ISx in Chapel

Ben Harshbarger, Cray Inc.

CHIUW @ IPDPS

May 27, 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.

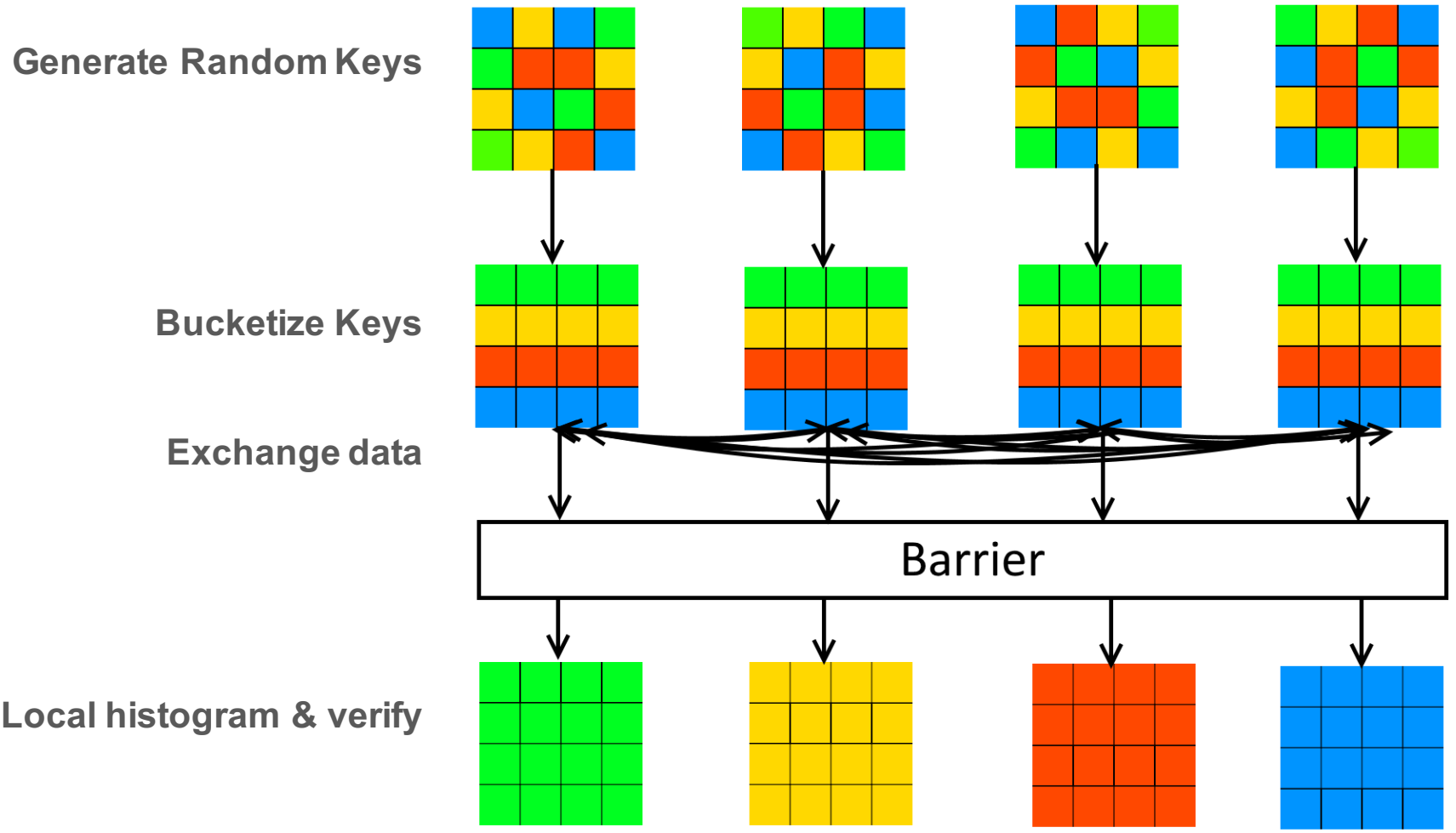
What Is The ISx Benchmark?

