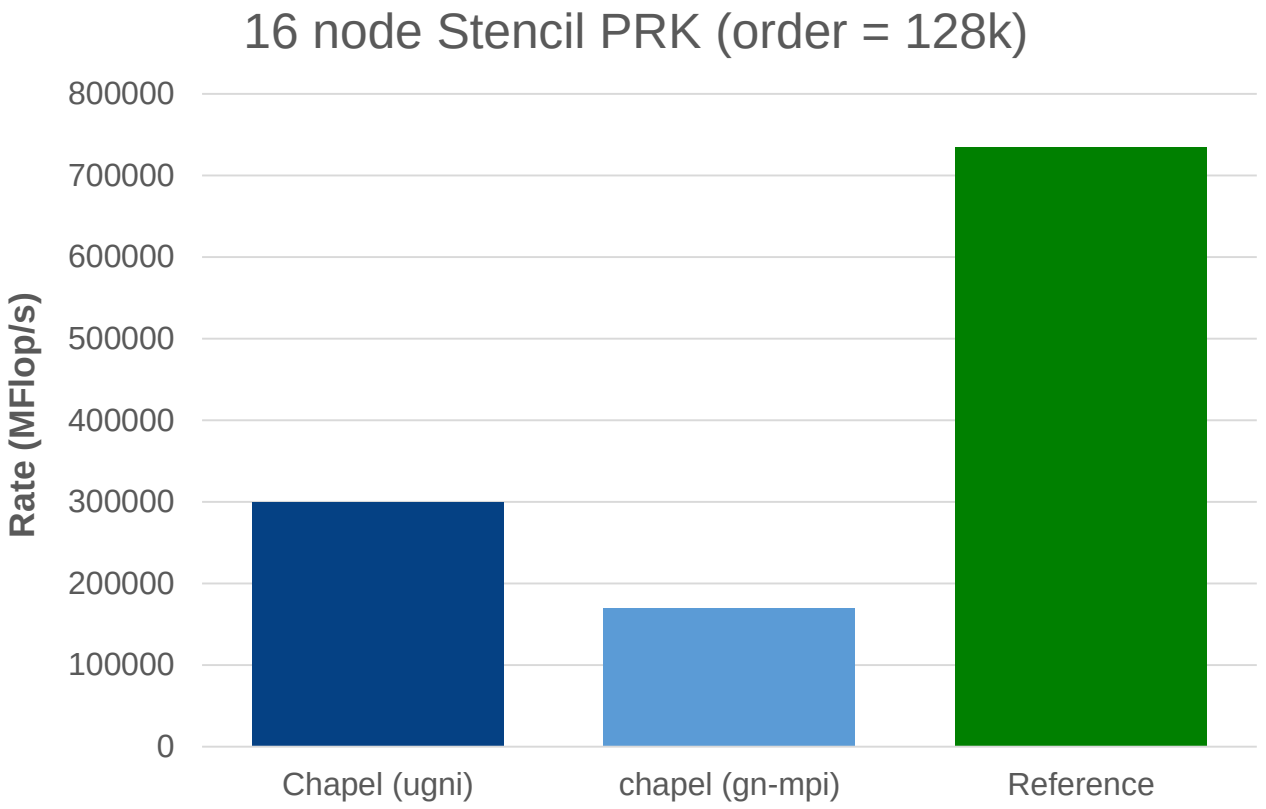
- Improved Stencil PRK and NPB MG performance
- Optimization enabled by default

Next Steps:

- Performance tuning
  - Not worth overhead for small numbers of elements
  - Find threshold that triggers the optimization

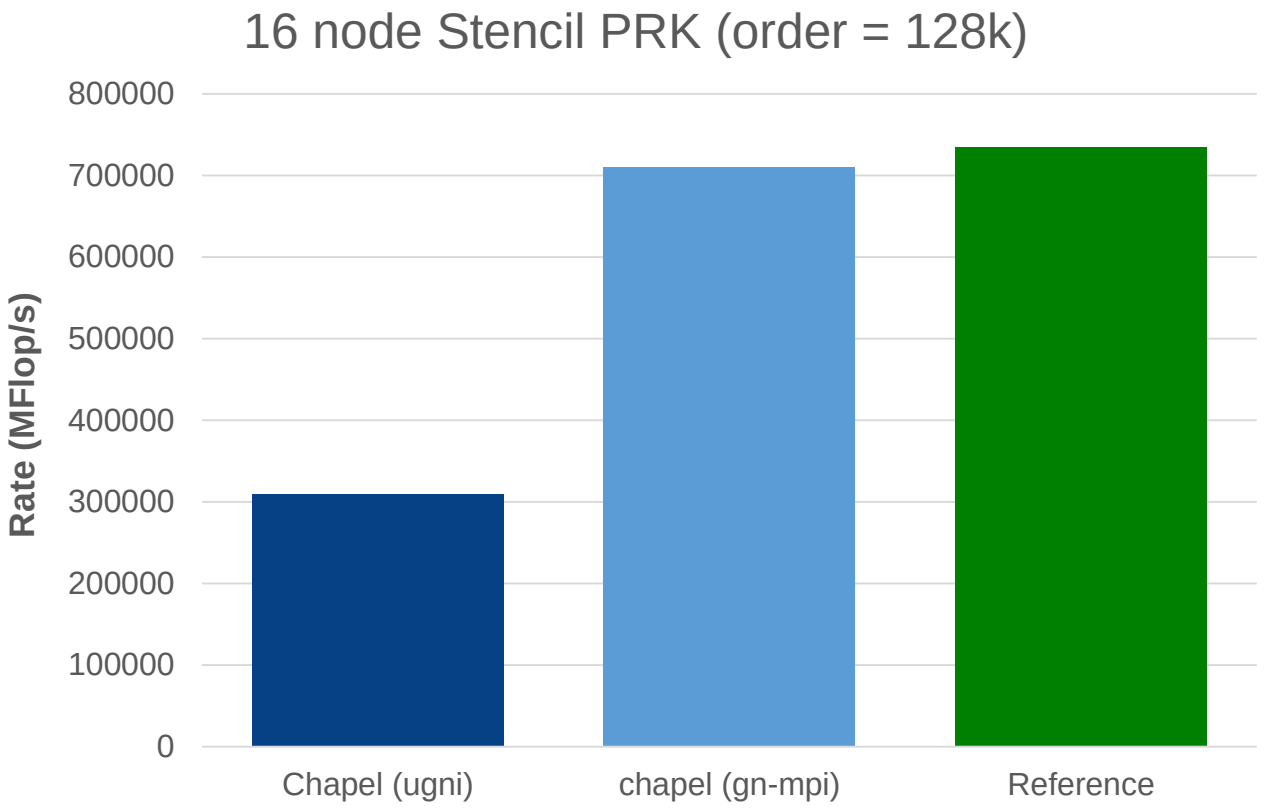
updateFluff: Stencil Impact

- **Previously Chapel was 2-4x slower than reference**



updateFluff: Stencil Impact

- **Previously Chapel was 2-4x slower than reference**
  - gn-mpi version is now on par with reference version (ugni still lagging)



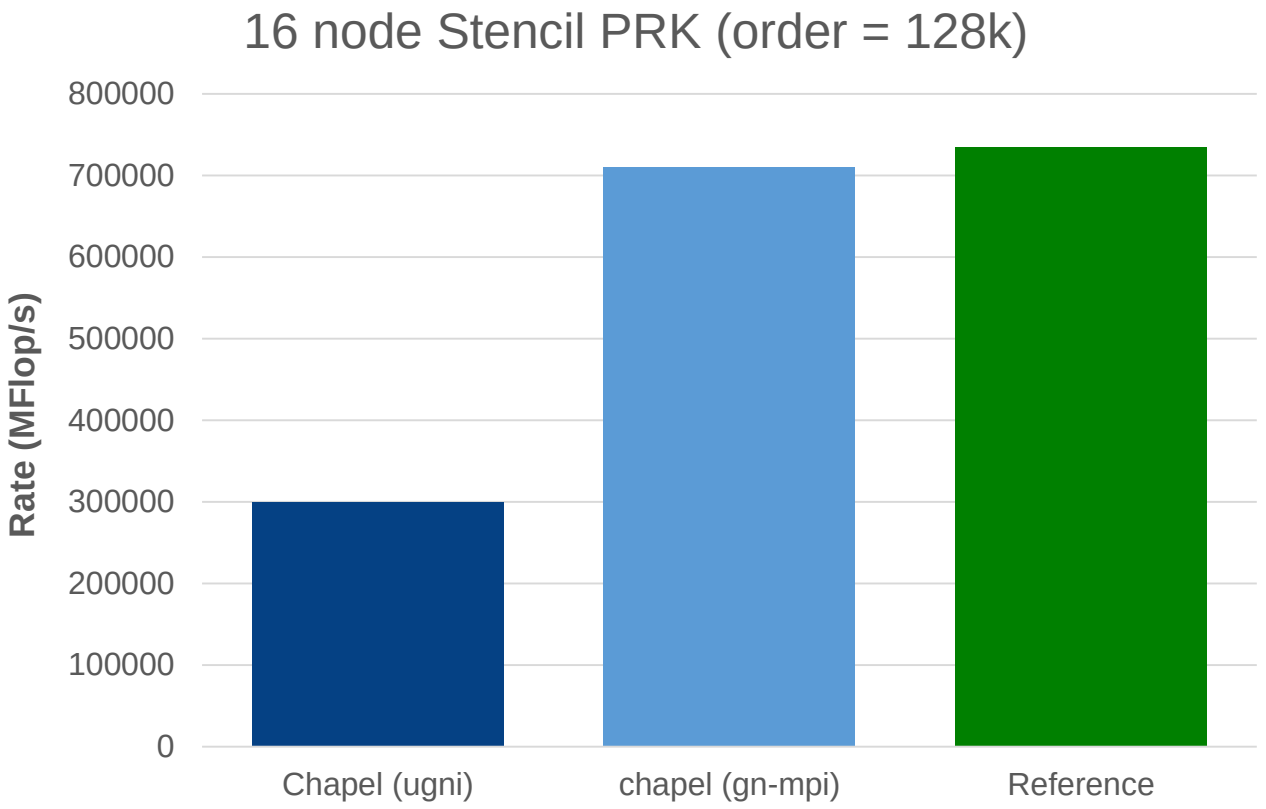
Dynamic Registration Impact

Dynamic Registration

- **Ugni performance was lagging behind gn-mpi**
  - Stencil PRK is sensitive to numa-affinity
- **Dynamic registration significantly improved performance**
  - Optimization detailed in runtime deck

