Performance Optimizations
Generated Code Improvements

Chapel Team, Cray Inc.

Chapel version 1.11

April 2, 2015

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

- Anonymous Range Iteration Optimization
- Vectorization
- Parallel Range Iteration Optimization
- Loop-Invariant Code Motion (LICM) Update
- The "local field" pragma
- The "assertNoSlicing" config param
- External Procedures with 'string' Arguments
- Other Performance Improvements
- Optimization/Codegen Priorities and Next Steps

Anonymous Range Iteration Optimization

Anonymous Range Opt: Background

- **Anonymous ranges: those not stored in a named variable**
  - cannot be referenced elsewhere
  - commonly used directly in a loop

```
for i in 1..10 do
for i in lo..hi do
```

- **Ranges are implemented as records**
  - as a result, each range literal constructs a record object
  - anonymous ranges are not captured and cannot be used again
    - so why waste time constructing them?

Anonymous Range Opt: This Effort

- **Eliminate construction for common anonymous ranges**
  - provide an optimized iterator when stride is known at compile time
  - eliminate cost of construction
  - permits generating loop termination with <= or >= rather than !=
    - (OpenMP/OpenACC pragmas can't be attached to loops terminated by !=)
  - allow back-end compiler to better optimize and auto-vectorize

- **Optimization occurs at parse time**
  - for-loop builder recognizes certain range patterns
  - replaces those with a direct range iterator
    - iterator takes low, high, stride as arguments
    - e.g., compiler replaces:

```
for i in 1..10 do
```

with:

```
for i in chpl_direct_range_iter(1, 10, 1) do
```

Anonymous Range Opt: Impact

- **Eliminates range construction for many common cases**

```
for i in 1..10 do writeln(i);
```

previously:

```
_build_range(INT64(1), INT64(10), &call_tmp);
low = (&call_tmp)->_low;
high = (&call_tmp)->_high;
for (i = low; i <= high; i += INT64(1))
  writeln(i);
```

now:

```
for (i = INT64(1); i <= INT64(10); i += INT64(1))
  writeln(i);
```

Anonymous Range Opt: Impact (continued)

- **Optimized iteration for strides known at compile time**

```
for i in 1..10 by 2 do writeln(i);
```

previously:

// function call to build range
// function call to apply 'by' operator to range
// function call and conditional check to see if range is ambiguous
// function call to compute the starting value
// conditional check to see if range is empty (e.g. 2..1)
// function call to compute the ending value

71 SLOC

```
for (i = start; i != end; i += str)  // finally iterate, but using !=
  writeln(i);
```

now:

```
for (i = INT64(1); i <= INT64(10); i += INT64(2))
  writeln(i);
```

Anonymous Range Opt: Impact (continued)

- **Stridable anonymous ranges amenable to offload pragmas**
  - when stride is known at compile time

- **Better back-end optimization and auto-vectorization**
  - range construction and other checks obfuscate iteration pattern
  - we now propagate range literals directly to the C for loop
    - helps create cleaner vectorized code (eliminates some loop peeling)
    - allows compiler to better select unrolling factor and trip count

- **No major changes seen in nightly performance graphs**
  - not terribly surprising
    - most time spent in loop body, not prelude
    - not many benchmarks iterate over nested anonymous ranges
    - lacked performance testing with modern vectorizing back-end compilers
      - have since started testing with the newest versions of Cray, GNU, Intel, and PGI

Anonymous Range Opt: Status

- Cases that are currently handled

```
for i in 1..10 do                               // works for simple ranges
for i in 1..10+1 do                             // works with expressions in ranges
var lo=1, hi=10; for i in lo..hi do             // works for variables
for i in 1..10 by 2 do                          // works for strided ranges
for (i, j) in zip(1..10, 1..10) do              // works for zippered iters
for (i, j) in zip(A, 1..10) do                  // following non-ranges also works
coforall i in 1..10 by 2 do                     // works for coforalls as well
```

- Cases that are not handled

```
for i in (1..) do                   // doesn't handle unbounded ranges
for i in 1..10 by 2 by 2 do         // doesn't handle more than 1 'by' operator
for i in 1..10 align 2 do           // doesn't handle 'align' operator
for i in 1..#10 do                  // doesn't handle 'count' operator
var r = 1..10; for i in r do        // not an anonymous range
forall i in 1..10 do                // does not get applied to foralls
```

Anonymous Range Opt: Next Steps

- **Handle additional cases**

```
for i in 1..#10 // used frequently in leader and standalone iterators
```

- **Move optimization from parse-time to after resolution**
  - requires that resolution is moved before normalization
  - would allow us to handle more cases
    - …and not be so careful about preserving user errors
  - would allow us to anonymize named ranges used only for iteration

```
var r = 1..10;
if debugParam then writeln(r);      // common in our iterators
for i in r do yield i;

var r = 1..10;
for i in r do A[i] = i;
for i in r do A[i] = A[i%10+1]; // common in benchmarks & user code
```

Vectorization

Vectorization: Background

- **Vectorization is crucial for achieving peak performance**
  - true for commodity and HPC systems
  - becoming increasingly important, particularly in HPC
    - AVX-512 (Xeon and Xeon Phi)
    - NEON (ARM)

- **Chapel relies on back-end compiler to auto-vectorize**
  - Chapel's primary back-end generates C code
  - C compilers are frequently thwarted by memory aliasing
    - must make conservative assumptions that inhibit auto-vectorization

Vectorization: Background (continued)

- **Chapel is well-suited for vectorization**
  - limited aliasing
  - support for array programing

    ```
    A = B + C;
    ```
  - parallelism is a first class citizen

    ```
    forall i in 1..10 do …
    ```

- **Need to convey Chapel semantics to back-end**
  - do not want to generate explicit vectorization
    - rather, convey when vectorization is legal
    - leverage back-end compilers' sophisticated and refined cost models

Vectorization: Background (continued)

- **Data-parallel operations are vectorizable**
  - user asserts there are no data dependencies or ordering constraints

```
A = B + C;
forall i in 1..n do A[i] = B[i] + C[i];
forall (a, b, c) in zip(A, B, C) do a = b + c;
```

- **Data-parallelism implemented in terms of task-parallelism**
  - for zippered case…
    - leader iterators create parallelism and assign work to followers
    - follower iterators serially do the chunk of work assigned by the leader
      - work assigned to followers should have no vector dependencies
  - in standalone case, iterator creates parallelism *and* does serial work
    - again, serial work should have no vector dependencies
    - here, we'll call this serial work "follower loops" for simplicity

Vectorization: This Effort

- **Mark follower loops with ‘#pragma ivdep’ in C code**
  - ‘ivdep’ tells the back-end compiler to ignore vector dependencies
    - each compiler has slightly different semantics for the pragma

