Benchmark Improvements

Chapel Team, Cray Inc.
Chapel version 1.14
October 6, 2016
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

- LCALS Improvements and Status
- Computer Language Benchmark Game (CLBG)
- Other Benchmark Improvements
LCALS Improvements and Status
LCALS: Background

- **LCALS: Livermore Compiler Analysis Loop Suite**
  - Loop kernels designed to measure compiler performance
  - Developed by LLNL
  - [https://codesign.llnl.gov/LCALS.php](https://codesign.llnl.gov/LCALS.php)

- Three loop subsets (30 kernels total)
  - Subset A: Loops representative of application codes
  - Subset B: Simple, basic loops
  - Subset C: Loops extracted from “Livermore Loops coded in C”

- Each kernel is run for three sizes (Short, Medium, Long)

- Each kernel is implemented in a number of “variants”
  - RAW (traditional C usage), OpenMP, C++ template-based, etc.
LCALS: This Effort

- All kernels/variants/sizes performance tested nightly
- Benchmark run on a 24-core Cray XC30 node
  - Compare serial performance vs. C++
  - Compare parallel performance vs. C++ with OpenMP
  - Compiled with g++ 6.2.0
- Working toward matching or beating g++ with OpenMP
  - Serial variant is on par with g++ for almost all kernels
  - Parallel variant is off by 2x-3x in most cases for Long problem size
    - Best case: PIC_2D nearly 40% faster
    - Worst case: COUPLE over 7x slower
  - Short problem size parallel variant is much faster than C++ version
    - --dataParMinGranularity=1000 option limits parallelism to reasonable level

Compilation/Execution commands:

- chpl --fast --no-iieee-float lcals-chpl --dataParMinGranularity=1000
- g++ -Ofast -fopenmp lcals.exe
LCALS: Improvements since 1.13

- Removed `-sassertNoSlicing` config param setting
  - Subsumed by the (default-on) compiler optimization `--optimize-array-indexing`

- Set `dataParMinGranularity` to 1000
  - Small and medium loop sizes created too many tasks without this setting

- Replaced several loops over arrays with whole array assignments

- Added an output mode that is easier to parse by scripts

- Enabled performance tracking for all variants, kernels, and loop sizes
  - Now test and track the performance of all 156 variants/kernels/sizes daily
  - Graphs are online at the [Chapel Performance Overview](https://www.chapel-lang.org/overview/) website

- Significantly increased `PIC_2D` performance with improvements to atomics
LCALS Status: Serial Performance v1.13.0

Long problem size
(Similar results for medium and short problem sizes)

Serial Chapel vs g++
Normalized time – serial reference is 1.0

Normalized Time

0 0.5 1 1.5 2 2.5 3

0 0.5 1 1.5 2 2.5 3

PRESSURE_CALC
ENERGY_CALC
VOL3D_CALC
DEL_DOT_VEC_2D
COUPLE
FIR
INIT3
MULADDSUB
IF_QUAD
TRAP_INT
HYDRO_1D
ICCG
INNER_PROD
BAND_LIN_EQ
TRIDIAG_ELIM
EOS
ADI
INT_PREDICT
DIFF_PREDICT
FIRST_SUM
FIRST_DIFF
PIC_2D
PIC_1D
HYDRO_2D
GEN_LIN_RECUR
DISC_ORD
MAT_X_MAT
PLANCKIAN
IMP_HYDRO_2D
FIND_FIRST_MIN

Normalized time – serial reference is 1.0

chpl --fast
--no-ieee-float

--fast

--no-ieee-float

Copyright 2016 Cray Inc.
LCALs Status: Serial Performance v1.14.0

Long problem size
(Similar results for medium and short problem sizes)

