Performance Optimizations

Chapel Team, Cray Inc.
Chapel version 1.15
April 6, 2017

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

- Task Spawning Case Study
- Qthreads Hybrid Waiting
- Stack-Allocate Argument Bundles
- Bounded Coforall Optimization
- Task Spawning Summary
- Other Performance Optimizations

Task Spawning Case Study

LCALS: Background

- **LCALS: Livermore Compiler Analysis Loop Suite**
  - Loop kernels designed to measure compiler performance
  - Developed by LLNL
  - https://codesign.llnl.gov/LCALS.php

LCALS Code
Richard D. Hornung
LCALS version 1.0
LLNL-CODE-638939
2013

- **30 kernels total (11 have parallel variants)**

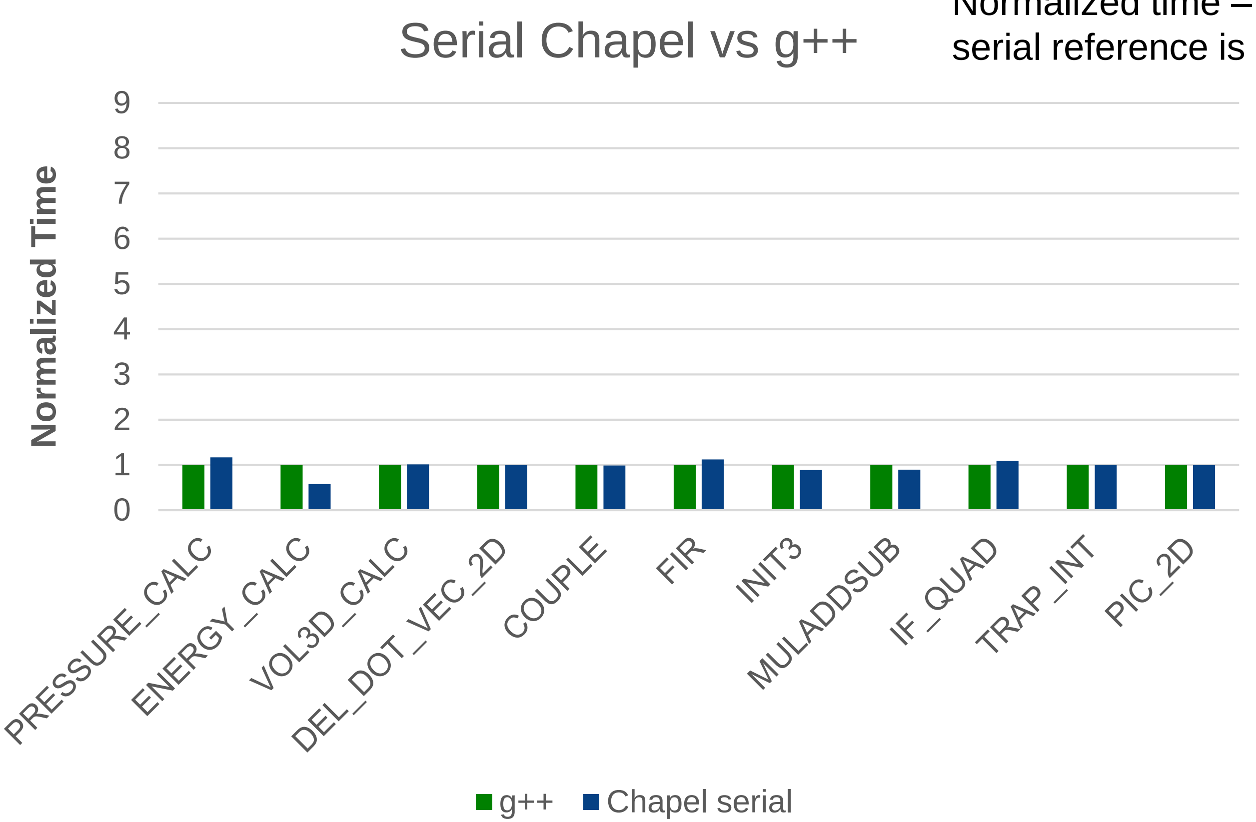
- **Each kernel is run for three sizes (Short, Medium, Long)**

- **Ported LCALS to Chapel in the 1.12 timeframe**
  - first released with Chapel 1.13
  - used to identify performance bottlenecks
  - current port is a pretty direct transliteration

LCALS: Background

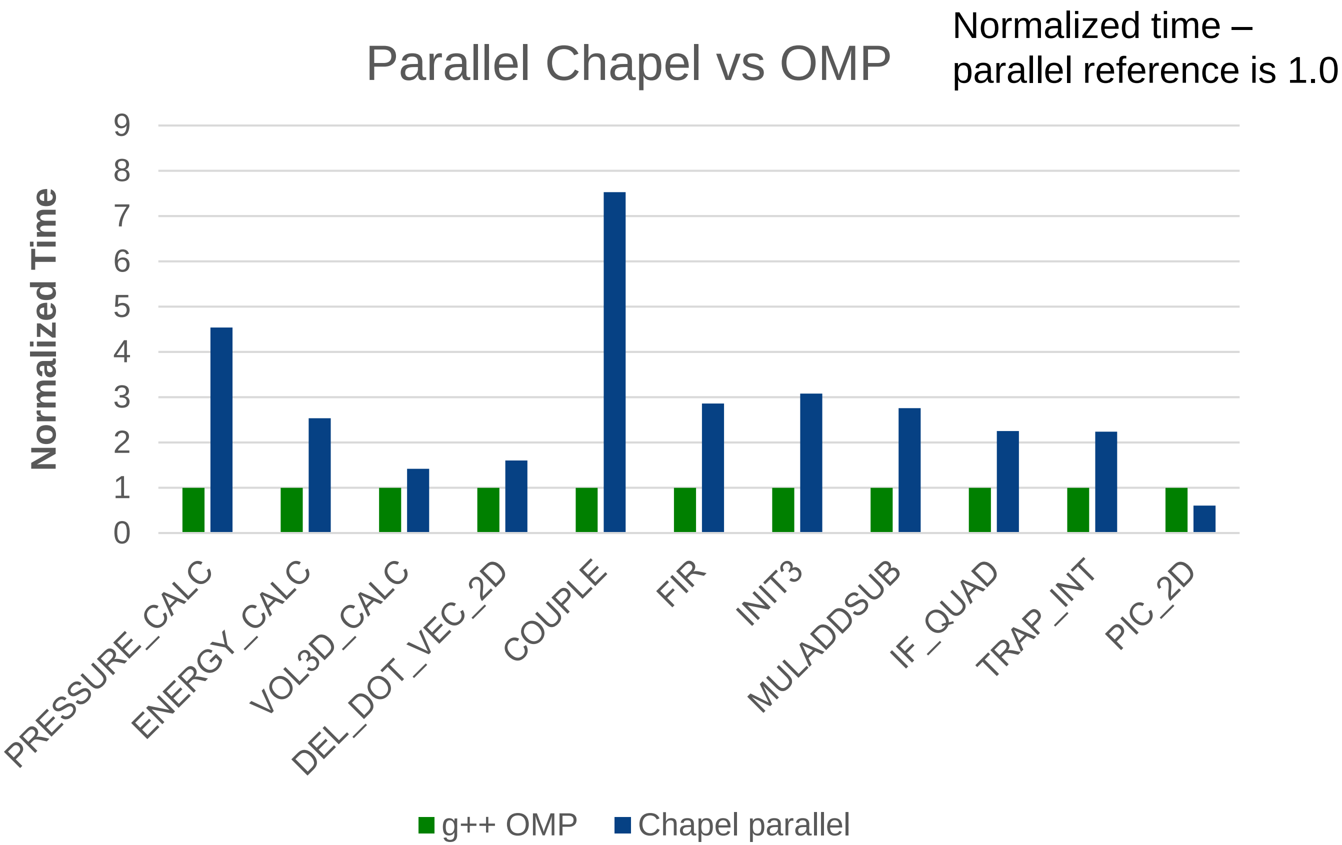
- **Serial performance on par with the reference since 1.14**
  - result of several array optimizations

Normalized time –
serial reference is 1.0



LCALS: Background

- **Parallel variants still lagged behind the reference in 1.14**
  - between 1.5X and 8X slower for long problem size



LCALS: Background

- **Discovered that most of the time is spent spawning tasks**
  - conceptually, kernels perform a simple parallel idiom in a trial-loop
    - e.g. code for the MULADDSUB kernel

Chapel

```
for 0..#num_samples {
  forall i in 0..#len {
    out1[i] = in1[i] * in2[i];
    out2[i] = in1[i] + in2[i];
    out3[i] = in1[i] - in2[i];
  }
}
```

C/C++ OpenMP

```
for (s=0; s<num_samples; ++s) {
  #pragma omp parallel for
  for (i=0 ; i<len ; i++) {
    out1[i] = in1[i] * in2[i];
    out2[i] = in1[i] + in2[i];
    out3[i] = in1[i] - in2[i];
  }
}
```

