Performance Optimizations

Chapel Team, Cray Inc.

Chapel version 1.12

October 1st, 2015

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

- STREAM Case Study
- Parallel Array Initialization Optimization
- Array Allocation Improvement
- Running Task Count Improvements
- Muxed Thread Limit Improvement
- Impact of Hugepages
- Optimizing Task Counters
- Stream Performance Summary
- Locality Optimizations
- Performance Improvements Summary

STREAM Case Study

STREAM Case Study: Background

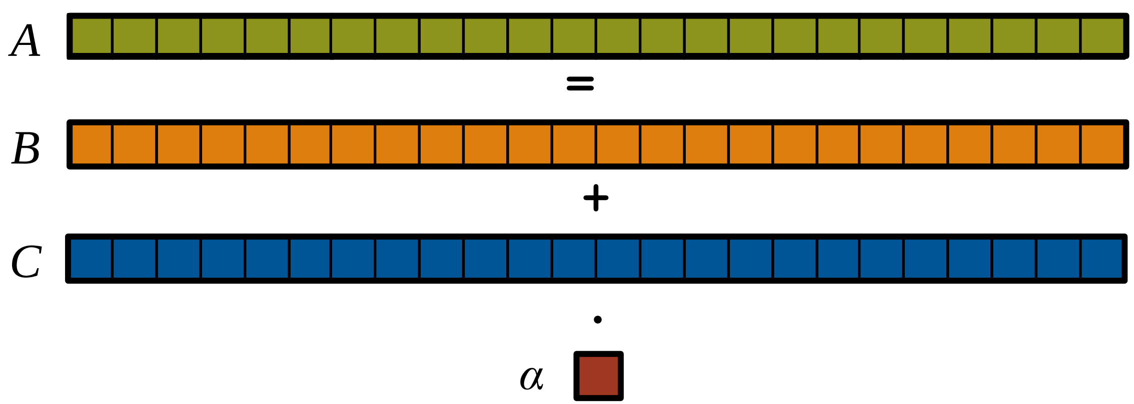
- **Previous releases focused on single-locale performance**
  - More and more, Chapel is becoming competitive with C/C++
- **For this release we shifted our focus to multi-locale**
  - Using STREAM as a case study to motivate optimizations
    - a simple benchmark, but important to get right
- **Several important optimizations resulted from this work:**
  - Parallelized array initialization
  - Switched from calloc() to malloc() for array allocation
  - Corrected running task counts
  - Removed thread limit for muxed
  - Investigated hugepage issues
  - Optimized task counters

STREAM: a trivial parallel computation

Given: m-element vectors A, B, C

Compute: $\forall i \in 1..m,\ A_i = B_i + \alpha \cdot C_i$

In pictures:

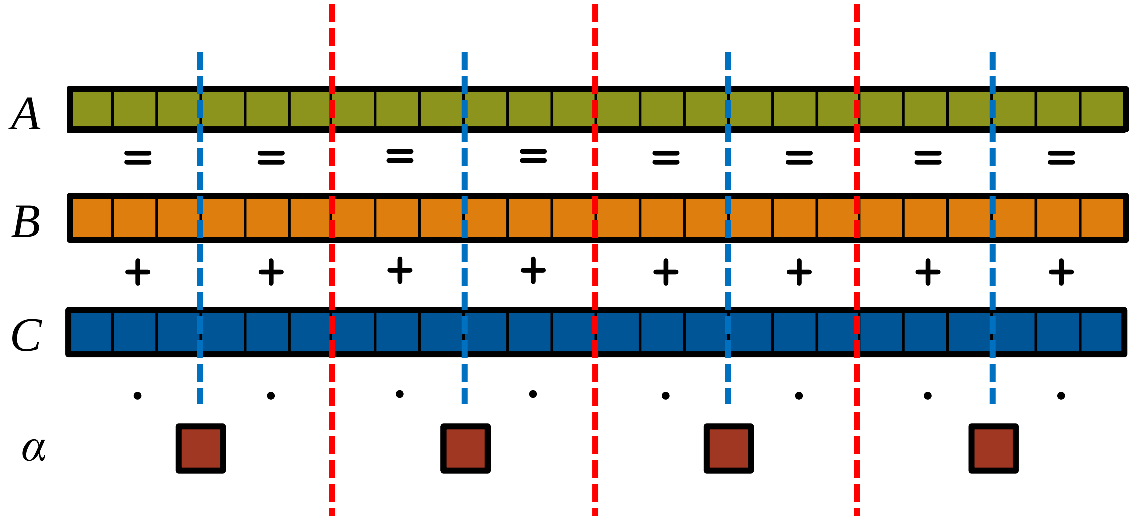


STREAM: a trivial parallel computation

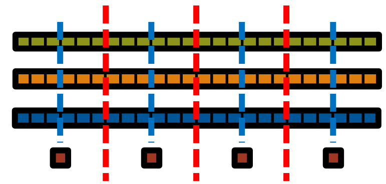
Given: m-element vectors A, B, C

Compute: $\forall i \in 1..m,\ A_i = B_i + \alpha \cdot C_i$

In pictures, in parallel (distributed memory multicore):



STREAM: MPI+OpenMP

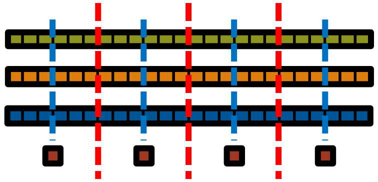


HPCC Reference

```
#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif

static int VectorSize;
static double *a, *b, *c;

int HPCC_StarStream(HPCC_Params *params) {
  int myRank, commSize;
  int rv, errCount;
  MPI_Comm comm = MPI_COMM_WORLD;

  MPI_Comm_size( comm, &commSize );
  MPI_Comm_rank( comm, &myRank );

  rv = HPCC_Stream( params, 0 == myRank);
  MPI_Reduce( &rv, &errCount, 1, MPI_INT, MPI_SUM,
    0, comm );

  return errCount;
}

int HPCC_Stream(HPCC_Params *params, int doIO) {
  register int j;
  double  scalar;

  VectorSize = HPCC_LocalVectorSize( params, 3,
    sizeof(double), 0 );

  a = HPCC_XMALLOC( double, VectorSize );
  b = HPCC_XMALLOC( double, VectorSize );
  c = HPCC_XMALLOC( double, VectorSize );
```

```
  if (!a || !b || !c) {
    if (c) HPCC_free(c);
    if (b) HPCC_free(b);
    if (a) HPCC_free(a);
    if (doIO) {
      fprintf( outFile, "Failed to allocate memory
    (%d).\n", VectorSize );
      fclose( outFile );
    }
    return 1;
  }

#ifdef _OPENMP
#pragma omp parallel for
#endif
  for (j=0; j<VectorSize; j++) {
    b[j] = 2.0;
    c[j] = 0.0;
  }

  scalar = 3.0;

#ifdef _OPENMP
#pragma omp parallel for
#endif
  for (j=0; j<VectorSize; j++)
    a[j] = b[j]+scalar*c[j];

  HPCC_free(c);
  HPCC_free(b);
  HPCC_free(a);
```

STREAM EP: Chapel

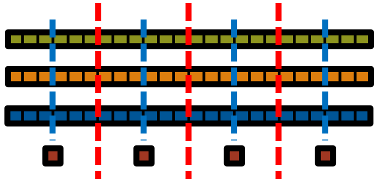


Chapel Stream EP (1.11 version)

```
coforall loc in Locales do on loc {          // create task per locale
  local {                                    // assert code in this block is local
    var A, B, C: [1..m] elemType;            // declare per-locale vectors
    initVectors(B, C);                       // initialize vectors

    startTimer();                            // start timed portion of code

    forall (a, b, c) in zip(A, B, C) do      // parallel vector iteration
      a = b + alpha * c;                     // multiply-add-assign

    stopTimer();                             // stop timed portion of code
  }
}
```

- **Written in traditional SPMD style (not elegant Chapel)**
  - Spawns one task per locale (outside of timed region)
  - 1.11 version used **local** block to help squash communication

Global STREAM: Chapel

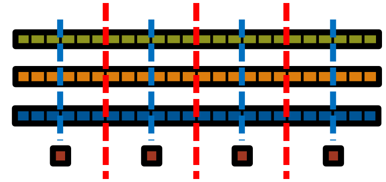


Chapel Global Stream

```
const ProblemSpace = {1..m}
                     dmapped Block({1..m}); // create distributed domain

var A, B, C: [ProblemSpace] elemType;       // create distributed vectors
initVectors(B, C);                          // initialize vectors

startTimer();                               // start timed portion of code

forall (a, b, c) in zip(A, B, C) do          // parallel vector iteration
  a = b + alpha * c;                         // multiply-add-assign

stopTimer();                                // stop timed portion of code
```

- **Elegant Chapel version**
  - Uses distributed (global) arrays
  - Spawning tasks on other locales happens within timed region

STREAM: Chapel



Stream EP

```
coforall loc in Locales do on loc {
  local {
    var A, B, C: [1..m] elemType;

    initVectors(B, C);

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

Global Stream

```
const ProblemSpace = {1..m} dmapped …;

var A, B, C: [ProblemSpace] elemType;

initVectors(B, C);

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

- **Our main performance goals for 1.12:**
  - Improve the compiler, runtime, and modules such that:
    - stream-ep performs as well as the reference
    - global stream is competitive with the reference
  - Improve compiler locality analysis and optimizations such that:
    - the **local** block in stream-ep can be removed

