Runtime Library Improvements

Chapel Team, Cray Inc.

Chapel version 1.10

October 2nd, 2014

COMPUTE | STORE | ANALYZE

Safe Harbor Statement

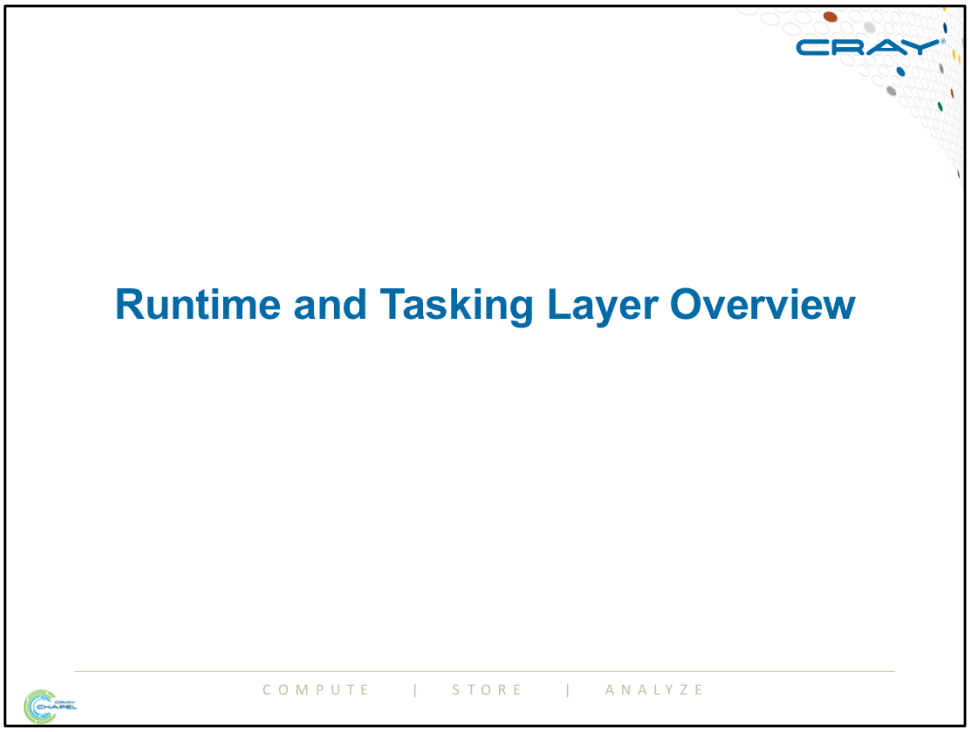
This presentation may contain forward-looking statements that are based on our current expectations. Forward looking statements may include statements about our financial guidance and expected operating results, our opportunities and future potential, our product development and new product introduction 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.

Executive Summary

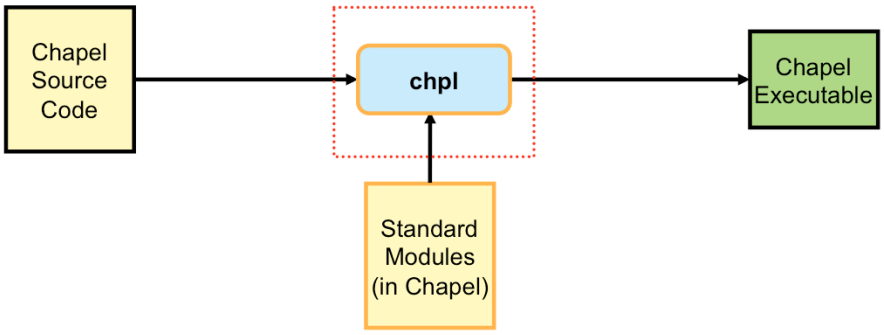
- **Chapel has traditionally used 'fifo' tasking**
  - maps each task to its own POSIX thread, runs to completion
  - heavier-weight than ideal
- **Lighter-weight tasking has long been available**
  - Several options available when building from source
- **This release switches to using Qthreads by default**
- **Includes a number of modifications to improve qthreads tasking performance**
- **Also improves interface consistency across taking layers**

Outline

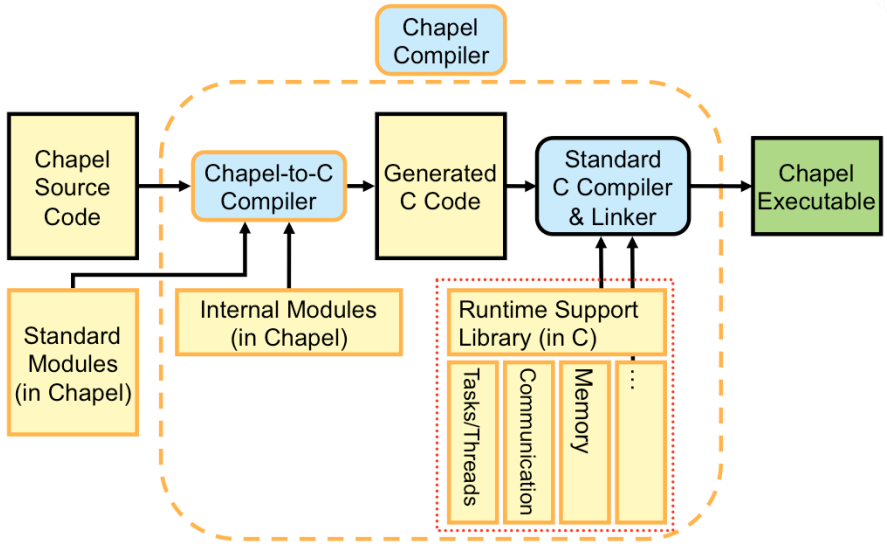
- Background
  - Runtime and Tasking Layer Overview
  - Qthreads Overview
- **Qthreads as Default Tasking Layer**
- **Specifying Number of Threads Symbolically**
- **Tasking Layer Specifies the Default Degree of Parallelism**
- **Stack Overflow Checking Unification**
- **Generalize CHPL_RT_* Environment Settings**
- **Other Runtime Library Improvements**