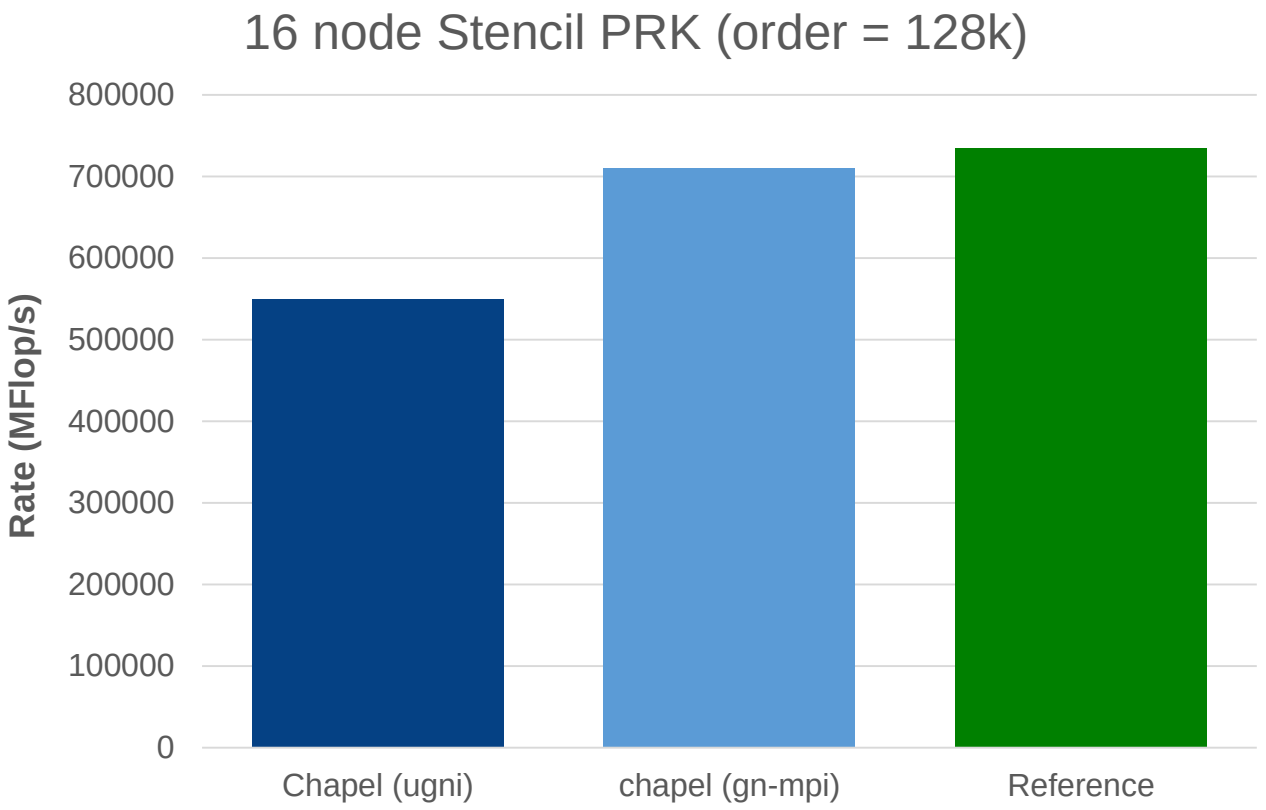
Dynamic Registration: Stencil Impact

- Previously ugni version was ~2.5x slower than reference



Dynamic Registration: Stencil Impact

- **Previously ugni version was ~2.5x slower than reference**
  - now 2x faster than before, though still behind reference and gn-mpi



Array Locality Optimization

Local Array Fields: Background

- **Optimized PRK used an inelegant local block**
  - Used to squash potential communication in array accesses

```
forall (i,j) in innerDom do local // instead of forall (i, j) in innerDom
```

- **"local field" pragma introduced in 1.11**
  - Marks a field as having the same locale as its parent
    - could be applied to classes, but not arrays

Local Array Fields: This Effort

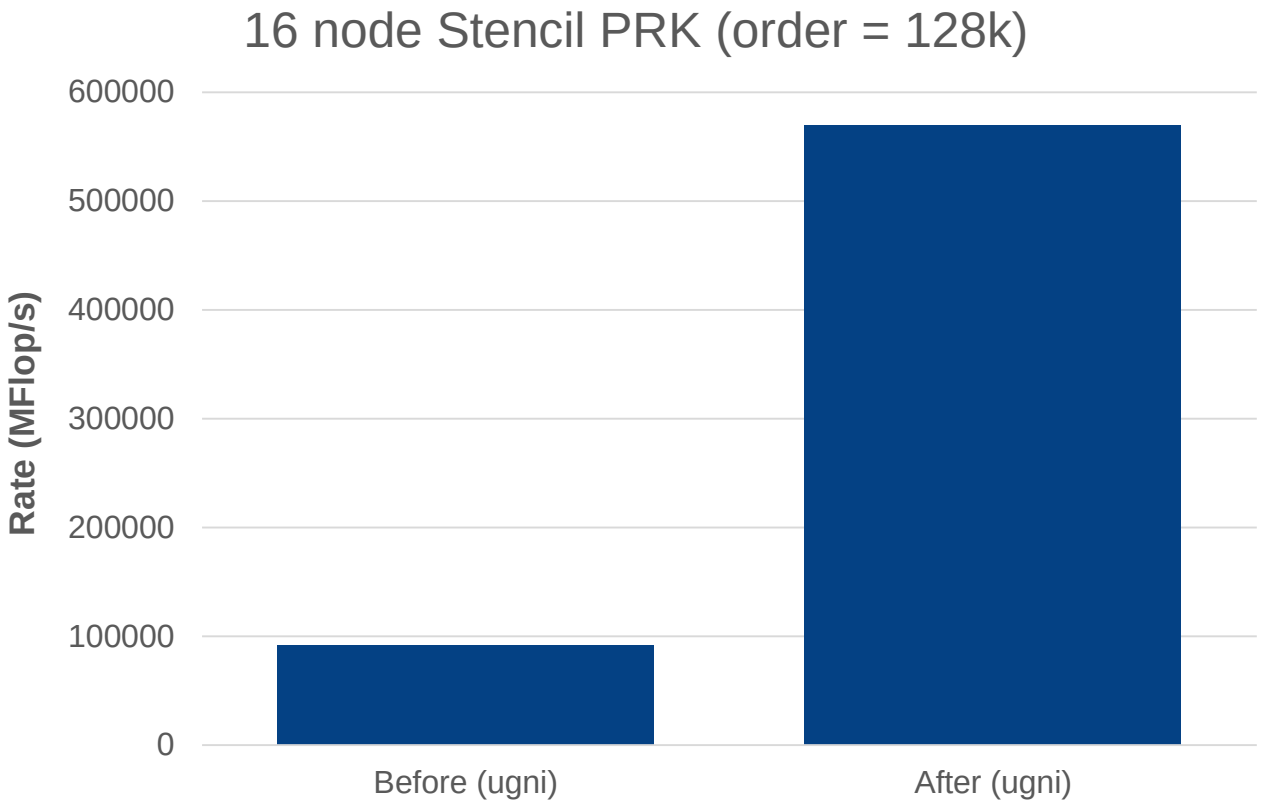
- Allow ‘local field’ pragma on arrays

```
class Wrapper {
  pragma "local field"
  var A : [1..10] int;
}
```

- Apply pragma in StencilDist, used by Stencil PRK
- Remove local block from optimized Stencil PRK

Local Array Fields: Impact

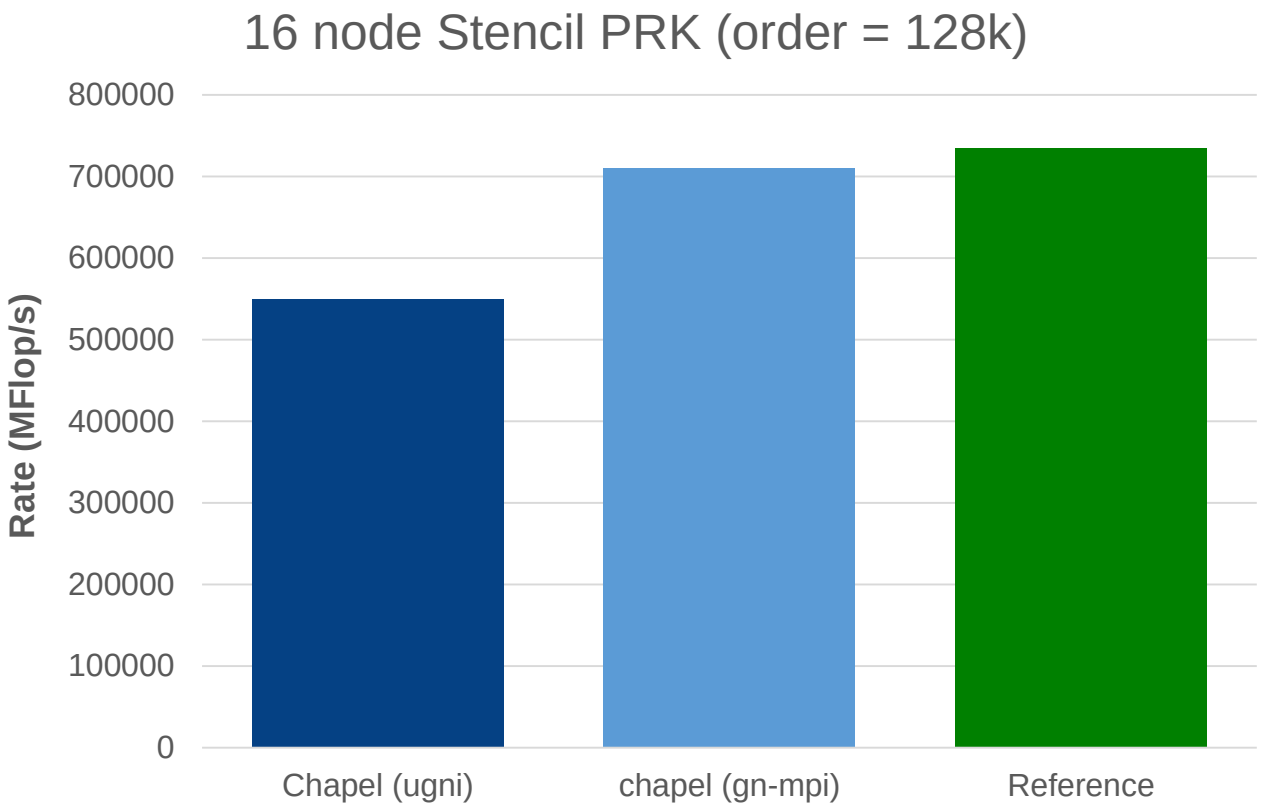
- **Performance without local block immensely improved**



