Locality-Based Optimizations in the Chapel Compiler

Engin Kayrakliogu
Hewlett Packard Enterprise
engin@hpe.com

Elliot Ronaghan
Hewlett Packard Enterprise
elliot.ronaghan@hpe.com

ABSTRACT

In the recent releases, we have added two locality-based optimizations to the Chapel compiler. These optimizations enable the compiler to statically determine the locality of array accesses and aggregate fine-grained copy operations. In this talk, we summarize how they are implemented, their impact on various programming idioms, associated performance improvements and pertinent future directions.

KEYWORDS

parallel programming, compiler optimizations, productivity

1 INTRODUCTION

Chapel is a parallel programming language that supports partitioned global address space (PGAS) memory model. The PGAS model allows programmers to use a single address space, which improves productivity by making all available system memory accessible by every locale without explicit communication. Chapel combines the PGAS memory model with other high-level concepts such as distributed arrays and data parallel distributed loops to create an expressive programming language. However, this paradigm is prone to writing code with poor performance and scalability because of implicit communication.

On the other hand, common programming idioms represented by Chapel's first-class, high-level language concepts also enable the compiler to make automatic optimizations that would be impossible in low-level paradigms such as message passing. This talk focuses on two such optimizations that significantly mitigate common performance overheads with no programmer effort.

Both optimizations that this talk covers are in the Chapel 1.24 release, and can be readily used by the programmers.

2 AUTOMATIC LOCAL ACCESS

Accesses to Chapel arrays are implemented with a method named `this` on the array type that is automatically called by the compiler. A simplified implementation of `this` for a distributed array type is shown in Listing 1.

```
1 proc this(idx) {
2   if isLocalIndex(idx) then
3     return localAccess(idx);
4   else then
5     return nonLocalAccess(idx);
6 }
```

Listing 1: A Simplified Implementation of Distributed Array Access

Note that, in line 2, the implementation checks whether `idx` is local, because if that is the case, the local data can be accessed in a much faster manner. However, this check itself has some small but noticeable overhead. Consider a STREAM-Triad implementation in Chapel that uses indexed access into distributed arrays, as shown in Listing 2.

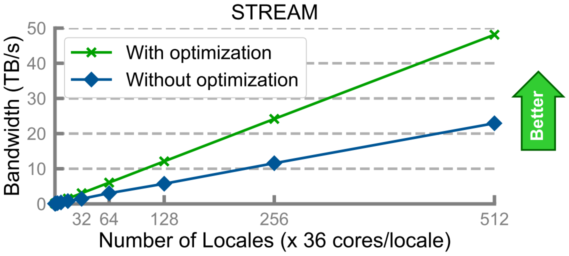

Figure 1: STREAM Bandwidth

```
use BlockDist;

var D = newBlockDom(1..n);
var A: [D] int,
    B: [D] int,
    C: [D] int;

forall i in A.domain do
  A[i] = B[i] + alpha * C[i];
```

Listing 2: STREAM kernel with indexed access

In this snippet, the three distributed arrays are accessed by index in the `forall` loop body, and they would normally incur the locality checks as discussed above. However, these checks can be avoided because:

- The `forall` loop will distribute the work in the same way the loop domain (`A.domain`) is distributed
- All arrays are distributed the same way as the loop domain is distributed
- All three distributed arrays are accessed at the `i`th index, which is the loop index

Starting in Chapel 1.23, the compiler is able to make this analysis in cases like the above. Moreover, it can also transform the code to do a once-per-loop dynamic check to use `localAccess` automatically if only a subset of the requirements can be proven at compile time.

Figure 1 shows how this optimization improves STREAM performance, where the kernel is implemented similarly to the one in Listing 2, With this optimization, indexed STREAM performs about twice as fast, reaching the limits of the system. This performance is virtually identical to other idioms that do not use indexed access into distributed arrays.

3 AUTOMATIC COPY AGGREGATION

Another common overhead in languages with a PGAS memory model occurs due to fine-grained communication. In some cases where the fine-grained access is predictable, caching and/or prefetching the remote data can help mitigate some of these overheads. However, especially in cases where the remote data is accessed randomly, such approaches are generally not very impactful. A solution for these scenarios is aggregating the communication and transferring data in bulk with fewer messages.

Listing 3 shows a simplified version of the `index_gather` kernel from the bale effort [1].

```
var cycArr = newCyclicArr(...);
var blockArr = newBlockArr(...);

fillRandom(blockArr);

var tmp: [blockArr.domain] int;

forall i in blockArr.domain do
  tmp[i] = cycArr[blockArr[i]];
```

Listing 3: Simplified Sketch of the index_gather Kernel

The `forall` loop iterates over a block-distributed domain, while copying data from a cyclic-distributed array into a block-distributed one. This element-wise, random-access copy operation causes fine-grained communication. However, this operation can be done in an aggregated fashion because:

- `temp[i]` (and `blockArr[i]`) are local accesses because the `forall` is over the same domain as theirs. Furthermore, this will be recognized as such by the automatic local access optimization that was discussed above,
- Individual copy operations that will execute at each iteration of the, loop can be reordered without impacting the application behavior.

Starting in Chapel 1.24, the compiler is able to perform this analysis and use aggregation in these scenarios. This optimization relies heavily on other compiler capabilities and module-level optimizations developed before. The aggregation is facilitated through module-level aggregation objects that were designed for this study and have been used in Arkouda [2] in similar scenarios. Therefore, the required AST transformation for this optimization is relatively small. On the other hand, the Chapel compiler already had some analysis to check for safety of unordered execution in similar cases that enabled unordered forall optimization. Automatic aggregation relies on that analysis to make aggregation decisions.

Figure 2 shows that without any optimization, this benchmark does not scale (light blue). Unordered forall optimization, firing automatically with no user effort, improves performance by enabling out-of-order communication (medium blue). Finally, manual aggregation (dark blue) and automatic aggregation (solid green) perform very similarly and much better than the other versions, where the latter does not require any user effort, at all.

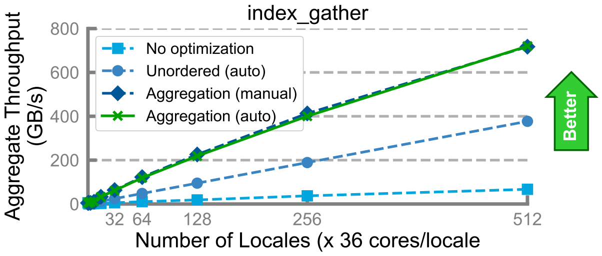

Figure 2: Bale index_gather

4 CONCLUSION

This work covers automatic local access and automatic aggregation optimizations that were implemented in the Chapel compiler in the recent releases. The talk gives a brief overview on how they are implemented and how Chapel's high-level features enable them. It demonstrates different idioms where such optimizations do and do not fire. Finally, it concludes by discussing potential future improvements.

ACKNOWLEDGEMENT

The authors would like to thank Michael Ferguson, who has implemented some of the key abilities in the Chapel compiler that this optimization is built on.

REFERENCES

[1] https://github.com/jdevinney/bale
[2] https://github.com/mhmerrill/arkouda