Benchmark Improvements

Chapel Team, Cray Inc.
Chapel version 1.15
April 6, 2017
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 Benchmark Improvements
- ISx Benchmark Improvements
- Computer Language Benchmarks Game (CLBG)
LCALS Benchmark Improvements
LCALS Improvements: Background

- Livermore Compiler Analysis Loop Suite
- Chapel port first released with version 1.13.0
- Version 1.14.0 saw significant performance improvements
  - Serial variant matching reference version
  - Parallel variant still had room for improvement
LCALS Improvements: This Effort

- **Kernel-by-kernel improvements**
  - Made nested loops into loops over 2D domains
  - Improved style of array initializations
  - Changed a 64-bit based range to 32-bit to match the reference version

- **Changed some compiler/execution default options**
  - dropped --data-par-min-granularity=1000
  - added --no-ieee-float
    - both option changes done for a more equal comparison to the reference

- **Added SPMD variant of parallel kernels**
  - Start all tasks
  - Block the kernels up across tasks
  - Execute blocks serially within each task
  - Barrier at the end
LCALS Improvements: Task Startup Times

● The amount of work in each kernel is small
  ● Split up across ~24 tasks the amount of work per task is very small
  ● Kernels are repeated a large number of times
  ● Leads to millions of tasks doing very little work each

● Task startup times are a large factor in total execution time

● Optimization added to the Chapel tasking interface
  ● Enable threads looking for work to more quickly pick up new tasks
  ● Dramatically improves task startup times
LCALS Improvements: Impact

- Improved task startup time increased performance overall

Improvement for short loop size was masked in nightly testing by another change, but is clear in release-over-release testing.
Long problem size

Parallel Chapel vs g++/OMP

Normalized time – parallel reference is 1.0

chpl --fast
--no-ieee-float

g++ -Ofast -fopenmp
Long problem size

Parallel Chapel vs g++/OMP

Normalized time – parallel reference is 1.0

Normalized Time

0 1 2 3 4 5 6 7 8 9

g++ OMP Chapel parallel

chpl --fast
--no-ieee-float

++ -Ofast -fopenmp
LCALS Improvements: Status and Next Steps

Status:
● The parallel kernels have improved dramatically since version 1.14.0
● A few still lag slightly behind the C+OpenMP reference versions
● But most are effectively matching reference versions

Next Steps:
● There is still some startup overhead to work through
  ● Some ideas:
    ● start multiple tasks simultaneously
    ● improve task counting/sync mechanism
ISx Benchmark Improvements
ISx: Background

- **Scalable Integer Sort benchmark**
  - Developed at Intel, published at PGAS 2015
  - SPMD-style computation with barriers
  - Punctuated by all-to-all bucket exchange pattern
  - References implemented in SHMEM and MPI

- **Chapel implementation introduced in 1.13 release**
  - Motivation: a common distributed pattern

- **Initial results showed roughly 10x worse performance**
ISx: Background

- 64 nodes on Cray XC

![Graph showing ISx weakISO Total Time vs Nodes]

**ISx weakISO Total Time**

- Time (seconds)
- Nodes
- Chapel 1.14
- SHMEM
ISx: Background

- Developed an inelegant optimized version for study
  - Uses language internals
  - Also called the ‘heroic’ version, referring to user effort
  - Competitive with SHMEM reference

- Found several areas for improvement:
  - Reference counted arrays
  - Sliced assignment between arrays
  - Wide-pointer overhead for serial loops
ISx: This Effort - Reference Counting

- Reference counting arrays was costly at scale
  - Lots of on-statements and atomic operations

- Array memory management work resolved this problem
  - Eliminated need for reference counting entirely

![ISx weakISO Total Time Graph]

