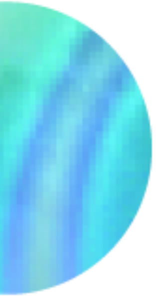
Affine Loop Optimization using Modulo Unrolling in CHAPEL



Aroon Sharma, Joshua Koehler, Rajeev Barua

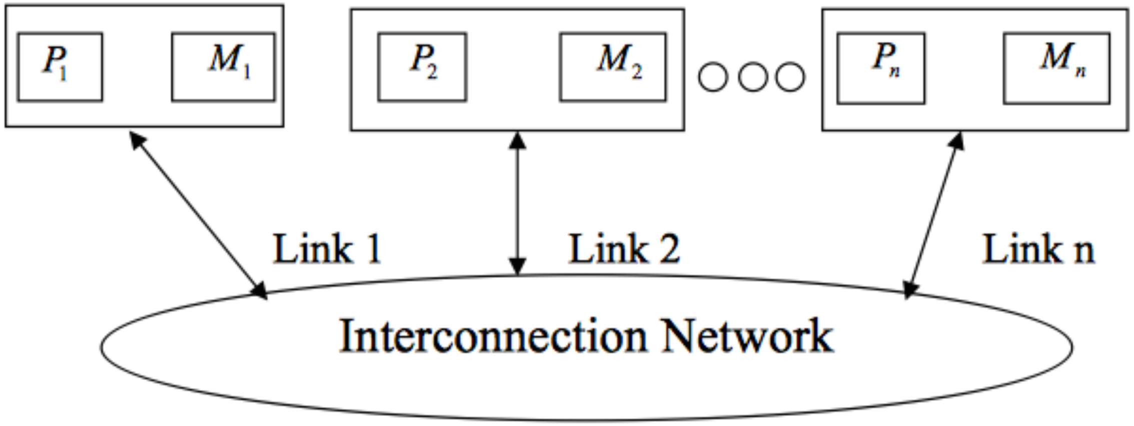
LTS POC: Michael Ferguson

Overall Goal

- Improve the runtime of certain types of parallel computers
  - In particular, message passing computers
- Approach
  - Start with an explicitly parallel program
  - Compile using our method to minimize communication cost between nodes of the parallel computer
- Advantage: Faster scientific and data processing computation

Message Passing Architectures

- Communicate data among a set of processors with separate address spaces using messages
  - Remote Direct Memory Access (RDMA)
- High Performance Computing Systems
- 100-100,000 compute nodes
- Complicates compilation



PGAS Languages

- Partitioned Global Address Space (PGAS)
- Provides illusion of a shared memory system on top of a distributed memory system
- Allows the programmer to reason about locality without dealing with low-level data movement
- Example - CHAPEL

CHAPEL

- PGAS language developed by Cray Inc.
- Programmers express parallelism explicitly
- Features to improve programmer productivity
- Targets large scale and desktop systems
- Opportunities for performance optimizations!

Our Work's Contribution

We present an optimization for parallel loops with **affine array accesses** in **CHAPEL**.

The optimization uses a technique known as **modulo unrolling** to aggregate messages and improve the runtime performance of loops for distributed memory systems using **message passing.**

Outline

- **Introduction and Motivation**
- Modulo Unrolling
- Optimized Cyclic and Block Cyclic Dists
- Results

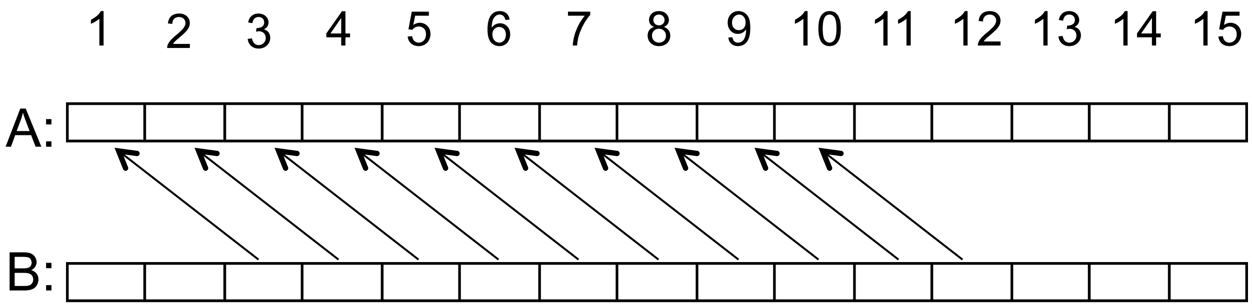
Affine Array Accesses

- Most common type of array access in scientific codes
  - A[i], A[j], A[3], A[i+1], A[i + j], A[2i + 3j]
  - A[i, j], A[3i, 5j]
- Array accesses are affine if the access on each dimension is a linear expression of the loop indices
  - E.g. A[ai + bj + c] for a 2D loop nest
  - Where a, b, and c are constant integers

Example Parallel Loop in CHAPEL

```
forall i in 1..10 do
        A[i] = B[i+2];
```

What happens when the data is distributed?

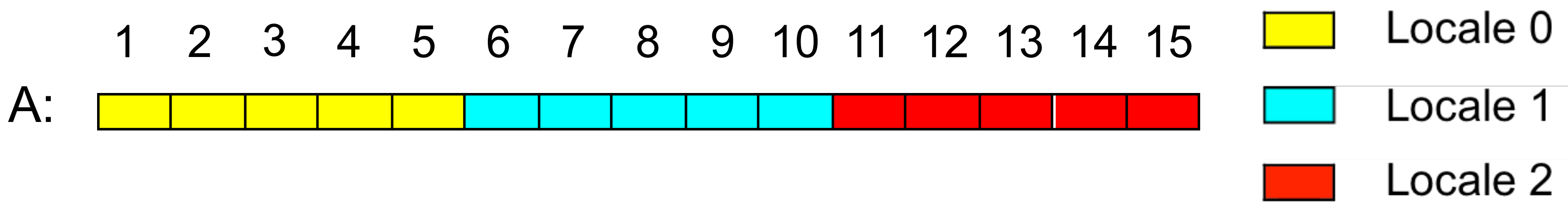


Data Distributions in CHAPEL

- Describe how data is allocated across locales for a given program
  - A locale is a unit of a distributed computer (processor and memory)
- Users can distribute data with CHAPEL’s standard modules or create their own distributions
- Distributions considered in this study
  - Cyclic
  - Block
  - Block Cyclic

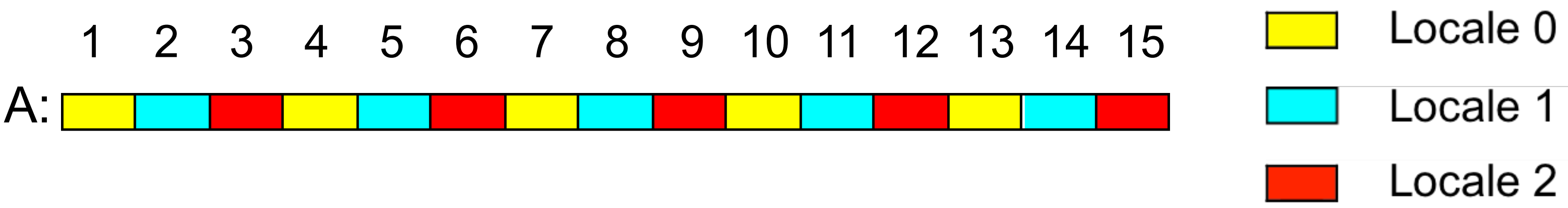
Data Distributions in CHAPEL - Block

```
use BlockDist;

var domain = {1..15};
var distribution = domain dmapped Block(boundingBox=domain);
var A: [distribution] int;
// A is now distributed in the following fashion
```