PRK Summary

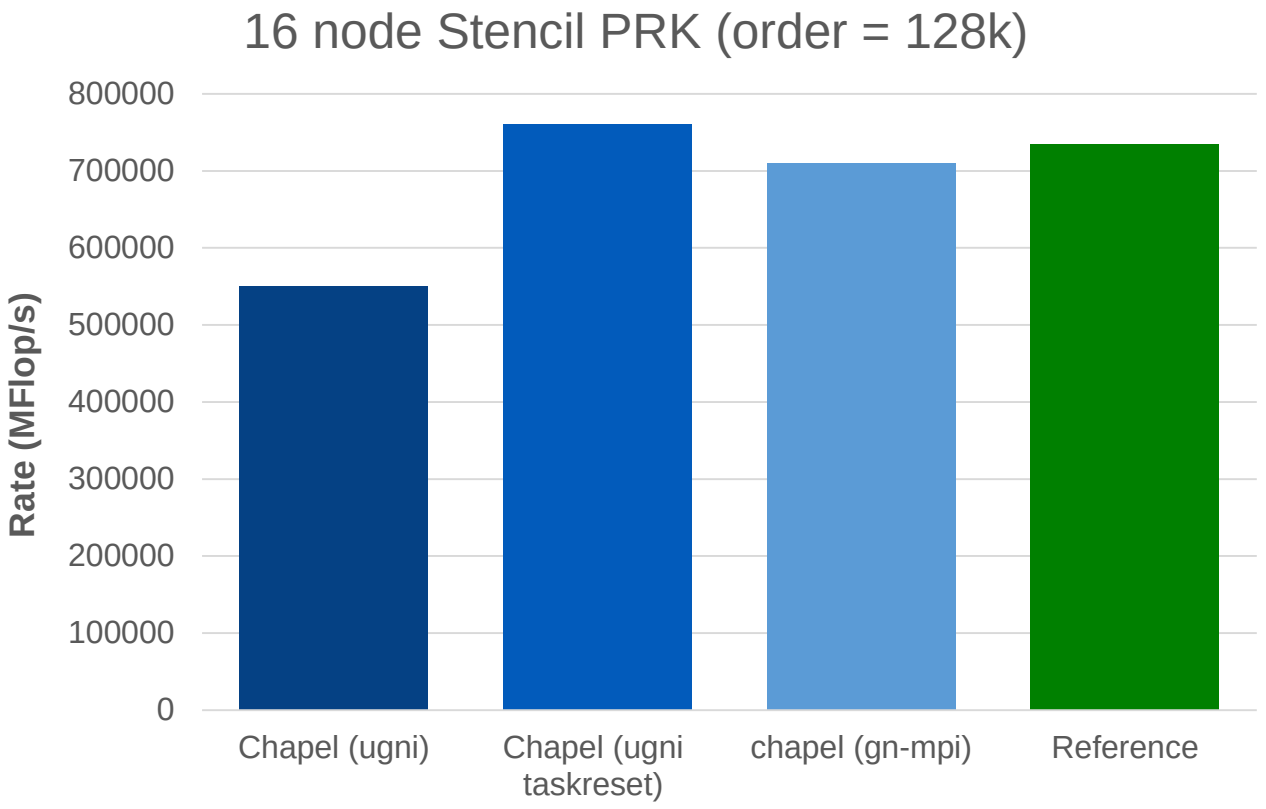
PRK: Summary

- **Optimized performance now mostly on par with reference**
  - Have removed some of the inelegant workarounds



PRK: Next Steps

- **Improve stencil performance for ugni comm layer**
  - Performance gap believed to be a result of poor task-affinity
    - experimental task-resetting work improves perf, but is not in 1.16



PRK: Next Steps

- **Reduce diff between elegant and optimized Stencil PRK**
  - use a local array view to avoid *localAccess* call
    - requires task-local variables. i.e. rewrite:

```
forall (i, j) in innerDom do
  in.localAccess[i, j];
```

as something like:

```
forall (i, j) in innerDom with (ref localIn=in.localView()) do
  localIn[i, j];
```

- **Continue to study PRKs**
  - Run PRK Stencil at larger scales
  - Explore additional PRKs

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: bucket-exchange is a common distributed pattern
- **Optimized version competes with the reference version**
  - But optimized version was slightly less elegant than we wanted

Record Serialization

Serialization: Background

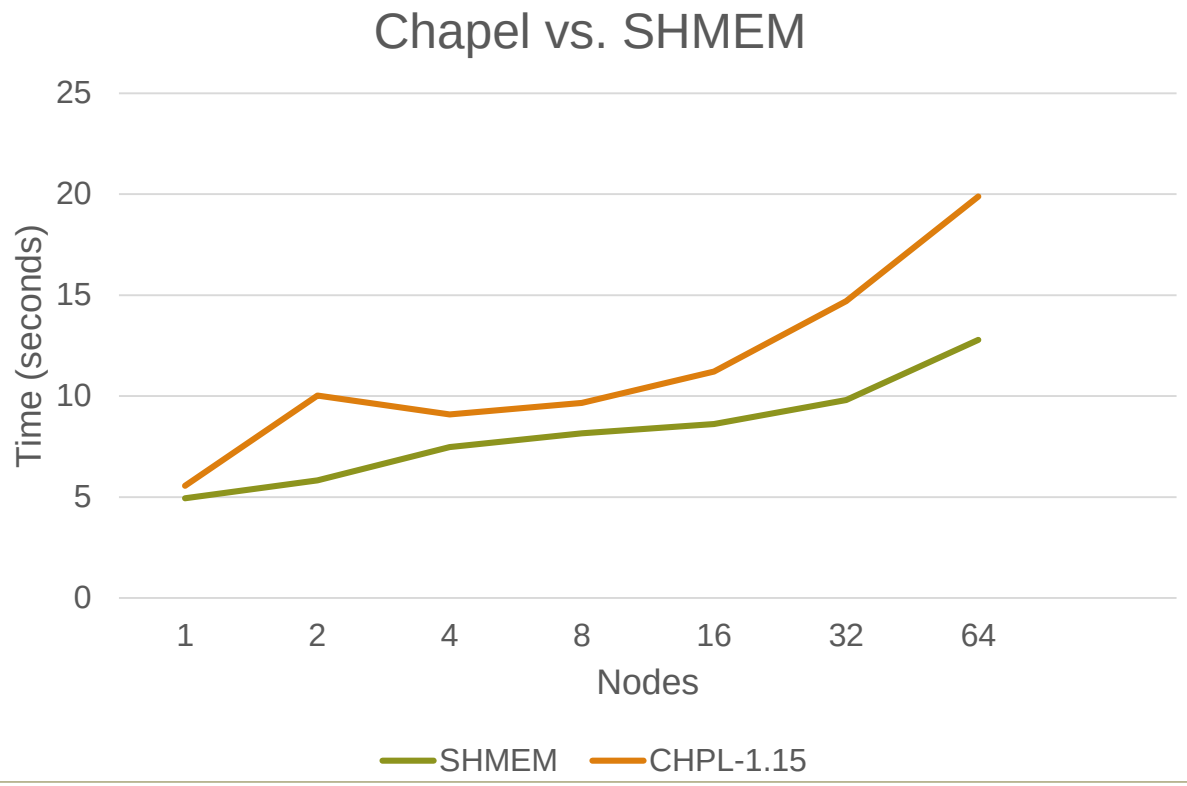
- **ISx declared task-local arrays over global const domain**

```
var myArray : [globalConstDom] int;
```

- **Domains must track their arrays**
  - If a domain is resized, related arrays must be resized as well
  - Uses on-statement to locale on which domain was created
  - Acquires a lock to update a list of arrays

Serialization: Background

- **Observed poor scaling in 1.15 compared to SHMEM**
  - All cores contending for lock on root locale
  - Note that the SHMEM version scales better than the MPI version

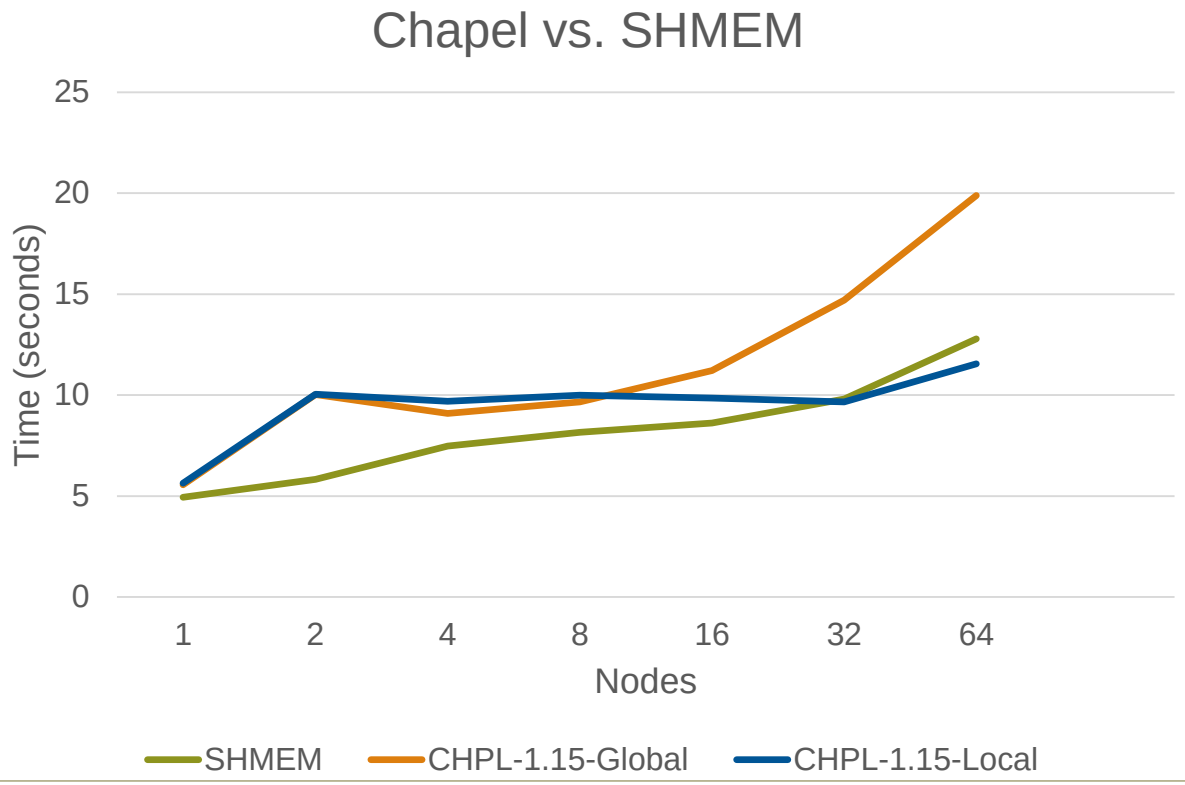


Serialization: Background

- **Temporary solution: use range literals**

```
var myArray : [1..n] int;
```

  - Avoids overhead because each array has its own local domain



Serialization: This Effort

- **Observation: global const domains can be replicated**
  - Their indices do not change, so they don't need to resize arrays
  - Locking restricted to intra-locale
- **Problem: existing replication performs shallow copies**
  - Domains implemented with records and classes
  - Need a way to replicate complex aggregate types
- **Solution: User-defined serialization across locales**