- **Graph Description**
  - X-axis: Nodes
  - Y-axis: Time (seconds)
  - Red line: no reference counting
  - Green line: SHMEM

```c
// Example code snippet
#include <stdlib.h>
int main()
{
    // Data structures
    int *data;

    // Allocate memory
    data = malloc(sizeof(int) * N);

    // Reference counting operations
    // ...

    // Finalize
    free(data);
    return 0;
}
```
The following pattern was quite slow at scale:
- Core operation in all-to-all exchange step
  ```
  // 'Dest' is a remote 1D array, 'Src' is local
  Dest[1..10] = Src[1..10];
  ```

Slicing the remote array involved an on-statement
- Used to initialize array metadata
- In all-to-all step, this might mean numCores^2 on-statements
- Problem worsens as we scale to more locales

On-statement also used for slicing Src’s domain
- Src was declared over a remote const global domain
Fix #1: ArrayViews

- Allows for much cheaper creation of a remote array slice
- Avoids the on-statement entirely
- Still able to bulk-transfer the array data (single PUT/GET)

Fix #2: Declare Src over local domain

- Slicing 'Src' now an entirely local operation
- Before:
  ```javascript
  var Src : [NumTasksSpace] keyType;
  ```
- After:
  ```javascript
  var Src : [0..#numTasks] keyType;
  ```
ISx: Impact – Sliced Assignment Overhead

- **Result:** 2x performance improvement
  - Also less noise
  - ugni-qthreads, 16 nodes on Cray XC
ISx: This Effort – Wide-pointer overhead

- Compiler inserts wide-pointers for potentially-remote data
  - At its simplest, a struct containing a locale ID and pointer
  - Sometimes thwart backend compiler optimizations

- Can introduce overhead for array accesses
  - Problematic for subsections like this in ISx:
    ```
    for key in myKeys do
        sizes[key/width] += 1;
    ```

- 1.14 optimized some cases, regressed in 1.15 dev cycle

- For 1.15 a similar approach is used, but only for arrays
  - Disabled with “--[no-]infer-local-fields” compiler flag
ISx: Impact – Wide-pointer overhead

- Small, but useful, improvements in serial loops

- Work remains to match timed subsections of reference
  - Roughly 20% worse in some cases
  - Note: exchange step, not impacted here, dominates at scale
ISx: Performance Summary

- Gathered on Cray XC with default problem size

ISx weakISO Total Time

Time (seconds)

Nodes

Chapel 1.14
Chapel 1.15
MPI
SHMEM
ISx: Performance Summary

- Gathered on Cray XC with default problem size
  - Let’s zoom in...

ISx weakISO Total Time

- Time (seconds)
- Nodes

- Chapel 1.15
- MPI
- SHMEM
ISx: Performance Summary

- **1.15 with 1.14’s release version of source code**
  - 1.14 release version did not declare arrays over local domains

### ISx weakISO Total Time

![Graph showing the comparison between ISx weakISO Total Time for 1.15 with 1.14’s release version and Chapel 1.15 across different node counts.](image)
ISx: Status and Next Steps

● **Status:** Big improvements with minor user-code changes

● **Next Steps:**
  ● Address known performance issues
    ● Under-counting tasks bug leading to oversubscription
      ● Likely to impact other benchmarks as well
    ● Eliminate additional wide-pointer overhead
    ● Investigate performance at larger scales
    ● Compiler should optimize references to remote const domains
  
  ● Continue to investigate performance results
    ● Why does Chapel beat SHMEM and MPI at 64 nodes?
    ● Why does Chapel scale poorly from 1-4 nodes, then recover?
Computer Language Benchmarks Game (CLBG)
The Computer Language Benchmarks Game

64-bit quad core data set
Will your toy benchmark program be faster if you write it in a different programming language? It depends how you write it!

