Standard Library and Interoperability 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

- FileSystem and Path Modules
- BitOps: Parity & Rotate
- FFTW Module
- Specifying Library Dependences
- pyChapel: Python-Chapel Interoperability
- Other Standard Library Improvements
- Other Interoperability Improvements
- Standard Library/Interoperability Priorities and Next Steps

FileSystem and Path Modules

FileSystem/Path Module

Background:

- File functionality historically limited to open/close/channel creation
- C routines accessible, but native solution preferred
- Last release:
  - Added 6 Chapel routines to existing modules
  - Defined strategy for remaining routines and sought user feedback
  - Added placeholder “Filerator” module for prototype file/directory iterators

FileSystem/Path Module

This Effort:

- Created two new modules: FileSystem and Path
  - **FileSystem:** file/directory properties, contents, program state routines, etc.
  - **Path:** Manipulation of strings-as-paths
- Added 19 routines, revamped 2
  - A few contributed by Francisco Souza (community developer)
- Added parallel versions of Filerators and folded into FileSystem

Status:

- **FileSystem:** nearly finished
- **Path:** started, but more remains
- Documented online:
  - http://chapel.cray.com/docs/latest/modules/standard/FileSystem.html
  - http://chapel.cray.com/docs/latest/modules/standard/Path.html

FileSystem Module: Interface Updates

- **Clarified chdir() and cwd()**
  - **Before:**
    - chdir() and cwd() altered and accessed locale-specific information
    - No interface clues to this behavior
  - **Now:**
    - chdir() and cwd() → locale.chdir() and locale.cwd()
    - Interface directly links these routines to a locale
    - Applies naturally to remote locales, e.g.

      ```
      var x = locales[index].cwd()
      ```

    - Can also be applied to all locales through promotion:

      ```
      Locales.chdir("/tmp");
      ```

- **Added locale.umask():**
  - Alters the file creation mask within a locale

FileSystem Module: New Routines

- **Given a path argument, determine property from OS:**
  - **getGID():** Obtain group id
  - **getUID():** Obtain user id
  - **getMode():** Obtain permissions
  - **exists():** Does arg refer to a file or directory that exists?
  - **isDir():** Does arg refer to a directory?
  - **isFile():** Does arg refer to a file?
  - **isLink():** Does arg refer to a link?
  - **isMount():** Does arg refer to a mount point?
  - **sameFile():** Are these two string paths or files equivalent?

FileSystem Module: More New Routines

- **File/Directory Manipulations:**
  - **copy():** Copy file's contents, permissions, owner, access times, …
  - **copyFile():** Copy contents only
  - **copyTree():** Copy directory and children, using copy() for files
  - **symlink():** Create a symbolic link
  - **chmod():** Change permissions
  - **copyMode():** Copy permissions

FileSystem Module: Parallel Filerators

- **Moved Filerator iterators into FileSystem module**
- **Added parallel versions of iterators as appropriate**
  - **glob():** supports standalone and zipperable parallel iteration
    - easy to anticipate number of items, randomly access them, load balance
  - **walkdirs()/findfiles():** only support standalone parallel iteration
    - can't anticipate depth of directory tree a priori
    - load balancing is currently reasonably naive
  - **listdir():** supports serial iteration only
    - underlying C interface is serial in nature

Path Module: New Routines/Values

- **Path module:**
  - **[file.]realPath():** obtain canonical path from string or file obj
  - **curDir:** string representing current directory, e.g. “.”
  - **parentDir:** string representing parent directory, e.g. “..”
  - **pathSep:** string representing path separator, e.g., “/”

FileSystem/Path Module: Next Steps

- **Provide remaining FileSystem routines:**
  - moveDir()
  - rmTree()
  - other requested features from users
- **Further improvements to filerators:**
  - distributed memory versions?
  - improve dynamic load balancing of findfiles()/walkdirs()?
  - support dynamic filtering of findfiles()/walkdirs()?
  - respond to user feedback
- **Complete Path module once string library is complete**
- **Expand testing of new routines**
  - Basic testing present
  - Lacking more complex cases