- **‘ivdep’ permits back-end to ignore assumed dependencies**
  - iteration dependence, memory aliasing, etc.
  - back-end may unconditionally vectorize loops with potential aliases
    - instead of two loops with a runtime check to see if the vector version is safe
  - back-end can vectorize loops that it assumed were illegal before

Vectorization: This Effort (continued)

- **Compiler approach for marking follower loops with ivdep**
  - mark yielding follower loops as order-independent during resolution
    - these are the loops that will execute the body of a forall loop
    - (others may do bookkeeping unrelated to the loop's forall semantics)
  - propagate order-independence during iterator lowering/inlining
    - loops that cannot be inlined are not order-independent
      - advance() function cannot be vectorized
    - a zippered iterator is order-independent iff all iterands are & they are inlined
  - if vectorization is enabled, annotate these order-independent loops
    - generate CHPL_PRAGMA_IVDEP, defined in the runtime for each compiler

- **Added extensive test suite**
  - uses a reporting mechanism to ensure correct loops are annotated
    - and other loops are not mistakenly annotated

Vectorization: Impact

- **Many serial follower loops are annotated**

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

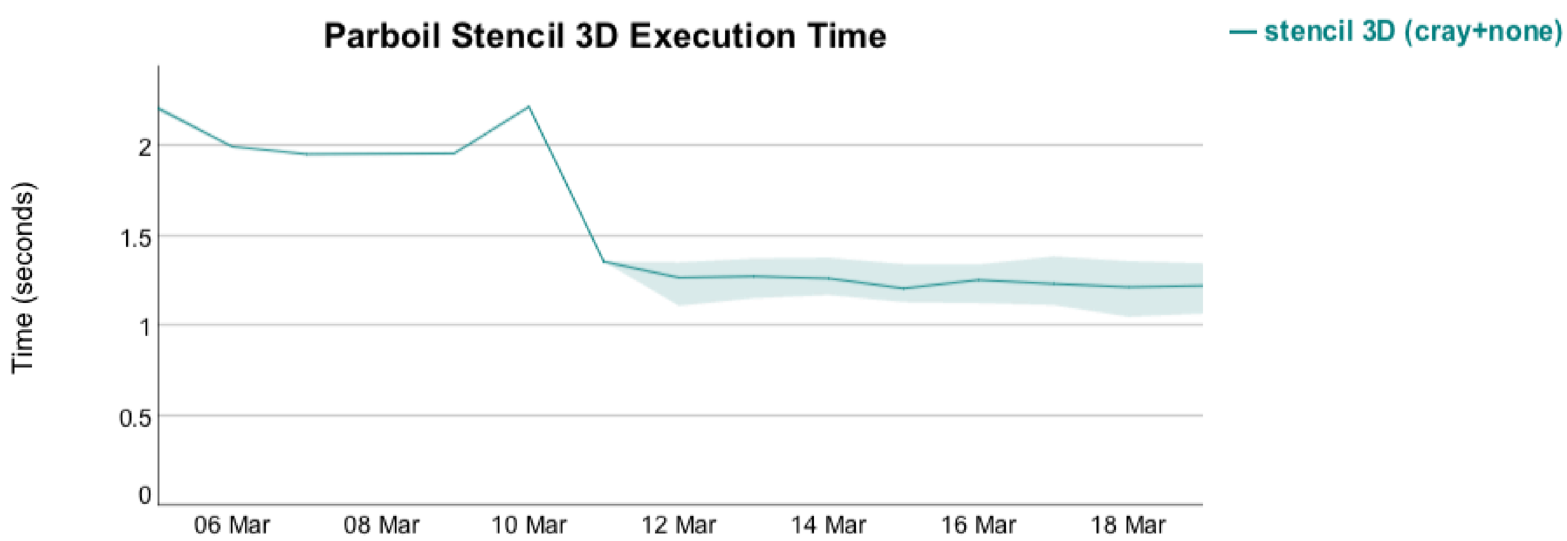
generates:

```
…
CHPL_PRAGMA_IVDEP
for (i = low; i <= high; i += INT64(1)) {
  call_tmp = (shiftedData + i);
  *(call_tmp) = i;
}
```

- **Improves vectorization of loops**
  - determined via back-end vectorization reporting output
    - fewer conditional checks at runtime
    - some previously non-vectorizable loops are now being vectorized

Vectorization: Impact (continued)

- **Performance improvements**
  - 20% performance improvement of stream-ep on Intel KNC
    - runtime checks were more expensive on KNC vs. Xeon
  - improvements for benchmarks with complex array access patterns



Vectorization: Status

- **Vectorization is enabled with the --vectorize flag**
  - automatically enabled with --fast
  - controls whether order-independent loops are marked with ivdep
    - will control more settings in the future (hence generic name)

- **Ran into issues with Cray as the back-end compiler**
  - ‘ivdep’ has slightly different semantics compared to other compilers
    - discovered late in release cycle
    - conservatively stopped annotating with ‘ivdep’ for Cray
    - additional work required to re-enable in appropriate cases

Vectorization: Related Next Steps

- **Add more loop and vectorization benchmarks**
  - Livermore Compiler Analysis Loop Suite (LCALS)
    - (formerly Livermore Loops)
- **Add tests to inspect back-end vectorization reports**
  - to detect which loops are actually being vectorized
- **Start performance testing on Xeon Phi**
- **Explore options with Cray compiler**
  - see what additional analysis we need to attach ‘ivdep’

Vectorization: Additional Next Steps

- **Align memory allocations and generate alignment hints**
  - eliminate loop peeling, cleaner vectorization
- **Mark non-aliasing pointers with ‘restrict’ keyword**
  - perform alias analysis at Chapel level and annotate restricted pointers
    - Chapel has limited aliasing, this helps convey that to the back-end
    - should help with vectorization and other performance optimizations
- **Investigate potential generated code improvements**
  - engage back-end compiler developers for recommendations
- **Explore what we can do with LLVM**
  - we may become constrained by what we can express in C
  - might be able to convey more Chapel semantics to LLVM back-end

Parallel Range Iteration Optimization

Parallel Range Optimization: Background

- **Discovered that parallel iteration over a range was slow**
  - dramatically slower than iterating over a 1-dimensional domain
    - surprising since the domain iterator forwards to a range iterator
    - unfortunate since we tend to advise “ranges are cheaper than 1D domains”
- **Range iterator is more complicated than domain iterator**
  - ranges have to support iteration over an unbounded space
- **Determined that range followers were not being inlined**
  - iterators need to be inlined for optimal performance
    - otherwise an expensive advance() function is called in every iteration
- **Also found that parallel zippered range iteration was slow**
  - zippered iterators have stricter inlining constraints

Parallel Range Optimization: This Effort

- **Update follower iterator so it can be directly inlined**
  - previous "optimization" for single-element ranges prevented inlining
- **Added a special case for non-stridable ranges**
  - non-stridable ranges now utilize a more optimized iterator
    - domain follower already had this optimization
- **Update follower so it can be inlined into zippered iterators**
  - early returns for length == 0 prevented inlining
    - tricky to work around, solution is fast but not elegant

