Good afternoon everyone. The title of my thesis defense is “Affine Loop Optimization using Modulo Unrolling in Chapel.” Before we get started, I would like to thank my thesis committee members, Dr. Franklin and Dr. Papamanthou, my advisor Dr. Barua, Darren Smith and Joshua Koehler, two former UMD students who spent time working on this research, and Michael Ferguson, our program manager at the Laboratory for Telecommunication Sciences in College Park, Maryland.

x1

The overall goal of this research is to improve the runtime of programs run on message passing machines. These types of machines are used throughout high performance scientific computing to solve large problems. A message passing architecture consists of a number of computing locales connected by an interconnection network. Each locale is made up of a set of processors and memory that it can access locally. However, to access the memory on another locale, communication between locales must take place over the interconnection network. This is commonly referred to as “Remote Direct Memory Access” (RDMA). Common message passing architectures are made up of tens or hundreds of locales, but the largest systems today contain hundreds of thousands of locales! At this scale, compilation becomes complicated in part because of all the communication.

x2

Compilation for message passing architectures can be simplified if the programming language is a PGAS language. PGAS stands for “Partitioned Global Address Space”, and this model provides the illusion of shared memory on a real distributed memory system. This allows the programmer to reason about locality without constantly dealing with low-level data transfers between locales. Chapel is the PGAS language that this research focuses on.

x1

Chapel is a new explicitly parallel programming language developed by Cray in an effort to increase programmer productivity for high performance computing developers. What is appealing about this language is that it is open source, allowing anyone to dive into the code, and it targets both large scale and desktop systems. This means that I can run the same program that I run on a supercomputer on my laptop.

x1

Our work’s contribution is as follows: We present an optimization for parallel loops with affine array accesses in Chapel. The optimization uses a method known as modulo unrolling without unrolling to aggregate messages and improve the runtime performance of loops for distributed memory systems that use message passing.

x1

So, let’s start with the Introduction and Motivation of the problem we are trying to solve.

x3

I mentioned before that our optimization targets parallel loops with affine array accesses. These types of array accesses are extremely common in scientific computing, and here are some common examples.

x1

An array access is considered affine if the access on each dimension of the array is a linear expression of the loop induction variables.

x2

For example, if array **A**, shown here, is 1-dimensional and in a loop nest with induction variables **i** and **j**, the expression **ai+bj+c** is affine if **a,b**, and **c** are constant integers.

x2

In Chapel, programmers can use affine array accesses with parallel forall loops, as seen here in this simple example. The two affine array accesses are A[i] and B[i+2]. This loop assigns a subset of elements of array **B** to a different subset of elements of array **A** in parallel, so there is no guaranteed order of loop iterations completing.

x1

This process is simple when all data is on a single locale, but the following question arises: What happens when the data is distributed?

x1

In Chapel, a data distribution describes how data is allocated across locales of the system for a particular program. We define a locale to be any unit of a distributed computer consisting of its processors and memory. In Chapel, programmers have the freedom to distribute data using Chapel’s built-in modules, or they have the freedom to develop their own distributions. In this research, we considered three of Chapel’s built-in distributions: Block, Cyclic, and Block Cyclic.

x5

The Block distribution divides up the data into equal chunks of consecutive elements, and each locale gets one chunk. This example shows how an array **A** is distributed among 3 locales using Block, with each locale getting 5 consecutive elements. Array elements that are the same color reside on the same locale. For this and the next two slides, you don’t have to understand the code shown above, only the final picture of how data is distributed.

x5

The Cyclic distribution assigns data to locales in a round robin manner. This results in consecutive elements on a single locale being separated by a fixed “cycle” in the global view of the data. This “cycle” is equal to the number of locales. For example, now array **A** is distributed among 3 locales using Cyclic, and any two consecutive elements on a locale are separated by 3.

x5

The Block Cyclic distribution can be thought of as a hybrid between the Block and Cyclic distributions. Here, the user specifies a block size parameter, which is the number of consecutive elements to assign to a single locale. This pattern is then repeated, cycling through all locales until all array elements have a home. In this example, array A is distributed with Block Cyclic with a block size parameter of 3.

x1

The past three examples all distributed 1-dimensional arrays, but similar Chapel code and intuition follows for multi-dimensional arrays.

x3

So, our original parallel loop example looks like this if the data is distributed with Block over three locales. However, notice now that during some iterations of the loop, we want to read and write array elements that don’t reside on the same locale. This means that a message containing the remote data element must be sent.