STREAM: Motivation

- **Relatively simple and straightforward benchmark**
  - Easy for us to debug and find performance issues
  - Has a minimal amount of communication
    - makes it easy to isolate other performance and scaling bottlenecks
  - Stream-inspired optimizations should improve most benchmarks
  - Serves as a proxy for embarrassingly/pleasingly parallel computations

- **Affinity is crucial for getting good performance**

- **Utilizes all cores and significant amounts of memory**
  - Should help identify weak links in tasking, memory, and comm layers

- **Global version demonstrates productivity of domain maps**
  - Competitive results will help abate long-term performance concerns
    - i.e. show that productivity and performance are not mutually exclusive

STREAM: Testing Configuration

- **Run on a Cray XC40:**
  - 24 core (48 HT) IvyBridge Processor (2 numa domains)
  - 128 GB RAM per node
  - GCC 4.9.2

- **Studied cross product of tasking, memory, comm layers**
  - To make isolating performance issues easier
  - To ensure that there are no glaring issues with any given layer

- **Test Results**
  - Will show several configurations compared to reference
    - and impact of individual changes
  - Most slides will show stream-ep vs. reference (GB/s per node)
    - will do a comparison of global stream at the end

STREAM: Initial Performance

- **In general we were just over 2x worse than reference**
  - Slightly worse for muxed and fifo

CHPL_COMM	CHPL_TASKS	CHPL_MEM	1 locale	16 locales
none	fifo	cstdlib	34 GB/s	N/A
	muxed		31 GB/s	
	qthreads		35 GB/s	
		dlmalloc	35 GB/s	
		tcmalloc	35 GB/s	
gasnet-mpi	qthreads	cstdlib	35 GB/s	35 GB/s
gasnet-aries	qthreads	dlmalloc	35 GB/s	35 GB/s
ugni	muxed	tcmalloc	30 GB/s	31 GB/s
	qthreads		35 GB/s	35 GB/s
Reference			74 GB/s	74 GB/s

Parallel Array Initialization Optimization

Parallel Initialization: Background

- **Uninitialized variables are assigned a default value**

```
var i: int;              // default initialized to 0
var A: [1..10] int;  // each element default initialized to 0
```

- **Array initialization has traditionally been serial**
  - Initialization is typically responsible for “first-touch”
    - incorrect first-touch results in poor affinity, which can hurt performance

- **Need a principled way to get good first-touch**
  - Short term: we want to default to some sort of parallel initialization
  - Long term: domain map author should specify parallel initialization
    - permits parallelization strategy to match parallel iterators
    - requires finalizing and implementing our constructor proposal
    - also want to permit users to ‘noinit’ arrays

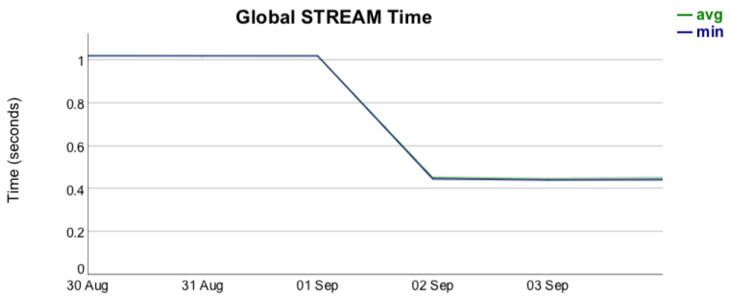
Parallel Initialization: This Effort

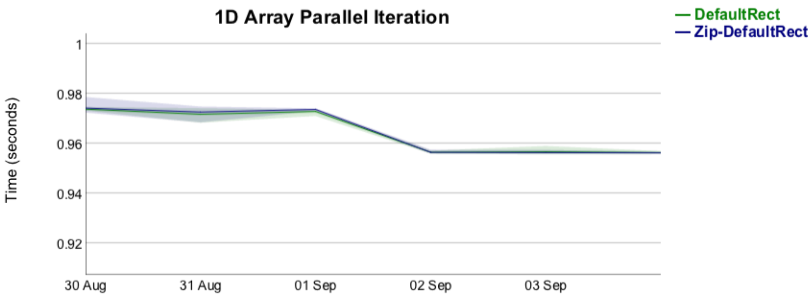
- **Determine when parallel initialization is appropriate**
  - Consistent to assume that *most* arrays will be used in parallel
    - other operations are parallel by default (promotion, reductions, etc.)
  - However, parallel initialization is not always the right choice
    - e.g. code with many small arrays (especially if constructed in a loop)

- **Heuristic: Parallel initialize numeric arrays > 2MB**
  - Initial attempts at heuristics were naïve
    - tried to parallel initialize all arrays, then tried several unreliable heuristics
  - Moved to experimentally determining a good size
    - 2MB is good for 2 core laptop, 8 core desktop, 24 core XC, 240 core KNC
  - Decided to only parallel initialize numeric arrays
    - serious performance regressions for arrays of arrays
    - will be addressed in future releases, but was not high priority now
    - stepping stone will be to enable for arrays of plain old data (POD) types

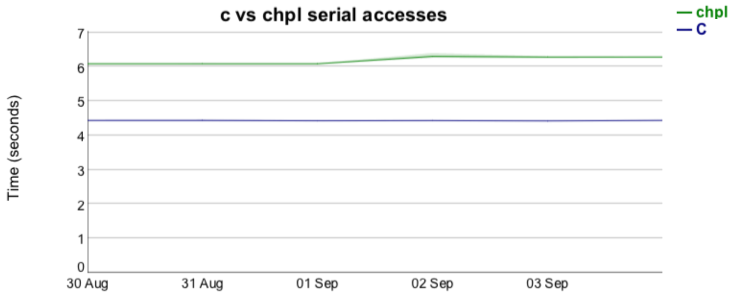
Parallel Initialization: Impact

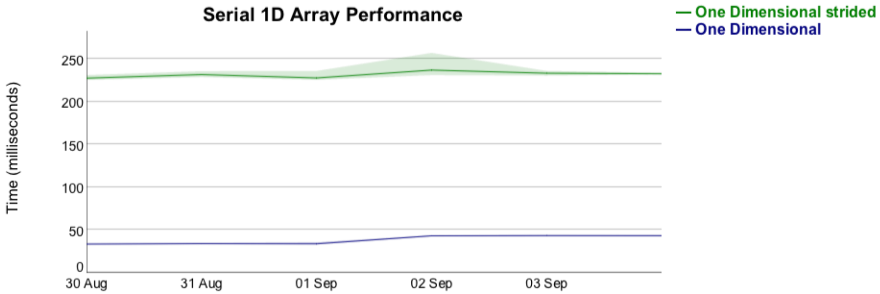
- **Improvements for several benchmarks**





- **Regressions for benchmarks testing serial array access**
  - understood, and acceptable (not representative of "real" code)





Parallel Initialization: Stream Impact

- **Substantial performance improvements**
  - for all tasking layers
  - for gasnet-mpi
  - for most memory layers

CHPL_COMM	CHPL_TASKS	CHPL_MEM	1 locale	16 locales
none	fifo	cstdlib	64 GB/s	N/A
	muxed		48 GB/s	
	qthreads		66 GB/s	
		dlmalloc	66 GB/s	
		tcmalloc	35 GB/s	
gasnet-mpi	qthreads	cstdlib	67 GB/s	71 GB/s
gasnet-aries	qthreads	dlmalloc	35 GB/s	35 GB/s
ugni	muxed	tcmalloc	30 GB/s	31 GB/s
	qthreads		35 GB/s	35 GB/s
Reference			74 GB/s	74 GB/s

Parallel Initialization: Stream Impact

- **Substantial performance improvements**
  - for all tasking layers
  - for gasnet-mpi
  - for most memory layers (except tcmalloc)

CHPL_COMM	CHPL_TASKS	CHPL_MEM	1 locale	16 locales
none	fifo	cstdlib	64 GB/s	N/A
	muxed		48 GB/s	
	qthreads		66 GB/s	
		dlmalloc	66 GB/s	
		tcmalloc	35 GB/s	
gasnet-mpi	qthreads	cstdlib	67 GB/s	71 GB/s
gasnet-aries	qthreads	dlmalloc	35 GB/s	35 GB/s
ugni	muxed	tcmalloc	30 GB/s	31 GB/s
	qthreads		35 GB/s	35 GB/s
Reference			74 GB/s	74 GB/s

Array Allocation Improvement

Array Allocation: Background

Background: Noticed that tcmalloc still had bad first-touch

- Discovered that array allocation was being done with calloc()
- tcmalloc always uses memset() with calloc(), which touches pages
  - dlmalloc and cstdlib check if mmap() zeros pages and avoid memset()

This Effort: Switch to using malloc() instead of calloc()

- There was no reason for us to be using calloc()
  - we initialize arrays in the modules after allocation
  - calloc() was inadvertently introduced in early hierarchical locales work