Serialization: This Effort

- **Implemented as methods on records:**
  - This method returns data necessary to recreate the record
    - Returns primitive type or record so it can be reclaimed later

```
proc myRecord.chpl__serialize() : X;
```

  - This type method accepts data and returns a record

```
proc type myRecord.chpl__deserialize(data : X) : myRecord;
```

- **Both methods required to trigger optimization**

Serialization: This Effort

- **Optimization triggers for local records**
  - If they can be remote value forwarded (sent as part of on-stmt)

```
const constR: R
 on Locales[1] {
  func(constR);
}
```

roughly converted into:

```
const constR: R;
var serialR = constR.chpl__serialize();
on Locales[1] /* serialR passed as part of arg bundle, no extra comm */ {
  const constR = R.chpl__deserialize(serialR);
  func(constR);
}
```

Serialization: This Effort

- **Optimization triggers for global const records**

```
const constR: R;
on Locales[1] {
  func(constR);
}
```

roughly converted into:

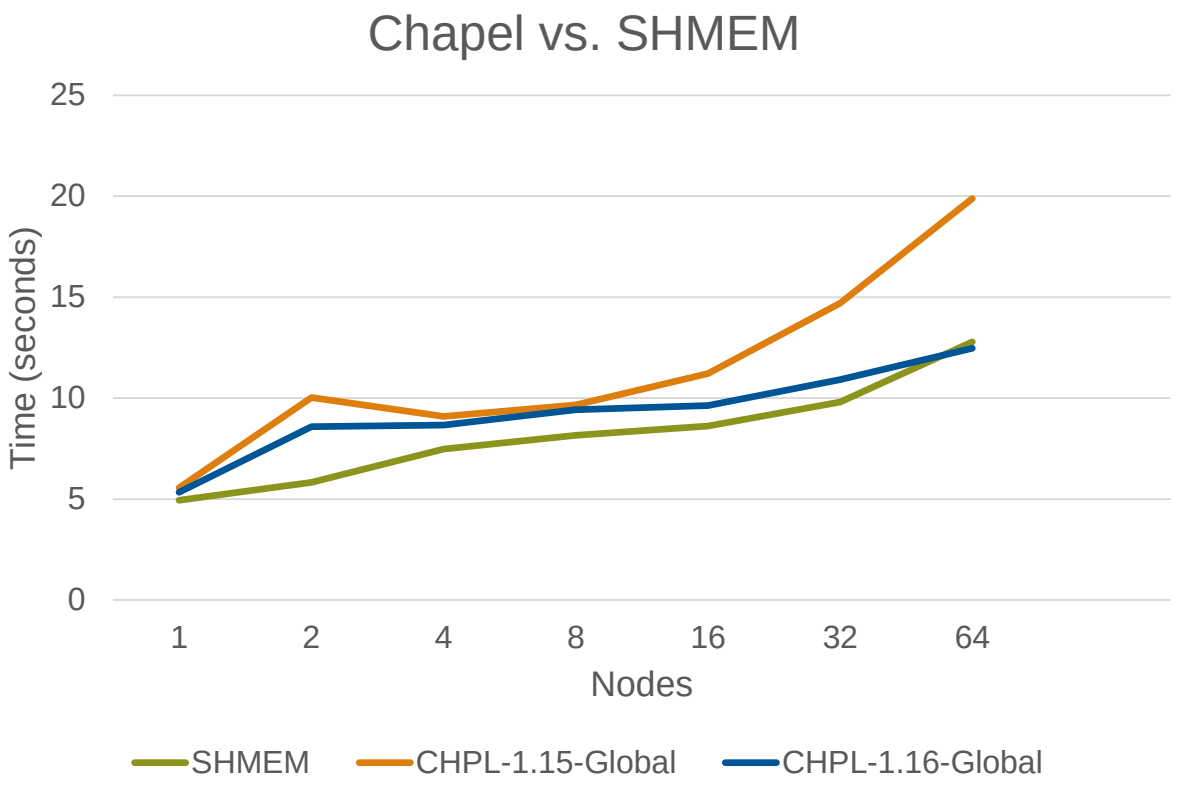
```
const constR: R;
bcastGlobal(constR); // serialize on loc 0, broadcast, deserialize on non-0 locs
on Locales[1] {
  func(constR); // locale-private copy, no communication required now
}
```

Serialization: This Effort

- **Implemented serialization for DefaultRectangular domains**
  - Very common and a good place to start
  - Can open to other domains as we gather experience
- **Also implemented serialization for strings**
  - Long-desired optimization
  - Unfortunately we lack distributed string benchmarks for comparison
  - Should still serve as a good stress-test for the optimization
- **Controlled by '--[no-]remote-serialization' flag**
  - Optimization is on by default

Serialization: Impact

- Can write ISx more elegantly without performance loss



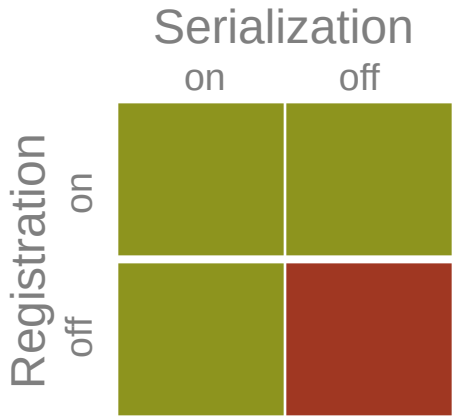
Serialization: Next Steps

- **Avoid serialization for local on-statements**
  - May add overhead depending on user's implementation
- **Standardize interface and semantics**
  - Current implementation mainly intended for internal use

Serialization - Supplemental

Serialization: Supplemental

- **Dynamic registration also helped the ISx scaling issue**
  - Unexpected result
  - "Resolves" scaling issue without serialization enabled
- **Hypothesis: Dynamic registration reduced lock contention**
  - Measured less time spent locking under dynamic registration
  - Something in dynamic registration code may be locking as well
    - This may offset tasks such that contention is less likely
- **Next Steps:**
  - Confirm hypothesis



Task Counting Improvements

Task Counting: Background

- **Running task count is used to determine forall parallelism**
  - Want to utilize all cores, without oversubscribing the system

```
forall i in 1..n do                 // should create here.maxTaskPar tasks
  forall j in 1..n do               // should not create any additional tasks
```

- **An inaccurate running task counter hurts performance**
  - Can result in too few or too many tasks being created

- **Task counting for tasks migrated via on-stmts was wrong**
  - Did not decrement local counter for moved tasks

```
on Locales[numLocales-1] do body();   // did not decrement locale 0
```

  - Did not track remote tasks for non-blocking ons

```
coforall loc in Locales do on Loc do
  // running task count was 0 for all non-0 locales
```

Task Counting: Background

- **Improper task-counting hurt ISx**

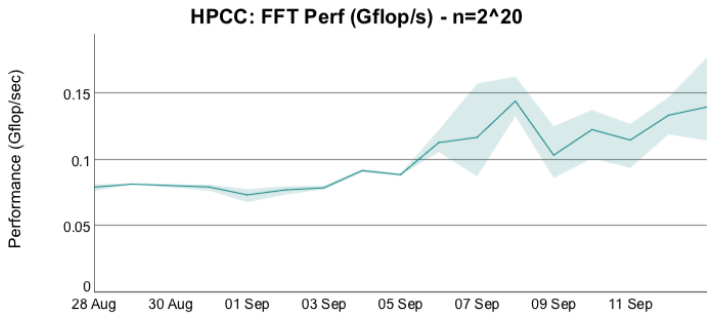
```
coforall loc in Locales do on Loc do
  coforall tid in 0..#perBucketMultiply {
    // task counter was numCores-1 instead of the true numCores
    ...
    // resulted in parallel init for some myKeys, which hurt affinity in later uses
    var myKeys: [0..#keysPerTask] keyType;
    ...
  }
```

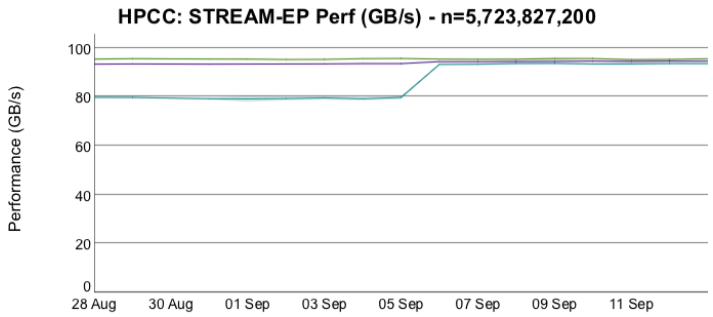
- **Required an ugly workaround to disable parallel array init**
  - *-schpl_defaultArrayInitMethod=ArrayInit.serialInit*

Task Counting: This Effort and Impact

- **This Effort: Improved accuracy of the running task counter**
  - Now correctly track running tasks for migrated tasks
  - Added a significant number of tests to lock-in behavior
    - inaccurate task-counting has been a longstanding and recurrent issue

- **Impact: Improved multi-locale performance**
  - Allows us to remove array initialization workaround for ISx

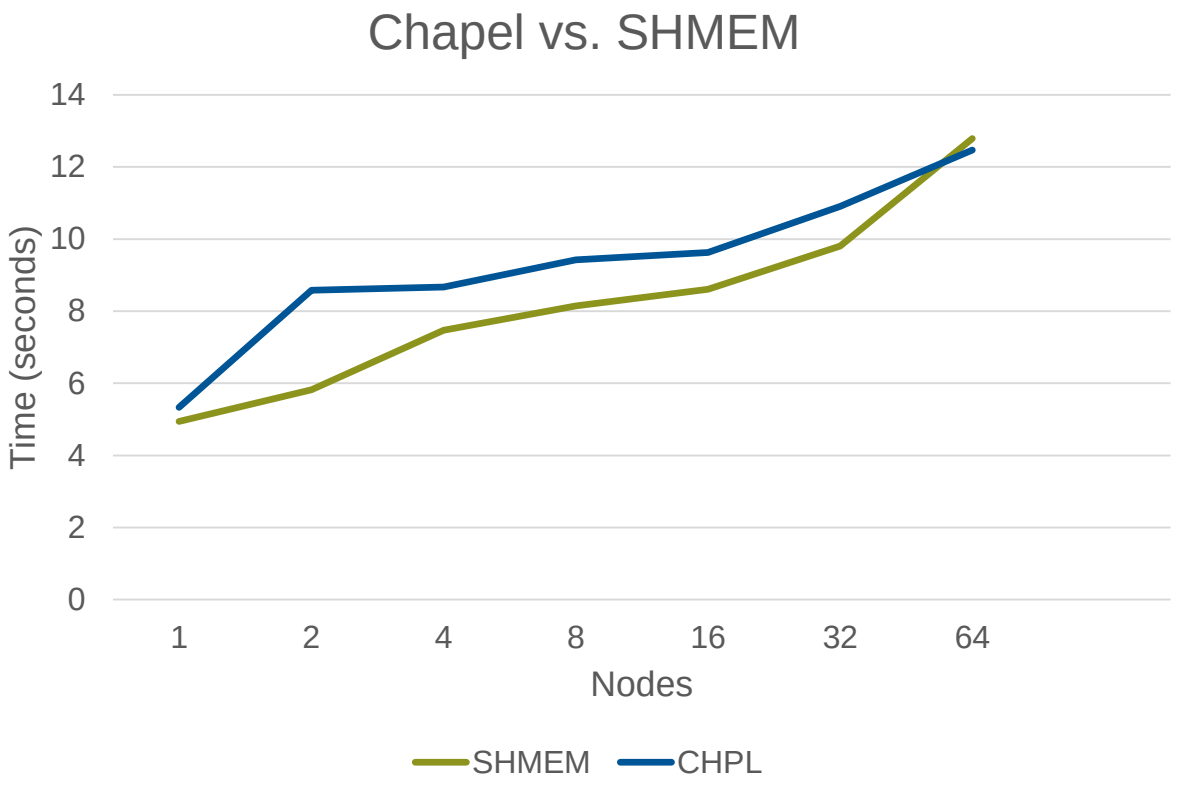




ISx Summary

ISx: Summary

- **ISx still performing on par with reference**
  - Previous inelegant workarounds have been removed