Runtime and Tasking Layer Overview
COMPUTE | STORE | ANALYZE

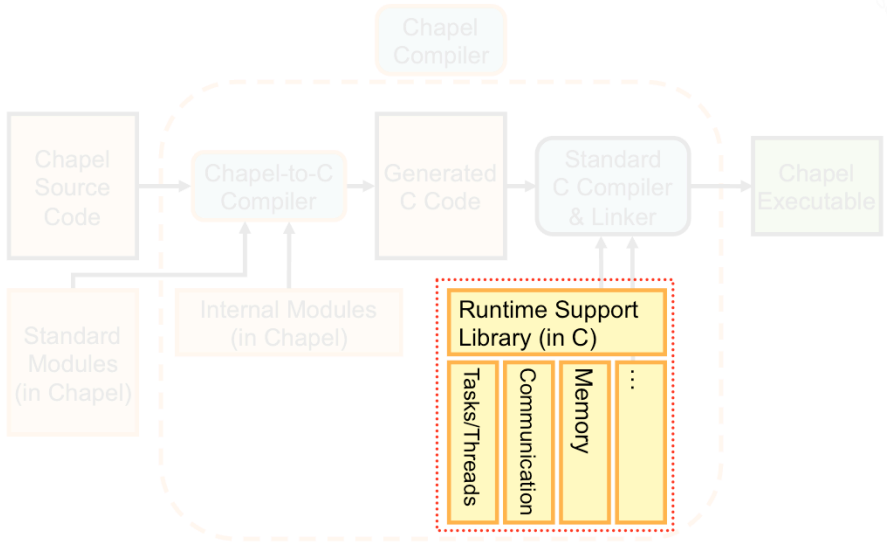
Compiling Chapel



Chapel Compilation Architecture



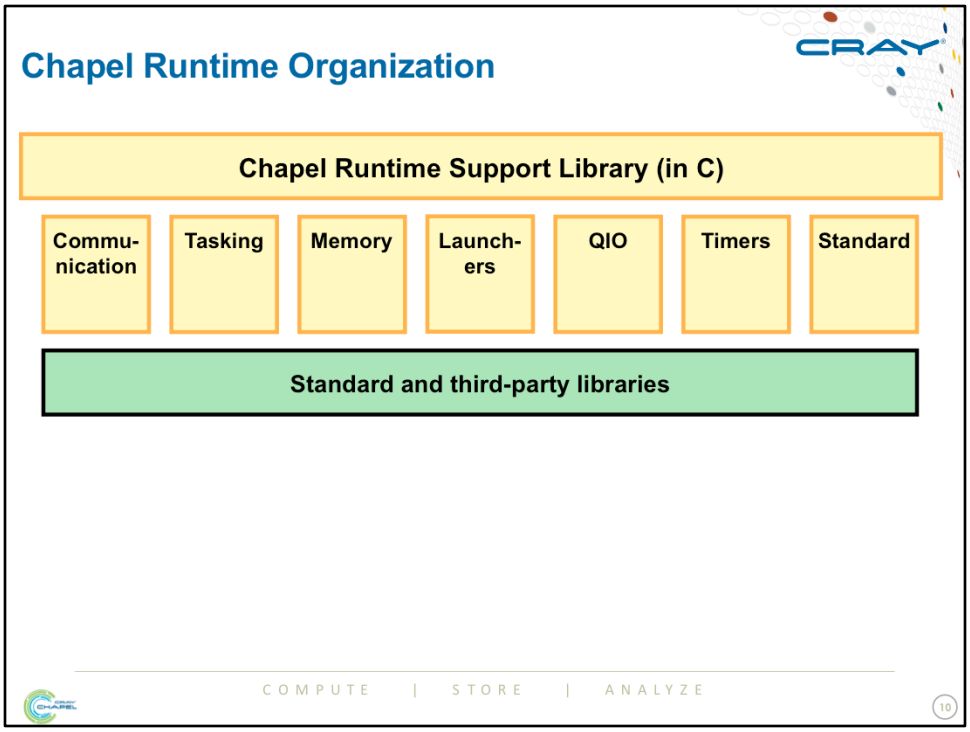
Chapel Compilation Architecture



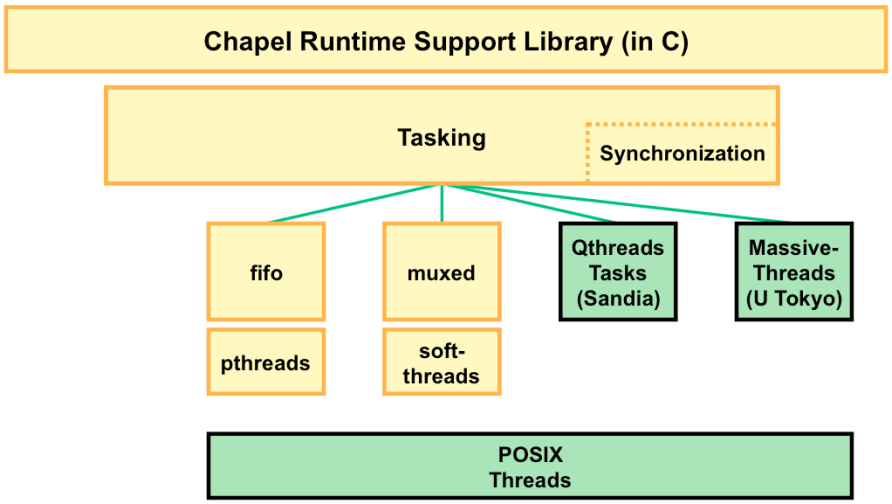
Chapel Runtime

- **Lowest level of Chapel software stack**
- **Supports language concepts and program activities**
- **Relies on system and third-party services**
- **Composed of *layers***
  - A misnomer – these are not layers in the sense of being stacked
  - More like *posts*, in that they work together to support a shared load
  - Standardized interfaces
  - Interchangeable implementations
- **Environment variables select layer implementations when building the runtime**
  - And when compiling a Chapel program, also select which already-built runtime is linked with it

Chapel Runtime Organization

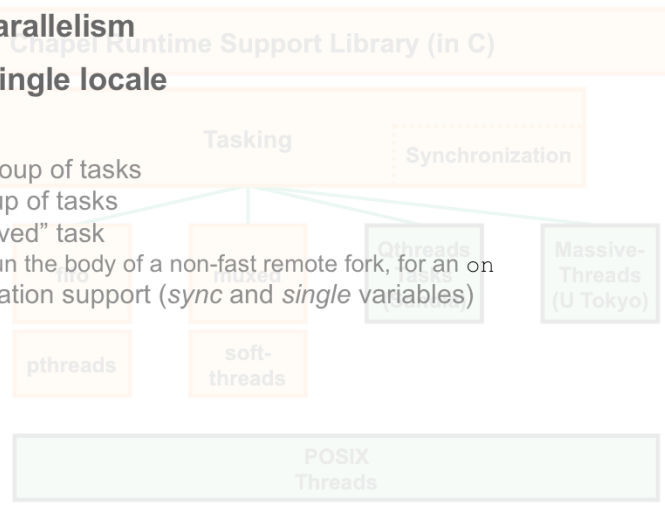


Runtime Tasking Layer



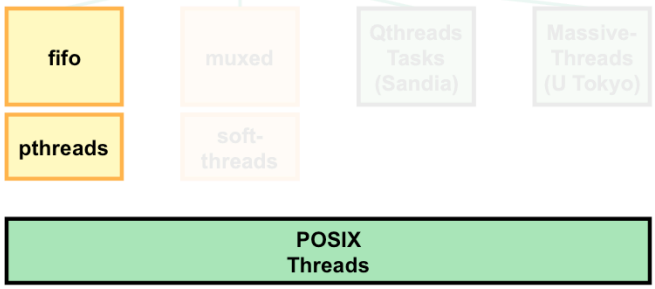
Runtime Tasking Layer

- **Supports parallelism**
- **Local to a single locale**
- **Operations**
  - Create a group of tasks
  - Start a group of tasks
  - Start a "moved" task
    - Used to run the body of a non-fast remote fork, for an `on`
  - Synchronization support (*sync* and *single* variables)

