LAPPS: Locality-Aware Productive Prefetching Support for PGAS

ENGIN KAYRAKLIOGLU, The George Washington University, USA
MICHAEL P. FERGUSON, Cray Inc., USA
TAREK EL-GHAZAWI, The George Washington University, USA

Prefetching is a well-known technique to mitigate scalability challenges in the Partitioned Global Address Space (PGAS) model. It has been studied as either an automated compiler optimization or a manual programmer optimization. Using the PGAS locality awareness, we define a hybrid tradeoff. Specifically, we introduce locality-aware productive prefetching support for PGAS. Our novel, user-driven approach strikes a balance between the ease-of-use of compiler-based automated prefetching and the high performance of the laborious manual prefetching. Our prototype implementation in Chapel shows that significant scalability and performance improvements can be achieved with minimal effort in common applications.

CCS Concepts: • **Computing methodologies → Parallel programming languages**; **Distributed programming languages**;

Additional Key Words and Phrases: PGAS, Chapel, prefetching, runtime system

ACM Reference format:
Engin Kayraklioglu, Michael P. Ferguson, and Tarek El-Ghazawi. 2018. LAPPS: Locality-Aware Productive Prefetching Support for PGAS. *ACM Trans. Archit. Code Optim.* 15, 3, Article 28 (August 2018), 26 pages.
https://doi.org/10.1145/3233299

1 INTRODUCTION

The Partitioned Global Address Space (PGAS) model aims to reduce programming complexity in distributed memory architectures by providing a global view of the memory system. Unlike the shared memory and message passing models, the PGAS model carries locality awareness to the language system, where optimizations can be made without requiring significant programmer effort. Some examples of programming languages and libraries with a PGAS memory model are Chapel [12], UPC [20], UPC++ [43], Fortran 2008 [36], and OpenSHMEM [15].

PGAS languages can have performance overheads due to fine-grained communication. For remote accesses, PGAS languages rely on one-sided communication calls (i.e., GET and PUT), which result in many small messages on the interconnection network, causing significant overhead. Hence, it can be challenging to achieve scalability without programmer optimizations.

This is a new paper, not an extension of a conference paper.
Authors' addresses: E. Kayraklioglu, The George Washington University, 800 22nd Street NW, Washington, D.C. 20052; email: engin@gwu.edu; M. P. Ferguson, Cray Inc., 901 5th Avenue, Seattle, WA 98164; email: mferguson@cray.com; T. El-Ghazawi, The George Washington University, 800 22nd Street NW, Washington, D.C. 20052; email: tarek@gwu.edu.

1544-3566/2018/08-ART28 $15.00
https://doi.org/10.1145/3233299

Table 1. User Effort in Prefetching Tasks Using Alternative Approaches

Task	Manual	Automatic	*LAPPS*	Task Description
Decide	Yes	No	*Yes*	Determine what to prefetch
Initialize	Yes	No	*No*	Create local arrays; copy remote data
Access	Yes	No	*No*	Access local arrays as appropriate
Maintain	Yes	No	*No*	Propagate updates; reclaim memory

Typically, prefetching is used to reduce overheads associated with fine-grained access to high-latency memory. For example, some architectures support instructions that prefetch data in main memory to low-latency hardware caches. Such instructions generally can be exploited by compiler pragmas [2] or built-in compiler functions [3]. These constructs hint the compiler to issue prefetch instructions several iterations ahead, to ensure that data is in the hardware cache before it is needed. Prefetching is also used for remote databases and web services [35].

In HPC-oriented distributed memory architectures, however, data aggregation plays an important role while moving remote data closer to processing elements due to the significantly higher latency difference between local and remote memory. In PGAS context, for example, applications are generally optimized to prefetch the remote data in bulk before entering computationally intensive loops. Practically, this entails implementing data movements similar to message passing applications.

There are four major tasks that require effort while prefetching remote data: *Decide:* Determine whether data need to be prefetched depending on the expected/measured performance characteristic of the application implementation. *Initialize*: Compute the needed remote indices depending on factors such as access pattern, data size, data distribution, and number of processing elements and issuing appropriate communication. *Access*: Access the prefetched data, likely through temporary local arrays. *Maintain*: Propagate updates as necessary and/or reclaim the allocated memory. Although programmer knowledge pertinent to these tasks can make manual prefetching very effective in terms of performance and scalability, it can get tedious and increase the "time to solution" significantly, which PGAS promises to decrease. On the other end of the spectrum, automatic prefetching by static/dynamic analysis relieves programmers of the burden of prefetching. However, it can be very difficult to devise an analysis that does not incur runtime overheads, which is able to handle all the tasks of prefetching to achieve performance improvements in wide range of use cases. In practice, due to limited capabilities of automated approaches, programmers inescapably implement prefetching with data aggregation, which makes PGAS programming similar to message passing. As effective as this approach is, it reduces programmer productivity.

We present locality-aware productive prefetching support (LAPPS) for the PGAS model. LAPPS is a novel feature for PGAS languages that is a tradeoff between two approaches for remote data prefetching. With LAPPS, the programmer decides whether prefetch is necessary and uses appropriate functions to inform the language system. Details are handled by various layers of the locality-aware software stack afforded by the PGAS model. The runtime system manages the memory allocated for prefetched data and carries the burden of maintaining consistency. Internal language libraries handle accessing the prefetched data transparently. This way the programmer still decides what/when to prefetch, yet data movement, access and maintenance is handled by the PGAS software stack. We believe that this prefetching approach can fully exploit locality awareness in the language system without significant programmer effort. A summary of the programmer's involvement in different prefetch approaches is given in Table 1.

Listings 1–3 demonstrate the effort of using LAPPS and manual prefetching in matrix transpose. Listing 1 shows the base implementation inspired by Transpose kernel in the Parallel Research

```
1
2 for i in 1..numIter                {
3   forall (i, j) in B.domain do
4     B[i,j] = A[j,i];
5   forall a in A do a += 1.0;       }
```

Listing 1. Matrix Transpose Example.

```
1 A.transposePrefetch();
2 for i in 1..numIter                {
3   forall (i, j) in B.domain do
4     B[i,j] = A[j,i];
5   forall a in A do a += 1.0;       }
```

Listing 2. Matrix Transpose Example with LAPPS.

```
1  coforall l in Locales do on l {
2    var locIdxs = B.localSubdomain();
3    var tDom = {locIdxs.dim(2), locIdxs.dim(1)};
4    var localA: [tDom] real;
5    for i in 1..numIter {
6      localA = A[transposeDom];
7      forall (i, j) in bLocSubDom do
8        B[i,j] = localA[j,i];
9      forall (i, j) in A.localSubdomain() do
10       A[i,j] += 1.0;             }}
```

Listing 3. Matrix Tranpose with Manual Optimization.