ISx: Next Steps

- **Improve ISx performance at smaller scales**
  - Performance slightly exceeds reference at 64 nodes
  - But is slightly behind at smaller node counts
- **Run ISx at even larger scales**
  - Identify and fix any scaling bottlenecks

Computer Language Benchmarks Game (CLBG) Update

CLBG: Background

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…

CLBG: Background (Chapel's Approach)

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: Background (Results)

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					100% 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: Background (Pair-wise Comparisons)

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: Background (Browsing Programs)

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: This Effort

This Effort:

- No real focus on CLBG improvements this release cycle
  - most performance work focused on distributed memory benchmarks
- Some minor changes to released versions of CLBG programs:
  - added faster mandelbrot & chameneos versions to the examples directory
    - described in 1.15 release notes
    - had already been submitted to CLBG site

    `examples/benchmarks/shootout/mandelbrot-fast.chpl`

    `examples/benchmarks/shootout/chameneos-fast.chpl`
  - changed knucleotide to use the default parSafe mode
    - enabled by removal of locking from associative array accesses
    - minor elegance / code size improvement, no performance impact
    - not yet submitted to CLBG site

    `examples/benchmarks/shootout/knucleotide.chpl`

CLBG: Impact

Impact: Overall, no major changes

- Chapel execution times / ratios largely the same
- A few slips in rank / ratio, largely due to new entries being submitted
  - particularly for Rust, C#, F#, Java

8 / 13 programs in top-25 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-25 smallest:
  chameneos-redux
  mandelbrot
  meteor-contest

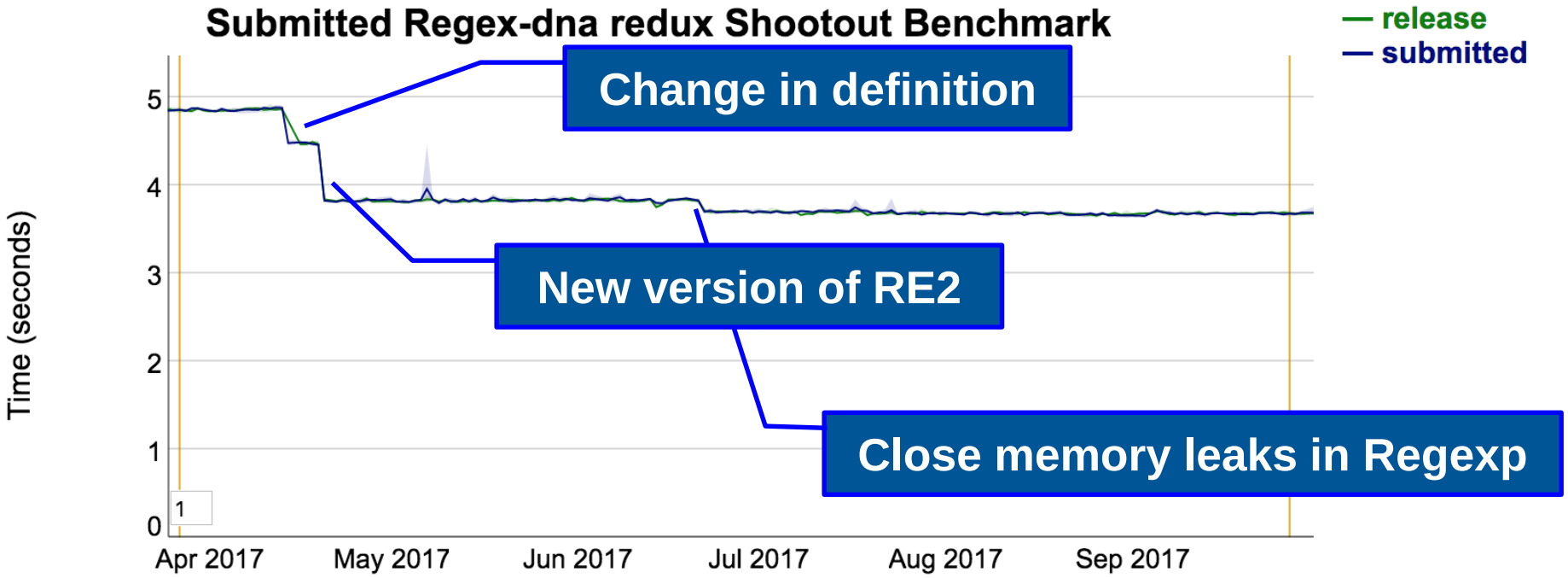
12 / 13 programs in top-25 fastest:

- one #1 fastest:
  pidigits
- 3 others in the top-5 fastest:
  chameneos-redux
  meteor-contest
  thread-ring
- 2 others in the top-10 fastest:
  fannkuch-redux
  fasta
- 6 others in the top-25 fastest:
  binary-trees
  k-nucleotide
  mandelbrot
  n-body
  regex-redux
  spectral-norm

CLBG: Impact (Regex-redux)

Impact: Regex-redux saw the most significant improvements

- primarily due to a new version of RE2
- to a lesser extent, due to closing memory leaks
- also changed its definition during this release cycle

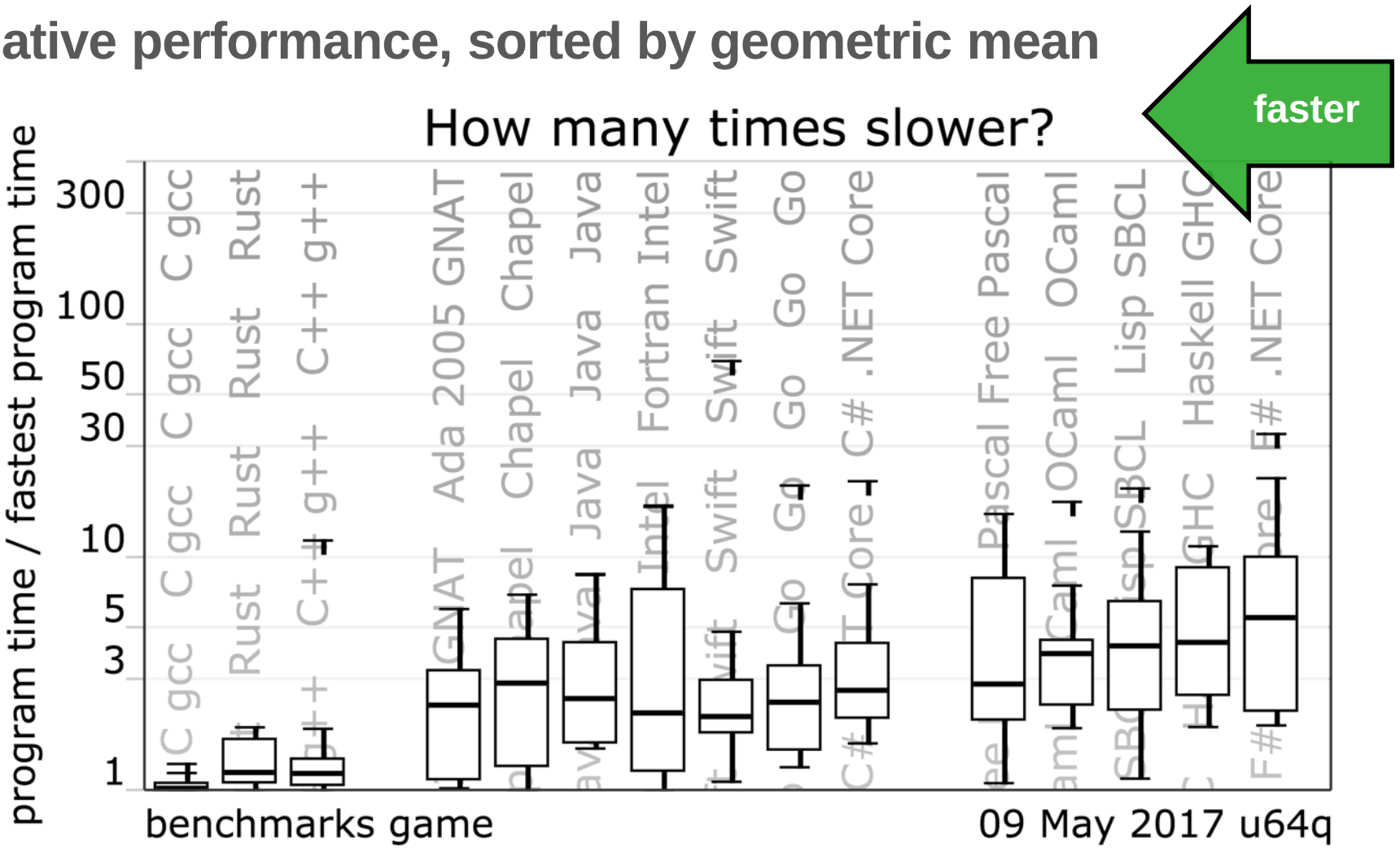


CLBG: Status (Notes on the following graphs)

- **Graphs that follow are taken from the CLBG website**
  - each column summarizes the fastest programs for a given language
  - threadring, chameneos-redux, meteor-contest are not included
  - sorted by geometric mean execution time, scaled to fastest entries
  - horizontal line indicates mean execution time
  - box indicates one standard deviation
  - whiskers indicate two standard deviations
  - additional whiskers indicate outliers

