Benchmark Improvements

Chapel Team, Cray Inc.
Chapel version 1.9
April 17th, 2014 (released) / May 2014 (documented)
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.
Three types of graphs in this presentation

1. **Historical:**
   - shows performance for each release, for the benchmark at that time
   - for a few new benchmarks, we’ve gathered this retroactively
   - tracks improvements to both the benchmark and to Chapel

2. **Nightly:**
   - shows the automated results we gather on a nightly basis
   - can be viewed at [http://chapel.sourceforge.net/perf/chap04/](http://chapel.sourceforge.net/perf/chap04/)

3. **Release-over-release:**
   - measures today’s benchmark code using prior releases
   - factors out changes to the benchmark itself
     - not to mention changes to OS, back-end compiler versions, etc.
   - in some cases, today’s code doesn’t compile with older versions
   - can be viewed at: [http://chapel.sourceforge.net/perf/chap04/releaseOverRelease/](http://chapel.sourceforge.net/perf/chap04/releaseOverRelease/)

*Unless noted otherwise, performance is for an 8-core workstation (chap04)*

Note that the vast majority of graphs in this slide deck are reporting execution times, so lower is better. The primary outliers are the HPC Challenge benchmark results near the end, which typically use performance metrics like GB/s. These cases are called out in their notes sections.
Outline

- **Computer Language Benchmark Game** ("shootout") Codes
  - New benchmarks
    - Meteor
    - Fannkuch-redux
  - Revised benchmarks
    - Spectral-norm
    - Mandelbrot
    - Chameneos-redux
    - Fasta
  - Performance changes in stable benchmarks
    - Pidigits
    - Thread-ring
    - N-body
    - Binary-trees

- **Other Notable Single-Locale Benchmark Results**
Computer Language Benchmarks Game

Contest’s Goal:
- see how languages compete over 13 modestly sized benchmarks
  - performance: serial or multicore, timing complete program execution
  - code size: as measured by the code’s compressed size
- for more information: http://benchmarksgame.alioth.debian.org/

Our Goal:
- assemble an entry that is competitive with C in performance
- use this to raise awareness of Chapel in mainstream/open-source
- use this as a forcing function for looking at serial performance
- the effort has had a positive impact on multi-locale cases as well
Meteor
Each of the 10 pieces is a unique shape that can be rotated and flipped before being placed on the board. Every piece must be used in the solution.
These versions are known as meteor-parallel and meteor-parallel-alt inside of the test/studies/shootout/meteor/kbrady/ directory.
On this graph, the previously mentioned meteor is marked as ‘release/meteor-parallel’ and meteor-fast is ‘meteor-parallel-alt’.

The series meteor and meteor-implicit-domain are single threaded versions of meteor-parallel. The gap between them is caused by a large number of copies that occur when array type is fully specified for formal arguments.
Meteor: historical (gathered retroactively)

![Meteor Graph]

Copyright 2014 Cray Inc.
The noise in this graph is puzzling. For the Oct 13 and Apr 14 data points, the first of the three trials got an unexplainable timing of 1.x seconds compared to the norm which is a tiny fraction of a second (as seen in the Apr 13 timings and the nightly graph shown earlier. We’re including this graph here for completeness rather than because it’s particularly useful. The historical graph of the previous slide is essentially the same information (since it’s all retroactively gathered) and far more indicative of the performance we typically see. It seems likely that there is some artifact in our testing system that is causing the overhead for the first run of these release-over-release timings.
Meteor

Next steps:

- **meteor-fast**
  - Its intrinsics can be made portable using the new BitOps module
  - Promote into the release

- **Remove penalty for fully-specified formal arguments**

  ```
  var A: [1..n] real;
  
  proc foo(A: [1..n] real) { ... }
  proc bar(A: [] real) { ... }
  proc baz(A: [?D] real) { ... }
  ```

  - Of these three routines, the first is more expensive by far
    - the reason: this syntax reindexes the actual to match the formal's domain
    - the common case of the domains being the same is not optimized
  
  - The language, compiler, and/or modules should be updated to remove this penalty

Meteor-fast was very close to being in the 1.9 release, but was held back due to portability issues. The BitOps module committed after the 1.9 release will let us fix that easily.
Fannkuch-Redux
The benchmark takes every permutation of \{1,\ldots,n\} and performs a few steps over them:

1. Take the first element, X
2. Reverse the first X elements of the sequence
3. Repeat until the first element is 1
Fannkuch-redux: historical (gathered retroactively)
Fannkuch-redux: release-over-release

Fannkuch-Redux (n=12)

- Brad compact version
- Brad int8 version
- Kyle version
- Release version
- Brad version

Time (seconds)

JUL 12 | OCT 12 | JAN 13 | APR 13 | JUL 13 | OCT 13 | JAN 14 | APR 14

0 | 20 | 40 | 60 | 80 | 100

Copyright 2014 Cray Inc.
‘Brad compact version’ is noisy due to the use of a reduction
Without a parallel version we will not stack-up against other versions in the multi-core tests.

In analyzing the final optimized assembly I noticed that the C compilers (gcc 4.8 / clang) were turning one of the loops in the reference C version into a memcpy, but not in ours. Getting our loops into a form where the backend compiler will perform this optimization would be a small performance win.
Spectral-norm
Conjugate transpose is the transposition \((A(i,j) \Rightarrow A(j,i))\) of a conjugate matrix, where the conjugate of a complex number \(a + bi\) is \(a - bi\).

Eigenvalues are a special set of scalars associated with a linear system of equations.
Spectral-norm: historical

![Spectral-Norm Chart]

- Chapel
- C reference

1.7 release, 1.8 release, 1.9 release
• these results were gathered on chap03, a 2-core workstation

• ‘Blockdist’ and ‘barrier’ were the original versions, written by Albert Sidelnik

• The line labeled “spectralnorm” (in light blue, visible on the second set of lines from the top) was an initial cleaned up version, which did not use block
distributions or a barrier.

- The two lines labeled “two-at-a-time” and “two-at-a-time-barrier” were based on the gcc #4 reference version, where tasks were created every two iterations instead of every single iteration.
The fastest version shown here involved manually removing nested parallelism. This was motivated by observations related to different task creation policies on small-core-count machines (shown on the next slide). The impact of this result led us to examine – and eventually flip-- the default value of dataParIgnoreRunningTasks. This improved most of the remaining versions which relied on nested parallelism (specifically, a reduction within a forall loop).

The “brad” version squashes the reduction’s parallelism using ‘serial’ statements (and it also included other style changes, including different writes, division instead of bit shifts, formal argument domain query syntax, and alternate methods of array access). This version’s performance improved when we reduced the number of tasks used for data parallel constructs within serial statements.
Spectral-norm

**Next Steps:**
- The benchmark itself is in very good shape
- Remaining performance gap likely due to general Chapel overheads
  - e.g., make Chapel’s reduction more competitive with the hand-coded one
Mandelbrot
Mandelbrot

Overview:
- Computes & plots the Mandelbrot set [-1.5-i, 0.5+i] on an n x n bitmap
- Emphasizes small unsigned integers, multidimensional arrays, and binary output, as well as some bit operations
Mandelbrot: historical

![Mandelbrot Chart]

- Chapel
- C reference

1.7 release
1.8 release
1.9 release

seconds

Copyright 2014 Cray Inc.
Mandelbrot: release-over-release
The ‘complex’ and ‘dist’ versions were the original Chapel Mandelbrot versions written by Jacob Nelson. They were very slow, 19x – 40x slower than the reference version and are shown here to emphasize the improvement made by all subsequent versions. ‘dist’ used a Block distribution, setting it up for distributed memory execution, but adding overhead for the shared-memory shootout competition. The ‘complex’ version uses complex types and math rather than scalar floating point values.

The ‘no-dist’ version is based on a version named ‘mandelbrot-fancy’, also developed by Jacob, but which ran out of memory.
The original versions of the benchmark did not make use of Chapel I/O

The ‘blc’ version is essentially a cleaned-up version of mandelbrot-unzipped.