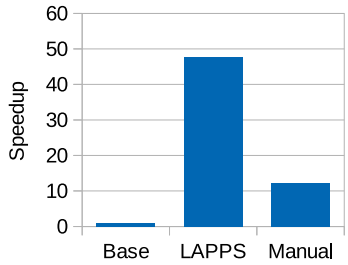


Fig. 1. Speedup with 32 nodes on infiniband.

Kernels (PRK) [41]. The kernel is run multiple times, and the input is updated between iterations. With LAPPS (Listing 2), the programmer decides to prefetch data and calls an appropriate method, which can be thought of as a high-level, blocking communication call that is handled internally by the software stack. In contrast, Listing 3 shows that the programmer needs to use more effort to implement a similar optimization. Specifically, lines 2–4 correspond to *Initialize*, line 6 corresponds to *Maintain*, and the modification in line 8 corresponds to *Access* tasks. In addition, this optimization uses a wider variety of language features, which requires more expertise. Figure 1 shows that in this example, LAPPS outperforms manual optimization and achieves 48× speedup over the base implementation.

We provide a prototype implementation of LAPPS in the Chapel software stack and analyze its performance. In more detail, this article contributes the following:

- —A design of standard library and runtime system coordination for remote data prefetching that leverages locality- awareness within the programming paradigm to move data efficiently.
- —A prototype implementation built on top of the Chapel runtime system and standard libraries.
- —A detailed performance and memory footprint analysis using two different interconnection networks and communication middleware with synthetic benchmarks and well-known application benchmarks including the PRK.
- —An analysis of the impact of our design on programmer productivity.

This article is organized as follows: Section 2 summarizes some of the related studies in literature in contrast to our work. Section 3 gives background information on Chapel with emphasis on relevant features. Section 4 provides an overview of our design. Section 5 describes how our design fits into the Chapel software stack and discusses a use case of our implementation using an example. Section 6 presents benchmark results, along with memory footprints and a discussion on programmer productivity. Section 7 concludes the article.

Table 2. Summary of the Most Related Works

Study	Optimization Techniques	Software Layer(s)	Usability Overhead	Performance Overhead	Application Specific
Alvanos [4]	Agg	C, R	None	Analysis	No
Barik [9]	StR	C	None	None	No
Chandra [14]	Loc	C	Trivial	None	No
Chen [16]	Agg, Ovl, StR	C	None	None	No
Choi [17]	Agg, Ovl, StR	C	None	None	No
El-Ghazawi [19]	Agg, Loc, Rep	A	Full	None	Yes
Haque [23]	Loc, Rep	A	Full	None	Yes
Hayashi [24]	Agg, Loc	C	None	None	No
Iancu [25]	Agg, Loc, Ovl	C, R	None	Analysis	No
Kayraklioglu [27]	Agg, Loc, Rep	A	Full	None	Yes
Müller [33]	Agg, Ovl	C	None	None	No
This study	*Agg, Loc, Rep*	*R, L, C*	*Trivial*	*None*	*No*

Abbreviations used for optimization techniques: *Agg:* aggregation; *Loc:* localization; *Ovl:* overlap; *Rep:* replication; *StR:* strength reduction. Abbreviations used for software layers: *C:* compiler; *R:* runtime; *L:* library; *A:* application.

2 RELATED WORK

With the PGAS model, it is challenging to achieve scalability without significant optimizations. Table 2 summarizes some of the most relevant studies in the literature proposing different techniques to improve the PGAS performance and scalability.

Static compiler optimizations have been studied widely as a way to mitigate scalability limitations. The most important benefit of relying on the compiler is to optimize performance with no overhead or programmer intervention. Barik et al. [9] studied array-of-structs to struct-of-arrays transformations and scalar replacement techniques in the X10 compiler. These techniques allowed the authors to reduce the message size entering asynchronous blocks. Chandra et al. [14] added local types to the X10 type system to allow declaring some variables as local. This approach is similar to our design as it relies on the programmer's *a priori* knowledge about the locality. Chen et al. [16] studied several compiler optimizations in UPC, including aggregation, overlapping and eliminating redundant communication through AST (abstract syntax tree) analysis. Hayashi et al. [24] used LLVM [29] to aggregate messages lexically close in the LLVM-IR. Although these papers report speedups without any user intervention, they do not leverage the locality awareness in the language system. In all of them, techniques are based on the analysis of individual GET and PUT calls. In contrast, our technique is based on the access pattern to individual distributed arrays.

KarHPFn, an HPF [28] compiler was used to implement a prefetching strategy using hardware capabilities [32, 33]. Authors implemented a compilation technique where parallel loops are transformed into a mix of prefetcher and accessor loops. This approach effectively hides access latency by overlapping communication and computation. They also demonstrate a vectorization technique that aggregates communication. First, however, their work is implemented for and tested only on Cray T3E, yet optimal prefetch distance is a function of architecture. Thus, how performance portability can be achieved with more modern architectures is unclear. Second, the approach is focused on the parallel `forall` construct in HPF, a rather synchronous language, where static analysis is arguably easier than those with asynchronous tasks (such as Chapel). Such highly automated techniques have not been adopted in mature PGAS languages conceivably due to these practical difficulties. In contrast, LAPPS purposefully leaves the decision-making to the programmer to

circumvent these difficulties, while handling the onerous tasks associated with data movement in an architecture-oblivious manner.

ZPL [11] has been used to demonstrate several compiler techniques where the language semantics can hint the compiler in terms of figuring out optimal data movements statically. The ZPL grammar and compiler was designed specifically for global array operations. Choi showed that techniques such as aggregation and overlapping can be implemented based on the high-level array operations specified by the programmer [17]. There are similar ideas to LAPPS as they show that if there are language constructs that allow more descriptive operations on arrays, communication optimizations can be handled by the language software stack to achieve scalability.

There are other studies where the compiler injects calls that helps the runtime system optimize [5, 25] the communication. Similarly, these studies do not leverage the access pattern and locality awareness. Furthermore, there is identifiable cost of on-the-fly analysis.

Similar techniques can also be applied as hand optimizations by the end programmer [19, 21, 27]. Coarfa et al. propose several optimizations for Co-array Fortran (CAF) [34] and UPC [18]. They show that both models can be very competitive with bulk communication optimization. However, for both languages it needed to be implemented by the programmer. The authors observe that this becomes tedious especially with strided access. In another example, Haque et al. show that manually creating local copies of the remote data can deliver orders of magnitude performance improvement in the Chapel implementation of CoMD [23]. Hand optimizations can deliver significant performance improvements as they rely on the programmers' awareness of data distribution and access patterns. However, they decrease the programmer productivity greatly. LAPPS is also based on programmer awareness; but aims to achieve similar or better performance with much less effort.