Array Allocation: Stream Impact

- **Substantial performance improvements**
  - for single locale tcmalloc

<table>
<tr><th>CHPL_COMM</th><th>CHPL_TASKS</th><th>CHPL_MEM</th><th>1 locale</th><th>16 locales</th></tr>
<tr><td rowspan="5">none</td><td>fifo</td><td rowspan="3">cstdlib</td><td>64 GB/s</td><td rowspan="5">N/A</td></tr>
<tr><td>muxed</td><td>48 GB/s</td></tr>
<tr><td rowspan="3">qthreads</td><td>66 GB/s</td></tr>
<tr><td>dlmalloc</td><td>66 GB/s</td></tr>
<tr><td>tcmalloc</td><td>66 GB/s</td></tr>
<tr><td colspan="5"></td></tr>
<tr><td>gasnet-mpi</td><td>qthreads</td><td>cstdlib</td><td>67 GB/s</td><td>71 GB/s</td></tr>
<tr><td colspan="5"></td></tr>
<tr><td>gasnet-aries</td><td>qthreads</td><td>dlmalloc</td><td>35 GB/s</td><td>35 GB/s</td></tr>
<tr><td colspan="5"></td></tr>
<tr><td rowspan="2">ugni</td><td>muxed</td><td rowspan="2">tcmalloc</td><td>48 GB/s</td><td>31 GB/s</td></tr>
<tr><td>qthreads</td><td>66 GB/s</td><td>35 GB/s</td></tr>
<tr><td colspan="5"></td></tr>
<tr><td colspan="3">Reference</td><td>74 GB/s</td><td>74 GB/s</td></tr>
</table>

Array Allocation: Stream Impact

- **Substantial performance improvements**
  - for single locale tcmalloc
  - (overall, single locale was still slightly behind reference)

<table>
<tr><th>CHPL_COMM</th><th>CHPL_TASKS</th><th>CHPL_MEM</th><th>1 locale</th><th>16 locales</th></tr>
<tr><td rowspan="5">none</td><td>fifo</td><td rowspan="3">cstdlib</td><td>64 GB/s</td><td rowspan="5">N/A</td></tr>
<tr><td>muxed</td><td>48 GB/s</td></tr>
<tr><td rowspan="3">qthreads</td><td>66 GB/s</td></tr>
<tr><td>dlmalloc</td><td>66 GB/s</td></tr>
<tr><td>tcmalloc</td><td>66 GB/s</td></tr>
<tr><td colspan="5"></td></tr>
<tr><td>gasnet-mpi</td><td>qthreads</td><td>cstdlib</td><td>67 GB/s</td><td>71 GB/s</td></tr>
<tr><td colspan="5"></td></tr>
<tr><td>gasnet-aries</td><td>qthreads</td><td>dlmalloc</td><td>35 GB/s</td><td>35 GB/s</td></tr>
<tr><td colspan="5"></td></tr>
<tr><td rowspan="2">ugni</td><td>muxed</td><td rowspan="2">tcmalloc</td><td>48 GB/s</td><td>31 GB/s</td></tr>
<tr><td>qthreads</td><td>66 GB/s</td><td>35 GB/s</td></tr>
<tr><td colspan="5"></td></tr>
<tr><td colspan="3">Reference</td><td>74 GB/s</td><td>74 GB/s</td></tr>
</table>

Running Task Count Improvements

Running Task Count: Background

Background: Discovered locale 0 was mostly unused

- non-blocking on-stmts were being counted as local running tasks
  - as was the task waiting for a coforall/cobegin to complete
- Iterators select degree of parallelism based on running task count
  - incorrect running task count led to iterators creating too few tasks

```
coforall loc in Locales do on loc {
  // running task count on locale 0 was numLocales+1 here, rather than 1
}
```

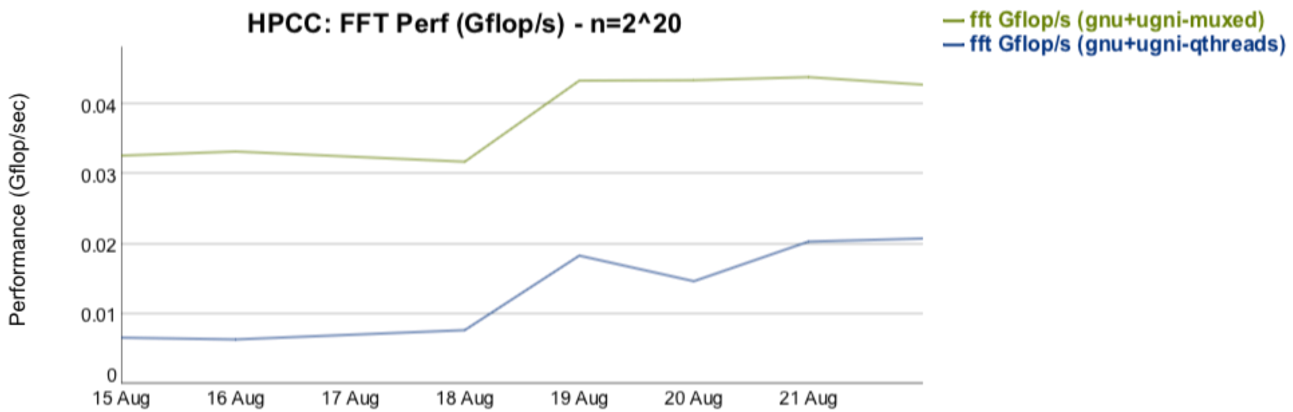
This Effort: Improve accuracy of running task count

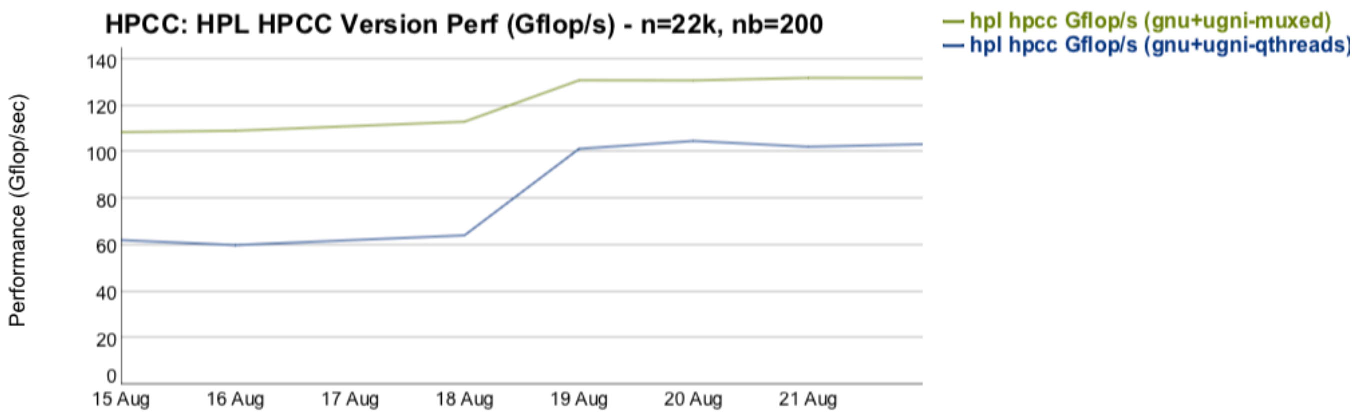
- Stop counting non-blocking on-stmts as tasks
- Stop counting the task waiting for a coforall/cobegin to finish

```
coforall loc in Locales do on loc {
  // now, running task count is 1 on all locales
}
```

Running Task Count: Impact

- **Several performance improvements**
  - Larger values are better





Running Task Count: Stream Impact

- **Substantial performance improvements**
  - qthreads is on par with reference!
  - gasnet-mpi 1- and 16-locale are on par with reference!
  - fifo was close to reference

CHPL_COMM	CHPL_TASKS	CHPL_MEM	1 locale	16 locales
none	fifo	cstdlib	68 GB/s	N/A
	muxed		48 GB/s	
	qthreads		74 GB/s	
		dlmalloc	74 GB/s	
		tcmalloc	74 GB/s	
gasnet-mpi	qthreads	cstdlib	74 GB/s	74 GB/s
gasnet-aries	qthreads	dlmalloc	35 GB/s	35 GB/s
ugni	muxed	tcmalloc	48 GB/s	31 GB/s
	qthreads		74 GB/s	35 GB/s
Reference			74 GB/s	74 GB/s

Running Task Count: Stream Impact

- **Substantial performance improvements**
  - qthreads is on par with reference!
  - gasnet-mpi 1- and 16-locale are on par with reference!
  - fifo was close to reference (muxed did not change)

CHPL_COMM	CHPL_TASKS	CHPL_MEM	1 locale	16 locales
none	fifo	cstdlib	68 GB/s	N/A
	muxed		48 GB/s	
	qthreads		74 GB/s	
		dlmalloc	74 GB/s	
		tcmalloc	74 GB/s	
gasnet-mpi	qthreads	cstdlib	74 GB/s	74 GB/s
gasnet-aries	qthreads	dlmalloc	35 GB/s	35 GB/s
ugni	muxed	tcmalloc	48 GB/s	31 GB/s
	qthreads		74 GB/s	35 GB/s
Reference			74 GB/s	74 GB/s