Which programs are fast?
Which are succinct? Which are efficient?

| Ada | C   | Chapel | Clojure | C#  | C++ |
| Dart    | Erlang | F#   | Fortran | Go  | Hack |
| Haskell | Java  | JavaScript | Lisp | Lua |
| OCaml  | Pascal | Perl  | PHP    | Python |
| Racket | Ruby  | JRuby | Rust   | Scala |
| Smalltalk | Swift | TypeScript |

Website that supports cross-language game / comparisons
- 13 toy benchmark programs
- exercises key features like:
  - memory management
  - tasking and synchronization
  - vectorization
  - big integers
  - strings and regular expressions
- specific approach prescribed

Take results w/ grain of salt
- other programs may be different
- not to mention other programmers
- specific to this platform / OS / ...

That said, it’s one of the only games in town...
Chapel’s approach to CLBG:
- want to know how we compare
- strive for entries that are elegant rather than heroic
  - e.g., “Want to learn how program x works? Check out the Chapel version.”

### 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></th>
<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Erlang</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F#</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fortran</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Go</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hack</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Haskell</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JavaScript</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lisp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lua</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OCaml</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pascal</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Perl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PHP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Python</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Racket</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ruby</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JRuby</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rust</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scala</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Smalltalk</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swift</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TypeScript</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Can sort results by execution time, code size, memory or CPU use:

### The Computer Language Benchmarks Game

**chameneos-redux**

description

<table>
<thead>
<tr>
<th>program source code, command-line and measurements</th>
</tr>
</thead>
<tbody>
<tr>
<td>× source</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>1.0</td>
</tr>
<tr>
<td>1.2</td>
</tr>
<tr>
<td>1.7</td>
</tr>
<tr>
<td>2.3</td>
</tr>
<tr>
<td>3.3</td>
</tr>
<tr>
<td>5.6</td>
</tr>
<tr>
<td>6.8</td>
</tr>
<tr>
<td>8.0</td>
</tr>
<tr>
<td>8.5</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>13</td>
</tr>
</tbody>
</table>