Not shown is the no-local improvement generated by using the bulk array write.
Mandelbrot

Next Steps:
- Determine the cause of remaining ~3.5x performance gap
- Consider strength reduction for divisions by powers of two
  - or ensure that the C compiler will do this for us

Future Work:
- Explore use of complex types/operations rather than uints
  - What optimizations would be required to minimize differences?
Chameneos-redux
The stylistic improvements alluded to include removing redundancy while simplifying the code, moving the ownership of certain procedures, and converting some variables to constants.
Chameneos-redux: historical

![Chameneos Chart]

Copyright 2014 Cray Inc.
Note: nightlies show nearly no change between 1.8-9
Chameneos-redux: nightly

Chameneos Redux Shootout Benchmark (n=6,000,000)

Time (seconds)

Oct 13 Nov 13 Dec 13 Jan 14 Feb 14 Mar 14 Apr 14

Chameneos-spin-loop
Chameneos-yield-loop
Chameneos-waitFor
Chameneos-cas
Chameneos-bic
release
Note: data has been smoothed to increase clarity of actual trends. However, due to the high variability of data in these versions, some trends are visible without necessarily meaning anything. The envelopes provided overlap almost completely, indicating that while a test may on average be faster than others, it will not necessarily be the fastest each night. The only notable exception is the new version of chameneos. The gentle
slope seen for the red line hides the sharp drop experienced when the release version was converted to adopt code from chameneos-blc.
Chameneos-redux

Next Steps:
- Make Qthreads tasking the default and check impact on performance
- Find and optimize overheads in the scalar code paths
Fasta
This is the only real change to the fasta code since the last release – both in terms of benchmark code changes and performance code changes.
Fasta: historical
Fasta: release-over-release

Fasta Shootout Benchmark (n=25,000,000)
Pidigits
The fact that Chapel has traditionally beat the top C versions in our comparisons seemed suspicious, but we hadn’t taken the chance to investigate until now (see following slides)
Pidigits: historical

- Traditionally, pidigits in Chapel has been surprisingly fast:
The C versions weren’t measured against the hand-built GMP versions mostly out of laziness – it’s slightly painful to override the system version of GMP and the trends were pretty clear from these measurements.
These timings are from a different machine than the previous slide; that’s why the numerical values don’t match.
These timings are from a different machine than the previous slide; that’s why the numerical values don’t match.
Pidigits Next Steps: defaults in releases

• **Decide whether to build GMP by default**
  • Possible proposal:
    • try to build, ignoring failures
    • current CHPL_GMP logic will default to bundled version if build worked
    • main downside: won't default to using system version (same as today)

• **Decide whether to switch to bundled version for Crays**
  • today we're using the system version on Crays by default
    • this turns out to be 4.2.3
    • so, bundled version could be better
The aforementioned 32- vs. 64-bit portability bug:

```plaintext
var k: uint;

do {
  do {
    k += 1;
    const y2 = 2^k + 1;
    ...
    mpz_mul_ui(accum, accum, y2);
    mpz_mul_ui(numer, numer, k);
    mpz_mul_ui(denom, denom, y2);
  } while (...);
  ...
} while (...);
```

Problem: `mpz*ui` routines expect a C unsigned long.

Chapel's `uint` type is 64 bits and not guaranteed to fit in a C unsigned long.
Not ideal because if, to use GMP, you have to use C types everywhere, what does that imply for your Chapel code?

The primary alternative would be to have Chapel's GMP routines downcast its int arguments to the appropriate C types; but at what cost/risk? Could, for example, have a safer but more expensive vs. cheaper and more risky mode which is guided by a --fast-controlled flag.
More specifically, what we might want/need to support this are:

- promotion of initialization assignments to a language-level concept
- and/or the ability to define a defaultInitialize() function on a specific type when its initializer isn’t present
- and the promotion of user-defined casts to a language-level concept

In the cast case, there’s also a challenge related to the desire to take the mpz_t that I’m imagining would be created and returned by the cast function and steal it for use by ‘numer’ rather than requiring a copy from one mpz_t to another.
The challenges alluded to in the final bullet here relate to the fact that to use GMP best, you’d really want to recognize and match against multi-expression templates. Failure to do so requires extra temporary variables that would either have to be reference counted or leaked.

But how to support such multi-expression templates for external types that the compiler doesn’t know about or know how to reason about?
Why are mpz_t overloads ambiguous with existing operators? Because mpz_t types are 1-element arrays in C and for that reason are represented as 1-tuples in Chapel, conflicting with our tuple overloads.