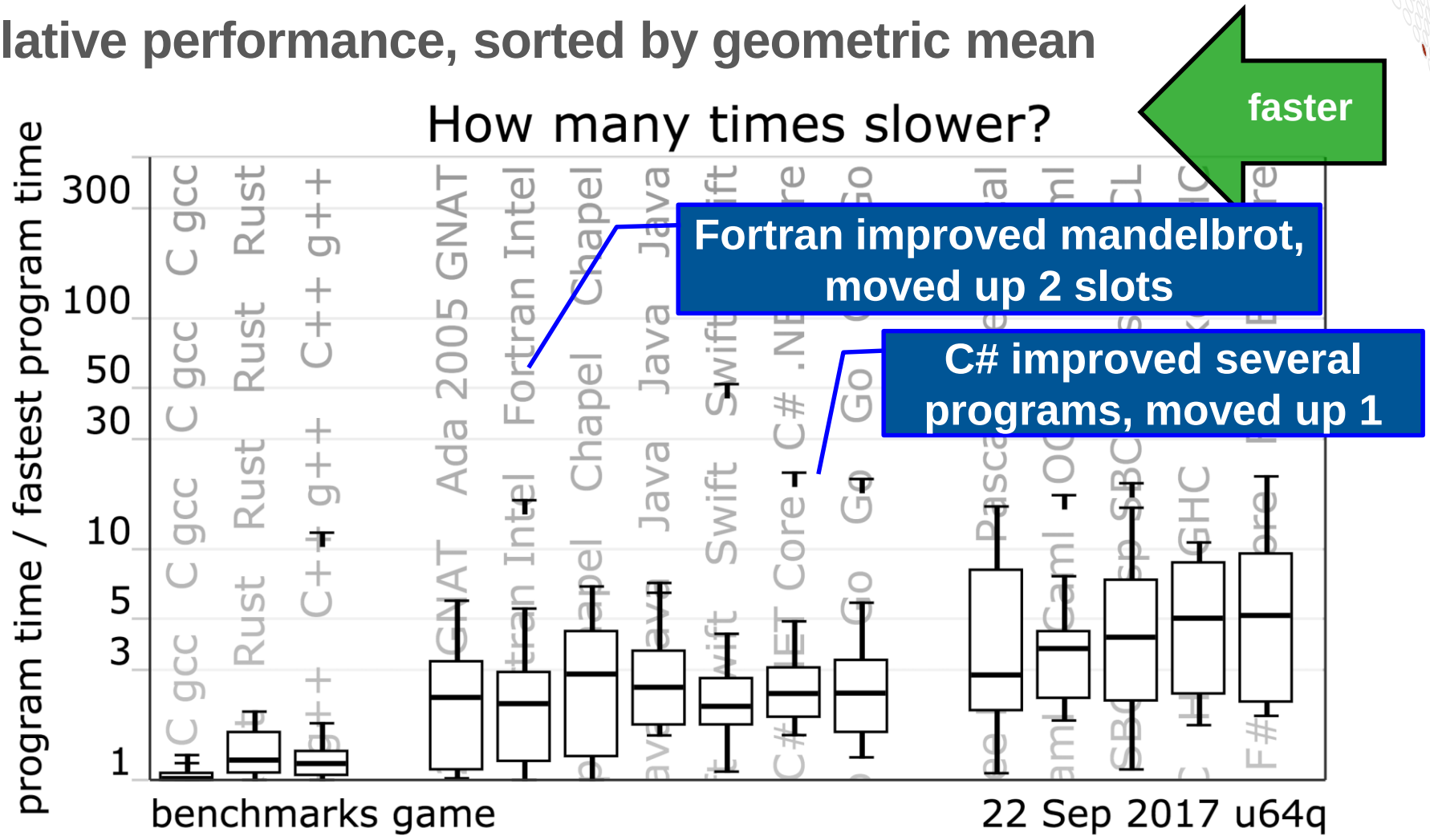
CLBG: Fast-faster-fastest graph (May 2017)

Relative performance, sorted by geometric mean



CLBG: Fast-faster-fastest graph (Sept 2017)

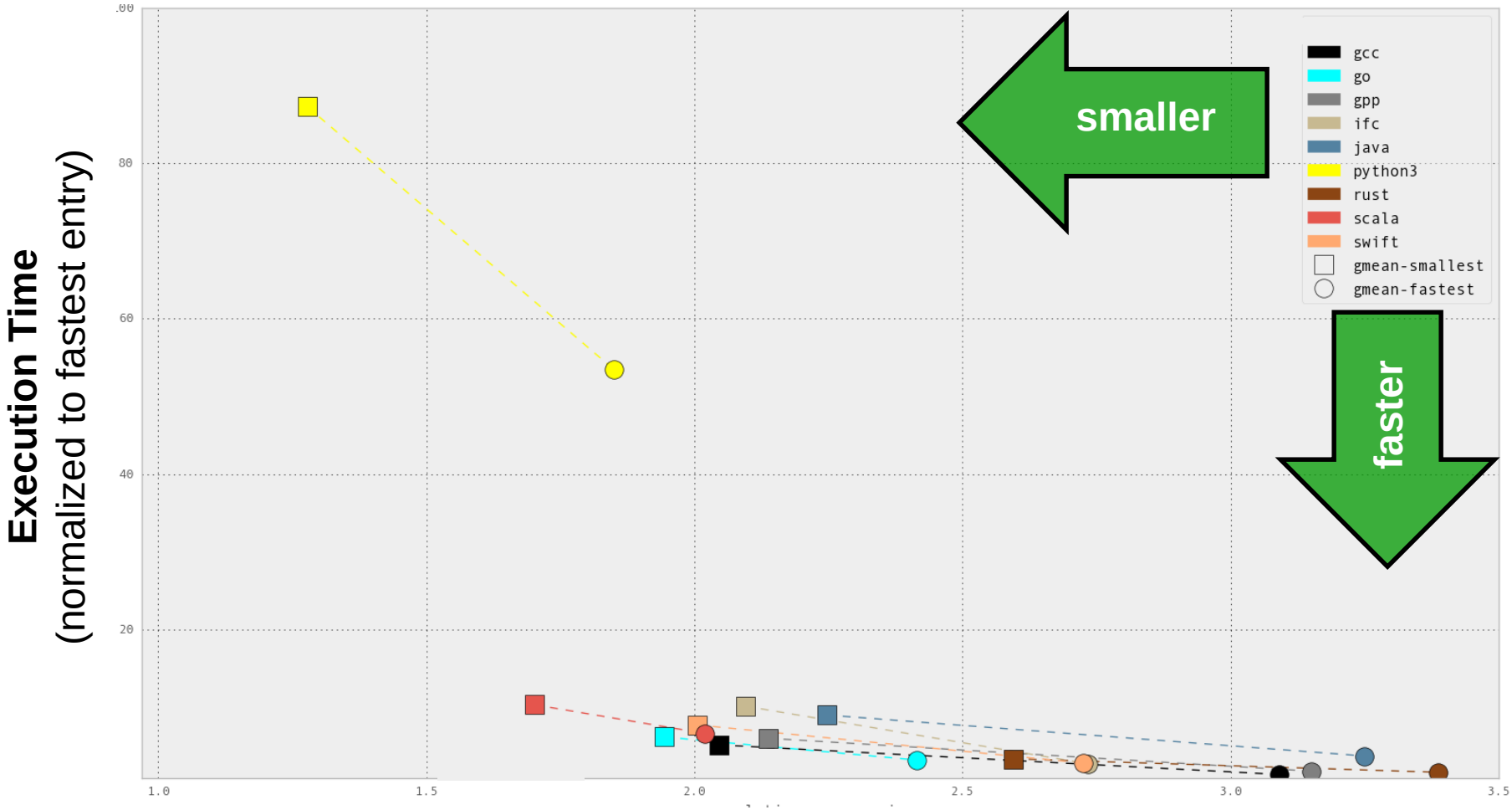
Relative performance, sorted by geometric mean



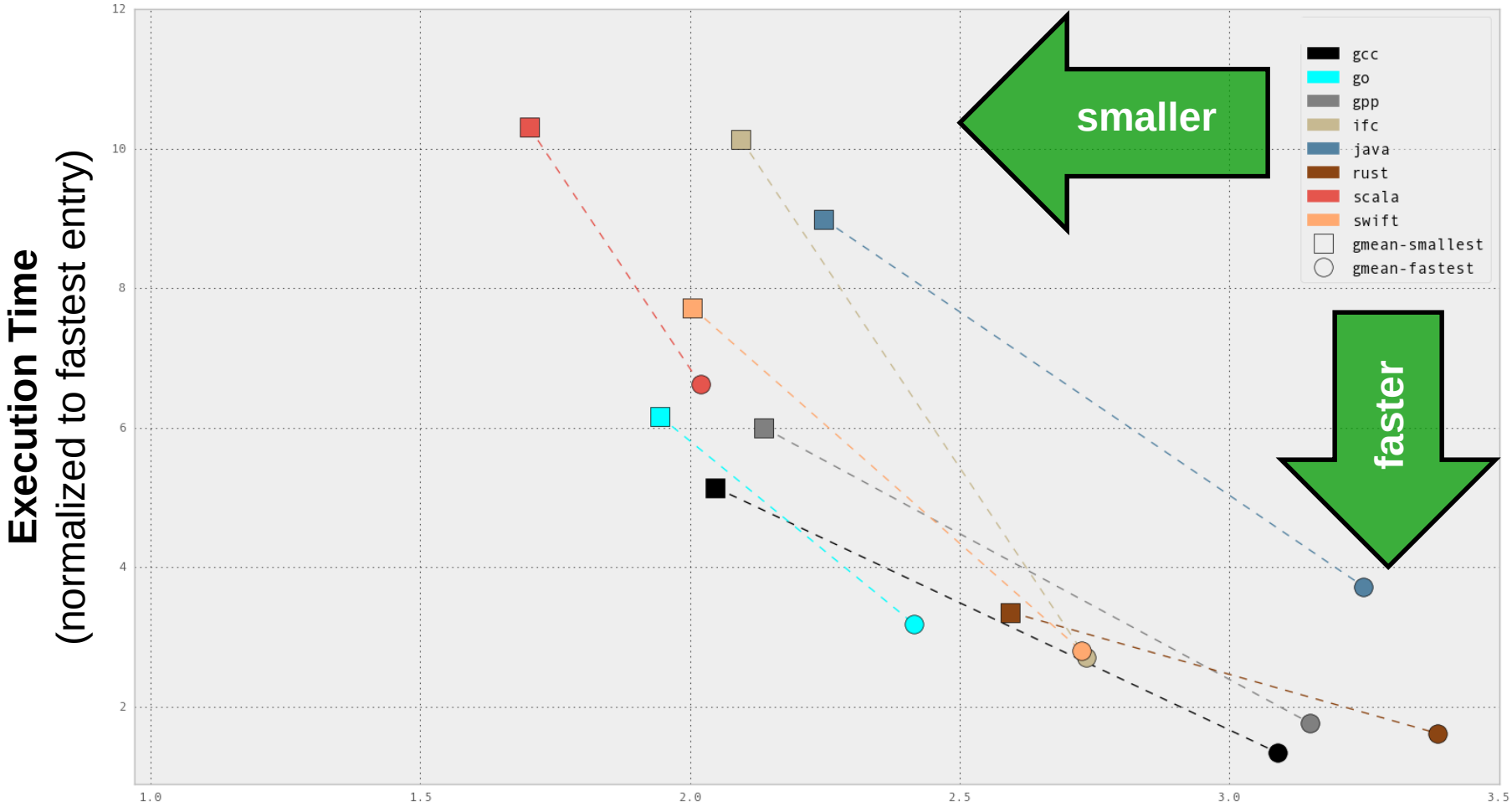
CLBG: Status (Notes on the following plots)

