Compiler Optimization for Irregular Memory Accesses in Chapel

Thomas B. Rolinger
tbrolin@cs.umd.edu
University of Maryland
College Park, MD, USA

Alan Sussman
als@cs.umd.edu
University of Maryland
College Park, MD, USA

ABSTRACT

The Chapel programming language and runtime provides programmers with productivity advantages for developing applications with irregular memory access patterns. Specifically, language support for PGAS-style operations and implicit remote communication lessen the burden on the programmer to reason about remote memory accesses to distributed arrays. However, these productivity benefits naturally lead programmers to write code that causes fine-grained remote communication, which poses significant performance challenges on modern computing systems. Our prior work has demonstrated that the performance of these irregular codes in Chapel can be dramatically improved by manually applying optimizations that selectively replicate remotely accessed data at runtime. In this talk, we will discuss our current efforts on bridging the gap between performance and productivity for irregular applications in Chapel by automating the selective data replication optimization. To this end, we have designed and implemented a compiler optimization that automatically identifies candidate irregular accesses using static analysis and applies transformations to produce optimized code, all without requiring user intervention. Our results show that runtime performance can be improved by as much as 52x on a Cray XC system with a low-latency interconnect and 364x on a standard Linux cluster with an Infiniband interconnect.

KEYWORDS

PGAS, Chapel, irregular applications, compiler optimizations, inspector executor

1 INTRODUCTION

Irregular memory access patterns are commonly found in applications that include graph analytics [3], sparse linear algebra [7] and scientific computing [2]. Such access patterns pose significant challenges for both performance and user productivity on distributed-memory systems. Specifically, irregular memory access patterns exhibit weak spatial locality and lead to fine-grained remote communication. Furthermore, developing irregular applications is challenging because the access patterns are not known until runtime, forcing users to write code that determines where remote data is located. Such code can be error-prone and sensitive to data distribution choices.

However, the Chapel programming language [1] offers productivity advantages for writing codes with irregular memory accesses to distributed data. At the language level, Chapel provides high-level constructs for shared- and distributed-memory parallelization as well as data distribution support. Additionally, Chapel's runtime performs implicit remote communication, allowing users to write code that accesses remote data in an array in the same manner as accessing local data. Together, these features allow users to write distributed irregular applications in Chapel that closely resemble a shared-memory implementation.

Unfortunately, these user productivity advantages naturally lead programmers to write code that causes fine-grained remote communication. When developing traditional MPI-based applications, programmers often initiate collective communication operations before a compute kernel to gather all remote data so it can be used locally in the kernel. However, Chapel allows users to access remote data "on-demand" without gathering it locally beforehand via implicit one-sided communication calls (i.e., puts and gets). This becomes a problem for irregular memory accesses because the remote accesses can be sparse and spread out across the array, leaving little chance for remote caching or automatic aggregation. Therefore, while Chapel provides high productivity for irregular applications, it can cause significant performance issues. This is especially true for users who are unfamiliar with parallel programming and high performance computing (HPC).

In our prior work [4, 5] we have demonstrated that dramatic performance gains are possible for irregular applications in Chapel through manually applied optimizations. One such optimization utilizes the *inspector-executor* technique [6] to selectively replicate data that is remotely accessed in `forall` loops that contain indirect accesses like **A[B[i]]**. An example of such code is shown in Listing 1, where the indirect access is `x[col_idx[k]]` on line 4. The *inspector* performs memory access analysis at runtime to determine remote communication to **A** and the *executor* runs the original `forall` loop but redirects remote accesses to replicated copies to avoid repeated communication. However, such optimizations have been applied by hand to the Chapel code, which clearly does not improve user productivity.

```
1 forall row in Rows {
2   var accum : real = 0;
3   for k in row.offsets {
4     accum += values[k] * x[col_idx[k]];
5   }
6   b[row.id] = accum;
7 }
```

Listing 1: Sparse Matrix-Vector Multiply (SpMV) kernel

In this talk, we will discuss our current efforts on automating selective data replication by implementing it as a compiler optimization. With compiler support, the user is not required to change their original code in order to achieve significant performance gains. Our compiler optimization uses static analysis to identify candidate `forall` loops for the optimization, and then applies code transformations to construct the inspector and executor routines. We will show that, with our optimization, performance can be improved for three irregular applications by as much as 52x on a Cray XC system with a low-latency interconnect and 364x on a standard Linux cluster with an Infiniband interconnect.