Muxed Thread Limit Improvement

Muxed Thread Limit

Background: Discovered muxed was only using 16/24 cores

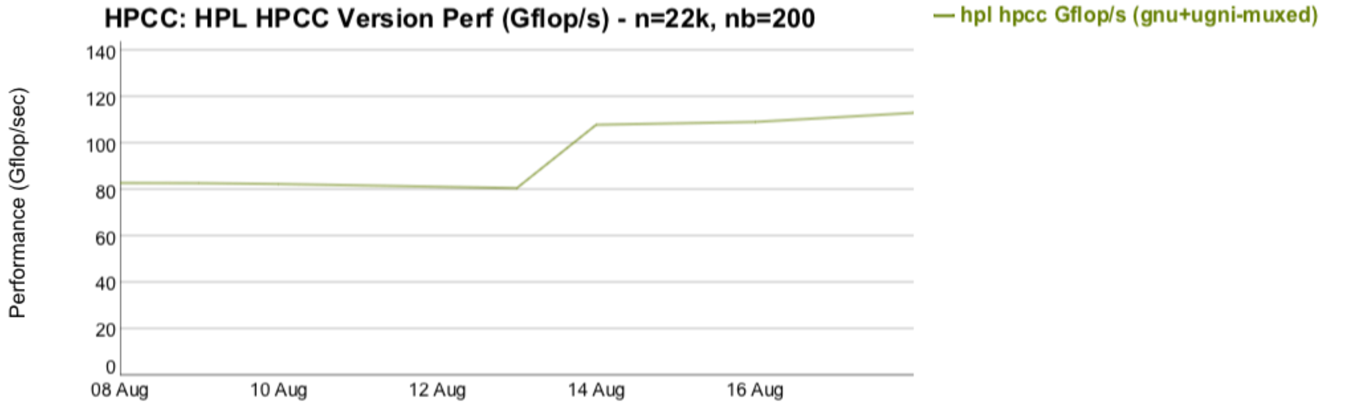
- Outdated code limited muxed to max of 16 hardware threads
  - muxed was originally tuned for Gemini and SSCA#2 in HPCS days
- Hardware and muxed configuration have changed since then
  - hard-coding this thread limit unnecessary and undesirable
- Missed this in previous releases since our machines only had 16 cores

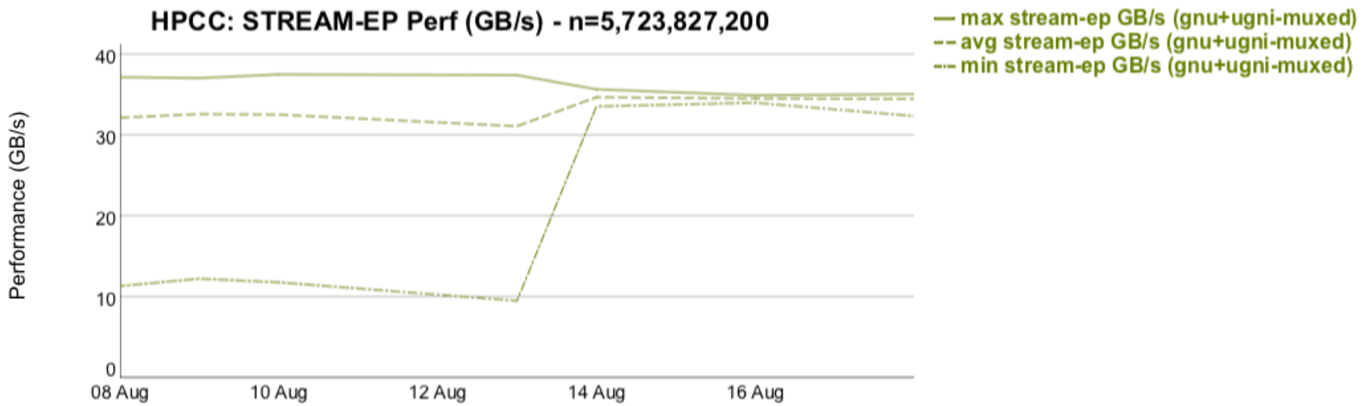
This Effort: Remove limit on number of hardware threads

- Default is now the number of physical cores
  - can be changed by user up to a comm layer limit (as with qthreads/fifo)

Muxed Thread Limit: Impact

- **Several performance improvements**
  - Larger values are better





Muxed Thread Limit: Stream Impact

- **Substantial performance improvements**
  - 1-locale muxed is really close to reference
    - qthreads (our default for 1.12) has better support for affinity and pinning
  - 16-locale muxed was slightly better than before

CHPL_COMM	CHPL_TASKS	CHPL_MEM	1 locale	16 locales
none	fifo	cstdlib	68 GB/s	N/A
	muxed		70 GB/s	
	qthreads		74 GB/s	
		dlmalloc	74 GB/s	
		tcmalloc	74 GB/s	
gasnet-mpi	qthreads	cstdlib	74 GB/s	74 GB/s
gasnet-aries	qthreads	dlmalloc	35 GB/s	35 GB/s
ugni	muxed	tcmalloc	70 GB/s	35 GB/s
	qthreads		74 GB/s	35 GB/s
Reference			74 GB/s	74 GB/s

Muxed Thread Limit: Stream Impact

- **Substantial performance improvements**
  - 1-locale muxed is really close to reference
    - qthreads (our default for 1.12) has better support for affinity and pinning
  - 16-locale muxed was slightly better than before (16-locale ugni still bad)

CHPL_COMM	CHPL_TASKS	CHPL_MEM	1 locale	16 locales
none	fifo	cstdlib	68 GB/s	N/A
	muxed		70 GB/s	
	qthreads		74 GB/s	
		dlmalloc	74 GB/s	
		tcmalloc	74 GB/s	
gasnet-mpi	qthreads	cstdlib	74 GB/s	74 GB/s
gasnet-aries	qthreads	dlmalloc	35 GB/s	35 GB/s
ugni	muxed	tcmalloc	70 GB/s	35 GB/s
	qthreads		74 GB/s	35 GB/s
Reference			74 GB/s	74 GB/s

Impact of Hugepages

Hugepages: Background

Background: ugni and gasnet-aries use hugepages

- Cray Gemini/Aries NICs must register (pin) memory to access it
  - gasnet-aries always registers, ugni only registers when numLocales > 1
- Registration is page-based, NIC has limited number of entries
  - registering significant memory requires huge pages
- Currently Chapel registers the entire heap and data segment
  - unfortunately, registration touches pages causing bad first-touch

This Effort: Investigate solutions for bad first-touch with ugni

- Work to resolve this issue is underway
  - but did not make it into the 1.12 release
- We will show the performance of that work for stream
  - to see the impact of later optimizations
- Did not investigate gasnet-aries yet
  - suspect memory registration also leads to bad first-touch

Hugepages: Stream Impact

Reminder: This work did not make it into 1.12

- **Substantial performance improvements**
  - for 16-locale ugni (still off from reference though)

CHPL_COMM	CHPL_TASKS	CHPL_MEM	1 locale	16 locales
none	fifo	cstdlib	68 GB/s	N/A
	muxed		70 GB/s	
	qthreads		74 GB/s	
		dlmalloc	74 GB/s	
		tcmalloc	74 GB/s	
gasnet-mpi	qthreads	cstdlib	74 GB/s	74 GB/s
gasnet-aries	qthreads	dlmalloc	35 GB/s	35 GB/s
ugni	muxed	tcmalloc	70 GB/s	64 GB/s
	qthreads		74 GB/s	55 GB/s
Reference			74 GB/s	74 GB/s

Optimizing Task Counters

Optimizing Task Counters: Background

- **Chapel has support for network-based atomics**
  - Chosen globally using CHPL_NETWORK_ATOMICS
    - changes default type for all **atomic** variables
  - Availability depends on hardware/environment
    - currently available only for comm=ugni

- **Internally, Chapel uses atomics for:**
  - the running task counter on each locale
  - tracking the # of completed tasks for a parallel construct (‘*endcounts*’)

    `begin`, `cobegin`, and `coforall`

Optimizing Task Counters: Background

- **Historically, task counters used default atomic type**

```
// Processor or network atomic, depending on CHPL_NETWORK_ATOMICS
var runningTaskCounter: atomic int;
// Ditto for compiler-generated endcounts for sync, cobegin, coforall statements…
```

- **A parallel construct's endcount may be provably local**

```
coforall i in 1..10 {…}    // lack of on-stmt means it's local
cobegin {…}
```

- **Processor atomics are much faster for local operations**

Optimizing Task Counters: This Effort

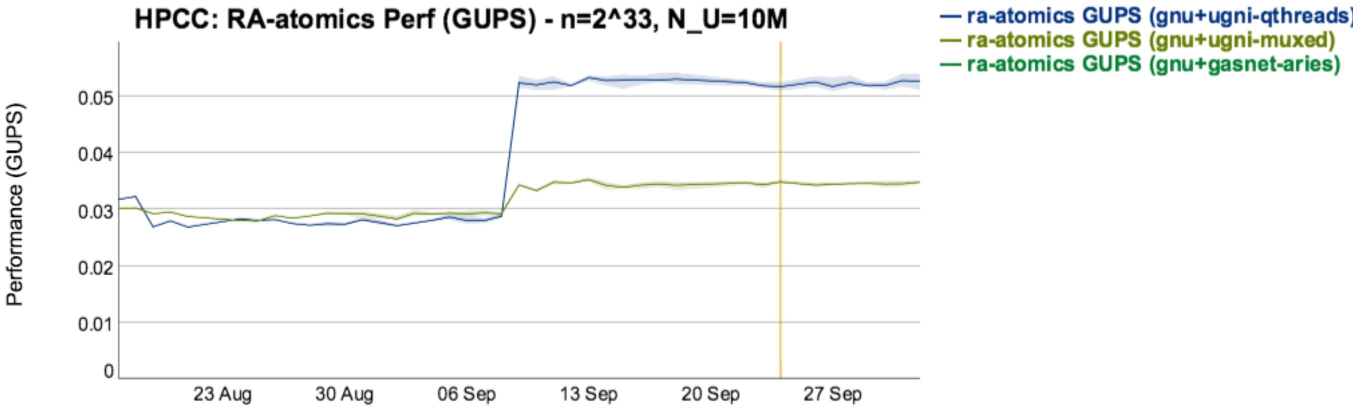
- **Force use of processor atomics for running task counter**
  - It is a per-locale counter that is always accessed locally
  - Current method of forcing processor atomics not intended for users
    - future work to provide a user-facing mechanism
    - possibly repurposing the “local” keyword

