Affine Loop Optimization for Distributed Memory Systems

Darren Smith
Department of Computer Science
University of Maryland,
College Park, MD 20742
darrenks628@gmail.com
April 2013, Updated September 2013

Abstract
Compilation of shared memory programs for distributed memory architectures is a vital task with large potential for speedups over existing techniques. The simplicity of the shared memory model increases programmer productivity, however due to the cache-coherence problem this model does not scale well to large numbers of processors. Naive emulation of shared memory on distributed memory architectures is feasible, but introduces large inefficiencies. In this paper a new method is introduced, which solves this problem for a common type of parallel loop. Improving the result by aggregating messages and eliminating dynamic locality checks for affine array indices.

1. Introduction

This paper contributes to solving the problem of compilation for distributed memory, by introducing a new optimization for parallel for loops with affine array indices. Parallel for loops are useful in parallel programs because of their conceptual simplicity and are quite common especially in scientific computing, a predominate application of large scale parallel computing. Affine array indices (arrays accessed by any linear functions of a loop's induction variable) are also quite common in scientific computing, and the most general type of array access that can be reasonably statically analyzed. These types of loops are the bottleneck for a great deal of numeric calculations, optimizing them would give great speedup.

Previous approaches to this problem attempt to analyze the memory footprint of arrays using polyhedra [2, 5, 8, 9, 10, 12, 18]. Boundaries are traced for each array use and these are intersected with the data distribution (usually block). The loop is then duplicated for each resulting polyhedron, inside each polyhedra each array use has a uniform memory bank, which is statically analyzable, allowing for good message aggregation and elimination of dynamic locality checks. The polyhedral method is a powerful framework for shared memory architectures [12].

Footprint methods require careful tracking of polyhedra, which becomes very complex on non-trivial programs. The complexity is further compounded when these polyhedra are in intersected with data distributions for distributed memory. Due to this complexity, existing methods tend to restrict inputs to simple index functions. The polyhedral method navigates these issues for shared memory, but provides no guidance for intersecting polyhedra with data distributions, so all of these challenges remain when compiling for distributed memory.

The proposed approach utilizes a cyclic distribution. The memory bank used by arrays with affine array indices repeats every N iterations of a loop (where N is the number of memory banks the array is cyclically distributed over). Therefore if a loop is unrolled by a factor of N, affine array indices can be statically analyzed since they are linear functions of the loop induction variable. In practice, a cyclic distribution causes excessive memory communication since adjacent memory locations are no longer on the same bank. A block cyclic distribution can be used to mitigate this cost, using a block cyclic distribution requires an additional transformation for the proposed approach to be general enough to handle it correctly.

The proposed approach overcomes these drawbacks because it does no footprint analysis. Multiple dimensions, multiple statements, and complex array indices cause no increase in complexity of the analysis so long as all array indices are still affine functions of the loop induction variables. Only low level unrolling and static evaluation of affine functions are needed for analysis. Since no footprint analysis is performed the issue of intersecting polyhedra with the data distribution disappears.

Automatic compilation using the proposed method is a work in progress. It is planned to be implemented in Chapel as an LLVM pass. However some results are given by performing the method by hand on C code for the UPC compiler, yielding significant speedup and reduced communication.

2. Background

In a typical scheme for compilation to distributed memory, a parallel loop (referred to here as a forall loop) is compiled to be executed in parallel by assigning some iterations to each processor in a block fashion. For any memory access, an "if" check is performed since this memory could be local or on another node, requiring a message to get the memory. It may perform a transformation like this, from:

```
forall i=0:99
      a[i]=a[i+2]-3
```

Transformation to execute the code in parallel with a block distribution of size 100/N:

```
for i'=0:ceil(100/N)
      i=i'+$*ceil(100/N)
      break if i>99
      a[i]=a[i+2]-3
```

Transformation to use distributed memory:

```
for i'=0:ceil(100/N)
      i=i'+$*ceil(100/N)
      break if i>99

      if is_local(&a[i+2])
            a_get=local_get(&a[i+2])
      else
            a_get=remote_get(&a[i+2])
      a_set=a_get-3
      if is_local(&a[i])
            local_set(&a[i], a_set)
      else
            remote_set(&a[i], a_set)
```

Here $ is processor id number, the notation `a:b:c` means from a to c with step size of b. local/remote get/set perform the actual low level function described by their name.

