# Affine Loop Optimization using Modulo Unrolling in CHAPEL

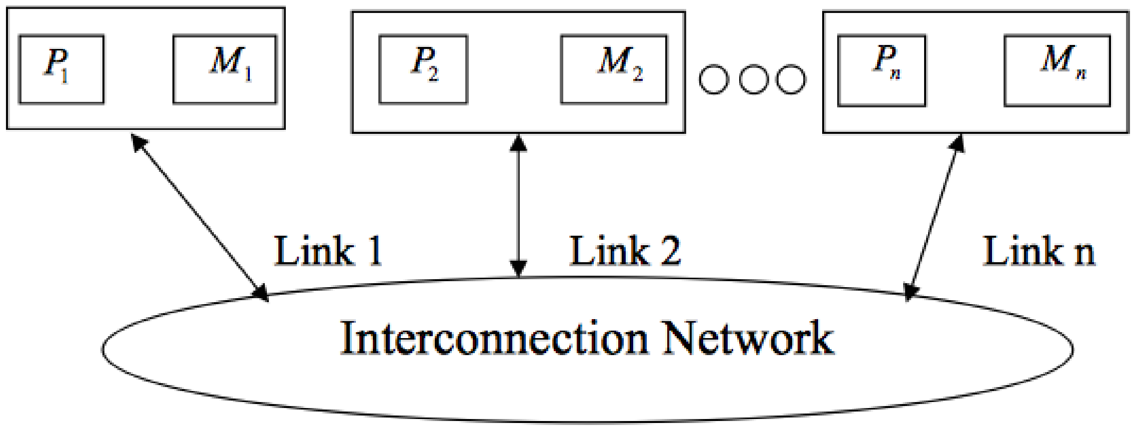
Aroon Sharma, Darren Smith, Joshua Koehler, Rajeev Barua, Michael Ferguson

# Overall Goal

- Improve the runtime of certain types of parallel computers
  - In particular, message passing computers
- Approach
  - Start with an explicitly parallel program
  - Use modulo unrolling to minimize communication cost between nodes of the parallel computer
- Advantage: Faster scientific and data processing computation
- How can this method be applied to other PGAS languages besides Chapel?

# 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

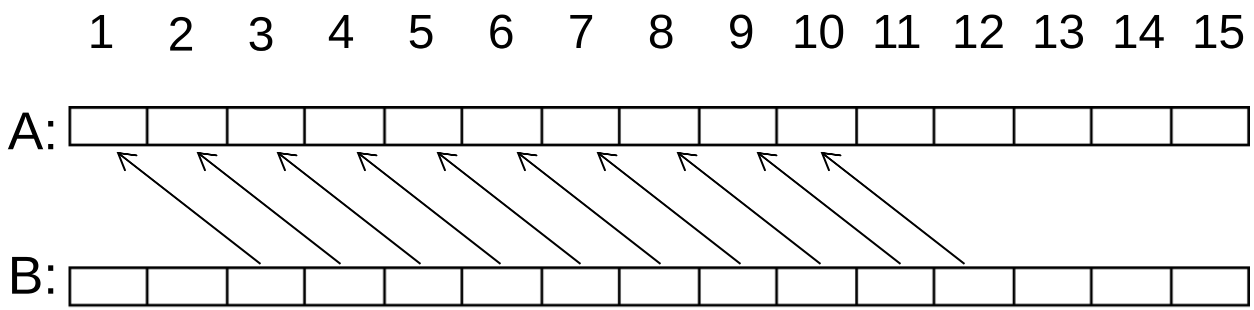
# Affine Array Accesses

- 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?

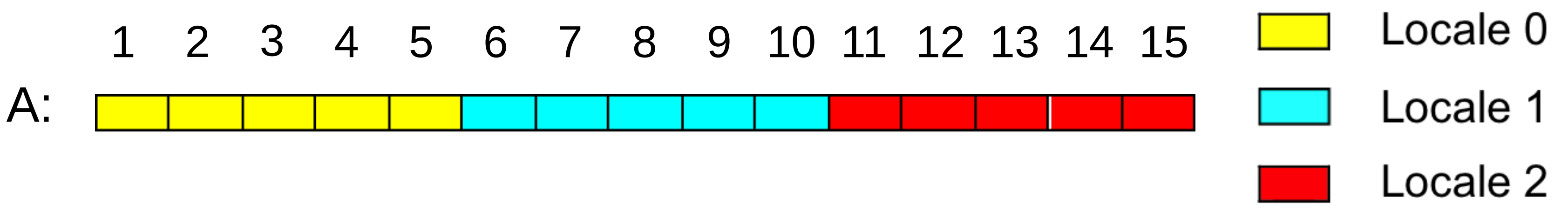


# Data Distributions in CHAPEL

- 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

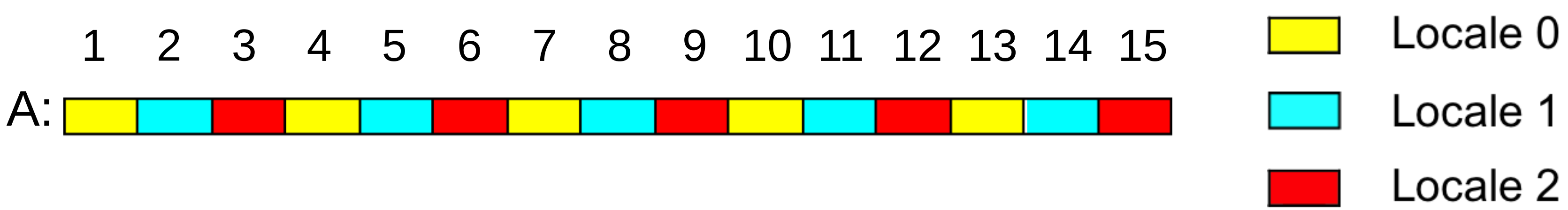
# Data Distributions in CHAPEL - Block

```
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
```