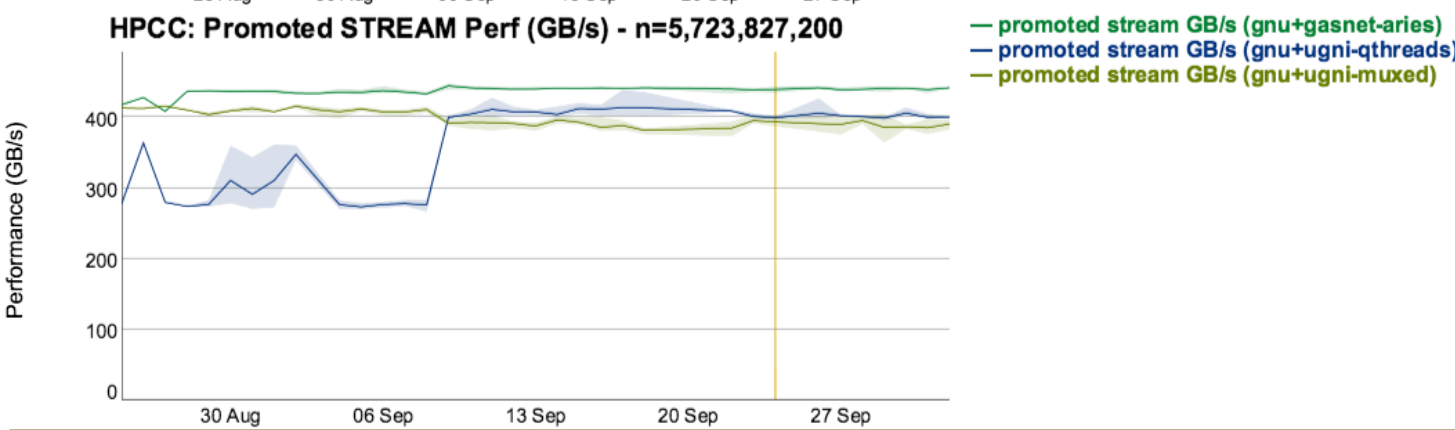
- **Have compiler choose atomic type for endcounts**
  - Use processor atomics for local **cobegin** and **coforall** statements
    - i.e. blocking parallel constructs that do not have an **on**-statement

- **Note that these changes are invisible to users**
  - Contained within compiler and internal modules

Optimizing Task Counters: Impact

- **Positive impact for multi-locale programs**
  - For CHPL_COMM=ugni
  - Larger values are better





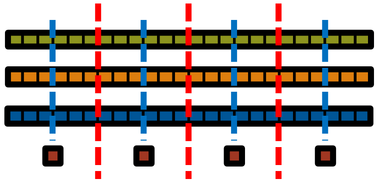
Optimizing Task Counters: Stream Impact

- **Substantial performance improvements**
  - Note: these numbers are with the hugepage workaround
  - 16-locale ugni is on par with 1-locale – qthreads on par with reference

CHPL_COMM	CHPL_TASKS	CHPL_MEM	1 locale	16 locales	16 w/o hugepages
none	fifo	cstdlib	68 GB/s	N/A	
	muxed		70 GB/s		
	qthreads		74 GB/s		
		dlmalloc	74 GB/s		
		tcmalloc	74 GB/s		
gasnet-mpi	qthreads	cstdlib	74 GB/s	74 GB/s	74 GB/s
gasnet-aries	qthreads	dlmalloc	35 GB/s	35 GB/s	N/A
ugni	muxed	tcmalloc	70 GB/s	35 GB/s	70 GB/s
	qthreads		74 GB/s	35 GB/s	74 GB/s
Reference			74 GB/s	74 GB/s	74 GB/s

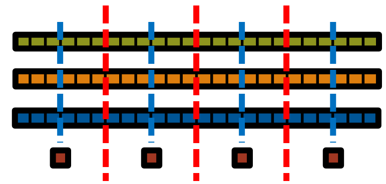
Stream Performance Summary

Stream Performance Summary



- **Summary of Improvements:**
  - Parallelized array initialization
  - Switched to malloc() for array allocation
  - Corrected running task count
  - Removed thread limit for muxed
  - Investigated hugepage issues
  - Optimized task counters

Stream Performance Summary



Stream EP

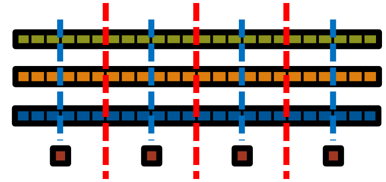
```
coforall loc in Locales do on loc {
  local {
    var A, B, C: [1..m] elemType;

    initVectors(B, C);

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

Global Stream

```
const ProblemSpace = {1..m} dmapped …;

var A, B, C: [ProblemSpace] elemType;

initVectors(B, C);

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

- **Our main performance goals for 1.12:**
  - Improve the compiler, runtime, and modules such that:
    - ❑ stream-ep performs as well as the reference
    - ❑ global stream is competitive with the reference
  - Improve compiler locality analysis and optimizations such that:
    - ❑ the **local** block in stream-ep can be removed

Stream Performance Summary

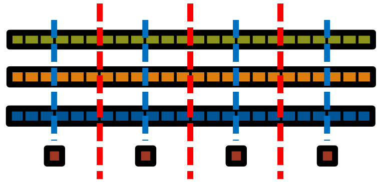


CHPL_COMM	CHPL_TASKS	CHPL_MEM	1 locale	16 locales	16 w/o hugepages
none	qthreads	cstdlib	74 GB/s	N/A	
gasnet-mpi			74 GB/s	74 GB/s	74 GB/s
gasnet-aries		dlmalloc	35 GB/s	35 GB/s	N/A
ugni		tcmalloc	74 GB/s	35 GB/s	74 GB/s
Reference			74 GB/s	74 GB/s	74 GB/s

Stream EP Performance:

- **Blue configurations perform as well as the reference!**
  - Still have a little work to do for ugni
    - last remaining issue is understood and already being worked on
  - Gasnet-aries still has first-touch problems
    - not a high priority (only used if building from source on a Cray)

Stream Performance Summary

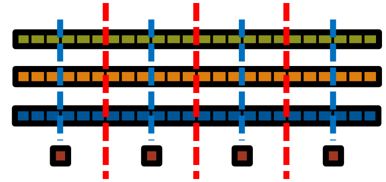


CHPL_COMM	CHPL_TASKS	CHPL_MEM	1 locale	16 locales	16 w/o hugepages
none	qthreads	cstdlib	74 GB/s	N/A	
gasnet-mpi			74 GB/s	72 GB/s	72 GB/s
gasnet-aries		dlmalloc	35 GB/s	34 GB/s	N/A
ugni		tcmalloc	74 GB/s	35 GB/s	73 GB/s
Reference			74 GB/s	74 GB/s	74 GB/s

Global Stream Performance:

- **Blue configurations perform close or as well as reference!**
  - No overhead for 1 locale
  - Slight (3%) overhead for 16 locales
    - most likely because remote task creation is inside timed section
    - could also mean that our block distribution could use some tuning

Stream Performance Summary



CHPL_COMM	CHPL_TASKS	CHPL_MEM	1 locale	16 locales	16 w/o hugepages
none	qthreads	cstdlib	74 GB/s	N/A	
gasnet-mpi			74 GB/s	72 GB/s	72 GB/s
gasnet-aries		dlmalloc	35 GB/s	34 GB/s	N/A
ugni		tcmalloc	74 GB/s	35 GB/s	73 GB/s
Reference			74 GB/s	74 GB/s	74 GB/s

- **Our main performance goals for 1.12:**
  - Improve the compiler, runtime, and modules such that:
    - ✓ stream-ep performs as well as the reference
    - ✓ global stream is competitive with the reference
  - Improve compiler locality analysis and optimizations such that:
    - ❑ the **local** block in stream-ep can be removed

Locality Optimizations

Locality optimizations: Background

- **STREAM-EP in 1.11:**

```
coforall loc in Locales do on loc {
  local { // Permits compiler to squash overheads related to wide pointers
    var A, B, C: [1..m] elemType;
    initVectors(B, C);

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

- **Goal: get rid of the local block**
  - Cumbersome language feature in general
  - Compiler **should** be able to eliminate all overhead in this case
    - code within local block can be trivially seen to be local
- **Wide pointers are the main source of overhead**

Locality optimizations: Background

- **Wide pointers represent potentially remote data**

```
typedef struct {
  int localeID;  // where this object lives
  Foo addr;      // pointer to data
} wide_Foo;      // wide pointer for class Foo
```

- **Use runtime GETs and PUTs to read/write data**
  - Will short-circuit if data is local

- **Significant source of overhead**
  - Some overhead for runtime calls
  - Potential for communication thwarts back-end compiler optimizations

Locality optimizations: Background

- **'chpl' has traditionally introduced wide pointers liberally**
  - ✓ Simple implementation
  - ✓ Easier to ensure program correctness
  - ✗ Unnecessary overhead, often for cases that seem easy

- **Particularly bad for arrays**
  - Wide pointer overhead for every array access
  - Reason STREAM-EP had a local block