Improving locality by mapping tasks that communicate closer [7, 8], and caching remote data for potential reuse [22] are other directions which were studied for improving the performance of the PGAS memory model.

3 CHAPEL BACKGROUND

Chapel [12] is a parallel programming language built to support the PGAS memory model. In this section, we discuss locality management and distributed arrays in Chapel, which are some of the language features that are relevant to our work.

3.1 Distributed Arrays

Chapel supports distributed arrays through *domain maps*. Domain maps are library-level objects written in Chapel that describe the distribution of an array and provide the means to access and manipulate it [13]. Domain map classes must implement the Domain Map Standard Interface (DSI). Methods and fields required by the DSI are used by the compiler and the runtime system to implement Chapel's distributed array functions (e.g., access, slice, copy). A domain map consists of global and local descriptor objects. Distribution descriptors define how the unbounded index space is distributed across locales. Domain descriptors define which section of the index space belongs to which locale. Array descriptors define the actual data on the domain. Each global descriptor has an array of local descriptors allocated on distinct locales.

3.2 Locality Management

In Chapel, the processing unit that establishes data locality is called *locale*. For the context of this article, a locale is identical to a compute node. Chapel implements its PGAS memory view through *wide references*. A wide reference bundles locale ID with memory address. In contrast, a *narrow reference* is simply a memory address. The Chapel compiler generates wide references in

Table 3. Software Layers in Chapel and Their Roles Relevant to LAPPS

Layer	Role
Application	Application logic
Distribution libraries (domain maps)	Mapping distributed arrays to locales
Internal libraries	Base functionality, calls to runtime
Compiler	Creating wide references and communication calls
Runtime system	Tasking, communication, synchronization

cases where it cannot statically determine whether a variable is stored locally. Accesses to wide references are handled via runtime procedures, which access the data locally or remotely. The programmer can use the `local` statement to prevent the compiler from creating wide references conservatively. Another locality management concept is the on statement. This statement can be used to move the execution of a task from one locale to another. Chapel programmers can use on statements to bring tasks closer to the data they use.

3.3 Memory Consistency Model and Remote Data Cache

Chapel's memory consistency model is described in the language specifications [1]. Here, we summarize the guarantees the language system gives in terms of memory ordering and how the software cache [22] supports them. The specifications define *unordered memory operations*, an abstract relaxation that allows access reordering and data caching. Although these memory operations do not follow the program order strictly, reordering is limited by memory fences.

Ferguson and Buettner [22] design and implement a software cache in the Chapel runtime. When the cache is enabled, all memory operations are unordered. To establish correctness with the discussed memory consistency model, the authors rely on *acquire* and *release* fences, which can be considered as sequentially consistent atomic operations in the memory space as a whole. These fences have the following semantics. An acquire fence implies that *any* reads issued after the fence cannot be started before the fence. A release fence implies that *any* writes issued before the fence cannot be finished after the fence. Note that, by definition, these rules apply to unordered memory operations, as well. To enforce the memory consistency, the Chapel compiler injects acquire and release fences appropriately at the beginning and the end of parallel tasks. This guarantees adherence to the memory consistency model even with the software cache.

4 LAPPS: CONCEPTUAL OVERVIEW

To support productive and efficient user-driven prefetching, we design a per-node, write-through, dynamic prefetch buffer. We follow Chapel's multi-layer design (Table 3) for extensibility, and object orientation for usability. This section gives an overview of the important concepts in LAPPS.

System Architecture: LAPPS has multiple layers (Figure 2). The topmost layer extends the distributed array implementations. The most important entities in this layer are the *prefetch patterns*, which represent distinct access patterns and are used by the programmer to request prefetch. Also, at this layer, array accessors are modified to interact with the prefetch subsystem. As existing array accessors are modified to interact with the prefetch subsystem, the *Access* task is not the responsibility of the programmer. This allows efficient utilization of the global memory view. Finally, this layer also implements utility functions which are used by the lower levels of the prefetch subsystem to interact with the distributed array objects during prefetch initialization and update.

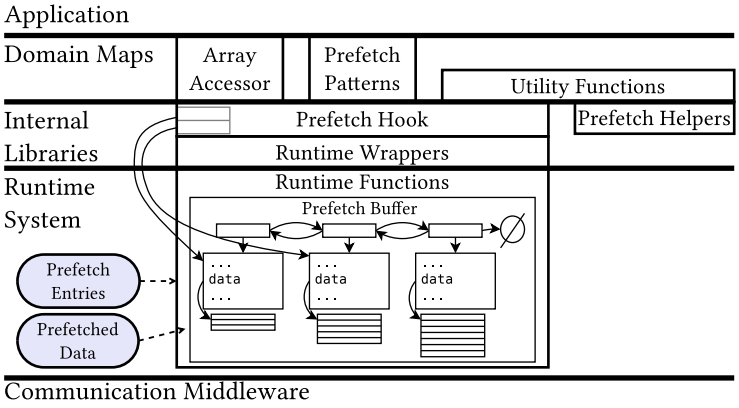


Fig. 2. System overview.

```
1
2
3  do{
4    diffuse(inGrid,outGrid);
5
6    del=max reduce abs(B-A);
7    step+=1;
8  } while(del>epsilon);
9
10
```

(a) Base

```
1  A.stencilPrefetch(true);
2  B.stencilPrefetch(true);
3  do{
4    diffuse(inGrid,outGrid);
5
6    del=max reduce abs(B-A);
7    step+=1;
8  } while(del>epsilon);
9  A.evictPrefetch();
10 B.evictPrefetch();
```

(b) Automatic Consistency

```
1  A.stencilPrefetch(false);
2  B.stencilPrefetch(false);
3  do{
4    diffuse(inGrid,outGrid);
5    outGrid.updatePrefetch();
6    del=max reduce abs(B-A);
7    step+=1;
8  } while(del>epsilon);
9  A.evictPrefetch();
10 B.evictPrefetch();
```

(c) Manual Consistency

Listing 4. Simple heat diffusion implementation along with different uses of LAPPS.

At the lower layer, internal language libraries implement *prefetch hooks*. Array objects creates one hook per locale which is used for interaction between the object and the runtime system components.

Finally, the runtime system is extended to support prefetch operations and manage the prefetched data. At this level, data is stored in a per-locale linked list of *prefetch entries*. An entry is added for every chunk of data brought in as a result of prefetch pattern executions. Therefore, every entry represents data coming from a single locale. Also note, this data structure is per-locale and not per-array, thus, it may store different arrays' prefetch data. Prefetch entries store bookkeeping information such as owner locale ID, consistency information, and locks that pertain to managing the data associated with it. Entries also store addresses of the local hook object that initiated the prefetch and vice versa. These links are used for interaction between the distributed array objects and the runtime system while updating the prefetch data.