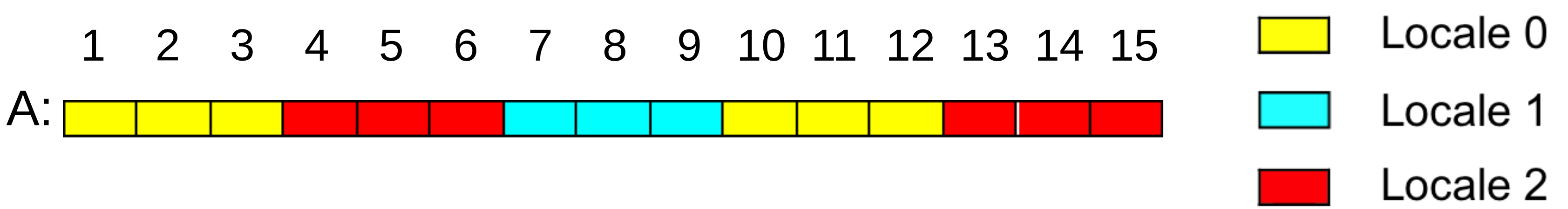
# Data Distributions in CHAPEL - Cyclic

```
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

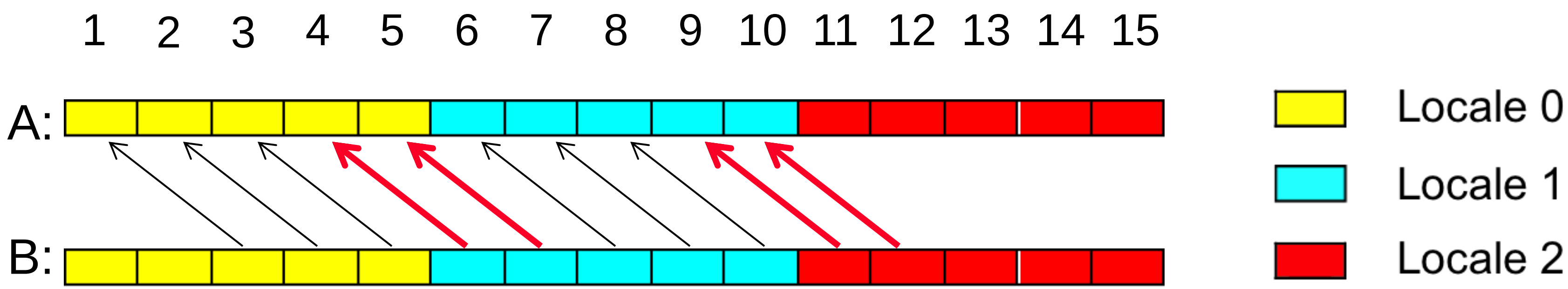
```
use BlockCycDist;