Parallel Range Optimization: Code Impact

- **Range follower iterator can now be inlined in all cases**

  generated follower loop code for:

```
forall i in 1..10 do writeln(i);
```

  previously:

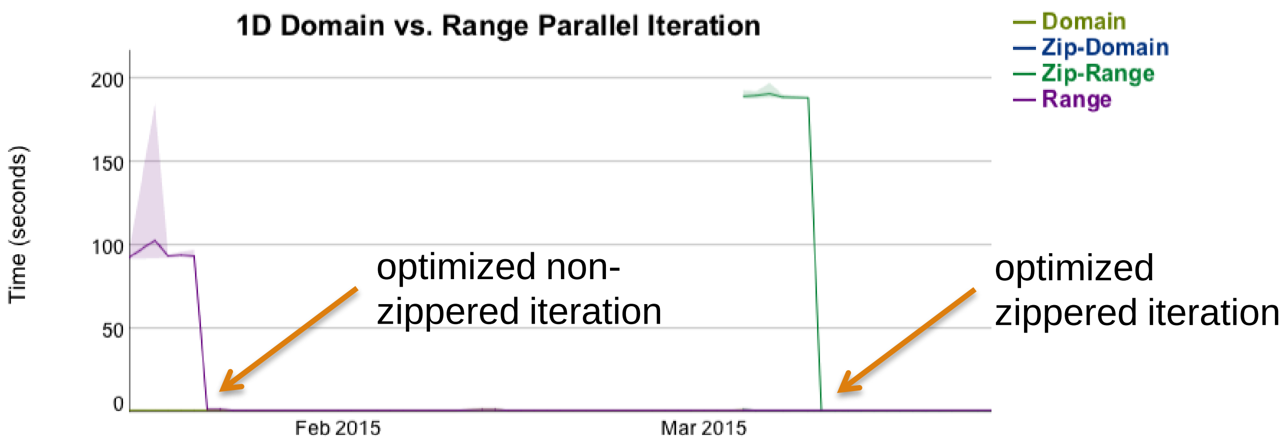
```
...
advance(_ic_);
for (; (T3 = (_ic_)->more,T3); ) {
  T2 = (_ic_)->value;
  writeln(T2);
  advance(_ic_);
}
```

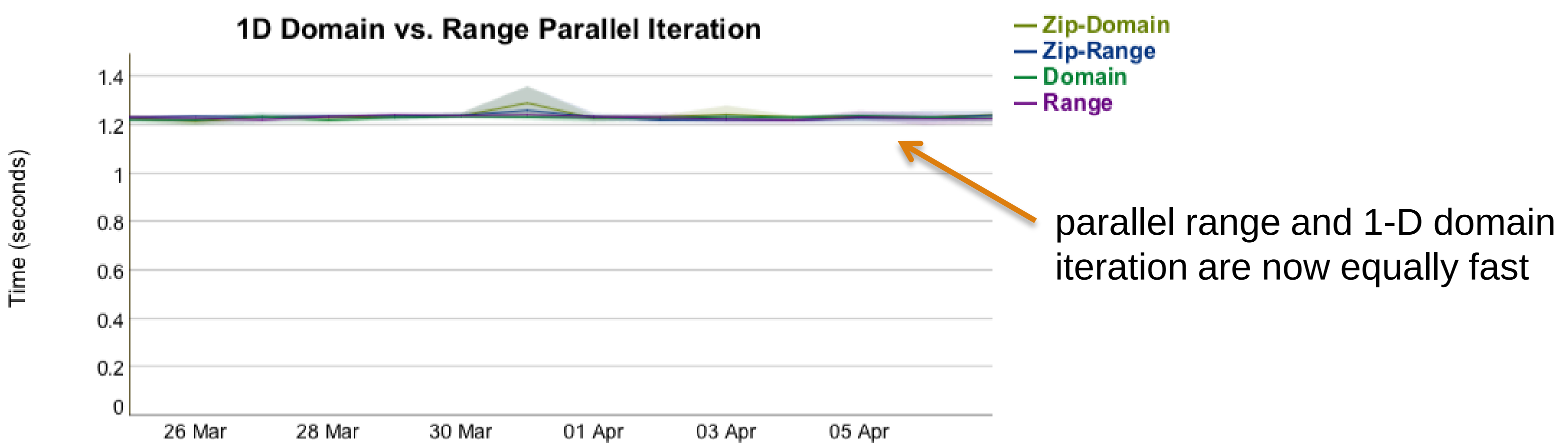
  now:

```
...
for (i = low; i <= end; i += INT64(1))
  writeln(i);
```

Parallel Range Optimization: Perf Impact

- Parallel range iteration is competitive with domains





Parallel Range Optimization: Next Steps

- **Add user-accessible documentation on iterator inlining**
  - guidelines for optimal iterator performance

- **Enhance iterator inlining reporting**
  - current reporting is limited and developer-focused
    - only reports iterators that were successfully inlined
  - want user-friendly reporting with specific reasons if inlining fails
  - could be part of a broader --performance-hints flag

- **Relax zippered iterator inlining constraints**
  - believed to be stricter than they need to be
    - likely part of upcoming “leader/follower 2.0” work

Loop-Invariant Code Motion (LICM) Update

LICM: Background

- **LICM hoists invariant code out of loop bodies**
  - may improve execution performance

```
for i in 1..10 { var t = 10 * someConst; } // can hoist t
```

- **LICM is performed on a per-function basis**
  - modifications to local variables are directly visible
    - through direct assignment or a function call (when passed by reference)

```
for i in 1..10 { local = i; f(local); var t = 10 * local; }
```

  - modifications to module-level variables may not be directly visible
    - e.g. through an arbitrary function call

```
for i in 1..10 { modifyGlobal(); var t = 10 * global; }
```

- **Stopped hoisting module-level variables for 1.10**
  - discovered that analysis for module-level variables was incorrect
  - resulted in performance regression for some variants of Fannkuch
    - only affected the slower versions, did not affect the fastest versions