LCALS: Background

- **Discovered that most of the time is spent spawning tasks**
  - conceptually, kernels perform a simple parallel idiom in a trial-loop
    - e.g. code for the MULADDSUB kernel
  - exacerbated for the “short” problem size

Long Problem Size

```
// Modest num trials, modest trip count
for 0..#12000 {
  forall i in 0..#44217 {
    ...
  }
}
```

Short Problem Size

```
// huge num trials, tiny trip count
for 0..#15000000 {
  forall i in 0..#171 {
    ...
  }
}
```

Task Spawning: Background

- **Decided to focus on improving task spawning speed**
  - created a no-op task-spawn micro-benchmark to investigate
    - Chapel, OpenMP, and native qthreads variants

Chapel

```
for 1..trials do
  forall 1..cores do ;
```

OpenMP

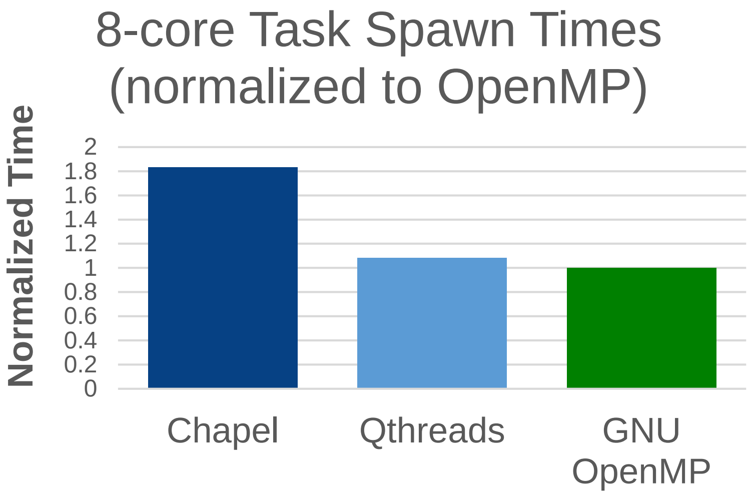
```
for (i=0; i<trials; i++)
  #pragma omp parallel for
  for (j=0; cores; j++) { }
```

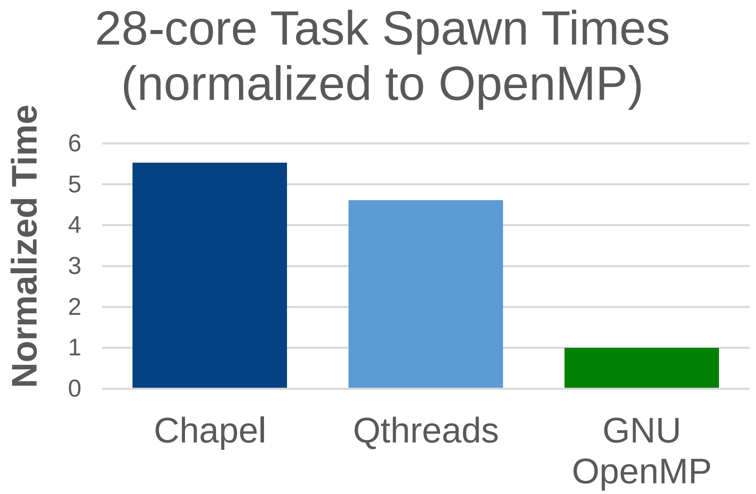
Qthreads

```
for (i=0; i<trials; i++) {
  qthread_incr(&endCount, cores)
  for (j=0; j<cores; j++)
    qthread_fork(decEndCount, ...);

  while (endCount != 0) qthread_yield();
}
```

Task Spawning: Background

- **Results of task spawning micro-benchmark with 1.14:**
  - performance wasn't too far off for lower core-count machines
    - run on an 8-core (16 HT) Nehalem node, with gcc 6.3
    - Chapel within 80% of OpenMP, qthreads within 10%
  - performance was further off of OpenMP for high core-count machines
    - run on a 28-core (56 HT) Broadwell node, with gcc 6.3
    - Chapel was ~6x slower than OpenMP, qthreads was ~5x slower





Task Spawning: Background

- **Task spawning performance goals for this release:**
  - determine if qthreads can be competitive with OpenMP
    - if not, need to explore other tasking layer options
  - minimize tasking overhead that Chapel introduces
    - minimize overhead introduced by the compiler, modules, runtime shim

Qthreads Hybrid Waiting

Hybrid Waiting: Background

- **Idle workers have 2 mechanisms to wait for work**
  - set at qthreads configure time:
    - spinwait (busy-waiting -- continuous spinloop)
    - condwait (sleep -- uses a pthread condwait)

- **Our default wait-policy was condwait**
  - chosen while investigating qthreads as our default over fifo
    - spin-waiting killed performance of single/low-threaded codes
  - condwait hurt performance of some highly-parallel code
    - but not dramatically, investigation done on an 8-core machine

- **Determined that pure condwait hurts task-spawn speed**
  - significant penalty on high core-count machines
  - needed to implement a new waiting mechanism

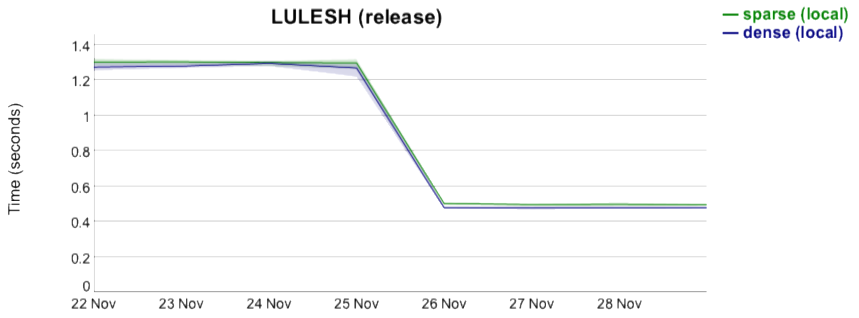
Hybrid Waiting: This Effort

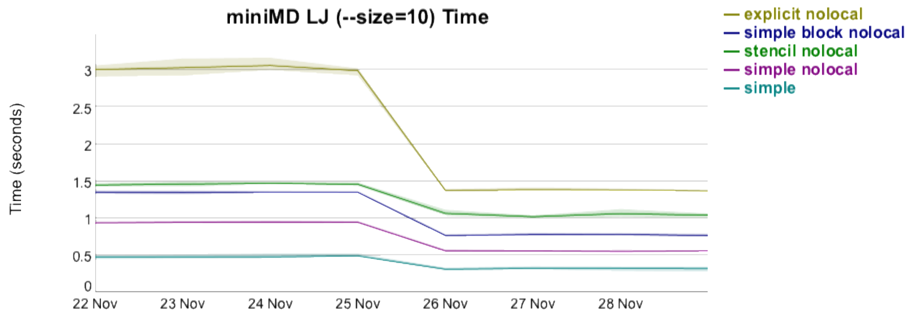
- **Implemented a hybrid spin/condwait scheme**
  - conceptually simple
    - spin for some amount of time before giving up and sleeping
  - difficult part was choosing spin duration

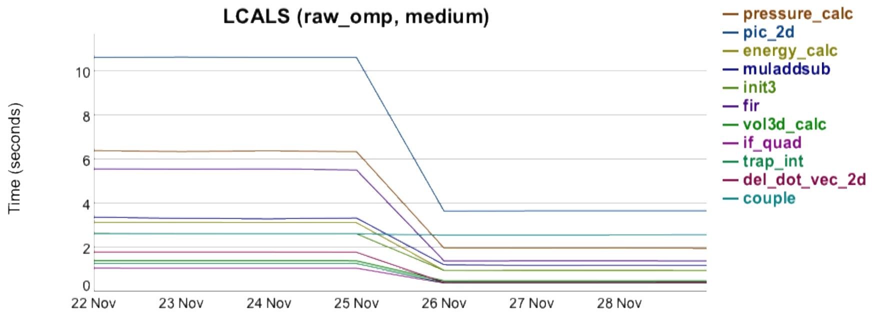
- **Current strategy: spin 300,000 times before sleeping**
  - opted for a spincount-based strategy (instead of a time-based)
    - low overhead, easy to implement
  - experiments showed 100k-300k provided best task-spawn perf
  - went with the upper bound, since Chapel is a parallel language
    - also happens to match GNU OpenMP spincount policy
  - on Crays, default is bumped to 3,000,000
    - applications that warrant a Cray are likely to be very parallel

