Performance Optimizations

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

- **Array Optimizations**
  - Array Indexing Optimization
  - Promoted Fast Followers Improvements
  - Strided Bulk Transfer
  - Array-as-vec Improvements
- **Locality Optimizations**
  - Wide Pointer Analysis Improvements
  - Reducing Task Counting Overhead
  - Local On Statements Optimization
- **Qthreads Improvements**
  - Native Qthread Sync Vars
    - Reduction Lock Improvements
  - Qthreads “Distrib” Scheduler
- **Runtime Optimizations**
  - Jemalloc Changes
  - Faster Complex .re/.im
- **Other Performance Optimizations**

Array Optimizations

Array Indexing Optimization

Array Indexing Opt: Background

- **Chapel arrays are significantly richer than C/C++ arrays**
  - first-class language concept with support for:
    - non-0 based indexing, slicing, rank changing, and much more

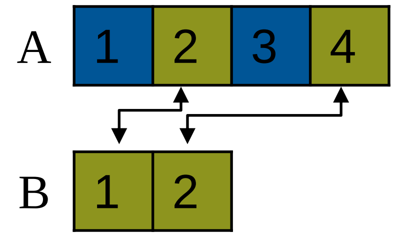
- **Historically, these features had a performance cost**
  - previous optimizations have lessened much of the performance cost
    - shifted base pointer optimization, loop invariant code motion, etc.

- **A remaining cause of overhead was a multiply for indexing**
  - multiply is only needed for some specific/rare use-cases
    - rank-change, re-indexing, strided slice aliases
  - but all arrays were paying the price

Array Indexing Opt: Background

- **For example, multiply needed for a strided slice alias**

```
var A: [1..4] int;
var B: [1..2] => A[2..4 by 2];
```



- B is an alias to a subset of A (B is a "view" into A's data)
  - logically, accesses of B are translated to accesses of A

indexing is translated with a "blk" offset

`B[2]` => A[2 * blk] => A[2 * 2]

previously, multiply occurred for all arrays

`A[2]` => A[2 * blk] => A[2 * 1]

Array Indexing Opt: This Effort

- **“Array Views” will remove multiply in a principled manner**
  - but that work wasn’t completed in time for 1.14

- **Added a simple compiler optimization in the interim**
  - removes inner multiply, when it can prove no array needs it
    - i.e. if even one array requires it, all arrays will have the inner multiply
    - unfortunate, but few programs require it, and it’s better than the status quo

Array Indexing Opt: Impact

- **Now generating ideal 1D array indexing code**

```
for i in 1..10 do
  A[i] = i;
```

Used to generate

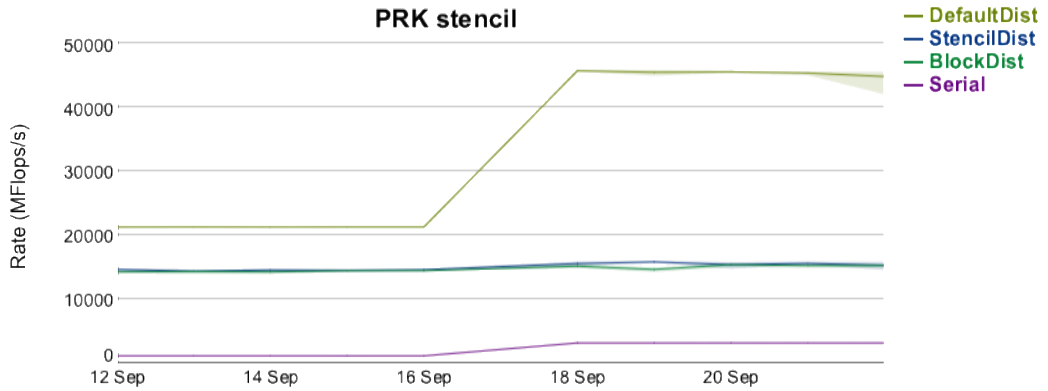
```
for (i = INT64(1); i <= INT64(10); i += INT64(1)) {
  index = INT64(0);
  rank1_index = (i * blk);    // blk was always 1
  index += rank1_index;
  elem_ptr = (arr_base + index);
  *(elem_ptr) = i;
}
```

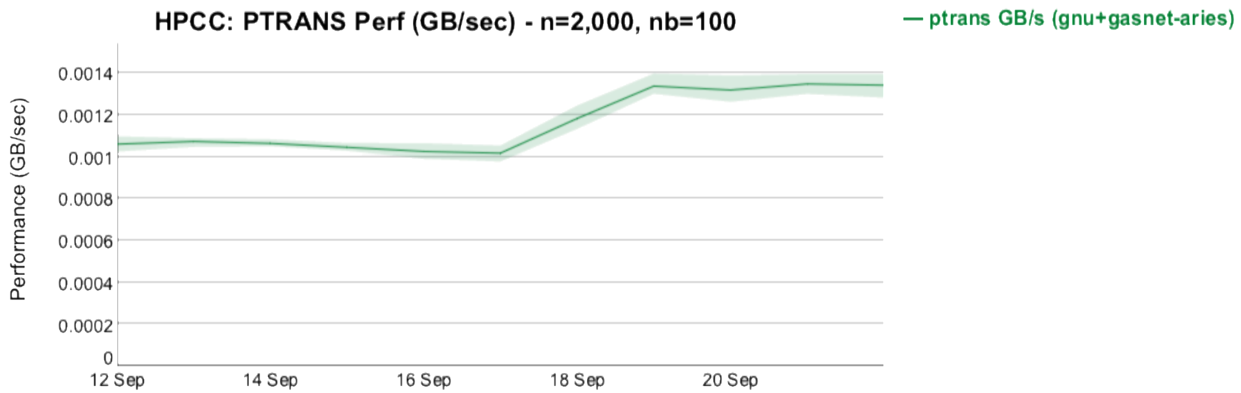
Now generates

```
for (i = INT64(1); i <= INT64(10); i += INT64(1))
  *(arr_base + i) = i;  // identical to arr_base[i] = i;
```

Array Indexing Opt: Impact

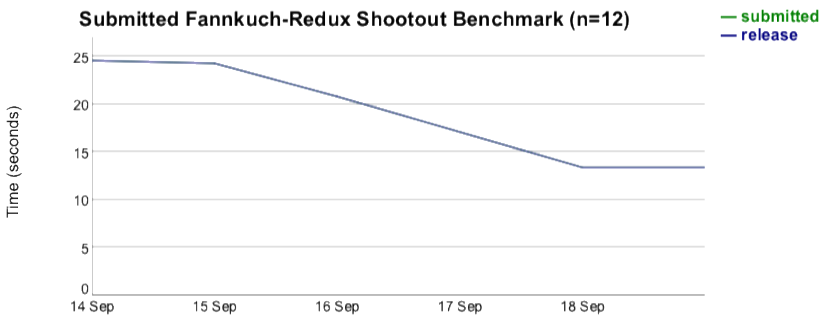
- **Saw significant performance improvements**
  - particularly for array-heavy benchmarks (higher is better)

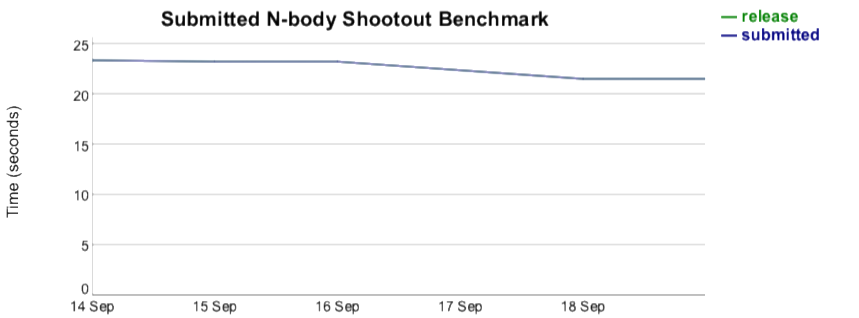


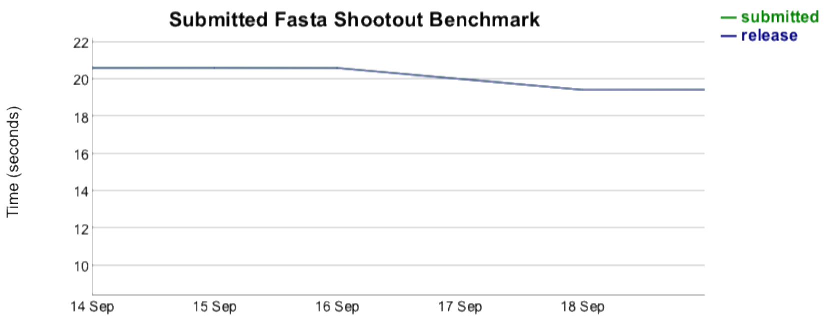


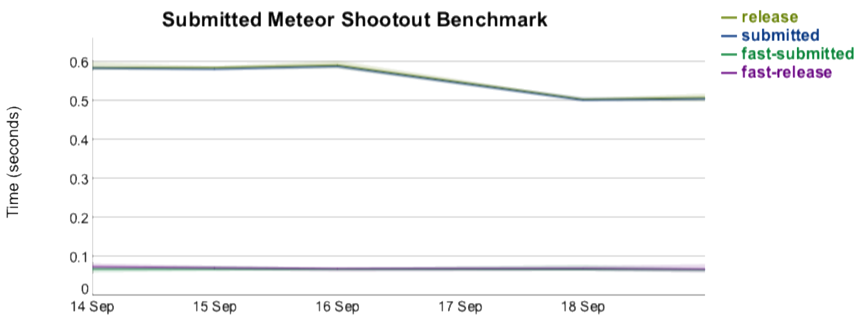
Array Indexing Opt: Impact

- **Saw significant performance improvements**
  - several shootout benchmarks also benefited (lower is better)