```
local { // Squashes overhead for wide pointers
  forall (a, b, c) in zip(A, B, C) do
    a = b + alpha * c;
}
```

Locality optimizations: Improving the compiler

- **Eventually, hope to remove all local blocks**
  - Used in other benchmarks like FFT, LULESH, etc.
  - Also used inside standard distributions like Block
- **Many cases dependent on compiler improvements**
  - For other cases, we intend to move to data-centric locality assertions
    - e.g., “access the local slice of this array”
- **First step: improve part of compiler architecture**
  - Make it easier to write new optimizations
  - Reduce complexity of existing analysis

Locality optimizations: Improving the compiler

- **1.11 had two compiler passes to manage wide pointers**

Wide Pointers

All data considered local, initially

Pass #1: Insert wide pointers for **all** data

Resulting program state is overly conservative, expensive

Pass #2: Perform analysis and undo “widening” from Pass #1

Final output

Locality optimizations: Improving the compiler

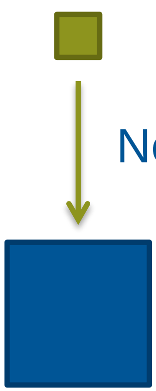
Problem: Easy for wide pointers to stick around

- First pass inserts many unnecessary wide pointers
- Second pass was often not smart enough to remove them

Solution: Merge two passes into one

- Only insert wide pointers when necessary
- Fewer variables will be wide pointers by default

Wide Pointers



All data considered local

New Pass: Analyze AST and insert wide pointers

Final output

Locality optimizations: Improving the compiler

- **New pass is less complex**
  - Less code (by several hundred lines)
  - Only handles cases that involve wide pointers
  - Easy for developers to see when/why a wide pointer was inserted

- **Easier to add new optimizations**
  - Can manipulate AST without completely restarting analysis
  - More utility functions for developers

- **Should be able to improve compiler analysis more quickly**

Locality optimizations: Better analysis

- **Problem: Fields in aggregate types are wide pointers**
  - For a simpler code-generation implementation
- **Tuples are represented as records in AST**
  - Fields “x1”, “x2”, etc. will be wide pointers
- **Stream’s arrays are eventually wrapped in a tuple**
  - Due to implementation of zip

```
forall (a, b, c) in zip(A, B, C) do ...
```

- **Those arrays are then referred to using wide pointers**
  - Adds significant overhead on every read/write on array

Locality optimizations: Better analysis

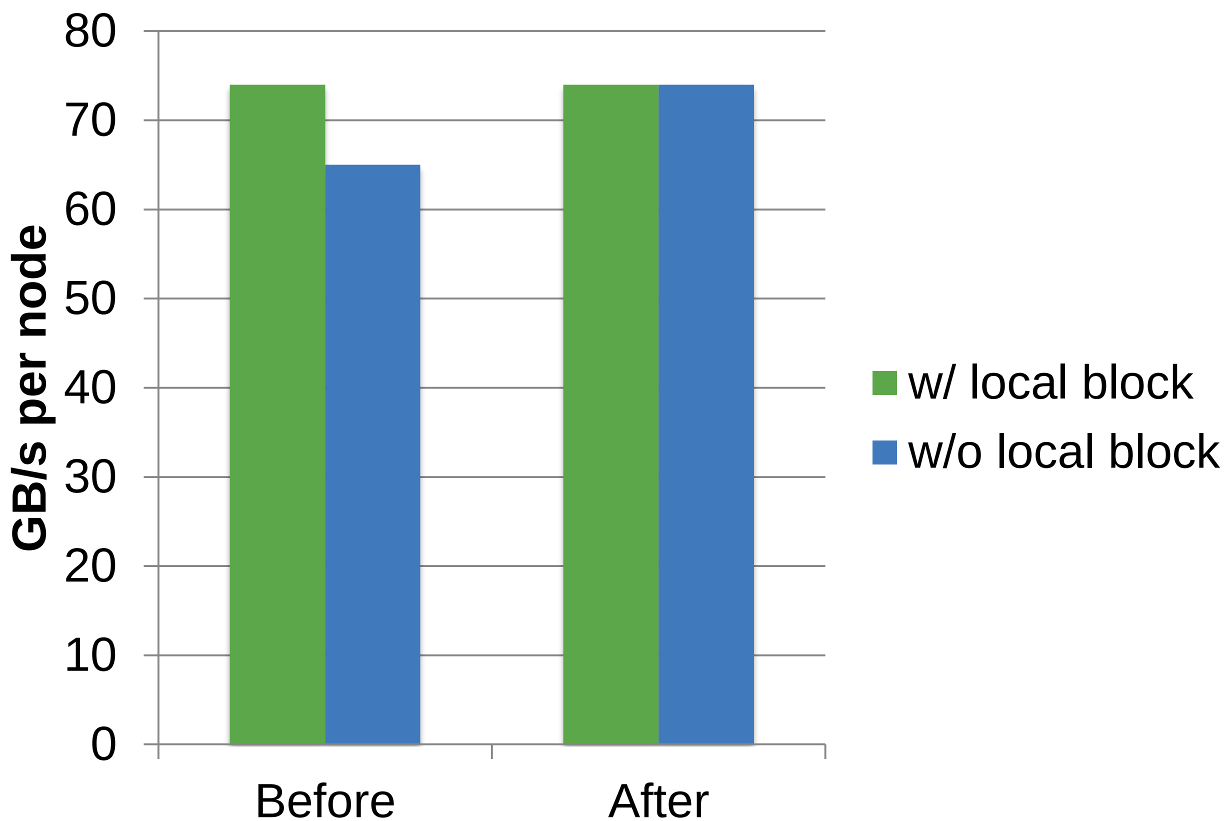
- **Solution:** Compiler should not widen every field by default

- **Only insert a wide pointer…**
  - when a field is visible to another locale
  - if a field is assigned to by another wide pointer

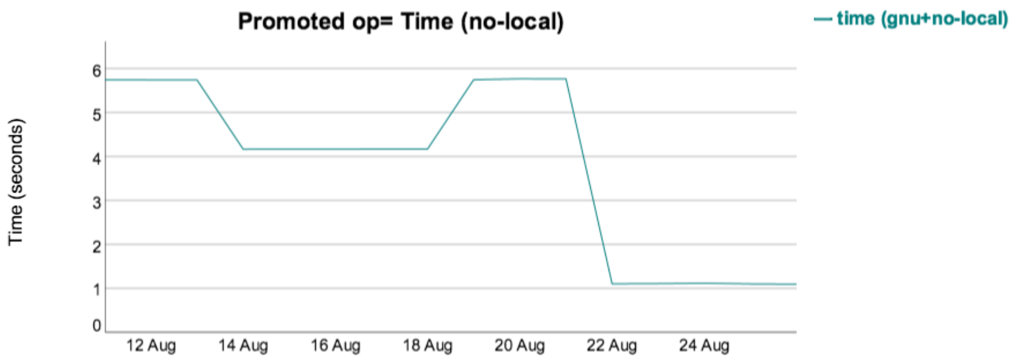
- **Reduces overhead for compiler-inserted classes/records**

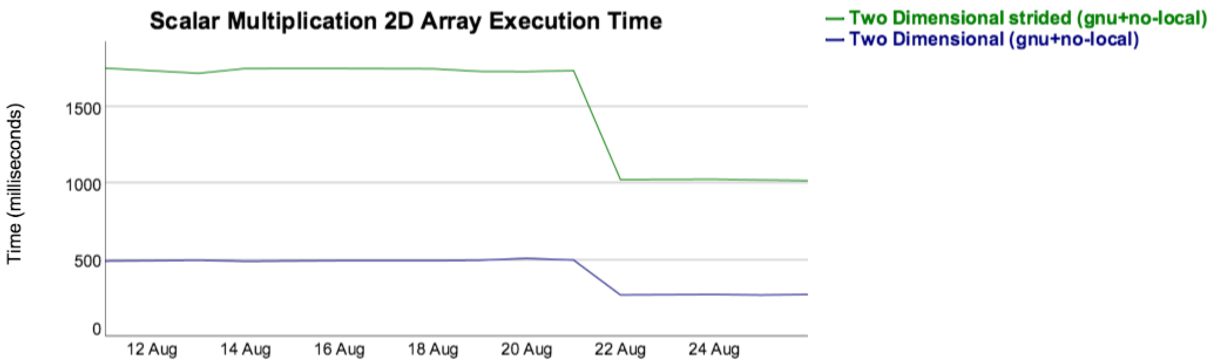
Locality optimizations: Impact

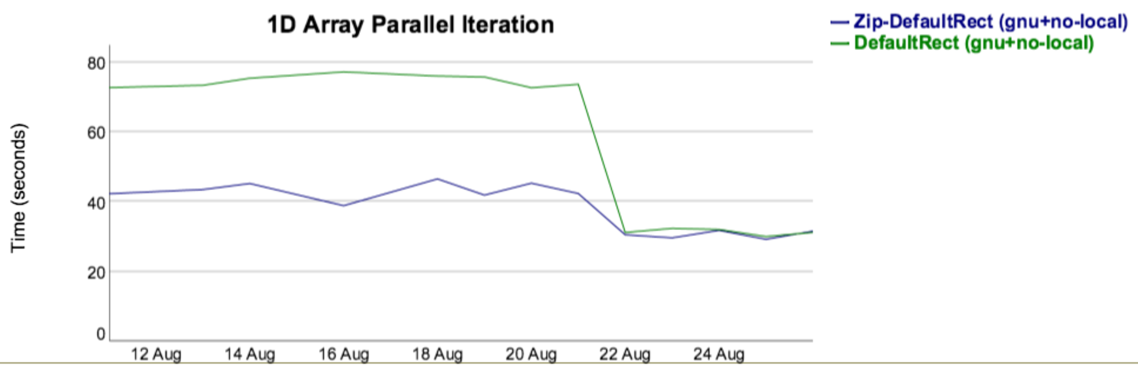
- **STREAM-EP on 16-node XC40**
  - For gasnet-mpi



Locality optimizations: Impact







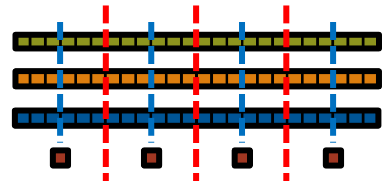
Locality optimizations: Next Steps

- **Goal: Eliminate use of local block in other benchmarks**
  - HPCC FFT
  - LULESH
  - HPL

- **Continue improving compiler's locality analysis**

- **Provide data-centric locality support**
  - Repurpose "local" keyword in variable/type/indexing contexts
  - See CHIUW 2015 talk *Data-Centric Locality in Chapel* for details

Stream Locality Summary



Stream EP

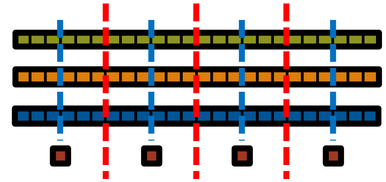
```
coforall loc in Locales do on loc {
  local {
    var A, B, C: [1..m] elemType;

