**Quarterly report for project**

**October-December, 2013.**

**Title**: Optimizing the Chapel compiler **Targeted area**: High-performance computing

**PI**: Rajeev Barua **Cognizant LTS officer**: Michael Ferguson

This project is improving the quality of the code output by the compiler for the Chapel language, to improve its run-time on parallel computers. Chapel is an explicitly parallel language developed by Cray Inc. as part of the DARPA HPCS program. The LTS has already demonstrated that Chapel is a promising parallel programming language for its needs, but research questions remain for its performance and usability, some of which are being addressed in this project.

Project personnel this quarter include Aroon Sharma, a graduate student at the University of Maryland, Joshua Koehler, a senior undergraduate student at the University of Maryland, and PI Barua. All are working closely with Mr. Michael Ferguson, an LTS staff member who is leading an effort to improve the Chapel platform. Aroon has taken the lead in the design and implementation of the compiler optimizations in the Chapel compiler, and is working under the active guidance of PI Barua. Past personnel include former University of Maryland graduate student Darren Smith.

We are investigating ways to reduce the run-time of compiled code by exploring how the message passing code can be optimized for parallel global address space (PGAS) languages such as Chapel. The run-time of message-passing code generated by the Chapel compiler has been found to be not competitive with the state of the art. To understand the problem, consider that for message passing programs, the Chapel language allows the programmer to optionally specify the data partitioning among nodes, whereas the loop partitioning is automatically decided by the compiler. The Chapel language specifies parallelism in a declarative style using *forall* loops, using which the programmer is encouraged to express all available parallelism.

Given its PGAS nature, the current Chapel compiler accesses each array element using a run-time check which checks if the run-time-computed address of that element is local to the processor on which that iteration is running. If it is local, then a local memory access is executed; otherwise a message is sent to a remote memory to access the array element. There are three run-time costs incurred in the generated code: First, the run-time check itself adds some overhead. Second, since loop and data partitions are chosen without regard to locality, sub-optimal locality often results; leading to many more remote accesses than in code with good locality. Third, since the default compilation of PGAS languages generates messages for each accessed remote memory word separately, no aggregation of messages is done. This is sub-optimal because the per-word run-time cost of messages is lower in larger messages than smaller ones.

**Modulo Unrolling** In this quarter we have designed a method for compiling PGAS code on message-passing hardware, based on modulo unrolling [1], a method designed by the PI Barua while a graduate student at MIT. Modulo unrolling was originally intended for a different purpose – to enable compilation of serial programs to the MIT Raw architecture with instruction-level parallelism. Modulo unrolling is useful for the Raw machine since Raw uses a *static network*, i.e., a network in which the presence and path of messages is decided at compile-time, and their routing is done explicitly using instructions executed on each *message processor* on each intermediate node on the path of the message. The intuition behind modulo unrolling is the following: in programs where the arrays are cyclically distributed and accessed by affine accesses, there always exist certain loop unroll factors (that can be calculated by the theory) such that if the loop nest is unrolled by those factors, then in the unrolled loop, each array access will access only one memory node. This is called the *static residence property*, since the array element can thereafter be accessed from any node using messages on the static network.

At first glance modulo unrolling does not seem to be useful for message passing machines, since they have a dynamic (rather than a static) network. However, upon closer examination an opportunity arises. We notice that in PGAS languages, it is not clear what node(s) each memory access references; hence it is difficult to optimize for locality of the access. We can use modulo unrolling to solve this problem. When modulo unrolling is applied, the target node of each memory reference becomes a single node which is known at compile-time. This will enable the compiler to reason about locality, and place the reference on (or close to) its target node.

Using modulo unrolling for compiling PGAS languages for message passing machines reduces all three sources of overhead mentioned above in the current Chapel compiler. First, the run-time check that checks whether the target node for each memory reference is local or remote can be eliminated, removing its overhead. This is because with modulo unrolling, the target node of each memory reference is no longer variable, but rather is a single known node. As a result, the outcome of the run-time check is known at compile-time, allowing its elimination. Second, the overhead of excessive communication arising from poor locality can be reduced. This is because modulo unrolling reveals the target node for each array reference, enabling the reference to be scheduled on a node that is the same as the target node, or close to it. Third, opportunities for message aggregation become easy to discover and implement. Our proposed code generation strategy with modulo unrolling will place all the references to a single target node in a single loop. Thereafter it will be straightforward to replace that portion of the loop with a single aggregate message to the target node.

The strategy above provides significant advantages compared to existing methods of compilation for message-passing machines which require very complex memory foot print analysis to achieve the same level of optimization (eg [2]). Those methods are extraordinarily difficult to implement and have limited scope in terms of the programs they can handle.

**Progress and implementation** So far, our group has designed a variant of modulo unrolling that will work for a message passing machine. In this design, the SPMD code is parameterized in the loop using the node id. The memory accesses themselves can be used to predict the portions of data on each node.

As of September 2013, our group has implemented a version of the cyclic and block-cyclic distributions that take advantage of modulo unrolling in the Chapel compiler. Both distributions contain modified iterators that determine whether modulo unrolling can be applied. Both of these new distributions achieve improved performance for message passing programs across multiple locales by observing that loops with affine array accesses in Chapel can be converted to loops using zippered iteration. Using zippered iteration to implement modulo unrolling simplifies the complexity of aggregating messages because each iterator can decide whether data is local or not.