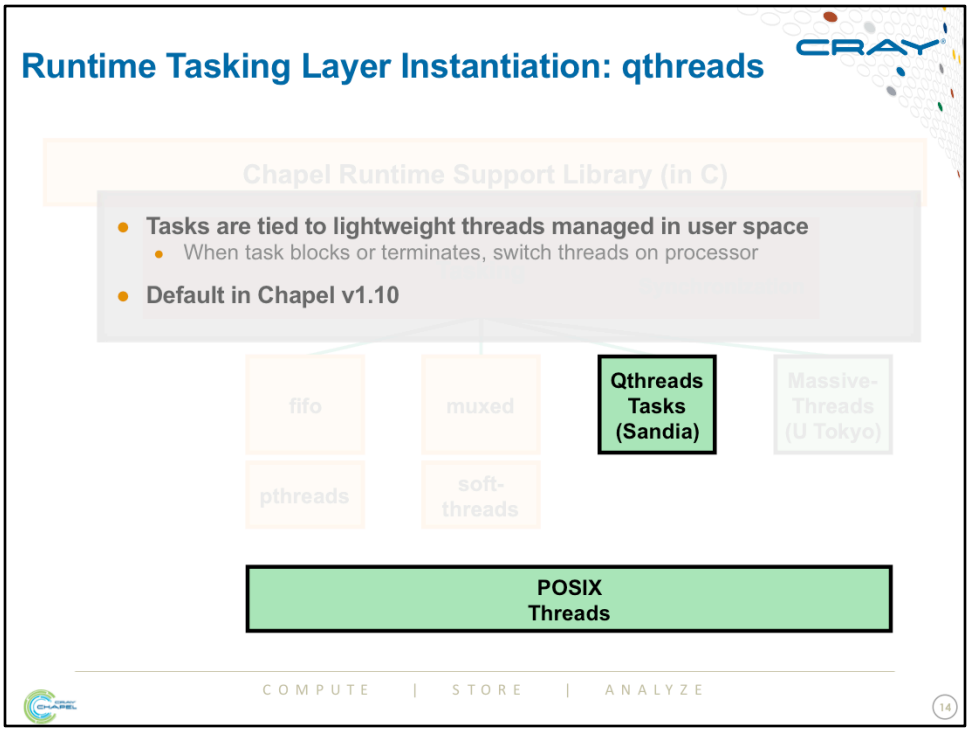


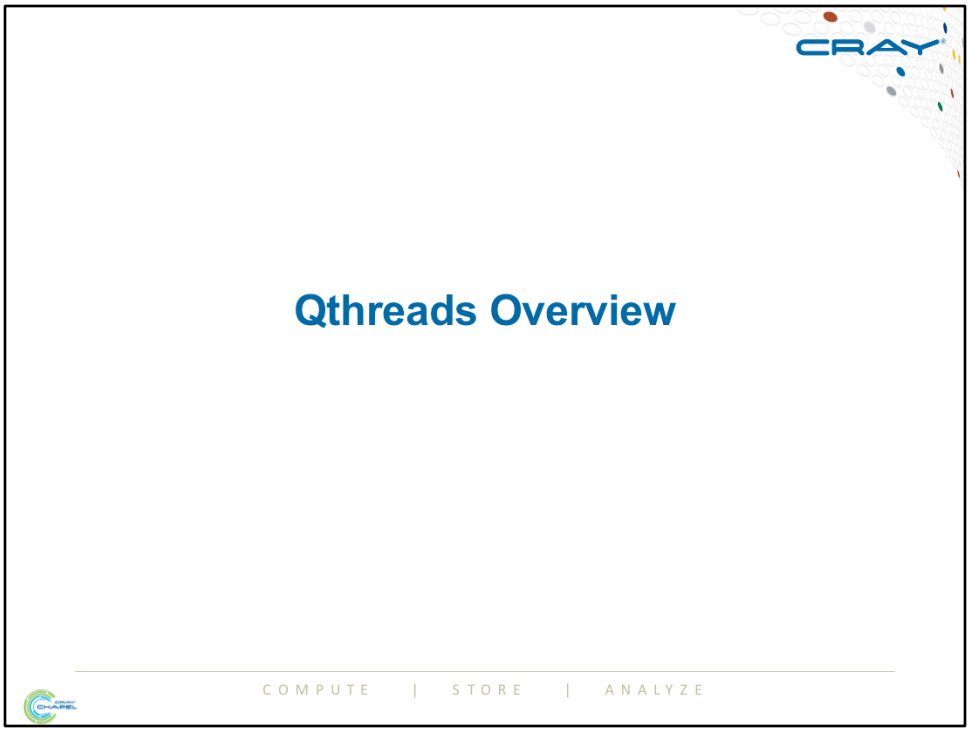
Runtime Tasking Layer Instantiation: fifo

- **Chapel tasks tied to POSIX threads**
  - When a task completes, its host pthread finds another to run
  - Acquire more pthreads as needed
  - Don't ever give pthreads up
- **Default in Chapel v1.9 and earlier**



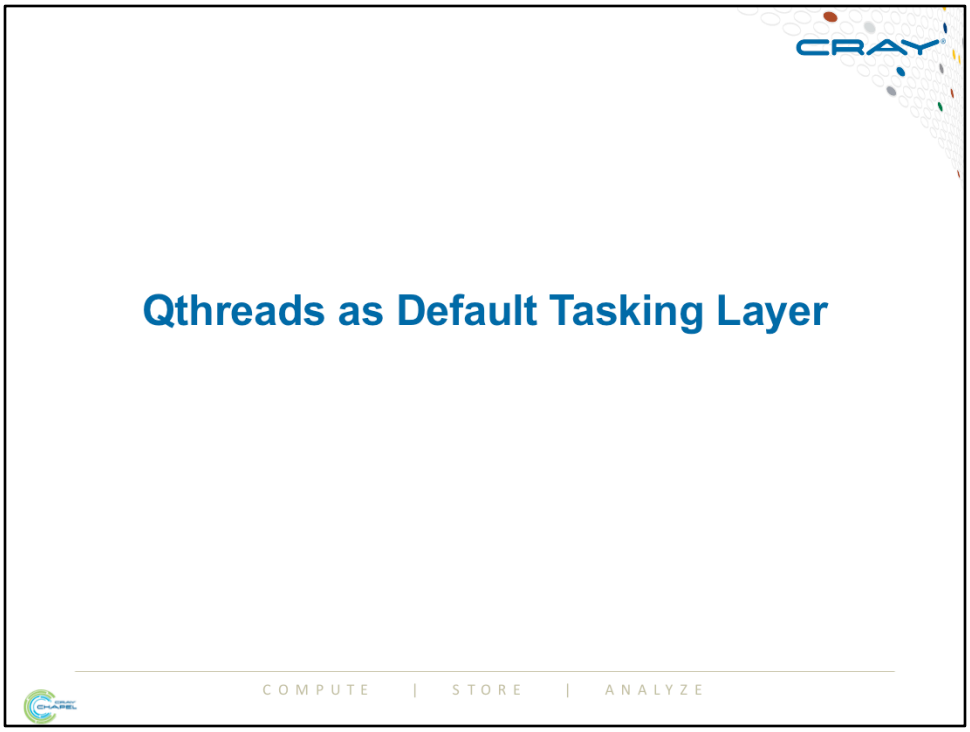
Runtime Tasking Layer Instantiation: qthreads

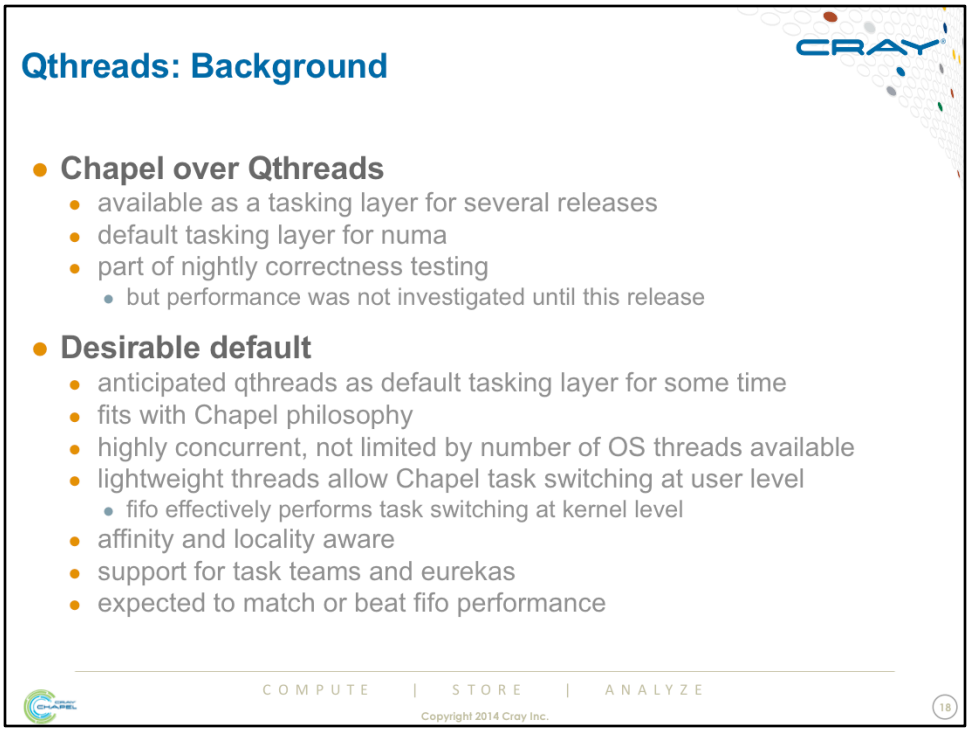



Qthreads Overview
COMPUTE | STORE | ANALYZE

Qthreads Overview

- **Lightweight, locality-aware tasking library**
  - developed and maintained by Sandia National Laboratories
  - locality-aware when affinity/topology library is available
    - hwloc is the best-supported affinity and topology library for qthreads
    - bundled with Chapel and enabled by default when CHPL_TASKS=qthreads
  - threads are entirely in user space
  - designed to be highly concurrent
    - run millions of threads, limited only by available memory
  - highly portable
  - multiple scheduler options
    - single threaded shepherds (simple fifo, lifo, etc. schedulers)
    - multi-threaded shepherds (locality-aware, work stealing scheduler)


Qthreads as Default Tasking Layer
COMPUTE | STORE | ANALYZE



Note that task teams and eurekas are available at the Qthreads level, but not the Chapel level (yet)