This transformation has two major drawbacks. What once was a simple and fast array lookup now is wrapped in an "if" statement at every iteration of the loop. With branch miss prediction this could be expensive. Secondly messages that must be sent to lookup or set memory are only handling one element at a time. There is a large overhead for such a message, whereas there is less overhead per element if messages are sent that request multiple elements at once.

A common way to achieve these benefits is with footprint analysis. Find which ranges are on which processor; break the iterations into parts that handle each case. For example if there are four processors with the data of 'a' being distributed in a block fashion, the code could be transformed to:

```
for i'=0:22
      i=i'+$*25
      local_set(&a[i], local_get(&a[i+2])-3

for i'=23:24
      i=i'+$*25
      local_set(&a[i], remote_get(&a[i+2])-3
```

The second loop could further be transformed to aggregate both remote_get calls:

```
a_buf = remote_get(&a[25:26])
for i'=23:24
      i=i'+$*25
      local_set(&a[i], a_buf[i'-23])-3
```

This is very near optimal code for this example. But this approach completely falls apart for more complicated examples. Notice how the loop was duplicated. If there were more statements in the loop, that required different points to be broken, even more duplication would occur (in an exponential fashion). If the array was multidimensional or its index function was more complicated than just a linear function of i, complexity could become unmanageable.

2.1 Modulo Unrolling

Modulo Unrolling [1] is a technique for statically disambiguating memory banks utilizing a cyclic distribution. The proposed method generalizes modulo unrolling and introduces an additional optimization to improve communication costs (which was not necessary in the original architecture which it was developed for due to fast communication). The insight behind Modulo Unrolling is to use a cyclic distribution with a size of N where N is the number of processors. Once a loop is unrolled by a factor of N each affine array access will reside on the same memory bank.

Suppose N is 4, the previous example would become:

```
for i=0:4:99
      a[i+0]=a[i+2+0]-3
      a[i+1]=a[i+2+1]-3
      a[i+2]=a[i+2+2]-3
      a[i+3]=a[i+2+3]-3
```

Since the memory uses a cyclic distribution also size 4, the first array assignment will always be to bank 1, first array access will be bank 3, second array assignment will be bank 2, second array access will be bank 4, bank number is 1+(index mod 4).

In modulo unrolling each unrolled statement would be on a different processor, but because we are targeting SPMD model and not MDMD, there is no need to actually unroll the loop, we can use the processor number instead:

```
for i=0:4:99
      a[i+$]=a[i+2+$]-3
```

Since we know that 0=i*4 mod 4 for any integer i, and 2=i*4+2 mod 4. We can statically disambiguate the memory banks. We know the array access will always be non-local and the array assignment will always be local. Rather than doing a remote get on every iteration, the messages can be aggregated to all occur at once before the loop:

```
_buf = a[$+2:4:99+2] #remote get of all needed elements
h=0
for i=0:4:99
      a[i+$] = _buf[h]-3 #local get and set
      h++
```

3. The Optimization

The basic idea behind the proposed optimization is to perform Modulo Unrolling, but with a block-cyclic distribution (instead of a cyclic distribution) to reduce the amount of communication, and message aggregation to reduce time spent doing any remaining communication.