### The Computer Language Benchmarks Game

**chameneos-redux**

description

<table>
<thead>
<tr>
<th>program source code, command-line and measurements</th>
</tr>
</thead>
<tbody>
<tr>
<td>× source</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>1.0</td>
</tr>
<tr>
<td>1.0</td>
</tr>
<tr>
<td>1.1</td>
</tr>
<tr>
<td>1.1</td>
</tr>
<tr>
<td>1.1</td>
</tr>
<tr>
<td>1.2</td>
</tr>
<tr>
<td>1.3</td>
</tr>
<tr>
<td>1.3</td>
</tr>
<tr>
<td>1.3</td>
</tr>
<tr>
<td>1.3</td>
</tr>
<tr>
<td>1.3</td>
</tr>
<tr>
<td>1.6</td>
</tr>
<tr>
<td>1.6</td>
</tr>
<tr>
<td>1.6</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>Task</th>
<th>Chapel (secs)</th>
<th>Go (secs)</th>
<th>Chapel (mem)</th>
<th>Go (mem)</th>
<th>Chapel (g)</th>
<th>Go (g)</th>
<th>Chapel (cpu)</th>
<th>Go (cpu)</th>
<th>Chapel (cpu load)</th>
<th>Go (cpu load)</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>regex-redux</em></td>
<td>10.02</td>
<td>29.51</td>
<td>1,022,052</td>
<td>352,804</td>
<td>477</td>
<td>798</td>
<td>19.68</td>
<td>61.51</td>
<td>99% 72% 14% 12%</td>
<td>77% 49% 43% 40%</td>
</tr>
<tr>
<td><em>binary-trees</em></td>
<td>14.32</td>
<td>34.77</td>
<td>324,660</td>
<td>269,068</td>
<td>484</td>
<td>654</td>
<td>44.15</td>
<td>132.04</td>
<td>100% 58% 78% 75%</td>
<td>95% 97% 95% 95%</td>
</tr>
<tr>
<td><em>fannkuch-redux</em></td>
<td>11.38</td>
<td>15.81</td>
<td>46,056</td>
<td>1,372</td>
<td>728</td>
<td>900</td>
<td>45.18</td>
<td>62.92</td>
<td>100% 99% 99% 100%</td>
<td>100% 100% 99% 99%</td>
</tr>
</tbody>
</table>
Can also browse program source code (but this requires actual thought):

```plaintext
proc main() {
    printColorEquations();
    const group1 = [i in 1..popSize1] new Chamaneos{i, ((i-1)%3):Color};
    const group2 = [i in 1..popSize2] new Chamaneos{i, colors10[i]};
    cobegin {
        holdMeetings(group1, n);
        holdMeetings(group2, n);
    }
    print(group1);
    print(group2);
    for c in group1 do delete c;
    for c in group2 do delete c;
}

// Print the results of getNewColor() for all color pairs.
//
// proc printColorEquations() {
//    for c1 in Color do
//        for c2 in Color do
//            writeln(c1, " + ", c2, " -> ", getNewColor(c1, c2));
//            writeln();
//}

// Hold meetings among the population by creating a shared meeting // place, and then creating per-chamaneos tasks to have meetings.
//
// proc holdMeetings(population, numMeetings) {
//    const place = new MeetingPlace(numMeetings);
//    coforall c in population do // create a task per chamaneos
c.haveMeetings(place, population);
//    delete place;
}

void get_affinity(int* is_smp, cpu_set_t* affinity1, cpu_set_t* affinity2) {
    cpu_set_t active_cpus;
    FILE* f;
    char char const* buf [2048];
    pos;
    int cpu_id;
    int physical_id;
    int core_id;
    int cpu_cores;
    int apic_id;
    size_t cpu_count;
    size_t i;
    char const* processor_str = "processor";
    size_t processor_str_len = strlen(processor_str);
    char const* physical_id_str = "physical_id";
    size_t physical_id_str_len = strlen(physical_id_str);
    char const* core_id_str = "core_id";
    size_t core_id_str_len = strlen(core_id_str);
    char const* cpu_cores_str = "cpu_cores";
    size_t cpu_cores_str_len = strlen(cpu_cores_str);
    CPU_ZERO(&active_cpus);
    sched_getaffinity(0, sizeof(active_cpus), &active_cpus);
    cpu_count = 0;
    for (i = 0; i <= CPU_SETSIZE; i += 1) {
        if (CPU_ISSET(i, &active_cpus)) {
            cpu_count += 1;
        }
    }
    if (cpu_count == 1) {
        is_smp[0] = 0;
        return;
    }
    is_smp[0] = 1;
    CPU_ZERO(affinity1);
}
```

excerpt from 1210.gz Chapel #2 entry

excerpt from 2863.gz C gcc #5 entry
What’s new with the CLBG?

- **Two programs changed their official definitions:**
  - **binary-trees:**
    - improved checksum to avoid false positives at 1/2, 1/4, 1/8 the memory
    - eliminated per-node data field
    - changed what trees are allocated and freed, slightly
    - increased the problem size
  - **regex:**
    - changed the regular expression used
    - renamed the test to regex-redux
    - several versions are not currently passing due to these changes
      - our current standings may be due in part to this
What’s new with the Chapel CLBG entries?

- We’ve submitted some new versions:
  - **binary-trees**: used an initializer rather than a factory type method
What’s new with the Chapel CLBG entries?

- We’ve submitted some new versions:
  - **binary-trees**: used an initializer rather than a factory type method
  - **chameneos-redux**: increased parallelism and tuned a spin-wait

---

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

- **original version**
- **new version**
What’s new with the Chapel CLBG entries?

● We’ve submitted some new versions:
  - binary-trees: used an initializer rather than a factory type method
  - chameneos-redux: increased parallelism and tuned a spin-wait
  - fasta: implemented a parallel version and tuned for clarity and speed
  - also, changed some ‘var’ declarations due to const-checking improvements
What’s new with the Chapel CLBG entries?