Serial Chapel vs g++
Normalized time – serial reference is 1.0

<table>
<thead>
<tr>
<th>Function</th>
<th>g++ serial</th>
<th>Chapel serial</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRESSURE_CALC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENERGY_CALC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VOL3D_CALC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEL_DOT_VEC_2D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIRE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INIT3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULADD_SUB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IF_QUAD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRAP_INT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HYDRO_1D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INNER_PROD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BAND_LIN_EQ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRIDIAG_ELIM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EOS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INT_PREDICT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIFF_PREDICT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIRST_SUM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIRST_DIFF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PIC_2D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PIC_1D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HYDRO_2D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GEN_LIN_RECUR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DISC_ORD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAT_X_MAT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PLANCKIAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HYDRO_FIRST_MIN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIND_FIRST_MIN</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

chpl --fast
--no-ieee-float

g++ -Ofast -fopenmp

Copyright 2016 Cray Inc.
LCALS Status: Serial Performance v1.14.0

Long problem size
(Similar results for medium and short problem sizes)

Serial Chapel vs g++
Normalized time – serial reference is 1.0

---replace-array-accesses-with-ref-temps optimization fixes IF_QUAD performance

chpl --fast
--no-ieee-float

g++ -Ofast -fopenmp
LCALS Status: Serial Performance v1.14.0

Long problem size
(Similar results for medium and short problem sizes)

Serial Chapel vs g++
Normalized time – serial reference is 1.0

Chapel serial

Normalized Time
0.0
0.5
1.0
1.5
2.0
2.5
3.0

g++ serial

Long problem size

Mis-translation: The main iteration range of TRAP_INT should use int(32) instead of int(64). Fixed on master after 1.14.

chpl --fast
--no-ieee-float

Long problem size

g++ -Ofast -fopenmp

Copyright 2016 Cray Inc.
Long problem size
(Similar results for medium and short problem sizes)

Serial Chapel vs g++
Normalized time – serial reference is 1.0

Normalized Time

0.5
1
1.5
2
2.5
3

Precision CALC
ENERGY_CALC
VOL3D_CALC
DEL_DOT_VEC_2D
FIR
INIT1
MULADD1
IF_QUAD
TRAP_INT
HYDRO_1D
CCG
INNER_PROD
BAND_LIKE
TRIDIAG_ELIM
EOS
ADI
INT_PREDICT
DIFF_PREDICT
FIRST_SUM
FIRST_DIFF
PIC_2D
PIC_1D
HYDRO_2D
GEN_LIN_RECUR
DISC_ORD
MAT_X_MAT
PLANCKIAN
IMP_HYDRO_2D
FIND_FIRST_MIN

chpl --fast
--no-ieee-float

Applying loop fission to two main loops fixes HYDRO_2D performance.

g++ serial
Chapel serial

Long problem size (Similar results for medium and short problem sizes)

Applying loop fission to two main loops fixes HYDRO_2D performance.

chpl --fast
--no-ieee-float

g++ -Ofast -fopenmp
Long problem size

Parallel Chapel vs g++/OMP

Normalized time – parallel reference is 1.0

chpl --fast
--no-ieee-float

g++ -Ofast -fopenmp
LCALS Status: Parallel Performance v1.14.0

Long problem size

Parallel Chapel vs g++/OMP

Normalized time – parallel reference is 1.0

Normalized Time

Normalized time

chpl --fast
--no-ieee-float

g++ -Ofast -fopenmp
Long problem size

Parallel Chapel vs g++/OMP

Normalized time – parallel reference is 1.0

Setting `--dataParMinGranularity=1000` caused COUPLE to not use enough tasks.

`g++ OMP`  `Chapel parallel`

`chpl --fast --no-ieee-float`

`g++ -Ofast -fopenmp`
LCALS Status: Parallel Performance

Short problem size

Parallel Chapel vs g++/OMP

Normalized time – parallel reference is 1.0

Normalized Time

<table>
<thead>
<tr>
<th>Function</th>
<th>g++ OMP</th>
<th>Chapel parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRESSURE_CALC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENERGY_CALC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VOL3D_CALC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEL_DOT_VEC_2D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COUPLE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INIT3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULADD_SUB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IF_QUAD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRAP_INT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PIC_2D</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

chpl --fast
--no-ieee-float

g++ -Ofast -fopenmp
LCALS: Next Steps

- **Continue optimization effort for parallel kernels**
  - Understand the remaining parallel performance gaps
  - Bring all parallel kernels in line with reference versions
  - Avoid need for minGranularity setting