Qthreads: This Effort

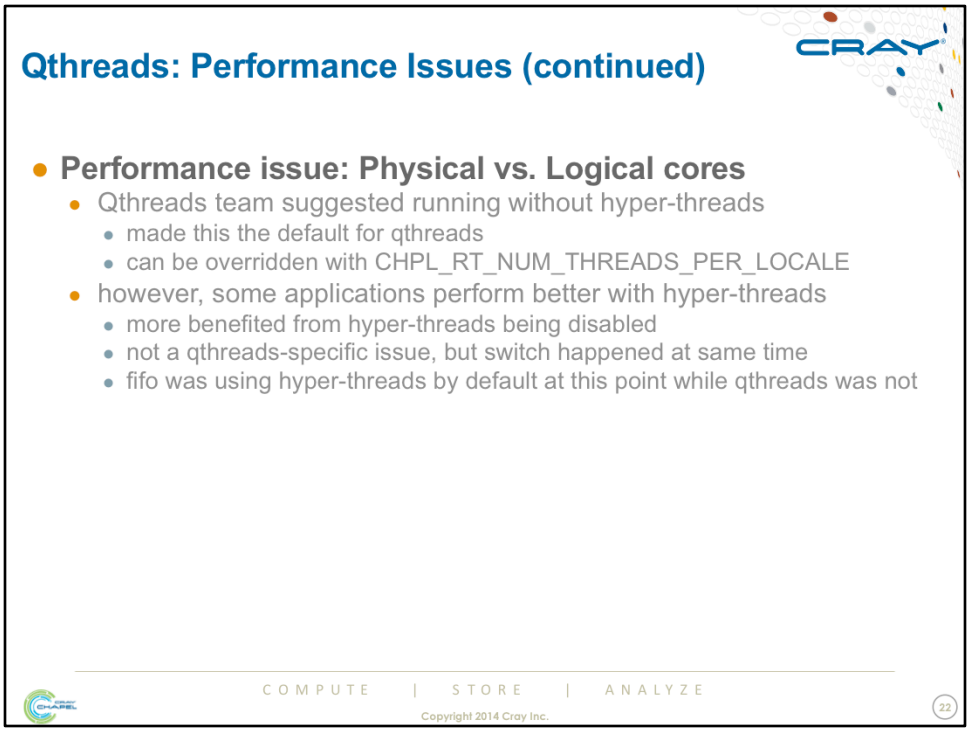
- **Address correctness regressions**
  - full single locale correctness testing was already running
    - many regressions had been addressed in previous releases
    - fix the few remaining regressions
  - started full nightly multi-locale correctness testing
    - fix test environment to run in an oversubscribed manner
    - fix regressions
- **Investigate performance**
  - believed qthreads should have better performance
    - but never tested in a formal way
  - started with a few manual runs of performance suite
    - tedious and time consuming
  - moved to automated nightly testing
    - qthreads performance results graphed alongside fifo
    - made comparison and performance tracking easy

Qthreads: Performance Issues

- **Performance Issue: Guard pages**
  - guard pages should be disabled for performance testing
    - initially just turned off guard pages for nightly runs
    - now turned off automatically with --no-stack-checks
    - and implicitly through --no-checks and thus –fast
- **Performance Issue: Busy-waiting**
  - by default, idle workers busy-wait while waiting for work
    - hurt performance for serial and low-task-count applications
    - now build Qthreads with –enable-condwait-queue
    - idle workers no longer busy-wait
    - slightly increases latency once more work is available

Qthreads: Performance Issues (continued)

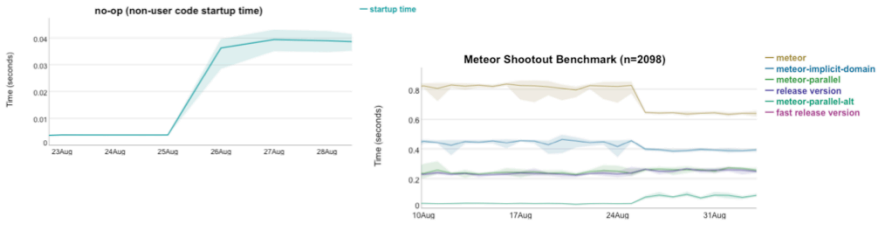
- **Performance Issue: Scheduler**
  - default scheduler (sherwood) did not have good overall performance
  - sherwood scheduler is a locality-aware, work-stealing scheduler
    - optimized for massive number of short-lived and steal-able tasks
    - did not have desired performance for longer-running tasks where work-stealing was not appropriate
    - no way to turn off work-stealing without crippling scheduler
  - switched to single-threaded scheduler (nemesis)
    - except for numa (which relies on locality capabilities of sherwood)
    - better overall performance



In hindsight we would have liked to have switched to qthreads as default and made the hyperthreads vs non-hyperthreads change on separate days to make performance characterization simpler.

Qthreads: Performance Issues (continued)

- **Performance Issue: Start-up time**
  - qthreads assumes long-running applications
    - grabs resources up-front expecting cost to be amortized
  - performance hit for short-lived benchmarks using external timers
    - such as shootout benchmarks
  - regression has not been addressed
    - start-up cost is still relatively small (~.04 seconds)



The startup time of qthreads is something we would like to investigate. It's not a high priority since the amount of time is still rather small. However, it would be nice to at least understand where the cost is coming from and if possible do something to address it

The startup time went from ~.004 seconds to ~.04 which is still a relatively small amount of overhead. Benchmarks that finish in under .1 seconds aren't running long enough to be much of an indication of real performance (

To add to the mystery, start-up time on Macs seems unaffected, only on Linux boxes.

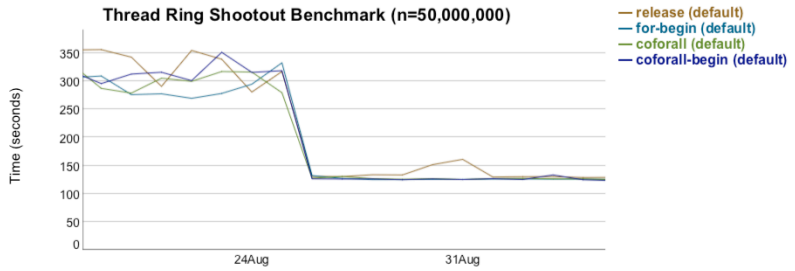
Qthreads: This Effort (continued)

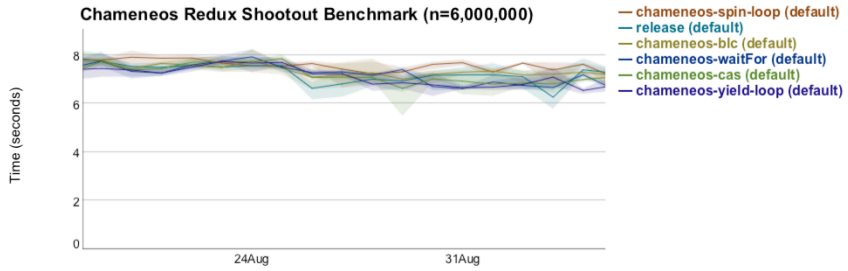
- **Made Qthreads default**
  - once correctness and performance regressions were addressed
  - except for cygwin and knc
    - not clear whether cygwin is officially supported by Qthreads
    - originally observed poor performance on knc
    - knc was tested with sherwood scheduler, needs re-testing with nemesis
  - nightly testing was mostly clean
    - ran into multiplication overflow in Qthreads source on 32-bit platforms
    - still use fifo for nightly valgrind testing

Qthreads: Performance Impact

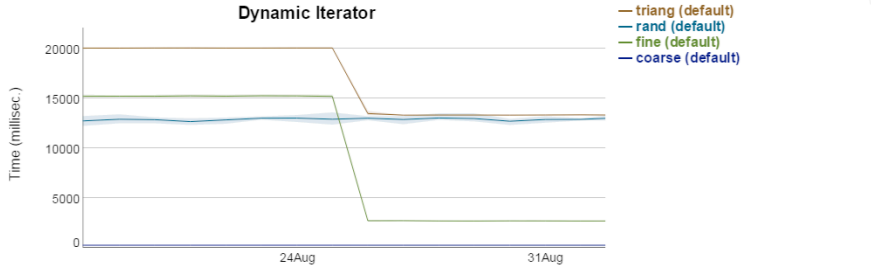
- **In general, performance is better with qthreads**
- **The following slides demonstrate performance improvements from qthreads**
  - ignoring policy change to use physical cores by default
- **These were collected on chap03**
  - 64-bit Linux
  - 1 dual-core AMD Opteron processor
  - no hyper-threads
    - isolate switch to qthreads from policy to use physical cores by default

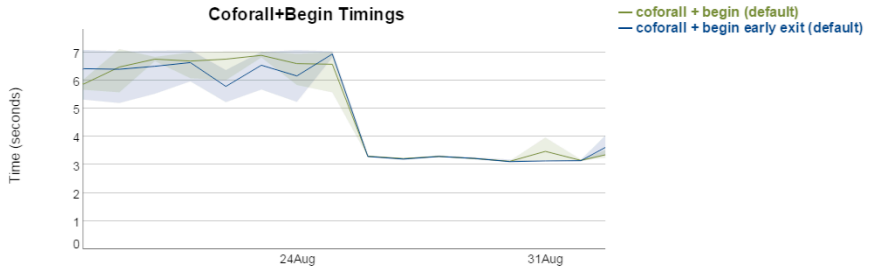
Qthreads: Performance Impact (continued)