x9

In this example, there are 4 messages denoted by the red arrows, which arise when B[i+2] resides on a different locale than A[i].

x1

In the previous simple example, we can identify areas for data communication improvement in Chapel.

x1

First, we can remove the locality check that happens for every loop iteration because it is clear which loop iterations will require messages before the loop starts. For our example, this locality check is equivalent to asking “Is B[i+2] local or remote?” Checking the locality of each array element slows down the loop.

x1

Each message only contains 1 data element.

x1

In the previous example, we could have aggregated messages using strided bulk communication primitives (such as GASNET). Specifically, elements B[6] and B[7] could be sent to locale 0 in a single message, and elements B[11] and B[12] could be sent to locale 1 in a single message, for a total of 2 messages instead of 4.

x1

These issues with data communication are a larger problem for programs with larger input sizes and more locales, where communication takes up more time than computation.

x1

To further motivate why aggregation will improve runtime performance, lets analyze a real interconnection network, Infiniband. This graph measures the latency of Infiniband as a function of the message size in bytes. It’s clear from this graph that the latency increases as message size increases, but only by a small amount (only 1 microsecond). The latency cost per byte, which isn’t shown in this graph but can be derived from dividing each data point by its message size, actually decreases with message size. So, it is faster per byte to access larger portions of data.

x1

This graph measures the bandwidth of Infiniband as a function of message size. Bandwidth increases as message size increases, supporting the case to aggregate messages because more data can be sent over the network in the same amount of time. To sum up, it is much faster to transfer data in bulk as opposed to individually because communication networks are generally limited by latency and not bandwidth.

x1

So, how can we improve these problems?

x1

Well, we can use knowledge about how data is distributed and loop access patterns to aggregate messages and reduce the runtime or parallel affine loops.

x1

It’s important to note here that we are not trying to apply automatic parallelization in Chapel, come up with a superior data distribution, or develop a theory to override the programmer’s choice of data distribution.

x1

Our goal is to improve Chapel’s existing data distributions to perform better than their original implementation.

x1

Next, let’s briefly review some related work in the field of compiler loop optimizations and message aggregation.

x4

The traditional method of compiler loop optimizations consists of transformations such as loop fission, loop fusion, loop peeling, etc. These methods are very good at discovering parallelism, increasing the granularity of parallelism, and improving cache performance.

x1

However, these methods use complicated message passing code generation steps, causing production compilers to limit message aggregation only for simple cases. The message aggregation step requires the footprint of a loop to be intersected with the data distribution. These intersections then need to be split into remote portions coming from a single locale. None of this can be modeled with a matrix transformation, which makes this step difficult.

x6

In the polyhedral method, each array use of a parallel loop is intersected with the data distribution in order to determine remote data. This method is comparable to the traditional method because remote data sequences can found in a single step. This method also is good at increasing the granularity of parallelism and improving cache performance. However, just like the traditional method, the polyhedral method does not compute information for message passing code generation. Ad hoc methods have been added to the polyhedral method to do so, but these methods further complicate the compilation process in the same way as the traditional method. Our method is unique in that it does not require complex intersections between the loop’s footprint and the data distribution. This will become clearer in later slides. It’s important to note here that our method is not a replacement to the traditional and polyhedral methods, only an improvement on their message passing code generation steps.

x6

The last branch of related work involves message aggregation methods for other PGAS languages. For example, the UPC compiler performs message aggregation using methods such as redundancy elimination, split-phase communication, and communication coalescing. This work improves the performance in programs with fine-grained communication. However, aggregation can only occur for Block and Block Cyclic distributions and not the Cyclic distribution. Our method targets both Cyclic and Block Cyclic. For their method applied to Block Cyclic, aggregation occurs within blocks, whereas our method aggregates across blocks. Aggregating across blocks is superior because the total number of blocks for a distribution is almost always more than the number of elements within a block.

x1

Now, let’s explore modulo unrolling, the method we use in this research to aggregate messages.

x1

Modulo unrolling is a method developed by Dr. Barua that statically disambiguates affine array accesses inside any loop, serial or parallel. Static disambiguation is the property that each affine array access will map to a single locale across all loop iterations. We will see in the next few slides how this is done, but the main idea to this method is to unroll the loop by a factor equal to the number of locales of the system. This method was originally intended to improve memory parallelism on tiled architectures, and it is applicable to the Cyclic and Block Cyclic distributions.

x4

Here is a simple sequential loop with three different affine array accesses: 2 **A[i]** accesses (one read and one write) and 1 **B[i+1]** access. In this example, arrays **A** and **B** are distributed cyclically, and it is clear that both affine array accesses map to different locales on successive loop iterations.