A block-cyclic distribution is not compatible with Modulo Unrolling, however, if forall loops are transformed so that step sizes are a multiple of the block size, then effectively the loop would “act” like a loop over a cyclically allocated array. Each successive iteration would access the next memory bank, just like a step over a cyclic array, therefore Modulo Unrolling could be used on such a loop. Note that such a transformation is possible, and that it is legal since in a forall loop the order iteration evaluations is unspecified. There is one additional cravat: Modulo Unrolling requires array indices to be affine functions of the loop induction variable, but after the loop transformation, array indices that were affine now have terms that are not a function of the loop induction variable. However these extra terms are constant throughout the loop (but not the program). If the calculations like which bank to buffer from that were performed statically for Modulo Unrolling are now performed dynamically, but before the loop starts, the benefit of Modulo Unrolling can still be had, with very little extra overhead, since any dynamic logic is moved from inside the loop to outside. This generalization to Modulo Unrolling could be useful in other cases besides just the ones created by the initial transformation.

The idea behind message aggregation is if we can statically determine that an array access inside a loop is going to be remote, rather than request it every iteration of the loop, creating an excessive number of messages, and suffering the latency at every iteration, just make all the memory requests before the loop and save them into a local buffer. This transformation is completely straightforward for affine array indices if you can pre-determine which memory bank the access will be from (which Modulo Unrolling gives you).

3.1 Conventions and notes for the transformations

n = loop step size
s = starting loop value
e = ending loop value
B = block size
N = number of processors and memory banks
$ = processor id
# = comment

All non-inner forall loops converted to regular for loops (we will only optimize the innermost loops).

Array accesses are targeted, in which all indices are affine functions of the loop variable plus any expression that is constant throughout the loop.

Arrays of any dimension will work, however arrays shown in these examples are only one-dimensional.

3.2 Transformation to enable optimization for block-cyclic distribution

Transform forall loops to have a step size that is a multiple of B. Now, when the new forall loop is stepped through with step N*B, then affine expressions will be constant mod N*B, resulting in the same bank being used.

Convert forall loops where n is not a multiple of B from:

```
forall i=s:n:e
      {code}
```

to:

```
for k=0:lcm(B,n)/n-1
      forall i=s+k*lcm(B,n):lcm(B,n):e
            {code}
```

3.3 Assigning iterations to processors

In order to get the best performance, iterations of the forall loop should be executed by the processor that has most of the array accesses already local. This reduces the amount of communication needed.

There may be many array references in the forall loop, for best performance, we must attempt to calculate which bank has the most references in common. To do this, we count all unique affine array accesses. The most common affine expression in this count is called the **owning expression.** If the owning expression is a function, say f, and f(i) = z, then processor number z is the owner of loop iteration i.

When counting unique affine array accesses, expressions are unique if they are used by different arrays or are used in both a set and a get to an array. Basically anytime they would cause additional memory to be passed. For example if the code was `a[i]+b[i]`, both expressions are "`i`", but it is used in different arrays, so both are unique. If the code was `a[i]+2*a[i]` the memory used is the same, so only one is unique. If the code was `a[i]=a[i]+1`, both would be considered unique since one is used in a get and the other in a set (the reason for this importance will be clear later, a get requires copying before the loop, and a set requires copying after the loop, so they each cause message passing).

Note that these expressions need not solely be functions of just the loop induction variable, they can also be functions of variables that are constant within the loop. In this case it may be more difficult for the compiler to evaluate if expressions are equivalent (for example is j+i+1 the same as j+1+i ? yes).

Putting this all together, let us find the owning expression for this code:

```
forall i=0:B:50
      a[i]=b[i*2]+a[i-1]+a[i]+a[i+j]+a[i*i]
      b[i-1]=a[i-1]*a[i]
```

The counts for each unique affine array expressions are:

```
i = set a[i], a[i] = 2
i-1 = set b[i-1], a[i-1] = 2
i*2 = b[i*2] = 1
i+j = a[i+j] = 1
```

In this example, there is a tie for the most common expression so either could be chosen as the owning expression. Note that a[i*i] was not considered since it is not affine.

3.4 Final Transformation

Now that the groundwork has been created, the final transformation is not very complex. The forall loop is converted to have a step size that is a multiple of N*B, making any expression that is an affine function of the loop induction variable constant mod N*B. Therefore arrays with affine indices will always access the same memory bank. Now that the loop skips iterations, an outer loop is needed to iterate through all of the original loop iterations.