- **Optimize the serial kernels that are still lagging**
  - IF_QUAD, HYDRO_2D still need some effort

- **Explore more elegant Chapel loop expressions**
  - Make further use of whole-array operations, array slicing, etc.
Computer Language Benchmark Game (CLBG)
CLBG: Website supporting cross-language comparisons

- based on 13 simple serial/shared-memory computations:
  - binary-trees: memory management stressor
  - chameneos-redux: tasking coordination via a shared resource
  - fannkuch-redux: compute permutations on a small array
  - fasta, k-nucleotide, regex-dna, reverse-complement: string manipulation
  - mandelbrot: compute the Mandelbrot set
  - meteor: solve a puzzle (program startup is the bottleneck for Chapel)
  - n-body: simulate the solar system’s largest bodies (wants vectorization)
  - pidigits: compute pi (wants GMP or equivalent)
  - spectral-norm: compute matrix-vector operations
  - thread-ring: pass token between tasks as quickly as possible
- must follow prescribed algorithm (except meteor, which is free-form)
- website’s summaries don’t include three benchmarks:
  - chameneos-redux, thread-ring, meteor
  - the first two are parallel, which makes them of interest to us
  - program startup time is our bottleneck for the third, so we include it as well
Can sort results by execution time, code size, memory or CPU use:

<table>
<thead>
<tr>
<th>source</th>
<th>secs</th>
<th>KB</th>
<th>gz</th>
<th>cpu</th>
<th>cpu load</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0 C++ g++ #3</td>
<td>9.30</td>
<td>1,712</td>
<td>1763</td>
<td>9.29</td>
<td>100% 1% 1% 0%</td>
</tr>
<tr>
<td>1.0 C++ g++ #8</td>
<td>9.37</td>
<td>1,152</td>
<td>1544</td>
<td>9.36</td>
<td>1% 0% 100% 1%</td>
</tr>
<tr>
<td>1.0 C gcc #4</td>
<td>9.56</td>
<td>1,000</td>
<td>1490</td>
<td>9.56</td>
<td>1% 100% 1% 1%</td>
</tr>
<tr>
<td>1.0 C++ g++ #7</td>
<td>9.65</td>
<td>940</td>
<td>1545</td>
<td>9.64</td>
<td>100% 1% 1% 1%</td>
</tr>
<tr>
<td>1.1 Fortran Intel #5</td>
<td>9.79</td>
<td>516</td>
<td>1659</td>
<td>9.78</td>
<td>1% 0% 1% 100%</td>
</tr>
<tr>
<td>1.2 Ada 2005 GNAT #2</td>
<td>10.99</td>
<td>1,952</td>
<td>2604</td>
<td>10.99</td>
<td>0% 1% 100% 1%</td>
</tr>
<tr>
<td>1.3 C++ g++ #5</td>
<td>11.76</td>
<td>1,728</td>
<td>1749</td>
<td>11.75</td>
<td>1% 100% 1% 0%</td>
</tr>
<tr>
<td>1.9 Ada 2005 GNAT #5</td>
<td>18.00</td>
<td>2,028</td>
<td>2436</td>
<td>18.00</td>
<td>0% 0% 100% 1%</td>
</tr>
<tr>
<td>2.1 C++ g++ #6</td>
<td>19.20</td>
<td>1,096</td>
<td>1668</td>
<td>19.19</td>
<td>100% 1% 0% 0%</td>
</tr>
<tr>
<td>2.1 C++ g++</td>
<td>19.37</td>
<td>1,056</td>
<td>1659</td>
<td>19.36</td>
<td>1% 100% 0% 0%</td>
</tr>
<tr>
<td>2.1 Fortran Intel #2</td>
<td>19.84</td>
<td>508</td>
<td>1496</td>
<td>19.83</td>
<td>1% 0% 1% 100%</td>
</tr>
<tr>
<td>2.1 Fortran Intel</td>
<td>19.96</td>
<td>512</td>
<td>1389</td>
<td>19.95</td>
<td>0% 1% 0% 100%</td>
</tr>
<tr>
<td>2.2 C++ g++ #4</td>
<td>20.20</td>
<td>684</td>
<td>1428</td>
<td>20.19</td>
<td>0% 0% 1% 100%</td>
</tr>
<tr>
<td>2.3 C gcc #3</td>
<td>20.97</td>
<td>952</td>
<td>1208</td>
<td>20.96</td>
<td>1% 1% 1% 100%</td>
</tr>
</tbody>
</table>

gz == code size metric strip comments and extra whitespace, then gzip
Can also compare languages pair-wise:

<table>
<thead>
<tr>
<th>The Computer Language Benchmarks Game</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapel programs versus Swift</td>
</tr>
<tr>
<td>all other Chapel programs &amp; measurements</td>
</tr>
<tr>
<td>by benchmark task performance</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>regex-dna</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>source</td>
<td>secs</td>
<td>KB</td>
<td>gz</td>
<td>cpu</td>
<td>cpu load</td>
</tr>
<tr>
<td>Chapel</td>
<td>9.35</td>
<td>1,787,668</td>
<td>468</td>
<td>18.46</td>
<td>100% 15% 15% 69%</td>
</tr>
<tr>
<td>Swift</td>
<td>103.03</td>
<td>270,796</td>
<td>712</td>
<td>102.95</td>
<td>1% 1% 0% 100%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pidigits</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>source</td>
<td>secs</td>
<td>KB</td>
<td>gz</td>
<td>cpu</td>
<td>cpu load</td>
</tr>
<tr>
<td>Chapel</td>
<td>1.60</td>
<td>22,088</td>
<td>501</td>
<td>1.60</td>
<td>99% 4% 2% 1%</td>
</tr>
<tr>
<td>Swift</td>
<td>5.02</td>
<td>7,524</td>
<td>1017</td>
<td>5.01</td>
<td>100% 1% 1% 0%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n-body</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>source</td>
<td>secs</td>
<td>KB</td>
<td>gz</td>
<td>cpu</td>
<td>cpu load</td>
</tr>
<tr>
<td>Chapel</td>
<td>21.55</td>
<td>20,688</td>
<td>962</td>
<td>21.55</td>
<td>100% 0% 1% 0%</td>
</tr>
<tr>
<td>Swift</td>
<td>23.66</td>
<td>4,388</td>
<td>1253</td>
<td>23.65</td>
<td>100% 0% 0% 1%</td>
</tr>
</tbody>
</table>
site has a sound philosophy about too-easy answers

We want easy answers, but easy answers are often incomplete or wrong. You and I know, there's more we should understand:

- stories
- details
- fast?
- conclusions

yet, most readers probably still jump to conclusions
- execution time dominates default/only views of results
- it’s human nature

we’re interested in elegance as well as performance
- elegance is obviously in the eye of the beholder
  - we compare source codes manually
  - but then use CLBG’s code size metric as a quantitative stand-in
- want to be able to compare both axes simultaneously
- to that end, we used scatter plots to compare implementations
Background:

- We ported these to evaluate Chapel serial/tasking performance
- Many “top” entries are more heroic than typical programmers would write
- We strived for implementations that balance elegance with speed
- FAQ isn’t particularly encouraging of adding new languages:

  Why don't you include language X?
  ... my favorite language implementation?
  ... Microsoft® Windows®?

  Because I know it will take more time than I choose. Been there; done that.

  Measurements of “proggit popular” language implementations like Crystal and Nim and Julia will attract attention and be the basis of yet another successful website (unlike more Fortran or Ada or Pascal or Lisp). So make those repeated measurements at multiple workloads, and publish them and promote them.