Programming Interfaces: There are two programming interfaces: (1) a high-level interface implemented by the domain maps to be used at the application level; and (2) a low-level interface provided by the runtime system and the standard libraries that is intended to be used by the domain maps.

Important parts of the high-level interface are shown in Listing 4. It works between domain maps and the user application providing ways to request remote data prefetching (lines 1 and 2), as well as to update (line 5) and evict (lines 9 and 10) existing prefetch entries. The low-level

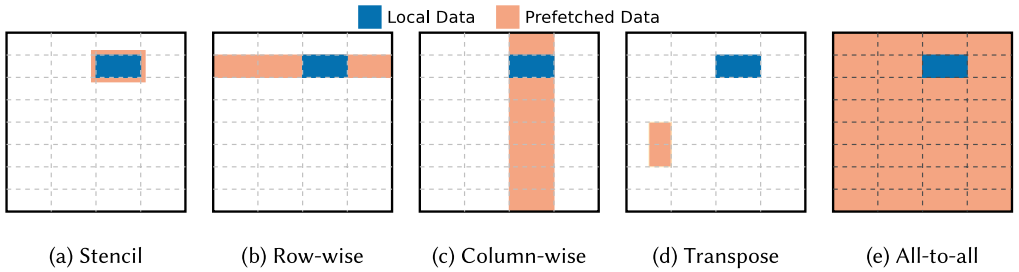


Fig. 3. Different prefetch patterns for a square array block-distributed across 32 Locales. Local and prefetched data are shown for one node.

interface is used by domain maps and discussed in detail in Section 5.2. It provides prefetch functions for domain maps such as request, access, update, and evict. These functions are provided by the prefetch hook objects. See Appendix A for full API reference.

Prefetch Patterns: Different applications have different prefetch requirements. Prefetch patterns represent different access patterns (Figure 3) and are implemented as methods of distributed array classes. As the array objects are locality aware, computing the parts of the index space that need to be prefetched from each locale is handled at the domain map level. As discussed before, domain maps use prefetch hooks in the low-level interface to interact with the runtime system and initiate a prefetch. The low-level prefetch request method in the prefetch hook allows the prefetching of data owned by another locale as a whole or a slice.

Data Movement Protocol: All communication induced by LAPPS is blocking and one-sided. The prefetch data is moved as a chunk in a single message. However, overall data movement is not completely monolithic and involves coordination across nodes. To initialize prefetch entries for arbitrary prefetch patterns, prefetcher nodes execute one or more on statements on the owner nodes. These on statements can cause the owner node to create a serial buffer to be able to move the data in a single message. Note, however, a node generally acts as both owner and prefetcher for a prefetch pattern. Therefore, even though a single thread may be blocked due to an incoming prefetch data, other threads within the node can act as an owner and handle outgoing prefetches. LAPPS does not provide any mechanism for managing outgoing prefetch request and relies on the runtime system and communication middleware capabilities to manage on statements. This communication protocol is exemplified in Section 5.2.

Memory Consistency: When multiple copies of the same data exist, they must be kept consistent. PGAS languages generally adopt relaxed memory consistency models with explicit/implicit synchronization points, which runtime systems leverage to maintain consistency [20, 22].

In order to support consistency, the runtime system must be recording synchronization events such as fences. The prefetch subsystem needs to use these event records to mark the creation of a prefetch entry and use this information for every access to determine if the data is still fresh or needs to be updated. This implies that not only creating prefetch entries, but also accessing them must involve runtime system calls. Note that prefetch entries are shared among intra-node threads which means that multiple threads in the same locale may be trying to access/update prefetch entries. Therefore, the runtime system also needs to protect the prefetch entries from data races.

Our design also allows manually consistent prefetching to avoid unnecessary synchronization if the programmer knows that prefetched data does not change in the owner node, or wants to

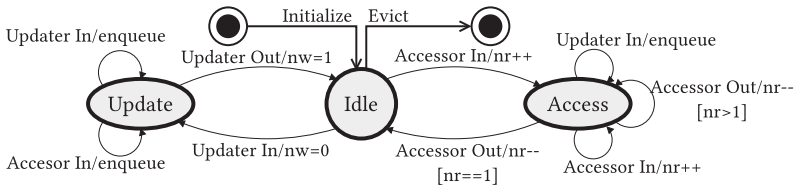


Fig. 4. State transition diagram for a prefetch entry.

control updates manually. Line 5 in Listing 4(c) shows an example of using manual consistency. The programmer uses the pattern-oblivious `updatePrefetch` method in the high-level interface (a method of distributed arrays) to update the data from the origin node. We believe manual consistency can be useful for applications which have too much synchronization that is irrelevant with respect to the changes to the prefetched data. Another benefit of manually consistent prefetching is that accesses to prefetched data can be satisfied without calls to runtime system.

Concurrency: Unlike access to prefetched data, local prefetch buffers are not thread-safe to reduce synchronization. Therefore, users cannot call prefetch patterns in parallel. For Chapel, in particular, this does not impact programmability as the execution is sequential outside explicitly parallel regions. Furthermore, prefetch patterns and low-level prefetch calls are blocking. After they return it is safe to call other prefetch patterns on other objects or use the data brought in by the pattern. On the other hand, prefetch patterns can create inter- and intra-node parallelism internally where nodes can request prefetch and serve requests by other nodes concurrently.

Once prefetch pattern returns, local threads can access (read/write) the local data in the prefetch buffer without explicit synchronization. However, the data needs to be protected against potential data races where a thread is updating the entry while others are accessing it. To do that, prefetch entries are protected with multiple accessors/single updater synchronization semantics. Figure 4 depicts the state transition of a prefetch entry. Note that this synchronization is only necessary if the user requests automatically consistent prefetching.

Memory Management: The data is stored and managed by the runtime system in a per-locale prefetch buffer. This buffer stores prefetch entries in a doubly linked list. A prefetch entry stores bookkeeping information for the chunks of data brought in from another locale along with the data itself. This bookkeeping information includes but is not limited to data size, synchronization constructs, and memory consistency records. Neither the buffer, nor the individual entries are of bounded size. Moreover, LAPPS does not maintain an internal memory pool for memory allocation and reclamation. We have considered limiting the memory used by the prefetch buffer. However, we chose not to for the following reasons. (1) Memory space requirement would depend on remote memory access patterns of specific applications, limiting the possible performance improvements due to management overhead. (2) In a typical optimized PGAS application where programmers implement bulk data movement, they are responsible for adhering to memory limitations. Based on these observations, LAPPS does not impose memory limitations, or incur overheads that would be caused by decision-making implied by such limitations. This also implies that the eviction policy in LAPPS is driven by the user. This functionality is provided by a simple pattern-oblivious `evictPrefetch` method in the high-level interface. We believe this design choice also improves performance predictability.

