**Quarterly report for project**

**July-September, 2014.**

**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 and PI Barua. Both 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 and Joshua Koehler.

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 continued our design of 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 message passing machines. 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. With Michael’s help and supervision, we have tested our loop optimization using modulo unrolling on 17 benchmarks on the Golgatha cluster at LTS.

This quarter, Aroon has finished his conference paper submission to the 2014 PGAS Conference in Eugene, Oregon. His work has officially been accepted to the conference, where he will give a half hour presentation describing his work and results in early October.

Also this quarter, Aroon is working on a journal article submission, which will contain additional experiments and improvements to his conference paper. First, he is conducting an input variation experiment. For a subset of the benchmarks we tested with modulo unrolling in Chapel, he will measure runtime and communication performance improvements as a function of different input sizes. Keeping all other parameters constant, input size directly determines the amount of communication aggregation possible for a program. Higher input sizes lead to more aggregation. The goal with this experiment is to find the input size at which using modulo unrolling to aggregate messages first starts to improve performance in both runtime and communication. This information can be used in modulo unrolling’s implementation to dynamically turn off the optimization for programs with input sizes that are too small. Ultimately, our goal here is to fall back on the original follower iterator’s implementation for these cases in order to achieve better overall performance.

Aroon’s second experiment involves the Block Cyclic implementation of modulo unrolling in Chapel. He is testing what affect different block sizes have on the performance improvements seen by modulo unrolling. Normalized runtime and communication count numbers will be collected for block sizes small to large. This experiment will help determine the block size where modulo unrolling provides the best performance improvement over the existing Block Cyclic follower iterator.

Finally, Aroon is continuing to work on submitting his work on modulo unrolling to the Chapel compiler repository. Our goal is to have the work in the next release of the compiler.

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 In addition, Aroon and PI Barua meet together once a week to discuss the progress on the project.

**Future Work** Additional improvements in communication performance within the Cyclic and Block Cyclic distributions can be achieved through the use of a non-blocking communication scheme that performs aggregation. Currently, modulo unrolling performs aggregation using a blocking communication scheme. This means that during aggregation and communication, computation stops. If we were able to pipeline communication and computation, we would overlap both tasks and possibly get better performance. To do this, we would need to use a non-blocking communication scheme, which is possible within Chapel but has not yet been explored in this project. We would like to modify our optimized distributions to use a non-blocking communication scheme and test the performance improvements.

**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.