In the modified cyclic distribution implemented as a Chapel module, the follower iterator (CyclicArr.these(param tag: iterKind, followThis, param fast: bool = false) var where tag == iterKind.follower) has been modified to perform modulo unrolling. In the modified block-cyclic distribution implemented as a Chapel module, both the leader iterator (iter BlockCyclicDom.these(param tag: iterKind) where tag == iterKind.leader) and the follower iterator (iter BlockCyclicArr.these(param tag: iterKind, followThis) var where tag == iterKind.follower) have been modified to perform modulo unrolling.

Our group has written three benchmarks to test the performance of our optimized cyclic and block-cyclic distributions that take advantage of modulo unrolling versus the existing distributions that do not. For the Pascal benchmark, we see improvements in runtime and message count for both cyclic and block cyclic distributions with modulo unrolling. Cyclic with modulo unrolling ran 3.25 faster with 90 percent fewer messages compared to cyclic without modulo unrolling. Block-cyclic with modulo unrolling ran 3.77 times faster with 90 percent fewer messages compared to block-cyclic without modulo unrolling. For the Folding and Jacobi benchmarks, we see similar improvements in runtime and message count for the cyclic distribution with modulo unrolling (block-cyclic with modulo unrolling was not tested). Our group has also written a paper summarizing these results and implementation.

This quarter, Aroon has been working on modifying the optimized cyclic and block-cyclic distributions to reduce the memory pressure on local memory from messages sent from remote locales. The current implementation of both optimized data distributions allows messages to be sent that are unbounded in the number of array elements they contain. This results in messages potentially taking up a significant portion of a locale’s memory for computations of large data sets. To solve this issue, the message size should be checked, split into multiple messages if it is deemed too big, and sent in portions of an appropriate size. In terms of an implementation, a technique known as strip mining is used to achieve this functionality. We do not anticipate a major performance impact in terms of message count by applying this technique. Although large messages will be sent as multiple messages, the threshold for judging whether a message is too large will be very high.

Aroon has successfully applied this strip mining memory usage technique to the optimized cyclic distribution for one-dimensional array accesses. He has also run experiments confirming the correctness of sample loops using this technique. This technique still needs to be extended to handle data sets that are multiple dimensions in size. It then needs to be applied to the optimized block-cyclic distribution.

Under Aroon’s guidance this quarter, Josh has been responsible for putting together a suite of 15 parallel Chapel benchmarks that will be used to evaluate our modulo unrolling optimizations. These benchmarks will be used in addition to the Pascal, Folding, and Jacobi benchmarks created by Darren last quarter. Many benchmarks currently exist in the source tree for the Chapel compiler, but they are not implemented using zippered iteration. Currently, benchmarks need to be implemented using zippered iteration in order for our modulo unrolling optimizations to be applied. Josh is working on converting the implementations of these benchmarks to zippered iteration. Some parallel benchmarks (primarily in the Polybench suite) that we would like to consider are not implemented in Chapel. Josh has also been working to translate these benchmarks from their current languages into Chapel. As of the end of this quarter, Josh has implemented 3 parallel benchmarks in Chapel using zippered iteration: 2mm, 3mm, and correlation.

We have reached an agreement with Mr. Ferguson where Aroon works at LTS one day a week, thereby allowing him to get guidance from Mr. Ferguson promptly on coding matters in the Chapel. Mr. Ferguson provides feedback and direction to Aroon and both are working to have this work be a part of a future release of the Chapel compiler. In addition, both Aroon and PI Barua meet with Mr. Ferguson every two weeks at LTS to discuss overall strategy and provide status updates. Aroon and Josh meet together once a week so Josh can get assistance on his benchmark implementation and better understand the Chapel programming language. In addition, Aroon, Josh, and PI Barua meet together once a week to discuss the progress on the project.

**Future Work** To reduce message count further, Aroon would like to add additional checks to the Chapel compiler in order to determine whether zippered iteration variables are read from or written to during each loop iteration. The intuition behind this optimization is that if a variable is not read, there is no need to send a message retrieving nonlocal data. If a variable is not written to, there is no need to send a message updating nonlocal data. Therefore, for each zippered iteration loop variable, the Chapel compiler must set two flags: one if the variable is read from during the loop iteration and one if the variable is written to during the loop iteration. This read and write information must be communicated back to the optimized iterators in order for them to decide whether a message should be sent or not.

A possible future direction to this project involves figuring out how our optimized cyclic and block-cyclic distributions can be used more frequently in message passing code. One way to do this is to recognize that any parallel *forall* loop in Chapel that uses affine array accesses can be converted to an equivalent zippered iteration loop. Therefore, modulo unrolling provided by our optimized distributions can be applied to loops written in a non-zippered iteration style. Aroon is exploring ways for the compiler to recognize loops written in a non-zippered iteration style and convert them to zippered iteration during compilation so modulo unrolling can also be applied to these loops. To do this, Aroon would work on a compiler pass that transforms non-zippered intermediate representation (IR) to zippered IR. This way, a user would be able to write a loop in source code in either style but still take advantage of the modulo unrolling optimization.

**References:**

[1] Barua, Rajeev, Walter Lee, Saman Amarasinghe, and Anant Agarwal. *"Maps: a compiler-managed memory system for raw machines."* ACM Proceedings of the 26th annual international symposium on Computer architecture (ISCA '99). Volume 27 Issue 2, May 1999. Pages 4-15.

[2] Yelick, Katherine, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L. Graham et al. *"Productivity and performance using partitioned global address space languages."* In International Conference on Symbolic and Algebraic Computation: Proceedings of the 2007 international workshop on Parallel symbolic computation, vol. 27, no. 28, pp. 24-32. 2007.