The compiler-introduced temps bullet refers not only to the fact that the compiler’s inserting such temps (which we probably don’t want), but also to the fact that such temps are not l-values, yet most GMP functions currently take their arguments by ref. This could potentially be resolved by changing such read-only GMP arguments to take their arguments by const ref – I haven’t tried that yet.
Despite the challenges, the potential is tantalizing:

```c
var numerator, accumulator, denominator, temp1, temp2, temp;
mpf_init_set_ui(numerator, 1);
mpf_init_set_ui(accumulator, 0);
mpf_init_set_ui(denominator, 1);
mpf_init(temp);
mpf_init(temp2);
```

```c
var numerator = 1;
accumulator = 0; denominator = 1;
temp1 = temp2 = temp;
```

VS.

```c
var numerator = 1; numerator = 2;
accumulator = y2;
numerator = b1;
denominator = y2;
```

```c
while (mpf_cmp(numerator, accumulator) > 0) {
    mpf_mul_ui(temp1, numerator, 1);
    mpf_add(temp1, temp1, numerator);
    mpf_add(temp1, temp1, accumulator);
    mpf_div_ui(temp1, temp1, denominator);
    mpf_add(temp2, temp2, numerator);
} while (mpf_cmp(temp2, denominator) >= 0);

const d = mpf_get_ui(temp1);
mpf_mant_set_ui(accumulator, denominator, d);
mpf_mant_ui(accumulator, denominator, 10);
mpf_mant_ui(numerator, denominator, 10);
```
Note that if we went directly to this approach, it would allow us to dodge several of the previous challenges.

Yet, the downside to doing so is that other user-defined external types would not enjoy these benefits.
Thread-ring
Thread-ring

Overview:
- Passes a token $n$ times among $n$ threads
- prints the thread that ends up holding the token
- Problem Size: $n = 50,000,000$, $n$ threads $= 503$
- Emphasizes tasking and synchronization variables
Thread-ring: historical

![Thread-Ring Diagram]

- Chapel
- C reference

1.5 release
1.3 release
Most of the historical timings time out for the four different versions. One of the three runs of the Oct12 execution happened to not time out which is why there is one line with two data points. This graph primarily shows that the versions have gone from (typically) timing out to completing in version 1.9.
Using begin statements within a coforall loop is still the fastest way to implement this program, although the other methods are not as far behind as they used to be. It was theorized that leftover threads were being repurposed instead of initializing new ones, leading to the timing difference. Testing prior to the sync variable change seemed to confirm that theory, although no tests were performed after that change and the change back to the old thread-waiting version.
Thread-ring

Next Steps:
- Work on making Qthreads the default to get lower tasking overheads
- Map sync vars to Qthreads’ full-empty variables
- Identify and fix remaining performance gaps compared to references
- Consider additional optimizations to ‘fifo’ tasking and sync vars
N-body
From this point out, less and less has been done with the benchmarks themselves, so we report less on work done and simply present the performance graphs and key changes.
N-Body: all versions

N-body variations

Time (seconds)

July 12 Oct 12 Jan 13 Apr 13 July 13 Oct 13 Jan 14 Apr 14

Slice-based b/c version
Forloop
Record domain
Zip-based b/c version
Original
Release version
Record
Blc version

Copyright 2014 Cray Inc.
N-Body: Nightly results

N-body variations

- slice-based blc version
- forloop
- record domain
- zip-based blc version
- original
- release version
- record
- blc version

Removed extra formal temps
N-Body: best versions

N-body variations

- slice-based blc version
- forloop
- record domain
- zip-based blc version
- original
- release version
- record
- blc version

Removed extra formal temps
Binary-trees
Binary-trees: release-over-release

Binary Trees Shootout Benchmark (n=20)
Binary-trees: nightly

Binary Trees Shootout Benchmark (n=20)

- binary-trees_iter
- binarytrees
- binary-trees
- binary-trees-freeWhileAdding

Time (seconds)

Oct 13 Nov 13 Dec 13 Jan 14 Feb 14 Mar 14 Apr 14
Shootout Summary
Shootout Performance Summary (v1.8)

Chapel x worse than best reference

The graph shows a comparison of performance metrics, with Chapel 1.8 performing worse than the best reference across various categories.
Shootout Performance Summary (v1.9)

