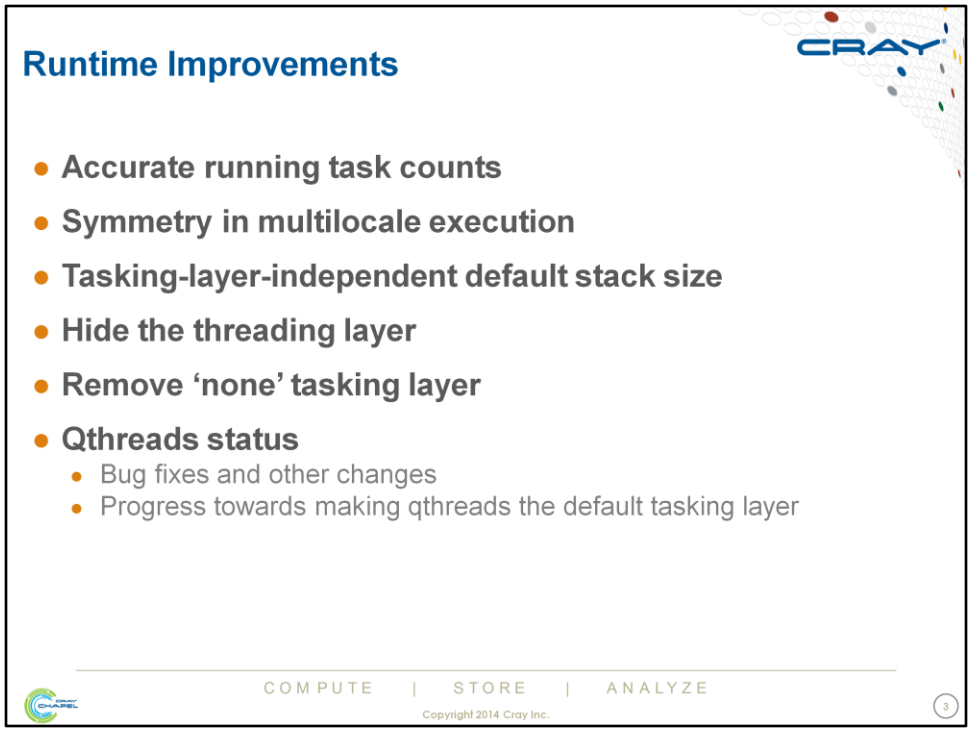
Runtime Improvements

Chapel Team, Cray Inc.
Chapel version 1.9 summary
April 17th, 2014 (released) / May 2014 (documented)

COMPUTE | STORE | ANALYZE

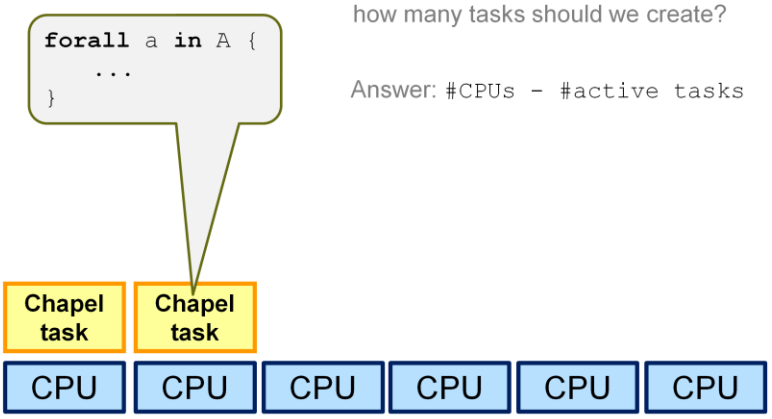
Safe Harbor Statement

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.



Note that though most of this work was motivated by preparing Qthreads to become the default tasking layer, the solutions were about making behavior consistent across tasking layers, regardless of which one was the default.

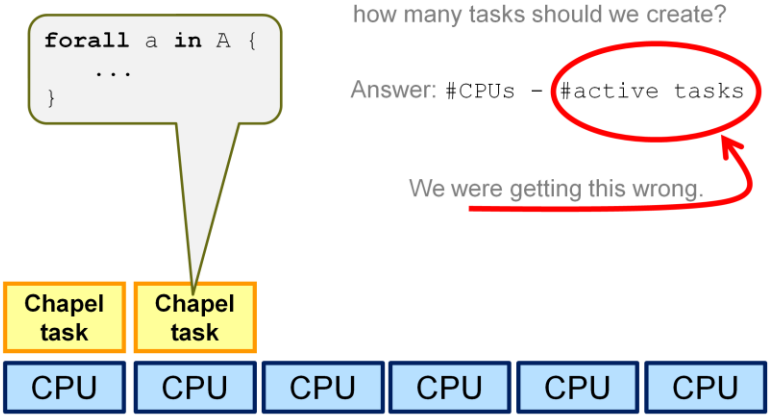
Accurate Running Task Counts



If we want to fully utilize the hardware, how many tasks should we create?

