Removing Temporary Arrays in Arkouda

Ben McDonald
Hewlett Packard Enterprise
USA
ben.mcdonald@hpe.com

ABSTRACT

This talk discusses experimental modifications made to the Arkouda (a NumPy-like Python package with a Chapel backend server) messaging layer to pass several operations together as a block of Lisp code to be parsed on the server in one message, as opposed to the existing model of each command being evaluated individually, requiring multiple messages to evaluate compound expressions. These modifications were made to eliminate the need for extra temporary array creation when executing compound operations in Arkouda. To improve the performance of the implementation, the initial code, which parsed the Lisp code once per-task, was optimized to parse only once per message and remove dynamic allocations. The implementation is evaluated by comparing it against current Arkouda performance. The results of the comparison show that the Lisp interpreter is not yet outperforming standard Arkouda code, but additional functionality can be supported through this new feature.

1 INTRODUCTION

Arkouda[1] is a NumPy-like Python package designed for data scientists to interactively use supercomputers and is implemented with a Chapel backend server. The building block of the Arkouda server is the Chapel block distributed array, and most Arkouda operations are performed in parallel across these arrays. When performing compound operations, such as 'a * x + y', Arkouda will pass two messages to the server: the first to execute 'a * x', returning a temporary array of the result, and the second to calculate the result of the operation added to 'y' (i.e., 'tmp + y').

This need to create temporary arrays and pass multiple messages to the server to evaluate compound operations in Arkouda creates additional messaging and memory overheads. To solve this problem, an Arkouda Lisp interpreter was implemented, turning compound expressions into a block of Lisp code to be parsed on the server, allowing these operations to be evaluated in a single message with no temporary arrays. The current performance of Lisp interpreted expressions in Arkouda is slightly behind standard Arkouda operations but removes the temporary array, reducing memory overhead, and adds support for new features, such as filter operations.

2 LISP INTERPRETER IMPLEMENTATION

Lisp[2] is a programming language with a simple syntax and a parenthesized, prefix notation. Lisp was chosen as the language for the task of combining multiple Arkouda server messages into one because of the relative ease of parsing Lisp expressions.

The initial implementation would take the compound expression to be executed from the Python client, convert it to an AST and then construct a Lisp expression as a string to be passed to the server. Once the Lisp string reaches the Arkouda backend server, the expression is parsed and executed, returning the result of the expression back to the client.

Given that interpreters were not part of the initial design scope of the Chapel language, the language features of Chapel made the implementation process go very smoothly. Chapels model for object-oriented programming, having both records and classes with multiple management types, allowed adequate flexibility for the data structures required. The biggest challenge in the implementation came from Chapels static typing system, which is not unique to Chapel, but requires an additional level of abstraction of data structures to allow multiple different types to be supported in the lisp interpreter.

3 BENEFITS OF LISP INTERPRETER

Operations between large arrays in Arkouda will cause an error if the addition of the temporary array exceeds the amount of available memory, whereas the Lisp interpreter will perform the operation in-place. This means that much larger operations can be evaluated by using the Lisp interpreter as opposed to regular Arkouda operations.

With Arkoudas current design, computing the expression 'a * x + y' will require up to 4 times the amount of memory of the largest array in the expression (1 for each array, 1 for the temporary array created by Arkouda), while the lisp interpreter will only at most require 3 times the amount of memory of the largest array.

Additionally, the Lisp interpreter adds support in Arkouda for "filter" operations, such as '((y+1) if (x < v) else (y-1))', where an expression is applied to each element of an array based on a conditional.

Finally, the Lisp interpreter allows users of Arkouda to define purely client-side functions of arbitrary code that will be executed on the server as a single message. Today, the support is limited to binary operations, but could be extended to support additional Python operators through addition to the server-side Lisp parser.

4 OPTIMIZING THE CHAPEL CODE

The initial Lisp interpreter implementation was significantly behind the performance of standard Arkouda operations due to

creating class instances, which are heap-allocated in Chapel, for each token in the Lisp expression for every task. This means that the expression 'a * x + y' would result in 5 dynamic allocations per element of the array with the initial implementation, as there are 5 symbols in the expression.