Array Indexing Opt: Status and Next Steps

Status:

- added an array indexing optimization
  - indexing into 1-D arrays is as efficient as C/C++ arrays
  - optimization is thwarted if any arrays require this multiplication

Next Steps:

- finish ”array-views” work
  - retire current compiler optimization

Promoted Fast Followers Improvements

Fast Promotion: Background

- **Chapel supports promoted expressions**

```
var A, B, C: [1..m] real;
A = B + alpha * C;
```

- **Promotion is implemented using zippered iteration**

  The promoted expression:

```
A = B + alpha * C;
```

  Is semantically equivalent to:

```
forall (a, b, c) in zip(A, B, C) do
  a = b + alpha * c;
```

- **Historically, promotion could hurt performance**
  - compiler did not build support for promoted fast-followers
    - fast followers: optimize zippered iteration for aligned distributed arrays
    - hurt performance for aligned promotion relative to explicit zippering

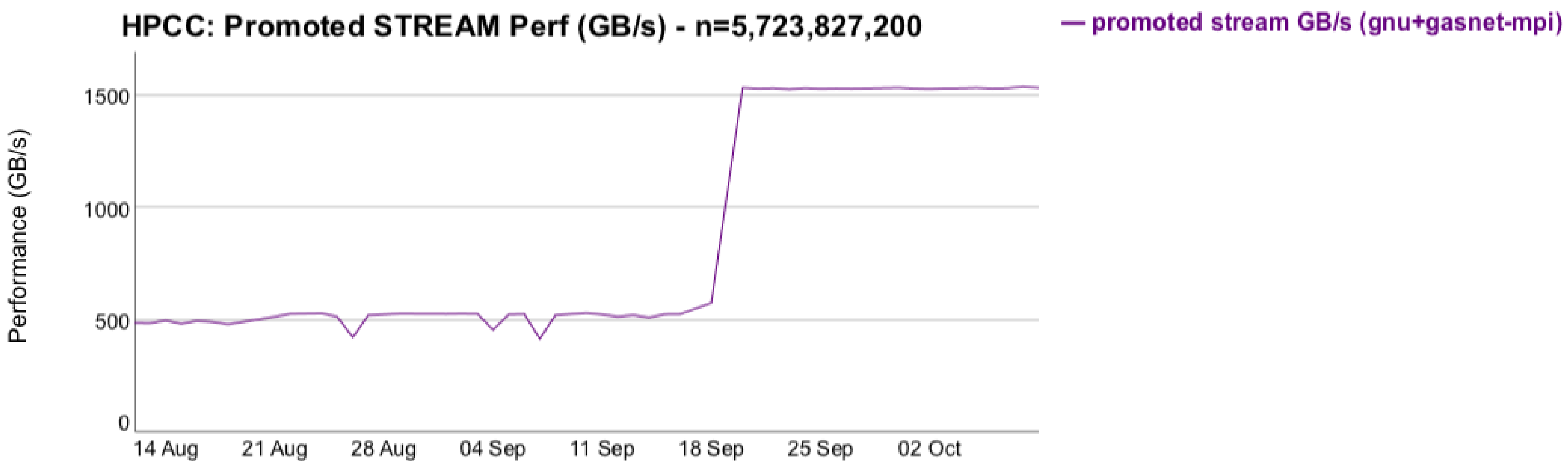
Fast Promotion: This Effort and Impact

This Effort: add support for promoted fast followers

- compiler builds checks required for fast-followers to trigger
  - at compile time: check that the promoted types support fast-followers
  - at runtime: check that promoted arrays are aligned (distributed identically)

Impact: improved performance of promoted expressions

- no longer any penalty for using promotion



Fast Promotion: Next Steps

Next Steps: eliminate runtime checks when possible

- arrays declared over the same distribution must be aligned

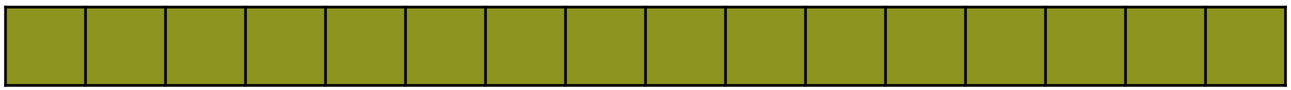
```
var A, B: [distDom] real; // alignment known at compile time

var A: [distDom1] real;
var B: [distDom2] real;     // alignment must be checked at runtime
```

Strided Bulk Transfer

Strided Bulk Transfer: Background

- **Whole-array assignment can use a single GET or PUT**

```
var A, B : [1..4, 1..4] int;
A[1..4, 1..4] = B[1..4, 1..4];
```



- **Slicing may select non-contiguous memory**

```
A[1..4, 1..2] = B[1..4, 3..4];
```

- **Can still bulk-transfer elements contiguous in memory**

Strided Bulk Transfer: Background

- **Initial implementation based on GASNet support**
  - Based on approach described by Dan Bonachea
    - http://upc.lbl.gov/publications/upc_memcpy.pdf
  - Contributed by Rafael Asenjo and Alberto Sanz (U. Malaga) for v1.6

- **Calls out to runtime functions to perform transfers**
  - Uses GASNet's interface when possible
  - Otherwise uses our own implementation for each comm layer
  - Module code computes necessary metadata about arrays

- **Enabled through 'useBulkTransferStride' config param**
  - Disabled by default due to lack of confidence in testing

Strided Bulk Transfer : This Effort

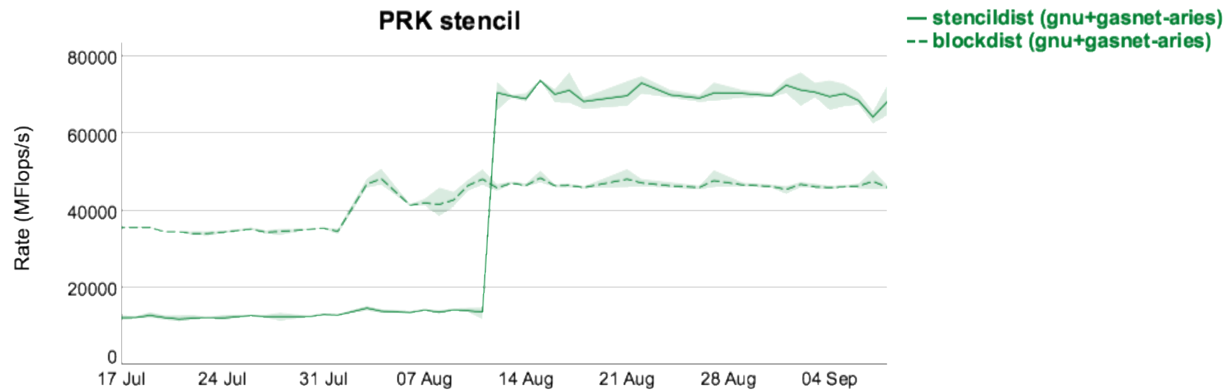
- **Significantly improved the implementation**
  - Fixed several bugs
  - Simplified code implementation
  - Revamped documentation

- **Improved testing for DefaultRectangular cases**
  - Rank changes
  - Strided domains
  - Many combinations of domains up to four dimensions