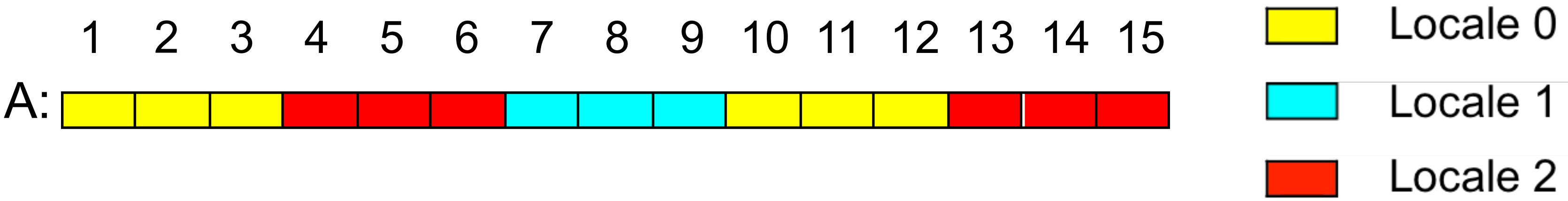
Data Distributions in CHAPEL - Cyclic

```
use CyclicDist;

var domain = {1..15};
var distribution = domain dmapped Cyclic(startIdx=domain.low);
var A: [distribution] int;
// A is now distributed in the following fashion
```



Data Distributions in CHAPEL – Block Cyclic

```
use BlockCycDist;

var domain = {1..15};
var distribution = dom dmapped BlockCyclic(blocksize=3);
var A: [distribution] int;
// A is now distributed in the following fashion
```



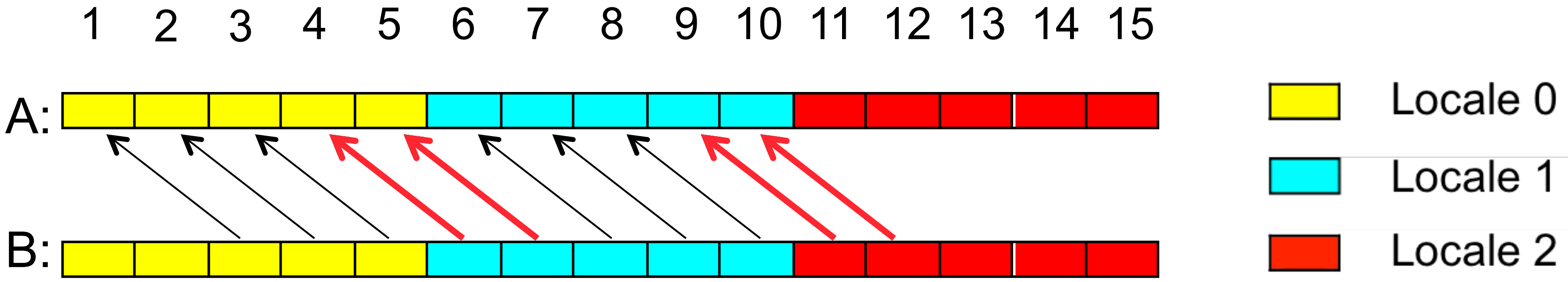
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A:	🟨	🟨	🟨	🟥	🟥	🟥	🟦	🟦	🟦	🟨	🟨	🟨	🟥	🟥	🟥

🟨 Locale 0
🟦 Locale 1
🟥 Locale 2

*similar code is used to distributed multi-dimensional arrays

Distributed Parallel Loop in CHAPEL

```
forall i in 1..10 do
        A[i] = B[i+2];
```



- 4 Messages
  - Locale 1 → Locale 0 containing B[6]
  - Locale 1 → Locale 0 containing B[7]
  - Locale 2 → Locale 1 containing B[11]
  - Locale 2 → Locale 1 containing B[12]

Data Communication in CHAPEL can be Improved

- Locality check at each loop iteration
  - Is B[i+2] local or remote?
- Each message contains only 1 element
- We could have aggregated messages
  - Using GASNET strided get/put in CHAPEL
  - Locale 1 → Locale 0 containing B[6], B[7]
  - Locale 2 → Locale 1 containing B[11], B[12]
- Growing problem
  - Runtime increases for larger problems and more complex data distributions

How to improve this?

- Use knowledge about how data is distributed and loop access patterns to aggregate messages and reduce runtime of affine parallel loops
- We are not trying to
  - Apply automatic parallelization to CHAPEL
  - Come up with a new data distribution
  - Bias or override the programmer to a particular distribution
- We are trying to
  - Improve CHAPEL's existing data distributions to perform better than their current implementation

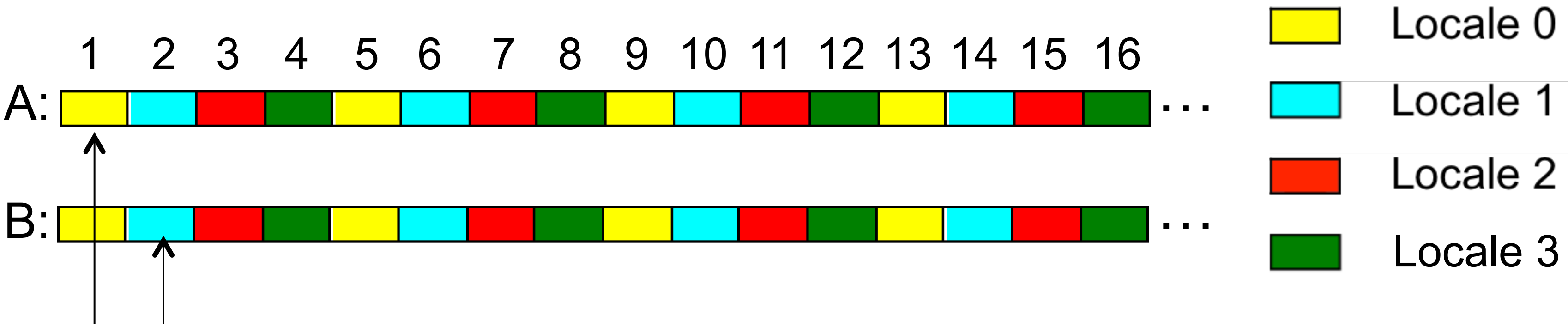
Modulo Unrolling – See Barua1999

- Method to statically disambiguate array accesses at compile time
- Unroll the loop by factor = number of locales
- Each array access will reside on a single locale across loop iterations
- Applicable for **Cyclic** and **Block Cyclic**

Modulo Unrolling Example

```
for i in 1..99 {
   A[i] = A[i] + B[i+1];
}
```

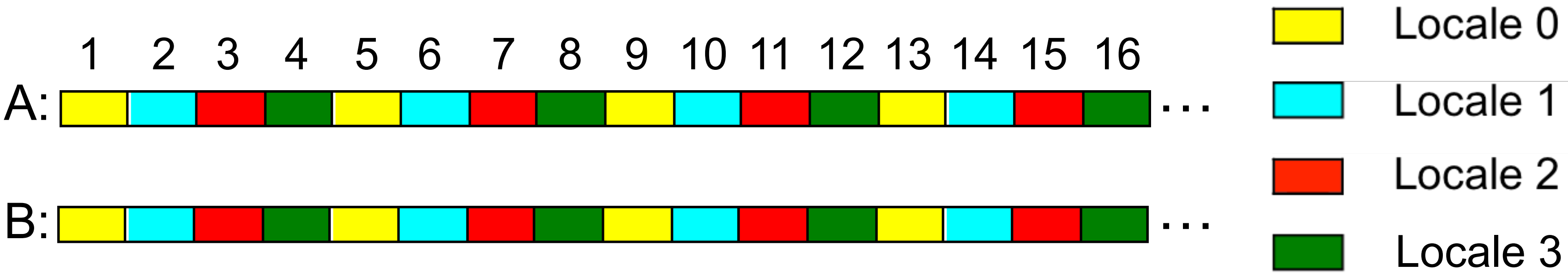
Each iteration of the loop accesses data on a different locale