BitOps: Parity & Rotate

BitOps: Parity & Rotate

Background: BitOps module supports bit-level operations

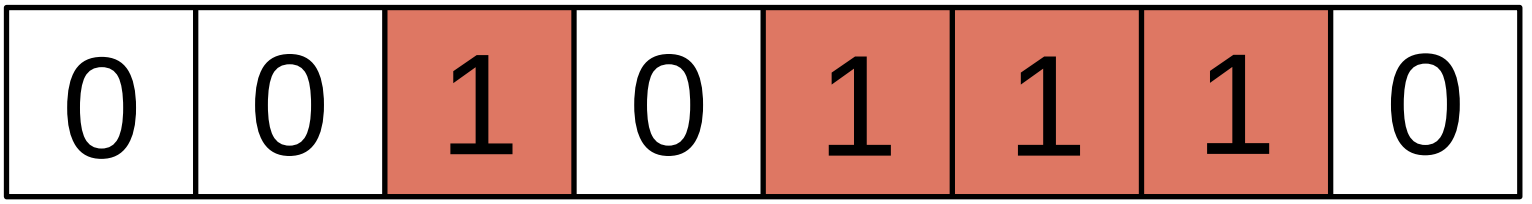
- Already supports operations like:
  - popcount
  - clz (count leading zeros)
  - …
- Implemented using C intrinsics when possible

This Effort: Add parity and rotate

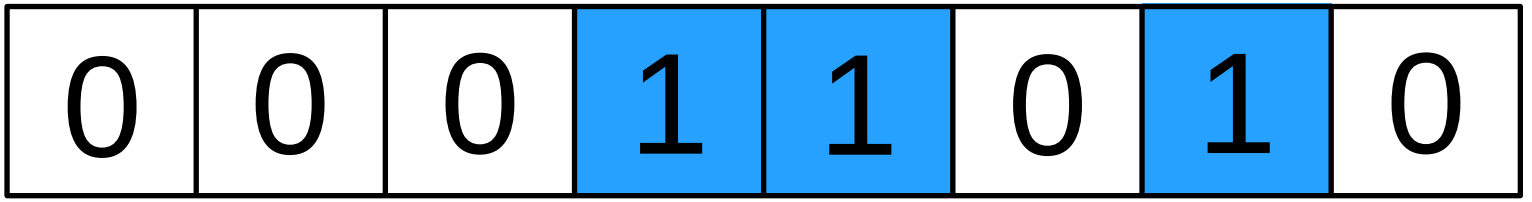
- **parity()**: Checks for an even or odd number of bits set
  - Parity uses intrinsics on most platforms
- **rotl()** & **rotr()**: Performs a bitwise rotation
  - Rotate left and right are generated as C idioms
  - Back-end compilers should pattern match and turn into a **ror** or **rol** on x86

BitOps: parity()