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




*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];
```

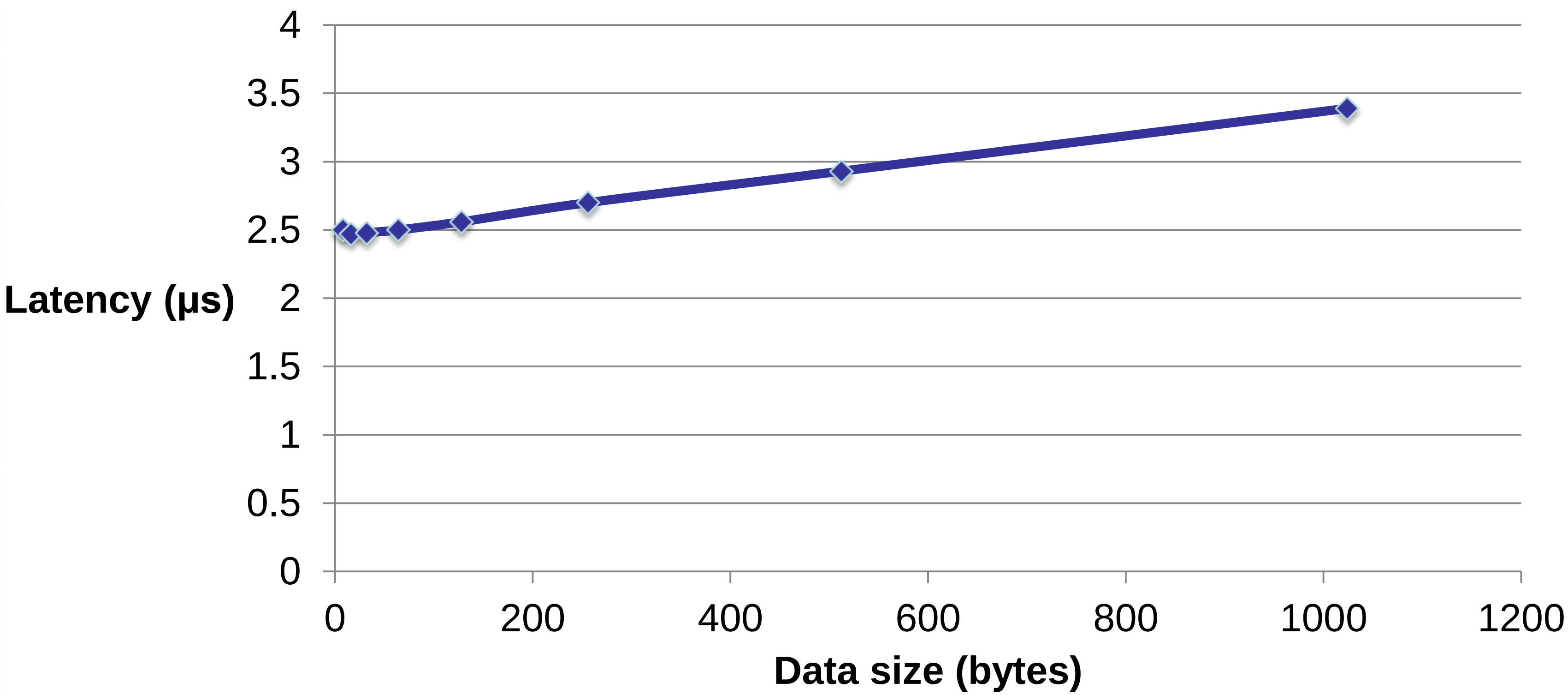



- 4 Messages
  - Locale 1 → Locale 0 containing B[6]
  - Locale 1 → Locale 0 containing B[7]
  - Locale 2 → Locale 1 containing B[11]
  - Locale 2 → Locale 1 containing B[12]

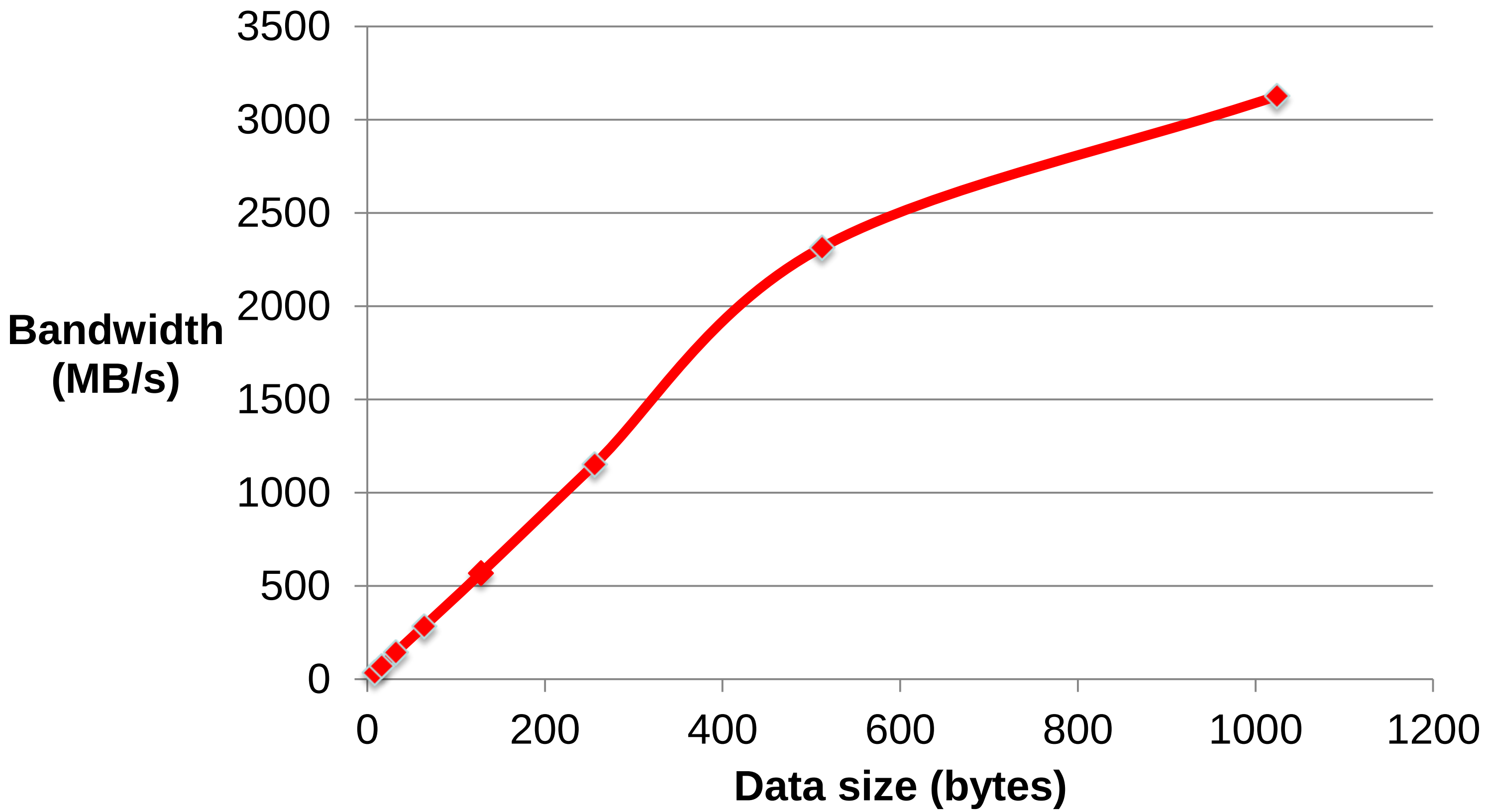
# Data Communication in CHAPEL can be Improved

- 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

# Data Transfer Round Trip Time for Infiniband



# Bandwidth measurements for Infiniband



# 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

# Outline

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

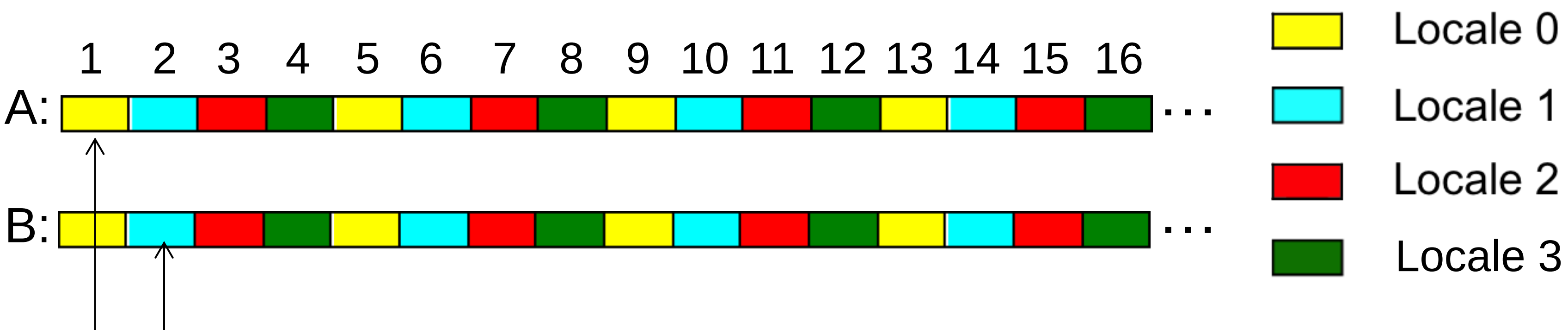
