# Locality-Based Optimizations in the Chapel Compiler

Engin Kayraklioglu(✉), Elliot Ronaghan, Michael P. Ferguson, and Bradford L. Chamberlain

Hewlett Packard Enterprise, Seattle, USA
{engin,elliot.ronaghan,michael.ferguson,blc}@hpe.com

**Abstract.** One of the main challenges of distributed memory programming is achieving efficient access to data. Low-level programming paradigms such as MPI and SHMEM require programmers to explicitly move data between compute nodes, which typically results in good execution performance at the expense of programmer productivity. High-level paradigms such as the Chapel programming language aim to reduce programming difficulty by supporting a global memory view. However, implicit communication afforded by the global memory view can make it easier for the programmers to overlook performance considerations. In this paper, we show that Chapel's high-level abstractions such as data-parallel loops and distributed arrays that enable easier programming can also enable powerful compiler analyses and optimizations, which can mitigate these overheads. We demonstrate two compiler optimizations added to the Chapel compiler in versions 1.23 and 1.24. These optimizations rely on the use of data-parallel loops and distributed arrays to strength-reduce accesses to global memory and aggregate remote accesses. We test these optimizations with STREAM-Triad and index_gather benchmarks and show that they result in around 2x performance improvements on a Cray XC supercomputer. Furthermore, we analyze two real-world applications, chplUltra and Arkouda, that use manual remedies to address the overheads addressed by these optimizations. We observe that more than half of the places in the source code where these remedies are applied can benefit from optimizations without any programmer effort.

**Keywords:** Parallel programming · Compiler optimizations · Productivity

## 1 Introduction

Chapel is a parallel programming language that supports the partitioned global address space (PGAS) memory model. The PGAS model allows programmers to use a single namespace, which improves productivity by making all variables in the lexical scope accessible without explicit communication. Moreover, unlike other PGAS languages, Chapel's execution model is not SPMD by default. This implies that the variables in a given namespace refer to a single address in the


X. Li and S. Chandrasekaran (Eds.): LCPC 2021, LNCS 13181, pp. 3–17, 2022.
https://doi.org/10.1007/978-3-030-99372-6_1

global memory rather than different ones in each processing element. 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.

Chapel's approach to distributed memory programming empowers several real-world applications. Chapel Multiphysics Software (CHAMPS) [18] is a CFD simulation library used for aircraft design and simulation and has close to 50 thousands lines of Chapel code. Arkouda [1] is a data-science-oriented Python library that is backed by a server implemented in Chapel for distributed memory programming. Arkouda has around 15 thousands lines of Chapel code. chplUltra [17] is an astrophysics software used for simulating the dynamics of ultralight dark matter and it consists of around 10 thousands lines of code.

On the other hand, developers using the PGAS model and high-level abstractions are prone to writing code with poor performance and scalability because of implicit communication. We show that common programming idioms supported by Chapel's high-level language concepts enable the compiler to perform automatic optimizations that would be impossible in low-level approaches such as message passing. Moreover, automatic optimizations based on high-level constructs tend to be portable as lower-level details are typically handled by the language runtime and communication middleware. This paper presents two such optimizations that significantly mitigate common performance overheads with no programmer effort. Specifically, our contributions are:

- design and implementation of an optimization where accesses to distributed arrays are made faster by avoiding locality checks in data-parallel loops
- design and implementation of an optimization that aggregates fine grained accesses in copy operations in data-parallel loops,
- experimental demonstration of performance improvements of these optimizations, and a discussion on their impact on real-world applications,
- discussion on how these optimizations and the Chapel compiler can be improved in general.

The rest of the paper is organized as follows. Section 2 gives a background on related Chapel concepts. Section 3 describes the two optimizations in detail. Section 4 shows some experimental and anecdotal results. Section 5 proposes future directions for the Chapel compiler and the optimizations presented here. Section 6 summarizes some related studies in the literature, and Sect. 7 concludes the paper.