The if statement is needed so that only the processor responsible for executing the iteration executes it. When each processor is responsible for the same number of iterations a further minor optimization is possible removing the if statement and changing the outer loop step size. This is the common case, but it is not always so, for example, when an array index is 2*i and there are 4 processors. Half of the array will be allocated on processor 1 and half on processor 3, while none on 2 and 4. If these were the only array accesses then only processors 1 and 3 would be doing the work since that is where the data would reside.

```
forall i=s:n:e #where n is a multiple of B
      {code}
```

to (note `f` is the owning expression for `code`):

```
for j=0:N-1
      if $ == mod(f(s+j)/B, N)
            {prebuf}
            h=0
            for i=s+j*n:n*N:e
                  {code*}
                  h++
            {postbuf}
```

Here `prebuf` is copying to local buffers for the entire strided access of any accessed arrays that could be non-local. Arrays are considered possibly local if it cannot be statically determined that evaluating the array index from 0 to N-1 is equivalent to the owning expression.

`postbuf` is copying from the local buffers to any non-local arrays that were written to.

In the main loop `code*`, known local array accesses now have their dynamic if check removed, just load the local memory. Replace all non-local array usages with their buffered equivalent. For example, if it is known that `a[i*2+3]` will reside non locally, the expression `a[i*2+3]` will be converted to just `_buf[h]` (where `_buf` is the buffer that message aggregation copied all elements to be looked up into during the `prebuf`).

3.5 Example #1

```
#N=4, B=2

#original code
forall i=0:99
      a[i]=a[i+2]-3

#after transformation for handling block cyclic
for k=0:1
      forall i=k:2:99
            a[i]=a[i+2]-3

#counting affine array accesses
# [i] = 1, [i+2] = 1, tie, choosing as owning expression: f(i)=i

#after final transformation
for k=0:1
      for j=0:3
            if $ == mod((k+j*2)/2, 4)
                  _buf1 = a[(k+j*2:8:99)+2]
                  h=0
                  for i=k+j*2:8:99
                        a[i] = _buf1[h]-3
                        h++
```

3.6 Example #2

```
#N=8, B=4
```

```
#original code
forall i=3:2:999
      a[i]=b[i*2]+a[i]
      b[i-1]=a[i-1]*a[i]

#after transformation for handling block cyclic
for k=0:1
      forall i=3+k*4:4:999
            a[i]=b[i*2]+a[i-1]+a[i]
            b[i-1]=a[i-1]*a[i]

#counting affine uses
# [i]=2, [i*2]=1, [i-1]=2
# tie between [i] and [i-1], choosing as owning expression: f(i)=i-1

#after final transformation
for k=0:1
      for j=0:7
            if $ == mod( (2+k*4+j)/4, 8)
                  _buf1 = a[3+k*4+j*4:32:999]
                  _buf2 = b[(3+k*4+j*4:32:999)*2]
                  h=0
                  for i=3+k*4+j*4:32:999
                        _buf1[h] = _buf2[h]+a[i-1]+_buf1[h]
                        b[i-1] = a[i-1]*_buf1[h]
                        h++
                  a[3+k*4+j*4:32:999] = _buf1
```

4. Results

A Jacobi benchmark is used (repeated simultaneous averaging of neighbors in a 2d array). The algorithm was implemented in UPC and Modulo Unrolling with message aggregation is performed by hand for comparison. The UPC code is compiled with network=UDP to force the program to do message passing instead of using the shared memory actually present on the systems tested.