- **ISx: Scalable Integer Sort benchmark**
  - Modern replacement for NPB IS to address its shortcomings
  - Developed at Intel, published at PGAS 2015
  - Computation style:
    - Local SPMD-style computation with barriers
    - Punctuated by all-to-all bucket exchange pattern

- **SHMEM and MPI reference versions available on GitHub**
  - https://github.com/ParRes/ISx

- **A good case study for Chapel**
  - A common parallel pattern for distributed memory programming

What Is The ISx Benchmark?



Chapel Implementation

```
const BucketSpace = {0..#numBuckets};
const DistBucketSpace = BucketSpace dmapped Block(BucketSpace);

// For each bucket, create an array to receive keys
var allBucketKeys : [DistBucketSpace] [0..#recvBuffSize] int(32);

// Create globally-visible barrier
var barrier = new Barrier(numBuckets);

// Start a task for each bucket, call bucketSort
coforall loc in Locales do on loc {
  coforall tid in 0..#bucketsPerLocale {
    const taskID = (loc.id * bucketsPerLocale) + tid;
    for i in 1..numTrials {
      bucketSort(taskID, i);
    }
  }
}
```

Chapel Implementation

```
const BucketSpace = {0..#numBuckets};
const DistBucketSpace = BucketSpace dmapped Block(BucketSpace);

// For each bucket, create an array to receive keys
var allBucketKeys : [DistBucketSpace] [0..#recvBuffSize] int(32);

// Create globally-visible barrier
var barrier = new Barrier(numBuckets);

// Start a task for each bucket, call bucketSort
coforall loc in Locales do on loc {
  coforall tid in 0..#bucketsPerLocale {
    const taskID = (loc.id * bucketsPerLocale) + tid;
    for i in 1..numTrials {
      bucketSort(taskID, i);
    }
  }
}
```

Chapel Implementation

```
const BucketSpace = {0..#numBuckets};
const DistBucketSpace = BucketSpace dmapped Block(BucketSpace);

// For each bucket, create an array to receive keys
var allBucketKeys : [DistBucketSpace] [0..#recvBuffSize] int(32);

// Create globally-visible barrier
var barrier = new Barrier(numBuckets);

// Start a task for each bucket, call bucketSort
coforall loc in Locales do on loc {
  coforall tid in 0..#bucketsPerLocale {
    const taskID = (loc.id * bucketsPerLocale) + tid;
    for i in 1..numTrials {
      bucketSort(taskID, i);
    }
  }
}
```

Chapel Implementation

```
// Within bucketSort…

var myKeys = makeInput(taskID);

var myBucketedKeys = bucketizeLocalKeys(taskID, myKeys);

// Exchange step
for i in 0..#numBuckets {
  const transferSize, dstOffset, srcOffset = …

  allBucketKeys[i][dstOffset..#transferSize] =
    myBucketedKeys[srcOffset..#transferSize];
}
barrier.barrier();

var keyCounts = countLocalKeys(taskID);

verify(taskID, keyCounts);
```

SPMD vs. Global-view

- SPMD: bucket per core
  - Serial for-loops
  - Example from 'countLocalKeys'