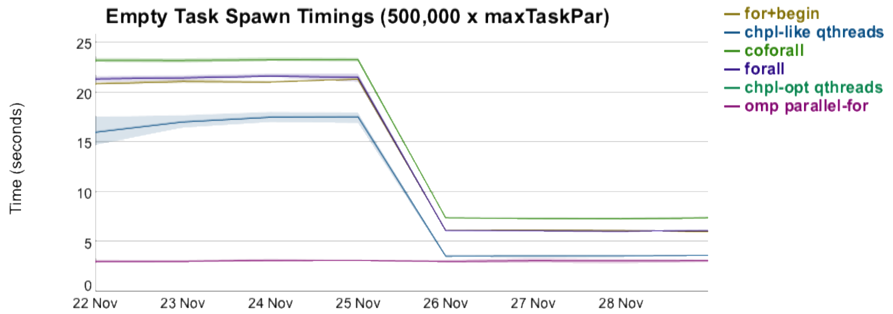
Hybrid Waiting: Impact

- **Resulted in significant performance improvements**
  - particularly for single-locale programs



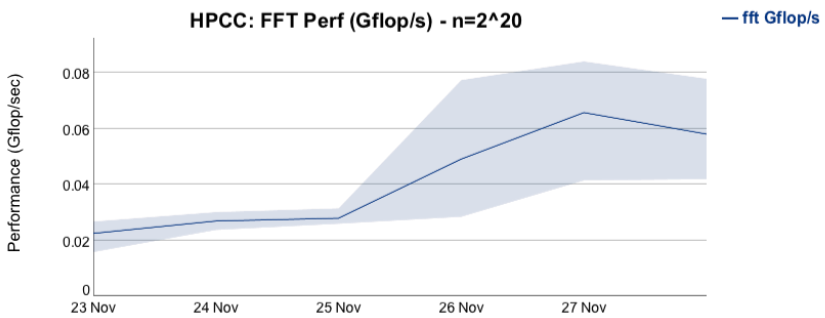


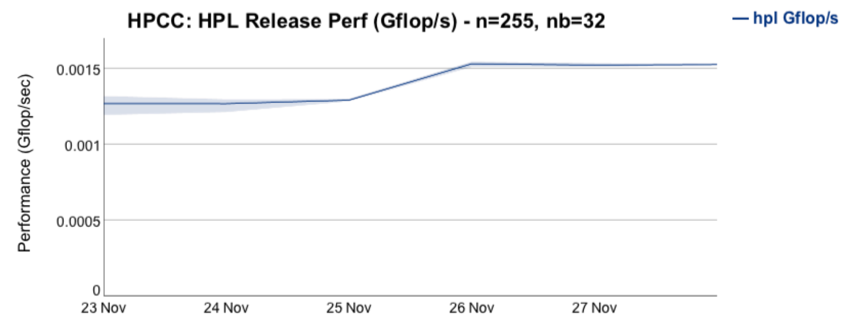


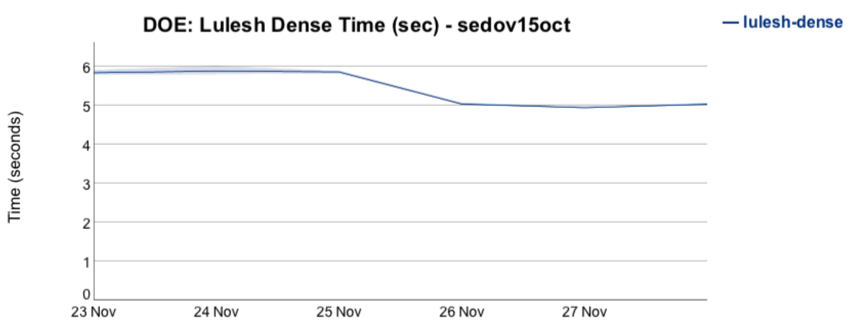


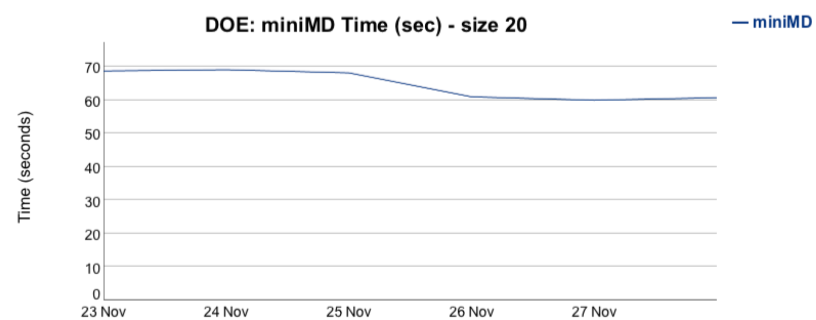
Hybrid Waiting: Impact

- **Resulted in significant performance improvements**
  - some nice multi-locale improvements as well
    - bumping spincount on Crays further improved fft/hpl (not shown here)



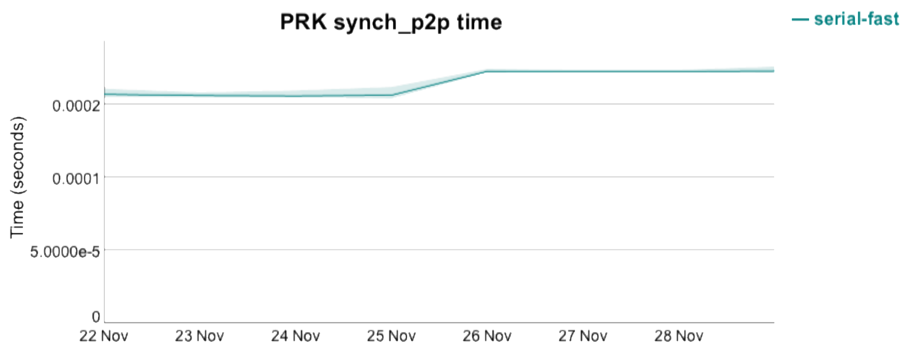


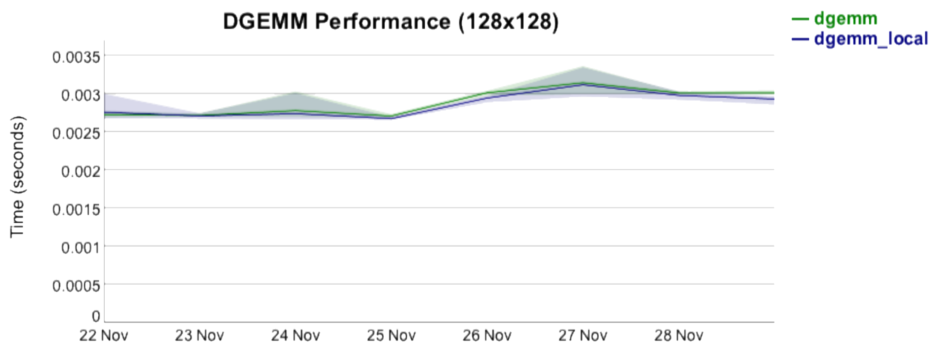




Hybrid Waiting: Impact

- **Some minor regressions**
  - for short-lived minimally-parallel benchmarks
    - acceptable in light of improvements on highly-parallel benchmarks





Hybrid Waiting: Status and Next Steps

Status:

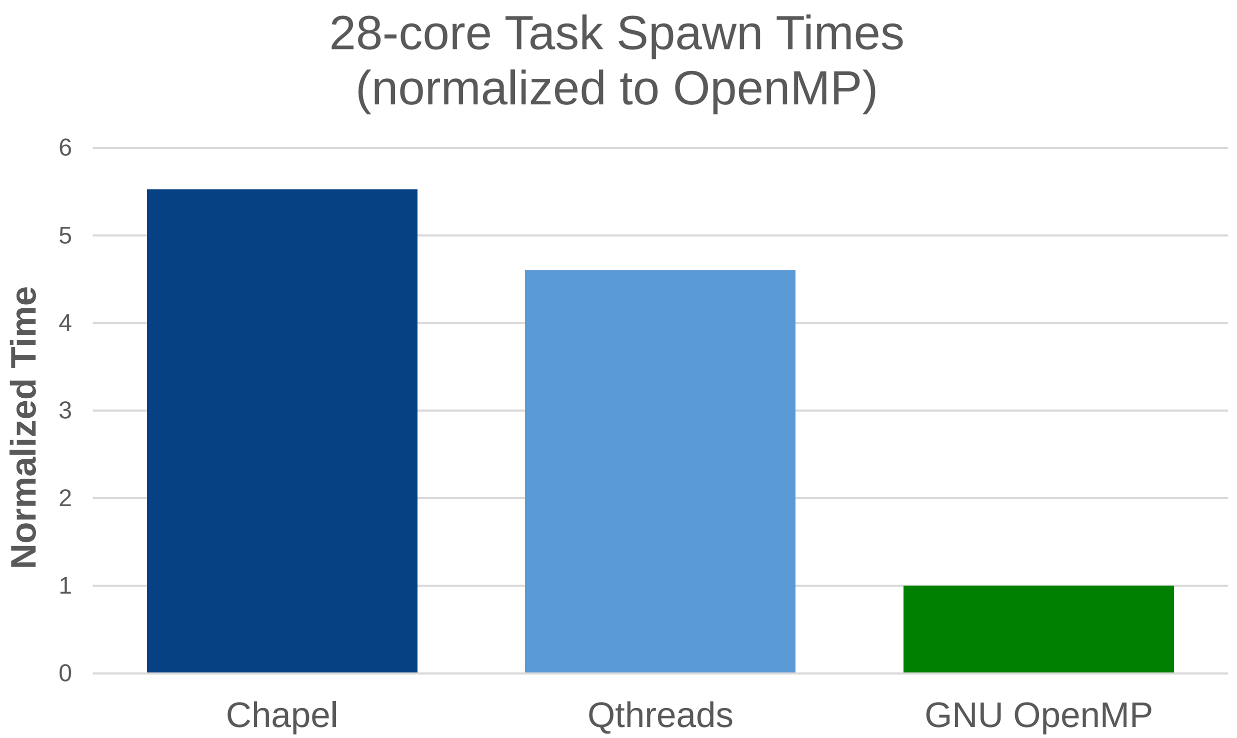
- Hybrid spin/condwait scheme implemented
  - contributed upstream
- Significantly improved speed of task-spawning
  - without seriously hurting serial applications