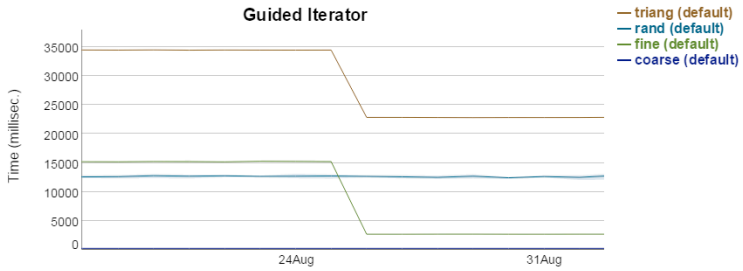


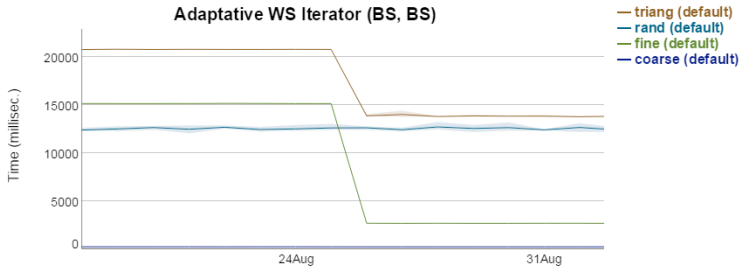
Qthreads: Performance Impact (continued)





Qthreads: Performance Impact (continued)



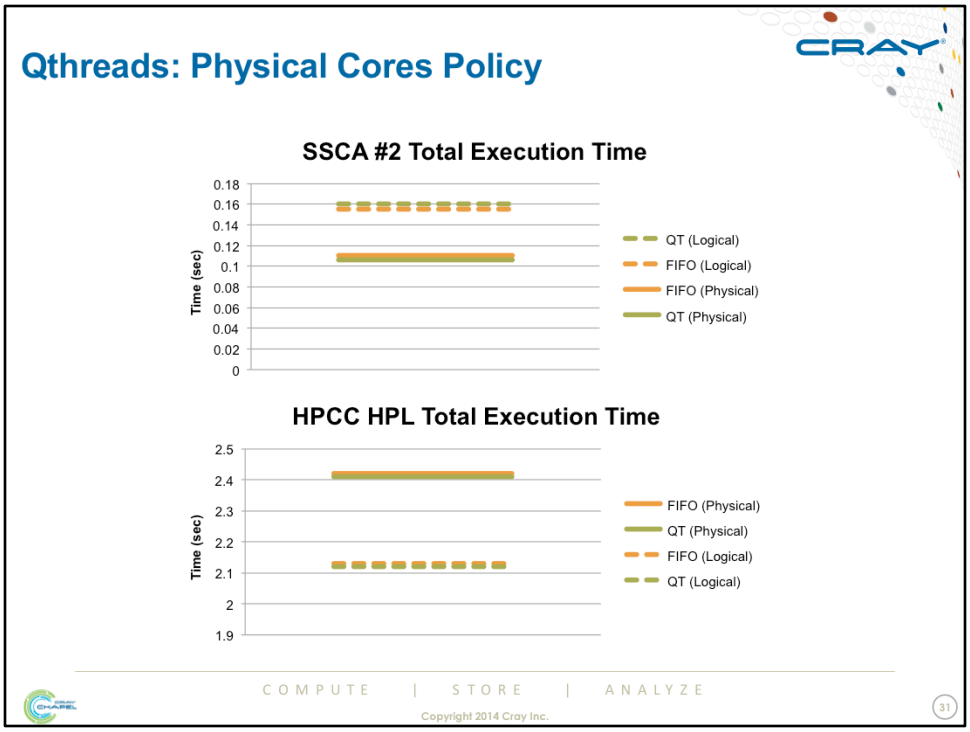


Qthreads: Physical Cores Policy

- **Physical vs. Logical cores**
  - qthreads defaulted to not using hyper-threads
    - better performance for most applications
  - at the time of the tasking layer switch, fifo used hyper-threads
    - makes isolating performance changes from switch difficult
    - in hindsight, policy and qthreads switch should have occurred separately

Qthreads: Physical Cores Policy

- **The following slide demonstrates that the use of hyper-threads affects qthreads and fifo similarly**
  - to show characterizations are not specific to qthreads
- **These were collected on chap04**
  - 64-bit Linux
  - 2 quad-core Intel Xeon processors with hyper-threading
    - =16 logical cores
  - 48 GB RAM



Different applications prefer using physical vs. logical cores.

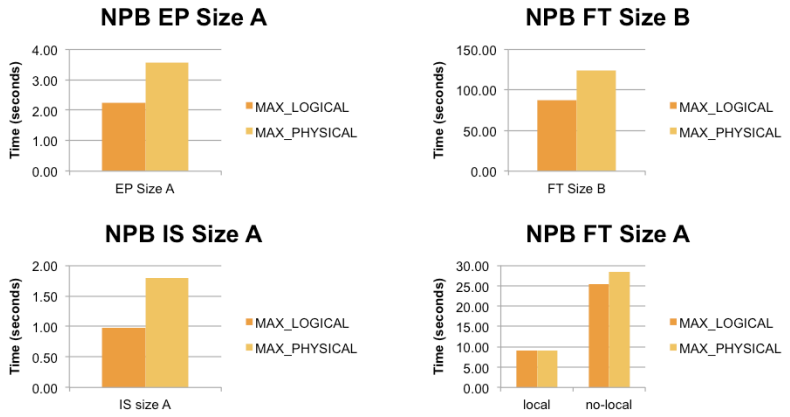
Note that the trends for which applications prefer physical vs. logical cores is the same for qthreads and fifo

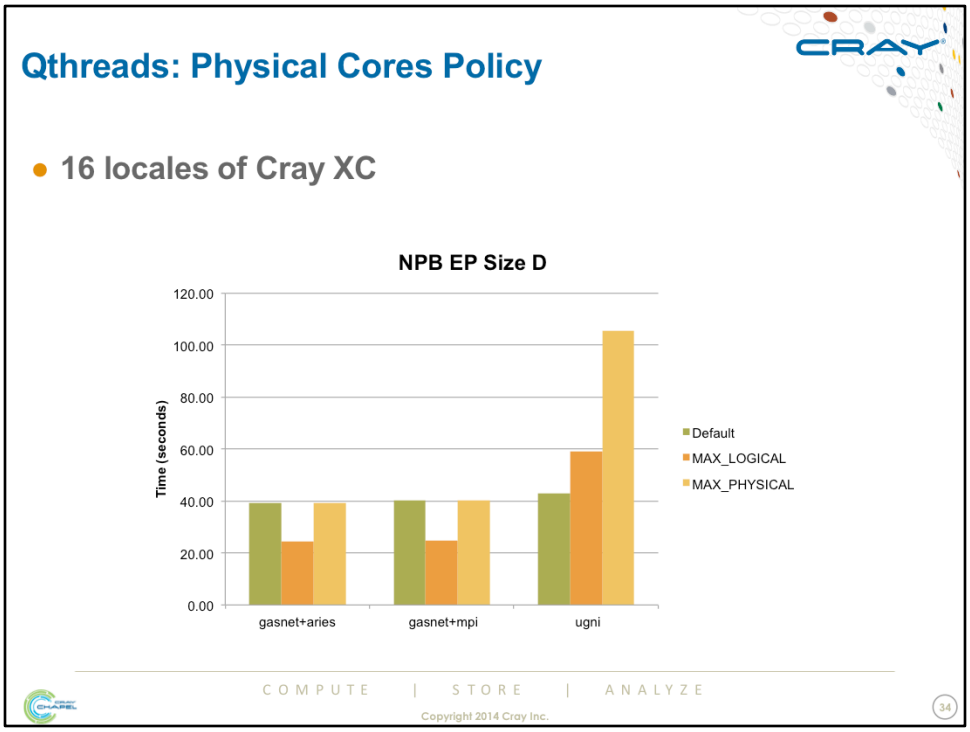
Qthreads: Physical Cores Policy

- **Some applications prefer using logical cores**
  - applications that prefer logical cores are almost entirely responsible for performance regressions in release-over-release graphs
- **The following slides demonstrate that preference for several applications**

Qthreads: Physical Cores Policy

- chap04, 8 core (16 HT)





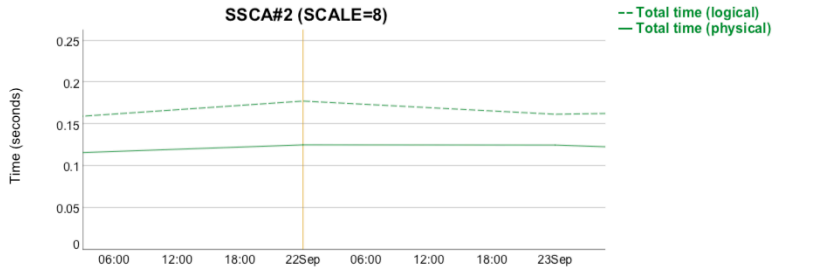
Note that MAX_PHYSICAL is the same for gasnet+aries and gasnet+mpi. They both use qthreads and the default is MAX_PHYSICAL.