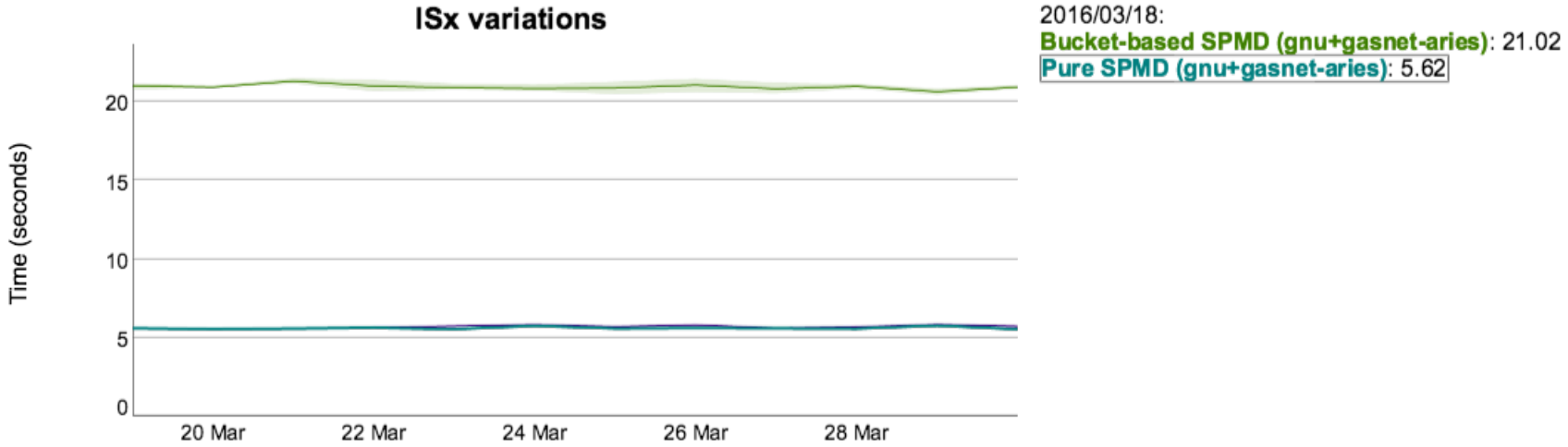
```
var keyCounts : […] int;
for i in 0..#myBucketSize do
  keyCounts[allBucketKeys[taskID][i]] += 1;
```

- Global view: bucket per locale
  - Forall loops for intra-locale parallelism
  - Atomics used to coordinate between loop iterations

```
var keyCounts : […] atomic int;
forall i in 0..#myBucketSize do
  keyCounts[allBucketKeys[taskID][i]].add(1);
```

SPMD vs. Global-view - performance

- **Global-view slower than SPMD version**
  - by up to 4x!