# Modulo Unrolling – See Barua1999

- 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
- Intended to improve memory parallelism for tiled architectures in sequential loops
- Applicable for **Cyclic** and **Block Cyclic**

# Modulo Unrolling Example

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

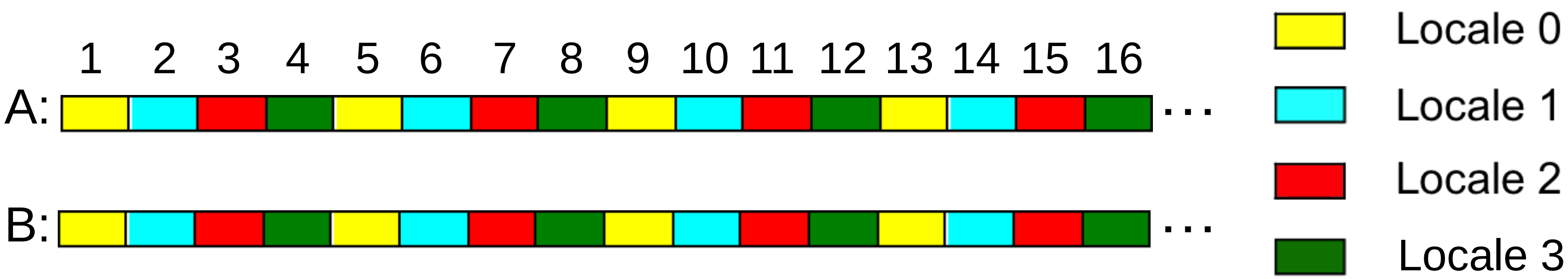
Each iteration of the loop accesses data on a different locale



# Modulo Unrolling Example

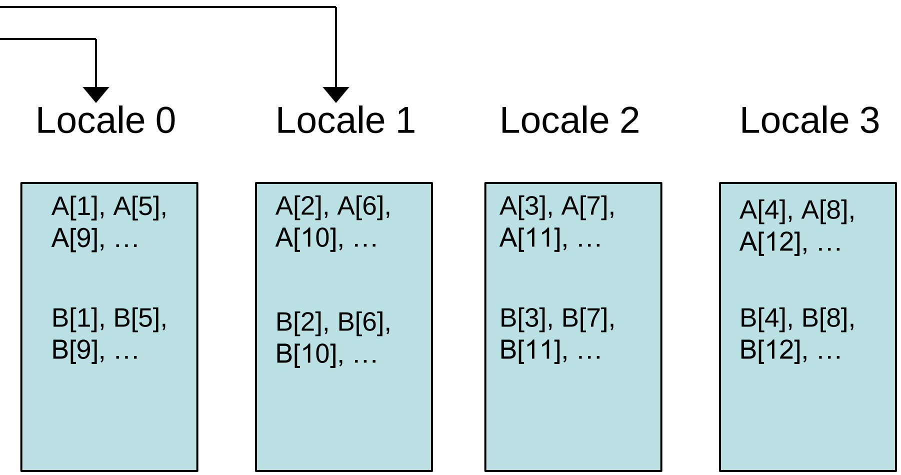
```
for i in 1..99 by 4 {
   A[i] = A[i] + B[i+1];
   A[i+1] = A[i+1] + B[i+2];
   A[i+2] = A[i+2] + B[i+3];
   A[i+3] = A[i+3] + B[i+4];
}
```

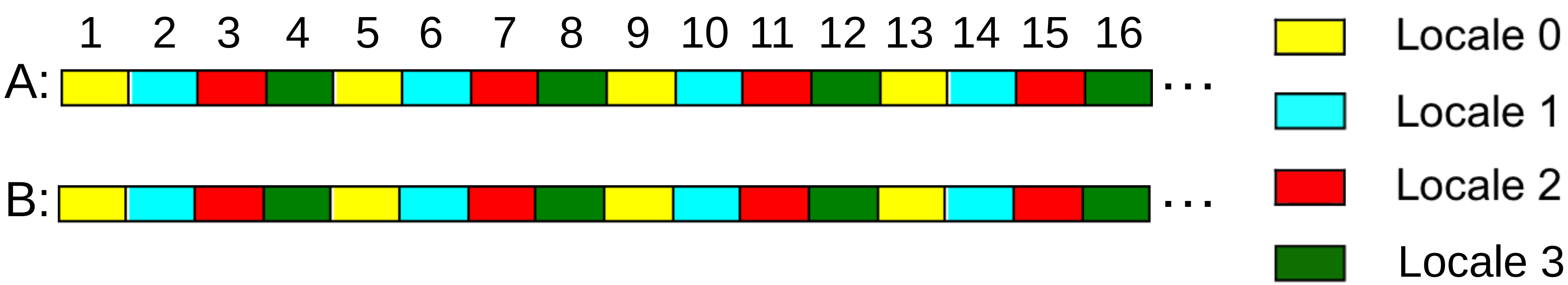
Loop unrolled by a factor of 4 automatically by the compiler



# 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];
}
```