    initVectors(B, C);

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

Global Stream

```
const ProblemSpace = {1..m} dmapped …;

var A, B, C: [ProblemSpace] elemType;

initVectors(B, C);

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

- **Our main performance goals for 1.12:**
  - Improve the compiler, runtime, and modules such that:
    - ✓ stream-ep performs as well as the reference
    - ✓ global stream is competitive with the reference
  - Improve compiler locality analysis and optimizations such that:
    - ❑ the **local** block in stream-ep can be removed

Stream Locality Summary



Stream EP

```
coforall loc in Locales do on loc {
  // local {
    var A, B, C: [1..m] elemType;

    initVectors(B, C);

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

Global Stream

```
const ProblemSpace = {1..m} dmapped …;

var A, B, C: [ProblemSpace] elemType;

initVectors(B, C);

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

- **Our main performance goals for 1.12:**
  - Improve the compiler, runtime, and modules such that:
    - ✓ stream-ep performs as well as the reference
    - ✓ global stream is competitive with the reference
  - Improve compiler locality analysis and optimizations such that:
    - ✓ the **local** block in stream-ep can be removed

      Note: removing it also enabled other code cleanups not shown above

Performance Improvements Summary

Performance: Summary

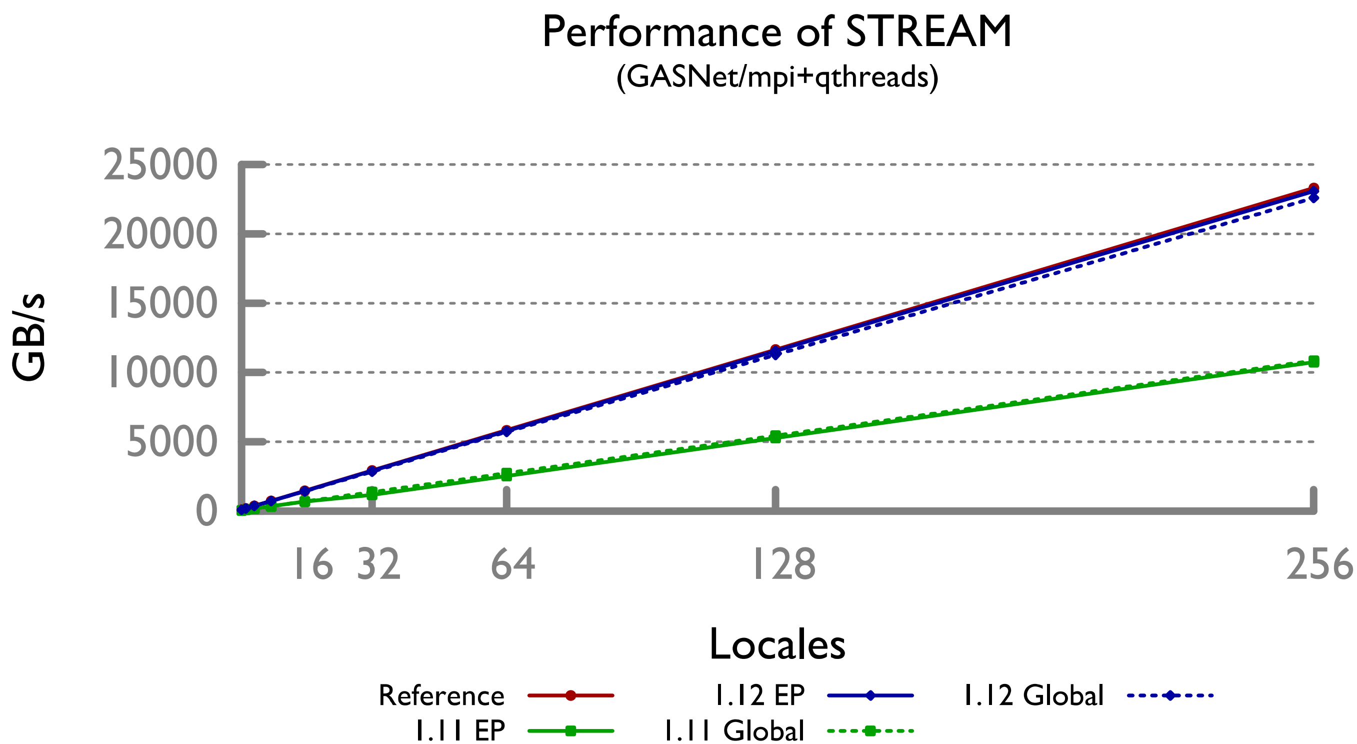
- **This release we focused on multi-locale performance**
  - We used steam as a case study to motivate optimizations
  - We achieved our performance goals for stream
    - resulting in our first truly competitive and scalable benchmark
    - as well as significant improvements for other many other benchmarks