- We’ve submitted some new versions:
  - **binary-trees**: used an initializer rather than a factory type method
  - **chameneos-redux**: increased parallelism and tuned a spin-wait
  - **fasta**: implemented a parallel version and tuned for clarity and speed
    - also, changed some ‘var’ declarations due to const-checking improvements
  - **mandelbrot**: accelerated by hoisting values and using tuples of values

![Submitted Mandelbrot Shootout Benchmark](image-url)
What’s new with the Chapel CLBG entries?

- **We’ve submitted some new versions:**
  - **binary-trees:** used an initializer rather than a factory type method
  - **chameneos-redux:** increased parallelism and tuned a spin-wait
  - **fasta:** implemented a parallel version and tuned for clarity and speed
    - also, changed some ‘var’ declarations due to const-checking improvements
  - **mandelbrot:** accelerated by hoisting values and using tuples of values
  - **meteor-fast:** fixed a race condition caused by array memory changes
    - textbook example of an array being used by a ’begin’ task
  - **pidigits:** submitted a version that uses ‘bigint’s
    - currently the #1 fastest version, and also quite elegant

- **Note that some of these changes followed the 1.15 release**
  - As such, not all are found in examples/benchmarks/shootout/ for 1.15
CLBG: Chapel Standings as of Oct 17th

- **8 / 13 programs in top-20 fastest:**
  - one #1 fastest: pidigits
  - 2 others in the top-5 fastest: meteor-contest thread-ring
  - 2 others in the top-10 fastest: chameneos-redux fannkuch-redux
  - 3 others in the top-20 fastest: binary-trees n-body spectral-norm

- **8 / 13 programs in top-20 smallest:**
  - two #1 smallest: n-body thread-ring
  - 2 others in the top-5 smallest: pidigits spectral-norm
  - 4 others in the top-20 smallest: chameneos-redux mandelbrot meteor-contest regex-dna
CLBG: Chapel Standings as of Apr 20\textsuperscript{th}

- **12 /13 programs in top-20 fastest:**
  - one #1 fastest:
    \textit{pidigits}
  - 3 others in the top-5 fastest:
    \textit{chameneos-redux}
    \textit{meteor-contest}
    \textit{thread-ring}
  - 3 others in the top-10 fastest:
    \textit{fannkuch-redux}
    \textit{fasta}
    \textit{mandelbrot}
  - 5 others in the top-20 fastest:
    \textit{binary-trees}
    \textit{k-nucleotide}
    \textit{n-body}
    \textit{regex-redux}
    \textit{spectral-norm}

- **8 / 13 programs in top-20 smallest:**
  - two #1 smallest:
    \textit{n-body}
    \textit{thread-ring}
  - 2 others in the top-5 smallest:
    \textit{pidigits}
    \textit{spectral-norm}
  - 1 other in the top-10 smallest:
    \textit{regex-redux}
  - 3 others in the top-20 smallest:
    \textit{chameneos-redux}
    \textit{mandelbrot}
    \textit{meteor-contest}
CLBG: Website’s fast-faster-fastest graph

Site summary: relative performance (sorted by geometric mean)

How many times slower?

benchmarks game

20 Apr 2017 u64q
CLBG: Website

- site has voiced good 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 (or only) views of results
  - it’s simply 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
Chapel entries (noting outliers)
Chapel vs. 9 other languages
Chapel vs. 9 other languages (zoomed out)
Chapel vs. C
Chapel vs. C (zoomed out)
Chapel vs. C++
Chapel vs. C++ (zoomed out)
Chapel vs. Fortran
Chapel vs. Fortran (zoomed out)
Chapel vs. Go

![Chapel vs. Go Diagram]

---

Copyright 2017 Cray Inc.
Chapel vs. Go (zoomed out)
Chapel vs. Rust
Chapel vs. Rust (zoomed out)
Chapel vs. Swift
Chapel vs. Swift (zoomed out)
Chapel vs. Java
Chapel vs. Java (zoomed out)
Chapel vs. Scala
Chapel vs. Scala (zoomed out)
Chapel vs. Python
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.