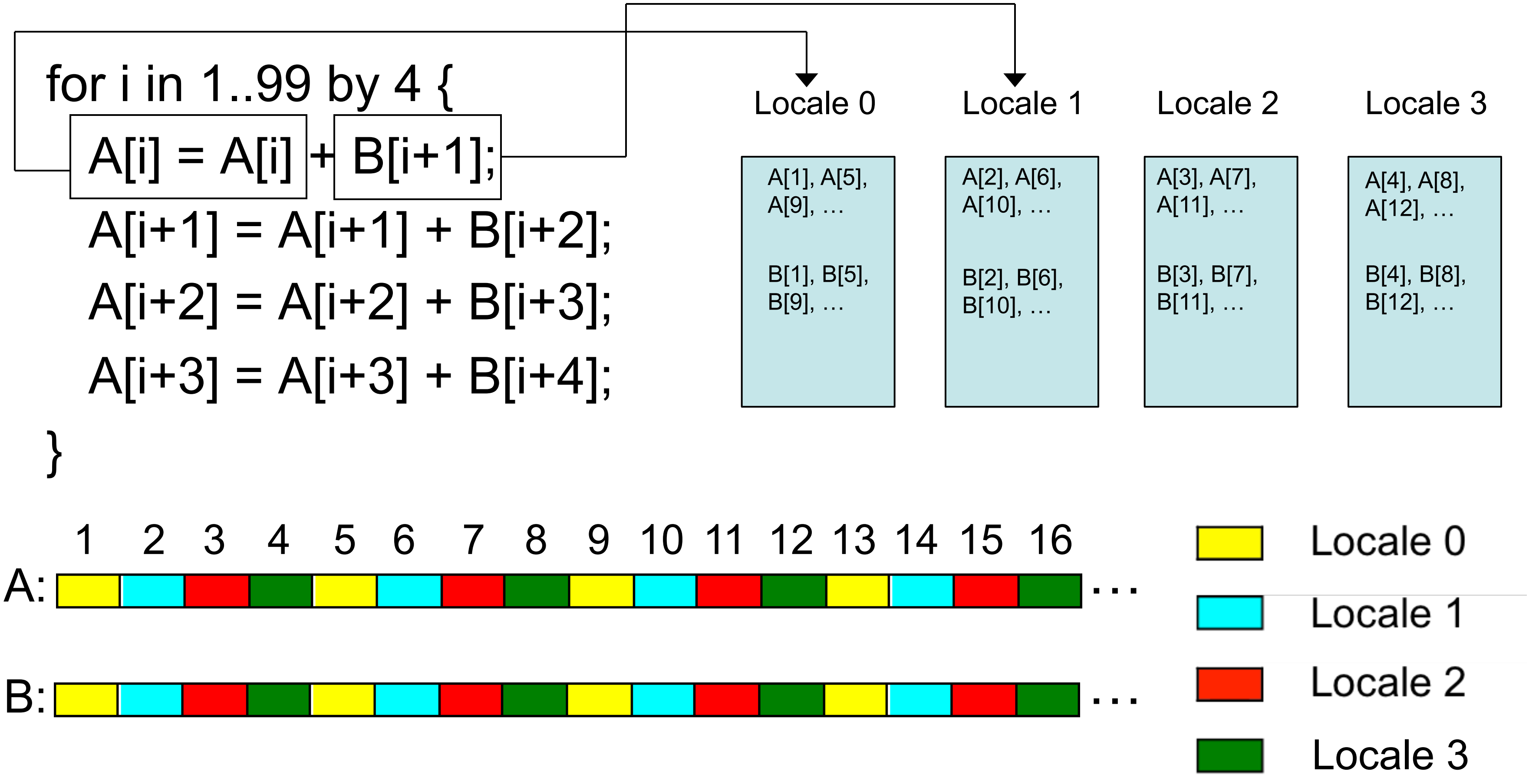
Modulo Unrolling Example

```
for i in 1..99 by 4 {
    A[i] = A[i] + B[i+1];
    A[i+1] = A[i+1] + B[i+2];
    A[i+2] = A[i+2] + B[i+3];
    A[i+3] = A[i+3] + B[i+4];
}
```

Loop unrolled by a factor of 4 automatically by the compiler



Modulo Unrolling Example



```
for i in 1..99 by 4 {
  A[i] = A[i] + B[i+1];
  A[i+1] = A[i+1] + B[i+2];
  A[i+2] = A[i+2] + B[i+3];
  A[i+3] = A[i+3] + B[i+4];
}
```

Locale 0	Locale 1	Locale 2	Locale 3
A[1], A[5], A[9], …	A[2], A[6], A[10], …	A[3], A[7], A[11], …	A[4], A[8], A[12], …
B[1], B[5], B[9], …	B[2], B[6], B[10], …	B[3], B[7], B[11], …	B[4], B[8], B[12], …

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
A:	🟨	🟦	🟥	🟩	🟨	🟦	🟥	🟩	🟨	🟦	🟥	🟩	🟨	🟦	🟥	🟩	…
B:	🟨	🟦	🟥	🟩	🟨	🟦	🟥	🟩	🟨	🟦	🟥	🟩	🟨	🟦	🟥	🟩	…

🟨 Locale 0

🟦 Locale 1

🟥 Locale 2

🟩 Locale 3

How do we apply this concept in Chapel?

CHAPEL Zippered Iteration

- Can be used with parallel for loops
- Leader iterator
  - Creates tasks to implement parallelism and assigns iterations to tasks
- Follower iterator
  - Carries out work specified by leader (yielding elements) usually serially

CHAPEL Zippered Iteration

Follower iterators of A, B, and C will be responsible for doing work for each task

```
forall (a, b, c) in zip(A, B, C) {
  code…
}
```

Because it is first, A's leader iterator will divide up the work among available tasks

*See Chamberlain2011 for more detail on leader/follower semantics

CHAPEL Zippered Iteration

- It turns out any parallel forall loop with affine array accesses can be written using zippered iteration over array slices

```
forall i in 1..10 {
   A[i] = B[i+2];
}
```