5 LAPPS: CHAPEL PROTOTYPE

Although LAPPS can be implemented in different programming paradigms that support the PGAS model, some Chapel features augment the applicability and efficiency. First, user-defined arrays

```
1 var descTable: [{0..#numLocales}] domain(2);
2 coforall l in Locales do on l { // populate the table in parallel
3   var myDomain = B.localSubdomain(); // indices that this locale owns
4   var tDom = {myDomain.dim(2), myDomain.dim(1)}; // indices needed
5   descTable[here.id] = tDom;   }  // store the indices (domain) in descTable
6 A.customPrefetch(descTable); // pass descTable to the custom prefetch pattern
7 forall (i,j) in B.domain do B[i,j] = A[j,i]; // basic transpose operation
```

Listing 5. Matrix Transposition with Custom Prefetch Pattern.

and multi-layered design minimized the effort of implementing LAPPS, as a significant part of the logic is implemented in the internal libraries rather than the runtime system. Second, multi-layer design allows future distributions added to the libraries or custom user-defined distributions [13] to benefit from LAPPS without modifications to internal libraries or the runtime system.

5.1 Software Stack Adaptations

Domain Maps: First, we added several prefetch patterns in Chapel's Block Distribution (Figure 3). We also implemented a custom pattern. Custom prefetch pattern can be used for more unique application needs and one example is shown in Listing 5 for matrix transposition. Custom prefetch pattern takes as an argument a 1D array of domains, where the *i*th index stores the indices required by the *i*th node (this is represented by Chapel domains, which we call *slice descriptors* in this context). Lines 1–5 in the listing create and populate such an array, where the prefetch pattern is called in line 6. In more complicated cases, lines 3 and 4 need to be adjusted for specific application needs.

Second, we implemented data serialization functionality using iterators. The data yielded by the iterators are used by the internal modules to create the serial buffers. Lastly, we added methods to domain maps that create deserialization containers and/or provide a means to access the serialized data. Deserialization containers for block distributed arrays are rectangular arrays, and serialized data access helpers are used to calculate the byte offset in the prefetch buffer of a given index. Both serialization and deserializaiton functionality are parts of utility functions that need to be implemented by the domain map implementer.

Standard Libraries: We added a module to the Chapel internal libraries that implements the `PrefetchHook` class, which provides an interface between domain maps and the runtime system. Domain maps request, update, evict, or access prefetch data using these hooks.

Runtime System: The runtime system provides the backbone for the prefetch mechanism. It creates prefetch entries and stores them in a doubly linked list. Along with the data and its size, a prefetch entry stores a wide reference to original data, consistency information (explained in more detail in Section 4), and, if applicable, a slice descriptor (set of indices required).

Supporting the memory consistency model is one of the important tasks of the runtime system. Recall from Section 3.3 that Chapel's relaxed memory consistency model [22] uses *acquire* and *release* memory fences as synchronization points. To support the consistency model, the runtime system perpetually counts acquired fences encountered. This counter is used by the prefetch buffer as a sequence number for the prefetch entries. When a new prefetch entry is created with auto-consistency, the current counter value is recorded as its sequence number. The sequence number is compared to the acquire counter to determine if the data is stale before each access. Acquire counters are thread-private. This implies that data might be stale for some threads and fresh for others. To protect the data against races, we use PThreads' reader-writer locks [31]. Reader-writer locks allow multiple readers to obtain a lock, but they only allow a single writer. Arguably, a lock-free

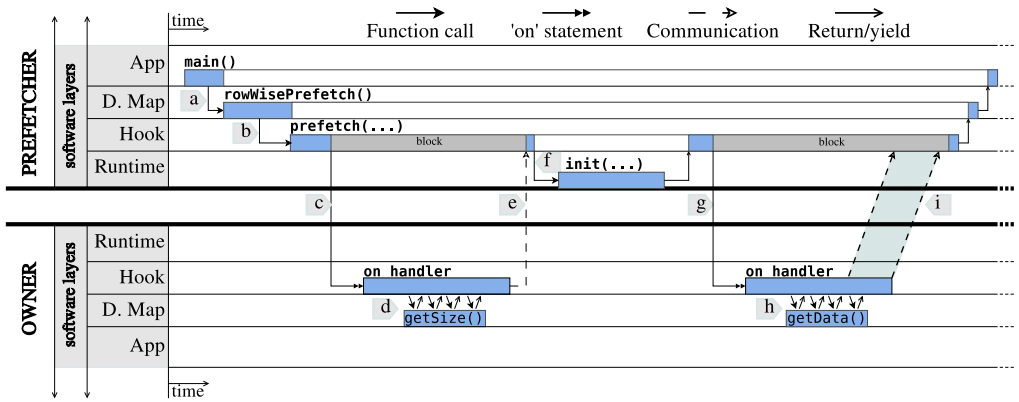


Fig. 5. Prefetch timeline for `rowWisePrefetch` as shown in Listing 6(b). Enumerated steps correspond to descriptions listed in Section 5.2.

```
                                A.rowWisePrefetch(true);    A.rowWisePrefetch(false);
                                B.colWisePrefetch(true);    B.colWisePrefetch(false);
forall (i,j) in C.domain {      forall (i,j) in C.domain {  forall (i,j) in C.domain {
 for k in 1..n do                for k in 1..n do            for k in 1..n do
  C[i,j]+=A[i,k]*B[k,j]; }        C[i,j]+=A[i,k]*B[k,j]; }    C[i,j]+=A[i,k]*B[k,j]; }
```

(a) Base (b) Automatic Consistency (c) Manual Consistency

Listing 6. Simple matrix multiplication implementation along with different uses of LAPPS.

approach or read-copy-update (RCU) [30] can help mitigate synchronization costs. Section 6.2.1 discusses the overhead of synchronization in detail.

In order to support consistency, accesses to prefetch references are satisfied through runtime system calls. Prefetch references created by the prefetch accessor bundle the address of the local copy of the data along with the entry that stores the data. This allows the runtime system to access necessary information about the prefetched data to provide consistency as explained above.

Compiler: We added a compiler primitive to generate prefetch references by bundling necessary information together. This primitive is used by the prefetch accessor to create a prefetch reference. During the code generation pass of the Chapel compiler, access to prefetch references are translated into GETs/PUTs similar to regular wide references.