On UMIACS cluster (8 cpu) Array size=100x100, iterations=200 (results are in seconds)

Block distribution	0.12
Cyclic distribution	0.39
Modulo Unrolling (cyclic)	0.07

Array size=500x500, iterations=20

Block distribution	0.28
Cyclic distribution	1.12
Modulo Unrolling (cyclic)	0.13

On Mac Book Pro (4 cpu) Array size=100x100, iterations=10

Block distribution	0.14

Cyclic distribution	27.24
Modulo Unrolling (cyclic)	0.13

Array size=500x500, iterations=1

Block distribution	0.09
Cyclic distribution	74.53
Modulo Unrolling (cyclic)	0.33

Total number of messages sent, Array size=100x100, iterations=1, Threads=4

Block distribution	588
Cyclic distribution	88167
Modulo Unrolling (cyclic)	199*

Total data sent, Array size=100x100, iterations=1, Threads=4 (numbers are in KB)

Block distribution	4.59
Cyclic distribution	696.79
Modulo Unrolling (cyclic)	696.79*

*The UPC trace tool did not correctly record aggregated message calls since they use a low level function. These numbers are hand calculated. For data sent, it is known that the same total amount of information must be sent whether Modulo Unrolling is used or not.

Modulo Unrolling with message aggregation always yielded a significant speed up. However in some cases a plain block distribution without Modulo Unrolling outperformed it. This is because this benchmark uses abundant adjacent array accesses, which do not require much communication for block distributions, however they do require large amounts of communication for cyclic distributions. This is precisely the motivation for transformations and generalization of Modulo Unrolling, to enable it to use block-cyclic distribution to get the benefit of a block distribution, reduced communication, while maintaining the speedup of Modulo Unrolling. At this time the transformation of the Jacobi benchmark to block-cyclic distribution with Modulo Unrolling has not been handwritten, so this benchmark is not yet shown.

5. Conclusion

The optimization presented is general enough to optimize common types of loops in parallel programs. The method avoids complexities of traditional analysis involving footprint analysis, by using simple unrolling and a cyclic distribution. A cyclic distribution will usually result in increased communication, but by adding an additional transformation the original optimization can work for block cyclic distributions too, mitigating the increase in communication. The final result of the optimization is code that can statically determine locality of memory for all affine array indices, maintain minimal total communication and aggregate messages to reduce message cost for communication that is required. The result is highly efficient code, which could be produced automatically with relatively simple algorithms.

6. Future Work

The optimization has yet to be implemented in a compiler. Future plans are to add support the Chapel programming language. Chapel supports LLVM byte code output so an LLVM pass can be written to perform the actual optimization. Further optimizations may be possible such as strip mining: rather than aggregate all messages into one per buffer, the loop could be strip mined, allowing for better overlaying of communication and computation.

7. Bibliography

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. Callahan, D., & Kennedy, K. (1988). Compiling programs for distributed-memory multiprocessors. *The Journal of Supercomputing , 2* (2), 151-169.
3. Chamberlain, B. L., Choi, S.-e., Lewis, E. C., Lin, C., Snyder, L., & Weathersby, W. D. (1998). ZPL's WYSIWYG performance model. *In Third International Workshop on High-Level Parallel Programming Models and Supportive Environments.*
4. Chamberlain, B., Choi, S.-e., Lewis, E. C., Lin, C., Snyder, L., & Weathersby, W. D. (1996). Factor-Join: A Unique Approach to Compiling Array Languages for Parallel Machines. *In Workshop on Languages and Compilers for Parallel Computing .*
5. Chavarria, D., & Mellor-Crummey, J. (2005). Effective communication coalescing for data-parallel applications. *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming* , 14-25.
6. Davidson, J. W., & Jinturkar, S. (1995). Improving instruction-level parallelism by loop unrolling and dynamic memory disambiguation. *Proceedings of the 28th annual international symposium on Microarchitecture* , 125-132.
7. Dion, M., Randriamaro, C., R, C., & Robert, Y. (1996). Compiling Affine Nested Loops: How to Optimize the Residual Communications After the Alignment Phase? *Journal of Parallel and Distributed Computing , 38*, 176-187.
8. Germain, C., & Delaplace, F. (1995). Automatic Vectorization of Communications for Data-Parallel Programs. *INTERNATIONAL EURO-PAR CONFERENCE ON PARALLEL PROCESSING* .
9. Gupta, M. (1992). Automatic Data Partitioning on Distributed Memory Multicomputers.