  If you’re interested in something not shown on the benchmarks game website then please take the program source code and the measurement scripts and publish your own measurements.
CLBG: This Effort

Feb 2016: Inquired about submitting a Chapel entry
Apr 2016: Got a positive response
May 2016: Submitted first program
Sep 2016: Submitted final program
Listed on the front page:
Oct 2017: Upgraded to 1.14

### 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?

<table>
<thead>
<tr>
<th>Ada</th>
<th>C</th>
<th>Chapel</th>
<th>Clojure</th>
<th>C#</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dart</td>
<td>Erlang</td>
<td>F#</td>
<td>Fortran</td>
<td>Go</td>
<td>Hack</td>
</tr>
<tr>
<td>Haskell</td>
<td>Java</td>
<td>JavaScript</td>
<td>Lisp</td>
<td>Lua</td>
<td></td>
</tr>
<tr>
<td>OCaml</td>
<td>Pascal</td>
<td>Perl</td>
<td>PHP</td>
<td>Python</td>
<td></td>
</tr>
<tr>
<td>Racket</td>
<td>Ruby</td>
<td>JRuby</td>
<td>Rust</td>
<td>Scala</td>
<td></td>
</tr>
<tr>
<td>Smalltalk</td>
<td>Swift</td>
<td>TypeScript</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CLBG: Improvements due to 1.14

1.14 improved many benchmarks with no code changes:

- **thread-ring**: benefitted from qthread native sync variables
  - climbed ~16 slots ⇒ now 5\textsuperscript{th} fastest after Haskell, Go, F\#, Scala
  - 1\textsuperscript{st} most compact code followed by Ruby, Racket, Erlang, Ocaml, Python

- **fannkuch-redux**: benefitted from optimized array accesses
  - climbed from ~#22 to #6 in performance
  - ~1.5–2x more compact than most other top entries

- **chameneos-redux**: benefitted from tasking improvements
  - climbed from ~#11 to #8 in terms of performance

- **binary-trees**: benefitted from jemalloc improvements
  - climbed ~2 performance slots as a result
  - still ~5x off from top entries which use explicit memory pools

- **n-body**: saw marginal improvements, but climbed ~17 slots

- **regex-dna, revcomp**: saw marginal improvements, climbed ~3 slots

- **meteor**: saw marginal improvements, climbed ~1 slot

- **fasta**: saw marginal improvements, no change in rank
CLBG: Improvements due to 1.14

1.14 enabled code improvements for other benchmarks:

- **pidigits**: created versions that use the ‘bigint’ type
  - **pidigits**: uses operators everywhere
  - **pidigits-fast**: uses methods to avoid assigning returned records
- **knucleotide**: needed bug fix due to buggy auto-‘use’ of ’Sort’ in 1.13
  - updated to new ‘Sort’ interfaces while here
  - also saw performance improvements from optimized array accesses
- **binary-trees**: created an initializer-based implementation
- **mandelbrot**: used complex values and the dynamic domain iterator
- **fasta**: removed a downcast on ascii(); simplified I/O due to a bug fix
- **meteor**: made use of enum.size
- **revcomp**: removed downcasts on ascii()

To date, have only submitted two of the above cases

- trying to avoid maintenance fatigue
- latest versions available in examples/benchmarks/shootout on GitHub
CLBG: Status

Chapel entry highlights (as of Oct 17th):

- **performance rankings:**
  - **top entries:** pidigits
  - **top-5 entries:** meteor-contest, thread-ring
  - **top-10 entries:** fannkuch-redux, chameneos-redux
  - **top-20 entries:** n-body, spectral-norm, binary-trees

- **code compactness rankings:**
  - **top entries:** n-body, thread-ring
  - **top-5 entries:** spectral-norm, pidigits
  - **top-20 entries:** mandelbrot, regex-dna, chameneos-redux, meteor
CLBG Scatter Plots
CLBG Scatter Plots

● **Made scatter plots to compare performance and code size**
  ● created these to help us understand where we’re falling short
    ● e.g., helps us identify performance outliers
  ● compared with the languages of highest interest to our team:
    ● **traditional**: C, C++, Fortran, Java
    ● **productive**: Python
    ● **modern**: Scala, Go, Rust, Swift
  ● each program is scaled by the fastest/smallest entries for that axis
    ● In *any* language -- not restricted to the subset we’re focusing on here
    ● e.g., binary trees is scaled by C’s time (fastest) and Ruby’s size (smallest)
  ● data is from the CLBG repository on Oct 18th