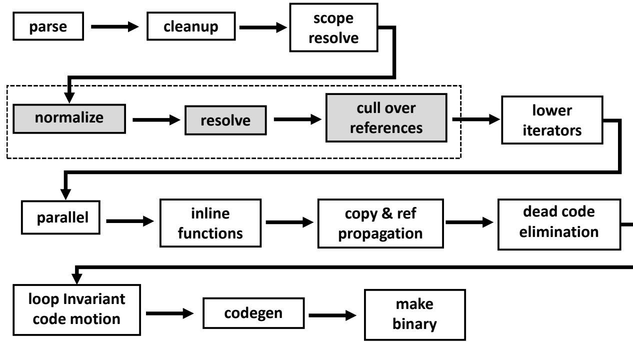

Figure 1: High-level overview of the Chapel compiler. There are roughly 40 passes in total, but most are omitted to simplify the diagram. The shaded passes within the dotted box represent the passes where our optimization performs static analysis and code transformations.

2 COMPILER OPTIMIZATION

Figure 1 presents a high-level overview of the Chapel compiler passes. We note that the compiler currently has roughly 40 passes, but most are left out of Figure 1 for simplicity. Our optimization operates entirely on Chapel's intermediate representation of the program, which we will refer to as the abstract syntax tree (AST). The optimization performs a majority of its code transformations during the normalize pass, where the AST has not yet been heavily modified/optimized, so it is easier to reason about.

The optimization starts by considering each `forall` loop that is present in the program. For each `forall` loop, our analysis looks for array accesses of the form **A[B[i]]**. While not always indicative of an irregular memory access pattern, such accesses are commonly found in irregular applications. However, array accesses do not have their own syntax in Chapel, meaning that **A[B[i]]** is represented as a call expression whose argument is another call expression. In other words, **A(B(i))**, where **A** and **B** are functions, looks identical to **A[B[i]]** within the AST. It is not until the resolve pass in the compilation process that our optimization will be able to determine that both **A** and **B** are arrays. To address this issue, a potential irregular access candidate is replaced with a compiler primitive that will be acted on during the resolve pass, at which point our optimization can perform further static analysis. If a `forall` loop has a candidate irregular access, the optimization continues by cloning the `forall` into the inspector and executor loops. During this process, the compiler inserts calls to Chapel procedures that we have written to perform various tasks related to selective data replication, such as setting up internal data structures, logging remote accesses, etc.

During the resolve pass of compilation, the candidate accesses found by the optimization are recognized since they were replaced by special primitives. It is at this point that our optimization can employ static analysis to ensure the validity of the transformations. These static checks include ensuring that **A** and **B** are arrays, **A** is a distributed array that supports the `localSubdomain` query and the `forall` loop being optimized is enclosed in at least one outer `for` loop. If anything is found to invalidate the optimization, then the code transformations performed previously will be ignored and removed via dead code elimination, leaving the original `forall` loop in its place. Otherwise, the primitive that represents **A[B[i]]** is replaced with a library call to either "inspect" or "execute" the access, depending on whether the primitive is in the inspector or executor loop.

