Benchmarks and Performance Optimizations

Chapel Team, Cray Inc.
Chapel version 1.17
April 5, 2018
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

● **Ugni Improvements**
  ● Extend and Register the Heap Dynamically
  ● Nonblocking Active Message Responses
  ● Comm Domain Limit
  ● Avoid ‘Bus Error’ Messages
  ● Scalability Improvements

● **ISx Improvements**
  ● Scalable Barrier
  ● Park the Main Process
  ● Reduce Progress Thread Interference

● **Meltown and Spectre Impact**

● **Reductions in Memory Leaks**

● **Other Performance Optimizations**
Ugni Improvements
Extend and Register the Heap Dynamically
Dynamic Heap: Background and Effort

**Background:** NIC-registered heap had unfortunate limitations

- **Performance**
  - Poor NUMA memory affinity, because registration pins to NUMA node 0
  - Up-front heap creation and registration increased program startup cost

- **Ease of use**
  - Fixed-size heap cannot be extended if not large enough
  - Pushed default to err toward too-large
  - If default nevertheless too small, computing a better size was impractical

**This Effort:** Extend and register heap dynamically

- Reuses infrastructure added in 1.16 for dynamic registration of arrays
Dynamic Heap: Positive Impact

- Faster startup

- Better non-array NUMA affinity (don’t have a specific test)

- Improved usability: no need to estimate max heap size
Dynamic Heap: Negative Impact

- Two performance regressions, not yet understood
Dynamic Heap: Next Steps

- Look into FFT and HPL performance regressions

- Could/should we do this in other configs with registration?
  - Explore options for gasnet-aries
Nonblocking Active Message Responses
AM Responses: Background

- **Active Message handlers slowed by response overhead**
  - Waited for network to acknowledge completion responses
  - Added 1-2 microseconds (i.e., 1 network round trip) per AM request handle

![Diagram showing AM request and response flow](image)
AM Responses: This Effort

- Use nonblocking PUTs for AM responses
  - Begin handling next request as soon as previous response is sent
  - Don’t wait for response ACKs, just consume them as they arrive

```
- originators  
  request  
  request  
  request

- network
  handle
  handle
  handle
  handle

- AM handler
  AM request: originator PUT, network ACK
  AM response: handler PUT, network ACK
```
AM Responses: Impact

- Performance improvements for AM heavy benchmarks
Comm Domain Limit
Comm Domain Limit

**Background:** Limited to at most 30 GNI comm domains on XC
- Legacy code from Gemini; Aries hardware limit is 128

**This Effort:** Raise limit on XC to 120 comm domains
- Can now make effective use of more than 30 cores

**Impact:**

ra-atomics with XC-16 perf settings on 36-core compute nodes

![Bar chart comparison of GUPS before and after](chart.png)
Avoid ‘Bus Error’ Messages
Avoid ‘Bus Error’ Messages

**Background:** Running out of memory caused ‘Bus Error’ halt
  ● Result of SIGBUS signal if page allocation failed when first touched
  ● Side effect of allocation technique that improved NUMA locality

**This Effort:** Emit usual “out of memory” message instead
  ● Only for SIGBUS due to touching new memory, not others

**Impact:** Improved ease-of-use
  ● Removes an awkward special case and associated documentation
Scalability Improvements
Scalability: Background

- 1.16 had significant performance improvements
  - But there were a few unknown performance mysteries
  - Stream Global scalability was worse than Stream EP

![Graph showing performance of STREAM](image-url)

- Performance of STREAM
  - GB/s vs Locales
  - Lines for different versions (1.15, 1.16) and choices (EP, Global)
  - Ref vs 1.15 EP vs 1.15 Global vs 1.16 EP vs 1.16 Global
  - Arrow indicating better performance
Scalability: Background

- 1.16 had significant performance improvements
  - But there were a few unexpected performance mysteries
  - PRK Stencil scalability lagged behind reference (but gn-mpi on par with ref)
Scalability: Background

- Remote task spawning included in Global Stream timers
  - EP spawns to all locales before starting timers

```plaintext
Stream EP
coforall loc in Locales do on loc {
  var A, B, C: [1..m] elemType;
  initVectors(B, C);

  startTimer();

  forall (a, b, c) in zip(A, B, C) do
    a = b + alpha * c;

  stopTimer();
}

Global Stream
const ProblemSpace = {1..m} dmapped ...;
var A, B, C: [ProblemSpace] elemType;
initVectors(B, C);

startTimer();
forall (a, b, c) in zip(A, B, C) do
  a = b + alpha * c;

stopTimer();
```