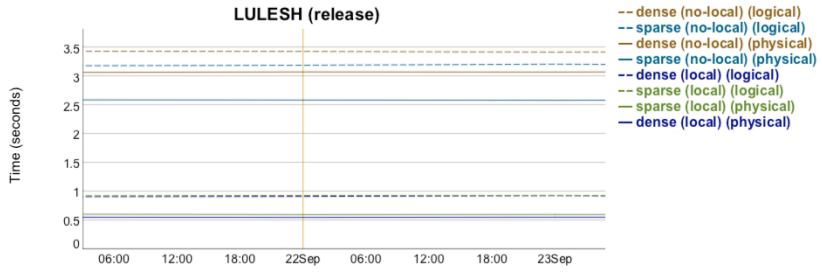
ugni refers to ugni+muxed. Muxed always uses the same number of hardware threads. The values refer to software threads. Default is 16x physical cores.

Qthreads: Physical Cores Policy

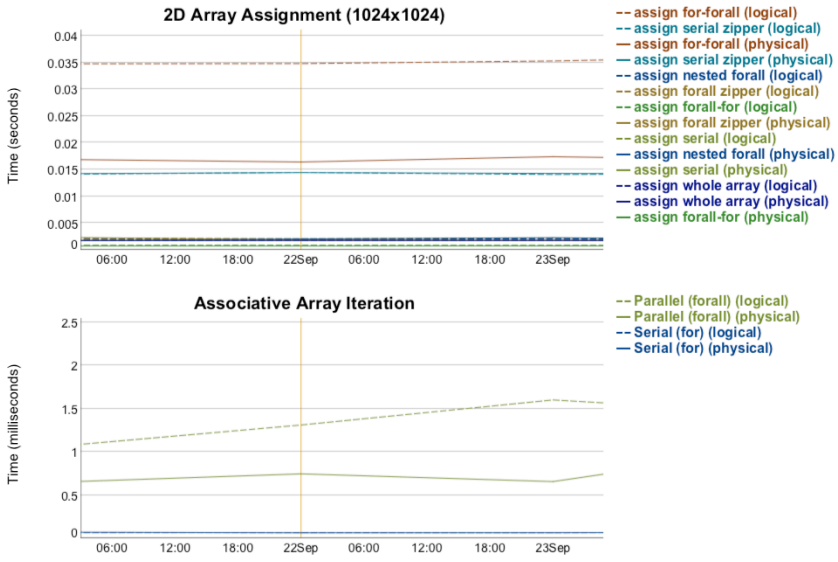
- **Majority of Chapel applications prefer physical cores**
  - as a result, the default for fifo was also changed to physical cores
- **The following slides demonstrate this preference**
  - collected on chap04, 8 cores (16 HT)

Qthreads: Physical Cores Policy

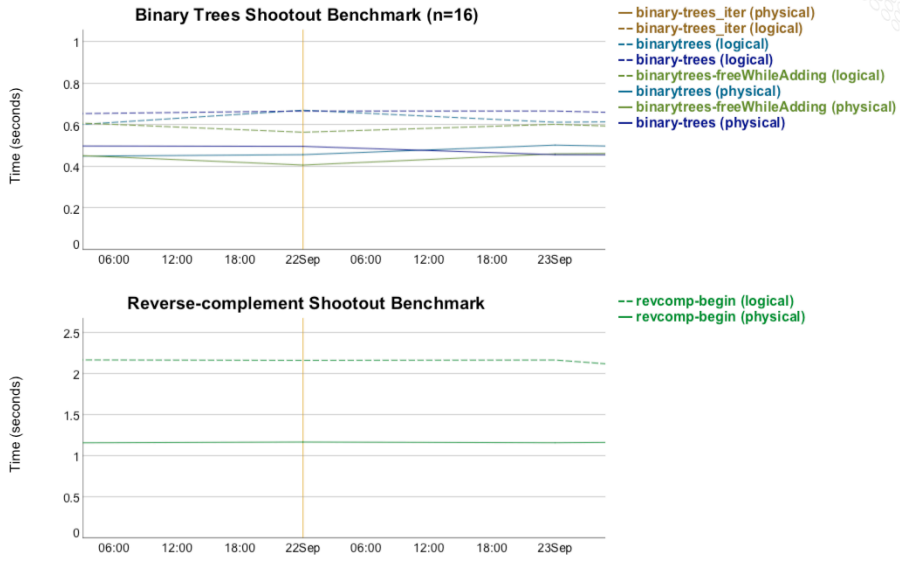




Qthreads: Physical Cores Policy



Qthreads: Physical Cores Policy

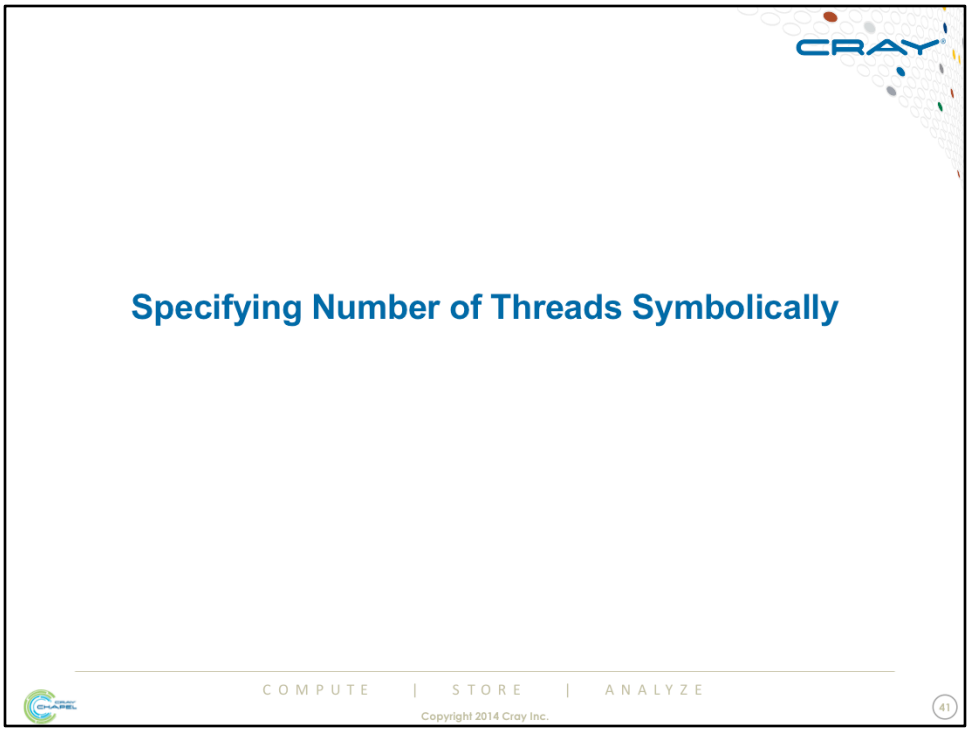


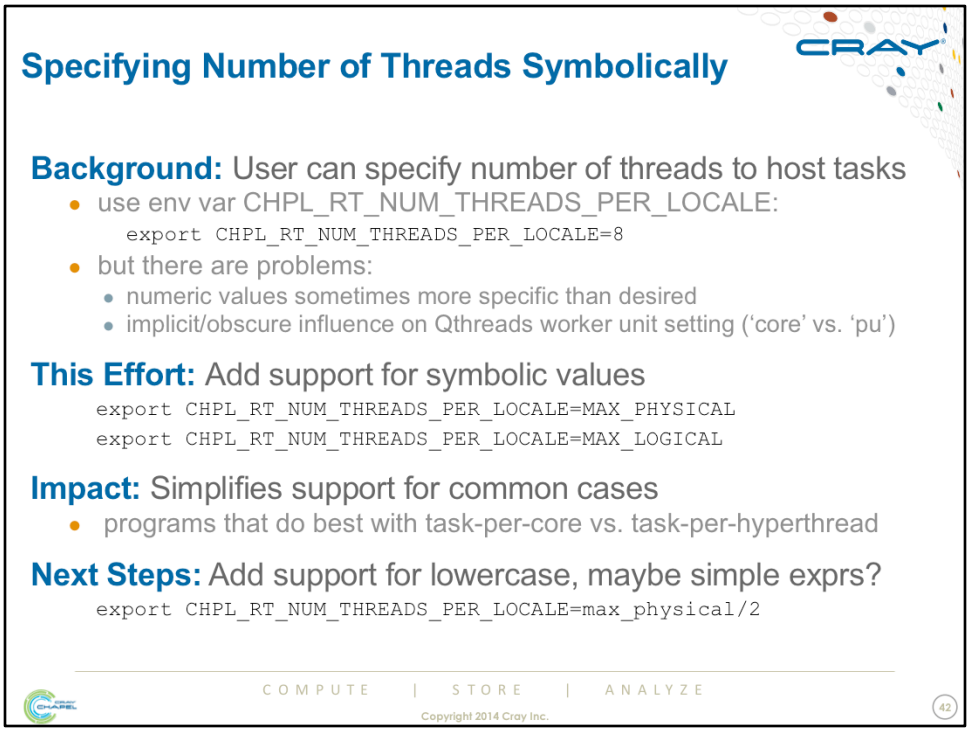
Qthreads: Impact Summary

- **Qthreads is now the default tasking layer**
  - In general, performance is better (with no user code changes)
    - except for short-running applications affected by start up time
    - and applications that benefit from hyper-threading
    - easy to enable hyper-threading for qthreads:
      `CHPL_RT_NUM_THREADS_PER_LOCALE=MAX_LOGICAL`
  - Number of tasks is now limited only by available memory
    - fifo may be limited by either available memory or available pthreads
    - qthreads is only limited by available memory