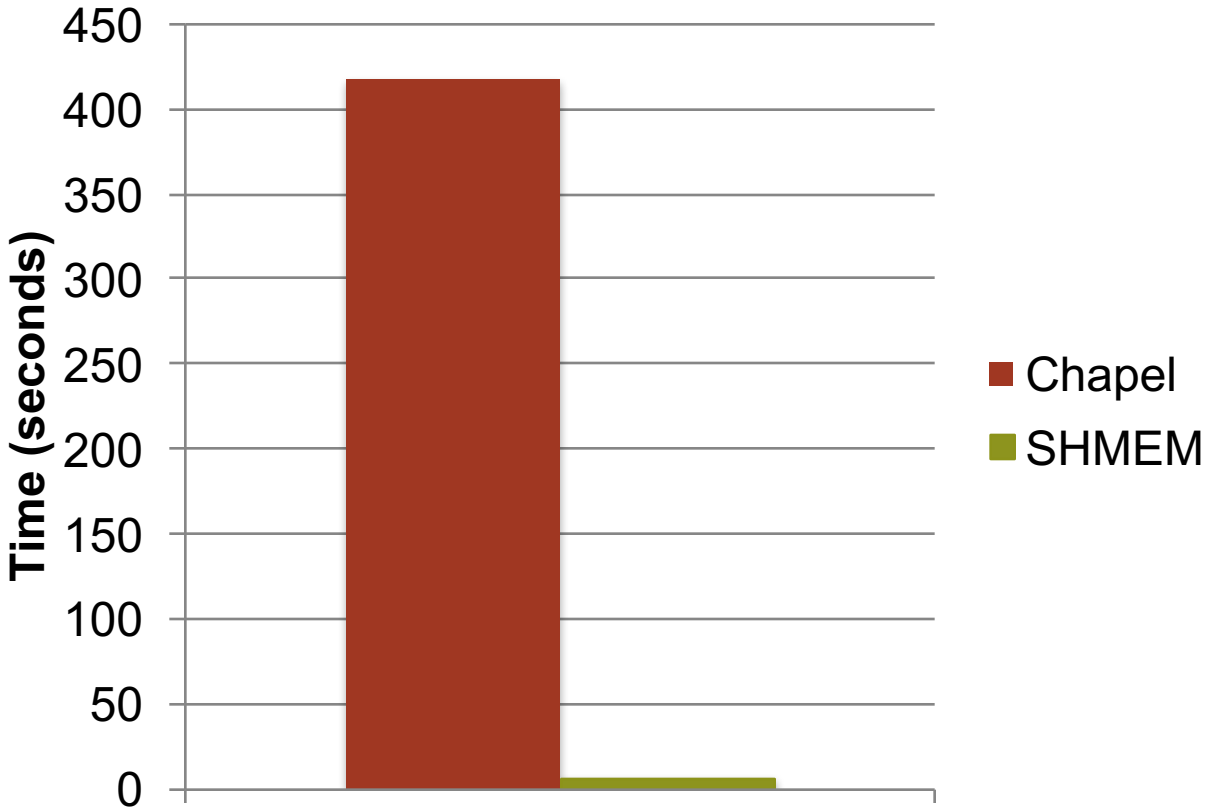
- **Likely due to atomics**
  - Global-view uses atomics to coordinate between forall-loop iterations
  - SPMD uses serial for loops, no atomics

Chapel vs. SHMEM

- **SPMD is faster and a more natural fit**
- **Initial port was much slower than SHMEM reference**
- **Numbers gathered with**
  - ISx reference version 1.1, weakISO scaling
  - Chapel 1.13, ugni-qthreads
  - gcc 5.1.0, -O3
  - cray-shmem/7.3.3
  - Cray XC, 36 broadwell cores per node
  - 134217728 (2^27) keys per bucket

Chapel vs. SHMEM

- **Initial comparison with two XC nodes: yikes!**
  - Nearly 80x worse!



Chapel vs. SHMEM

- Timing output indicated some problem areas

```
exchange = 355.502 (349.506..386.19)
count keys = 56.406 (16.9024..124.789)
```

- Exchange step ~350s, compared to SHMEM's 1.6s

- Counting step ~56s, compared to SHMEM's .2s

Chapel vs. SHMEM - Bulk Transfer

- **Exchange looks something like this:**
  - Slice and assign between two arrays

```
allBucketKeys[i][dstOffset..#transferSize] =
  myBucketedKeys[srcOffset..#transferSize];
```

- **We expect this to use Chapel's bulk transfer optimization**
  - One large GET/PUT/memcpy better than element-by-element