## 2 Chapel Background

Our focus in this paper is on Chapel's high-level, data-parallel concepts. In this section, we give a short background on distributed arrays and `forall` loops in Chapel since both of these are key concepts for this work. For a more complete introduction to Chapel refer to [5].

### 2.1 Distributed Arrays

Chapel decouples an array's distribution from its data thanks to its *domain* concept. Domains are index sets that can describe how the indexed data should be mapped to the system memory. All Chapel arrays have domains. Listing 1 shows how a domain can be declared, and how it can be used to declare an integer array[1].

```
// a local, 1-based, m-by-n domain (index set)
var myDomain = {1..m, 1..n};

// an integer array declared over that domain
var myArray: [myDomain] int;
```

**Listing 1.** Declaring non-distributed domains and arrays in Chapel

To create a distributed array, one needs only to declare the domain as distributed by using a standard or a user-defined distribution [7]. Listing 2 shows how a block-distributed domain and array can be created in Chapel. Note that the array declaration is identical to that in Listing 1.

```
use BlockDist;
var myDomain = {1..m, 1..n} dmapped Block(...);
var myArray: [myDomain] real;
```

**Listing 2.** Declaring distributed domains and arrays in Chapel

Listing 3 shows some of the most common ways Chapel arrays can be accessed and manipulated. These include but not limited to; whole-array operations, iteration over their elements, and indexed accesses.

```
// using promoted or whole-array operations
myArray = 1.1;

// using sequential iteration over its elements
for elem in myArray do
  elem = 2.2;

// using indexing (with sequential iteration over its domain)
for idx in myDomain do
  myArray[idx] = 3.3;
```

**Listing 3.** Common ways of accessing a Chapel array serially

### 2.2 Forall Loops

Chapel has several kinds of loops in order to support different parallel programming patterns. One such loop is the `forall` loop. A `forall` loop can parallelize

[1] There are shorter syntactic alternatives for creating arrays without an explicit domain declaration, such as `var A: [1..n] int;`. Nonetheless, all Chapel arrays have domains.

and/or distribute the iteration across the system depending on the iterand that drives it. For example, a `forall` loop over a non-distributed domain or array would typically use all of the cores on the local compute node to implement the loop; whereas one over a distributed domain or array would use all of the cores on all of the compute nodes over which the array is distributed. Use of `forall`s in conjunction with distributed arrays and domains guarantees that loop iterations are distributed similarly to the data that it is iterating over. This observation is key in implementing locality-based optimizations in the compiler.

Listing 4 shows the `forall` version of the two loops previously shown in Listing 3.

```
// using parallel/distributed iteration over its elements
forall elem in myArray do
  elem = 2.2;

// using indexing (with parallel/distributed iteration over its domain)
forall idx in myDomain do
  myArray[idx] = 3.3;
```

**Listing 4.** `forall` Loops Over Domains and Arrays

Note that the only syntactical difference from the loops shown in Listing 3 is the use of keyword `forall` instead of `for`.

## 3 Compiler Analysis and Optimizations

In this section, we first describe the automatic local access optimization that analyzes `forall` loops to determine local array accesses, and avoids dynamic locality checks for those accesses. This optimization is implemented in Chapel version 1.23 and it is on-by-default. Second, we summarize the automatic aggregation optimization that aggregates communication in the last statements in `forall` loop bodies. This optimization is added to the Chapel compiler in version 1.24, and can be enabled with the `--auto-aggregation` flag.