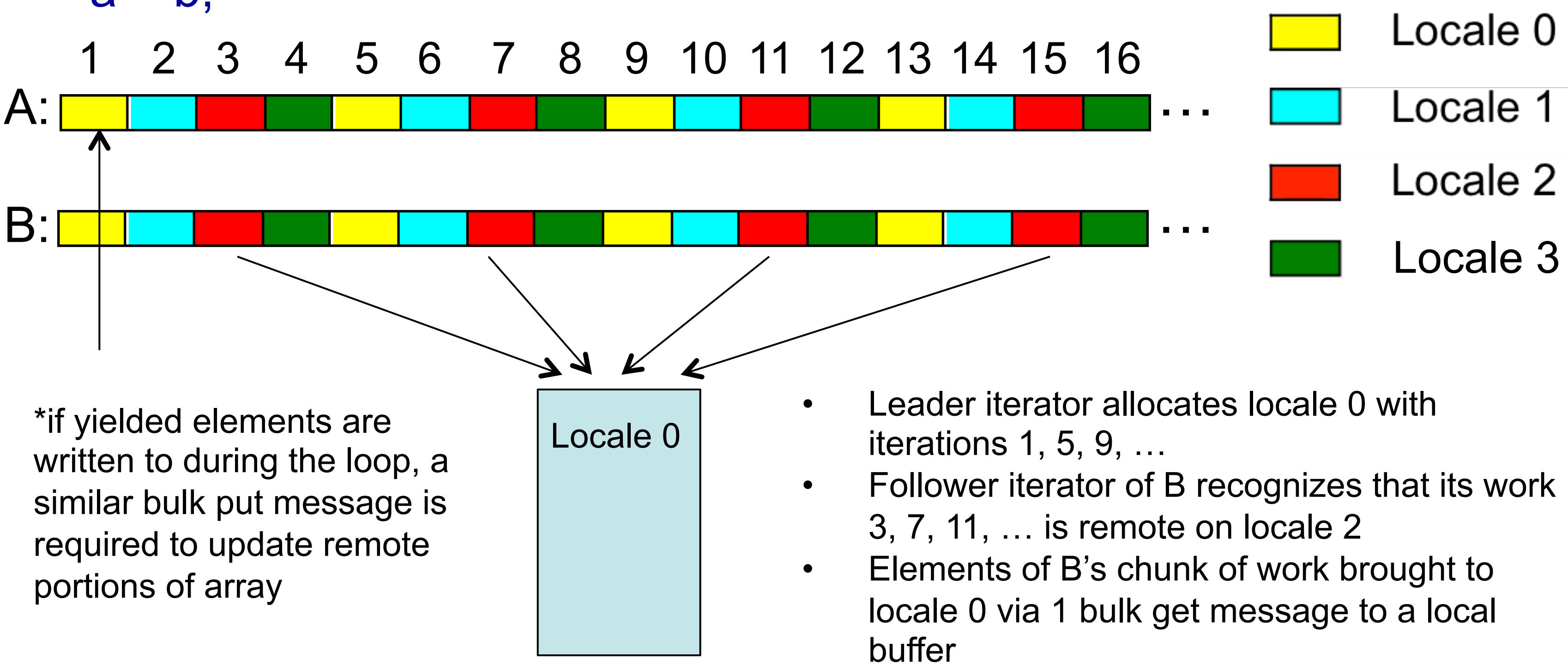
Zippered iteration →

```
forall (a,b) in zip(A[1..10], B[3..12]){
   a = b;
}
```

Implement modulo unrolling and message aggregation within the leader and follower iterators of the Block Cyclic and Cyclic distributions!

Modulo Unrolling in CHAPEL Cyclic Distribution

```
forall (a,b) in zip(A[1..10], B[3..12]) do
    a = b;
```



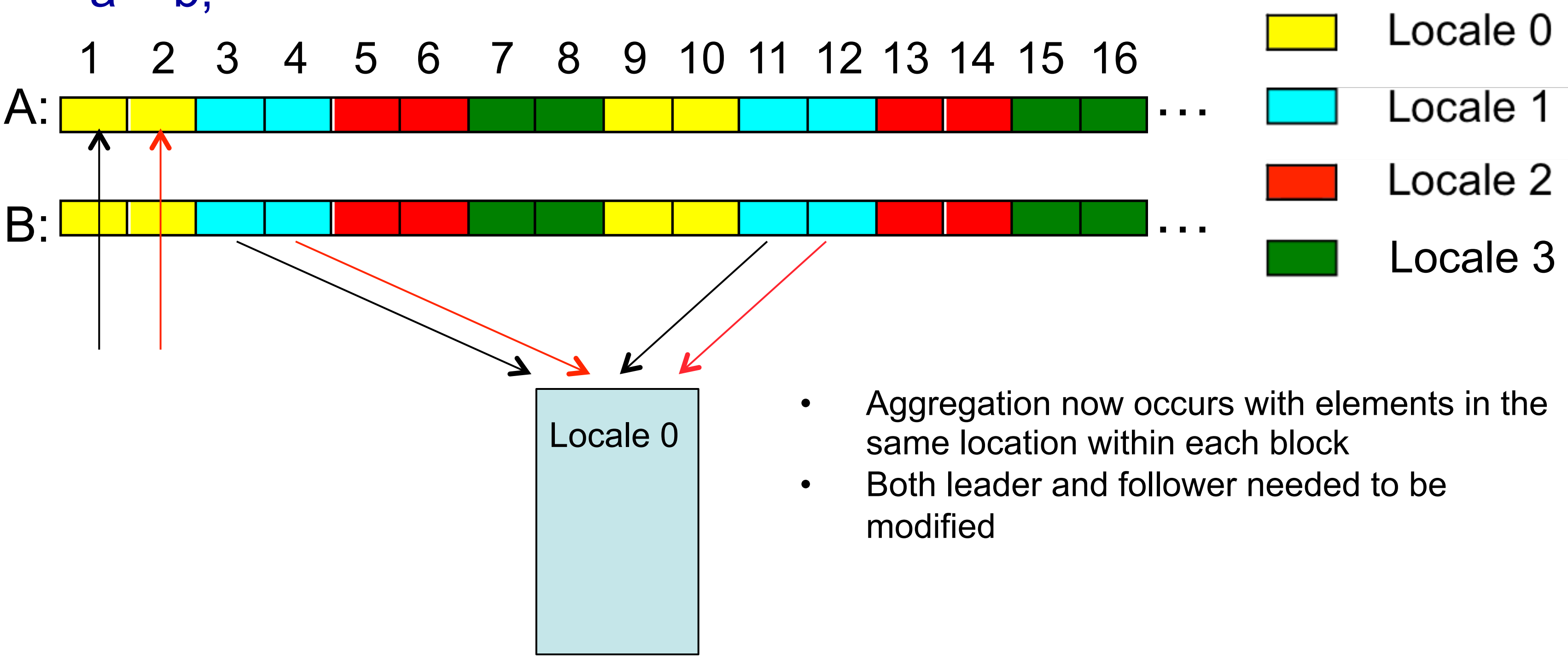
🟨 Locale 0
🟦 Locale 1
🟥 Locale 2
🟩 Locale 3

*if yielded elements are written to during the loop, a similar bulk put message is required to update remote portions of array

- Leader iterator allocates locale 0 with iterations 1, 5, 9, …
- Follower iterator of B recognizes that its work 3, 7, 11, … is remote on locale 2
- Elements of B's chunk of work brought to locale 0 via 1 bulk get message to a local buffer
- Elements of local buffer are now yielded back to loop header

Modulo Unrolling in CHAPEL Block Cyclic Distribution

```
forall (a,b) in zip(A[1..10], B[3..12]) do
  a = b;
```



- Aggregation now occurs with elements in the same location within each block
- Both leader and follower needed to be modified