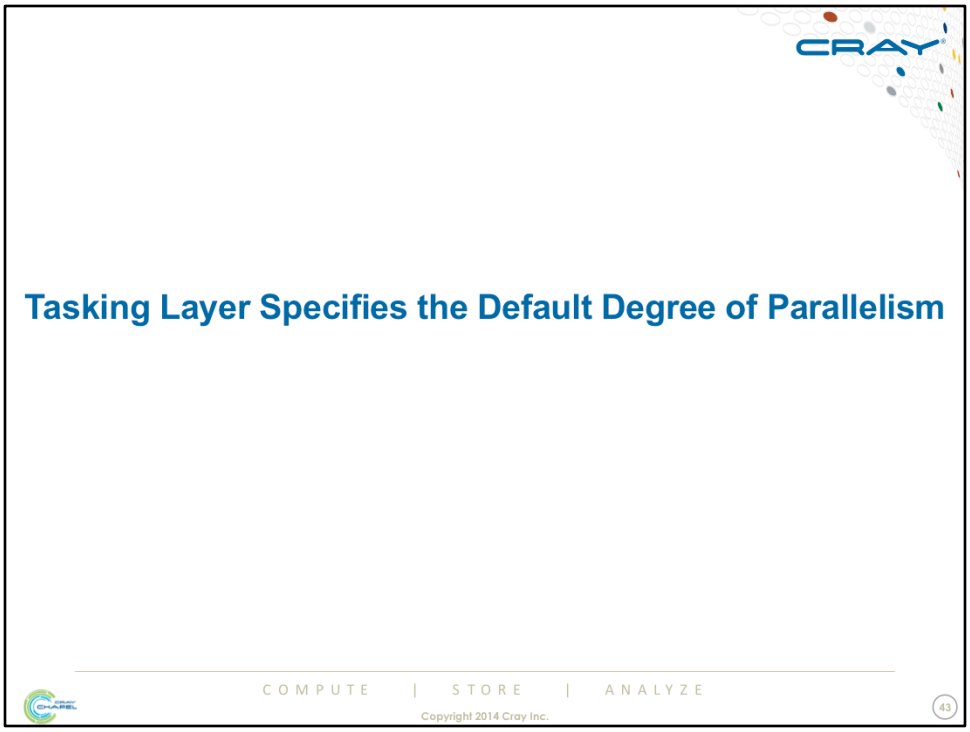
Qthreads: Next Steps

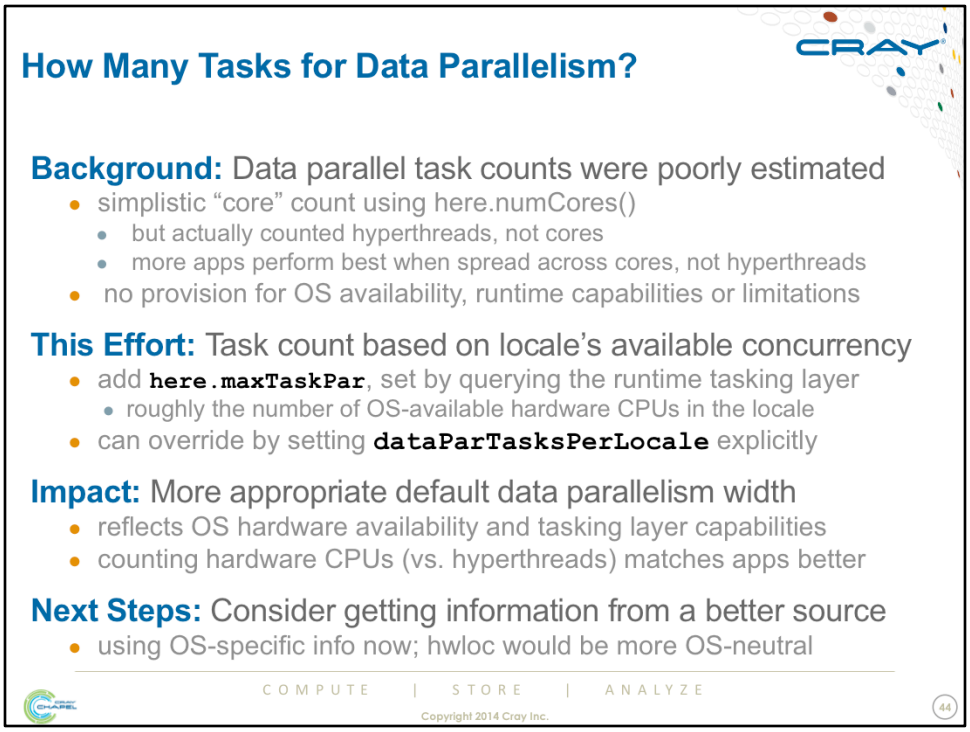
- work with Qthreads team to resolve sherwood performance
- additional performance tuning
  - there is plenty to investigate here
- investigate qthreads as default for cygwin and knc
- investigate start up time
- task teams?
- eurekas?


Specifying Number of Threads Symbolically
COMPUTE | STORE | ANALYZE
Copyright 2014 Cray Inc.
41

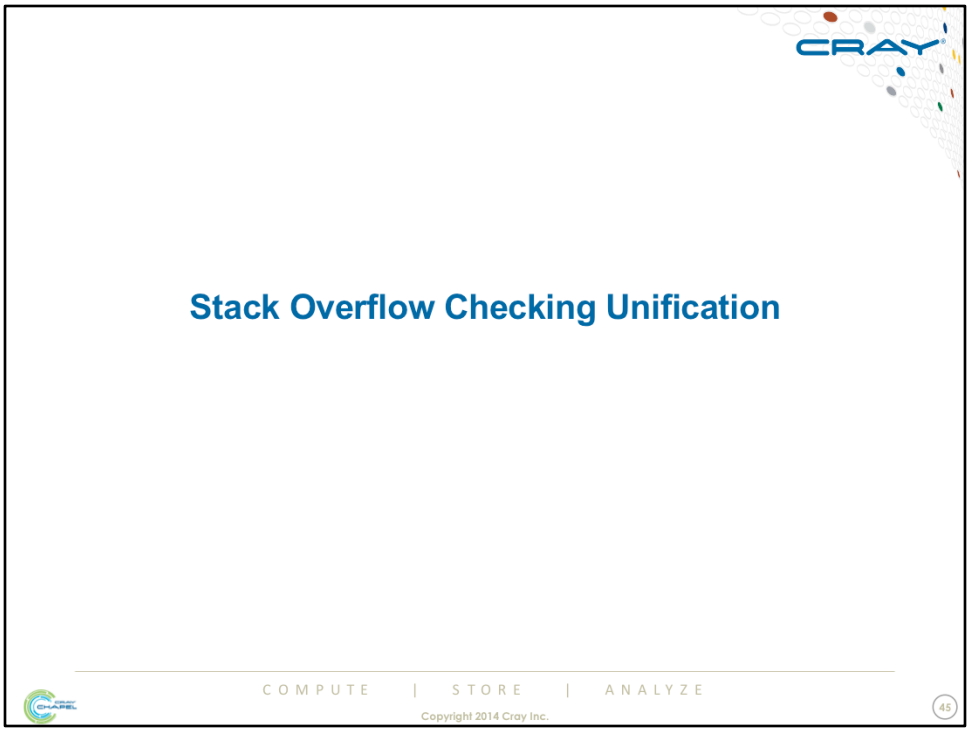


pu = "processing unit"


Tasking Layer Specifies the Default Degree of Parallelism
COMPUTE | STORE | ANALYZE
Copyright 2014 Cray Inc.
43



Note: for muxed, reflects soft-threads (a "runtime capability") rather than hardware threads.