10. Gupta, S. K., Kaushik, S. D., Mufti, S., Sharma, S., Huang, C.-H., & Sadayappan, P. (1993). On Compiling Array Expressions for Efficient Execution on Distributed-Memory Machines. *International Conference on Parallel Processing , 2*, 301-305.
11. Huang, A., Slavenburg, G., & Shen, J. (1994). Speculative disambiguation: a compilation technique for dynamic memory disambiguation. *Proceedings the 21st Annual Internation Symposium on Computer Architecture* , 200-210.
12. Iancu, C., Chen, W., & Yelick, K. (2008). Performance portable optimizations for loops containing communication operations. *Proceedings of the 22nd annual international conference on Supercomputing* , 266-276.
13. Li, J., & Chen, M. (1991). The data alignment phase in compiling programs for distributed-memory machines. *Journal of Parallel and Distributed Computing* (2), 213-221.
14. Pouchet, L.-N. (2011). Loop transformations: convexity, pruning and optimization. *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* , 549-562.
15. Ramanujam, J., & Sadayappan, P. (1991). Compile-Time Techniques for Data Distribution in Distributed Memory Machines. *IEEE Transactions on Parallel and Distributed Systems , 2* (4).
16. Shih, K.-P. (1998). Efficient Address Generation for Affine Subscripts in Data-Parallel Programs . *International Conference on Parallel and Distributed Systems*, (pp. 758-765).
17. Trifunovic, K. (2010). GRAPHITE Two Years After: First Lessons Learned From Real-World Polyhedral Compilation. *GCC Research Opportunities Workshop.*
18. Wei, W.-h., Shih, K.-p., & Sheu, J.-p. (1997). Compiling Array References with Affine Functions for Data-Parallel Programs. *Journal of Infromation Science and Engineering* , 695-723.

Updates after initial draft (Chapel specific)

Chapel Implementation

Chapel supports user-defined distributions. This is a natural way to implement Modulo Unrolling. The iterators for distributions can be rewritten to implement the optimization. The advantage of this is the optimization can be implemented in a high level language. The Chapel compiler also lacks some of useful buildings blocks to implement optimizations since it relies mainly on the C compiler for optimizing its generated code.

Zippered iteration

Each unique affine array index can be iterated over in parallel using zippered iteration. For example, these codes are equivalent:

```
forall i in 1..100 do
        a[i]=b[i]+b[i+1]

forall (a1,b1,b2) in zip(a[1..100], b[1..100], b[2..101]) do
        a1=b1+b2
```

For the second loop to actually execute iterations in parallel, a leader follower iteration style is used. See http://pgas11.rice.edu/papers/ChamberlainEtAl-Chapel-Iterators-PGAS11.pdf The leader iterator of a[1.100] is called which initiates parallelism and chooses which iterations each task is responsible for handling. The follower iterator actually iterates over the assigned work.

Intuition for advantages of modulo unrolling in Chapel

Because zippered iteration hides the complexity of multiple domains (each iterator is only responsible for yielding its own elements. The book keeping to aggregate blocks of consecutively identical locales would be simple enough. This would effectively allow one to write aggregation in a way similar to the Polyhedra method, using polygons to aggregate. However complexity in iterators can be magnified when translated away from its high level abstractions down to low level C code. Yield will become function calls, which are hopefully inlined. However with zippered iteration these functions could be deep and optimizing them may be impossible for the C compiler. Therefore keeping the iterators simple and not having multiple yields could provide a benefit. This is where Modulo Unrolling's strength is. If the optimization is applicable, all the data asked of by the follower iterator will be on the same locale, reducing the complexity of aggregating messages.

Cyclic implementation

The leader iterator, iter CyclicDom.these(param tag: iterKind) where tag == iterKind.leader, already chooses the work to maximize locality of its elements being iterated over. Therefore it does not need to be modified.