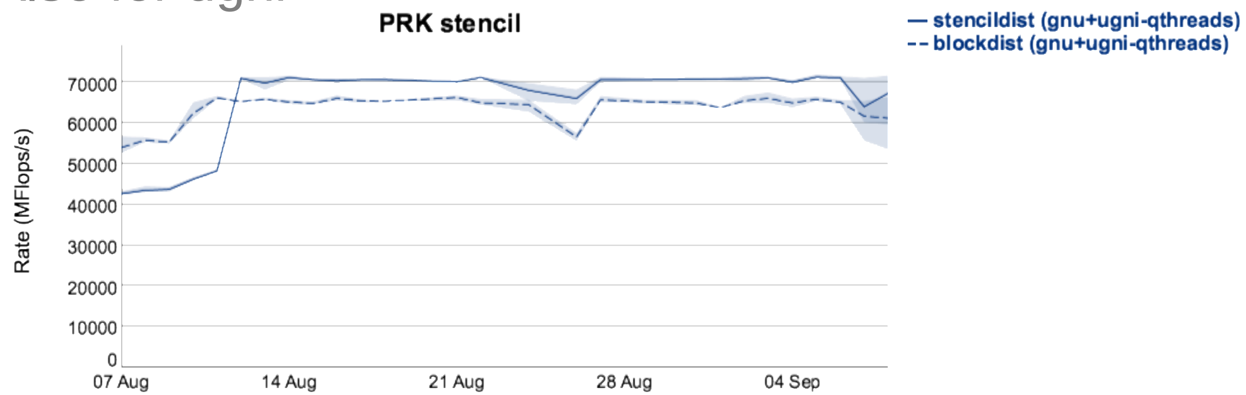
- **Enabled this optimization by default**
  - Good performance observed for Intel PRK Stencil app

Strided Bulk Transfer : Impact

- **Good improvements for PRK stencil**
  - Especially for GASNet



  - Also for ugni



Strided Bulk Transfer : Status and Next Steps

Status:

- Enabled by default for the 1.14 release

Next Steps:

- Investigate distributed array strided bulk transfer
- Module-level implementation for runtimes without custom support?

Array-as-vec Shrinking Improvement

Array-as-vec Shrinking Improvement

Background: Shrinking an array-as-vec left no room for growth

- After a shrink, the allocated size was set to the current array size
- Then push/popping a few elements could cause repeated reallocation

This Effort: Leave room for growth after shrinking the array

- Leave the allocation `growthFactor` times bigger than the number of array elements

Impact: Push/popping a few elements won't cause repeated resizing

Locality Optimizations

Wide Pointer Analysis Improvements

Wide Pointer Analysis: Background

- **Wide pointers represent remote data**

```
typedef struct {
  locale_id_t node;
  myClass*    addr;
} chpl__wide_myClass;
```

- **They introduce overhead when data is actually local**
  - Especially for array accesses
  - Runtime check required to see if data is local
  - Wide pointers may thwart back-end C compiler optimizations

Wide Pointer Analysis: Background

- Passing a wide pointer to a function has consequences

```
proc increment(this: myClass) {
  this._internalAdd(1);
  return this;
}
var foo = new myClass();
foo.increment(); // internally becomes increment(foo)

on Locales[numLocales-1] {
  // 'foo' is remote in this scope, so 'this' formal must be wide
  foo.increment();
}
```

- 'increment' will always return a wide pointer, even for local data

```
// types for 'increment' must change during compilation
proc increment(this: chpl__wide_myClass) {
  this._internalAdd(1);
  return this; // now a wide pointer!
}
```

Wide Pointer Analysis: Background

- **'increment' example is artificial, but this occurs in practice**
  - Array, domain, distribution constructors
  - Array slicing

- **For non-trivial programs, we eventually use a wide array**
  - Especially when domain maps are used

- **At callsite, the returned pointer refers to local data**

```
var foo = new MyClass();
foo.increment(); // 'foo' is local, but increment returns wide
```

  - Problem: the compiler could not detect this in 1.13

Wide Pointer Analysis: This Effort

- **Develop analysis to detect that the returned data is local**
  - If we pass in a local class, get a local class back
  - Passing in a wide class, gets a wide class back

- **Find functions that 'reflect' the wide-ness of arguments**
  - Reflects if returned symbol would be local if arguments are local

- **At a callsite, localize the returned wide pointer**
  - When the arguments are also local

Wide Pointer Analysis: This Effort

- **Consider the 'increment' function**
  - Where 'foo' is a **local** variable
    - Before this effort:

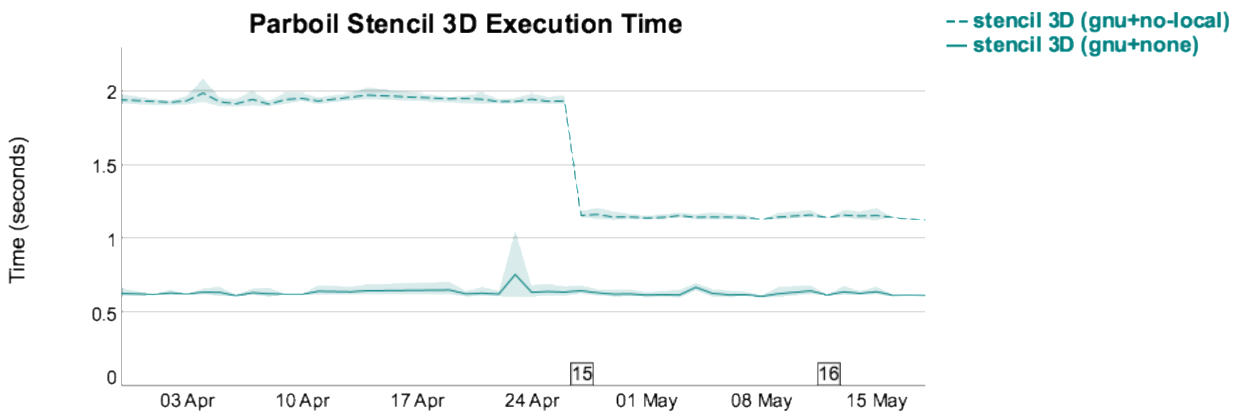
```
var temp : wide__myClass = increment(foo);
```

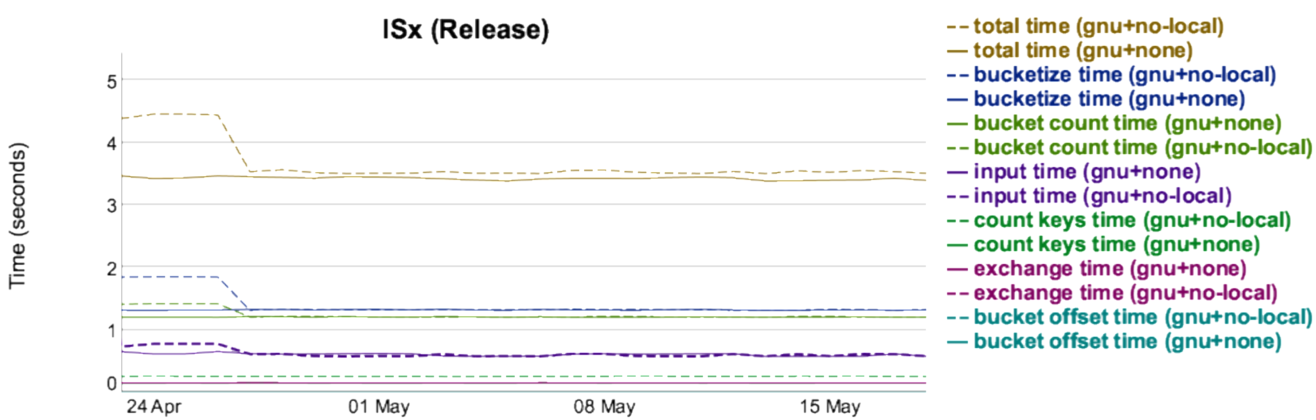
    - After analysis:

```
var wideTemp : wide__myClass = increment(foo);
var temp     : myClass       = wideTemp.addr; // The local pointer

// 'temp' is now local and avoids wide-pointer overhead for subsequent operations
```

Wide Pointer Analysis: Impact

- **Improvements for single-node –no-local**
  - Minimal impact, if any, on true multi-locale





Wide Pointer Analysis: Status and Next Steps

Status:

- New analysis enabled for 1.14 release
- Improves generated C code
- Performance gains less than hoped for

Next Steps:

- Improve analysis for return-by-reference formals
- Look for other patterns where this analysis is useful

Reducing Task Counting Overhead

Task Counting: Background

- **coforall statements wait for their tasks to complete**
  - implemented with atomic variables
  - when coforall spawns each task, the atomic variable is incremented
  - when a task completes, it decrements the atomic variable

- **But, decrementing created a new task!**
  - on the locale that owns the atomic variable
  - this is unnecessary overhead

Task Counting: Background

```
coforall loc in Locales {
  on loc {
    foo();
  }
}
```

// is converted by the compiler into something like this:

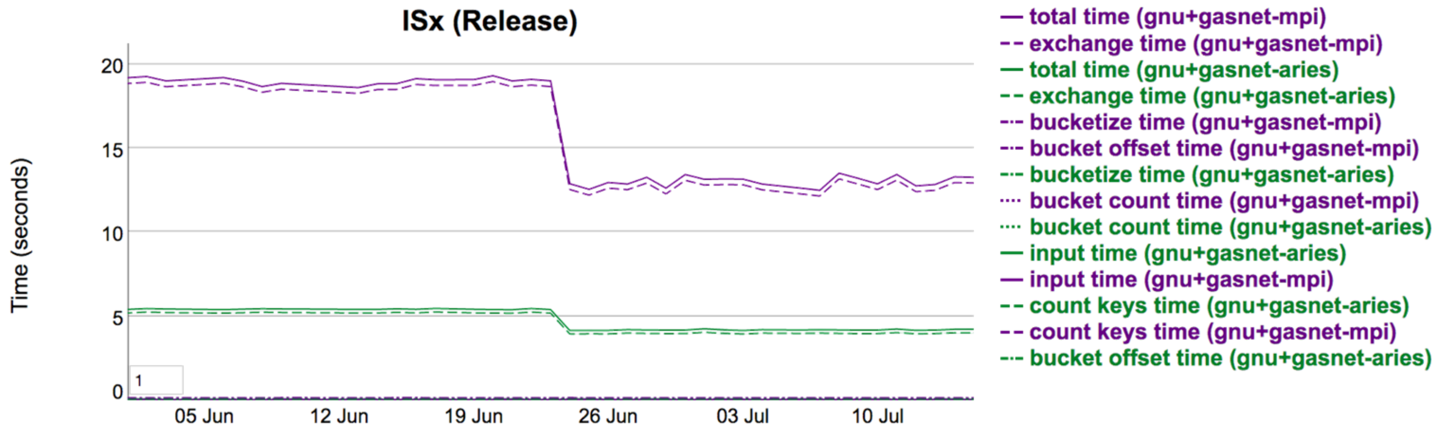
```
var tasksRunning: atomicInt; // processor atomic
for loc in Locales {
  tasksRunning.add(1);
  spawn_task_to_loc(loc, foo_wrapper());
}
tasksRunning.waitFor(0);

foo_wrapper() {
  foo();
  on Locales[0] do tasksRunning.sub(1);
}
```

Overhead: creates a task on Locale 0

Task Counting: This Effort

This Effort: Make compiler smarter about these decrements
- perform them in active message handler
- no need to start a task

Status:
- Implemented and in the release
- Improved performance of some tests, e.g. for this 16-node XC run:



Next Steps:
- Make additional short operations run in active message handlers

Reducing Overhead for Local On Statements

Local On: Background

- **Programs often have on-statements that run locally**
  - these are commonly there for generality
- **Some common examples:**
  - I/O from Locale 0
  - updating atomic values
- **These on-statements still add overhead because:**
  - an argument bundle is allocated
  - arguments are stored into the argument bundle
  - the runtime is invoked to possibly communicate

Local On: This Effort

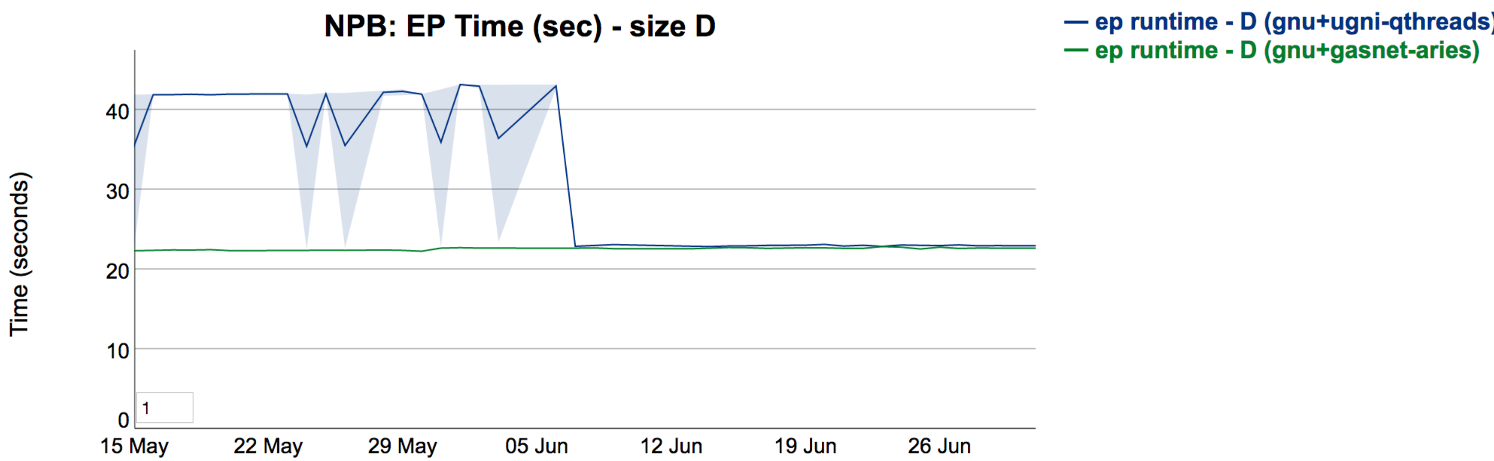
- **Compiler now generates a fast-path for on-statements**

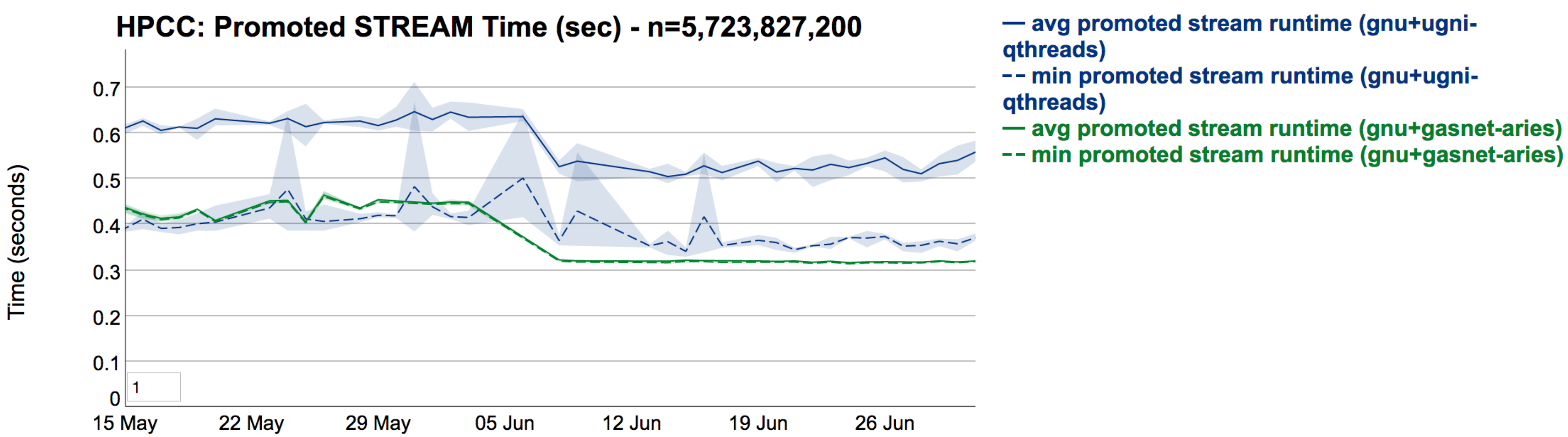
```
on targetLocale do f(a, b, c);
```

now translates into

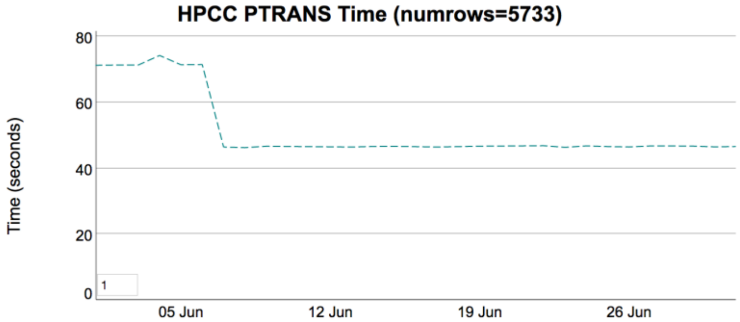
```
if (targetLocale == thisLocale) {   // local case
  f(a, b, c);
} else {                            // remote case
  arguments = malloc(...);
  arguments->a = a;
  arguments->b = b;
  arguments->c = c;
  chpl_executeOn( targetLocale, &f_wrapper );
  free(arguments);
}
```