How do we apply this concept in Chapel?

# Outline

- Introduction and Motivation
- Previous Work
- Modulo Unrolling
- **Optimized Cyclic and Block Cyclic Dists**
- Results
- What about Block?

# 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

# 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];
}
```

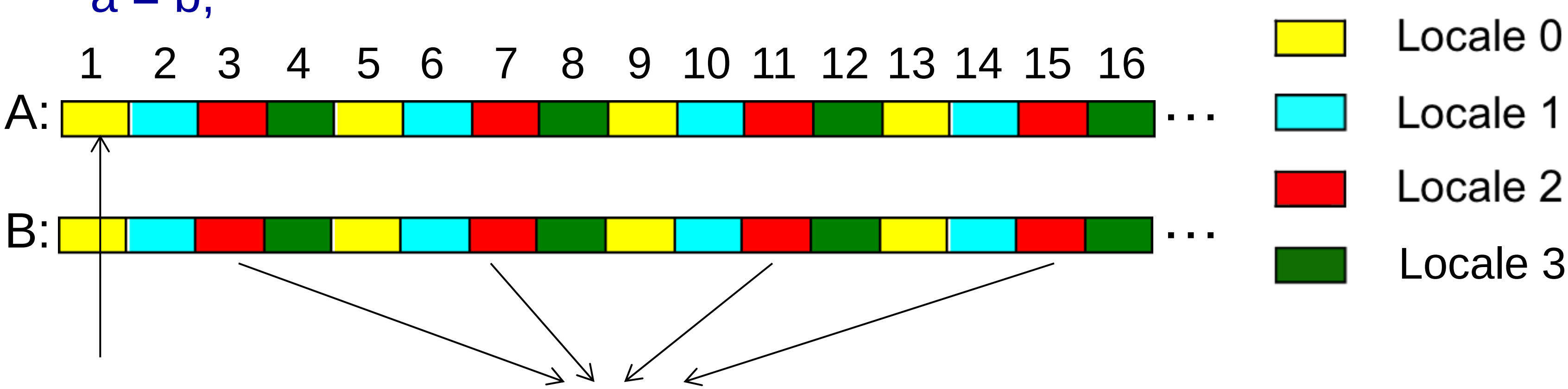
→ Zippered iteration

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

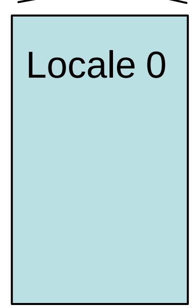
Implement modulo unrolling and message aggregation within the leader and follower iterators of the Block Cyclic and Cyclic distributions!

# Modulo Unrolling in CHAPEL Cyclic Distribution

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



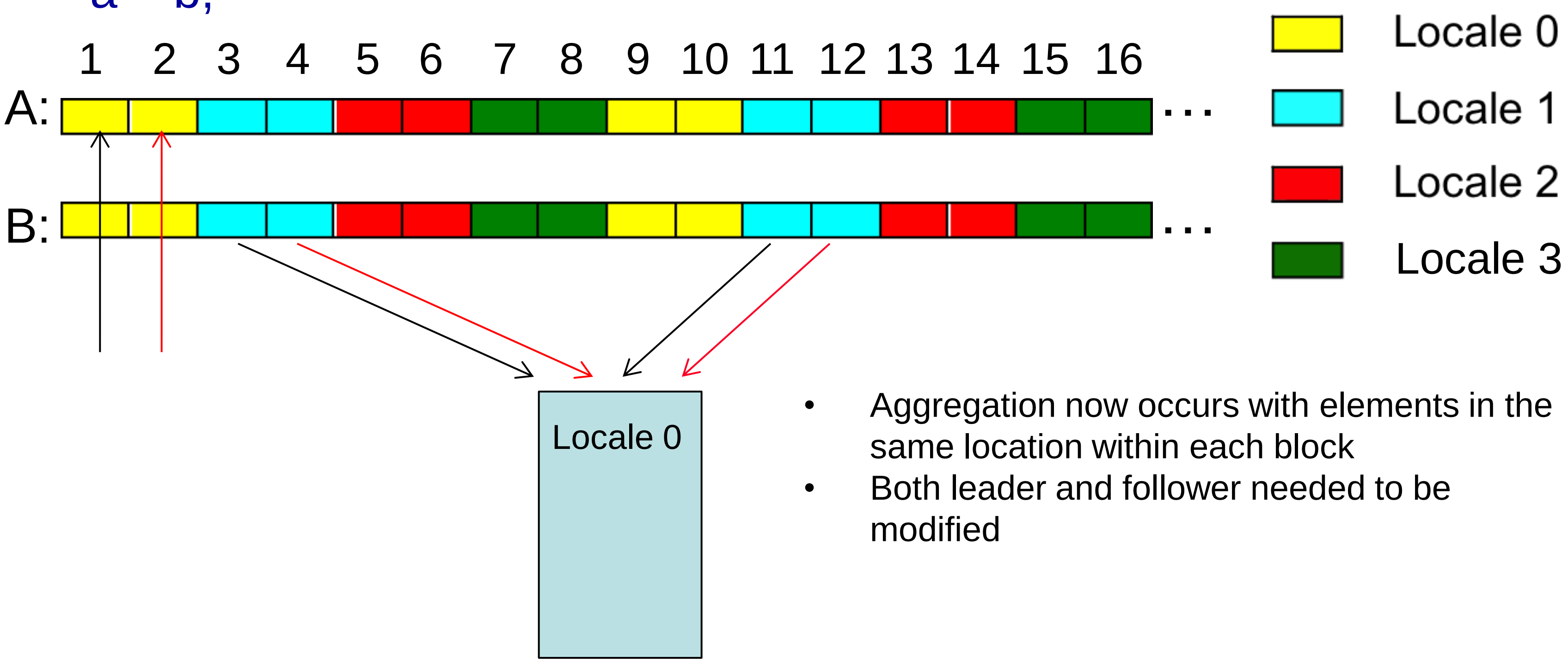

*if yielded elements are written to during the loop, a similar bulk put message is required to update remote portions of array




- 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

# Cyclic Follower Implementation