Next Steps:

- Add a friendlier user-facing policy mechanism
  - e.g. WAIT_POLICY={active, passive} vs. SPINCOUNT=30000
- Implement spin-wait policy across qthreads schedulers
  - currently nemesis (flat) only, need to expand to distrib (numa)
- Explore time-based spinning strategies
  - may offer a more “portable” balance across architecture
  - Intel’s OpenMP runtime uses a time-based strategy

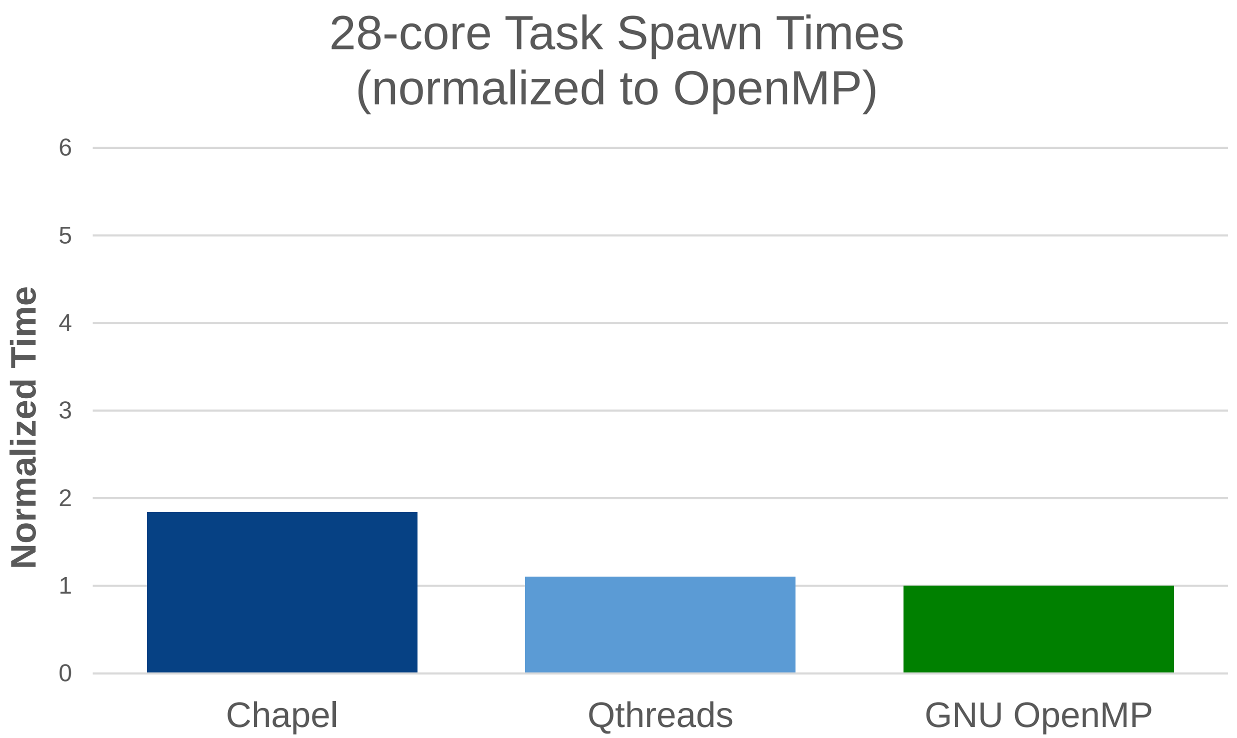
Hybrid Waiting: Task Spawning Impact

- **Previously Chapel and qthreads lagged behind OpenMP**



Hybrid Waiting: Task Spawning Impact

- **Hybrid waiting significantly closes the gap with OpenMP**
  - 3x faster: Chapel within ~80% of OpenMP, qthreads within 10%
    - next step is to reduce Chapel's overhead



Stack-Allocate Argument Bundles

Stack Arg Bundles: Background

- **on/begin/cobegin/coforall create argument bundles**
  - on-statement and task bodies are outlined
  - runtime calls outlined function in new task or on remote locale
  - argument bundles store variables to be passed to the outlined function

- **Generated code heap-allocated argument bundles**
  - adding overhead to tasks, on-statements

- **Heap allocation was redundant**
  - fifo allocated a task descriptor and could store arguments there
  - qthreads already had the ability to copy arguments into new task
  - comm layers needed to copy bundle in some cases
    - caller could free argument bundle immediately
    - but comm/tasking could delay call to outlined function & use of bundle

Stack Arg Bundles: This Effort

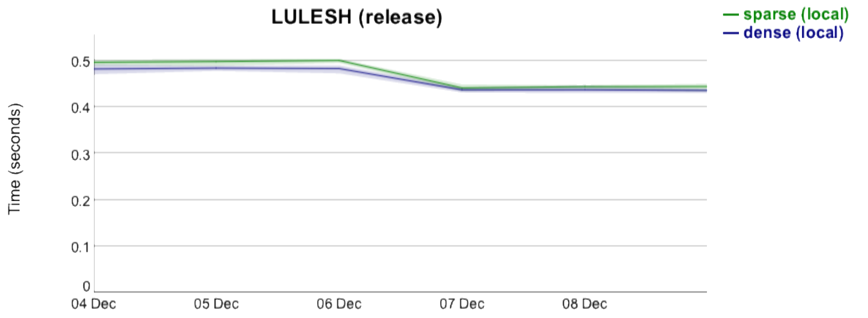
- **Adjust compiler to stack-allocate argument bundles**

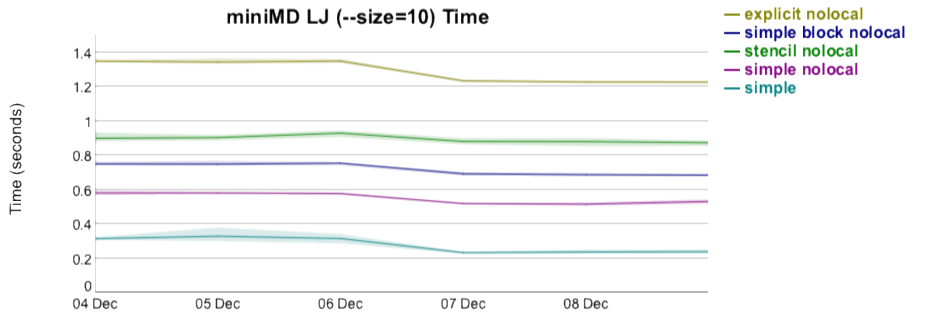
- **Further minimize copying within the runtime**
  - runtime often needs to add information to argument bundle
    - e.g. which function to run on remote locale
  - this should be contiguous in memory with argument bundle
    - e.g. to send in one network message
  - solution: include a struct for runtime information in the bundles…
    … to completely avoid heap-allocating or copying in many cases

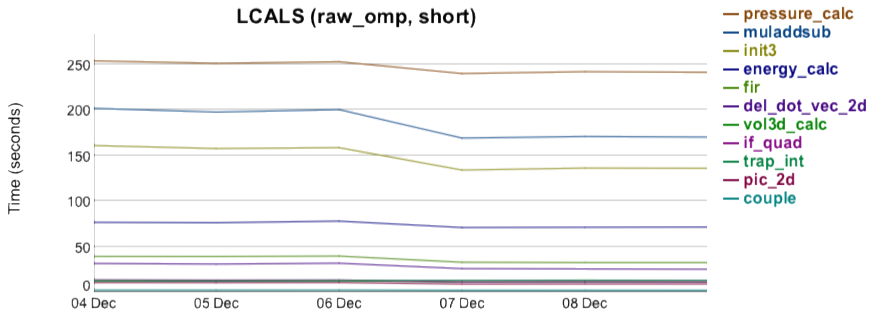
- **Adjust runtime to work with stack-allocated arg bundles**
  - including all tasking and communication layers
    - fifo, qthreads, muxed, ugni, gasnet
  - while there, minimized heap allocation calls in tasking & comms