The follower iterator, iter CyclicArr.these(param tag: iterKind, followThis, param fast: bool = false) var where tag == iterKind.follower, has been modified to perform that actual optimization.

```
for each dimension
      if followThis_stride * dom_stride % cycle_size != 0
            call original follower, optimization not applicable
            return
//we now know that all indices will be on the same locale

if arrSection.locale.id == here.id //if mem is local, iterate locally
      locallly execute:
            for e in arrSection.myElems(myFollowThis) do yield e
else //else mem is non local, use buffer to do bulk communication
      copy entire remote slice of the array into
      local buffer in 1 communication call

      var changed=false
      for i in buffer
            var old_val=i
            yield i
            if old_val != i then changed = true

      if changed then //copy back incase they modified it
            copy local buffer into original slice in
            1 communication call
```

Block-Cyclic implementation

The idea behind the Block-Cyclic implementation is largely the same. However the leader iterator: iter BlockCyclicDom.these(param tag: iterKind) where tag == iterKind.leader needs to be modified in addition to the follower iterator: iter BlockCyclicArr.these(param tag: iterKind, followThis) var where tag == iterKind.follower

The leader iterator chooses slices of work such that the stride is equal to the product of the block size and cycle size, that way each follower iterator is working with elements that are all in the same position of the block and the chosen blocks are all on the same locale. Therefore elements will all be on the same locale and if the other zippered members are using a BlockCyclic Distribution with the same block and cycle size they would be too. An intuitive way to understand the reason for this is to think of the block cyclic distribution as many cyclic distributions spliced together. The first elements of each block act like a cyclic distribution, the second elements do too, etc.

The implementation is incomplete, it is only implemented for the case of 1 dimension, and BlockCyclic as a whole needs to implement strided slices before this can be implemented with Modulo Unrolling too.

Benchmarks

Three benchmarks have been written, each specifically designed to test one aspect of the distributions.

Folding uses indices array indices like 2*i. This is something that modulo unrolling can handle but many other optimizations cannot since memory access will tend not to be adjacent.

Jacobi is the only multi dimensional benchmark. It uses adjacent array accesses, so Block distribution will have the least communication volume.

Pascal is one-dimensional but also uses adjacent array accesses. This is the only benchmark in which Modulo Unrolling in Block-Cyclic distribution can be used with at this time.

While benchmarking Cyclic with Modulo Unrolling and Block-Cyclic with Modulo Unrolling are both tested for correctness by doing the same calculation with another distribution and ensuring they are identical at all points in time. All distributions are tested for time and communication count (the number of messages, most of which are caused by non iterator related stuff). The Modulo Unrolling distributions can also have their communication volume (total array size sent) measured. This is the sum of the sizes of the low-level communication calls used to fill the buffer.

Results
(from Golgatha cluster at LTS)
FOLDING:
nl=10
n=50400
iterations=10

Block (B)
message count=3568376
took 22.2135 (s)

Cyclic (C)
message count=3349260
took 20.7506 (s)

Cyclic with modulo unrolling (CM)
message count=221340
took 8.40708 (s)

No distribution (NONE)
took 0.014889 (s)

JACOBI:
nl=10
n=400
epsilon=0.05

Block (B)
message count=158419
took 3.06303 (s)

Cyclic (C)
message count=3373170
took 14.2363 (s)

Cyclic with modulo unrolling (CM)
message count=238250
took 5.15606 (s)

No distribution (NONE)
took 0.049081 (s)

PASCAL:
nl=10
n1=100000
n2=100003
blocksize=16

Block (B)
message count=29527
took 0.691754 (s)

Cyclic (C)
message count=653839
took 5.11075 (s)

Cyclic with modulo unrolling (CM)
message count=59452
message volume=600003
took 1.57839 (s)

Block Cyclic (BC) (pascal only)
message count=982235

```
took 10.1771 (s)

Block Cyclic with modulo unrolling (BCM) (pascal only)
message count=94098
message volume=37499
took 2.69846 (s)

No distribution (NONE)
took 0.009233 (s)
```

Pascal follow-up test
===========
CM with no communication 1.58s
BCM with no communication 2.69s

For the folding benchmark Cyclic and Block both took about the same amount of time and generated the same amount of messages. Modulo Unrolling made Cyclic over twice as fast as it was originally and reduced the number of messages by over 90%

For the Jacobi benchmark Block has significantly less messages and is about 5 times faster than Cyclic. But with Modulo Unrolling the gap is much closer, Block is only 35% faster and 35% fewer messages.