- Remote task spawning included in PRK Stencil as well
Scalability: Background

- Remote coforalls are transformed by the compiler, from:

  ```
  coforall loc in Locales do on loc { body(); }
  ```

  roughly into:

  ```
  var endCount: atomic int;

  endCount.add(Locales.size);
  for loc in Locales {
    executeOnNB(loc, bodyWrapper, endCount);
  }

  endCount.waitFor(0);

  proc bodyWrapper(endCount) { body(); endCount.sub(1); }
  ```
Remote coforalls are transformed by the compiler, from:

```chpl
coforall loc in Locales do on loc { body(); }
```

roughly into:

```chpl
var endCount: atomic int;

endCount.add(Locales.size);
for loc in Locales {
    // inlining the call to executeOnNB(loc, bodyWrapper, endCount):
    chpl_comm_initiate_remote_fork(loc, ACK, ...);
    while (!received(ACK)) {
        chpl_task_yield(); // problem – yielded before all remote tasks started
    }
}
endCount.waitFor(0);

proc bodyWrapper(endCount) { body(); endCount.sub(1); }
```
Scalability: This Effort

- Avoid yielding when doing NB remote forks under ugni

```c
coforall loc in Locales do on loc { body(); }
```

now roughly transformed into:

```c
var endCount: atomic int;

endCount.add(Locales.size);
for loc in Locales {

    chpl_comm_initiate_remote_fork(loc, ACK, ...);
    while(!received(ACK)) {} // network round trip wait before next iteration

}
endCount.waitFor(0);

proc bodyWrapper(endCount) { body(); endCount.sub(1); }
```
**Scalability: Impact**

- Significantly improved scalability under ugni
  - Stream Global scaling close to Stream EP up to 256 nodes

![Performance of STREAM Graph](image-url)
Scalability: Impact

- Significantly improved scalability under ugni
- PRK Stencil performance is on par with reference up to 256 nodes

![Performance of Stencil PRK](chart)

Performance of Stencil PRK

- ref
- 1.16
- 1.17.1

GFlops/s vs Locales

16 32 64 128 256

0 2000 4000 6000 8000 10000 12000

better
Scalability: Next Steps

- **Scalability is good for up to 256 locales**
  - At higher scales (1024 shown below), scalability starts to suffer

![Performance of STREAM graph](image)

- **Locales**
- **GB/s**

1.17.1 EP  |  1.17.1 Global

Copyright 2018 Cray Inc.
Scalability: Next Steps

● **Same interface is used to create 1 task or 1 million tasks**
  ● Great for code reuse, but has scalability bottlenecks
    ● task spawning is serialized

```c
endCount.add(Locales.size);
for loc in Locales {
  chpl_comm_initiate_remote_fork(loc, ACK, ...);
  while(!received(ACK)) {} // network round trip wait before next iteration
}
endCount.waitFor(0);
```

● **Introduce a bulk spawning interface**
  ● Amenable to many optimizations
    ● Initiate multiple tasks at once, instead of one at a time
    ● Use an “end count” mechanism optimized for the network
    ● Do tree-based spawning instead of a 1-to-all spawning
ISx Improvements
ISx: Background

- **Scalable Integer Sort benchmark**
  - Developed at Intel, published at PGAS 2015
  - SPMD-style computation with barriers
  - Punctuated by all-to-all bucket-exchange pattern
  - References implemented in SHMEM and MPI

- **Chapel implementation introduced in 1.13 release**
  - Motivation: bucket-exchange is a common distributed pattern
Scalable Barrier
Scalable Barrier: Background

- Previously reported ISx scalability on par with reference
- Believed we were mostly done looking at ISx
Scalable Barrier: Background