Outline

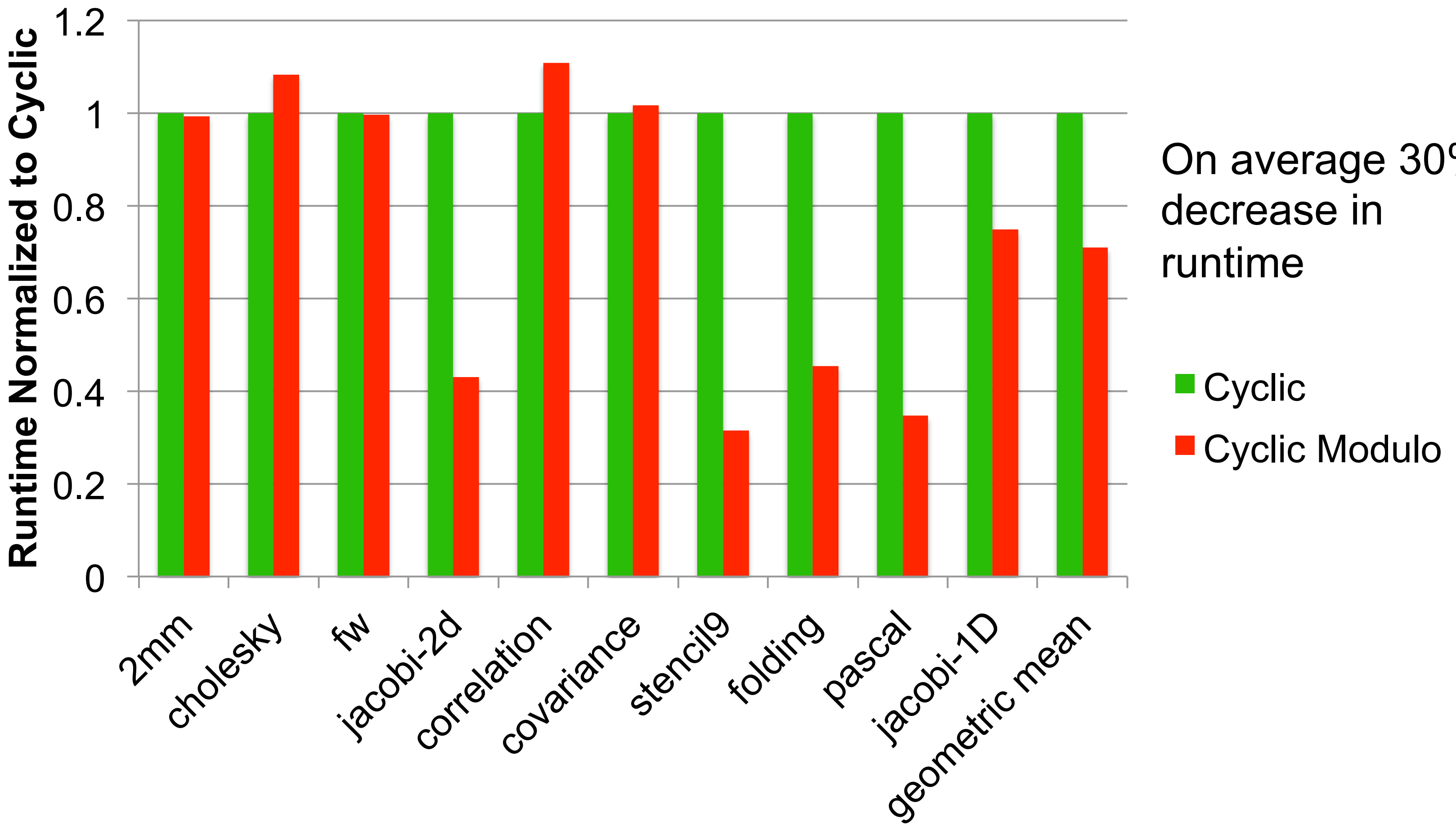
- Introduction and Motivation
- Modulo Unrolling
- Optimized Cyclic and Block Cyclic Dists
- **Results**

Benchmarks

Name	Dimension	Description	Input (elements)
2mm	2D	Matrix mulitplication	16 x 16
cholesky	2D	Cholesky decomposition	128 x 128
jacobi-2d	2D	Jacobi relaxation	400 x 400
jacobi-1d	1D	Jacobi relaxation	10000
stencil9	2D	9-point stencil calculation	400 x 400
folding	1D	Sum consecutive elements of array using strided access pattern	N = 50400, 10 iterations
pascal	1D	Computes rows of pascal's triangle	N1 = 10000, N2 = 10003
covariance	2D	Covariance calculation	128 x 128
correlation	2D	Correlation	64 x 64

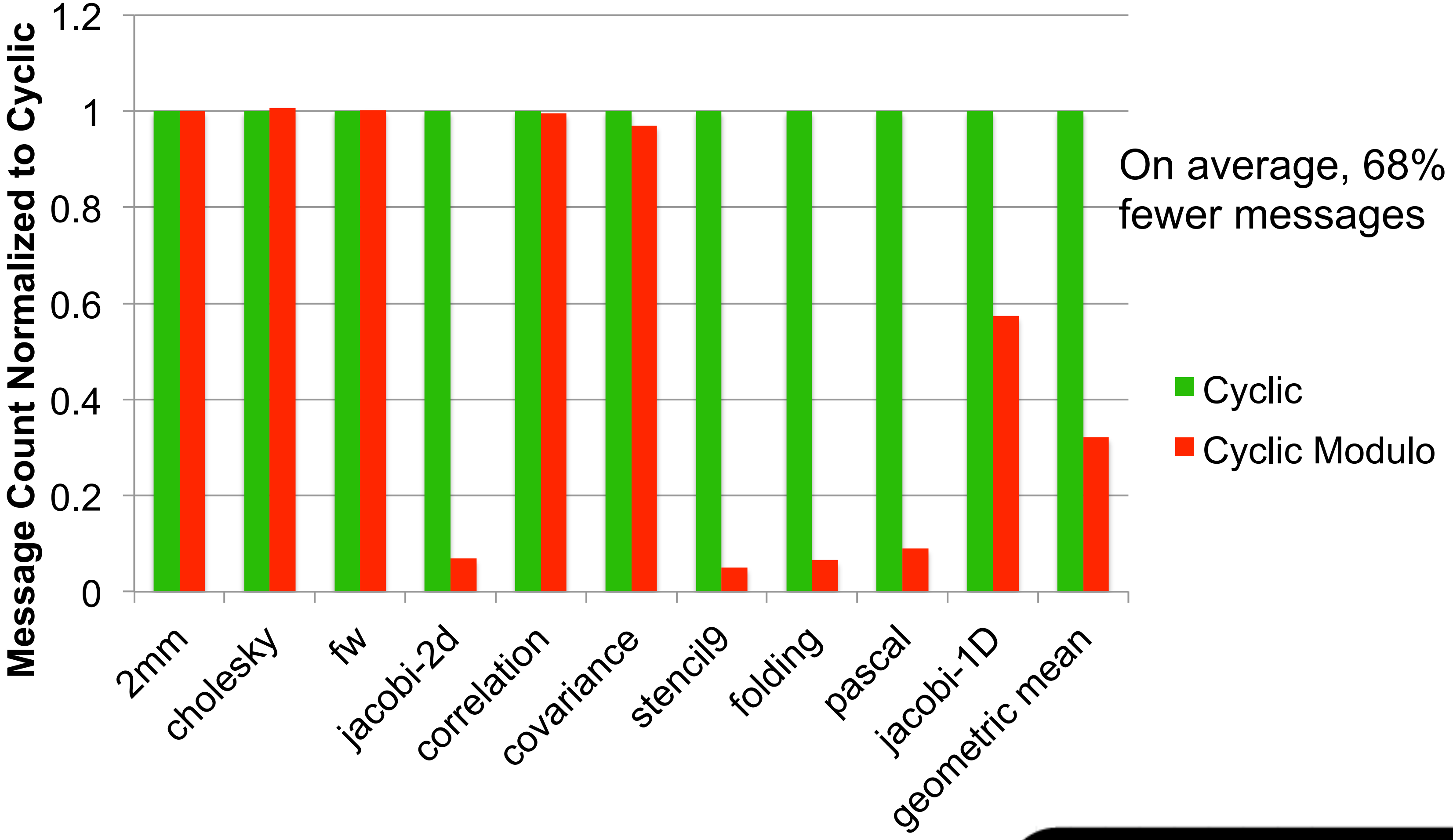
* Data collected on 10 node Golgatha cluster at LTS