### 3.1 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 5.

```
proc this(idx) {
  if isLocalIndex(idx) then
    return localAccess(idx);
  else
    return nonLocalAccess(idx);
}
```

**Listing 5.** A simplified implementation of distributed array access

Note that, in line 2, the implementation checks whether `idx` is local, because if it is, the array element can be accessed in a faster manner. However, this check itself has some small but noticeable overhead. The overhead is exacerbated if arrays are accessed in a tight inner loop—as is typically the case for conditionals inside such loops. Consider a STREAM-Triad [23] implementation in Chapel that uses indexed access into distributed arrays, as shown in Listing 6.

```
1 use BlockDist;
2 var Dom = {1..n} dmapped Block(....);
3 var A, B, C: [Dom] int;
4
5 forall i in Dom do
6   A[i] = B[i] + alpha * C[i];
```

**Listing 6.** STREAM-Triad 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 are provably unnecessary because:

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

The automatic local access optimization implemented in the Chapel compiler uses similar reasoning to improve the performance of local accesses to distributed arrays.

**Finding Candidate Expressions for Optimization.** Early in compilation, array accesses are simply call expressions that are indistinguishable from procedure calls[2]. On the other hand, by the time call expressions are resolved, and array accesses are differentiated, Chapel's AST is transformed significantly enough to make some of this analysis difficult. Therefore, during earlier compilation passes, we analyze and transform the AST, replacing all call expressions that are candidate for optimizations with a special compiler primitive. Listing 7 sketches a simplified version of how this initial analysis and call replacement works.

First, we iterate over all `forall` loops in the program. For each, we try to find the loop domain. A `forall` can iterate over a domain, which directly becomes the domain of the loop; or it can iterate over a domain query on an array (e.g. `myArray.domain`), in which case we try to find the array's declaration and deduce the domain from the declaration. If neither, we continue the analysis and try to optimize using dynamic checks (details are below).

[2] In Chapel, postfix parentheses and square brackets can be used interchangeably as long as the opening and closing delimiter is the same.

```
void findCandidates(loop) {
  loopDom = findDomain(loop) // can return NULL
  for call in loop.body.calls()
    if (call.localityDominator == loop &&
        call.arguments == loop.indices)
      maybeArr = call.base
      arrDom = findDomain(maybeArr) // can return NULL
      static = ( loopDom != NULL &&
                 arrDom != NULL &&
                 loopDom == arrDom )
      if static
        loop.staticCandidates.insert(call)
      else
        loop.dynamicCandidates.insert(call)
}
```

**Listing 7.** Pseudocode for candidate discovery

Then, for every call inside the loop body which has the same argument(s) as the loop index(es), we assume that the called expression is an array symbol and try to find its domain. If we can find symbols representing the loop's domain and the array's domain and they are the same symbol, we say that this access is a *static candidate* for automatic local access optimization. If we couldn't find the domain for the loop and/or the array, this call is a *dynamic candidate* for automatic local access optimization. We add the call to the appropriate list of candidates.

**Transforming AST For Static and Dynamic Checks.** After finding candidates for the optimization, we transform the AST for the loop to add static and dynamic checks. Static checks are necessary because the initial analysis and transformation happens before type resolution. Therefore, we add static checks for both static and dynamic candidates, and they only check whether what we assumed to be an array symbol is actually an array symbol (as opposed to a procedure symbol) and the domain type supports this optimization. A requirement for supporting this optimization is that the domain distributes indices in the same way as it distributes a parallel iteration over itself. All the standard domain maps in Chapel support this optimization, but a user-defined domain map could be imagined where this is not the case. To provide a general solution, we expect domain maps to provide a function that returns a boolean at compile time that informs the compiler as to whether the domain map supports this optimization or not.

On the other hand, dynamic checks are added for cases where the relationship between the array and the loop domains cannot be established statically. This also supports cases where a `forall` only traverses a slice of an array's domain.