- Unfortunately we discovered some issues
  - Found a bug in our port that reported min, rather than avg, timings
  - At larger scales performance drops drastically
  - A bug fix for dynamic registration further hurt performance

```
Corrected Optimized ISx Timings

Time (sec)

Locales

ref  1.16  faster

Copyright 2018 Cray Inc.
Scalable Barrier: Background

- **Identified barrier implementation as scalability limiter**
  - Barrier used a single atomic variable on one locale
    - all remote locales did active messages back to that locale
  - 36-cores on 256 locales results in ~10,000 tasks on barrier locale
    - huge bottleneck, and default size task-stacks led to OOMs
Scalable Barrier: This Effort

● Added a scalable allLocalesBarrier
  ● A singleton global barrier that must be called from all locales
  ● optionally, with multiple tasks on each locale
  ● Similar to `shmem_barrier_all()` or `MPI_Barrier(MPI_COMM_WORLD)`

```plaintext
use AllLocalesBarriers;

coforall loc in Locales do on loc {  
  allLocalesBarrier.barrier();
  writeln("After barrier");
}

allLocalesBarrier.reset(4);
coforall loc in Locales do on loc do  
  coforall tid in 1..4 do  
    allLocalesBarrier.barrier();
```
Scalable Barrier: Impact

- **allLocalesBarrier offers significantly better scalability**
  - Over 2,000 times faster at 256 locales (and scaling better)
  - No on-stmts, so no single-node bottleneck or massive task creation

![Barrier Timings (1,000 Trials)](image)

- Time (sec)
- Locales
- local barrier
- comm barrier
- faster
Scalable Barrier: Impact

- `allLocalesBarrier` offers significantly better scalability
  - Over 2,000 times faster at 256 locales (and scaling better)
  - No on-stmts, so no single-node bottleneck or massive task creation

```
<table>
<thead>
<tr>
<th>Locales</th>
<th>Comm Barrier Timings (1,000,000 Trials)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>5</td>
</tr>
<tr>
<td>64</td>
<td>25</td>
</tr>
<tr>
<td>256</td>
<td>35</td>
</tr>
</tbody>
</table>
```

Graph showing barrier timings with a faster trend as locales increase.
Scalable Barrier: Impact

- Significantly improved scalability of ISx
  - Raw performance still behind reference, but scaling well
  - No longer any on-stmts in ISx

Optimized ISx Timings

Time (sec) vs Locales for different versions of ISx:
- ref
- 1.16
- 1.17.1

The graph shows a comparison of time (in seconds) for different locales (16, 32, 64, 128, 256) across the versions, with a trend indicating faster performance with higher locales for the optimized ISx versions.
Scalable Barrier: Next Steps

- **allLocalesBarrier has some limitations**
  - All locales must participate
  - A singleton barrier, only one instance exists

- **Add more usable barrier implementations**
  - Ability to barrier between a subset or team of locales
  - Ability to create multiple barriers
    - e.g.
      ```
      var teamABarrier = new LocalesBarrier(Locales[0..5]);
      var teamBBarrier = new LocalesBarrier(Locales[6..10]);
      var allLocBarrier = new LocalesBarrier(Locales);
      ```
Park the Main Process
Main Process: Background and Effort

**Background:** allLocalesBarrier hooks into chpl_comm_barrier()
- Optimized for the network and comm layer
  - tree-based put barrier under ugni, dissemination barrier under gasnet-aries
- chpl_comm_barrier() was previously tied up by runtime
  - Main process on non-0 locales waited for locale 0 to shutdown

**This Effort:** Park the main process on a condition variable
- Signaled during shutdown by an active message from locale 0
- Frees up chpl_comm_barrier() for use in user-code
Main Process: Impact

- Parking main process improved gasnet-aries performance
Reduce Progress Thread Interference
Progress Thread Interference: Background

- For multi-locale programs we start a “progress thread”
  - Separate pthread that processes active messages (on-stmts)
  - Actively checked for messages, yielding if none found

```c
while (run_progress_thread) {
    if (new_am()) process_am();
    else sched_yield();
}
```

- Even with no on-stmts, progress thread interfered
  - Context switch between progress thread and thread hosting chpl tasks
  - Resulted in wide variations for tasks doing identical operations:

```markdown
input-step: 0.91 avg (0.81 min .. 1.28 max)
```
Progress Thread Interference: This Effort

● **Added an experimental blocking progress thread**
  ● ugni only, enabled with CHPL_RT_COMM_UGNI_BLOCKING_CQ=y