```
iter CyclicArr.these(param tag: iterKind, followThis, param fast: bool = false) var!
     where tag == iterKind.follower {!
!
//check that all elements in chunk are from the same locale!
for i in 1..rank {!
     if (followThis(i).stride * dom.whole.dim(i).stride % !
         dom.dist.targetLocDom.dim(i).size != 0) {!
          //call original follower iterator helper for nonlocal elements!
}    }!
if arrSection.locale.id == here.id then local {!
     //original fast follower iterator helper for local elements!
} else {!
 !    //allocate local buffer to hold remote elements, compute source and destination   !     !
     //strides, number of elements to communicate!
 !    !chpl_comm_gets(buf, deststr, arrSection.myElems._value.theData, srcstr, count);!
 !    !var changed = false;!
 !    !for i in buf {!
 !    !     !var old_i = i;!
 !    !      yield i;!
 !    !     !var new_val = i;!
 !    !     !if(old_val != new_val) then changed = true;!
 !    !}!
 !    !if changed then !
 !          chpl_comm_puts(arrSection.myElems._value.theData, srcstr, buf, deststr, count);!
}    }!
```

# Outline

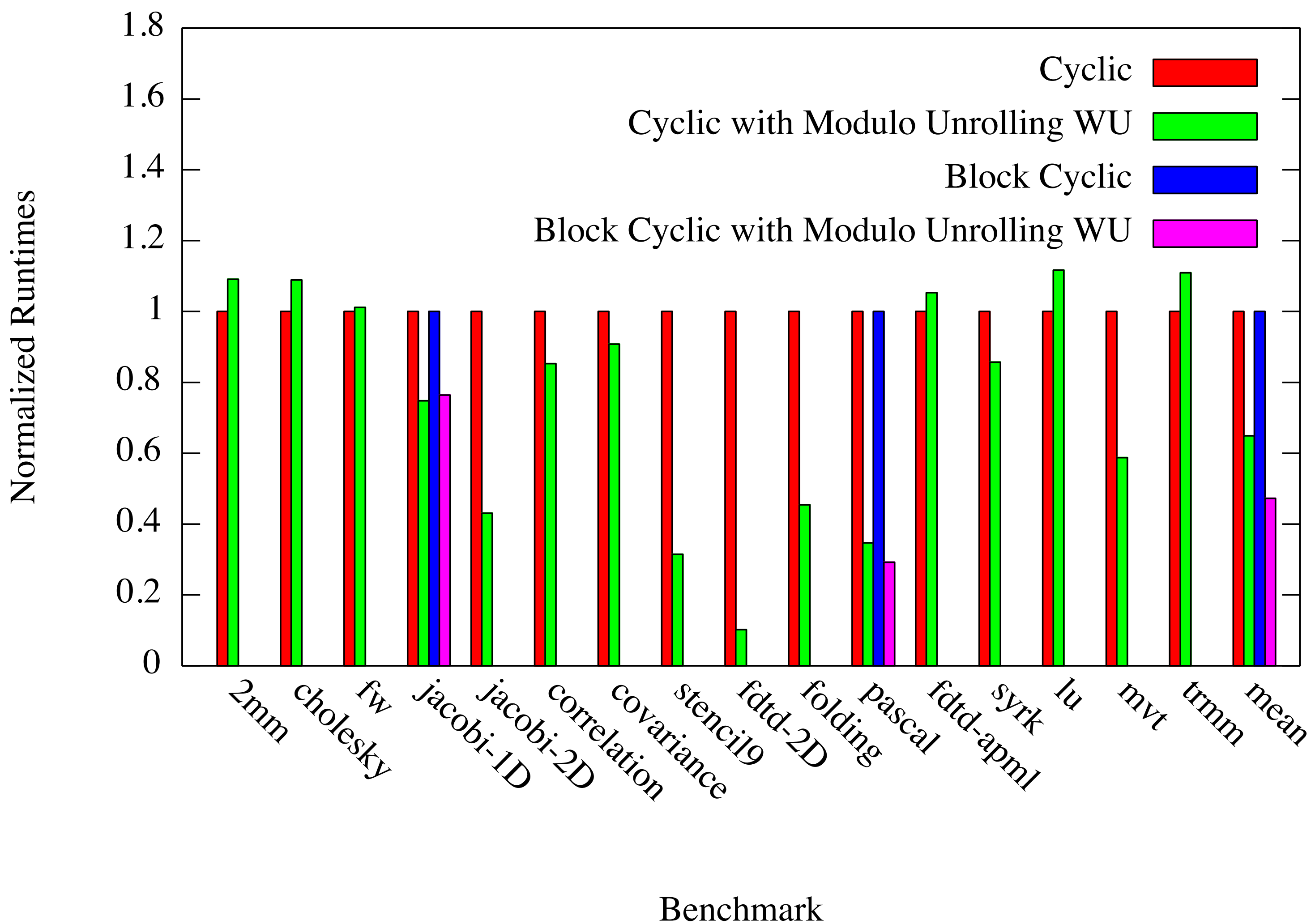
- Introduction and Motivation
- Previous Work
- Modulo Unrolling
- Optimized Cyclic and Block Cyclic Dists
- **Results**

# Benchmarks

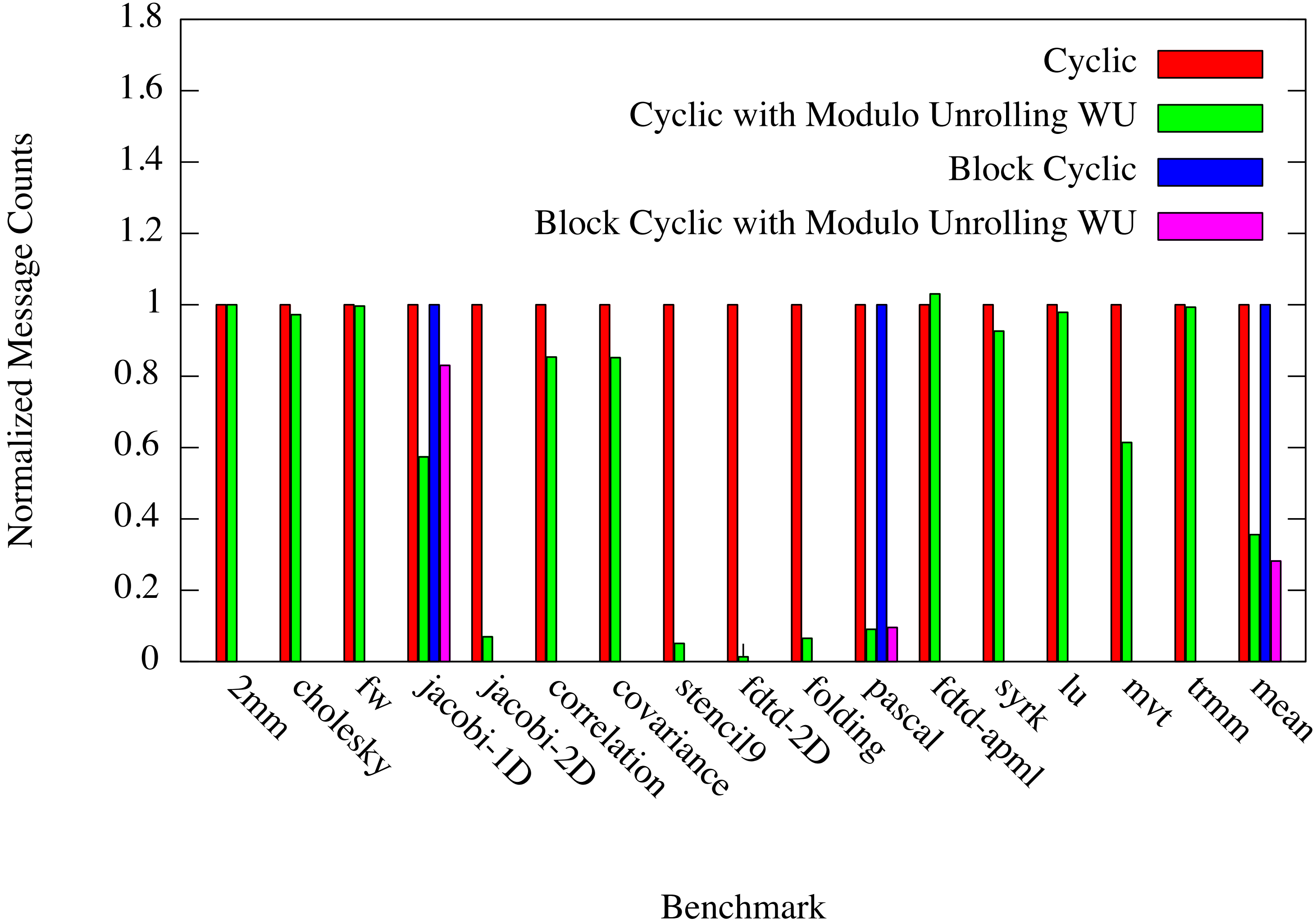
| Name | Lines of Code | Input Size | Description | Elements per follower iterator chunk |
|---|---|---|---|---|
| 2mm | 221 | 128 x 128 | 2 matrix multiplications (D=A*B; E=C*D) | 4 |
| fw | 153 | 64 x 64 | Floyd-Warshall all-pairs shortest path algorithm | 2 |
| trmm | 133 | 128 x 128 | Triangular matrix multiply | 8 |
| correlation | 235 | 512 x 512 | Correlation computation | 16 |