Answer: `#CPUs - #active tasks`

Accurate Running Task Counts



If we want to fully utilize the hardware, how many tasks should we create?

Answer: #CPUs - #active tasks

We were getting this wrong.

Accurate Running Task Counts

Background:

- New default in 1.9: dataParIgnoreRunningTasks=false
- When deciding how many tasks to create for a forall-stmt, take into account how many are already running
- Our running-task counting was inaccurate

Why Is This a Problem?

- Not enough tasks: under-perform due to not using available CPUs
- Too many tasks: under-perform due to increased overhead

Accurate Running Task Counts

What a mess!

fifo	Decrement on task termination late and not synchronized with end of parallel statement; sometimes not done before next parallel statement encountered. Intermittently over-counted tasks, under-utilized CPUs.
massivethreads	Did not maintain running task count at all, always said 0. Under-counted tasks, over-utilized CPUs.
muxed	Decrement on task termination late and not synchronized with end of parallel statement; sometimes not done before next parallel statement encountered. Intermittently over-counted tasks, under-utilized CPUs.
qthreads	Only counted running tasks on current shepherd. But often we have more than one shepherd. Under-counted tasks, over-utilized CPUs.

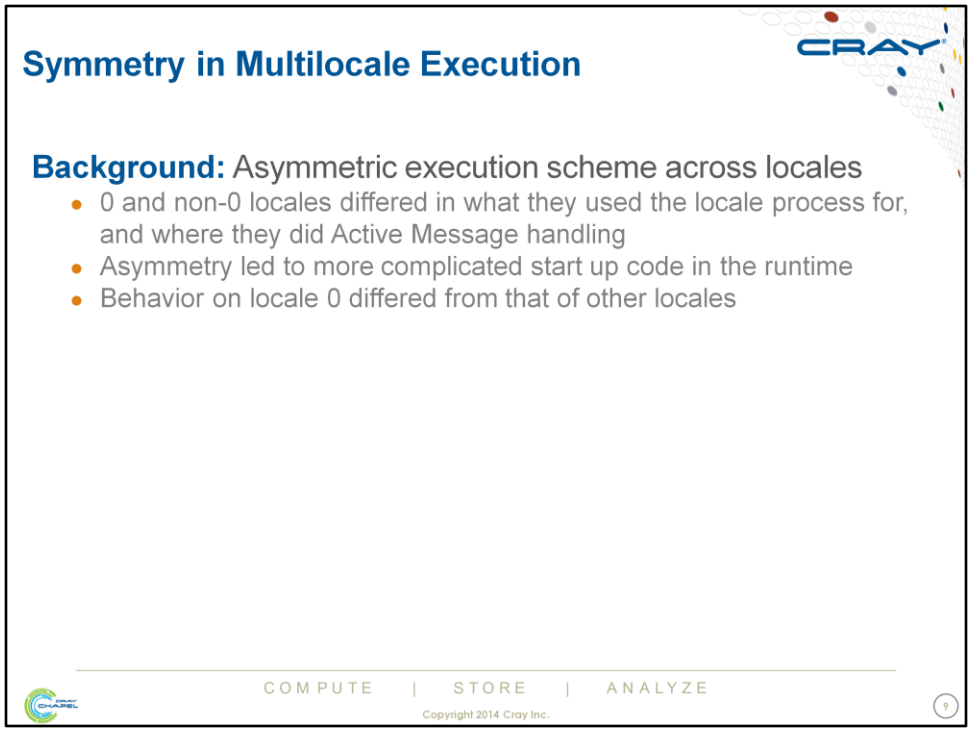
Accurate Running Task Counts

Solution: Moved running task count into the modules

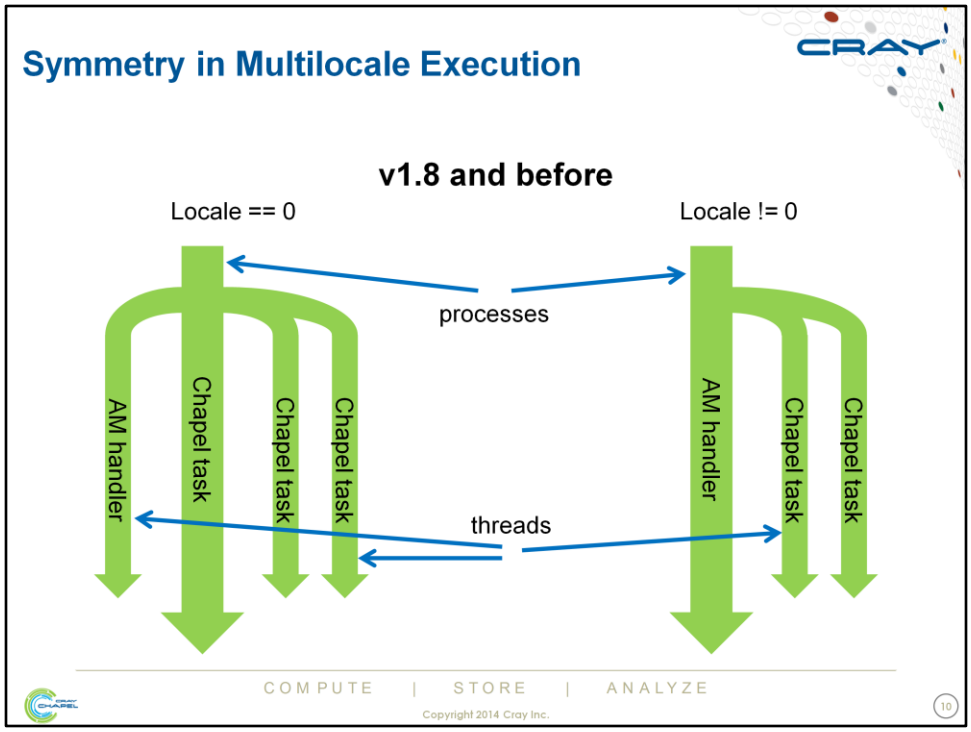
- Reduces duplicate code
- Gets the right answer!
  (for all tasking implementations)

Next Step:

- Remove little bit of remnant task counting code from runtime



"Behavior" (last bullet) in the sense of what happened there and why. For example, if you noticed that on locale 0 you were seeing interference between Active Message handling and Chapel tasks, you could not deduce that that was also happening on other locales.

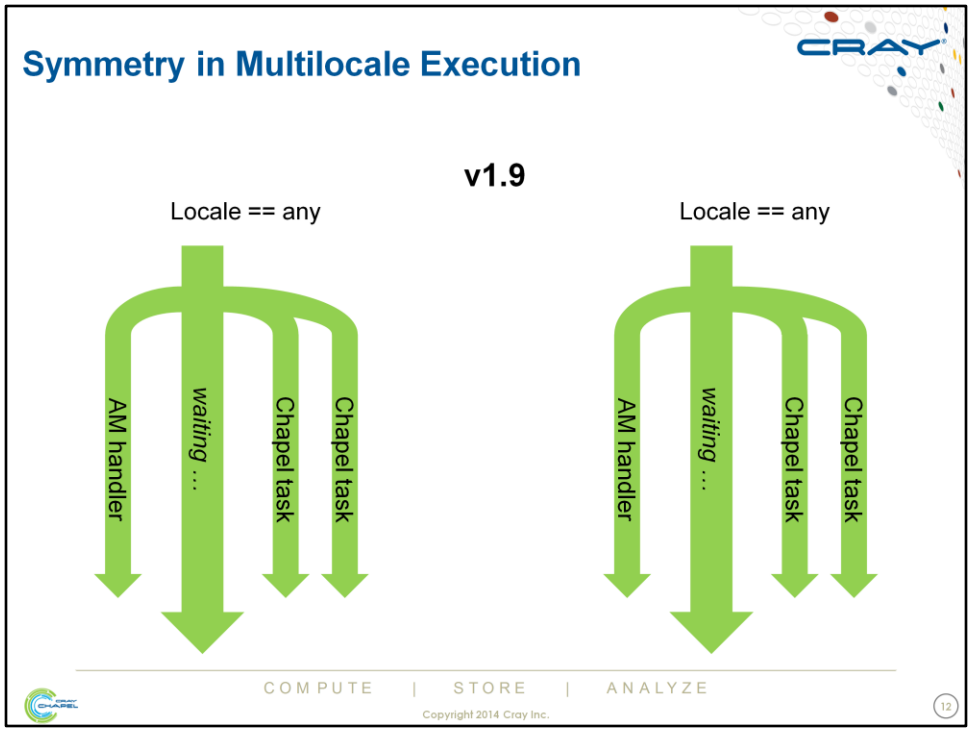


- Cannot configure tasking layers to behave the same on all locales
- Hard to reason about behavior on non-0 locales based on measurements taken on locale 0 and vice-versa