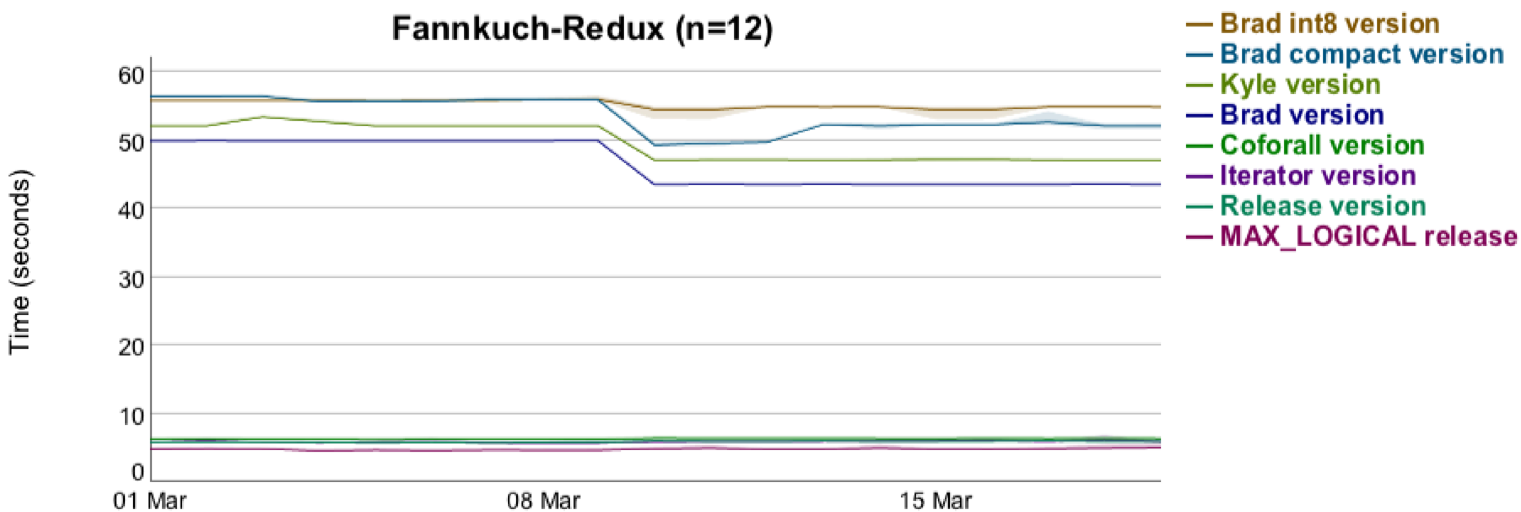
LICM: This Effort

- **Update analysis for module-level variables**
  - check if a loop contains any function calls
  - assume a function call will modify every module-level variable
    - conservative; but simple, cheap to compute, and handles most cases

- **Hoist module-level variables from loops w/o function calls**

LICM: Impact

- Resolved performance regression



- Improved communication counts for several tests

LICM: Next Steps

- **Could do limited interprocedural analysis**
  - instead of assuming function calls modify every module-level variable

- **LICM was not added as a traditional performance optimization**
  - introduced because array meta-data prevented offloading to accelerators
  - many opportunities to improve its analysis and capabilities
    - allow hoisting from loops that contain synchronization constructs
    - make alias analysis less conservative
    - do better analysis of argument intents
    - perform full interprocedural analysis

The “local field” pragma

Compile-time optimization to reduce communication overhead

Local fields: Background

- **Compiler conservatively inserts wide pointers**
  - This approach errs on the side of simplicity+correctness over speed
- **These calls introduce runtime overhead**
  - Structs are used to refer to remote memory
  - The compiler may refer to local memory through this struct

```
typedef struct {
  int localeID:
  void* memory;
} remoteThing;
```

```
int x = 0;
wide_x.localeID = here.id;
wide_x.memory = &x;
// when we want to read x
int local_x;
local_x = comm_get(wide_x);
```

  - Fortunately, the runtime will avoid communication for local memory

```
comm_get(src):
    if src.localeID == here.id :
        return *src.memory;
    else: actual runtime communication
```

Local fields: Background

- **The compiler always inserts communication for fields**
- **This is bad for the C pointers we use inside arrays**
  - Struct access overhead
  - Potential for communication thwarts back-end compiler optimizations
- **The 'local' block tends to save us in distributed code**

  Pros: fairly simple implementation with good performance results

  Cons: Imprecise; scoping issues; difficult to define precise semantics

```
proc dsiAccess(i : int) { // used to read array elements
  local {                 // asserts no comm required to reduce overheads
    if myLocalDomain.member(i) return myData[i];
  }
  // remote code…
}
```

Local fields: This Effort

- **Allow class designers to assert locality for fields**
  - Fine-grained, data-centric assertion
- **Approach: introduce a new pragma "local field"**
  - Only works for class fields within an aggregate type
  - Automatically applied to arrays in an aggregate type

```
class Foo {
  var x : int;
}
class Bar {
  pragma "local field"
  var f : Foo;
}
```

Local fields: This Effort

- **Applied this pragma to C pointers in DefaultRectangular**
  - DefaultRectangular is…
    …the domain map used to implement local arrays by default
    …also used as the guts of virtually every other domain map (e.g., Block)
  - Its pointers should never point to remote data
  - Represents a significant source of overhead given its widespread use

- **Runtime checks inserted to ensure correctness**
  - Invoked on reads or writes of such fields
  - Generates runtime error if field is assigned remote data
  - Can disable with "--no-local-checks"
    - Or with --no-checks or --fast

```
class Bar {
  pragma "local field"
  var f : Foo;
}
```

```
// should always return true
proc Bar.check() {
  return this.locale.id == this.f.locale.id;
}
```

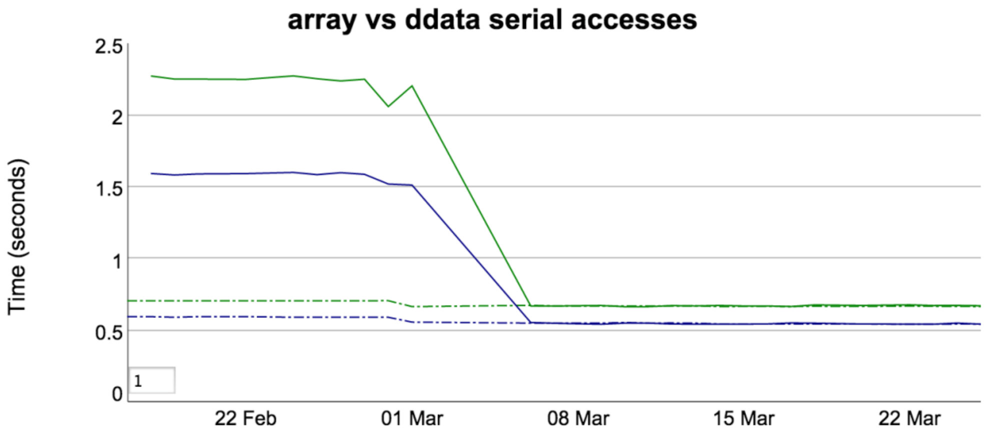
Local fields: Impact

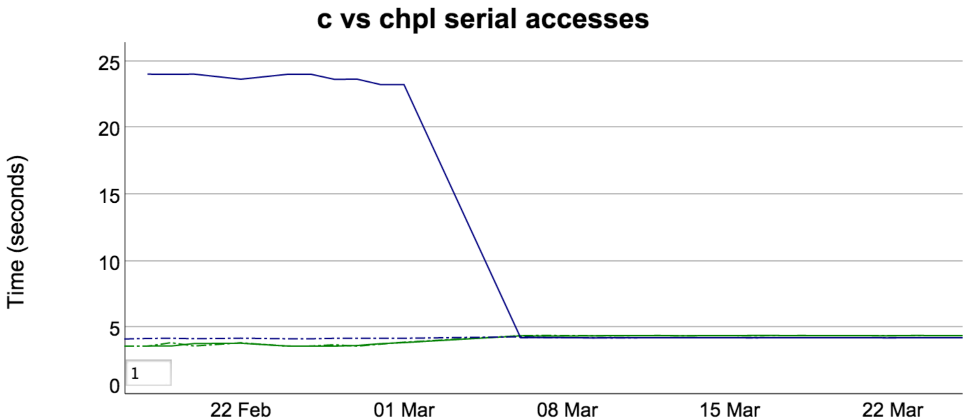
- **Reduced communication overhead for simple cases**

