Runtime Improvements

Chapel Team, Cray Inc.

Chapel version 1.14

October 6, 2016

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

- Stack Tracing on Halt
- Atomic Improvements
- Memory and Comm Improvements
- Other Runtime Improvements

Stack Tracing on Halt

Stack Tracing: Background

```
1    // test.chpl
2    run();
3    proc run() {
4      writeln(logarithm(-1.0));
5    }
6    proc logarithm(x:real) {
7      if x <= 0.0 then
8        halt("invalid x");
9      return log(x);
10   }
```

```
$ ./test
test.chpl:8: error: halt reached - invalid x
```

- In this example, the error is in the caller of logarithm()
- But, the error message points to the body of logarithm()
- The location of the call is important but missing

Stack Tracing: This Effort

- **Error messages from halt can be unhelpful without context**
- **The approach is to use libunwind and optionally addr2line**
  - libunwind provides the mechanism to get a stack trace
  - Chapel compiler generates a table to help with translation
    - function addresses into names and declaration line numbers
  - … but it would be preferable to see line numbers for function calls
  - addr2line can translate the address of a call into a line number
- **Contributed by Andrea Francesco Iurio (GSoC student)**

Stack Tracing: Impact

```
1    // test.chpl
2    run();
3    proc run() {
4      writeln(logarithm(-1.0));
5    }
6    proc logarithm(x:real) {
7      if x <= 0.0 then
8        halt("invalid x");
9      return log(x);
10   }
```

```
$ ./test
test.chpl:8: error: halt reached - invalid x
Stacktrace

halt() at $CHPL_HOME/modules/internal/ChapelIO.chpl:659
halt() at $CHPL_HOME/modules/internal/ChapelIO.chpl:650
logarithm() at test.chpl:9
run() at test.chpl:4
```

Stack Tracing: Status and Next Steps

Status:

- CHPL_UNWIND=libunwind and CHPL_UNWIND=system available
  - libunwind: builds libunwind from third-party
  - system: uses a pre-installed libunwind
- Support for Linux and Mac OS X
- Linux stack traces can include call sites with *-g --cpp-lines*

Next Steps:

- Nightly testing with stack trace on halt enabled
- Remove internal module paths from the error messages

Atomic Improvements

Atomic Improvements: Background

- **Chapel atomics were heavily modeled after C11 atomics**

- **Chapel runtime had two implementations of atomics**
  - locks
    - For older C compilers that do not implement atomic operations
    - Runtime maintains a lock for each atomic variable
  - intrinsics (updated in this effort)
    - Implemented with __*sync* builtins (Intel extension popularized by gcc)
    - Much faster than locks

Atomic Improvements: This Effort

- **C standard atomics (cstdlib) implementation created**
  - Runtime support implemented as thin wrappers around C routines
    - With some additional code for floating-point operations
  - Easiest to maintain
    - About 2/3 the size of the other implementations
    - Leverages C compiler vendors' testing across architectures
  - Long term, will be the most portable and performant implementation

- **Current challenges**
  - gcc performance bug inhibits optimizations around atomic operations
  - Clang uses atomic headers from operating system
    - Usually out-of-date and buggy
  - Consequently, CHPL_ATOMICS=cstdlib is not yet the default