● **Notes:**
  ● these only characterize the submitted programs
    ● i.e., better versions could potentially be written in each language
    ● however, this is the data we have to work with
  ● not all languages have entries for all programs
CLBG Scatter Plots: Chapel Programs
Pairwise Language Comparison Graphs

The first series of graphs compares languages pairwise

- For each language, we plot...
  ...the fastest version of each benchmark as a circle
  ...the smallest version of each benchmark as a square
  ...the mean of each set of benchmarks as a larger square/circle
- We also plot an oval at 1σ (a standard deviation away from the mean)
  - this provides an overall “profile” for the language’s fastest/smallest entries
- The axis scales are fixed across the graphs
  - in a few cases we zoom out in the subsequent slide to display outliers
CLBG Scatter Plots: Chapel vs. C
CLBG Scatter Plots: Chapel vs. C++

mandelbrot

thread-ring

binary-trees

rev-comp
CLBG Scatter Plots: Chapel vs. Fortran
CLBG Scatter Plots: Chapel vs. Java
CLBG Scatter Plots: Chapel vs. Java (zoom)
CLBG Scatter Plots: Chapel vs. Python

The graph compares the relative execution time against the relative source size for Chapel and Python. The data points are color-coded to indicate different categories such as fastest, smallest, mean-fastest, and mean-smallest. The graph highlights the performance differences between the two languages.
CLBG Fastest Entries: Chapel vs. Python (zoom)
CLBG Scatter Plots: Chapel vs. Scala
CLBG Scatter Plots: Chapel vs. Go

- binary-trees
- chameneos-rex
- meteor
- k-nucleotide
- regex-dna
- rev-comp

relative execution time
relative source size

chapel-go

- chapel
- go
- fastest
- smallest
- mean-fastest
- mean-smallest
CLBG Scatter Plots: Chapel vs. Rust
CLBG Scatter Plots: Chapel vs. Rust (zoom)
CLBG Scatter Plots: Chapel vs. Swift

mandelbrot

pidigits

k-nucleotide

mandelbrot

Copyright 2016 Cray Inc.
CLBG Scatter Plots: Chapel vs. Swift (zoom)
Language Summary Plots

- The following two graphs plot the means for all languages:
  - first: across the set of fastest entries
  - then: across the set of most compact entries

- Note that the y-axis is logarithmic:
  - otherwise, Python’s inclusion flattens all data along the x-axis
CLBG Fastest Codes: Averages (log scale perf)
CLBG Smallest Codes: Averages (log scale perf)
Per-Benchmark Comparison Graphs

The next graphs compare benchmarks across languages

- For each language, we plot its fastest and most compact version
- We connect these versions with a line to help the eye associate them
  - note that this line doesn’t imply a bound or a constraint, just a visual link
- Note that the y-axis is logarithmic
Binary-Trees: Language Comparisons

A graph showing the relative execution time versus relative source size for various languages. The graph includes markers for different programming languages, such as `chapel`, `gcc`, `go`, `cpp`, `ifc`, `java`, `python3`, `rust`, `scala`, and `swift`. The log of the relative execution time is plotted on the y-axis, and the relative source size is plotted on the x-axis. The graph shows a general trend of decreasing execution time with increasing source size for all languages.
Chameneos-redux: Language Comparisons

![Graph showing language comparisons](image-url)
Fannkuch-redux: Language Comparisons
Fasta: Language Comparisons
K-Nucleotide: Language Comparisons
Mandelbrot: Language Comparisons

![Mandelbrot: Language Comparisons Diagram](image-url)
Meteor: Language Comparisons
N-Body: Language Comparisons
Pidigits: Language Comparisons
Regexdna: Language Comparisons
Reverse-Complement: Language Comparisons
Spectral-Norm: Language Comparisons
Thread-Ring: Language Comparisons

![Graph showing language comparisons](image-url)
CLBG Comparison Graphs
CLBG: Comparisons to C/C++

- The following graphs compare Chapel to C/C++ versions
  - an update to what we did for the 1.11 release notes
  - run using our team’s systems:
    - 2 x 12-core Intel Xeon
    - gcc/g++
  - reflects more recent HW/compiler than the official CLBG system
  - yet imperfect:
    - some C/C++ entries are tuned specifically for the CLBG system
    - others rely on libraries that are not installed on our system
CLBG: Comparisons to C/C++

- **pidigits, all versions**
  - Time (seconds): 1.14
  - Bar graph comparing different versions.