x1

For instance, when i = 1, the loop accesses A[1] and B[2], which map to locale 0 and 1, respectively.

x1

Then, when i = 2, the loop accesses A[2] and B[3], which map to locale 1 and 2, respectively.

x2

Now, we unroll the loop by 4. This adds a stride to the loop header and increases the code size of the loop body.

x2

After unrolling, each affine array expression maps to s single locale. In other words, each affine array access will be statically disambiguated.

x2

For instance, A[i] will always map to locale 0 and B[i+1] will always map to locale 1, for all i in the range 1 to 99 stride by 4.

x1

We can perform the read to A[i] and B[i+1] in parallel because they reside on different locales, and this successfully improves memory parallelism. The question that remains is: how do we apply this concept in Chapel to perform message aggregation?

x1

To do so, we had to modify the modulo unrolling method into what we call “modulo unrolling without unrolling.” It takes as input a parallel loop containing affine array accesses where data is distributed cyclically or block cyclically and outputs an equivalent loop that aggregates the remote accesses of data into fewer messages. This new method takes three steps: first the Block Cyclic transformation step, then the owning expression calculation step, and finally the message aggregation step.

x2

Here is an illustration of the Block Cyclic transformation. The top block of code represents the input parallel loop with loop induction variable **i**, starting loop bound **s**, ending loop bound **e**, and loop stride **n**. The loop body can contain any number of affine array expressions. In the bottom block of code, we’ve performed a loop transformation called strip mining to the original loop. The transformation adds an outer loop nest to the code that iterates through each position within a block. The inner loop now handles iterations across blocks. The main observation is that after the transformation, loop iterations in the same position within a block are processed together. We end up with many “adjacent” Cyclic distributions.

x1

For example, the figure below shows how a loop with affine array access A[i] is strip mined. Loop iterations 1, 4, 7, 10, etc are processed together. Similarly, elements 2, 5, 8, 11, etc are processed together. Both of these sets of loop iterations can be thought of separate Cyclic distributions. After this step, affine array accesses are statically disambiguated with respect to the local view of data.

x1

We would like to minimize the number of remote data accesses in the loop body. So, we should assign loop iterations to locales where most of the affine array accesses are local. The owning expression calculation step involves finding the affine function of the loop induction variables that occurs the most number of times within the loop body. In our example code in the last slide, each affine array access occurred once, so either can be chosen as the owning expression. For the rest of this description, a1\*i+b1 is the owning expression.

x4

The last step to modulo unrolling without unrolling is the message aggregation step. It transforms the strip mined loop into an equivalent loop that aggregates remote elements into fewer messages. The pseudocode shown here is the final transformed loop. The details are fairly complex, so let me give you an intuition of what is happening. Line 1 is the same strip mining loop nest from the Block Cyclic transformation. However, line 2 adds another loop nest to iterate through each locale of the system (where the total number of locales is **N**). This loop nest, and the owning expression if check on line 3, finds the subset of loop iterations that each locale is responsible for. The next step is to fetch the elements from each non-owned affine array access using a single message and store them to a local buffer, which is done on line 6. Now we encounter the original parallel loop in lines 8-12, but we replace the remote affine array access with an access to the local buffer. Finally, in case that the elements in the local buffer are written to during the loop, we write back the local buffer to the remote locale in a single message, after the parallel loop finishes. This is shown in lines 13-16.

x1

Now that we have an intuition of how modulo unrolling without unrolling works, let’s review how this optimization was implemented into the Chapel programming language. This next section assumes some basic knowledge of Chapel syntax, but I will try to keep things as intuitive as possible.

x1

Modulo unrolling without unrolling is implemented using iterators in Chapel. An iterator is a block of code similar to a function, except that it can return multiple values back to the call site. Instead of using the keyword “return,” iterators are said to “yield” values. These constructs are heavily used in loops.

x1

For example, shown here is a Chapel iterator that yields the first **n** Fibonacci numbers. It takes in a single parameter **n.**

x3

We can use this iterator within a loop to print out the first 5 Fibonacci numbers. The Chapel code to do so looks like this. The variable **f** corresponds to a single Fibonacci number that the iterator will yield.

x1

As expected, the output of the resulting loop is indeed the first 5 Fibonacci numbers.

x1

There is nothing stopping us from traversing multiple iterators at the same in a Chapel loop. This is called zippered iteration. One condition is that both iterators must yield the same number of objects back to the call site.