- **Graphs that follow are generated by us**
  - plot results taken from the CLBG site on Oct 18, 2017
  - all benchmarks are included (unlike the previous graphs)
    - rationale: Chapel cares about task-parallelism
  - x-axis shows normalized compressed code size (gz metric from site)
  - y-axis shows normalized execution time
  - each language represented by a pair of points:
    - geometric mean of fastest entries in each language shown via circle
    - geometric mean of smallest entries in each language shown via square
    - line connects the two points
    - if either point falls outside the graph, point and line are not shown (TODO)

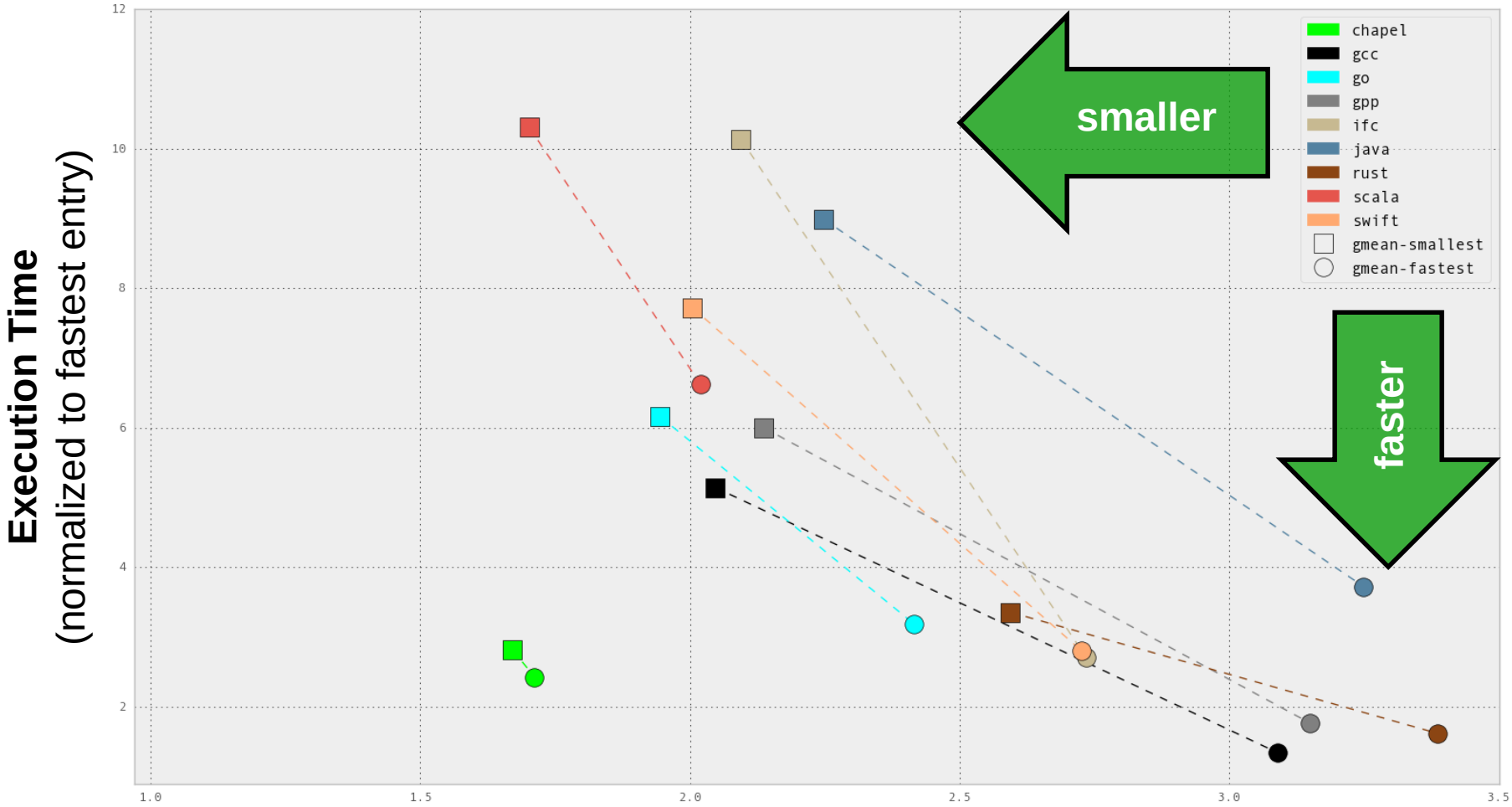
CLBG Language Cross-Language Summary
(May 2017 standings, languages of interest)



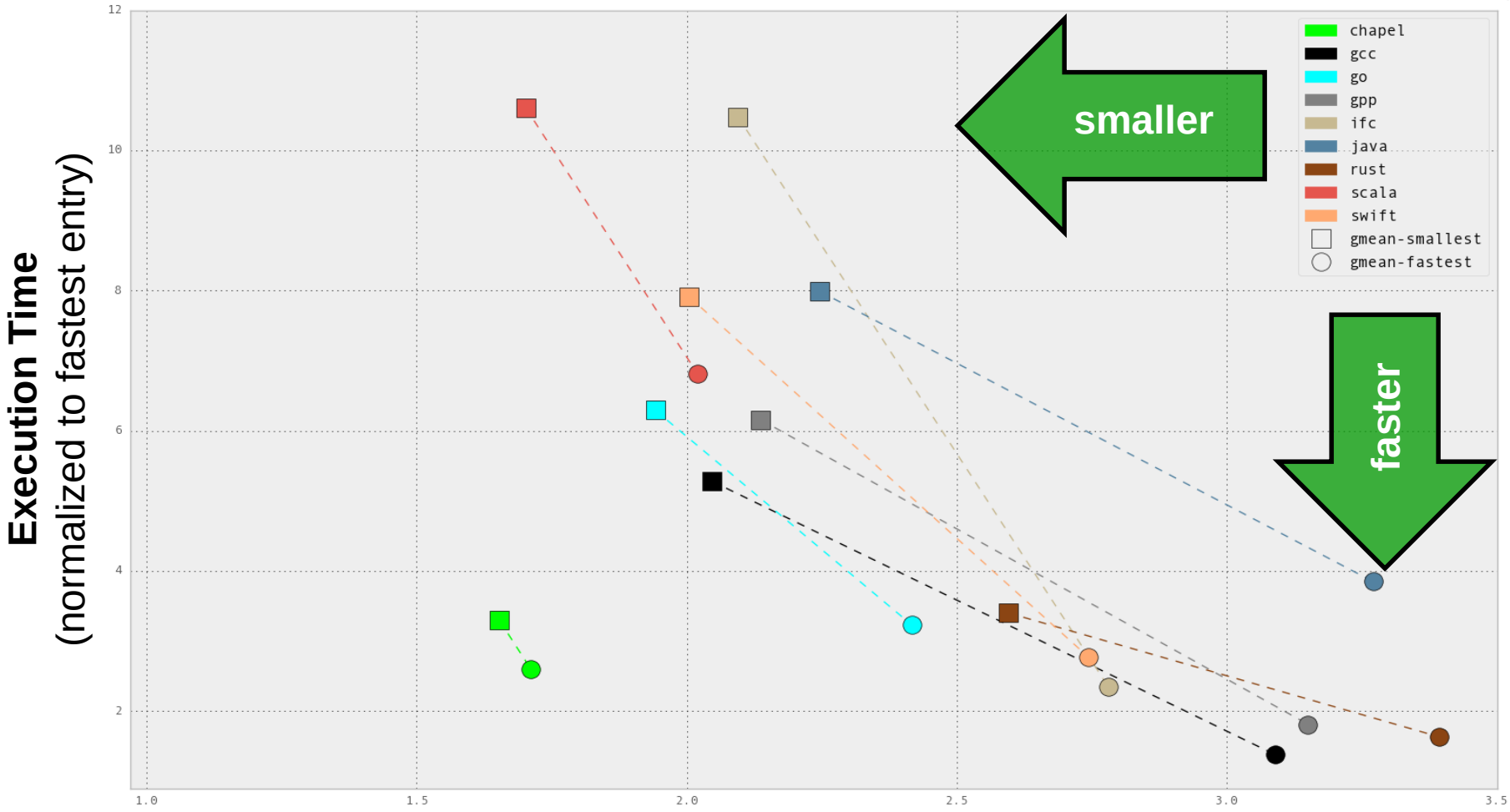
CLBG Language Cross-Language Summary
(May 2017 standings, without Python)



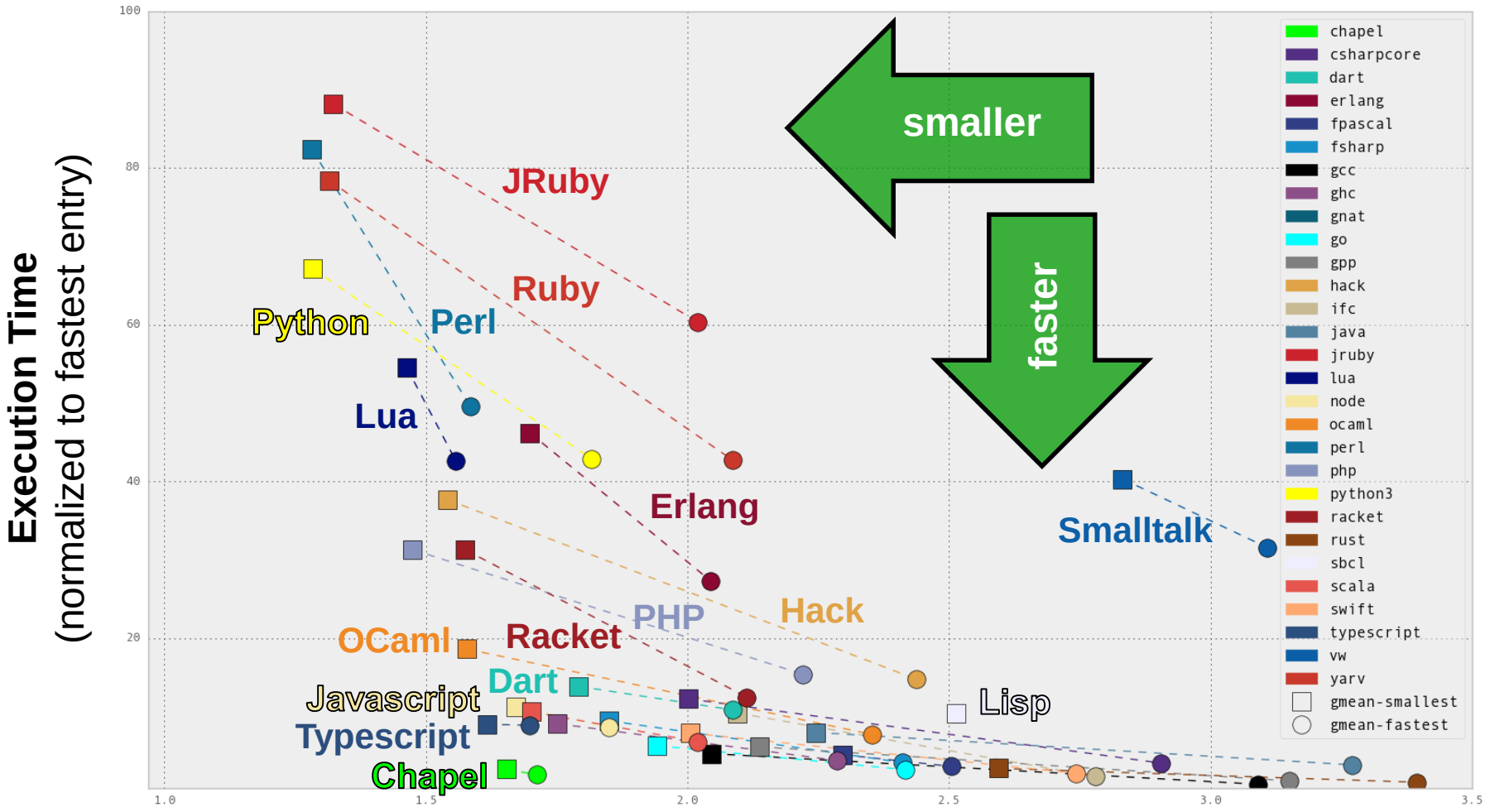
CLBG Language Cross-Language Summary
(May 2017 standings, with Chapel)