- **Most effective on programs without:**
  - Distributions
  - on-statements
  - local-blocks
  - User-defined classes or records (these don't have the pragma)

Local fields: Impact

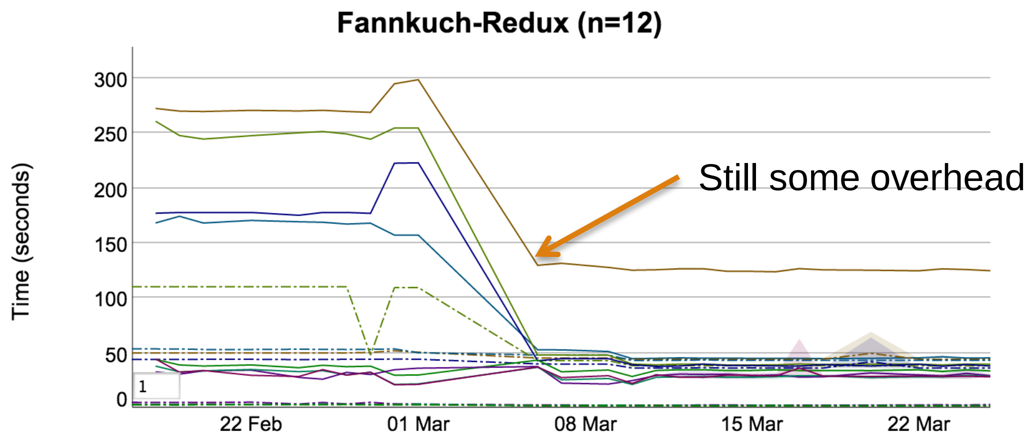
- **Some single-locale tests now have no --no-local overhead**
  - solid lines are --no-local compilations; dashed are --local

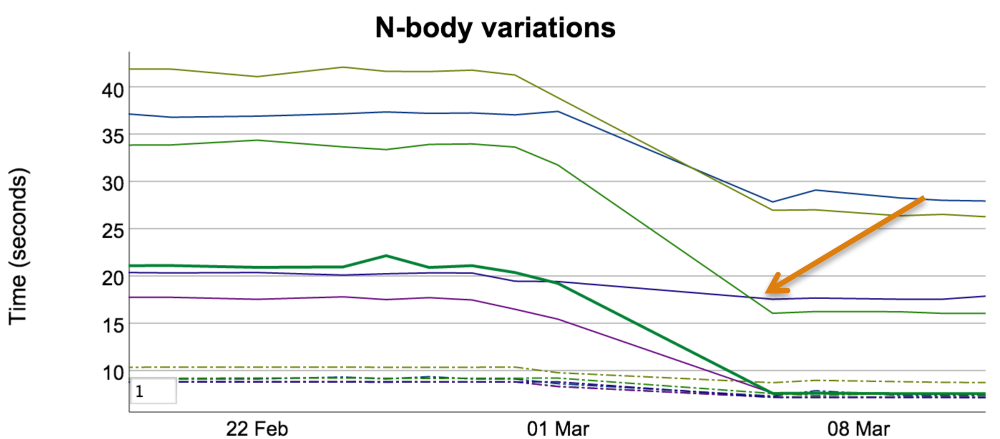




Local fields: Impact

- Other tests improved, but still have some overhead





Local fields: Status

- **Available in the 1.11 release**

- **Only used explicitly in DefaultRectangular**
  - May be applied elsewhere
  - Automatically applied to arrays in aggregate types
    - Based on Chapel semantics
    - These should always match the containing object's locale

- **Little impact on real distributed codes**
  - e.g., HPCC, SSCA#2
  - Use of 'local' blocks was likely eliminating overhead in kernels already
    - Future work: remove local blocks without affecting performance

Local fields: Next Steps

- **This data-centric notion of locality is valuable**

- **Replace pragma with a robust language-level construct**
  - Not just fields
    - Array elements
    - Regularly-scoped variables
    - Arguments? Returned variables?
  - Still in design phase
    - But here's an idea:

```
var baz : local Foo;

var data : [1..10] local Foo;

// Instead of a pragma…
class Bar {
  var f : local Foo;
}
```

Local fields: Next steps

- **Deprecate the 'local' block**
  - This statement is imprecise
  - Scoping rules limit its applicability
  - We would prefer finer-grained, data-centric locality assertions

- **Support Local Array Views**
  - Often a program wants to only work with local array data
    - typically results in similarly conservative "is this element remote?" checks
  - Doing so today is possible, but a bit clunky
  - Sketch of concept:

```
var myLocArrElts = Arr[local];
…myLocArrElts[i,j]…      // fast local access to A[i,j]; OOB if (i,j) is remote
```

  - Current array-view effort provides a framework for this feature

Local Fields: Next steps

- **Given "`on foo do …`"**

- **Avoid on-statement overhead**
  - If `foo` is local, we can avoid runtime overhead for on-statements
  - Namely, avoid allocating bundled arguments
    - This is important for atomic operations, which have on-statements

- **Optimize `foo` within the on-statement**
  - By definition, the on-statement will execute on `foo`'s locale
  - Thus, we know references to `foo` are local within the on-statement

Local Fields: Next steps

- **Determine other opportunities for optimization**
  - Distributed inner loops
  - Function specialization (create local and non-local versions)

The “assertNoSlicing” config param

Avoiding unnecessary multiplication for array accesses

assertNoSlicing: Background

- **Chapel uses a multiplication per dim for each array access**
  - Used to support rich array views: strided slicing, rank-change
  - In most common cases, multiplier for innermost dimension is 1
    - therefore, wasted math

- **These extra multiplications can hurt performance**
  - Particularly compared to C, which never requires such multiplications
  - For memory-bound code, typically a wash
    - e.g., STREAM Triad
  - For codes tuned for the memory hierarchy, can hurt performance
    - e.g., CSU's tiled iterator study for their ICS paper

assertNoSlicing

This Effort:

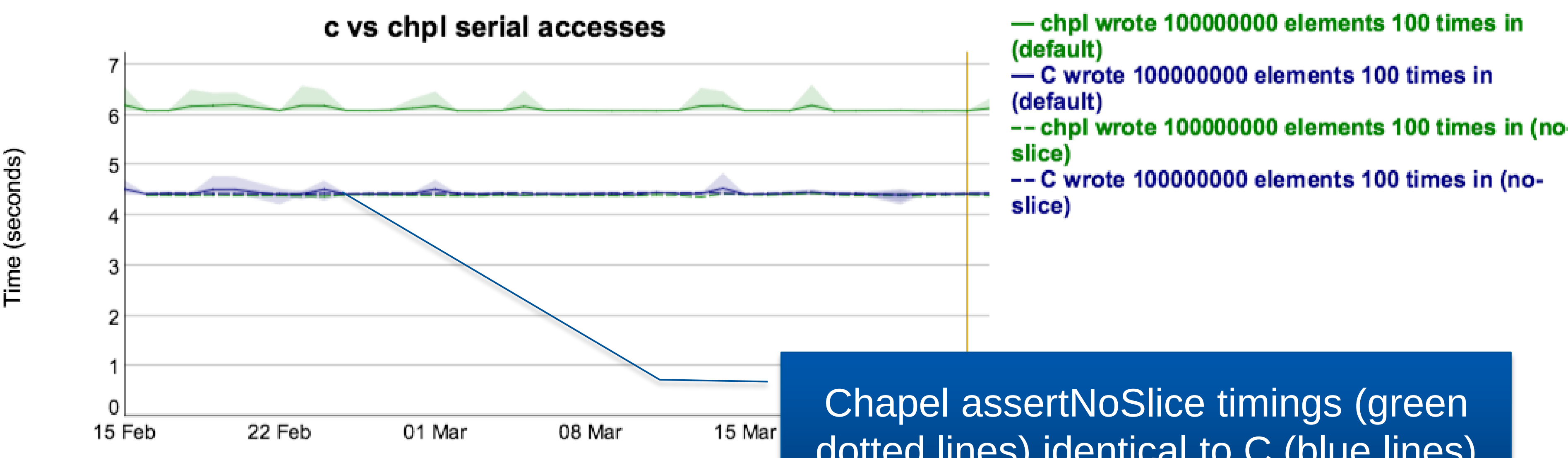
- As a stopgap, add a knob that lets users assert no such mults needed
  - This is a program-wide assertion about every array (so, a big hammer)

    ```
    chpl foo.chpl -sassertNoSlicing
    ```
- This squashes the extra multiplication for all array accesses
- If slicing does occur, the program may have errors
  - e.g., seg faults, incorrect results

assertNoSlicing: Performance improvements

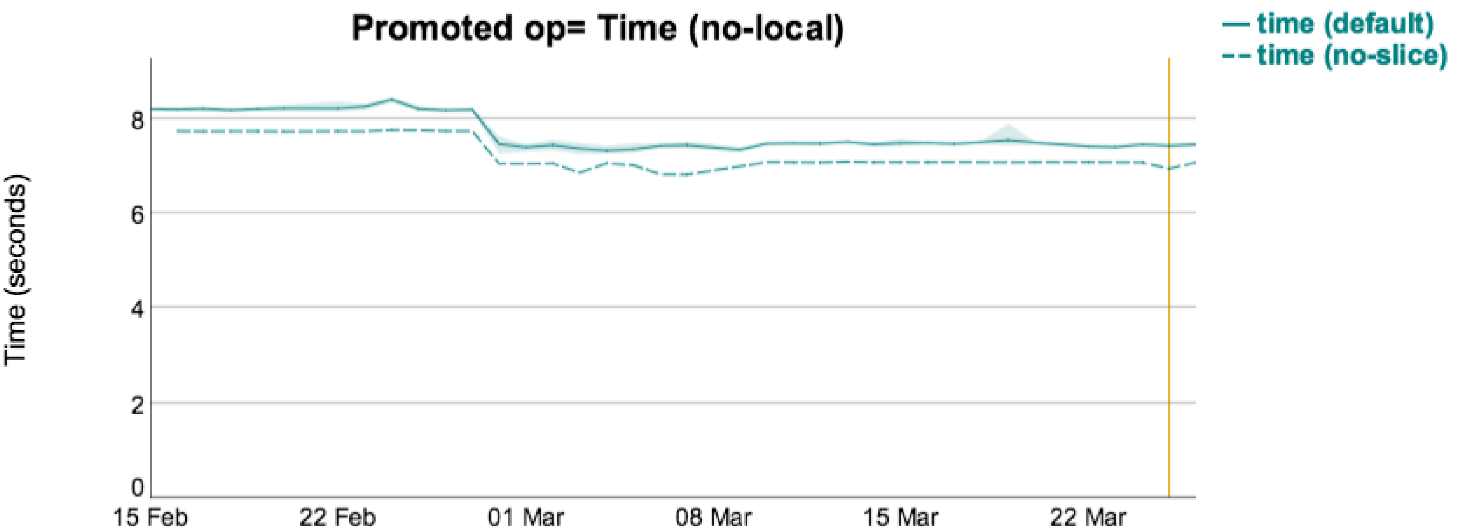
Impact:

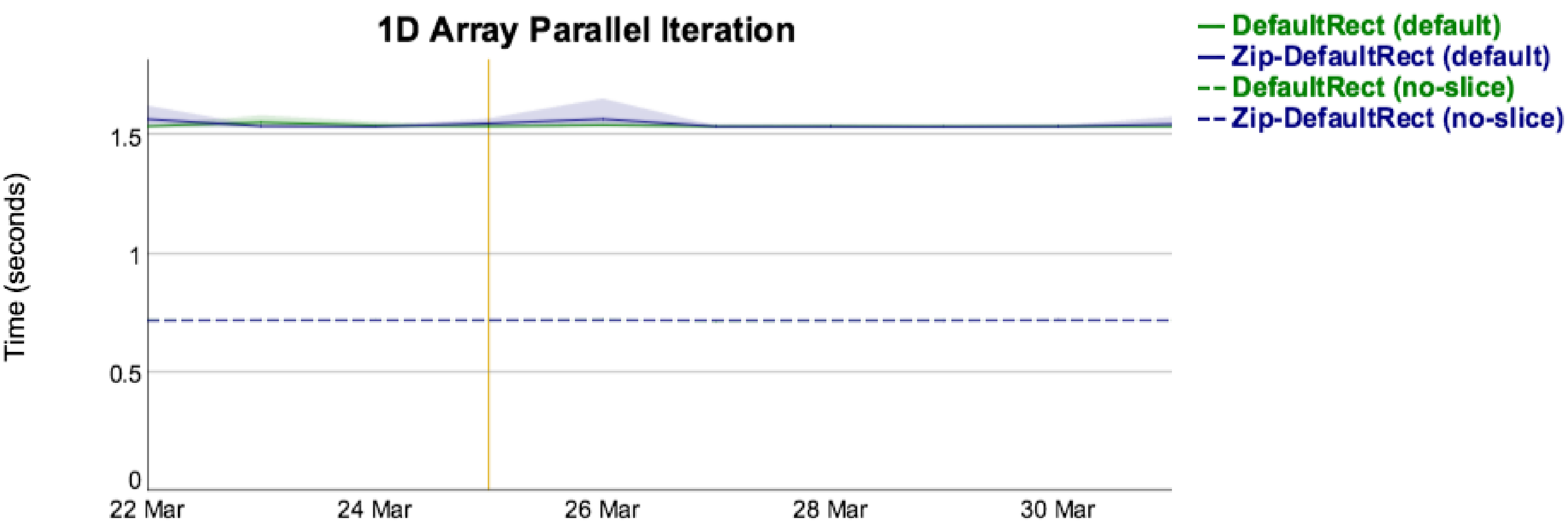
- Performance improvements seen in computationally intensive tests
- Nearly the same as C array performance