For the Pascal benchmark Block had a similar advantage over Cyclic. Modulo unrolling gave a similar speedup, about 300% faster. The Block Cyclic distribution could also be used in this benchmark. It had roughly 1/16 as much communication volume as cyclic (and similar communication count), but was still slower overall.

As an experiment to understand why Block Cyclic was slower than cyclic despite having fewer communication volume, the Pascal benchmark was run again but with communication calls removed. Therefore all time was a result of only the overheads not associated with communication. The speed was nearly identical indicating that Modulo Unrolling had nearly completely removed the cost of communication relative to other overheads. More investigation is needed, but it seems further gains will come from improving Chapel implementation or other inefficiencies with the implementation of the iterators.

How to run the code
Download TRUNK version of Chapel by running:

```
svn checkout http://svn.code.sf.net/p/chapel/code/trunk chapel-code
```

go to main Chapel directory (commands for tcsh)

```
setenv CHPL_COMM gasnet
setenv GASNET_SPAWNFN L #(to run simulating multiple locales)
```

```
make
chpl –fast program –o program
./program –nl 4 #to run with 4 locales
```

Low level chapel communication calls quick explanation

See the paper titled “Proposal for Extending the UPC Memory Copy Library Functions and Supporting Extensions to GASNet” by Dan Bonachea, http://www.escholarship.org/uc/item/5hg5r5fs?display=all#page-1

Bulk put/get and strided put/get low level commands have been added to Chapel. Their use is best understood by the paper. A useful example is on page 18. In Chapel the put command is:

```
__primitive("chpl_comm_put_strd",
                pointer to source,
                source strides,
                remote locale id,
                pointer to destination,
                destination strides,
                copy length,
                number of dimensions
              );
```

This command is pseudo multi dimensional, so the strides and copy length are arrays of strides and lengths. Basically memory is copied starting at the source pointer for copy_length[0] elements. Then from the source pointer move source_strides[0] elements forward. Repeat this copy_length[1] times, then step source_strides[1] elements forward from the source pointer. Repeat all this copy_length[2] times. Etc… The same is done for choosing the destination to place copied elements.

Future Work

1. Investigate how much overlap there is with an existing optimization in chapel for which handles copying distributed arrays: http://www.ac.uma.es/~compilacion/publicaciones/UMA-DAC-12-02.pdf
2. Remove non communication overhead of Chapel yield / iterator implementations
3. Fully implement Block-Cyclic Distribution (slicing)
4. Fully implement modulo unrolling optimization in Block Cyclic
    a. Handle n-dimensions instead of 1-d
    b. Handle slicing (once that is implemented)
5. Make the optimization perform strip mining
    a. Break the iteration into smaller sections to conserve memory

    b. Do non-blocking ahead-of-time memory gets to overlay communication with computation

6. Modify the Chapel compiler to convert affine array indices in loops to zippered iteration so that the optimized zippered iterator can be automatically applied more often. Optionally choose the leader of the zippered result to minimize communication (see section 3.3 of this paper).
7. Modify the Chapel compiler to inform the iterators if their yielded values could possibly be read or written to. If value is not used then don't prefetch into buffer; if value is not written to then don't update remote locale with buffers new values.
8. Currently the optimization is only applied if all dimensions of the array are compatible. But the non-compatible iterations could theoretically be looped over normally and the optimization could be applied to all the remaining dimensions. It is unclear how common this case is, it could be very rare.