- **Previous slides have shown performance at 16 locales**

- **Following slides will show performance up to 256 locales**
  - Run on 1-256 nodes of a Cray XC40:
    - 32-core (64 HT) Haswell Processors
    - 128 GB RAM per node
    - GCC 5.1.0

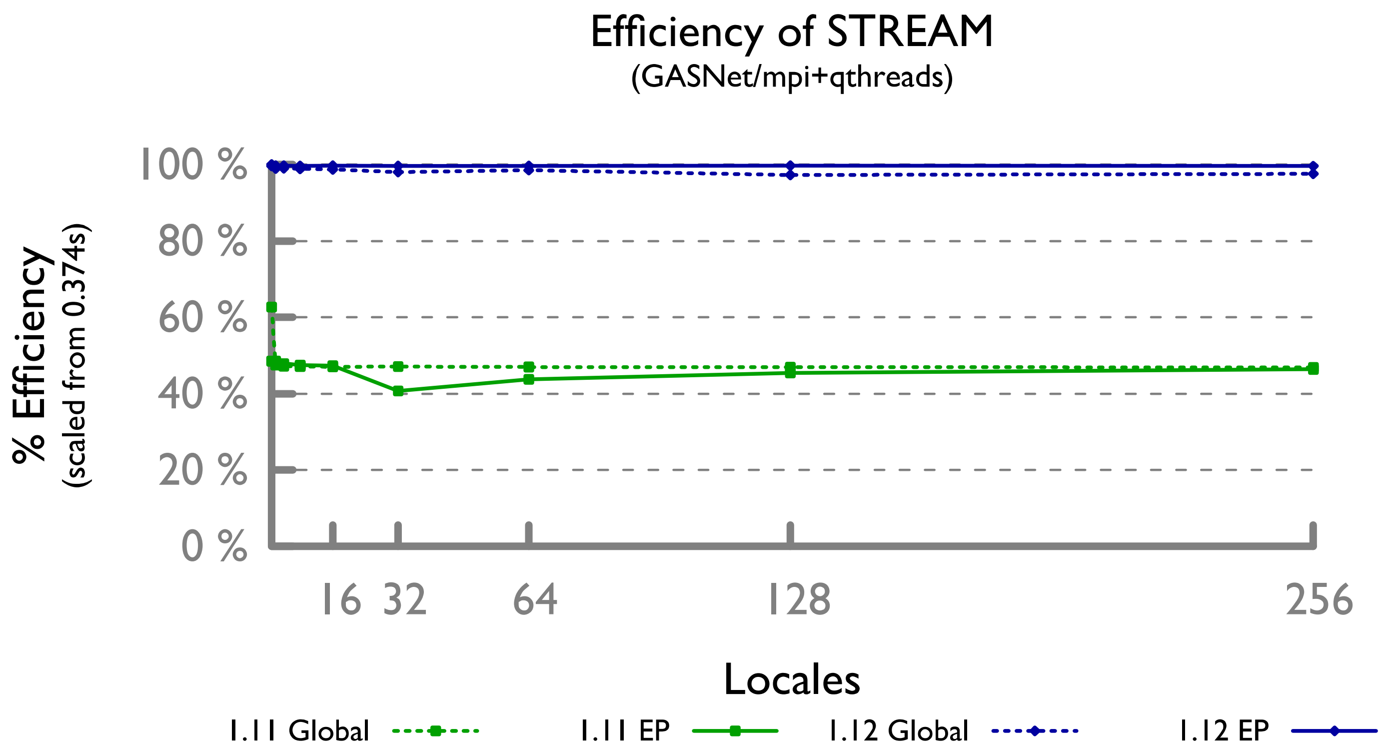
Performance: Summary

- **Performance trends are the same at higher node counts**
  - Performance has more than doubled since last release
  - EP is on par with the reference
  - Global is also very competitive (spinning up parallelism is scaling)



Performance: Summary

- **Performance trends are the same at higher node counts**
  - Performance has more than doubled since last release
  - EP is on par with the reference
  - Global is also very competitive (spinning up parallelism is scaling)



Performance: Next Steps

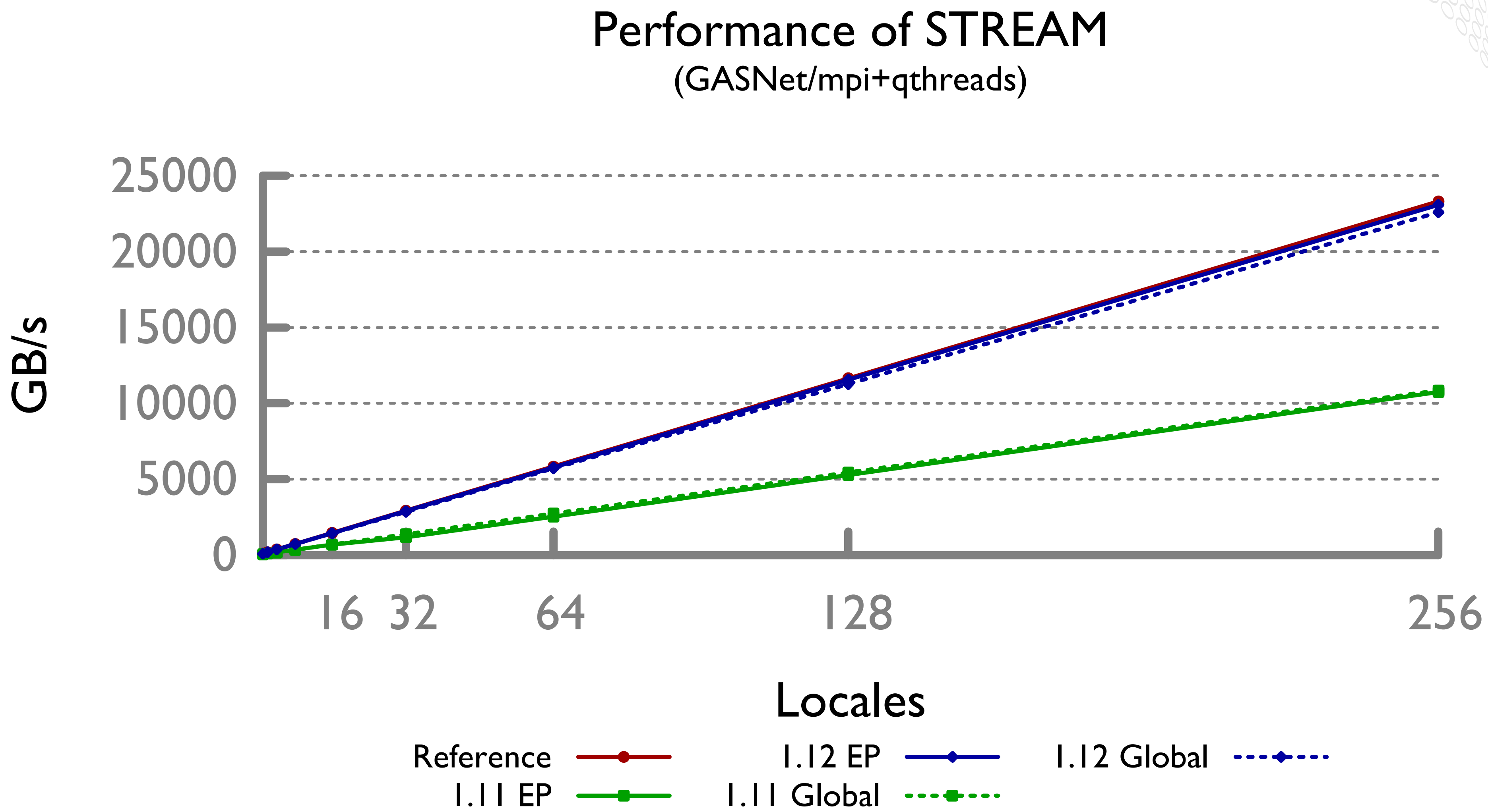
- **Complete remaining work for stream**
  - Resolve ugni hugepage performance issue
    - and possibly gasnet-aries as well
  - Determine if global stream performance can be improved
    - particularly as node counts grow
  - Compare/improve other variants of writing stream
    - e.g., promoted operator version; domain-based iteration + indexing

- **Optimize more complicated multi-locale benchmarks**
  - Likely starting with RA, other HPCC benchmarks, and ISx
    - possibly working towards an HPCC entry for SC16

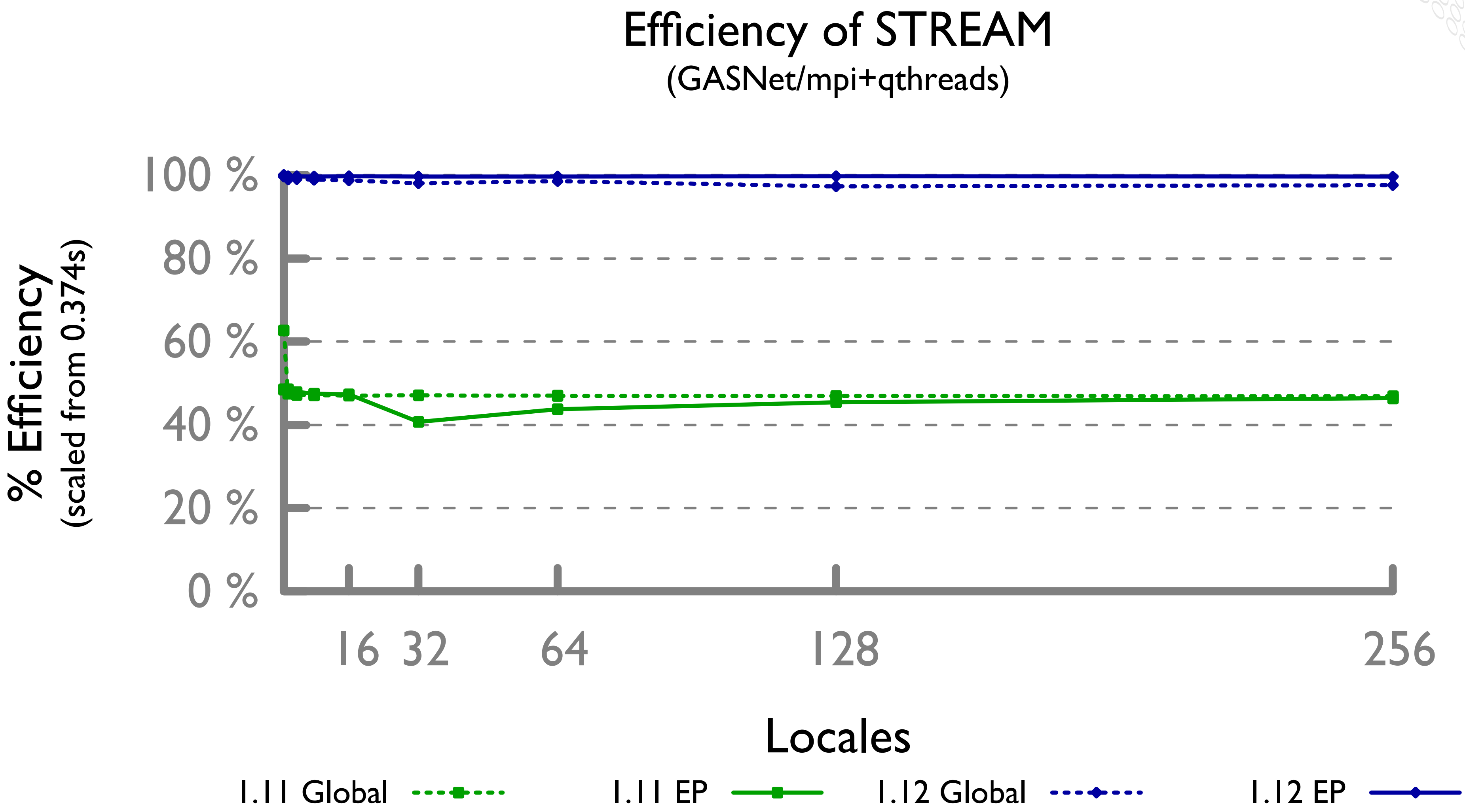
- **Continue improving compiler locality optimizations**
  - Guided by removing “local” blocks from other benchmarks

Appendix: Larger Stream Scalability Graph Images

Performance: Summary



Performance: Summary



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.

http://chapel.cray.com chapel_info@cray.com http://sourceforge.net/projects/chapel/