Symmetry in Multilocale Execution

This Effort: Make execution scheme symmetric across locales

- All locales have the initial process waiting for completion
- Start the AM handler as a separate thread/task on all locales



- Tasking layers behave the same on all locales
- Knowledge gained about behavior on one locale applies to other locales also

Symmetry in Multilocale Execution

Impact:

- Hides knowledge of comm layer AM handling inside comm layers
- Still have locale 0/non-0 asymmetry
  - Okay, because inherent to Chapel's execution model
- Improves ability to reason about overall behavior from observations or measurements done on one locale

Next Steps:

- Further simplification of runtime start up code

Tasking Layer Independent Default Stack Size

Problem:

- Tasking implementations set default call stack sizes themselves
- Duplicated effort without duplicating results
- Led to surprises when switching from one implementation to another

Solution:

- Moved default call stack size selection into common code

Benefits:

- Less code to maintain
- Fewer surprises; more dependability

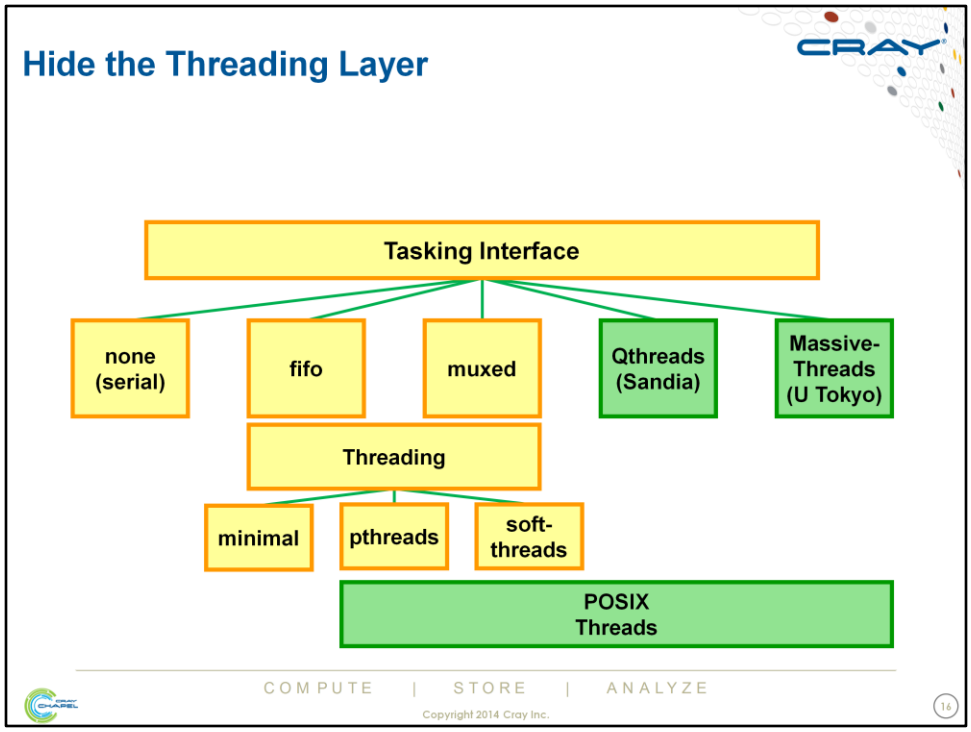
Hide the Threading Layer

Background:

- For some time, we've had both tasks and threads as 1st-class entities:

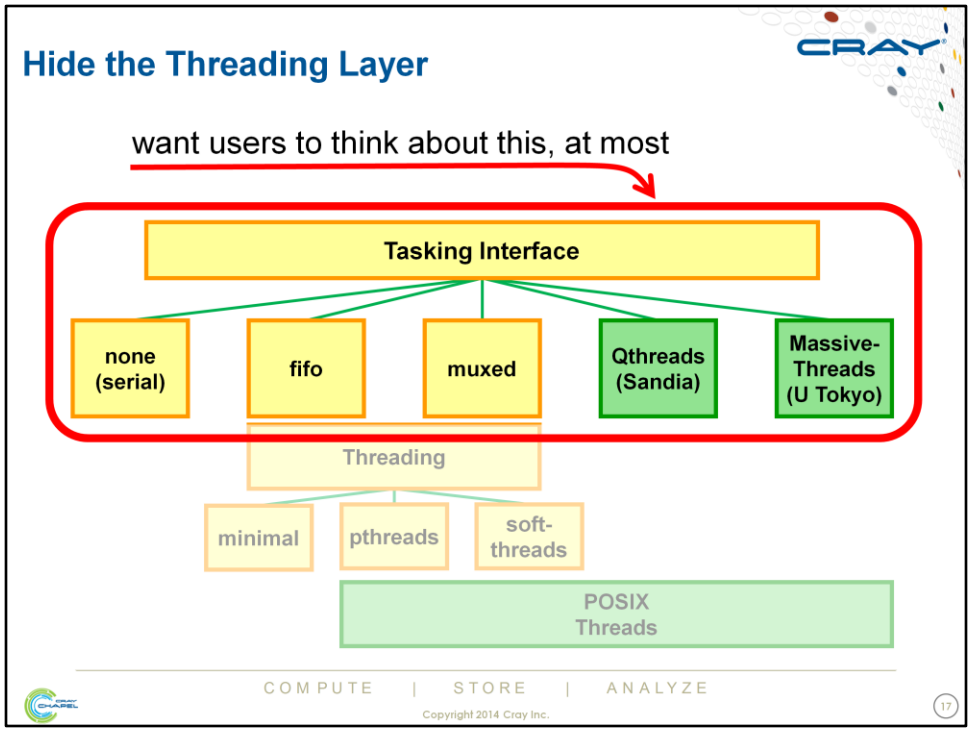
```
[gbt@host]$ printchplenv
…
CHPL_COMM: none
CHPL_TASKS: fifo
CHPL_THREADS: pthreads
CHPL_LAUNCHER: none
…
```

- But this is unnecessarily complicated

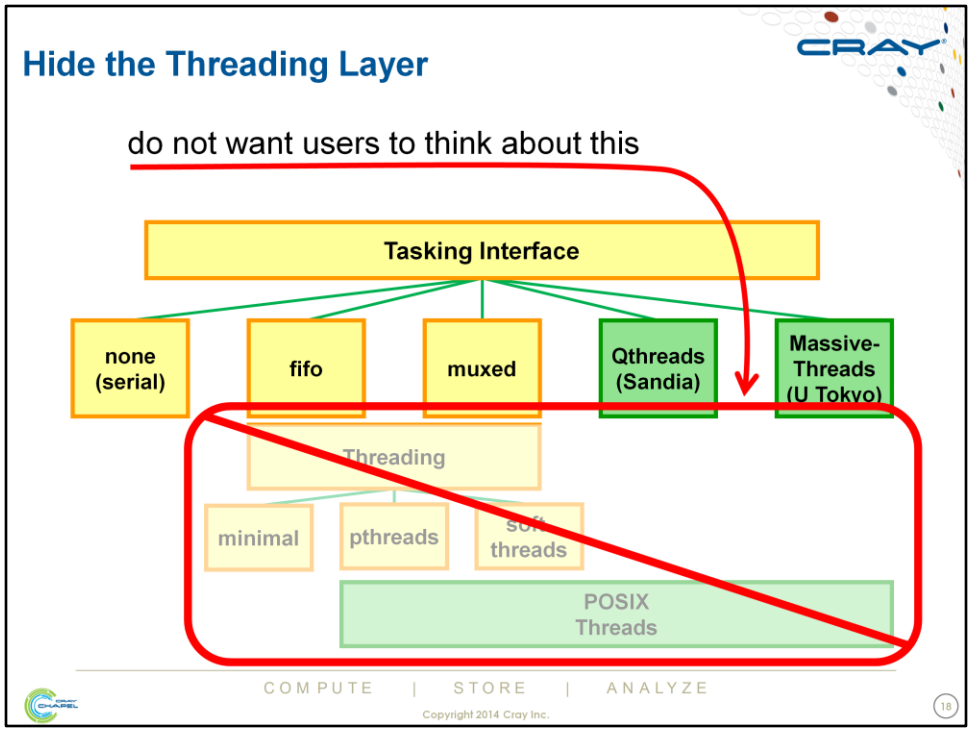


Here's a picture of how tasking is implemented.

From top to bottom, how far down do we want users to be thinking about this?



- **Tasks are the Chapel abstraction of execution**
  - Behave as described in the Chapel Language Specification
  - Parallelism and synchronization is expressed in terms of tasks
  - Want programmers to reason on this level



- **Threads are an underlying software abstraction by which tasking layers make use of hardware processors**
  - Defined and used differently by each tasking layer
    - For fifo, a thread is a Linux (UNIX) pthread and a Chapel task is bound to a single thread throughout its existence
    - For qthreads, a thread is a worker qthread and a Chapel task may shared its thread with other threads and/or change host threads during its life
    - Etc.
  - Don't really have anything to do with Chapel programming
  - Do not want programmers to be burdened with this level of detail