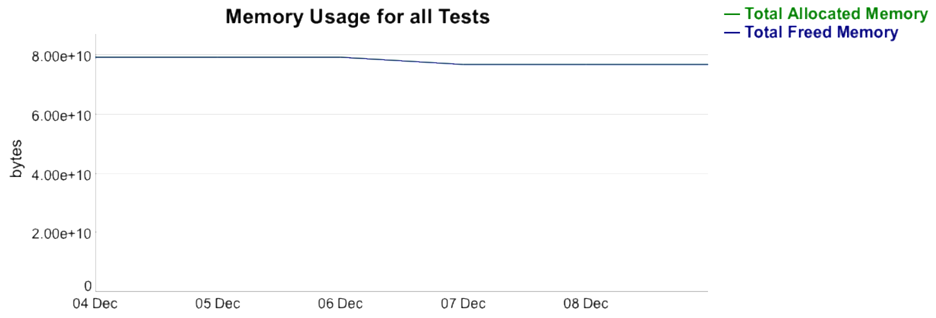
Stack Arg Bundles: Impact

- **Resulted in single-locale speedups**
  - and a decrease in the total amount of memory allocated



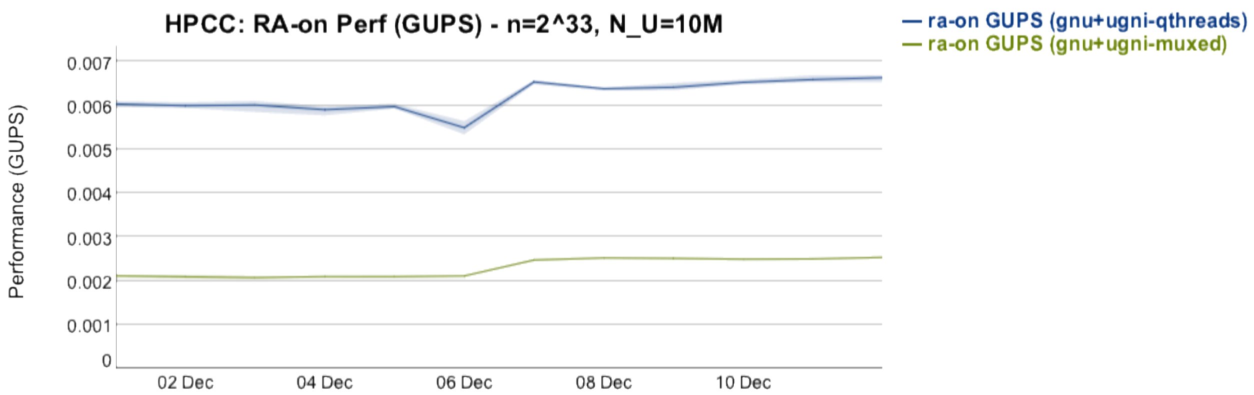






Stack Arg Bundles: Impact

- **10-15% improvement for multi-locale RA-on**
  - RA-on creates many on statements
  - stack-allocating reduced on statement overhead



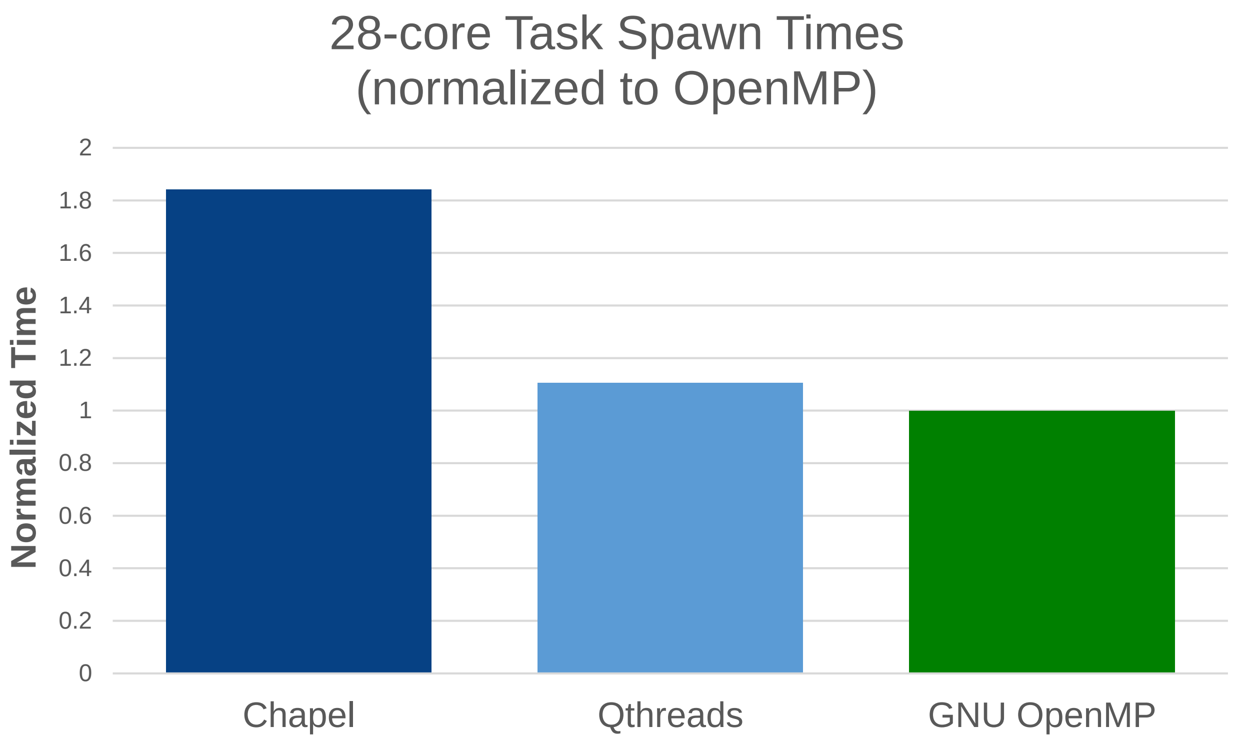
Stack Arg Bundles: Next Steps

Next Steps:

- Remove other unnecessary heap allocation in generated code
- Consider a heap-to-stack compiler optimization

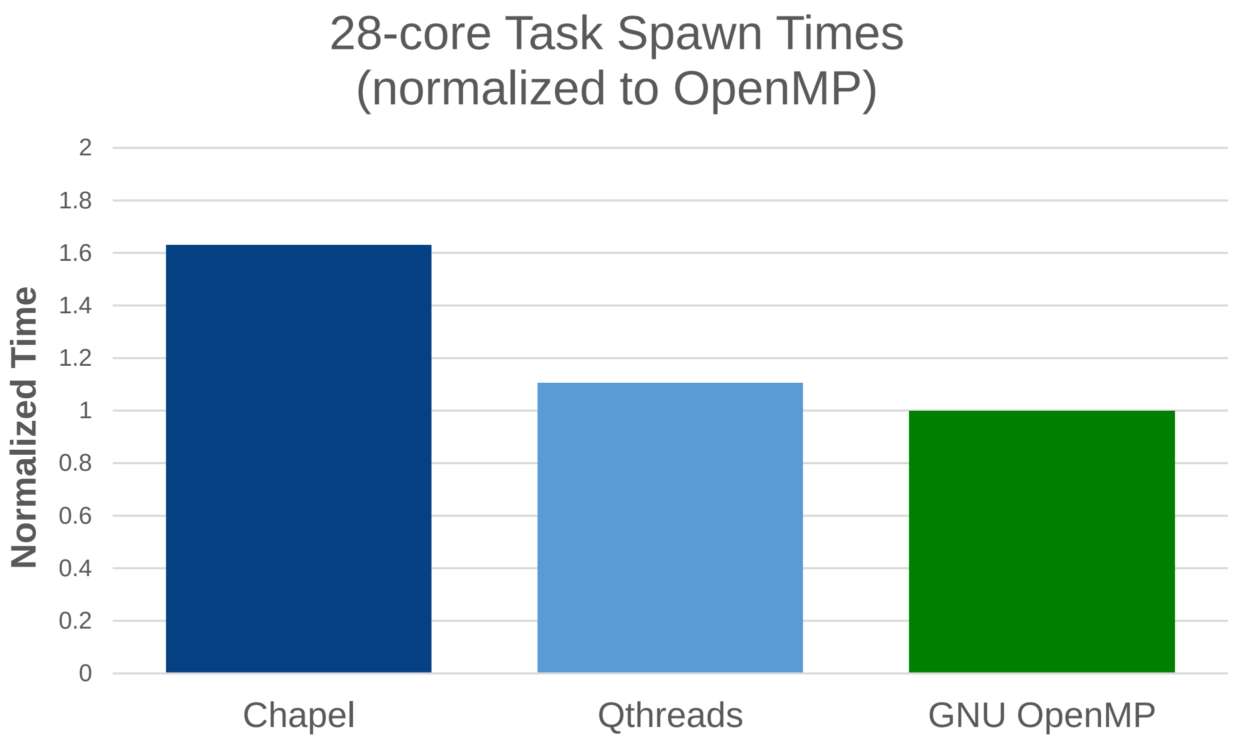
Stack Arg Bundles: Task Spawning Impact

- Chapel was within ~80% of OpenMP



Stack Arg Bundles: Task Spawning Impact

- **Stack allocating arg bundles reduces Chapel's overhead**
  - now within 60% of OpenMP