| covariance | 201 | 512 x 512 | Covariance computation | 16 |
| cholesky | 182 | 256 x 256 | Cholesky decomposition | 16 |
| lu | 143 | 128 x 128 | LU decomposition | 8 |
| mvt | 185 | 4000 | Matrix vector product and transpose | 250 |
| syrk | 154 | 128 x 128 | Symmetric rank-k operations | 8 |
| fdtd-2d | 201 | 1000 x 1000 | 2D Finite Different Time Domain Kernel | 16000 |
| fdtd-apml | 333 | 64 x 64 x 64 | FDTD using Anisotropic Perfectly Matched Layer | 4 |
| jacobi1D | 138 | 10000 | 1D Jacobi stencil computation | 157 |
| jacobi2D | 152 | 400 x 400 | 2D Jacobi stencil computation | 2600 |
| stencil9† | 142 | 400 x 400 | 9-point stencil computation | 2613 |
| pascal‡ | 126 | 100000, 100003 | Computation of pascal triangle rows | 1563 |
| folding‡ | 139 | 50400 | Strided sum of consecutive array elements | 394 |

* Data collected on 10 node Golgatha cluster at LTS

# Runtime Comparisons



# Message Count Comparisons



# Overall Improvement of Modulo Unrolling

- On average Cyclic with modulo unrolling results in
  - 36% reduction in runtime
  - 64% fewer messages
- On average Block Cyclic with modulo unrolling results in
  - 53% reduction in runtime
  - 72% fewer messages

# 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

# Future Work

- Scalability Testing
  - Strong (Varying number of locales)
  - Weak (Varying the input sizes)
  - Block Size

- Add dynamic checks to determine when to turn on/off modulo unrolling to achieve better overall speedups

- Experiment with non-blocking communication schemes to overlap communication and computation

# Questions?

# Backup Slides

# 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.

# Pseudocode of Compiler Transformation

(a)