Chapel x worse than best reference

[Bar chart showing the performance comparison between Chapel 1.9 and the best reference, with Chapel being worse in several categories.]
Shootout Performance Summary (v1.8-v1.9)

Chapel x worse than best reference
Outline

- Computer Language Benchmark Game ("shootout") Codes
- Other Notable Single-Locale Benchmark Results
  - LULESH
  - MiniMD
  - SSCA#2
  - HPC Challenge Benchmarks
  - NAS Parallel Benchmarks
Other Notable Benchmark Results

- The following benchmarks are important to us, yet have not received much attention in the 1.9 release cycle
  - we’ve focused on the shootouts to fix serial performance issues
  - these will be more compelling for HPC users than the shootouts
  - graphs show that they’ve generally improved in spite of this
    - though there are a few performance slips as well

- These are all multi-locale benchmarks
  - the test results here represent single-locale executions
    - some cases were compiled --no-local to track multi-locale code overheads
LULESH
LULESH: historical

![LULESH Diagram]

- **Chapel (dense materials)**
- **Chapel (sparse materials)**
- **Reference**

*Time (seconds)*

- 1.7 release
- 1.8 release
- 1.9 release
Note that the previous slide showed –local timings only. The –no-local cases got better during this release cycle.
We haven’t had the chance to check what the final performance regression in LULESH is due to. The most likely candidates are:

- we started using the –static flag for these performance tests
- we moved the task counting from the runtime to the module
- we changed some ‘inout’ intents to ‘ref’ intents in the module code (but primarily for I/O which seems unlikely to be the cause here)
LULESH Study: release-over-release

LULESH (bradc study version)

- release version
- dense (no-local)

Time (seconds)

0 1 2 3 4 5 6
Jul 12 Oct 12 Jan 13 Apr 13 Jul 13 Oct 13 Jan 14 Apr 14

COMPUTE | STORE | ANALYZE

Copyright 2014 Cray Inc.
LULESH Study: nightly

- hybrid sync var implementation and later bug fix
- Flipped dataParIgnoreRunningTasks
- Candidate pair removal

LULESH (bradc study version)

- release version
- dense (no-local)
miniMD
miniMD: historical
miniMD: release-over-release
miniMD: nightly

miniMD LJ (--size=10) Time

- stencil nolocal
- explicit nolocal
- simple block nolocal
- simple nolocal
- simple
- old

Nested parallelism
Improved -snoRefCount
SSCA#2
Note that this is a performance slide, and that therefore higher is better.
HPC Challenge Benchmarks
Note that these are performance slides and that higher is better.
Note that on this slide, higher is better
Note that on this slide, higher is better
Note that on this slide, higher is better
Note that on this slide, higher is better
NAS Parallel Benchmarks
NAS Parallel EP-D: historical

![Bar chart showing performance comparison between Chapel and C reference across different releases.](image)
NAS Parallel EP-B: nightly

NAS Parallel Benchmarks: EP timings - size B

- runtime - B

Oct 13 Nov 13 Dec 13 Jan 14 Feb 14 Mar 14 Apr 14

1 2 3 4 5 6 7 8 9 10

 Nested parallelism
 Flipped dataParIgnoreRunningTasks
 Improved constness and remote value forwarding
NAS Parallel CG-A: historical

![Bar chart showing performance of NAS CG-A across different releases](chart.png)
NAS Parallel FT-A: nightly

NAS Parallel Benchmarks: FT timings - size A

— runtime - A

hybrid sync var implementation and later bug fix
NAS Parallel IS-A: historical
NAS Parallel IS-S: release-over-release

NAS Parallel Benchmarks: IS timings - size S

- setup time - S
- key sort time - S
- total time - S
- verify time - S

July 12 Oct 12 Jan 13 April 13 July 13 October 13 January 14 April 14
NAS Parallel IS-S: nightly

NAS Parallel Benchmarks: IS timings - size S

- setup time - S
- key sort time - S
- total time - S
- verify time - S

Oct 13 Nov 13 Dec 13 Jan 14 Feb 14 Mar 14 Apr 14

hybrid sync var implementation and later bug fix
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.

Copyright 2014 Cray Inc.