Hide the Threading Layer

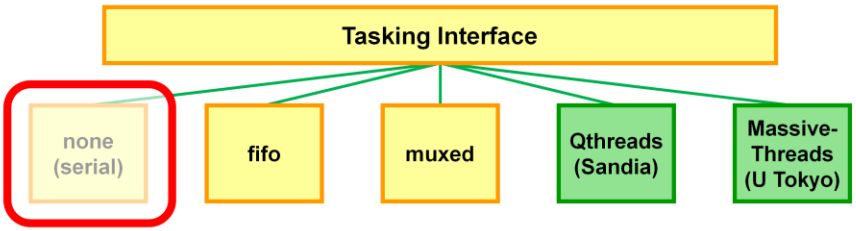
Result:

- printchplenv doesn't show CHPL_THREADS any more:

```
[gbt@host]$ printchplenv
…
CHPL_COMM: none
CHPL_TASKS: fifo
CHPL_LAUNCHER: none
…
```

- If CHPL_THREADS is set, an error message results
- The notion of threading still exists, but only deep in the implementation (and will likely fade away over time)

Remove 'none' Tasking Layer



- Minimalist tasking implementation
- Not regularly tested
- Not used
- Had a small but non-zero maintenance cost
- Got rid of it

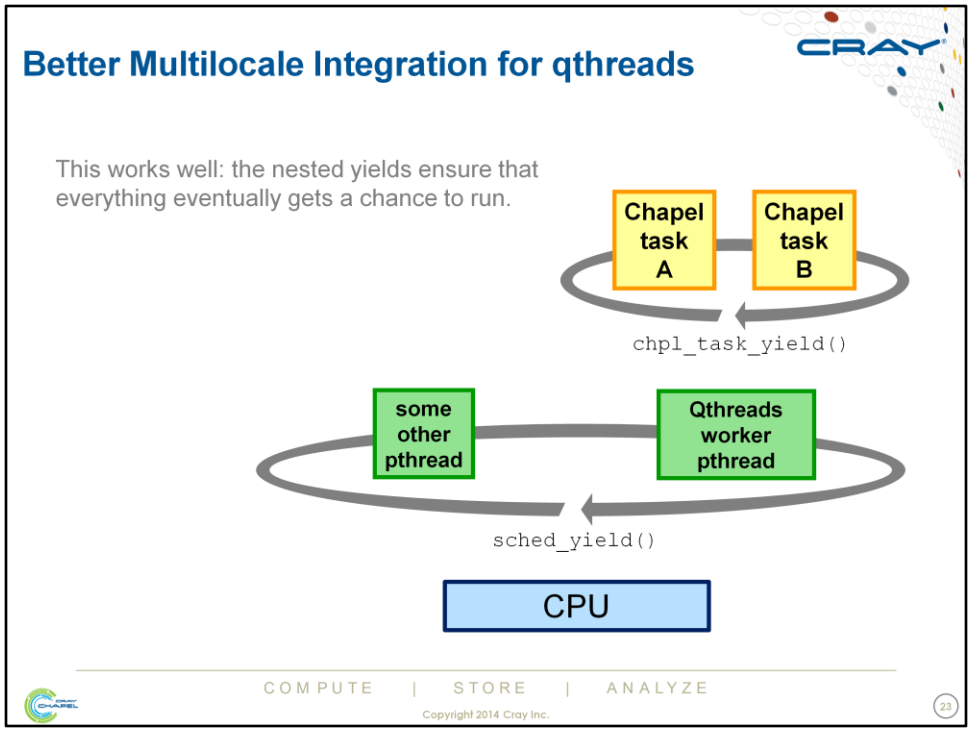
Qthreads Status

- **Bug fixes and other changes**
  - Better multilocale integration
  - Other optimization and tuning
- **Progress towards making qthreads the default tasking layer**