5.2 An Example: Matrix Multiplication

In this section, we demonstrate how communication is handled in LAPPS. We use basic matrix multiplication as in Listing 6 for simplicity. `A`, `B`, and `C` are block-distributed arrays. The `forall` loop creates parallelism based on `C`'s distribution, therefore, all accesses to `C` are local. Listings 6(b) and 6(c) show how our design can be used to prefetch remote data with auto- and manual consistency. Figure 5 demonstrates the timeline of events starting from the invocation of `A.rowWisePrefetch()`:

(a) The application calls `rowWisePrefetch()`, which is a prefetch pattern (high-level interface).
(b) `rowWisePrefetch` uses the hook (low-level interface) to initialize prefetch. Its simplified implementation is given in Listing 7. `targetLocales` is a 2D array of locales that stores

```
1 coforall prefetcherIdx in targetLocales.domain do
2   on targetLocales[prefetcherIdx] do
3     for i in targetLocales.dim(2) {
4       const sourceIdx = (prefetcherIdx[1], i);
5       hook[prefetcherIdx].prefetch(prefetcherIdx, sourceIdx,
6                                    consistent, staticDomain);    }
```

Listing 7. Simplified `rowWisePrefetch`. `hook` is a part of low-level interface.

the locales across which the array is distributed. Lines 1 and 2 start tasks on each of these locales. `prefetcherIdx` is a tuple that denotes the location of each prefetcher in `targetLocales`. Line 3 iterates over the columns of the `targetLocales` and in line 4 the index variable of this loop is used to construct another tuple that is the source locale index. Finally, line 5 calls the `prefetch` method of the hook.

(c) Within the `prefetch` method, the hook issues `on` statements on the owners to get metadata. At a minimum, the metadata include the size of the data to be prefetched.
(d) The owner locale uses utility functions to determine the size of the serial buffer.
(e) The prefetcher locale receives the necessary metadata to initialize a prefetch entry.
(f) The prefetcher calls a runtime system function to initialize a prefetch entry.
(g) Upon initializing the prefetch entry and allocating space to store the data, the prefetcher node issues the second `on` statement that will transmit the actual data.
(h) The owner locale uses utility functions to create a serial buffer.
(i) Buffer is transferred to owner locale via a PUT call.

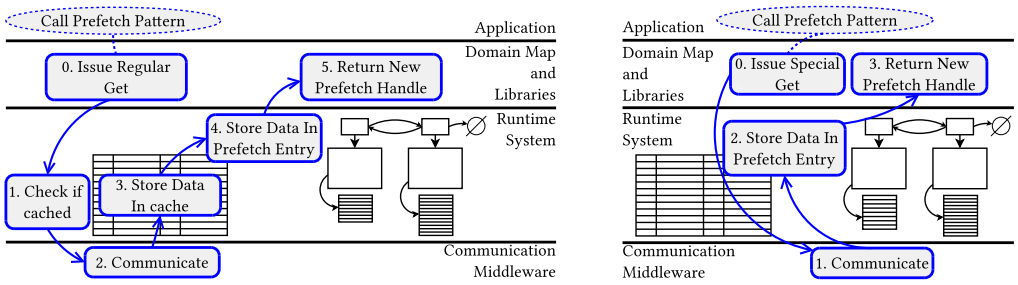
The utility functions are trivial and not shown to save space. In the above scenario, the prefetch pattern and the utility functions must be implemented by the domain map implementers. The rest of the implementation is oblivious to the data distribution and the access pattern.

5.3 Optimizations

We designed the prefetch support to be flexible and work with any data type and use case. However, there are common use cases which allow leveraging some optimizations in the Chapel implementation. In this section, we describe two such optimization opportunities, as follows.

5.3.1 Deferred Prefetching with Auto-Consistency. In the most common use cases for LAPPS, the pattern is called before entering a distributed loop, which implies an acquire fence [22]. This fence causes one redundant update on the prefetched data: Data is brought to the prefetcher node by the prefetch pattern method, then upon entering the loop it is invalidated by the acquire fence (see "Memory Consistency" in Section 4), and in the first access it is updated. To prevent this redundancy in auto-consistent prefetching, we do not copy remote data while initializing a prefetch entry, and only allocate space for it and make sure that it is stale the first time it is read. This optimization defers steps (g)–(i) in Section 5.2 if the data is prefetched with auto-consistency support.

5.3.2 Passive Owner If Domain Is Static. In LAPPS, updating a prefetch entry requires running two `on` statements (similar to (c) and (g) in Section 5.2). This allows consistency even in cases where an array is reallocated (e.g., grown or shrunken in dense case, or indices added in sparse case). However, such cases are not very common. Moreover, issuing `on` statements invokes handlers on the owner node causing significant disruptions in load balancing. To avoid this, we added support to keep the owner passive while a prefetcher is updating an entry. In order to enable this optimization, the programmer needs to guarantee that the domain of the array does not change. This optimization can be enabled by an optional flag in prefetch patterns. If this flag is set, the



(a) Regular GET (b) Special GET

Fig. 6. Flow of Events During Regular and Special GETs With Co-existing Software Cache (on the left in each figure) and Prefetch Buffer (on the right in each figure).

prefetcher node only queries the size and the starting address of the remote data during initialization that causes an on statement. Bringing the actual remote data during initialization and update is done via GETs from the prefetcher node (possibly a strided GET [37], if a slice of data prefetched). This update exchanges steps (g)–(i) in Section 5.2 with a GET from the prefetcher node. However, it requires that owner node also sends the address of the beginning of the requested data in step (e).

5.4 Prefetch Buffer and Software Cache

The Chapel runtime system has a per-core, software-managed, write-back remote data cache, which supports lookahead prefetching if it can detect a sequential access pattern.

Although remote data caching and user-driven prefetching are intertwined concepts, there is a key difference that affected our design choices. If the prefetch is user-driven, data needs to be locally persistent until the user instructs otherwise in order to mimic the behavior of manually creating local copies of the data at the application level. In contrast, the runtime system caches and evicts data without any input from the user. Thus, we implemented LAPPS as a separate system.

Following from our observation, we designed the prefetch buffers to be dynamically allocated, whereas the software cache is fixed size and allocated during runtime system startup. To increase the efficiency of the co-existing software cache and prefetch buffer, we added special GET and PUT functions in the runtime system that avoid the software cache. These functions are called by the prefetch hooks and are satisfied via communication without involving the cache. Data movement of a regular GET and the special GET we introduced are sketched in Figure 6. The impact of the special GET is twofold: (1) During initialization, there is no check to see if the data is cached. In common implementation idioms, it is unlikely that the data that is requested by the user is cached. Therefore, this check is redundant. (2) The data is not stored redundantly in the cache upon arrival. This avoids polluting the software cache that can store useful non-prefetched remote data.