- **Checks for an even or odd number of bits set**
  - Returns 0 when even

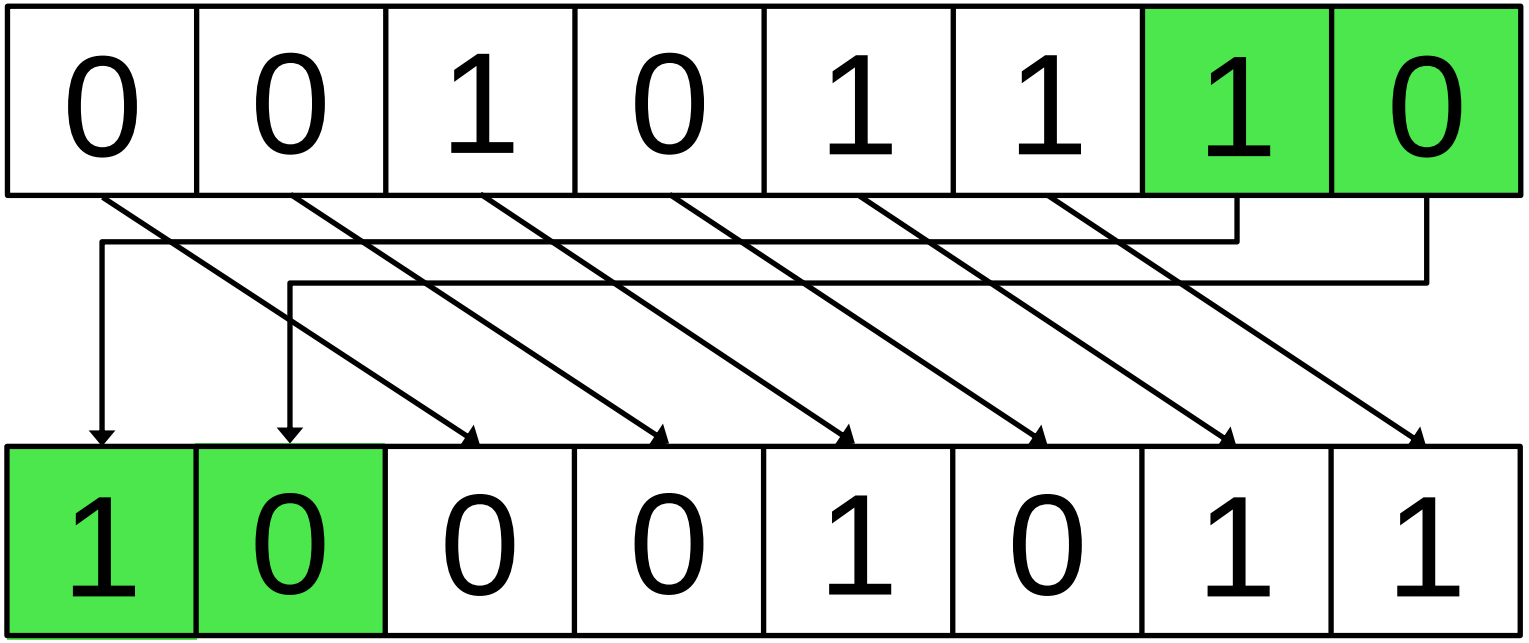


  - Returns 1 when odd



BitOps: rotl() & rotr()

- **Bitwise rotation**
  - Example: Rotate right by 2



BitOps: Next Steps

- **Most low hanging fruit is done**
- **Need to decide on level of support for other operations**
  - Many need to be implemented in assembly for performance
    - Do we want to do this?
    - For which platforms and compilers?

FFTW Module

FFTW: Background and This Effort

Background:

- Users are increasingly interested in numerical libraries out of the box
  - e.g., FFTW, BLAS, GSL, LAPACK, etc.
- As with other standard libraries, Chapel support has been thin
- Belief has been that domains/arrays should result in nicer interfaces

This Effort:

- Contributed by Nikhil Padmanabhan (Yale University professor)
- Add support for core FFTW routines
  - single- and multi-threaded implementations (FFTW vs. FFTW_MT modules)
  - in-place and out-of-place versions
  - at present, 64-bit versions only
- Documented online:
  - http://chapel.cray.com/docs/latest/modules/standard/FFTW.html
  - http://chapel.cray.com/docs/latest/modules/standard/FFTW_MT.html
- Requires user to have own FFTW installation
- Primer example available in release

FFTW: Status

Status:

- Supported routines:
  - plan_dft()
  - plan_dft_c2r() /plan_dft_r2c()
  - execute()
  - plan_with_nthreads()
  - destroy_plan() / cleanup()
- Hypothesis about cleaner interfaces holds up:

C:

```
fftw_plan_dft_2d(m, n, A, B, FFTW_FORWARD, FFTW_ESTIMATE);
```

or:

```
int sz[2] = {m, n};
fftw_plan(2, sz, A, B, FFTW_FORWARD, FFTW_ESTIMATE);
```

Chapel:

```
var A, B: [1..m, 1..n] complex;
plan_dft(A, B, FFTW_FORWARD, FFTW_ESTIMATE);   // rank-independent!
```

FFTW: Next Steps

Next Steps:

- add support for 32-bit versions
- add support for distributed-memory versions
  - an early example of Chapel-MPI interoperability
- consider other opportunities for additional cleanup/refactoring
  - replace C-oriented bit flags and consts with keyword+default arguments?

```
fftw_plan(A, B, dir=fftwdir.forward);    // defaults to "estimate"
```

or:

```
fftw_plan(A, B, forward=true);
```

  - make FFTW_MT a sub-module of FFTW?
- add other routines based on user needs/requests

Specifying Library Dependences

Specifying Library Dependences

Background:

- Wrapping libraries adds dependences on header and library files
- Traditionally, we've dealt with this by:
  - building the library into Chapel's Makefile system (e.g., RE2, GMP), or
  - requiring users to explicitly name those dependences on the command line

This Effort:

- Permit the library writer to specify these dependences in source code
  - currently, done by overloading the **`use`** keyword to take a string literal
- Support similar dependences as on the command-line
  - e.g.,

```
use "foo.h";
use "foo.c";
use "foo.o";
use "-lfoo";
```

Specifying Library Dependences

Status:

- Used by the FFTW and FFTW_MT modules:

  e.g., FFTW: `use "fftw3.h", "-lfftw3";`

- Note that search paths are not specified as Chapel code
  - files must be in standard paths
  - or specified using environment variables / command-line args

Next Steps:

- If the current form persists…
  - switch to a new keyword (e.g., **`requires`**) to avoid confusion
  - consider integrating with param resolution machinery
    - to support conditional dependences
    - to compute dependences/filenames using param strings
- Alternatively, support dependences through script-based mechanism
  - \+ Supports direct inspection of environment / file system
  - \+ Could help with search path issues (e.g., for multiple library versions)
  - − Requires multiple files (or additional source-based formatting)

pyChapel: Python-Chapel Interoperability

pyChapel: Background

Background:

- Python is well-loved for its usability …
  … but not so much for its performance
- Common solutions are to write libraries in C
  - e.g. NumPy
- But then someone in Python-land has to write in C…
  - They'd rather write productively
  - We'd prefer them to write Chapel

pyChapel: This Effort

This Effort:

- Prototype support for Python interoperability
  - via a Python module, **pyChapel**
  - developed by Simon Lund (University of Copenhagen)
- Use Python stub functions to interface to Chapel code
  - Can refer to Chapel code blocks or functions
  - Python stub describes argument and return types
- Three ways to wrap Chapel…

pyChapel: Inlined Chapel Approach

Approach 1: Inlined Chapel Code

- Chapel code lives in comment block of extern Python declaration

myfile.py:

```
@Chapel()
def inline_fn(x=float):
    """      // (start of Chapel code)
    var y = x*2;  // This variable can be created generically …
    writeln("Hello, world");
    return y;     // … and returned …
    """      // (end of Chapel code)
    return float # … but the return type must be fully declared
```

pyChapel: Sfile Approach

Approach 2: sfile ("source" file)

- Extern Python declaration names Chapel file with exported function(s)
- Requires specifying interface in both Chapel and Python

myfile.py:

```
@Chapel(sfile="somefile.chpl")
def addEm(x=int, y=int):
  return int

if __name__ == "__main__":
    print addEm(1, 2)  # prints 3
```

somefile.chpl:

```
export proc addEm(x:int, y:int):int {
  var ret = x+y;
  return ret;
}
// Other functions (exported or not)
…
```

pyChapel: Bfile Approach

Approach 3: bfile ("body" file)

- Extern Python declaration names file defining function body only
- Arguments are defined by the extern declaration, not the bfile
  - The bfile may reference these arguments and also return values

  (Note that bfiles are not valid standalone code when arguments/returns are used)

myfile.py:

```
@Chapel(bfile="somefile.chpl")
def addEm(x=int, y=int):
  return int

if __name__ == "__main__":
    print addEm(1, 2) # prints 3
```

somefile.chpl:

```
var ret = x+y;
// x, y defined in Python extern
return ret;
```

+ eliminates redundant procedure declarations
+ simplifies Chapel usage for new programmers
- limits Chapel code to a single routine per bfile

pyChapel: Type Declaration Requirements

- **Function declaration must be fully typed**
  - In Python:

Don't:

```
@Chapel(sfile="file.chpl")
def foo(x, y):
```

Do:

```
@Chapel(sfile="file.chpl")
def foo(x=int, y=int):
  return int
```

  - In Chapel:

Don't:

```
export proc foo(x, y)
```

Do:

```
export proc foo(x:int, y:int): int
```

  - Note: bfiles and inline functions don't define the header in Chapel
    - So only the Python declaration is affected
  - Would like to relax these requirements in the future

pyChapel: Current Limitations

- **Only certain types supported for arguments, return types**
  - Primitives (int, string, bool, etc.)
  - 1D arrays of real(64)s (other types can be added over time)
    - Support is through NumPy
  - Note: Can still use other types within Chapel code itself
    - And in non-exported function interfaces

- **No current support for Chapel 3rd party packages**
  - Including other libraries added challenges, so not yet supported
    - e.g., Qthreads, GMP, RegExp
  - Implications:
    - Must use 'fifo' tasking rather than 'qthreads'
    - Limited to a single locale execution
    - Limits using standard modules like GMP and Regexp

pyChapel: Using Multiple Chapel Modules

- **Chapel module 'use' enabled by listing paths in Python**
  - In Python, write:

```
@Chapel(module_dirs=[<exact paths to src dirs>], sfile=<file>)
def foo(<arguments>):
  …
```

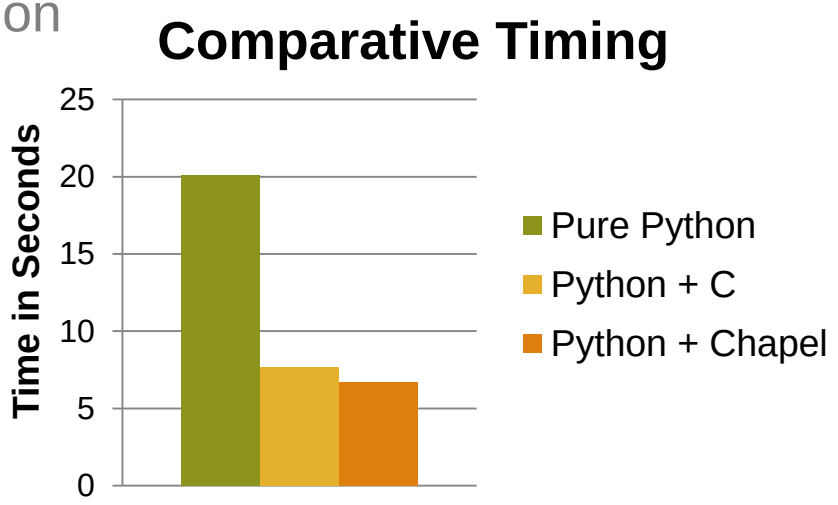
  - In Chapel:

```
export proc foo(<arguments>) {
  use ModName; // Module that lives in one of listed "module_dirs"
  …
```

pyChapel: Impact

Example 1: NumPy array, simple repeated math

- Boilerplate of program written in Python
- Python kernel:
  - Small part of total program
- C kernel:
  - Much larger than Python kernel
  - With ~3x speed up
- Chapel kernel:
  - Similar in size to Python kernel
  - With ~3x speed up, similar to C kernel
  - Uses “inlined” approach (adds 5 boilerplate lines)