x1

Here is an example. Now, the iterator fib(5) is traversed together with the range 1..5. The syntax of the loop header now changes, as we must specify a loop variable for each item in the zippering.

x1

This loop output shows that the jth yielded element of 1..5 and the jth yielded element of fib(5) are yielded together.

x4

Zippered iteration can be used in parallel loops as well, but in order work, each iterator in the zippering must have a leader iterator and a follower iterator. The leader iterator is responsible for creating parallel tasks and assigning loop iterations to these tasks. The follower iterator then carries out the loop iterations specified by the leader.

x1

Here is an example of a parallel loop that uses zippered iteration.

x1

By convention, the leader iterator of the first item in the zippering will be called to create parallel tasks. In this case, that is the leader iterator of **A.**

x2

Then, the follower iterators of all items in the zippering are called to handle each parallel task. More information on parallel zippered iteration semantics can be found here.

x1

It turns out that any parallel forall loop with affine array accesses has an equivalent form using zippered iteration. Each affine array accesses is replaced with an array slice, which is a smaller portion of the entire array. Array slices are iterable in Chapel.

x1

For example, the first parallel loop that we saw in this presentation can be written using zippered iteration as follows.

x2

We’ve replaced the affine array accesses A[i] and B[i+2] with the array slices A[1..10] and A[3..12].

x1

So, it is possible to implement modulo unrolling without unrolling directly into the leader and follower iterators of the Cyclic and Block Cyclic distribution modules. At a high level, the leader iterators are responsible for performing the Block Cyclic transformation and owning expression calculation, while the follower iterators are responsible for the message aggregation step. It’s quite remarkable that this optimization does not need to be implemented as a traditional compiler pass, and can be written in Chapel, the very language it is meant to optimize!

x1

Let us now return to the first distributed parallel loop that we examined at the beginning of this talk, now expressed using zippered iteration. When arrays **A** and **B** are distributed cyclically, the implementation with respect to locale 0 works as follows:

x2

The leader iterator of A assigns locale 0 iterations 1, 5, 9, etc.

x1

The follower iterator of B will be remote for all iterations of the loop. Its chunk of work contains elements B[3], B[7], B[11], etc.

x1

These elements are all remote on locale 2 and separated by a fixed stride of 4. Therefore, they can be brought to locale 0 in one strided get message.

x1

Each element is yielded back to the loop header from the local storage on locale 0.

x1

Finally, if any of the yielded elements were written to during the loop, a bulk put message writes the local storage back to the remote locale. This entire process is done by modifying only the Cyclic distribution’s follower iterator.

x2

Now arrays **A** and **B** are distributed block cyclically with a block size of 2. Focusing again just on locale 0, aggregation occurs with elements in the same position within each block.

x4

That means that the leader iterator creates two tasks for locale 0: one containing iterations 1, 9, etc, and a second containing iterations 2, 10, etc. The aggregation step is equivalent to the Cyclic example, but happens for both parallel tasks.

x1

For this to work, both the leader and follower iterators of the Block Cyclic distribution needed to be modified. The actual modifications to the leader and follower iterators are not shown here, but they are reviewed in detail in my thesis.

x1

Finally, lets review our experimental evaluation. We conducted four sets of experiments: a benchmark suite evaluation, a strong scaling experiment, a weak scaling experiment, and a blocksize variation experiment.

x1

For the benchmark suite evaluation, we gathered a set of benchmarks that contained parallel loops with affine array accesses and measured their runtime and message count performance with and without modulo unrolling for both the Cyclic and Block Cyclic distributions. Many of the benchmarks were taken from the Polybench suite and translated to Chapel by hand. Only two benchmarks, jacobi-1D and pascal, could be tested with the Block Cyclic implementation because of limitations within the Block Cyclic module that are outside the scope of this research. We report the input sizes of each benchmark, and as you can see, our suite of benchmarks uses one-, two-, and three-dimensional arrays. We also report the number of elements per follower iterator chunk for each benchmark, a number measured experimentally that indicates how many data elements are aggregated into a single message. The higher the number, the higher the degree of aggregation.

x1

Data was collected on the 10-locale Golgatha cluster computer at the Laboratory for Telecommunication Sciences. The cluster is equipped with Infiniband in order for nodes to communicate with one another. Each locale contains two 6-core Intel Xeon processors with 2-way hyperthreading for a total of 12 hardware threads per processor.