- **Investigation revealed bulk transfer not firing correctly**

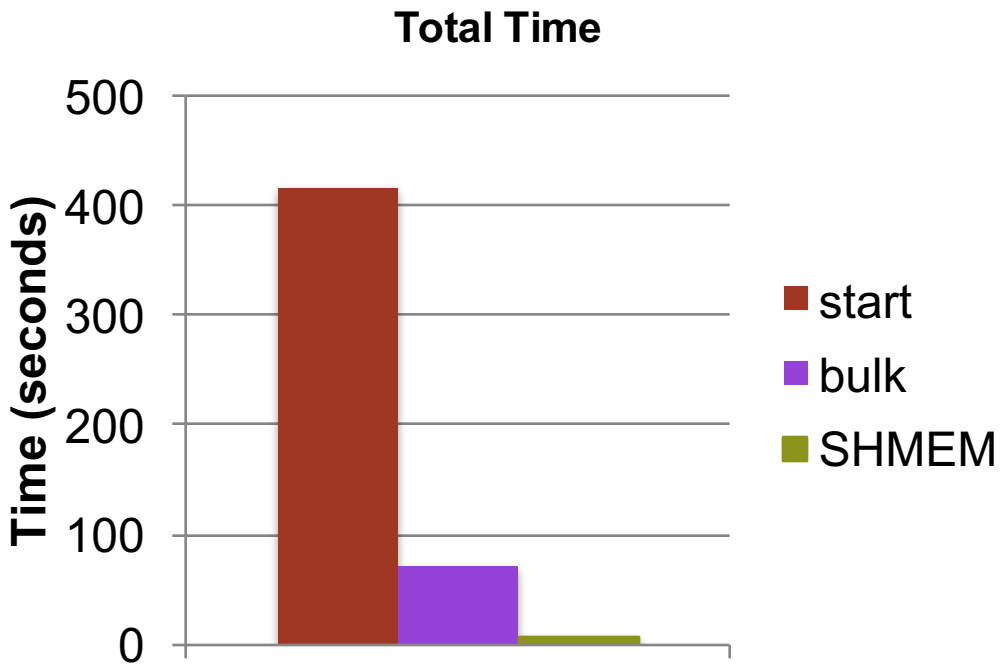
Chapel vs. SHMEM - Bulk Transfer

- **Solution: remove overly-conservative runtime check**
  - Prevented bulk transfers when slicing from the middle of an array
  - Near 6x improvement!

```
var A, B :[1..20] int;

// optimization works!
A[1..10] = B[1..10];

// Failed to bulk transfer!
// Fixed in 1.13 release
A[1..10] = B[5..15];
```



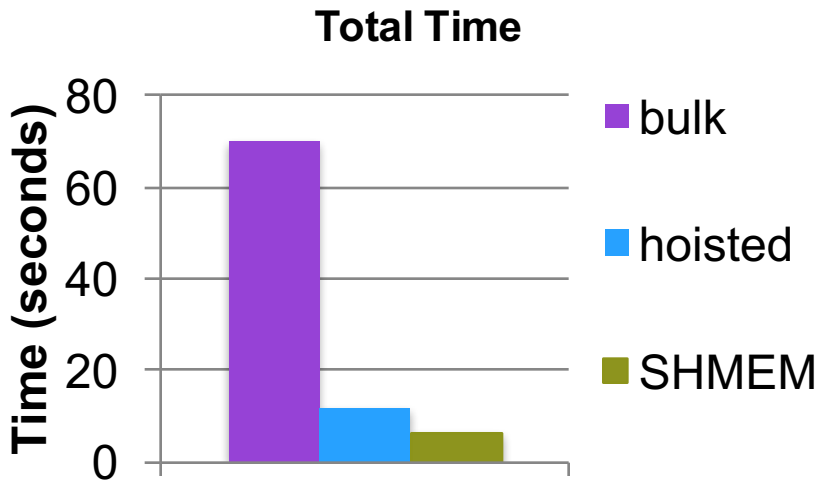
Chapel vs. SHMEM: Loop Hoisting

- Counting step slow (56s vs .2s)
- Solution: Manually optimize source code

```
for i in 0..#myBucketSize do
  keyCounts[allBucketKeys[taskID][i]] += 1; // loop-invariant
```