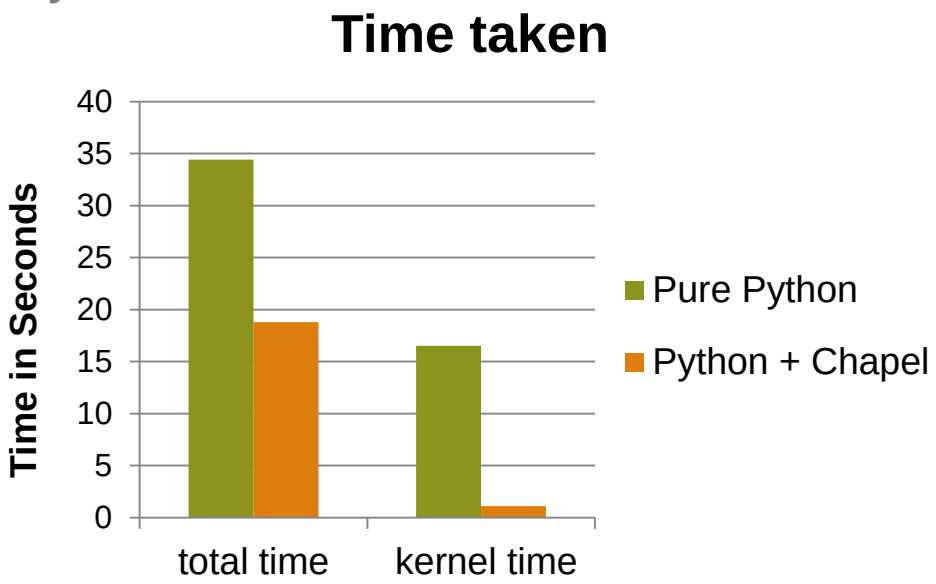


Language	Pure Python	Python+ C	Python+ Chapel
size of total code (SLOC)	17	38	23
size of kernel (SLOC)	4	20	5

pyChapel: Impact

Example 2: Finance example

- Boilerplate of program written in Python
- Python kernel:
  - Small part of total program lines
  - Half of program time
- Chapel kernel:
  - Similar in size to Python kernel
  - 16x kernel speed up
  - 2x program speed up
  - Uses “inlined” approach



Language	Pure Python	Python+ Chapel
size of total code (SLOC)	53	59
size of kernel (SLOC)	7	8

pyChapel: Next Steps

- **Generalize support for types at pyChapel interfaces**
  - additional array types and ranks
  - explore relaxing strict typing requirements
- **Improve Chapel libraries for precompiled use in pyChapel**
  - Useful depending on size, stability of Chapel code
- **Some bug fixes**
  - Updates to bfile/sfiles should trigger rebuilds
  - Other minor things
- **Longer term:**
  - Ability to generate Chapel classes or records from Python classes
    - Similar to extern record support for C
  - And maybe vice-versa

Other Standard Library Improvements

Other Standard Library Improvements

- **Added a .safeCast() method for safe integer downcasts**

```
42.safeCast(int(8))     // OK
257.safeCast(int(8))    // generates runtime error unless checks are disabled
```

  - used this in several standard modules to improve safety / portability
- **Added support for escaping #.####-style format strings**

```
writef("%{###}%{###}", i, j);
```

- **Added support for readf/writef of octal values**
- **Added support for additional C int types to SysCTypes**

```
c_ptrdiff, c_uintptr, c_intptr
```

Even More Standard Library Improvements

- **Simplified 'Random' module interfaces**
  - removed arguments that added complexity with clear value
- **Added a standard 'Assert' module defining assert()**
  - assert() has previously been defined in an internal IO module
- **Removed the 'Containers' module**
  - redundant given support for array-as-vector features in v1.10

Other Interoperability Improvements

Other Interop. Improvements

- **Added support for passing multidim. arrays externally**
  - now supported:

```
extern proc foo(X: [] real);
var A: [1..m, 1..n] real;
foo(A);
```

- **Generated errors when passing Chapel strings externally**
  - Should cast to c_string to pass to C, for example
- **Chapel identifiers now munged to reduce naming conflicts**
  - all user and standard module symbols munged
    - e.g., `var dft_plan: int;` ⇒ `int64_t dft_plan_chpl;`
  - export/extern symbols not renamed, obviously
  - internal module symbols still need to be protected similarly
  - feature can be disabled via a compiler flag (for developer use, e.g.)

Standard Library/Interoperability Priorities and Next Steps

Std Lib + Interop Priorities and Next Steps

- **Numerical Libraries**
  - FFTW: distributed memory versions
  - start working on additional key libraries: BLAS, GSL
- **Complete Path module**
- **Munging of internal module symbols**
- **Need-driven improvements to library dependence feature**
- **Improvements to pyChapel features as motivated by users**

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.

CRAY
THE SUPERCOMPUTER COMPANY

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