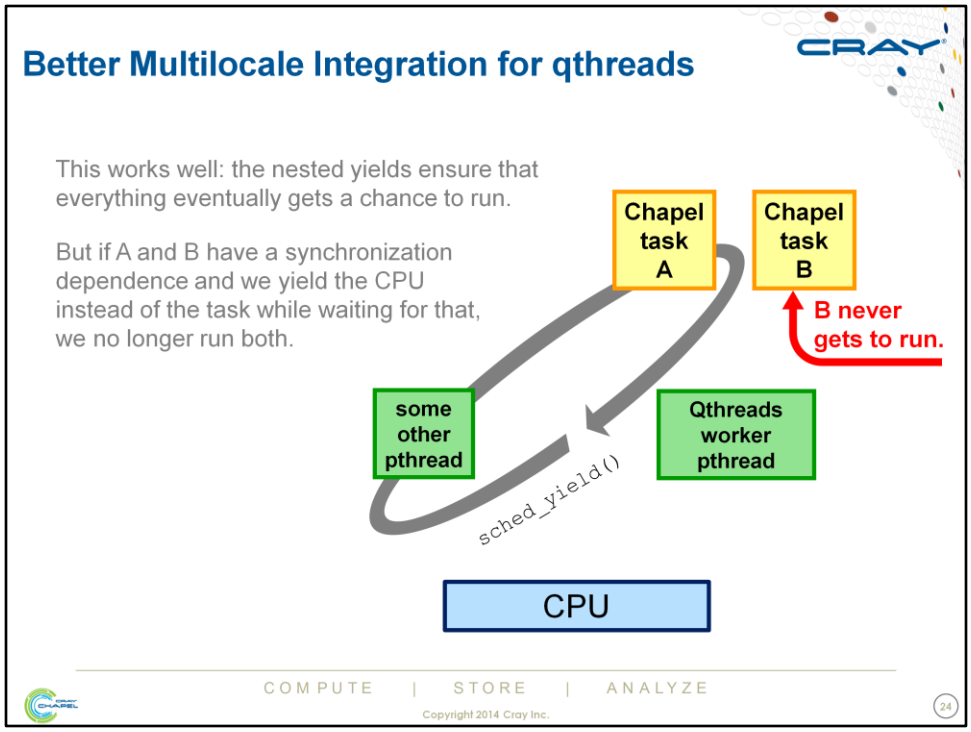
Better Multilocale Integration for qthreads

Background: Multilocale qthreads program were hanging intermittently

- Tasks were being starved



- We're time-sharing execution vehicles at two levels here: Chapel tasks on Qthreads worker pthreads, and pthreads of various kinds on the CPU.



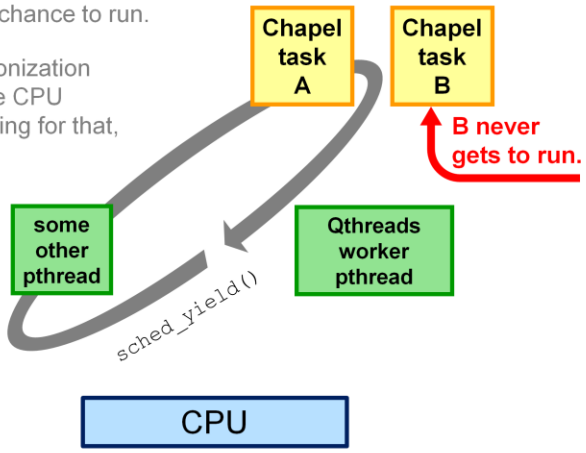
- Since we're yielding the pthread on the CPU instead of the task on the worker, we never go back into the Qthreads code to do a task switch to task B.
- Note that adding resources (more worker pthreads or more CPUs) doesn't solve the basic problem, it just changes how many things you need going on for it to happen.

Better Multilocale Integration for qthreads

This works well: the nested yields ensure that everything eventually gets a chance to run.

But if A and B have a synchronization dependence and we yield the CPU instead of the task while waiting for that, we no longer run both.

We did this in the GASNet comm layer, while waiting for remote task completion.
Result: hang.
Solution: yield the task while waiting in the comm layer instead.