Finally, after cull-over references, the optimization performs the remaining static analysis necessary to ensure the transformations are valid. The cull-over references pass resolves the *intents* of function arguments, where an argument's intent refers to whether it is passed in by value, reference, etc. The analysis performed at this stage includes detecting modifications (writes) to the relevant arrays and domains for the irregular accesses in the loop. It is important to track writes to the arrays/domains, as such changes could alter the memory access pattern **A[B[i]]**, and therefore require the inspector analysis to be rerun. The optimization waits to perform these checks until after cull-over references because it is significantly easier to detect writes to arrays/domains at this stage since their intents have been resolved. Writes to arrays/domains are performed via Chapel procedure calls where the array/domain is passed in as an argument. The compiler statically sets the intent of the argument depending on whether the procedure reads or writes to the array/domain (i.e., `ref` for writes, `const ref` for reads). Therefore, if the compiler detects that the intent of an array/domain is `ref`, then it can conclude that the array/domain is modified in the call.

3 PERFORMANCE EVALUATION

To demonstrate the performance of our compiler optimization, we will present an evaluation of three irregular applications across two different distributed-memory systems. Our goal is to show that the runtime performance of the original program can be significantly improved via our automatic optimization without requiring the user to modify the program. The three applications we evaluate are implemented in a high-level manner that adheres to Chapel's design philosophy, which is to separate data distribution details from the algorithm design. In this way, they are written with minimal programmer effort and prioritize code simplicity over performance. Our goal is to show that the runtime performance of the original program can be significantly improved via our automatic optimization without requiring the user to modify the program. Therefore, users can take advantage of the productivity benefits that Chapel provides while also achieving good performance.

In this talk, we will present results from a Conjugate Gradient (CG) benchmark, an implementation of the PageRank graph algorithm and an implementation of the Breadth First Search (BFS) graph algorithm. Each application presents unique challenges regarding the code structure, which stresses our optimization's static analysis, as well as having varying performance characteristics (i.e., input data set, number of iterations until convergence, etc.). Our preliminary results show runtime speed-ups as large as 52x on a Cray XC system and 364x on a standard Linux Infiniband cluster.

4 FUTURE WORK

For future work, we plan to improve upon our compiler optimization framework and address some of its limitations. We also plan to design and implement additional compiler optimizations that leverage our framework. These optimizations will serve as alternatives

to selective data replication when replication cannot be applied for correctness or performance reasons. Such optimizations include software prefetching for Chapel's remote cache and automatic aggregation that is more general than what is provided by Chapel today.

ACKNOWLEDGEMENTS

We would like to thank Vass Litvinov, Michelle Strout, Brad Chamberlain, Elliot Ronaghan and Engin Kayraklioglu from the Chapel team for providing guidance on working with the Chapel compiler, as well as providing access to the Cray XC system that was used in our performance evaluations.

REFERENCES

[1] Bradford L Chamberlain, Elliot Ronaghan, Ben Albrecht, Lydia Duncan, Michael Ferguson, Ben Harshbarger, David Iten, David Keaton, Vassily Litvinov, Preston Sahabu, et al. 2018. Chapel comes of age: Making scalable programming productive. *Cray User Group* (2018). https://cug.org/proceedings/cug2018_proceedings/includes/files/pap130s2-file1.pdf
[2] Jack Dongarra, Michael A Heroux, and Piotr Luszczek. 2016. High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems. *The International Journal of High Performance Computing Applications* 30, 1 (2016), 3–10.
[3] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. 2007. Challenges in parallel graph processing. *Parallel Processing Letters* 17, 01 (2007), 5–20.
[4] Thomas B Rolinger, Joseph Craft, Christopher D Krieger, and Alan Sussman. 2021. Towards High Productivity and Performance for Irregular Applications in Chapel. *2020 IEEE/ACM 4th Annual Parallel Applications Workshop: Alternatives To MPI+X (PAW-ATM)* (2021).
[5] Thomas B Rolinger, Christopher D Krieger, and Alan Sussman. 2021. Runtime Optimizations for Irregular Applications in Chapel. *The 8th Annual Chapel Implementers and Users Workshop (CHIUW)* (2021).
[6] Joel H. Saltz, Ravi Mirchandaney, and Kay Crowley. 1991. Run-Time Parallelization and Scheduling of Loops. *IEEE Trans. Comput.* 40, 5 (May 1991), 603–612.
[7] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. 2007. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *SC'07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. IEEE, 1–12.