6 EXPERIMENTAL ANALYSIS AND DISCUSSION

6.1 Testbed and Environment

In our analysis we used two different architectures: (1) an Infiniband QDR cluster equipped with dual-socket quad-core AMD Opterons clocked at 2.2GHz with 2MB last level cache (LLC) and (2) a Cray XE6 with Gemini interconnect. Compute nodes have dual AMD Magny Cours processors with 6MB LLC. Chapel locales are mapped to compute nodes and all tests use a thread per core.

We use Chapel pre-release version 1.16.0 with commit SHA `c87e36c`. We used QThreads [38] as the tasking layer. Infiniband tests are run with GASNet [10] with `ibv` conduit as the

Table 4. Prefetched Data Access Types

	Auto Consistent	Manually Consistent	Manually Consistent Within `local`
Serialized	5	6	7
Deserialized	8	9	10

communication layer. Gemini tests are run with Chapel's uGNI layer. These are the default settings for the architectures. Benchmarks are compiled with the `--fast` and `--cache-remote` flags.[1] Namely, compiler optimizations and the software cache are enabled in all tests.

6.2 Synthetic Benchmarks

We implemented the following synthetic benchmarks to measure specific overheads in our design.

6.2.1 Access. It is essential to quantify the access overhead to understand the performance of LAPPS. We compared access latency to prefetched data against four existing access types in Chapel: access to (1) a local non-distributed array, the local part of a distributed array from inside `local` block, (3) the local part of a distributed array, and (4) the remote part of a distributed array. Type 1 is different than the others as it is satisfied through Chapel's non-distributed arrays, whereas the others are accesses to block-distributed arrays, which include additional locality checks. Type 2 is different than type 3, as it is an access to a narrow pointer, that is done via dereferencing (Section 3.2). In contrast, type 3 requires a GET as it is an access to a wide pointer, even though it is a local access. Similarly, there are six different access types to prefetched data that are enumerated in Table 4.

We manually instrumented the C code generated by the Chapel compiler to take two measurements per access type: (1) *address computation*: time it takes to generate a narrow or wide pointer and (2) *access*: accessing the pointer via dereferencing or a GET. For each access type, we measured the per-access latency with varying strides between accesses.

The results of this benchmark are shown in Figure 7. As expected, remote access is slower than the others and the gap widens as stride gets larger. We attribute this to higher miss rates in the remote data cache. Among local accesses, types 3 and 2 are slower than type 1 by 25% and 15%, respectively. However, the difference is not as significant with longer strides. We believe this is due to the increased hardware cache miss rate starting to impact type 1 as stride increases, whereas others suffer from cache misses even with smaller strides due to access through the block-distributed array.

We observe that among prefetched data accesses, accesses to auto-consistent data (5 and 8) are 30–50% slower than manually consistent ones (6 and 9). This difference is due to checks for staleness and intra-node synchronization. We also note that accessing manually consistent data within a `local` block is up to 15% faster, due to avoiding the creation of wide pointers and GETs. We did not observe any significant difference between access to serialized (5–7) and deserialized data (8–10).

[1] In our discussions with the Chapel team, we were notified of the experimental `-suseBulkTransferDist` flag, which is intended to improve the performance of manually copying remote data. We used this flag in tests but noticed that only PRK-Transpose (Section 6.3) showed benefit, whereas in other tests the flag did not show any effect. Moreover, in some tests we observed that this flag caused occasional freezes. Nevertheless, we report PRK-Transpose results with this flag enabled as it significantly improved the performance of hand optimization. We believe this flag demonstrates the potential of hand optimization and should be used once it is more robust.

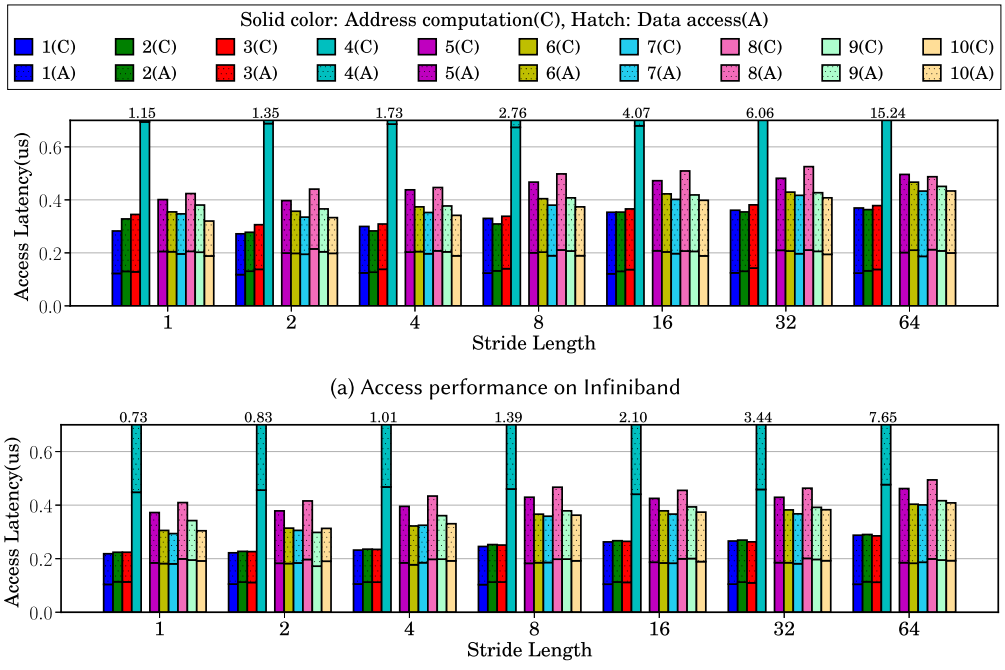


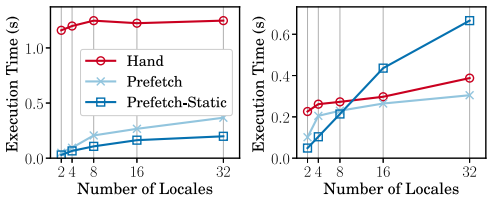
Fig. 7. Access performance synthetic benchmark results.

As expected, we see that accesses to prefetched data are significantly faster than accessing remote data. Moreover, access to prefetched data with lower spatial locality does not affect its performance as significantly as it does remote accesses. On the other hand, access to prefetched data is considerably slower than accessing local data. This is because of the way the block-distributed array accessor is implemented. This accessor checks if the data is local, and does local access if it is. Then, it checks if it is prefetched and does prefetch access if it is. Otherwise, it does a non-local access.

Between the two systems, (1) remote access is faster on Gemini. Access latency is 1.5× lower on Gemini with consecutive access. The difference exceeds 2× with strides of length 64. (2) There is no significant difference between prefetched data access latencies. (3) Local accesses are also faster on Gemini. This difference is not as pronounced as remote access and ranges between 10% and 15%.