For a scenario where there is one static and one dynamic candidate, as in Listing 8, we create AST equivalent to that shown in Listing 9.

```
var dom1 = {1..n} dmapped Block (...);
var dom2 = {1..m} dmapped Block (...);
var arr1: [dom1] int, arr2: [dom2] int;

forall i in dom1 do
  arr1[i] = arr2[i];
```

**Listing 8.** A case where `arr1` and `arr2` are static and dynamic candidates

```
// check all candidates statically:
if (staticCheck(arr1, loopDomain) &&
    staticCheck(arr2, loopDomain)) then

  // check dynamic candidates at execution time
  if (dynamicCheck(arr2, loopDomain)) then
    forall i in dom1 do
      arr1.maybeLocal[i] = arr2.maybeLocal[i];
  else
    forall i in dom1 do
      arr1.maybeLocal[i] = arr2[i];
else
  forall i in dom1 do
    arr1[i] = arr2[i];
```

**Listing 9.** Sketch of the generated AST for the snippet in Listing 8

The generated AST first does static checks on arrays that are optimization candidates. These checks are simple functions that return compile-time (in Chapel terminology, they are `params`) booleans. If all the candidates pass static checks, we dynamically check the dynamic candidates, as well. The first `forall` clone is where all static and dynamic candidates pass their checks, where the second is for the case for successful static, and failed dynamic checks. The final clone is identical to the user's loop, and does not have any optimizations.

**Finalizing the Optimization.** After the initial transformation is done, the generated AST is resolved more or less normally. Static checks are computed at compile time, and the conditional based on the static checks is folded. While resolving this AST, the only special case for this optimization is for resolving the `maybeLocal` calls. First, the compiler tries to resolve them as regular array accesses. If it can, it replaces them with a call to `localAccess` which avoids locality checks. If the compiler cannot resolve them as array accesses, they will be reverted to regular calls, and will be attempted to be resolved as such.

### 3.2 Automatic Aggregation

Another common overhead in PGAS languages 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 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 10 shows a simplified version of the *index_gather* kernel from the bale effort [2].

```
1 var cycArr = newCyclicArr(...);
2 var blockArr = newBlockArr(...);
3
4 fillRandom(blockArr);
5
6 var tmp: [blockArr.domain] int;
7
8 forall i in blockArr.domain do
9   tmp[i] = cycArr[blockArr[i]];
```

**Listing 10.** 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. In a straightforward implementation, this element-wise, random-access copy operation causes fine-grained communication. However, this operation can be done in an aggregated fashion because:

- `tmp[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 in the previous section,
- Because `forall` is a parallel loop, individual copy operations that will execute at each iteration of the loop can be reordered without impacting the application behavior.

The automatic aggregation optimization implemented in the Chapel compiler will use reasoning along these lines in order to apply aggregation to optimize communication performance.

**Locality Detection.** Currently, automatic aggregation is supported only if the operation is a simple copy operation where one side is local and the other is not. To detect whether either side is local, we use the same approach and code as presented for automatic local access. In fact, there's a single analysis pass that collects enough locality information for both optimizations that are presented in this paper.

**Avoiding Data Hazards.** The aggregated copy operation requires order independence—that is, that the iterations of the optimized loop can run in any order including in parallel. In the context of the Chapel language, the `forall` loop implies that the loop body has this property. In addition, the aggregated

copy operation only optimizes the last statements of loop bodies, because it implies that nothing in the loop body can depend on any writes that occur as a result of this statement. The Chapel compiler already had an optimization where such statements are executed in an unordered matter [8], and the automatic aggregation optimization uses the same analysis as the existing optimization.

**Module Support.** Aggregating communication requires allocating local buffers that can be used to store data temporarily before communicating and a mechanism to flush them as they fill up. Implementing this purely by compiler transformations is not very feasible. Instead, our optimization facilitates *Aggregator* objects that have been studied in Chapel before and have been heavily used in Arkouda, a data analytics software that is implemented in Chapel (server) and Python (client) [1].