Bounded Coforall Optimization

Bounded Coforall: Background

- Coforalls are transformed by the compiler, from:

```
coforall i in 1..10 { body(); }
```

roughly into:

```
var endCount: atomic int;
for i in 1..10 {
  endCount.add(1);                    // note: incrementing once per task
  spawnTask(bodyWrapper, endCount);
}
endCount.waitFor(0);

proc bodyWrapper(endCount) {
  body();
  endCount.sub(1);
}
```

Bounded Coforall: This Effort

- **Minimize end-count manipulation for “bounded” coforalls**
  - “bounded” coforalls have a known trip-count (range/domain/array)

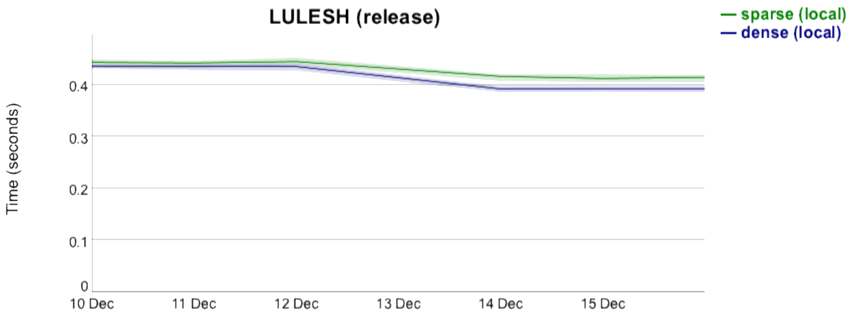
```
coforall i in 1..10 { body(); }
```

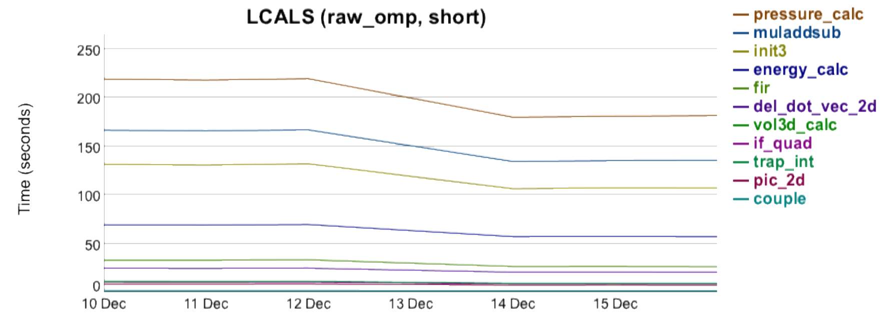
now roughly converted to:

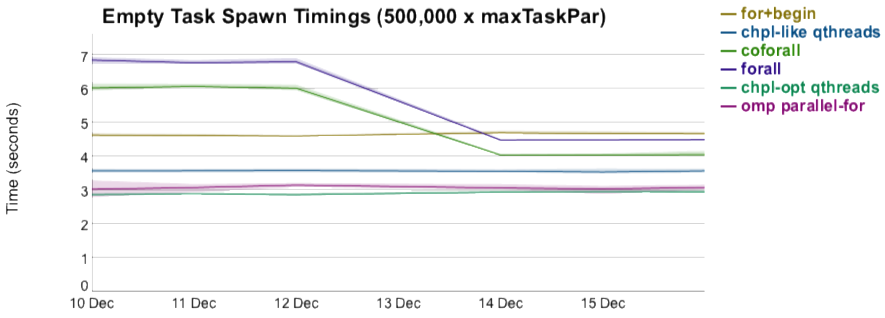
```
var tmpIter = 1..10;
var numTasks = tmpIter.size
var endCount: atomic int;

endCount.add(numTasks);   // single atomic op vs. op per task
for i in tmpIter {
  spawnTask(bodyWrapper, endCount);
}
endCount.waitFor(0);

proc bodyWrapper(endCount) { /* same as before */ }
```

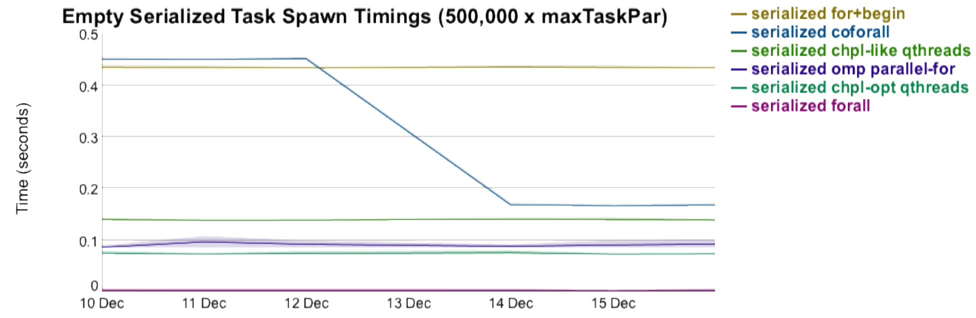
Bounded Coforall: Impact

- **Improved performance for many single-locale benchmarks**
  - no known regressions









Bounded Coforall: Status and Next Steps

Status:

- optimized coforalls over types with a known trip-count
  - currently ranges/domains/arrays
  - only done for "local" coforalls currently

Next Steps:

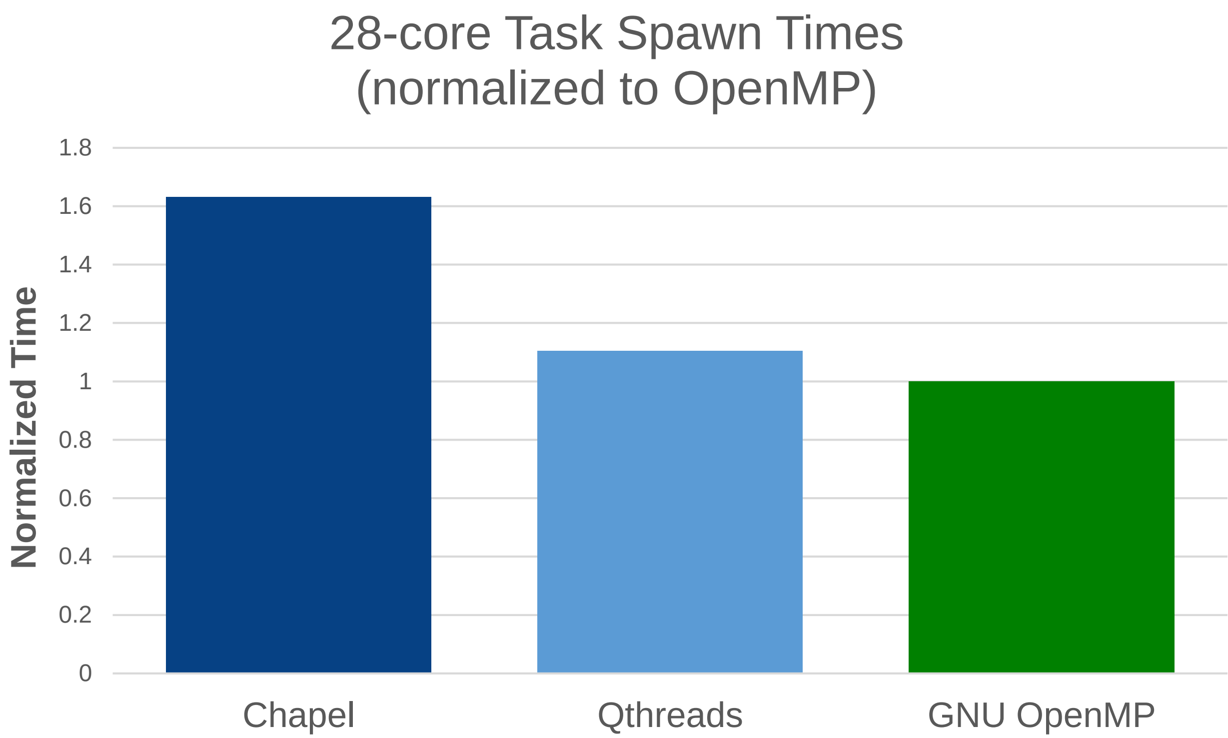
- Implement optimization for "remote" coforalls

```
coforall i in 1..10 do on Locales[i] { body(); }
```

- Add support for "bulk" spawning to our runtime interface
  - single runtime call to spawn all tasks instead of call per task
  - would further minimize overhead introduced by Chapel
- Add support for "bulk" spawning to qthreads
  - likely that this would permit qthreads optimizations