x1

Here are the runtime improvement results. Each benchmark is on the x-axis, and the normalized runtime with respect to the existing Chapel distribution is measured. Red bars correspond to the existing Cyclic distribution and are at 1 for each benchmark. Green bars correspond to the Cyclic distribution with modulo unrolling. Blue bars correspond to the existing Block Cyclic distribution and like the red bars, are at 1 for the two benchmarks that we tested with Block Cyclic. Pink bars correspond to the Block Cyclic distribution with modulo unrolling. The way to interpret the improvement of our method is to compare the red bars with the green bars, and the blue bars with the pink bars, for each benchmark. Bars lower than 1 mean that the benchmark ran faster using modulo unrolling. Overall, 10 out of the 16 benchmarks distributed with Cyclic improved in runtime when modulo unrolling was used, and both benchmarks distributed with Block Cyclic improved in runtime when modulo unrolling was used. The last set of bars labeled “mean” measures the geometric mean of all normalized runtimes. This gives us a sense of how well modulo unrolling performed overall. On average, Cyclic with modulo unrolling results in a 36% reduction in runtime, and Block Cyclic with modulo unrolling results in a 53% reduction in runtime.

x1

Here are the message count improvement results organized in the exact same way as the runtime results. Overall, 13 out of the 16 benchmarks distributed with Cyclic ran with fewer messages when modulo unrolling was used, and both benchmarks distributed with Block Cyclic ran with fewer messages when modulo unrolling was used. On average, Cyclic with modulo unrolling results in 64% fewer messages, and Block Cyclic with modulo unrolling results in 72% fewer messages.

x7

This slide just restates the runtime and message count averages that I just went over. You’ll notice that some benchmarks performed slower when the Cyclic distribution with modulo unrolling was used. If we go back to the benchmark table, we’ll see that all of these benchmarks that were slower are examples of programs with few data elements per follower iterator chunk. It turns out that there is a threshold of elements greater than 2 where sending a bulk message first starts to be faster than sending many individual messages. If the number of elements to aggregate is below this threshold, a bulk message will be slower than many individual messages. This threshold exists because of the differences between the bulk communication primitives (which do not have hardware support) and the default communication primitives (which do). Dynamic checks could be added to the modified distributions to fall back to the original implementation if not enough aggregation exists. This would ensure that slowdowns don’t occur.

x1

The observation that the number of elements per follower iterator chunk might have something to do with how much improvement we get with our method motivated us to perform some scaling experiments. We performed these experiments on a subset of benchmarks that improved the most with our method. In the strong scaling experiment, we ran benchmarks of the same input size while varying the number of locales, measuring the normalized runtime, normalized message count (as before), and the number of elements per follower iterator chunk.

x1

Here are the strong scaling results for the jacobi-2d benchmark. The scaling experiment figures are a bit complicated to read, so let me explain the structure for this one, and the same structure will apply for the remaining experiments. On the x-axis we vary the number of locales from 2 to 8. There are actually two y-axes; the right axis is labeled “normalized measurement” and refers to normalized runtime for the red curve and normalized message count for the green curve. The left axis measures the number of elements per follower iterator chunk and only is applicable to the blue curve. The key takeaway from the strong scaling results is that the performance improvement of our method decreases as the number of locales increases (red and green curves approach closer to 1). This is directly related to the fact that the fewer elements reside on each locale.

x1

In the weak scaling experiment, we keep the number of locales constant as we vary the input sizes of the same subset of benchmarks, measuring the same three things as before.

x1

Here are the weak scaling results for the jacobi-2d benchmark. Now input size is on the x-axis. As input size increases, so do the number of elements per follower iterator chunk and the performance improvement due to modulo unrolling (red and green curves approach 0). This is expected because a larger input size means more elements per locale.

x1

Finally, the block size variation experiment tests both benchmarks that were run using the Block Cyclic distribution. Here, the block size parameter changed while input size and number of locales remained constant.

x1

Here are the blocksize variation results for the pascal benchmark. Blocksize is shown on the x-axis. We can see that increasing blocksize decreases the number of elements per follower iterator chunk and decreases the performance improvement due to modulo unrolling (red and green curves rise above 1). While the number of elements on each locale is the same regardless of block size, increasing the block size creates more “adjacent” Cyclic distributions, and the number of elements in each Cyclic distribution has to go down.

x4

So, in conclusion, I’ve presented modulo unrolling WU, a compiler transformation that performs message aggregation in affine programs where data is cyclically or block-cyclically distributed. Our results show that our method implemented in Chapel improves runtime and communication performance for programs with sufficient parallel chunk sizes.

x3

Thank you, and I’ll now take your questions.