Aggregators are module-level objects that represent per-task buffers that temporarily store data to be communicated along with their address. These objects are typically created as *task intent*. A loop using an Aggregator object typically uses a `with` clause to create one instance per task, as the following example shows:

```
1 forall i in myDomain with (var agg = new Aggregator(int)) {
2   ...
3   agg.copy(arr[i], data); // equivalent to 'arr[i] = data'
4 }
5
```

**Listing 11.** Example of manual aggregator usage

**Transformations.** The `forall` loop in the index_gather kernel as shown in Listing 10 is transformed into something akin to Listing 12 early in compilation.

```
1 forall i in blockArr.domain with (var agg = new Aggregator(int)) do
2   if dummyAggregationMarker {
3     tmp[i] = cycArr[blockArr[i]];
4   }
5   else {
6     agg.copy(tmp[i], cycArr[blockArr[i]]);
7   }
```

**Listing 12.** Simplified transformation for automatic aggregation

Once hazard detection and other relevant passes, such as loop invariant code motion, are complete, we choose one of those branches and eliminate the other one. Therefore, there are no runtime checks of any sorts. Note that removing the `else` block also entails cleaning up any aggregator creation because they would be useless.

## 4 Results

We evaluate the performance of these optimizations by using the STREAM-Triad [23] and index_gather [2] benchmarks that motivate them. We compare

the automatically-optimized execution time against their manually-optimized counterparts which were shown to perform comparably to reference MPI and SHMEM versions [9,10]. We also analyze the code for chplUltra [17] and Arkouda [1] to assess how the optimizations can improve them. We show that both of them cause straightforward implementations of benchmarks to perform similarly to manually-optimized versions. They also help avoid significant portion of the relevant manual optimizations in real-world applications.

We used a Cray XC30 supercomputer for the performance studies. Compute nodes are dual-socket and equipped with 36-core Broadwell CPUs clocked at 2.1 GHz. Nodes are connected with the Aries interconnect. Automatic local access comparisons were done against Chapel 1.23 pre-release, whereas automatic aggregation comparisons are against Chapel 1.24 pre-release[3], so that they capture the performance improvement introduced by the optimization on the release that they were implemented. We used the default configuration for all of these tests. The executables are compiled with `--fast` flag. In addition, the automatic aggregation tests are compiled with `--auto-aggregation`.

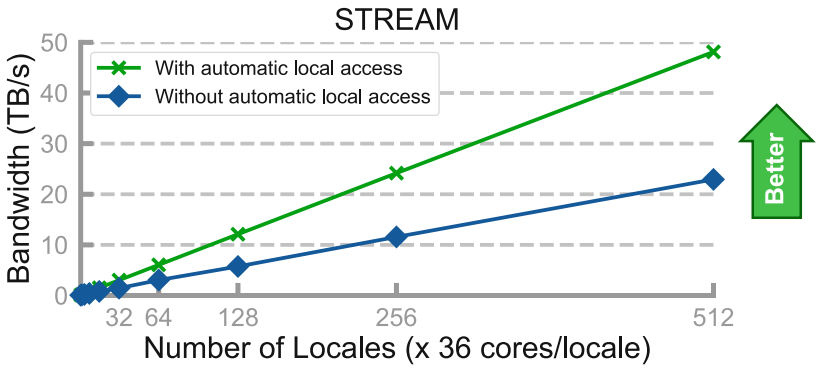


**Fig. 1.** STREAM-Triad bandwidth

**Automatic Local Access.** Figure 1 shows how this optimization improves STREAM-Triad performance. The kernel for this STREAM-Triad implementation is shown in Listing 13.

```
1 forall i in Dom do
2   A[i] = B[i] + alpha * C[i];
```

**Listing 13.** STREAM-Triad with indexed array access

[3] The most current Chapel release version is 1.24.1.

Without the automatic local access optimization, this kernel reaches only about half of system bandwidth (shown in dark blue with diamond markers), whereas other idioms for STREAM-Triad are able to reach the full system bandwidth. Other idioms are shown in Listings 14 and 15. The difference between the two types of idioms is that the distributed arrays are accessed by index in the first one, which causes overheads without the automatic local access optimization.

```
1 forall (a,b,c) in zip(A,B,C) do
2   a = b + alpha * c;
```

**Listing 14.** STREAM-Triad with zippered iteration over arrays

```
1 A = B + alpha * C;
```

**Listing 15.** STREAM-Triad with promoted expression

With this optimization, indexed STREAM-Triad 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.