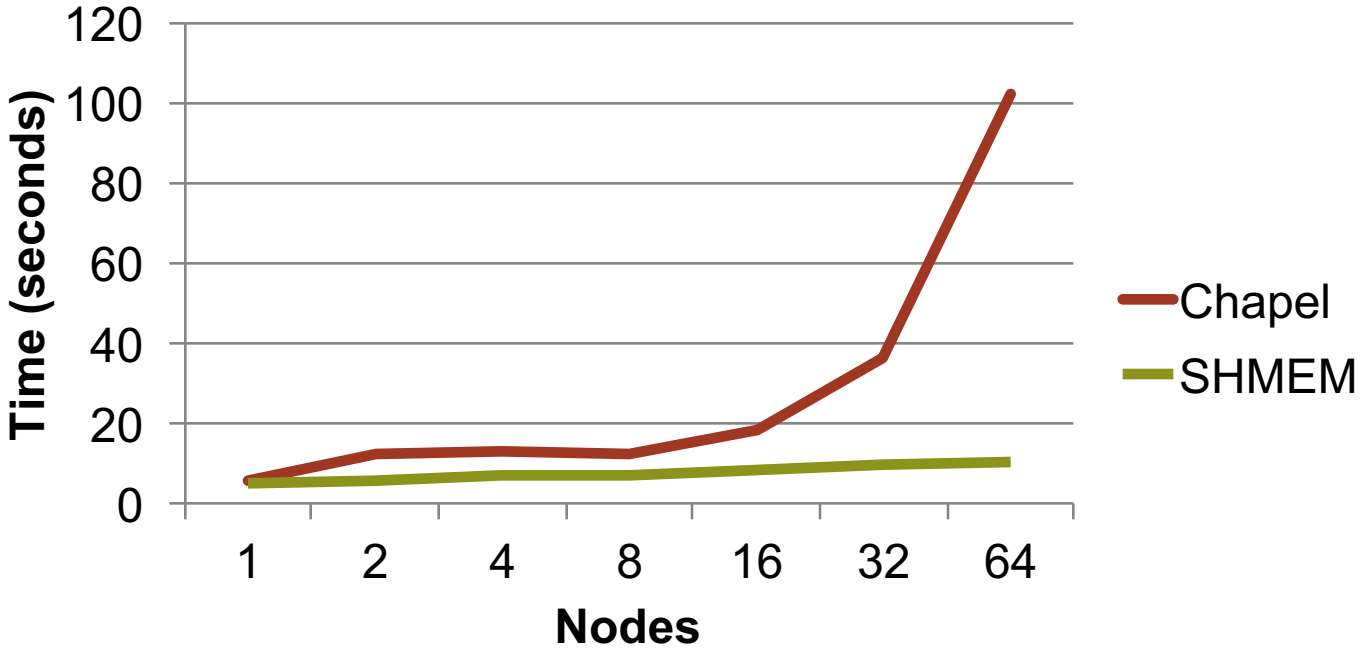
  - Manually hoisting helps tremendously
  - Compiler should perform this optimization in the future