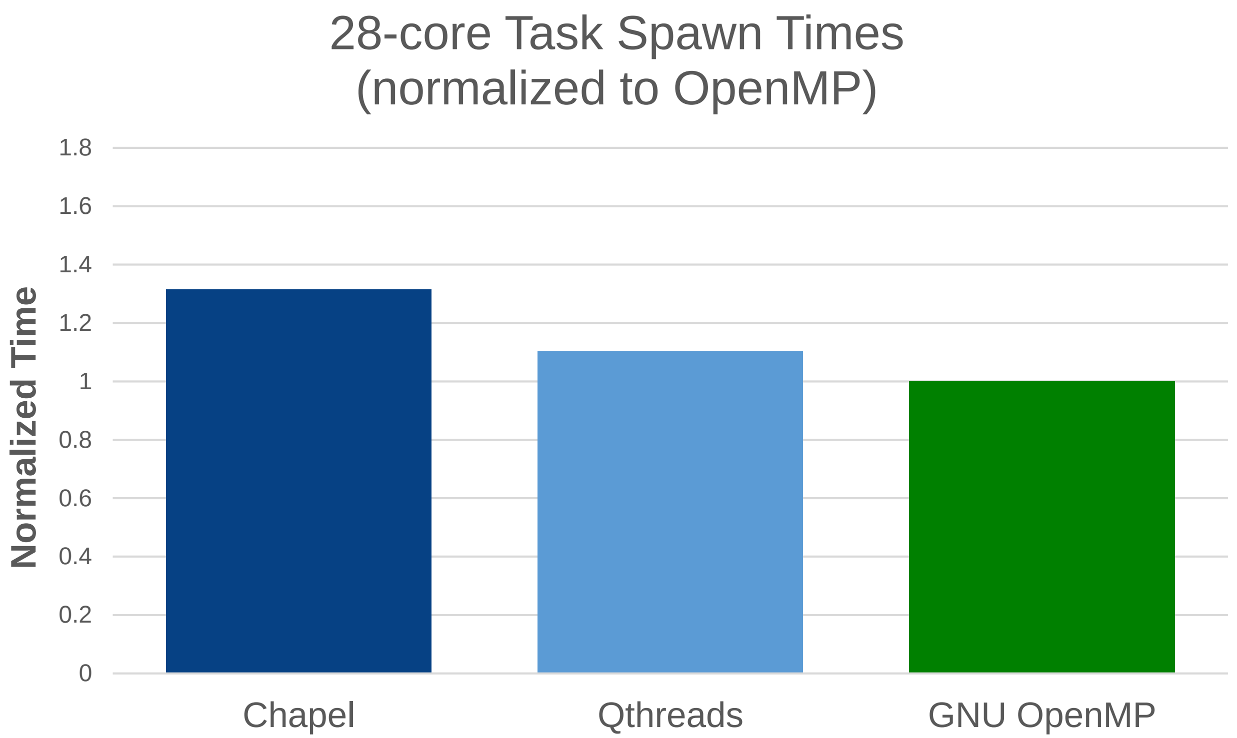
Bounded Coforall: Task Spawning Impact

- Chapel was within ~60% of OpenMP



Bounded Coforall: Task Spawning Impact

- **Coforall optimization further reduces Chapel's overhead**
  - now within 30% of OpenMP



Task Spawning Summary

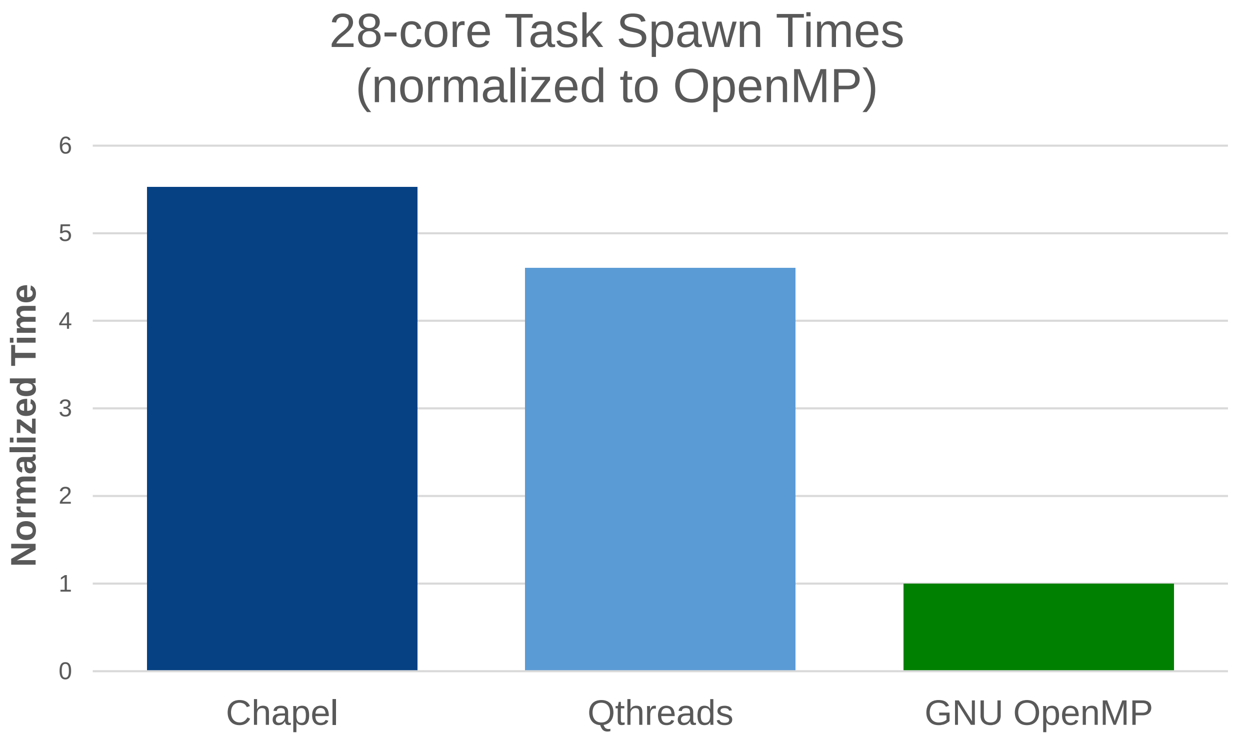
Task Spawning: Summary

- **Task spawning investigation led to several optimizations**
  - implemented a hybrid spin/condwait scheme in qthreads
  - moved argument bundles from the heap to the stack
  - minimized task counting overhead of bounded coforalls

- **Optimizations had a significant impact**
  - large performance improvements for many benchmarks
    - LCALS, MiniMD, Lulesh, SSCA2, and many others
  - task spawning is around 4 times faster now