In addition, we inspected the *chplUltra* [17] which relies on explicit use of `localAccess` for better performance. We have observed that, thanks to the automatic local access optimization, we can reduce the number of explicit calls to `localAccess` from 80 to 21, without sacrificing performance. The remaining explicit `localAccess` calls are either not within `forall` loops, or the index that they access is a function of the loop index.

**Automatic Aggregation.** Figure 2 shows that without any optimization, the index_gather benchmark, shown in Listing 10 does not scale (light blue, dashed line, square markers). The unordered forall optimization [8], 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.

To explore the impact of this optimization in user code, we analyzed Arkouda, the application for which user-level aggregators were implemented initially. Thanks to the automatic aggregation optimization, we were able to reduce the number of explicit aggregators from 61 to 22. The most common causes for the remaining 22 are identical to those limitations of the automatic local access optimization: (1) operation is not inside a `forall` or (2) the array access index is complicated. These two causes require 12 of the remaining 22 cases to use explicit aggregation. The remaining 10 require more investigation.

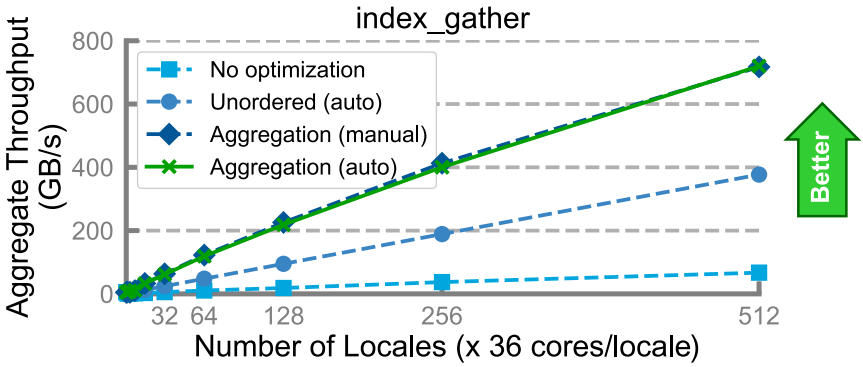


**Fig. 2.** Bale index_gather (Color figure online)

## 5 Future Work

We want to investigate extending the automatic local access optimization to handle array accesses where the index is an affine expression based on the loop index. Such accesses are common in linear algebra codes, and currently are not covered by this optimization. Furthermore, many loops in applications use `for` and `coforall`[4] loops and it would benefit these cases to extend the optimization beyond `forall` loops. These other loops do not have any guarantees about data locality, however, there are common idioms where they are used in a way that can benefit from this optimization.

As of Chapel 1.24, automatic aggregation is off-by-default and can be enabled by the `--auto-aggregation` flag. There are two main concerns for enabling it by default. First, the aggregator objects are not designed for handling all-local aggregation. This is because the initial use case for them was for the programmer to explicitly use them and with the assumption that they would use them only if they know for sure that there is communication. However, the compiler can automatically use aggregation in cases where both sides of a copy is actually local, even though some of the static analysis tries to prevent that. We observed that there can be around 2x slowdown in such cases. However, we believe that we can adjust the aggregator implementation to reduce the overhead in such cases. Second, aggregators use per-locale buffers on each Chapel task (typically a core). This poses issues when aggregators are used in systems with high locale and core counts. We would like to consider reducing the memory overhead of aggregators, potentially using multi-hop aggregation where some local aggregation takes place before communicating the data, thereby reducing its memory footprint.

The automatic aggregation optimization covers assignments that are the last statements in the loop bodies. This coverage can be expanded in two ways. First, we can support arbitrary operations to be aggregated. This would mean creating function pointers representing the operation and using that function to unpack

[4] A loop where each iteration is mapped to a parallel Chapel task.

the aggregated data instead of just copying them in local memory. Second, this optimization can cover all statements inside the loop body. This requires alias and dataflow analysis inside the loop body to avoid data dependences.