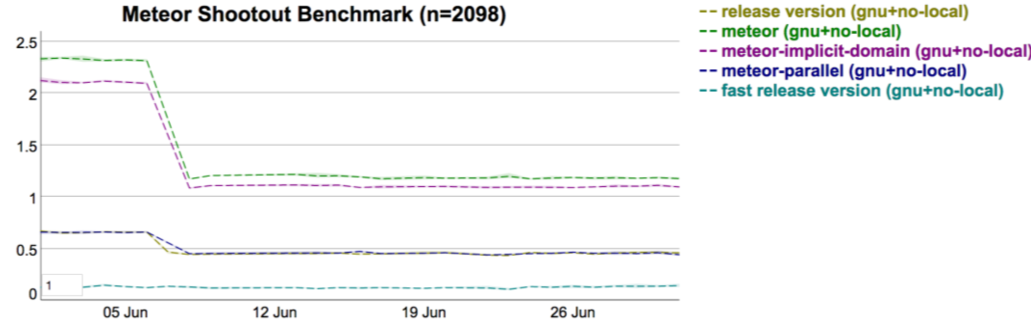
Local On: Impact on 16-node XC

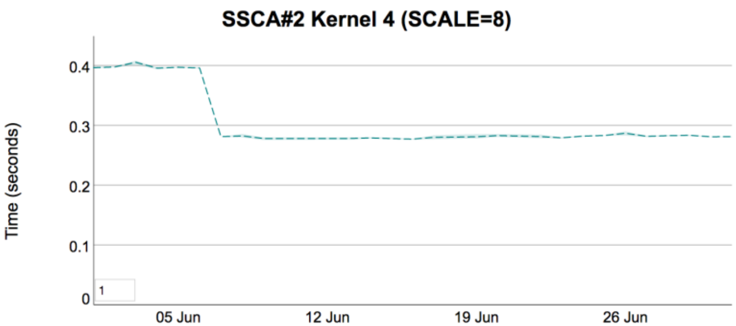


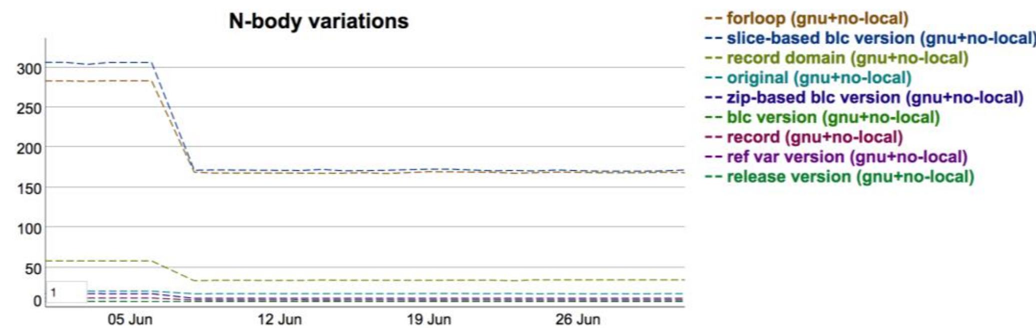


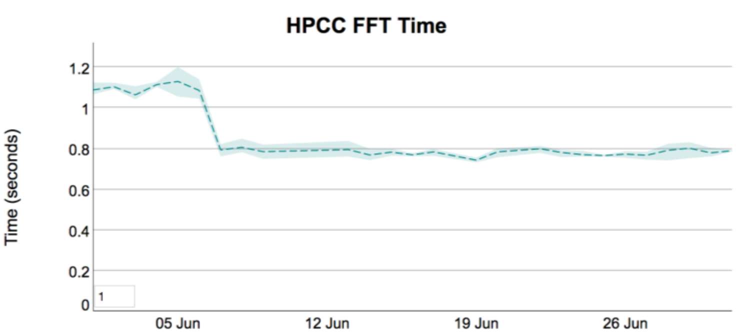
Local On: Impact on --no-local tests

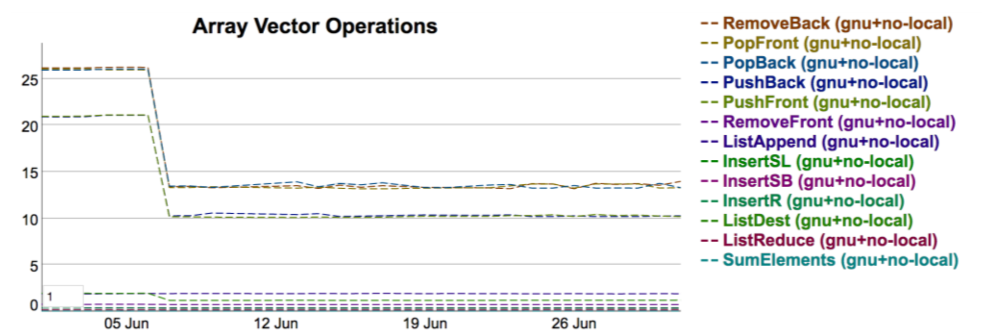












Local On: Status and Next Steps

Status:

- Optimization implemented
- Good improvement to --no-local compilation
- Some improvement in 16-node XC performance testing
- Reduces the number of cache flush events with --cache-remote

Next Steps:

- Apply the optimization to non-blocking on statements
- Fold away the remote case within local blocks

Qthreads Improvements

Native Qthread Sync Vars

Qthread Sync Vars: Background

Chapel Sync Var History:

- historically, Chapel permitted any type to be declared 'sync'/'single'
  - this was thought to be attractively general and orthogonal — for example:

```
var A$: [1..n] sync int;      // an array of synchronized integers
var B$: sync [1..n] int;      // a synchronized array of integers
```

  - synchronized arrays could be interpreted sensibly in some cases:

```
B$ = 0;                       // block until B$ is empty; zero; leave full
```

  - but others were less clear:

```
B$[3] = 1;                    // how should full/empty state be involved?
```

  - records, complexes had similar issues
- some time ago, decided sync/single should support simple types only
  - effectively, ones with a single logical value (int, bool, uint, real, imag, etc.)
  - compiler started enforcing that decision as of 1.13

Qthread Sync Vars: Background

Chapel Background:

- runtime support for sync/single has traditionally been heavyweight
  - because of historical support for sync/single on arbitrary data types
- a faster/simpler implementation is possible for simple data types

Qthreads Background:

- Qthreads was designed to emulate the Cray XMT architecture
  - has native sync var support because XMT had native sync vars
  - left out operations not typically needed in apps (readXX, writeFF, reset)

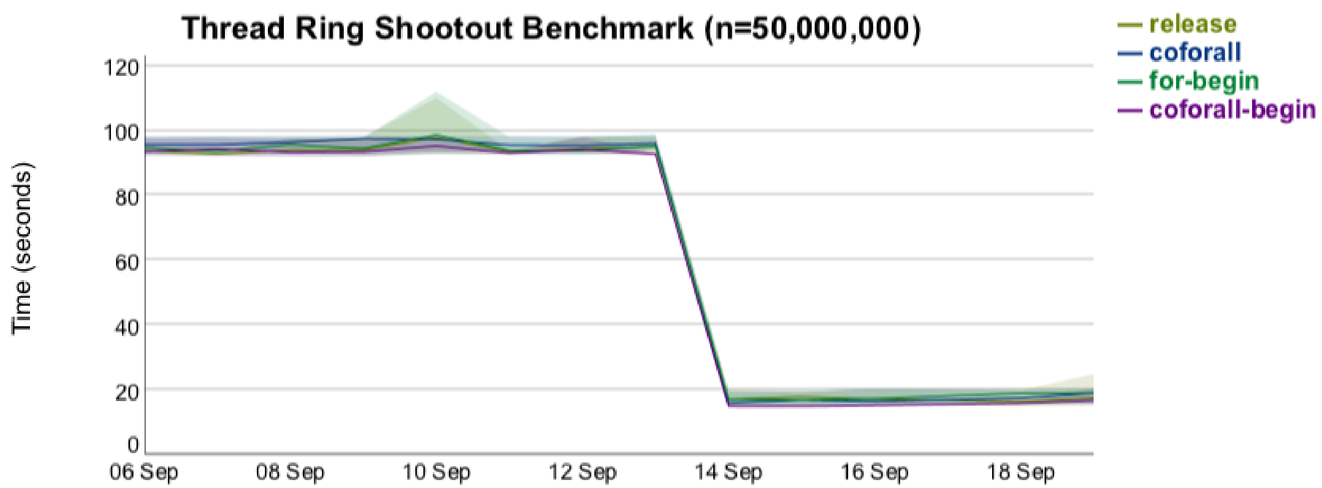
Qthread Sync Vars: This Effort

- **Implemented missing Qthreads sync var operations**
  - with lots of help from one of the original Qthreads developers
  - contributed upstream, included in Qthreads 1.11 release

- **Map Chapel sync vars down to native Qthreads versions**
  - currently implemented for int/uint/bool types
  - other data types don't trivially cast to qthreads sync var type
    - they fall back to the heavy-weight implementation