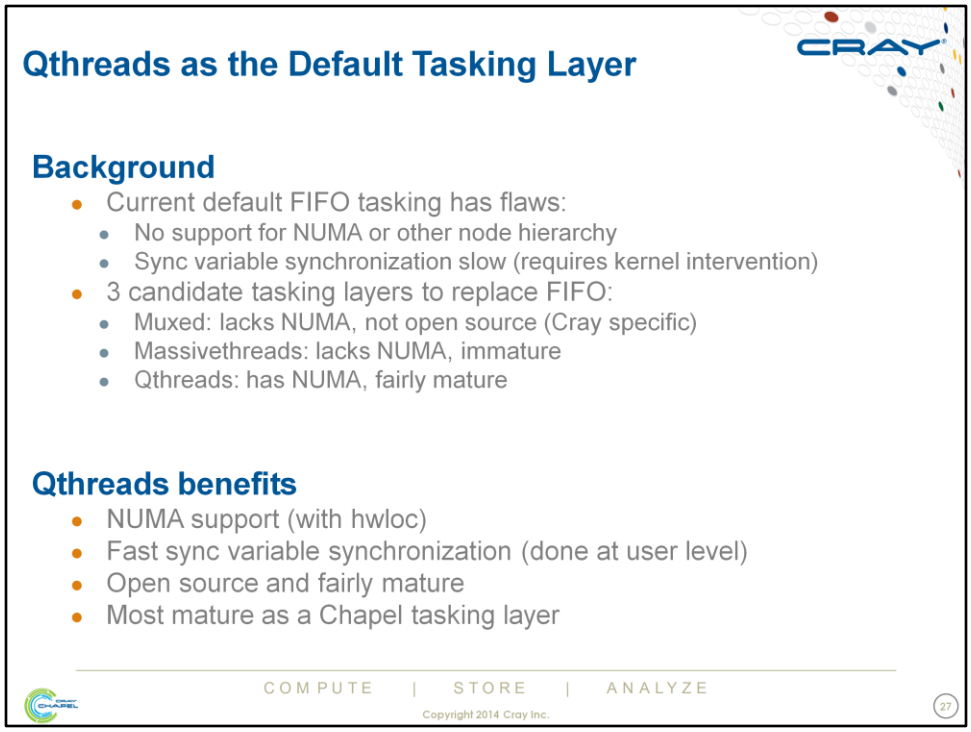
Qthreads Optimization and Tuning

Done In v1.9: Optimize

- Inline several small, frequently used utility functions
- Build with oversubscription enabled to support multilocale testing
- Configure and enable guard pages by default for functional testing, but publicize how to disable for performance testing

Done After v1.9: Tune the defaults

- Assign worker pthreads to cores
  (was assigning workers to hyperthreads, when those were present)
- Default number of workers = number of cores
  (was = number of hyperthreads, due to a bug)



- "Fairly mature": qthreads itself is quite mature, but the qthreads-based Chapel tasking layer is less so. It was at least being minimally tested in nightly testing, though, which was/is not the case for either muxed or massivethreads.

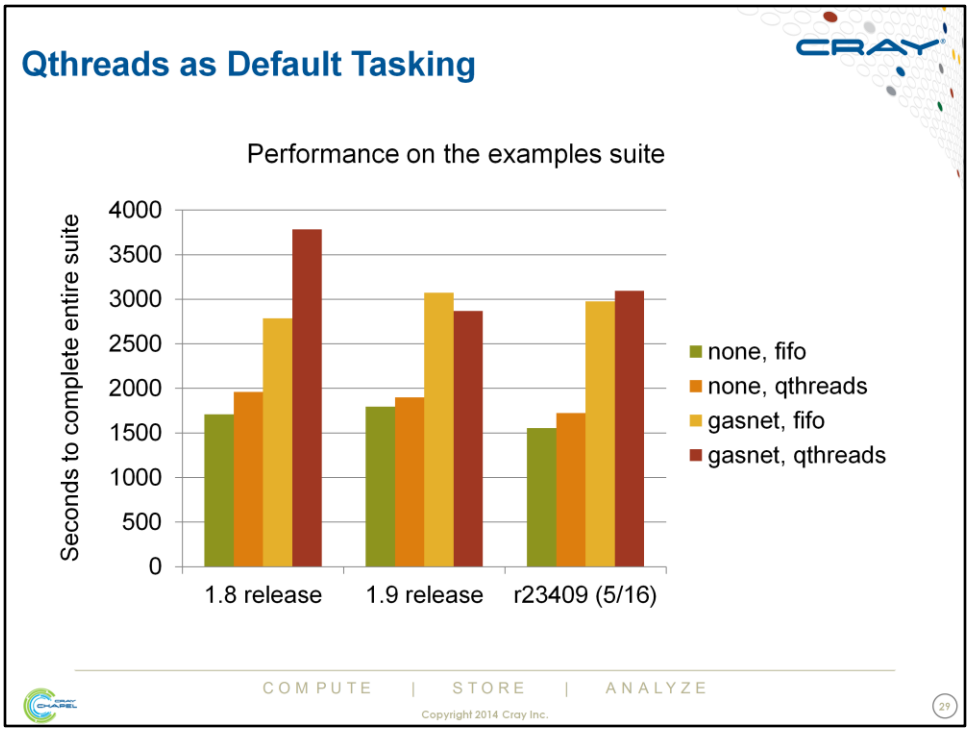
Qthreads as the Default Tasking Layer

Status:

- With comm=none
  - Passes same nightly tests as fifo tasking, and in 7.5 hr vs. 8.5 hr
- With comm=gasnet
  - Passes multilocale tests in nightly testing

Next Steps:

- Expand to full nightly testing
- Characterize performance
- Fix any problems
- Switch!
- Longer term:
  - Tie Chapel sync vars more directly to qthreads sync vars



Note that the bulk of the time here is actually compilation, but it still gives an indication that we're in the ballpark.

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/