- **fannkuchredux, all versions**
  - Time (seconds): 40
  - Bar graph comparing different versions.

- **threading, all versions**
  - Time (seconds): 50
  - Bar graph comparing different versions.

- **regexdna, all versions**
  - Time (seconds): 14
  - Bar graph comparing different versions.
CLBG: Comparisons to C/C++

revcomp, all versions

Time (seconds)

mandelbrot, all versions

Time (seconds)

spectralnorm, all versions

Time (seconds)

binarytrees, all versions

Time (seconds)
CLBG: Comparisons to C/C++

knucleotide, all versions

Time (seconds)

<table>
<thead>
<tr>
<th>Time (seconds)</th>
<th>gpp 3</th>
<th>chpl 1</th>
<th>1.14</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td>0.5</td>
<td>1</td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td>1.5</td>
<td>1</td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td>2.5</td>
<td>1</td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td>3.5</td>
<td>1</td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td>4.5</td>
<td>1</td>
<td>1.5</td>
<td>1.5</td>
</tr>
</tbody>
</table>

nbody, all versions

Time (seconds)

<table>
<thead>
<tr>
<th>Time (seconds)</th>
<th>gpp 8</th>
<th>gcc 7</th>
<th>gpp 3</th>
<th>gcc 6</th>
<th>gpp 7</th>
<th>gcc 4</th>
<th>gcc 1</th>
<th>chpl 1</th>
<th>gpp 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0.5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1.5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2.5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3.5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4.5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

meteor, all versions

Time (seconds)

<table>
<thead>
<tr>
<th>Time (seconds)</th>
<th>gpp 6</th>
<th>gcc 7</th>
<th>gpp 5</th>
<th>gcc 6</th>
<th>gpp 4</th>
<th>gcc 5</th>
<th>gcc 1</th>
<th>chpl 1</th>
<th>gcc 2</th>
<th>chpl 2</th>
<th>gcc 3</th>
<th>gcc 4</th>
<th>gcc 5</th>
<th>gcc 6</th>
<th>gcc 7</th>
<th>gcc 8</th>
<th>gcc 9</th>
<th>gcc 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0.05</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0.1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0.15</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0.2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0.25</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0.3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0.35</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0.4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0.45</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

fasta, all versions

Time (seconds)

<table>
<thead>
<tr>
<th>Time (seconds)</th>
<th>gcc 5</th>
<th>gcc 7</th>
<th>gcc 6</th>
<th>gcc 4</th>
<th>gcc 8</th>
<th>gcc 9</th>
<th>gcc 1</th>
<th>chpl 1</th>
<th>gpp 3</th>
<th>gcc 2</th>
<th>gcc 10</th>
<th>gcc 11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1.5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2.5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3.5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4.5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5.5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
CLBG: Future Work

- Continue improving Chapel and our entries:
  - **fasta:** parallelize and optimize our current version
  - **string benchmarks:** improve string operations and performance
  - **pidigits:** optimize “assign returned record” idioms
  - **meteor:** reduce startup time in `qthreads/hwloc`
  - **n-body:** enable vectorization
  - **k-nucleotide:** improve associative domain performance and features
  - **mandelbrot:** consider adding ‘unroll’ keyword to for loops
  - continue to study outliers and work on improving them

- Publish studies that dive beyond the superficial
  - e.g., show heroism of top versions, compare with more typical ones

- Consider submitting more heroic Chapel versions

- Don’t lose sight of multi-locale performance work
  - encourage HPC community to establish a CLBG equivalent
Other Benchmark Improvements
Other Benchmark Improvements

- Switched ISx to use the low-level PCG interface
  - results in identical data sets as the reference version
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.