```
forall i in s..e by n {!
    //affine array expressions!
    A1[a1*i+b1] = A2[a2*i+b2] + 3;!
}!
```

(b)

```
for k in 0..((lcm(B,n)/n)-1) {!
    forall i in (s+k*n)..e by lcm(B,n) {!
        //affine array expressions!
        A1[a1*i+b1] = A2[a2*i+b2] + 3;!
}    }!
!
!
```

(c)

```
for k in 0..((lcm(B,n)/n)-1) {!
   for j in 0..N-1 {!
      if(f(s+k*n+lcm(B,n)*j)/B mod N == $) {!
         //fetch elements from affine array expressions!
!        //that are not owning expressions of the loop!
         var buf1 = GET(A2[(s+k*n+lcm(B,n)*j)+b2..e+b2 by N*lcm(B,n)*a2]);!
         var h = 0;!
         forall i in (s+k*n+lcm(B,n)*j)..e by lcm(B,n)*N {!
            //affine array expressions!
            A1[a1*i+b1] = buf1[h] + 3;!
            h++;!
         }!
         //write buffer elements back if written to during loop!
!        if(buf1_is_modified)!
!           SET(A2[(s+k*n+lcm(B,n)*j)+b2..e+b2 by N*lcm(B,n)*a2]) = buf1;!
} } }!
!
```

(d)

s = starting loop bound
e = ending loop bound
n = loop stride
B = block size
N = number of locales
$ = current locale identifier

# 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.

# What about Block?

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

# 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

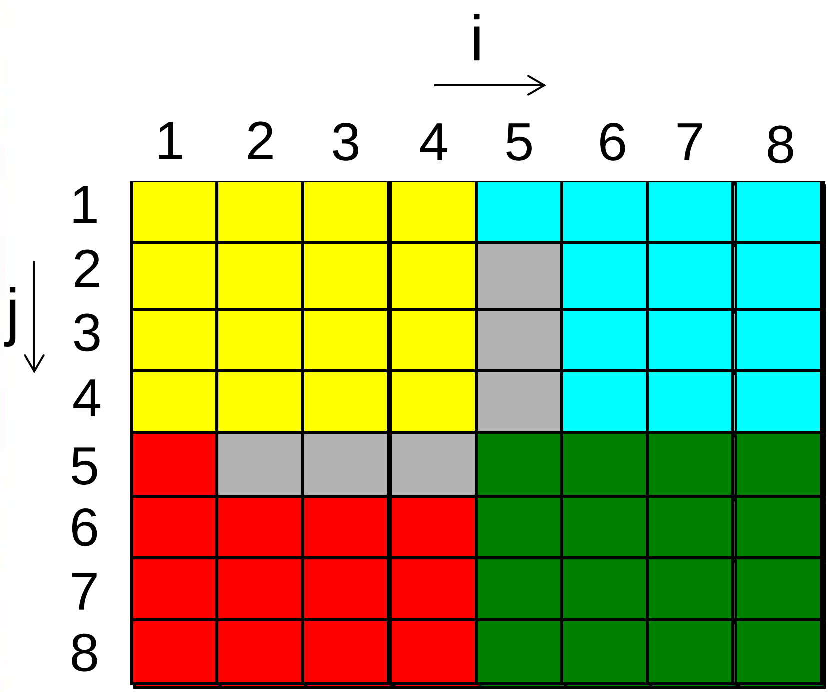
Do the computation using local buffers

# 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?

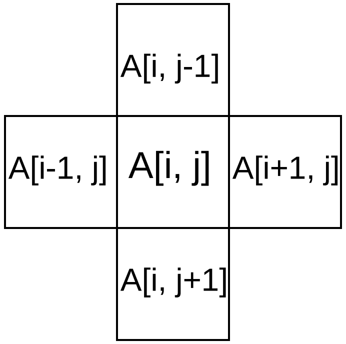
# What about Block?

## 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


forall (i,j) in {2..7, 2..7} {
  $A_{new}$[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