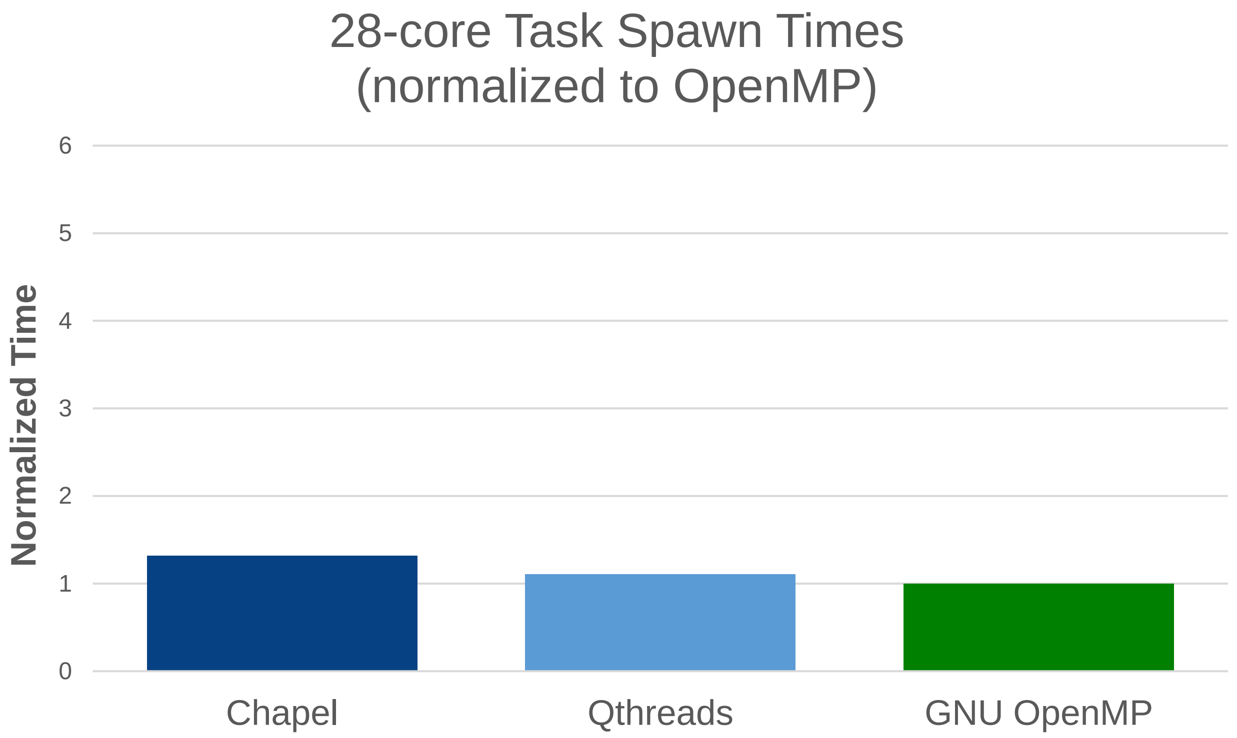
Task Spawning: Performance Impact

- **1.14 task spawning performance**
  - over 5x slower than GNU OpenMP



Task Spawning: Performance Impact

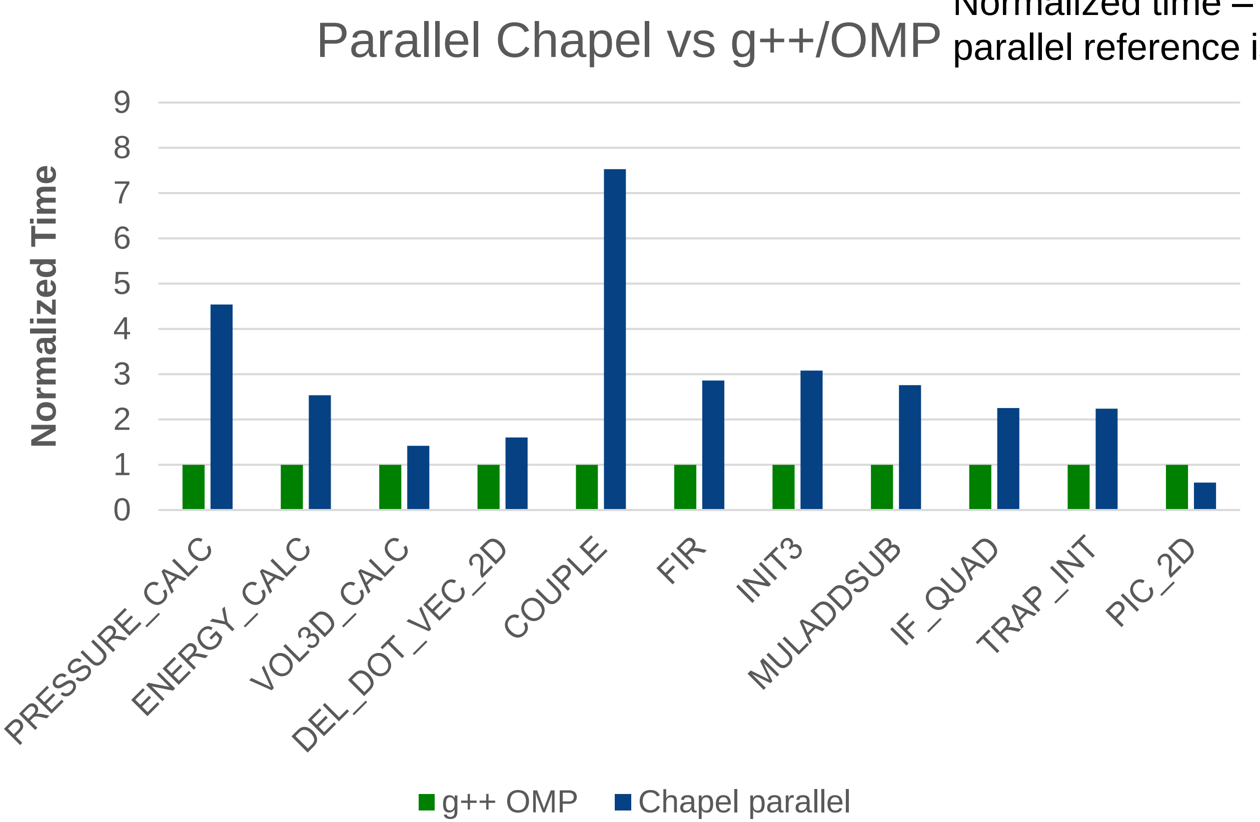
- **1.15 task spawning performance**
  - within 30% of GNU OpenMP for 28-core machine
  - within 5% for 8-core machine (not shown here)



Task Spawning: LCALS Performance Impact

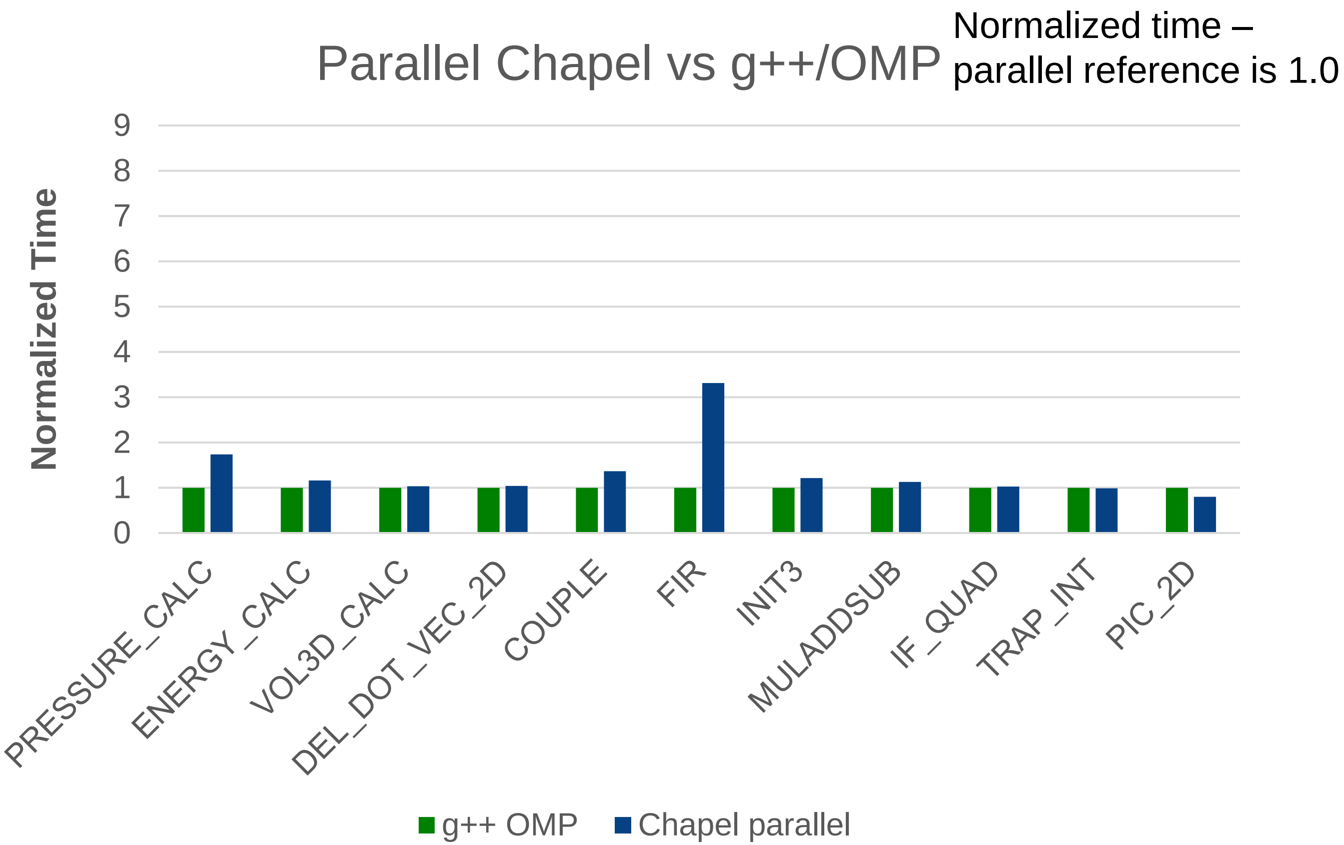
- **1.14 LCALS performance**

Normalized time – parallel reference is 1.0



Task Spawning: LCALS Performance Impact

- **1.15 LCALS performance**
  - ~3-4x speedup: on par or very close to reference for most kernels



Task Spawning: Next Steps

- **Continue to optimize task spawning**
  - minimize Chapel's overhead
    - add a bulk spawning interface to the runtime shim
  - further optimize qthreads
    - add a bulk spawning interface
  - explore alternatives to qthreads?
    - Argobots, Intel's OpenMP runtime, OCR
  - explore different task joining mechanisms
    - alternatives to our current atomic "end count"

- **Add additional task-spawning benchmarks**
  - add a stream-like variant
  - add nested parallelism variants

Other Performance Optimizations

Other Performance Optimizations

- Optimized iteration over 1D strided arrays
- Improved loop invariant code motion optimization
- Improved remote-value-forwarding optimization
- Improved performance of casting strings to numeric types
- Optimized <~> to avoid unnecessary reference counting

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