To remove the overhead of these allocations, a "memory pool" was implemented, which allows different tasks to share allocations. Instead of allocating every time a symbol is parsed, the memory pool is queried to see if there are any existing objects that can be used, if so, the existing allocation is used, if not, a new object is allocated, which is then returned back to the memory pool once it goes out of scope so that it can be reused by other tasks, instead of allocating a new object. This optimization led to around a 60% performance improvement, but still significantly underperformed in comparison to standard Arkouda operations.

The next optimization was a change in the implementation to parse the Lisp expression only once per message. The initial implementation parsed the Lisp expression for each task, meaning that each task was parsing the same piece of code and repeating work that may have already been accomplished by another task. To fix this problem, the Lisp expression is parsed only once prior to starting evaluation, passing each task a fully parsed version of the Lisp expression. This optimization led to a further ~25% performance improvement.

5 PERFORMANCE RESULTS

Table 1 shows the Arkouda performance of evaluating the operation 'a * x + y'[3], a simple compound expression where 'a' and 'x' are arrays storing 64-bit 'real' values and y is a 64-bit 'real' scalar value on a single locale of a Cray CS. Through the algorithmic optimizations discussed in section 3, the Lisp interpreter performance improved ~6x when operating on 100,000 element arrays and ~50x on 10,000,000 element arrays. The optimized code is ~2x behind the standard Arkouda operations, but it is believed there is further room for optimization. For explanations of "Memory pool" and "Single parse", please refer to section 3.

Version	100,000 Element	1,000,000 Element	10,000,000 Element
Arkouda	0.82 GiB/s	4.35 GiB/s	31.46 GiB/s
Initial Lisp	0.13 GiB/s	0.21 GiB/s	0.29 GiB/s
Memory pool	0.56 GiB/s	1 GiB/s	1.13 GiB/s
Single parse	0.79 GiB/s	3.82 GiB/s	15.45 GiB/s

Table 1: Execution performance on a Cray CS on a single locale

6 NEXT STEPS

The Lisp interpreter code currently resides in the "Arkouda contrib" repository [5], so can be pulled into an existing Arkouda server through the modular build mechanism supported in Arkouda[4]. Prior to merging it into the main repository, additional performance improvements need to be implemented.

Today, the Lisp interpreter does all evaluation on locale 0, even in a distributed setting, meaning that there will be a large amount of communication when operating on distributed arrays and the code does not scale as more nodes are added. Resolving this issue so that each locale will operate on local data, rather than migrating all data to locale 0 will need to be completed before merging into the Arkouda main repository.

Furthermore, the implementation currently supports binary operators, comparison operators, and conditional statements, but is capable of supporting a much wider set of features, such as sorting or grouping.

Finally, through performance experiments, the overhead of the Lisp interpreter preventing it from competing with regular Arkouda operations is coming from storing expression tokens as abstract class values and casting them to concrete tokens to evaluate the expression. These tokens have to be stored as abstract class values and then casted to concrete types since types in Chapel must be known at compilation time. This means, to support multiple different types, the lisp interpreter tokens must be cast to concrete values for each value in the array, operations that are about 5 times more expensive than a binary addition.

By simplifying the supported features and removing the class-based tokens while supporting only a subset of the functionality, the Lisp interpreter has been shown to outperform regular Arkouda operations. Based on these results, further investigation into how the overhead of the class-based tokens could be removed would be needed.

7 CONCLUSION

This work highlights a work in progress that aims to reduce the memory overhead and improve performance of Arkouda compound operations. The current implementation reduces the memory footprint of compound operations by eliminating temporary arrays but is behind the performance of standard Arkouda compound operation evaluation. Additionally, this work enables some new functionality in Arkouda, such as filter operations over Arkouda arrays. The problem of Arkouda memory usage and multiple message passing is still being worked on and next steps have been identified.

ACKNOWELDGEMENT

The concept of the Arkouda lisp interpreter was conceived by Mike Merrill, who started the implementation presented in this paper and co-created Arkouda. This work would not have been possible without the contributions of Mike Merrill.

REFERENCES

[1] https://github.com/bears-r-us/arkouda
[2] https://lisp-lang.org/
[3] https://github.com/Bears-R-Us/arkouda-contrib/blob/main/aklisp/test/lisp-stream.py
[4] https://bears-r-us.github.io/arkouda/setup/MODULAR.html
[5] https://github.com/Bears-R-Us/arkouda-contrib/tree/main/aklisp