Cyclic vs. Cyclic Modulo Normalized Runtime



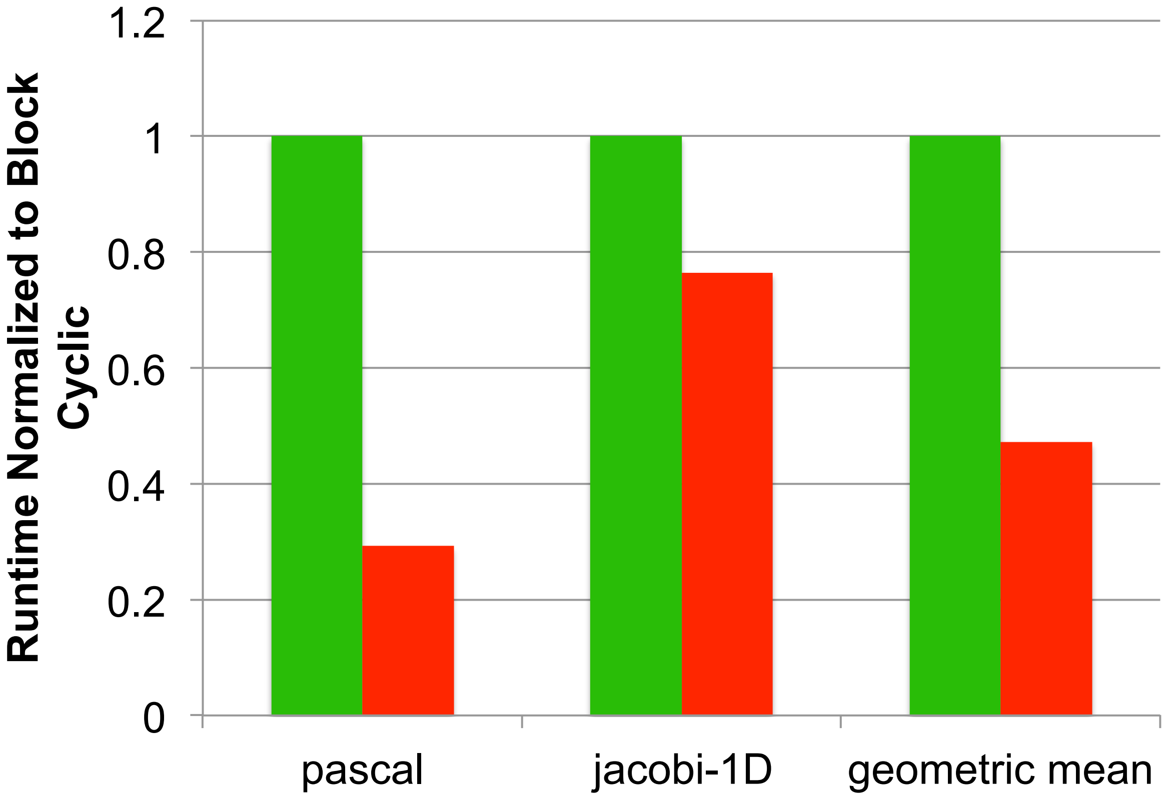
On average 30% decrease in runtime

Cyclic vs. Cyclic Modulo Normalized Message Counts



On average, 68% fewer messages

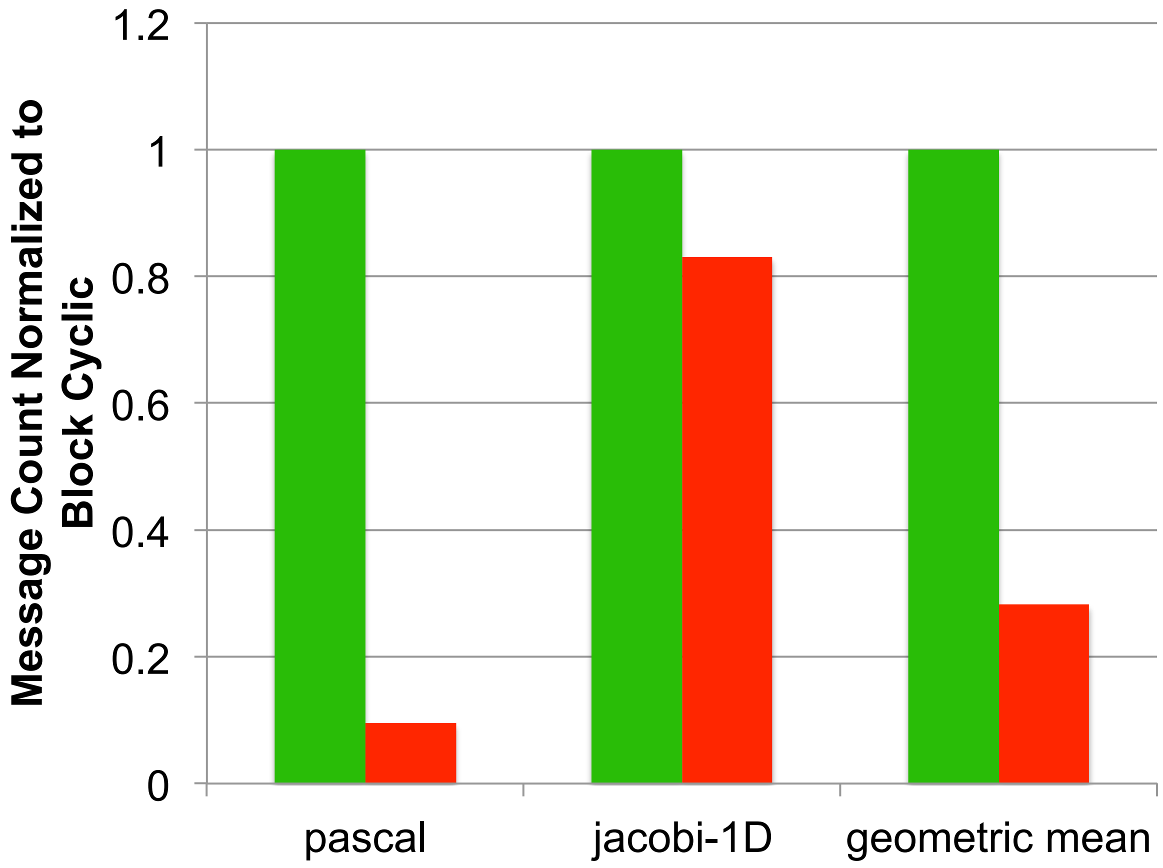
Block Cyclic vs. Block Cyclic Modulo Normalized Runtime



On average 52% decrease in runtime

🟩 Block Cyclic
🟥 Block Cyclic Modulo

Block Cyclic vs. Block Cyclic Modulo Normalized Message Count



On average 72% fewer messages

🟩 Block Cyclic

🟥 Block Cyclic Modulo

Conclusion

- We've presented optimized Cyclic and Block Cyclic distributions in CHAPEL that perform modulo unrolling
- Our results for Cyclic Modulo and Block Cyclic Modulo show improvements in runtime and message counts for affine programs over existing distributions

References

[1] Barua, R., & Lee, W. (1999). Maps: A Compiler-Managed Memory System for Raw Machine. *Proceedings of the 26th International Symposium on Computer Architecture*, (pp. 4-15).