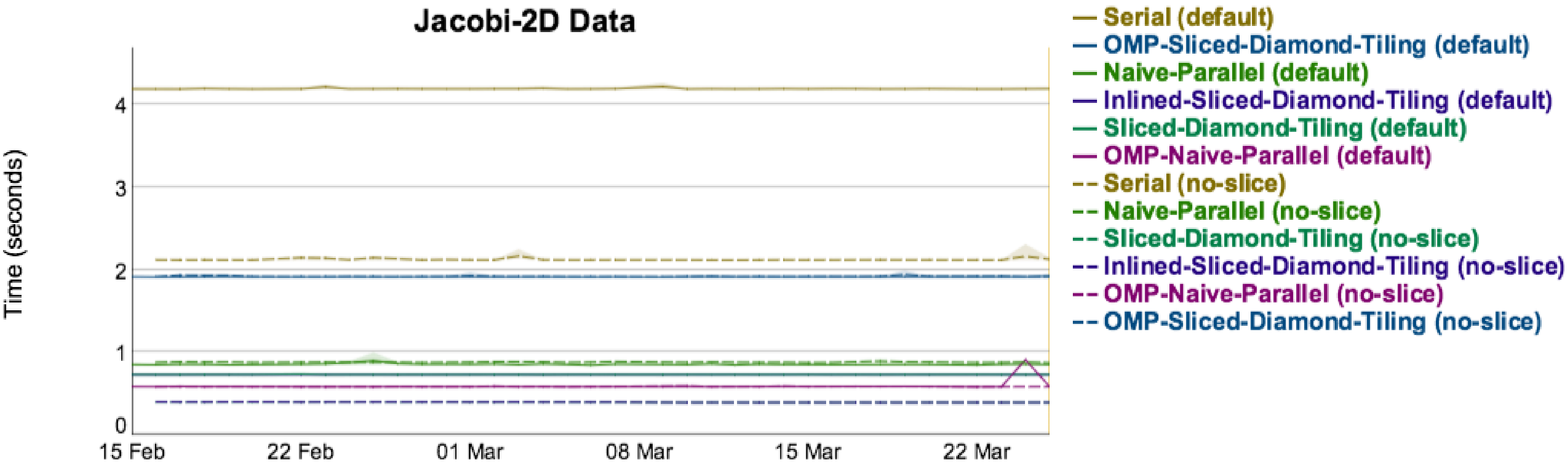
Chapel assertNoSlice timings (green dotted lines) identical to C (blue lines)

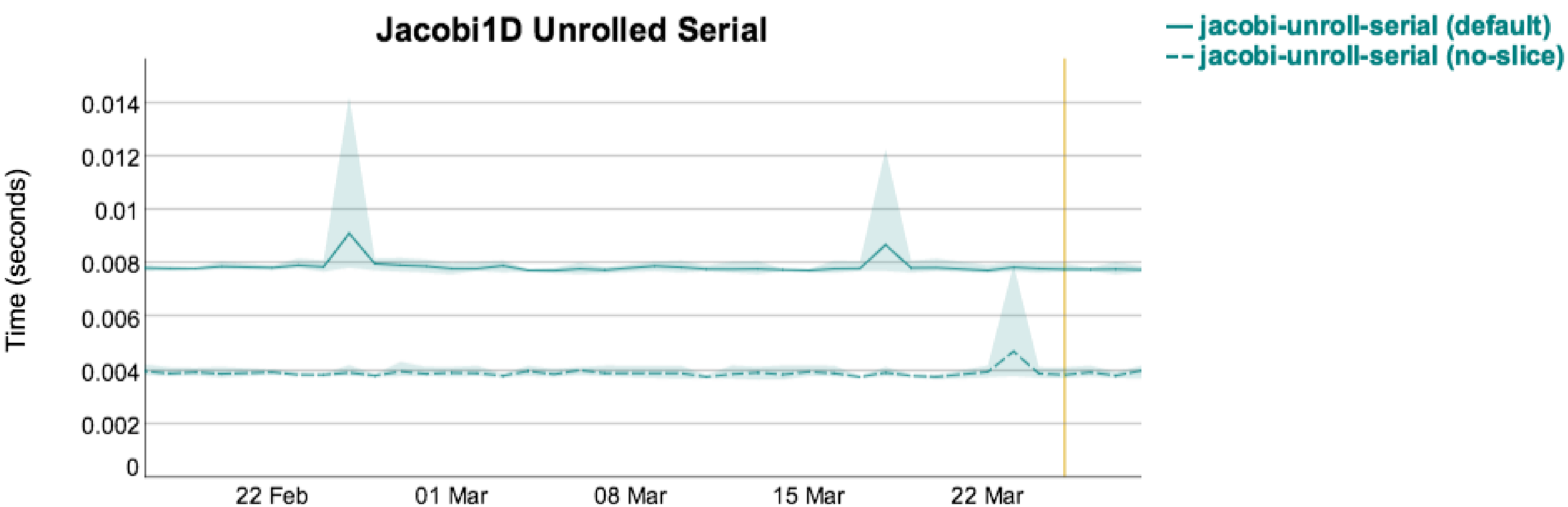
assertNoSlicing: Other Array Idioms





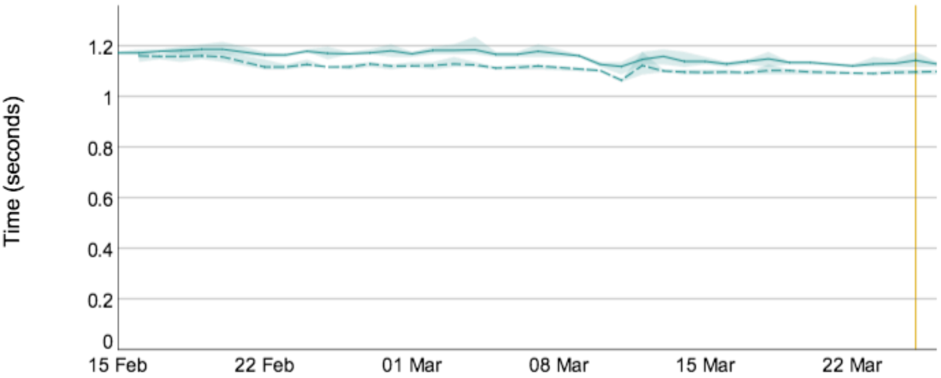
assertNoSlicing: Colorado State Benchmarks