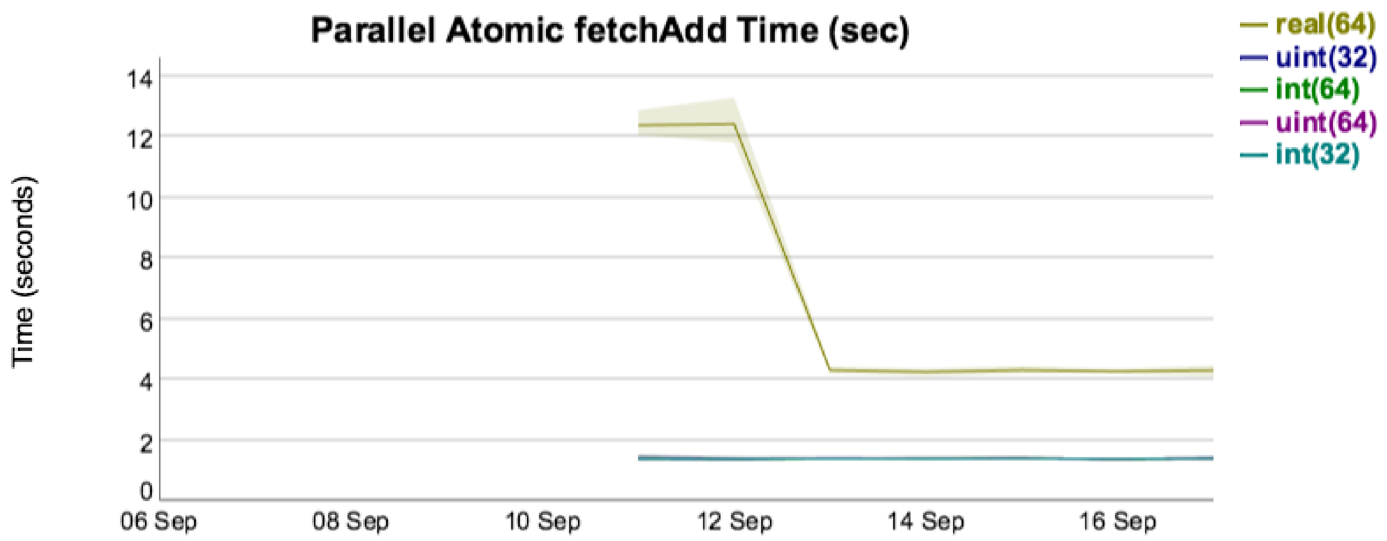
Atomic Improvements: This Effort

- **Fixed several bugs in the intrinsics implementation**
  - Made > wordsize loads/stores atomic on 32-bit platforms
  - Made < wordsize loads/stores atomic on 64-bit platforms
  - Removed type punning (undefined behavior)
  - Corrected floating-point fetch-and-add implementation

- **Improved performance of floating-point operations**
  - By eliminating use of volatile types and unnecessary memory barriers