```c
while (run_progress_thread) {
    am = block_for_am(); // kernel mediated, blocking call
    process_am(am);
}
```

● **Enabled for ISx, but not by default for 1.17**
  ● Improves ISx and benchmarks with few active messages
  ● But slightly increase latency of active messages
    ● mostly impacts microbenchmarks, but wanted more time to investigate

● **Enabled by default for 1.17.1**
  ● See Spectre/Meltdown slides for more information
Progress Thread Interference: Impact

- **Reduced variability for ISx steps**
  
  \[ \text{input-step: } 0.91 \text{ avg (0.81 min .. 1.28 max)} \]

  \[ \text{input-step: } 0.89 \text{ avg (0.81 min .. 0.95 max)} \]

- **Remaining variability due to dynamic array registration**
  - Kernel fault-in times for large allocations tend to vary
**ISx: Summary**

- **ISx scalability on par with reference**
  - (raw performance is still ~25% behind, but scaling well)
ISx: Next Steps

- **Continue to improve ISx performance**
  - Avoid dynamic registration for arrays
    - new dynamic heap extension can amortize cost of allocation/registration
    - dynamic array registration helps with parallel first-touch
    - but ISx runs in an SPMD manner, arrays initialized serially
  - Eliminate any extra communication compared to reference
  - Investigate using RDMA (BTE) for large puts/gets
    - currently only use FMA, but BTE should be better for large transfers
Meltdown and Spectre Impact
Meltdown and Spectre: Background

- **Meltdown and Spectre exploit security vulnerabilities**
  - Patches to mitigate exploits were expected to hurt performance
  - We had hoped that the impact on HPC/Chapel would be limited
    - overhead expected to be for I/O, system calls, etc. -- not HPC kernels
Meltdown and Spectre: Background

- Unfortunately patches hurt multi-locale performance
  - In some cases, performance regressions were significant
    - ~10% hit for stream-global, ~30% hit for ra-rmo
    - surprising, since stream is just memory bandwidth, RA just NIC operations

- Discovered overhead is from progress thread interference
  - Patches increased cost of context switches
  - Task running on core shared with progress thread slowed down
Meltdown and Spectre: This Effort

- **Reduce interference from the progress thread**
  - Fortunately, ISx investigation had us looking at this recently
  - Previously added an option to use a blocking progress thread
    - Now we just enable that functionality by default
    - Not resolved in time for 1.17 release, but is included in 1.17.1
Meltdown and Spectre: Impact

- Restored performance to pre-patch levels
  - In some cases performance is better than before
Meltdown and Spectre: Impact

- Caused some performance regressions
  - Using a blocking progress thread slightly increases on-stmt latency
Meltdown and Spectre: Next Steps

- Reduce progress thread interference for GASNet
  - Will need to work with the GASNet team on a strategy
Reductions in Memory Leaks
Memory Leaks

Background:

- Historically, Chapel testing has leaked a large amount of memory
- Chapel 1.15 and 1.16 closed major sources of large-scale leaks
- Remaining cases considered less concerning, but still undesirable

This Effort:

- Closed a number of additional sources of minor leaks:
  - distributed sparse domains and arrays
  - local caches of remote array metadata
  - iterator records
  - timezones in ‘DateTime’ module (using `Shared`)
  - rectangular arrays whose domains had been ‘clear’ed
  - temporary strings allocated in IO routines / zero-length strings
  - user-level leaks in test programs
- Also, reduced the memory footprint of non-stridable ranges
Memory Leaks: This Effort (October 2017)

Memory Leaks for all Tests

- cached remote array metadata
- user test leaks
- dist. sparse domains/arrays empty strings
- clear() on rectangular domains
- Reduced footprint of non-stridable ranges
Memory Leaks: This Effort (1.17 release cycle)

Impact: reduced leaks in nightly testing by ~50%
Memory Leaks: Remaining Leaks (as of April 10)

- Only 304 / 8398 tests still leaking
- ~1/3 of memory leaked by one test (SSCA#2)
- ~2/3 of leaking tests leak < 256 bytes
- ~1/3 of leaking tests leak < 64 bytes
Memory Leaks: Next Steps

Next Steps:

- Continue working through remaining leaks as a background task
Other Performance Optimizations
Other Performance Optimizations

- Improved remote value forwarding optimization for types with initializers
- Reduced wide-pointer overhead for domains and distributions
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 publicly 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.