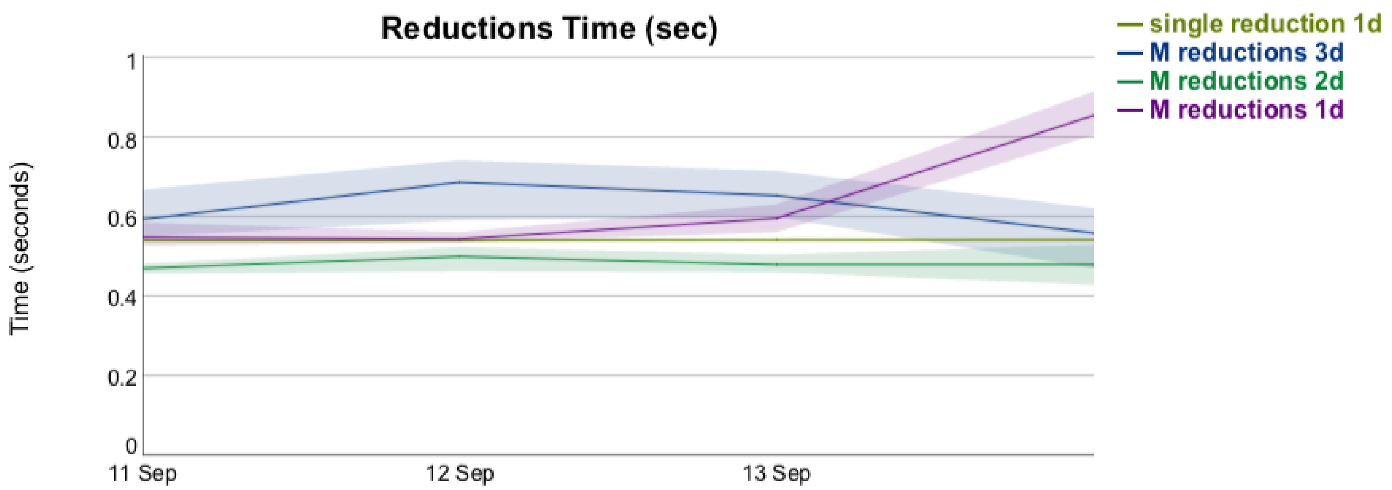
Qthread Sync Vars: Impact

- Dramatically improved perf of highly-contended syncs



Qthread Sync Vars: Impact

- **Hurt reduction performance**
  - regressions were then improved by using an atomic lock (next section)



Qthread Sync Vars: Impact

- **Hurt performance of reductions (cont.)**
  - reductions use a minimally-contended, short-lived lock to accumulate
  - perf loss indicates that qthreads sync var creation time is expensive
  - partially caused by unnecessary hash table manipulation
    - Qthreads uses a hash-table to correlate sync vars to F/E queues/state
    - full vars that have no pending ops are removed from the hash-table
    - this results in unnecessary hash table insertions/removals

```
// reduction accumulation code for each task
lock.writeEF();          // lock – fills (no pending ops): hash entry removed
accumReduction();
lock.readFE();           // unlock – empties: hash entry inserted
```

    - Chapel sync vars are only deleted when they go out of scope
    - want to keep qthreads syncs in hash-table until Chapel destroys them

Qthread Sync Vars: Status and Next Steps

Status:

- added missing sync var operations to qthreads
- mapped Chapel sync vars down to native qthreads sync vars
  - resulted in substantial performance boost for highly-contended sync vars
  - hurt performance for minimally-contended, short-lived sync vars

Next Steps:

- improve performance for short-lived sync vars
  - start by eliminating extra hash table manipulations
- use native qthread sync vars for more Chapel types
  - will require using memcpy in order to "cast" to qthreads sync var type

Reduction Lock Improvements

Reduction Lock: Background and This Effort

Background: reductions have used a sync var as a lock

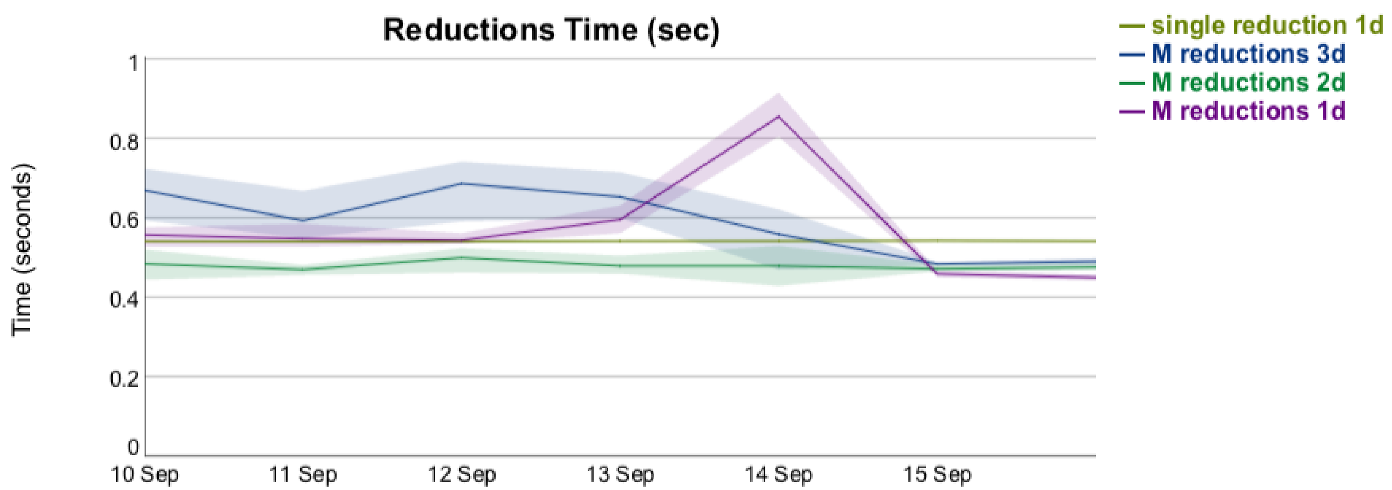
- lock is needed to accumulate parallel reductions
- Implemented as a sync var before atomics were introduced to Chapel
- switching to native qthread sync vars hurt reduction performance
  - qthread sync vars are relatively expensive to create
- accumulation is a short-lived, minimally-contended operation
  - spin-locks are better for this situation

This Effort: use an atomic spin-lock instead of a sync var

- implemented as an exponential backoff testAndSet loop

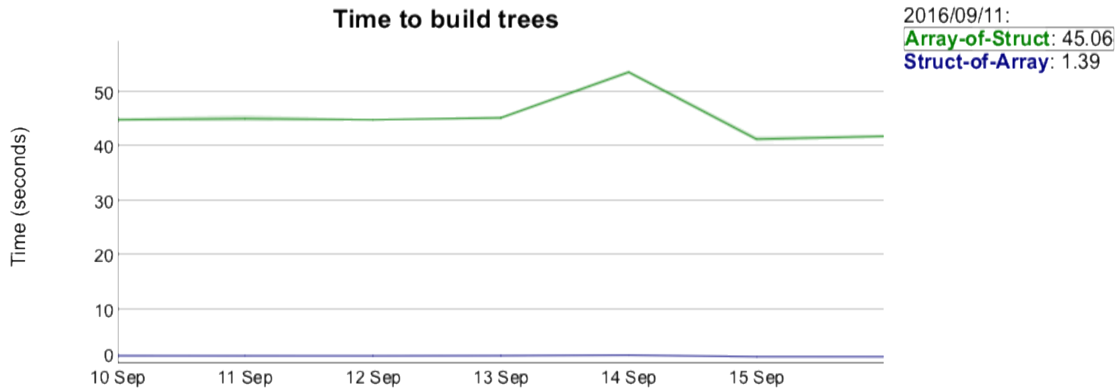
Reduction Lock: Impact and Status

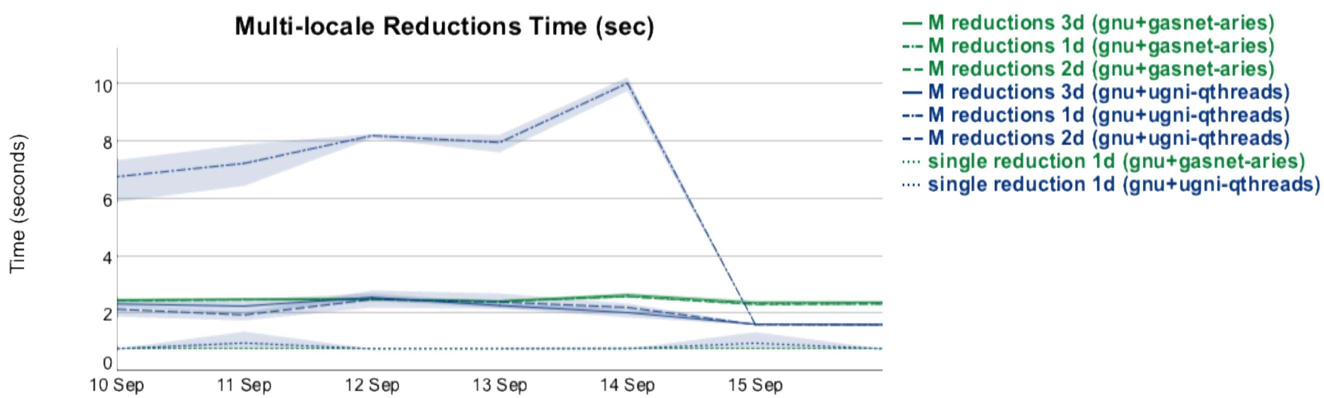
- Improved reduction performance
  - resolved regressions caused by qthread sync vars



Reduction Lock: Impact and Status (cont.)