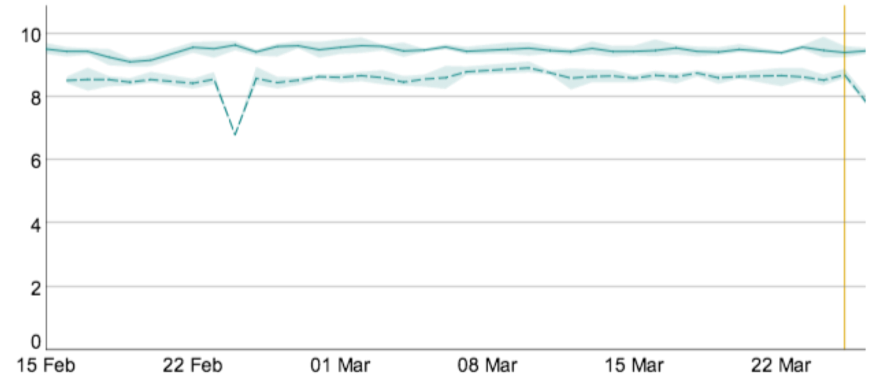


assertNoSlicing: Other Benchmarks

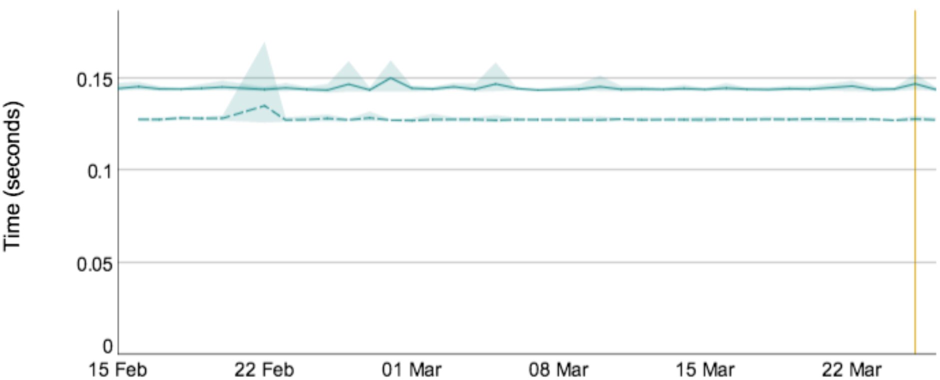
Reverse-complement Shootout Benchmark



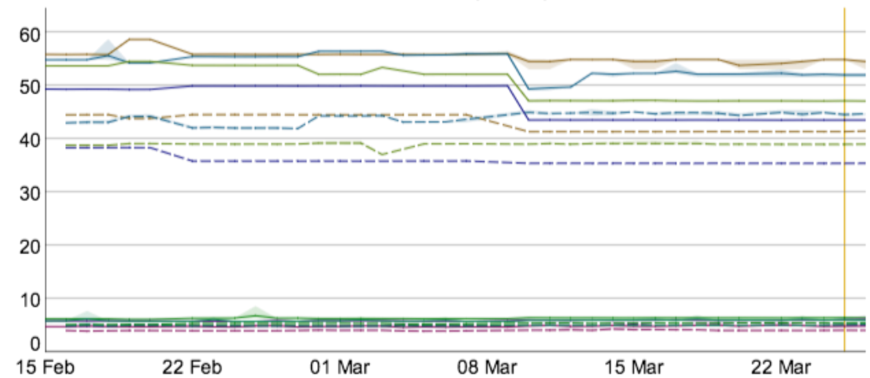
NAS Parallel Benchmarks: FT timings - size A



Parboil SAD Serial Execution Time



Fannkuch-Redux (n=12)



assertNoSlicing

Status: Off by default

- Must be for correctness w.r.t. advanced slicing/rank change operations
- Documented in the $CHPL_HOME/PERFORMANCE file

Next Steps:

- Automatically optimize away these multiplications
  - Effort currently underway as part of “array-view” domain map effort

External Procedures with ‘string’ Arguments

Extern procs with ‘string’ args: Background

Background:

- Some extern functions use line/filename info for error messages
  - They tend to expect an integer and a string

```
void proc foo(..., int line, char* filename);
```

- Such functions should be prototyped in Chapel as:

```
extern proc foo(..., line : c_int, filename: c_string);
```

…yet we were prototyping the filename argument as (Chapel) `string`
  - This works out because historically Chapel strings have been char*’s

- However, it resulted in memory leaks
  - String literal actuals were converted to Chapel strings, then never free’d

- Extern functions shouldn’t be passed Chapel strings anyway
  - The “string” internals are intended to be opaque

Extern procs with ‘string’ args: This Effort

This Effort:

- Fixed existing mismatched extern string/c_string arguments
- Added a compile-time error for extern prototypes taking string args
  - e.g., we now generate an error for:

```
extern proc foo(..., filename: c_string);
```

Impact: Less memory leaked

- Particularly beneficial for the internal NetworkAtomics module

Other Performance Improvements

Other Performance Improvements

- **Reduced overhead of module-scope variable references that are known to be local**
  - (in the "locality" sense, not the "lexical scoping" sense)
- **Localized additional remote variable references**
  - Reduced conservative widening of references that "may be remote"
- **Reduced memory leaks due to compilerWarning()s**
- **Avoided creating singleton tasks for serial scopes**
  - described in runtime deck

Optimization/Codegen Priorities and Next Steps

Opt/Codegen Priorities and Next Steps

- **Complete LCALS port and evaluate Chapel vectorization**
  - success/failure of Chapel vectorization compared to reference
  - performance of vectorized Chapel loops compared to reference
  - improve support if significant gaps exist
- **Continue locality optimization effort**
  - replace 'local' block with data-centric alternatives
    - 'local class' type annotations
    - local array view capability
  - optimize on-clauses
- **Automatically squash inner-dimension multiplications**
  - and deprecate –sassertNoSlicing config
- **Continue reducing size/complexity of generated code**
  - correlates to time spent in compilation

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, URIKA, and YARCDATA. 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.

http://chapel.cray.com chapel_info@cray.com http://sourceforge.net/projects/chapel/