Stack Overflow Checking Unification
COMPUTE | STORE | ANALYZE
Copyright 2014 Cray Inc.
45

Stack Overflow Checking

Background: Inconsistency in stack overflow checking

- differing techniques across tasking layer implementations
- differing enable/disable methods across tasking layer implementations

layer (method)	tasking layer build time	user program execution time
qthreads (guard pages)	make(1) variable CHPL_QTHREAD_NO_GUARD_PAGES	environment variable QT_GUARD_PAGES
fifo (guard pages)	--	-- (always enabled)
muxed (explicit checks)	preprocessor #define DO_STACK_OVERFLOW_CHECKS	environment variable CHPL_RT_STACK_CHECK_LEVEL
massivethreads (none)	--	--

Stack Overflow Checking: Improvements

This Effort: Improve consistency and uniformity

- enable/disable in the same way in all tasking layer implementations

Impact: A small step, but in the right direction

- set default at compile time; adjust at execution time

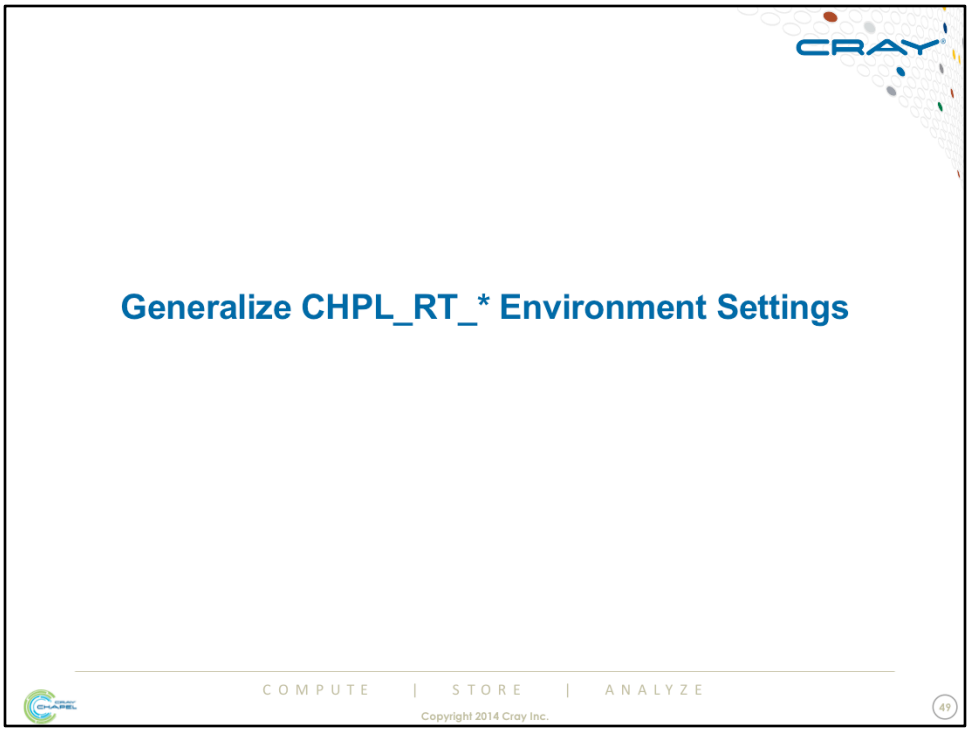
layer (method)	tasking layer build time	user program compile time	user program execution time
qthreads (guard pages)	make(1) variable CHPL_QTHREAD_NO_GUARD_PAGES	--[no-]stack-checks sets exec default	environment variable QT_GUARD_PAGES
fifo (guard pages)	--	--	-- (always enabled)
muxed (explicit checks)	preprocessor #define DO_STACK_OVERFLOW_CHECKS	--	environment variable CHPL_RT_STACK_CHECK_LEVEL
massivethreads (none)	--	--	--

Stack Overflow Checking: Next Steps

layer (method)	tasking layer build time	user program compile time	user program execution time
qthreads (guard pages)	make(1) variable CHPL_STACK_CHECKS	--[no-]stack-checks sets exec default	environment variable CHPL_STACK_CHECKS
…	…	…	…

and then:

layer (method)	tasking layer build time	user program compile time	user program execution time
all layers (guard pages)	make(1) variable CHPL_STACK_CHECKS	--[no-]stack-checks sets exec default	environment variable CHPL_STACK_CHECKS


Generalize CHPL_RT_* Environment Settings
COMPUTE | STORE | ANALYZE
Copyright 2014 Cray Inc.
49

Generalize CHPL_RT_* Environment Settings

Background: Support for runtime env vars varied widely

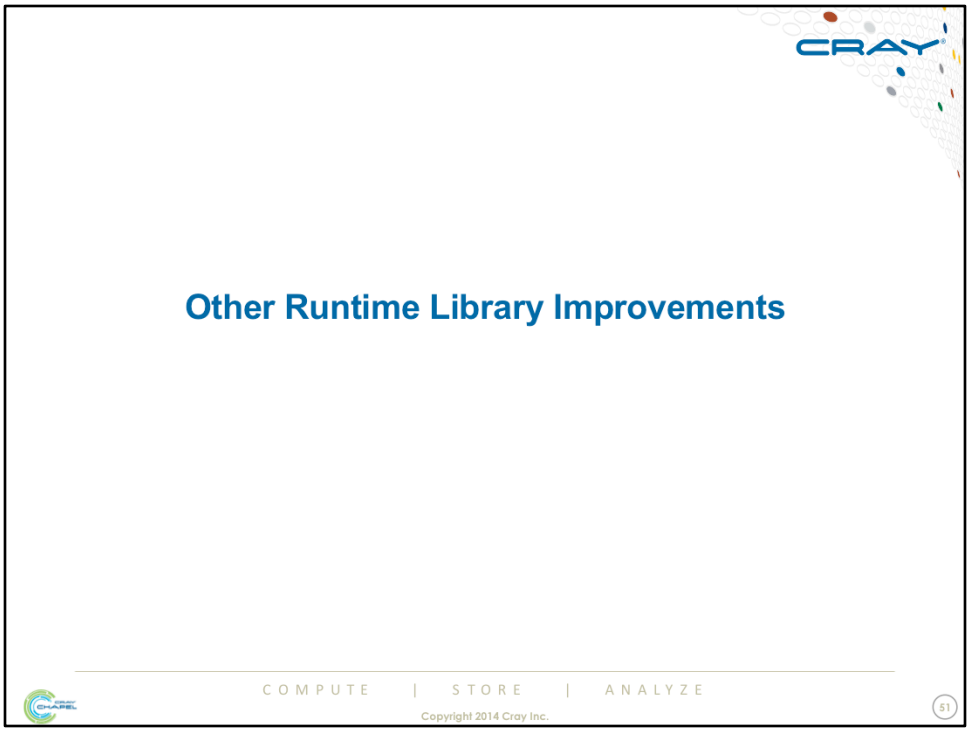
- runtime layer implementations differed in:
  - which environment variables were queried
  - how values were interpreted when they were set
  - default values when they weren't set

This Effort: More uniformity in most-common settings

- number of threads (`CHPL_RT_NUM_THREADS_PER_LOCALE`):
  - for fixed threads (qthreads, etc.): # threads, default: # hardware CPUs
  - for variable threads (fifo): max # threads, default: unlimited
- call stack size (`CHPL_RT_CALL_STACK_SIZE`):
  - default for all implementations: 8 MiB

Impact: Improved ease of use, less code, reduced maintenance

Next Steps: None planned


Other Runtime Library Improvements
COMPUTE | STORE | ANALYZE
Copyright 2014 Cray Inc.
51

Other Runtime Library Improvements

- added a warning if hugepages modules are in wrong state
- made Qthreads comm. progress a thread rather than task
- made barriers in runtime yield to avoid wasting resources
- launcher improvements:
  - improved the slurm-srun launcher
  - added environment forwarding to the amudprun launcher


Runtime Priorities/Next Steps
COMPUTE | STORE | ANALYZE
Copyright 2014 Cray Inc.
53

Runtime Priorities/Next Steps

- Continue improvements to Qthreads tasking
- Strategize regarding "logical vs. physical threads" choice
  - Would be nice to have a better story than "try both for your app"
  - Investigate optimal defaults for ugni/muxed
- Additional standardization of interfaces across task layers

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
CHAPEL
CRAY
THE SUPERCOMPUTER COMPANY
http://chapel.cray.com
chapel_info@cray.com
http://sourceforge.net/projects/chapel/