- Improved reduction performance
  - in many cases, performance is even better than it had been before





New Qthreads “Distrib” Scheduler

Distrib Scheduler: Scheduler Background

- **Qthreads has several scheduler options**
- **“nemesis” is our default**
  - simple, but fast
  - no work-stealing
    - new tasks distributed to queues round-robin, never re-balanced
  - no numa-awareness
- **“sherwood” was our default for the NUMA locale model**
  - numa-aware (required by NUMA locale model)
  - support for work-stealing
    - new tasks placed in a single queue, work-stealing required to balance
  - too slow to be our default everywhere

Distrib Scheduler: Sherwood Background

- **Sherwood was tuned for Unbalanced Tree Search (UTS)**
  - UTS presents significant load imbalance
    - fast solutions require dynamic load balancing using many tasks

- **Sherwood has very aggressive work-stealing**
  - ideal for UTS, horrible for balanced workloads
    - idle threads continuously try to steal, even when no work is available
  - disabling work-stealing cripples the scheduler
    - work distribution requires stealing, new tasks are added to a single queue

- **Overall, sherwood had poor performance**
  - especially for applications with balanced workloads

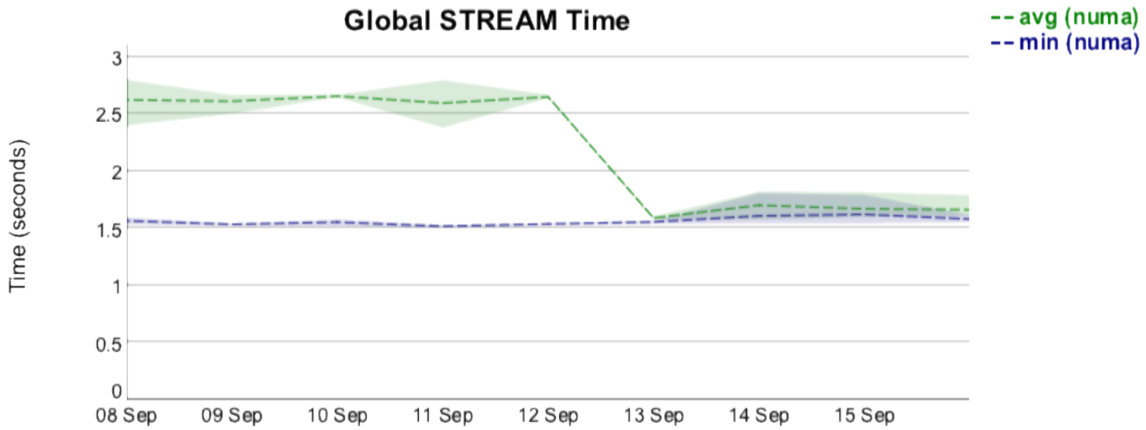
Distrib Scheduler: This Effort

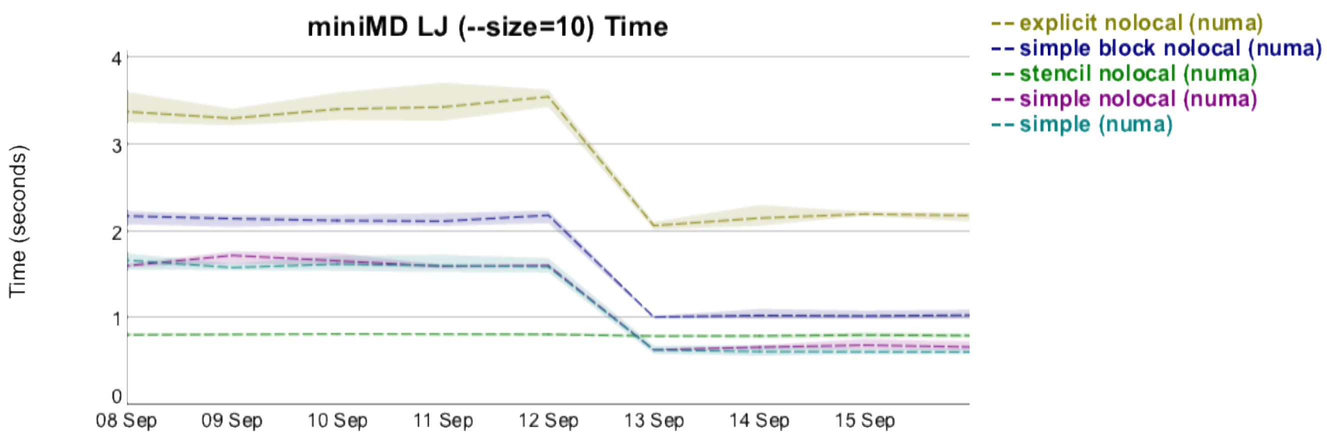
- **Qthreads team developed a new scheduler**
  - Qthreads team initially tried to improve sherwood scheduler
    - proved too difficult, so a new distrib scheduler was created
  - we worked closely with the Qthreads team to tune performance
    - iterated through many contention management and work-stealing strategies
  - as a result, distrib is a huge improvement over sherwood
    - significantly faster than sherwood, competitive with nemesis
    - still has support for work-stealing
    - still numa-aware

- **Made distrib our default scheduler for numa**
  - currently disable work-stealing by default
    - needed more time to tune performance

Distrib Scheduler: Impact

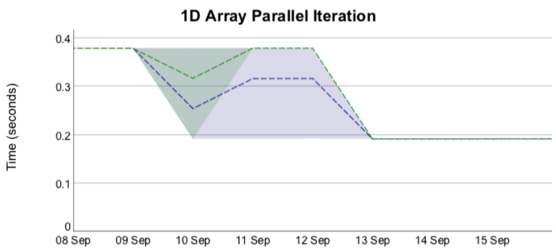
- Significant performance improvements for numa

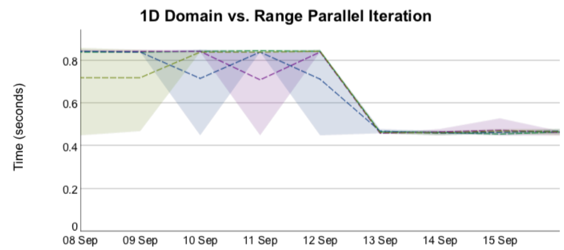


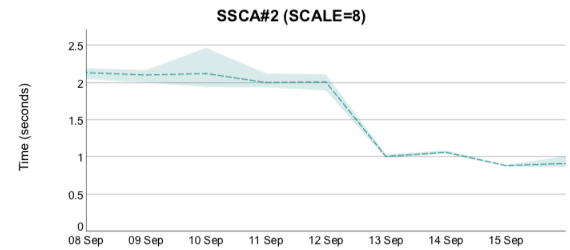


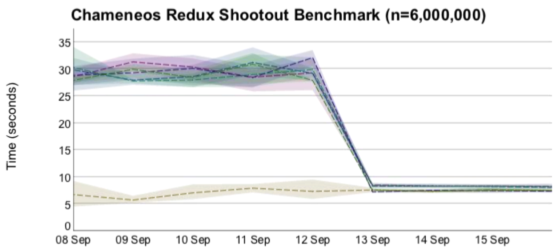
Distrib Scheduler: Impact

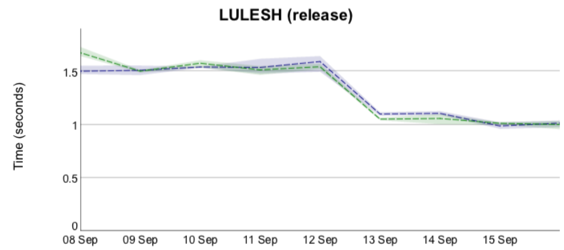
- **Significant performance improvements for numa**
  - … lots of them