[2] *User-Defined Parallel Zippered Iterators in Chapel,* Chamberlain, Choi, Deitz, Navarro; October 2011

[3] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In ETAPS International Conference on Compiler Construction (CC'2010), pages 283–303, Mar. 2010.

References

[4] Compile-time techniques for data distribution in distributed memory machines. J Ramanujam, P Sadayappan - Parallel and Distributed Systems, IEEE Transactions on, 1991

[5] Chen, Wei-Yu, Costin Iancu, and Katherine Yelick. "Communication optimizations for fine-grained UPC applications." *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*. IEEE, 2005.

Questions?

Backup Slides

CHAPEL Zippered Iteration

- Iterators
  - Chapel construct similar to a function
  - return or “yield” multiple values to the callsite
  - Can be used in loops

```
iter fib(n: int) {
  var current = 0,
  next = 1;
  for i in 1..n {
    yield current;
    current += next;
    current <=> next;
  }
}
```

Being used in a loop

```
for f in fib(5) {
  writeln(f);
}
```

f is the next yielded value of fib after each iteration

Output: 0, 1, 1, 2, 3

CHAPEL Zippered Iteration

- Zippered Iteration
  - Multiple iterators of the same size are traversed simultaneously
  - Corresponding iterations processed together

```
for (i, f) in zip(1..5, fib(5)) {
  writeln("Fibonacci ", i, " = ", f);
}
```

Output

```
Fibonacci 1 = 0
Fibonacci 2 = 1
Fibonacci 3 = 1
Fibonacci 4 = 2
Fibonacci 5 = 3
```

What about Block?

2D Jacobi Example – Transformed Pseudocode

```
forall (k1,k2) in {0..1, 0..1} {
  if A[2 + 3k1, 2 + 3k2].locale.id == $ then on $ {
    buf_north = get(A[2+3k1..4+3k1, 2+3k2-1..4+3k2-1]);
    buf_south = get(A[2+3k1..4+3k1, 2+3k2+1..4+3k2+1]);
    buf_east = get(A[2+3k1-1..4+3k1-1, 2+3k2..4+3k2]);
    buf_west = get(A[2+3k1+1..4+3k1+1, 2+3k2..4+3k2]);

    LB_i = 2+3k1;
    LB_j = 2+3k2;

    forall(i, j) in {2+3k1..4+3k1, 2+3k2..4+3k2} {
      $A_{new}$[i,j] = (buf_north[i-LB_i, j-LB_j] + buf_south[i-LB_i, j-LB_j] +
                     buf_east[i-LB_i, j-LB_j] + buf_west[i-LB_i, j-LB_j])/4.0;
    }
}
```

For each block in parallel

Bring in remote portions of array footprint locally

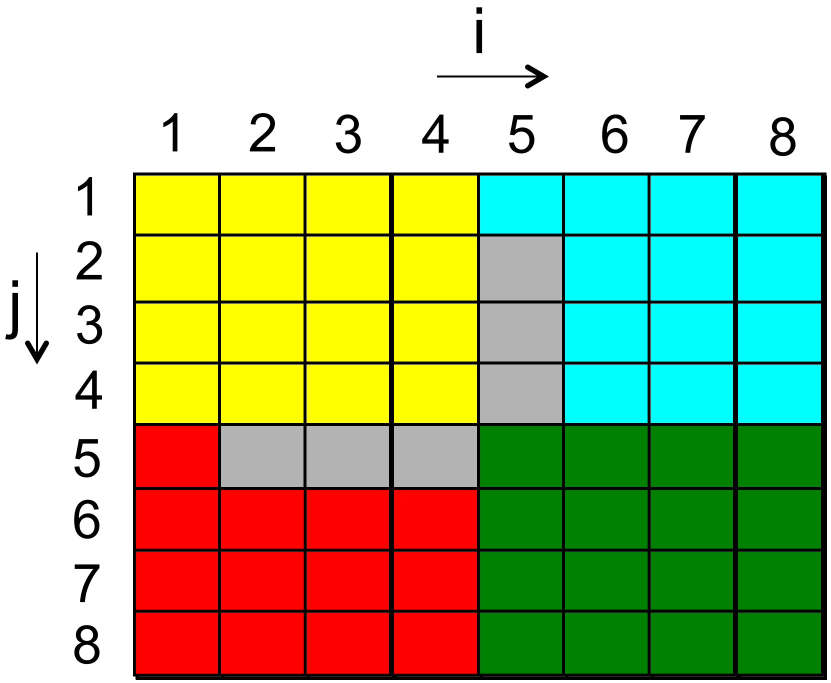
Do the computation using local buffers

What about Block?

- It seems that data distributed using Block naturally results in fewer messages for many benchmarks
- Makes sense because many benchmarks in scientific computing access nearest neighbor elements
- Nearest neighbor elements are more likely to reside on the same locale
- Could we still do better and aggregate messages?

What about Block?

2D Jacobi Example



- 2 remote blocks per locale → 2 messages
- 8 messages with aggregation
- 24 messages without
- Messages without aggregation grows as problem size grows

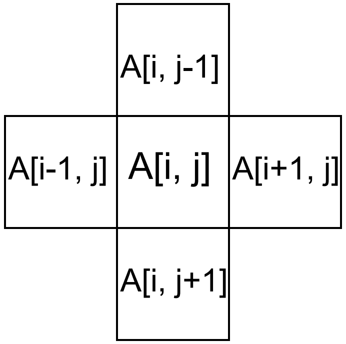
🟨 Locale 0

🟦 Locale 1

🟥 Locale 2

🟩 Locale 3

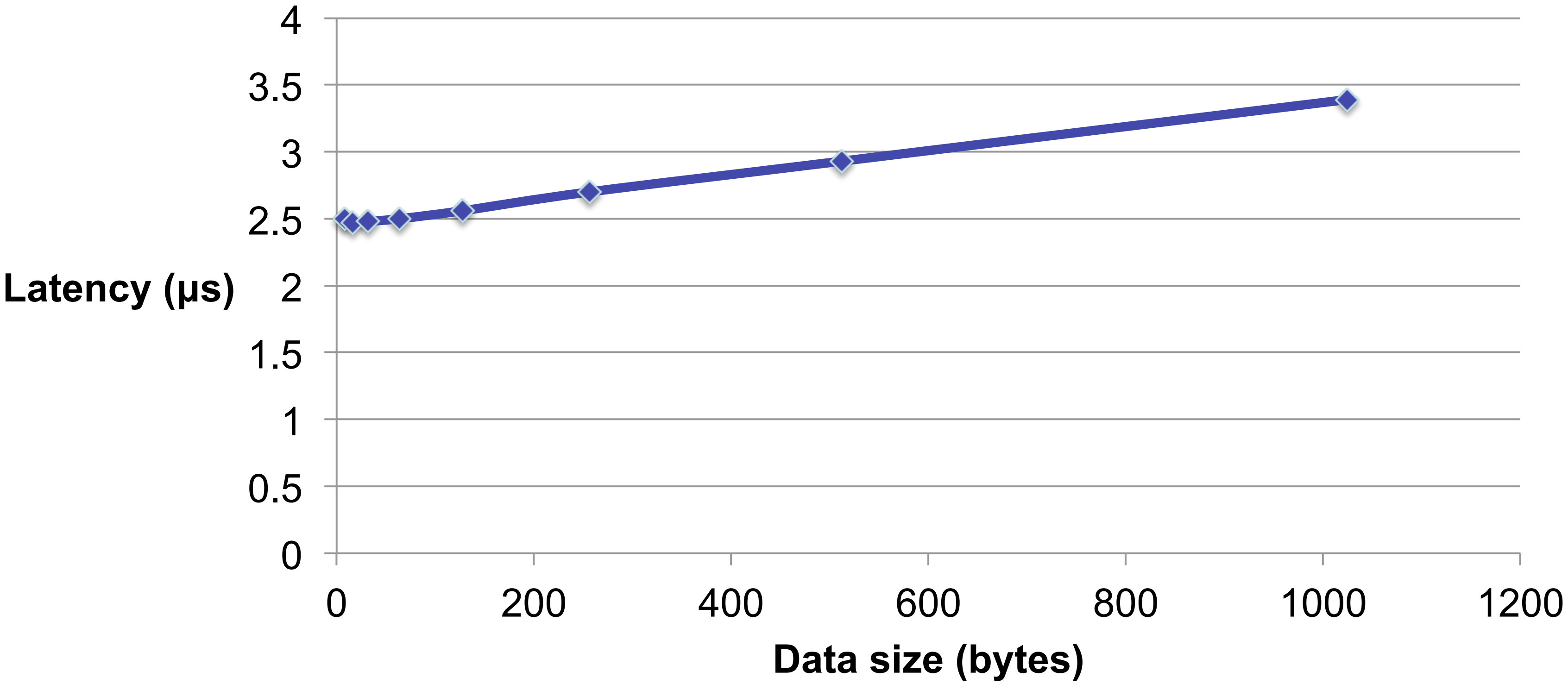
```
forall (i,j) in {2..7, 2..7} {
  Anew[i,j] = (A[i+1, j] + A[i-1, j] + A[i, j+1] + A[i, j-1])/4.0;
}
```