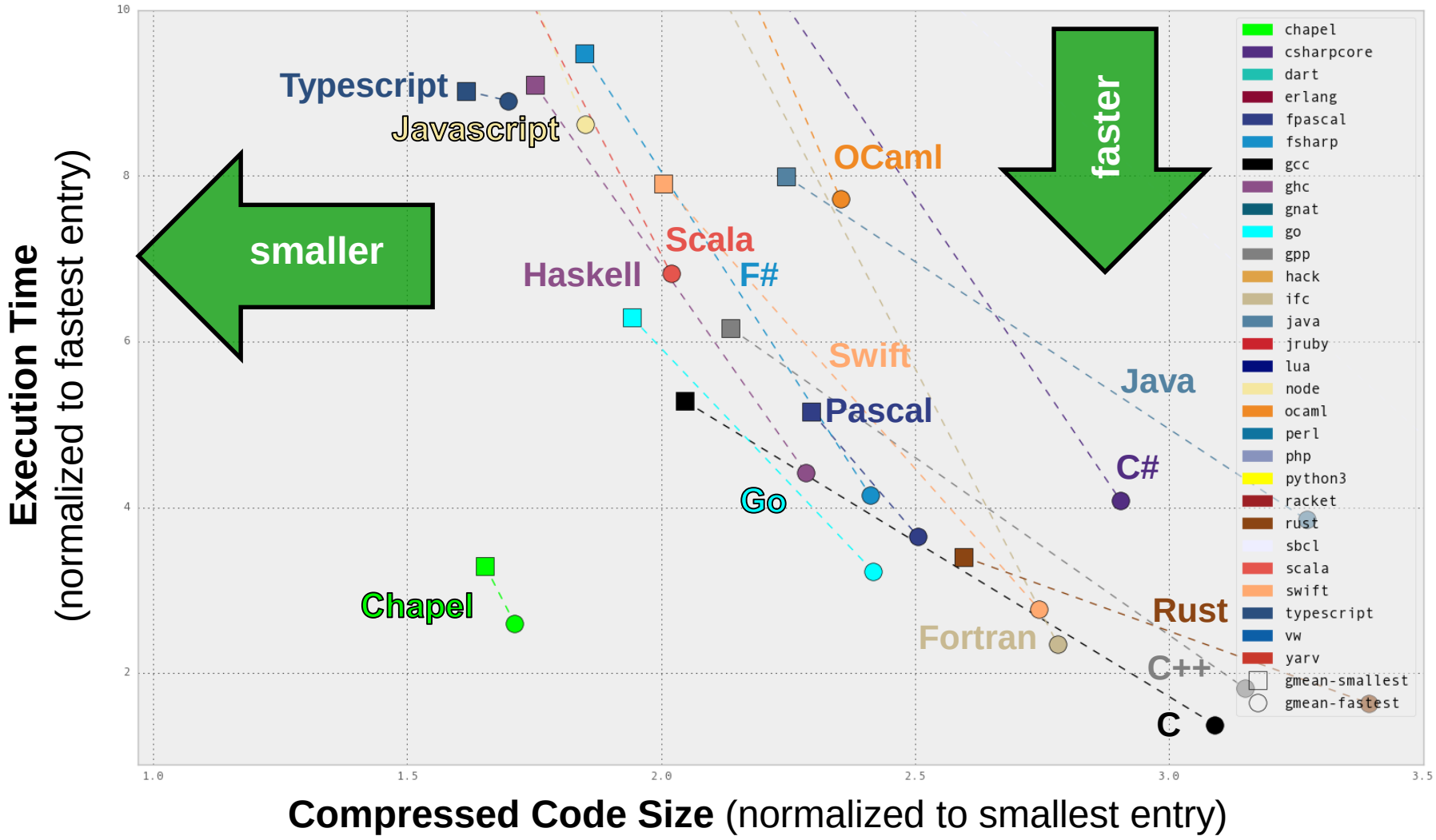
CLBG Language Cross-Language Summary
(Oct 2017 standings, with Chapel)



CLBG Language Cross-Language Summary
(Oct 2017 standings, all languages)



CLBG Language Cross-Language Summary
(Oct 2017 standings, all languages, zoomed in)



Reductions in Memory Leaks

Memory Leaks: Background

Background:

- Past few releases have closed major sources of leaks
  - Leaking of record fields due to missing destructor calls
  - Leaking of arrays due to bad memory management
- Postulated that most remaining leaks were in user-level code
  - e.g., tests that allocate without deleting:

```
var myC = new C();     // test invocation of initializer
// program fails to delete myC so leaks it…
```

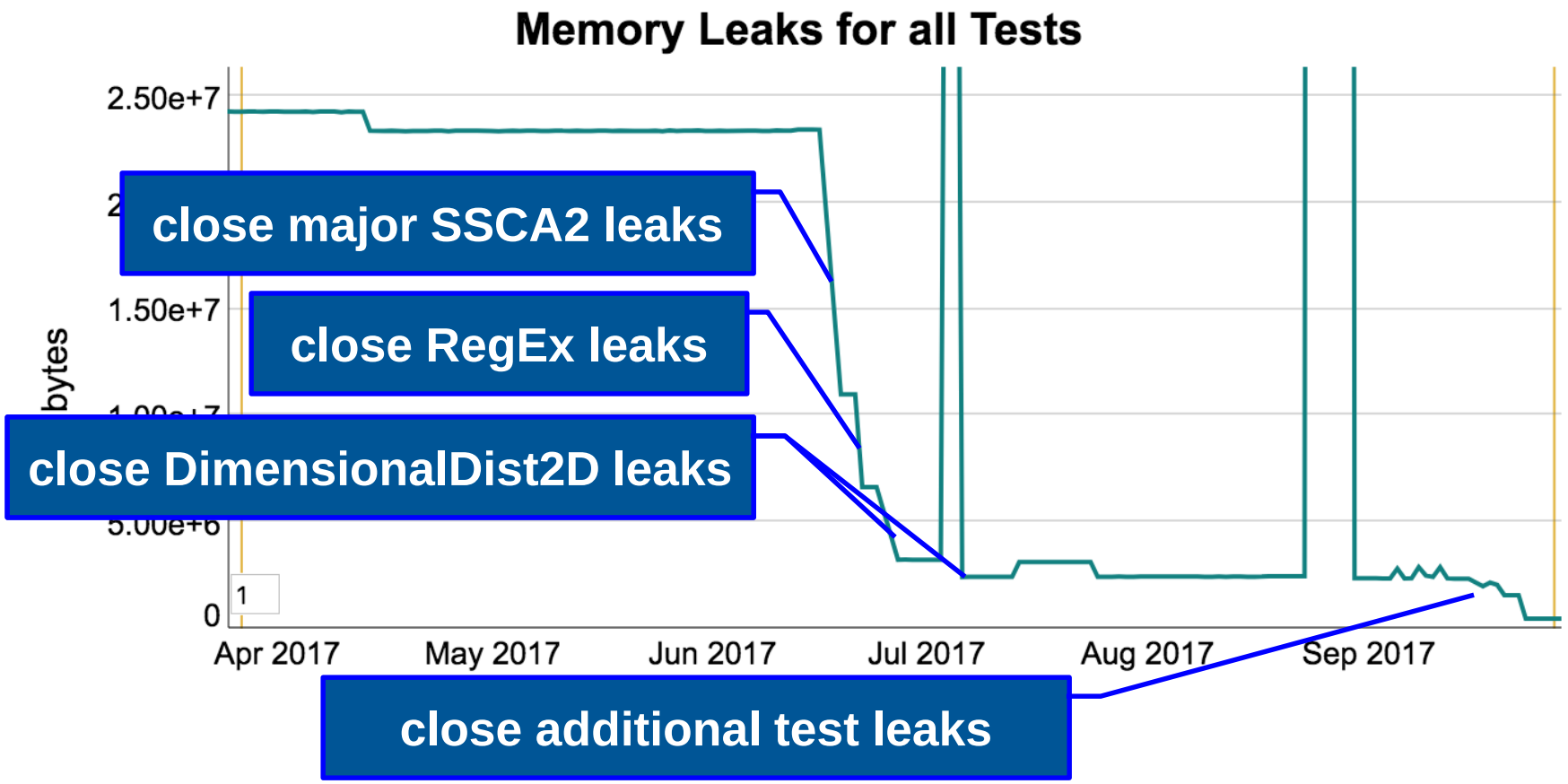
Memory Leaks: This Effort

This Effort: Continued to close memory leaks

- closed many significant test-based leaks:
  - SSCA2
  - AMR
  - Graph500
  - vertex coloring
  - bulk comm stencil tests
  - fock
  - NAS EP
  - label propagation
  - lsms
- also closed a few leaks in our library modules:
  - string leaks in Regex.subn(), qio_regexp_replace()
  - DimensionalDist2D leaks

Memory Leaks: Impact

Impact: 50x reduction in memory leaks



Memory Leaks: Next Steps

Next Steps:

- Close remaining leaks
  - Still believe user code is responsible for most remaining leaks
  - However, likely to be some remaining library-based leaks as well
- Tighten up nightly testing to prevent new leaks from being introduced

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