- Result: immensely better
  - ~7x improvement
- Now, let's look at scaling…



Chapel vs. SHMEM: Scaling

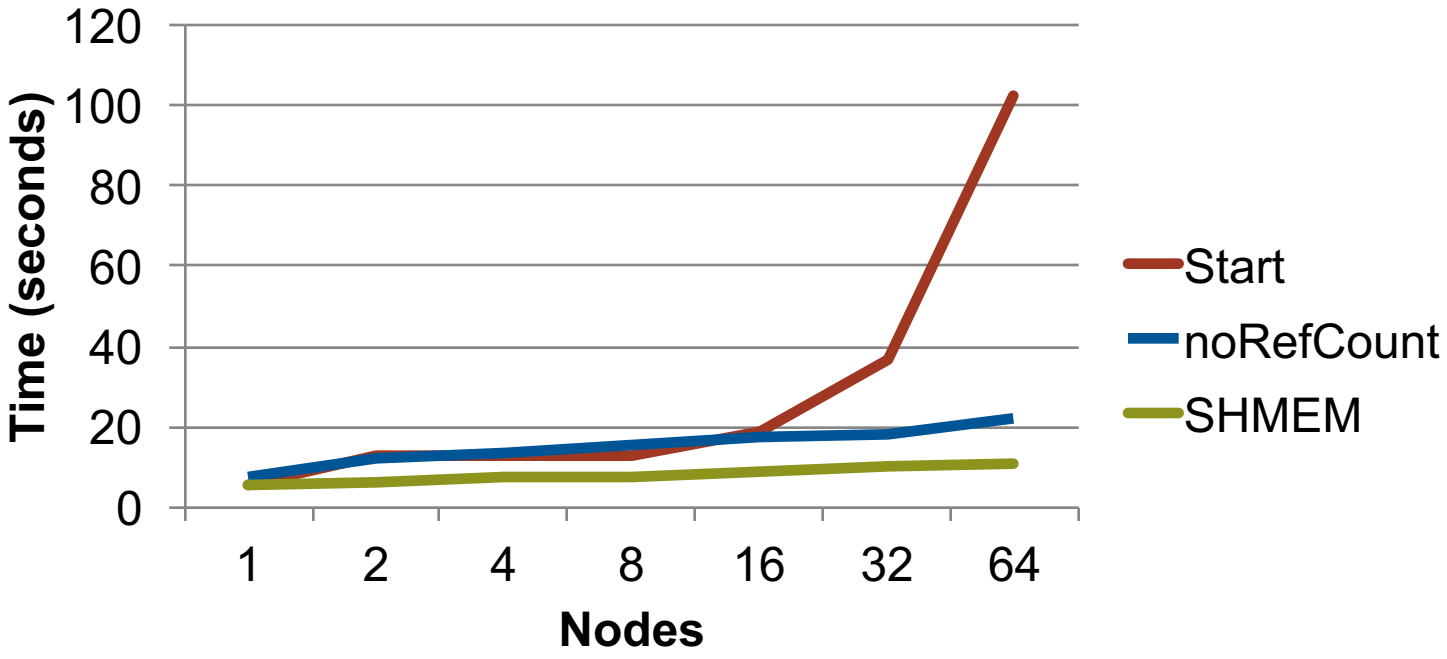
- Starts out OK, then goes off the rails…



- Exchange step still too long
  - ~96s vs SHMEM's ~8s

Chapel vs. SHMEM: Scaling

- **Bulk transfer is firing, what gives?**
- **Reference counting is a known source of overhead**
  - Especially for array/domain slicing…
  - Can be disabled with '–snoRefCount'



Chapel vs. SHMEM: Array Slicing

- **Observation: exchange is still slower than reference**
- **Suspicion: array slicing is at fault**
  - DefaultRectangular array slicing uses an on-statement
    - Ensures slices lives on same locale as actual array
  - # of ons equals numBuckets**2
- **Currently not a simple task to remove the on-statement**
  - Other optimizations rely on the existing semantics
- **Idea: avoid doing a full slice for bulk tranfers**
  - Recognize the slice is short-lived
  - Bulk transfer really only needs the slice's offset information

Conclusions

- **Easy to write in Chapel**
- **Without reference counting, about 2x worse**
  - Relatively good for Chapel, today
- **Future work for performance**
  - Improve reference counting
  - Better loop hoisting
  - Improve slicing performance

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.

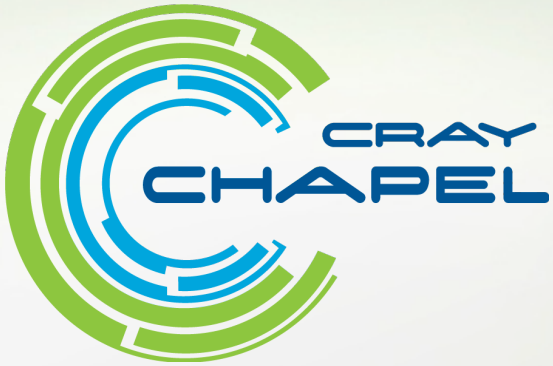
All products, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

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
THE SUPERCOMPUTER COMPANY

http://chapel.cray.com chapel_info@cray.com https://github.com/chapel-lang/chapel