## 6 Related Work

A relatively early study on how high-level language constructs can enable compiler optimizations is done by Choi and Snyder [12]. The authors show that array operations like shifts can be efficiently optimized by the compiler, if the language enables expressing such operations using high-level constructs, such as operators. This work is based on ZPL [6], an array programming language. However, unlike Chapel, ZPL was not a general-purpose language. As such it did not support operations like array indexing.

Hayashi et al. [14] implemented several LLVM optimizations for Chapel programs to reduce costs associated with distributed memory programming. The authors focus on GET/PUT operations injected by the Chapel compiler and try to find ways in which they can be coalesced or eliminated. These optimizations achieve significant performance improvements. Currently, some of the optimizations presented in this work can be used with an experimental flag `--llvm-wide-opt`. These optimizations focus on communication calls that happen inside a lexical scope and do not consider calls that can be invoked repeatedly inside a loop.

Other distributed memory optimizations have been studied in the contexts of other PGAS languages with compilers. Chen et al. [11] describes strength-reduction, communication and computation overlap and message coalescing techniques in the Berkeley UPC compiler. We believe that the set of optimizations presented in this work are thematically similar to those that were studied by Hayashi et al.

Other studies pertaining Chapel's performance include but not limited to; runtime optimizations, such as caching [13], prefetching [16], inspector/executor optimizations [20], profile-based optimizations [15]; module optimizations, such as iteration reorganization [3], complex bulk transfer [21]; GPU-related explorations [4], and finally general performance studies in comparison with other programming models [22].

Single-node loop optimizations for improving data access performance are common. A significant portion of the literature focuses on loop and data layout transformations based on the polyhedral model for related optimizations such as auto-vectorization [24] and improved cache utilization [19]. We believe, similar techniques can be used in the Chapel compiler. However, they typically focus on affine array accesses in loops which are not in scope for this paper.

## 7 Conclusion

In this paper, we show that well-designed high-level language abstractions not only make programming easier, but can also express key information about the

application that can enable powerful compiler optimizations. To demonstrate this point, we present two optimizations added to the Chapel compiler in recent releases that have been used in production-level applications. The first optimization, automatic local access, reduces the costs of accessing local parts of a distributed array. The second optimization, automatic aggregation, gathers communication operations locally before communicating them in bulk. Both of these optimizations are enabled by high-level concepts like `forall` loops and distributed arrays. They both can increase performance without adding any programming burden in benchmarks and real-world applications alike.

**Acknowledgement.** We would like to thank Michelle Strout for reviewing an early draft and sharing very valuable insights that contributed to this paper's quality.