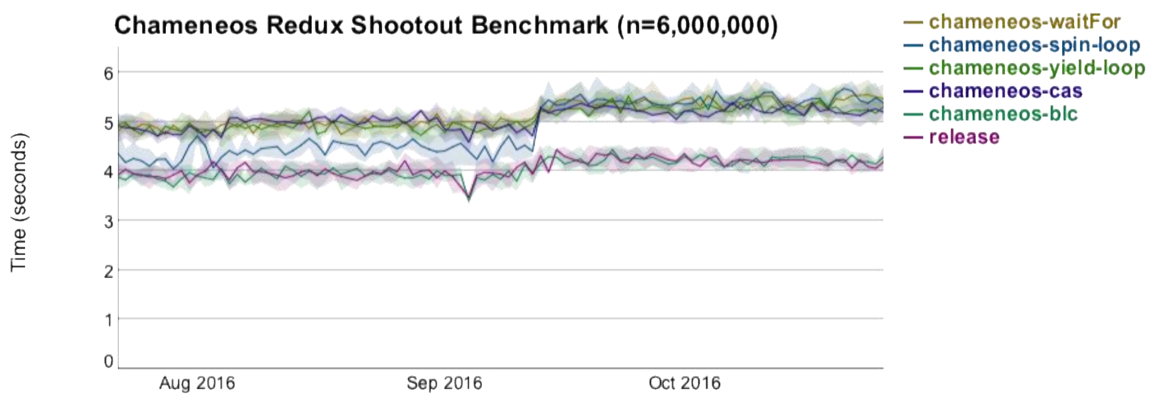
Atomic Improvements: Impact

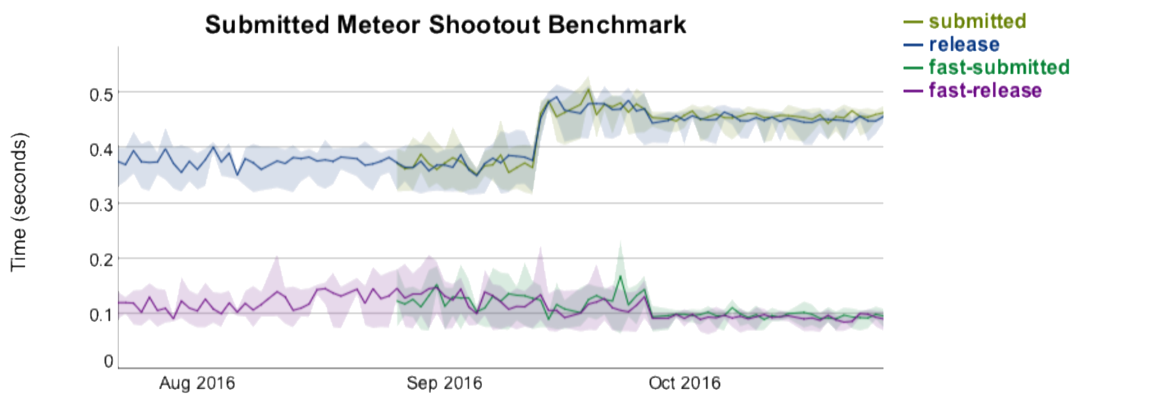
- Performance improvement for atomic real operations



Atomic Improvements: Impact

- **Minor regression for some shootout benchmarks**
  - Caused by atomic load bug fix





Atomic Improvements: Status and Next Steps

Status:

- C standard atomics available by setting CHPL_ATOMICS=cstdlib
- Intrinsics implementation has been overhauled
  - Serves as good default until C compiler cstdlib issues resolved

Next Steps:

- Work around clang atomic issues
  - Allow cstdlib atomics to become the default for clang
- Monitor gcc atomic performance regression issues
  - Contribute information to the bug reports where useful

Memory and Comm Improvements

CHPL_TASKS=fifo Stacks in Chapel Memory

Background: In fifo, a task runs on its host pthread's stack

- By default, pthread stacks are allocated directly from the OS
- Downside: task stacks were not in comm-layer-registered memory
  - examples: comm=gasnet and segment=fast, or comm=ugni
  - remote access to stack data was indirect (trampoline or Active Message)

This Effort: Get pthread/task stacks from Chapel heap

- Exception: want guard pages, but Chapel heap is on hugepages
  - can't change hugepage accessibility to create guard (huge)page
  - but hugepage stack alignment would be too wasteful of memory anyway

Impact: Performance improvement

- Much faster to stack-allocate Chapel variables than to heap-allocate
- Now stack allocation doesn't reduce communication performance

Stack-allocate Locals If Stack Is Communicable

Background: On-stmts may refer to function-local vars

- Historically, heaps could be remotely referenced and stacks not
- So, we heap-allocated locals ref'd in on-stmts to allow remote access
  - downside: heap allocation is much slower than stack allocation
- But now, some task stacks are remotely reachable

This Effort: Stack-allocate locals if stack is remotely reachable

- Examples:
  - comm=ugni, or comm=gasnet and segment=everything
  - tasks=fifo or muxed, without guard pages

Impact: Performance improvement

- Reduces allocation overhead

Next Steps: Do the same for tasks=qthreads

- Task stacks for tasks=qthreads are not remotely reachable
- Will need to change Qthreads itself, not just the Chapel shim

Optimize Local Non-blocking on-stmts

Background: *nonblocking* on-stmts optimize placed parallelism

- Used when source-code tasks do on-stmts and nothing else, as in:

```
begin on somewhere do …
cobegin { … ; on somewhere do … ; … }
coforall loc in Locales do on loc do …
```

- Runtime comm layer executeOnNB() initiates task on target locale
  - no runtime completion wait
- *But:* some such on-stmts turn out actually to be local
  - we used to handle this quite late, down in the runtime comm layer

This Effort: Optimize *local* nonblocking on-stmts

- Don't involve the runtime comm layer at all
- Instead, initiate on-stmt body directly, via runtime tasking layer

Impact: Reduced overhead

- Moreso for begin-on
- Not so much for cobegin-on, coforall-on which have termination sync

Other Runtime Improvements

Other Runtime Improvements

- **MassiveThreads working again for single-locale execution**
  - contributed by Kenjiro Taura

- **Bug fix: comm=gasnet out-of-segment NB GET/PUT fails**
  - out-of-segment + nonblocking is new combo, due to remote caching
  - solution: do these as blocking instead

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.

CRAY CHAPEL

CRAY

THE SUPERCOMPUTER COMPANY