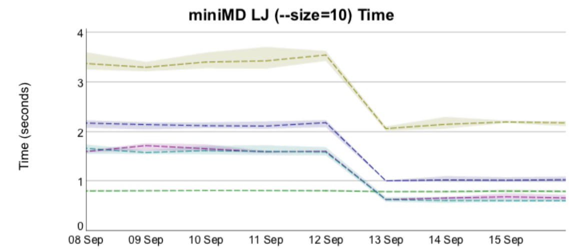


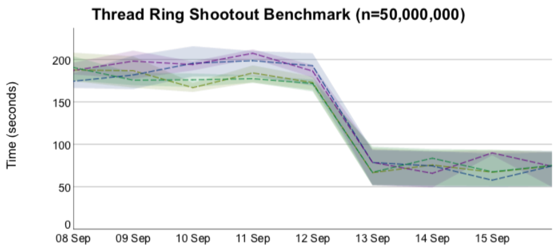


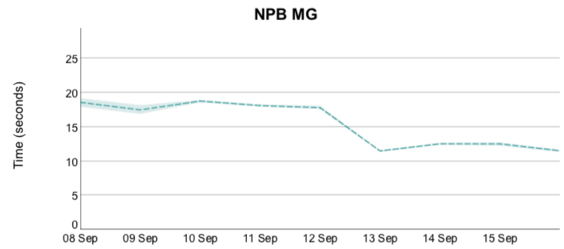


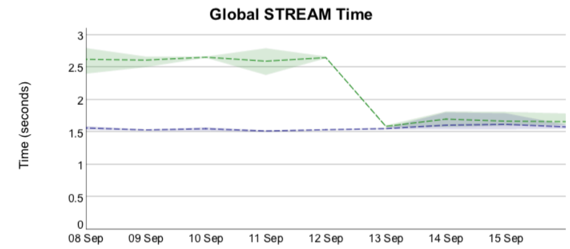












Distrib Scheduler: Status and Next Steps

Status:

- new distrib scheduler is being used for numa
  - results in significant performance improvements

Next Steps:

- continue to tune distrib scheduler
  - close remaining gap with nemesis, and default to distrib everywhere
- tune work-stealing algorithm
  - make the overhead small enough that we can enable it by default

Runtime Improvements

Jemalloc Changes

Jemalloc: Overview

- **Jemalloc is a general-purpose malloc implementation**
  - "scalable concurrency support"
  - "emphasizes fragmentation avoidance"
  - also supports an extended API
    - good alloc size, sized deallocation, etc.

- **Actively maintained on GitHub**
  - https://github.com/jemalloc/jemalloc

- **Large number of notable users**
  - FreeBSD and NetBSD
  - Mozilla Firefox
  - Facebook
  - Rust
  - Chapel!

Jemalloc: Portability Improvements

Background: made jemalloc our default memory layer in 1.13

- resulted in significant performance improvements
- however, we couldn't use jemalloc in a few configurations:
  - cce (build issues, but we worked around them in our makefiles)
  - osx+gnu (build issues)
  - pgi (segfaults at execution time)

This Effort: Improved portability for cce, osx+gnu, and pgi

- patches accepted upstream, will be in the next jemalloc release
  - manually applied to our copy of jemalloc in the meantime

Impact: jemalloc is now our default everywhere

- (except under cygwin, but cygwin performance is not a priority)

Jemalloc: Performance Improvements

Background: saw significant performance benefits with jemalloc

- jemalloc is frequently released, often with performance improvements
- we initially used the default configuration options

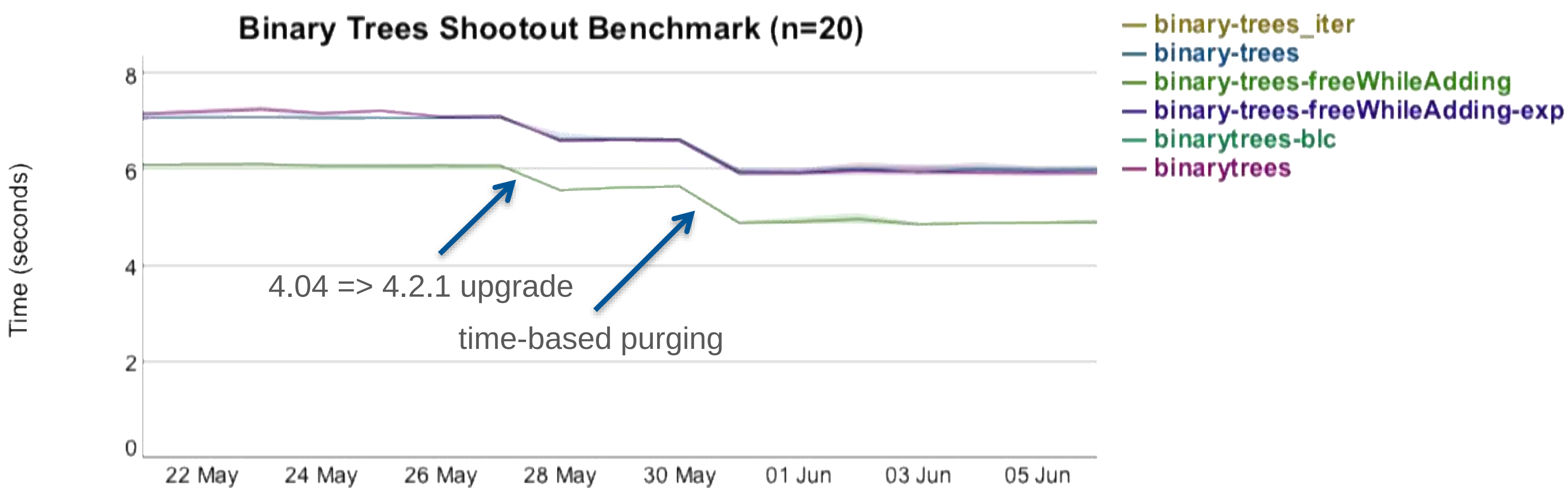
This Effort: upgraded jemalloc and streamlined configuration

- upgraded from jemalloc 4.0.4 to 4.2.1
- enabled a new time-based purging optimization
  - improves mechanism for returning memory to the OS
  - opt-in for 4.X, will likely be the default for 5.X
- disabled stats gathering by default
  - stats gathering is highly optimized, but still has a non-zero cost
  - to enable, set CHPL_JEMALLOC_ENABLE_STATS at build-time

Jemalloc: Performance Improvements

Impact: additional performance improvements

- saw performance improvements for several benchmarks
  - most notably for binary trees, which had > 20% speedup



Jemalloc: Summary and Next Steps

Summary:

- improved jemalloc's portability
  - for cce, pgi, and osx+gnu
- improved jemalloc's performance
  - through version upgrade and configuration optimization

Next Steps:

- use more of the extended API
  - add support for sized deallocation
  - use good_alloc_size() in more places (e.g., array-as-vector)
- use jemalloc for third-party libraries
  - this is already being done for GMP
  - would have the most impact for qthreads, and possibly re2

Faster complex.re and complex.im

Faster complex.re and complex.im

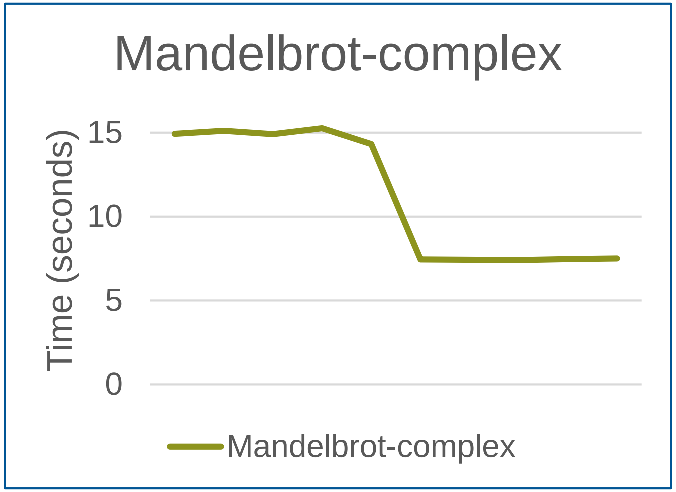
Background: `.re` and `.im` were slower than record field accesses

This Effort: Optimize them to match record access speed

Impact: Codes that use `.re` and `.im` got a performance boost

Mandelbrot-complex inner loop

```
for 1..maxIter {
  if (T.re + T.im > limit) then
    break;
  // Z = Z*Z + C
  Z.im = 2.0*Z.re*Z.im + C.im;
  Z.re = T.re - T.im + C.re;
  T.re = Z.re**2;
  T.im = Z.im**2;
}
```



Other Performance Optimizations

Other Performance Optimizations

- Optimized base**exp when 'base' is a param power of two

```
2**k => 2<<(k-1)
8**k => 8<<(3*(k-1))
```

- Eliminated compiler-created tuple for zippered serial loops

Legal Disclaimer

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

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

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

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

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

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

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

CRAY
CHAPEL

CRAY
THE SUPERCOMPUTER COMPANY