## References

1. Arkouda: NumPy-like arrays at massive scale backed by Chapel. https://github.com/Bears-R-Us/arkouda. Accessed 26 Jul 2021
2. Bale. https://github.com/jdevinney/bale. Accessed 26 Jul 2021
3. Bertolacci, I.J., et al.: Parameterized diamond tiling for stencil computations with chapel parallel iterators. In: Proceedings of the 29th ACM on International Conference on Supercomputing, ICS 2015, pp. 197–206. ACM (2015). ISBN 978-1-4503-3559-1. https://doi.org/10.1145/2751205.2751226
4. Carneiro, T., et al.: Towards Chapel-based Exascale Tree Search Algorithms: dealing with multiple GPU accelerators. In: HPCS 2020, p. 9 (2021)
5. Chamberlain, B.L.: Chapel, chap. 6. In: Balaji, P. (ed.) Programming Models for Parallel Computing, pp. 129–159. MIT Press (2015)
6. Chamberlain, B.L.: The design and implementation of a region based parallel programming language. University of Washington (2001)
7. Chamberlain, B.L., et al.: User-defined distributions and layouts in Chapel: philosophy and framework. In: Proceedings of the 2nd USENIX Conference on Hot Topics in Parallelism, HotPar 2010, p. 12. USENIX Association (2010)
8. Chapel 1.20 Release Notes: Benchmarks and Performance Optimizations. https://chapellang.org/releaseNotes/1.20/06-perf-opt.pdf. Accessed 26 Jul 2021
9. Chapel 1.23 Release Notes: Ongoing Efforts. https://chapel-lang.org/releaseNotes/1.23/05-ongoing.pdf. Accessed 26 Jul 2021
10. Chapel: Performance Highlights: STREAM Triad. https://chapel-lang.org/perf-stream.html. Accessed 26 Jul 2021
11. Chen, W.-Y., Iancu, C., Yelick, K.: Communication optimizations for fine-grained UPC applications. In: 14th International Conference on Parallel Architectures and Compilation Techniques, PACT 2005, pp. 267–278. IEEE (2005)
12. Choi, S.-E., Snyder, L.: Quantifying the effects of communication optimizations. In: Proceedings of the 1997 International Conference on Parallel Processing (Cat. No. 97TB100162), August 1997, pp. 218–222 (1997). https://doi.org/10.1109/ICPP.1997.622647
13. Ferguson, M.P., Buettner, D.: Caching puts and gets in a PGAS language runtime. In: 2015 9th International Conference on Partitioned Global Address Space Programming Models, September 2015, pp. 13–24 (2015). https://doi.org/10.1109/PGAS.2015.10

14. Hayashi, A., et al.: LLVM-based communication optimizations for PGAS programs. In: LLVM 2015, pp. 1–11. ACM Press (2015). ISBN 978-1-4503-4005-2. https://doi.org/10.1145/2833157.2833164
15. Kayraklioglu, E., Favry, E., El-Ghazawi, T.: A machine-learning-based framework for productive locality exploitation. IEEE Trans. Parallel Distrib. Syst. **32**(6), 1409–1424 (2021). https://doi.org/10.1109/TPDS.2021.3051348
16. Kayraklioglu, E., Ferguson, M.P., El-Ghazawi, T.: LAPPS: locality-aware productive prefetching support for PGAS. ACM Trans. Archit. Code Optim. **15**(3), 28:1-28:26 (2018). https://doi.org/10.1145/3233299
17. Padmanabhan, N., et al.: Simulating ultralight dark matter in Chapel. In: 2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), May 2020, pp. 678–678 (2020). https://doi.org/10.1109/IPDPSW50202.2020.00120
18. Parenteau, M., et al.: Development of parallel CFD applications with the Chapel programming language. In: AIAA Scitech 2021 Forum. American Institute of Aeronautics and Astronautics (2021). https://doi.org/10.2514/6.2021-0749
19. Patwardhan, A.A., Upadrasta, R.: PolyhedralModel guided automatic GPU cache exploitation framework. In: 2019 International Conference on High Performance Computing Simulation (HPCS), pp. 496–503 (2019). https://doi.org/10.1109/HPCS48598.2019.9188095
20. Rolinger, T.B., Krieger, C.D., Sussman, A.: Runtime optimizations for irregular applications in Chapel. https://chapel-lang.org/CHIUW/2021/Rolinger.pdf. Accessed 26 Jul 2021
21. Sanz, A., et al.: Global data re-allocation via communication aggregation in Chapel. In: 2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), October 2012, pp. 235–242 (2012). https://doi.org/10.1109/SBAC-PAD.2012.18
22. Slaughter, E., et al.: Task bench: a parameterized benchmark for evaluating parallel runtime performance. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE Press (2020). ISBN 9781728199986
23. STREAM Benchmark Reference Information. http://www.cs.virginia.edu/stream/ref.html. Accessed 26 Jul 2021
24. Trifunovic, K., et al.: Polyhedral-model guided loop-nest auto-vectorization. In: 2009 18th International Conference on Parallel Architectures and Compilation Techniques, pp. 327–337 (2009). https://doi.org/10.1109/PACT.2009.18