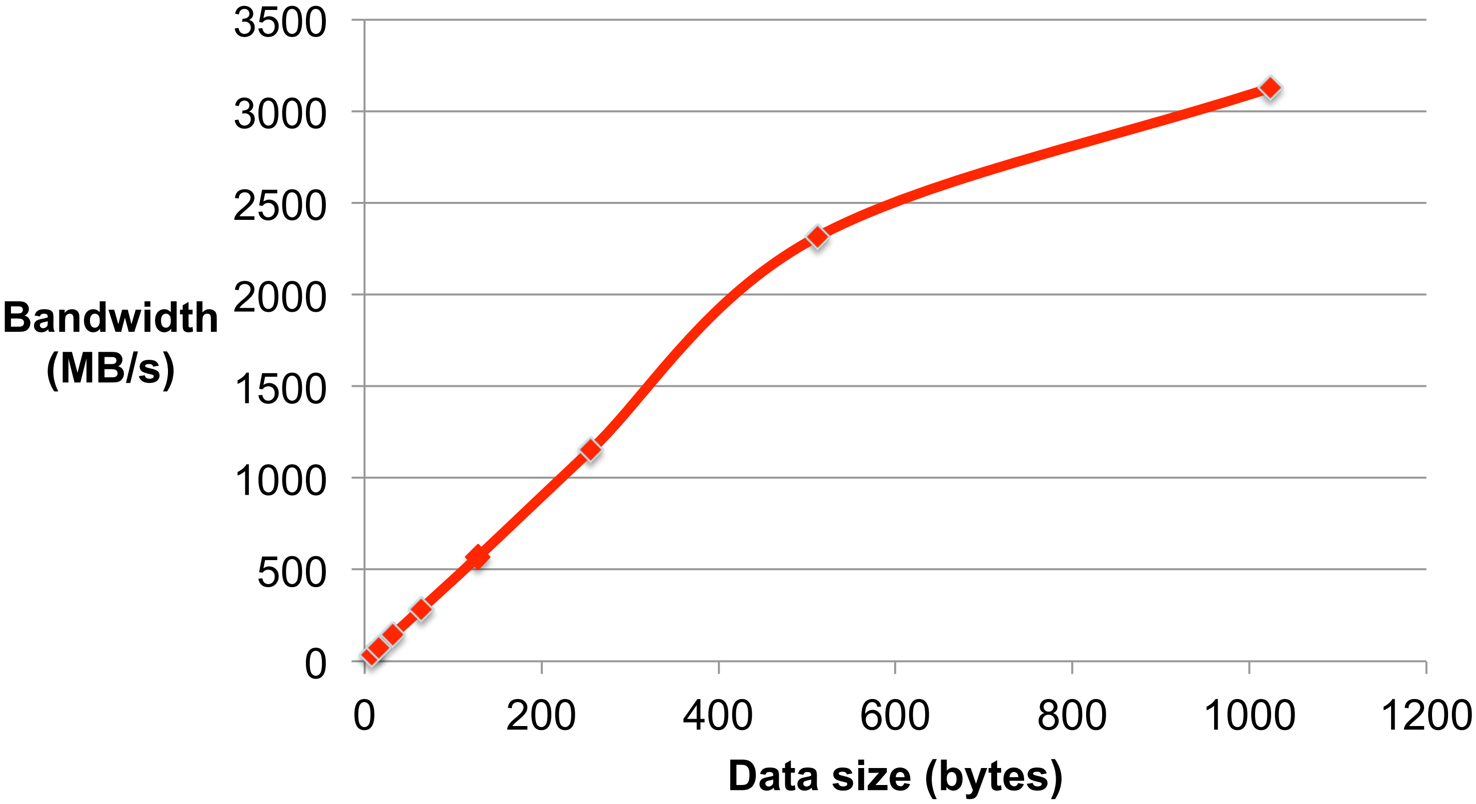
LTS Golgatha Cluster Hardware Specs

- 10 hardware nodes
- Infiniband communication layer between nodes
- 2 sockets per node
- Intel Xeon X5760 per socket
  - 2.93GHz
  - 6 cores (12 hardware threads w/ 2 way hyperthreading)
  - 24GB RAM per processor

Data Transfer Round Trip Time for Infiniband



Bandwidth measurements for Infiniband



Traditional Method – See Ramanujam1991

- Loop fission, fusion, interchange, peeling, etc.
- Software pipelining, scheduling, etc.
- Pros
  - \+ discovering parallelism
  - \+ increasing the granularity of parallelism
  - \+ improving cache performance

Traditional Method – See Ramanujam1991

- Cons
  - Code generation for message passing is complex and limiting
  - Needs
    - Footprint calculations which can be modeled with matrix calculations
    - Intersections of footprint with data distributions → result in irregular shaped which ***cannot*** be modeled with matrix transformations
    - Splitting footprints into portions per locale also complex and can't be modeled with matrix transformations
  - Real compilers limit aggregation to the simplest of stencil codes

Polyhedral Method – See Benabderrahmane2010

- Boundaries traced for each array use of a loop and intersected with the data distribution
- Applied to block distributions
- Pros

  + Has mathematical framework to express parallelism and find sequences of transformations in one step

  + Good at automatic parallelization and improves parallelism, granularity of parallelism, and cache locality
- Cons
  - Core polyhedral method does not compute information for message passing code generation
  - Uses ad hoc add-ons for message passing

PGAS Methods – See Chen2005

- Redundancy elimination, split-phase communication, communication coalescing
- Pros

  + eliminates the need for cross thread analysis

  + targets fine-grained communication in UPC compiler
- Cons
  - No locality analysis that statically determines whether an access is shared or remote

What about Block?

- Our method does not help the Block distribution
  - Reason: Needs cyclic pattern
- For Block, we use the traditional method