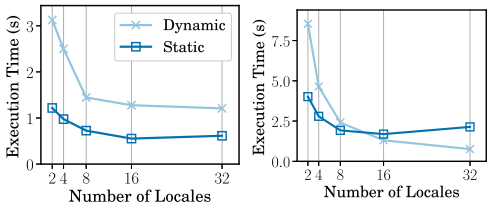
6.2.2 Initialization. Another overhead associated with prefetch is the initialization of the prefetch entry and initial copy of the data. To assess this overhead in comparison to hand optimization, we implemented a synthetic benchmark where a 2,048×2,048 block-distributed array is replicated on all locales using an all-to-all prefetch pattern, and an application level approach using whole array assignment. We compare the performance of the two with a varying number of locales.

The architectures we tested showed drastically different behaviors as shown in Figure 8. On Infiniband, prefetching is 6–10× faster than manually copying the data, whereas on Gemini the difference is not as significant. Furthermore, static domain optimization does not scale on Gemini as it does on Infiniband (see Section 6.2.3 for a discussion on the differences of two machines).



(a) Infiniband (b) Gemini

Fig. 8. Initialization performance.



(a) Infiniband (b) Gemini

Fig. 9. Update performance.

In further tests (not shown), we observed that distributed array copying in hand optimization has fundamental limitations and implementation deficiencies causing more GET calls than LAPPS.

6.2.3 Update. Updating a prefetch entry can be costly. To quantify this cost, we implemented a synthetic benchmark where prefetched data is accessed repeatedly by multiple locales. After every access, a barrier causes prefetched data to go stale. Hence, in every iteration data is updated. We used an all-to-all prefetch pattern on a block-distributed array of double precision floating point numbers. The array is 2,048×2,048 and 100 predetermined random remote indices are accessed by each locale. The results are shown in Figure 9. Similar to the initialization benchmark, the behavior is different on the two machines. Updating prefetch entries with static domains performs significantly better on Infiniband than it does on Gemini. We note that in terms of network communication the difference between dynamic and static domain prefetching amounts to executing a blocking on statement followed by a PUT and doing a GET. Also, recall that we use a GASNet communication layer on Infiniband, and uGNI on Gemini, which is a likely cause of this difference.

6.2.4 Performance Degradation. We identified two cases where our implementation can cause performance degradation. First, our implementation adds a check to the critical path of remote access in distributed arrays. This check can increase remote access latency for non-prefetched data. To assess the extent of this degradation, we used the benchmark in Section 6.2.1 to compare type 4 (remote) access latency with and without additional prefetch checks. We observed that with the checks, the change in latency ranges between +0.06% and +2.53% of the base where the mean is +1.20%. Second, our write-through design implies that every PUT to prefetched data results in an additional local write to the prefetch buffers, possibly incurring overhead. We used a benchmark similar to the one in Section 6.2.1 to quantify this overhead. We observed that the change in normalized PUT latency fluctuates between −2.27% and +9.41%, where the mean change is +1.17%. Neither test showed differences with different architectures or types of prefetch. Therefore, full results are not shown.

6.2.5 Effectiveness Analysis. Due to the overheads of prefetching itself, its effectiveness depends on the number of remote accesses. To quantify this behavior, we used PRK-Stencil in a synthetic manner. We used a star stencil pattern (data is read from the neighbors to the East, North, West, and South), a 4,096×4,096 block-distributed array on two locales, and ran the experiment with increasing operator sizes. This parameter sweep results in an increase in remote access ratio in a well-defined manner. In this setup, remote access ratio becomes a function of the operator size. Derivation of this function is straightforward and omitted. We run this experiment with auto- and manual consistency and report speedup over the non-optimized version as a function of remote access ratio.

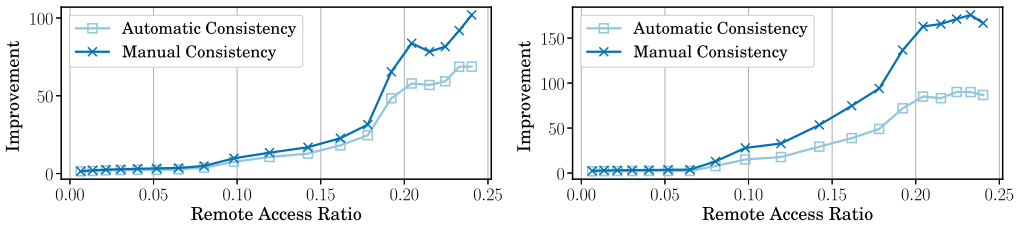


Fig. 10. Effect of remote access ratio on prefetch effectiveness.

The speedup as a function of remote access ratio is shown in Figure 10. Both networks show meager performance improvement with a very low amount of remote accesses. We observed that the break-even remote access ratio for gaining considerable speedup is between 5% and 10%. Especially between 10% and 20% speedup increases rapidly. At 25%, it reaches 100×–160× for manually consistent and 60×–80× for auto-consistent prefetching on Infiniband and Gemini, respectively.

This experiment shows the change in the effectiveness of prefetching as an application is made more communication-heavy. Another important parameter that can impact the effectiveness is the amount of computational load. However, when keeping the amount of communication and the number of processing elements constant, increasing the amount of computation arguably has a predictable effect according to Amdahl's law [6]. Therefore, we did not conduct such study.

6.3 Application Performance

To assess application performance, we used (1) some applications from those we had developed to analyze hand optimization opportunities [27] and (2) a subset of the PRK. The PRK are small application kernels that cover many common communication, computation, and synchronization patterns in HPC applications [41]. They also have been used for performance evaluation of PGAS languages [39, 42], including Chapel [26]. The kernels that we use represent common communication patterns. More details about the benchmarks are below.

Sobel Edge Detection: Two-dimensional, nine-point stencil operation for edge detection in an image. We use a square input image of size 20,000.

Heat Diffusion: Three-dimensional, six-point iterative stencil operation that simulates heat diffusion in a grid. We use a 3D square grid of size 400.

PRK-DGEMM: Blocked matrix multiply. Elements are double precision floating point numbers. Matrix size is 4,096×4,096. Block size is 32 elements.

PRK-Transpose: Blocked matrix transpose. Elements are double precision floating point numbers. Matrix size is 8,192×8,192. Block size is 8. The matrix is row-wise distributed.[2]

PRK-Sparse: Double precision sparse matrix-dense vector multiplication. Sparse matrix is square, of size 2^{20} with 25 non-zeros per row and row-wise distributed.

All applications distribute data using the default Block distribution in Chapel (with additions for prefetch support). Data sizes are selected to comply with the divisibility rules of the benchmarks, to obtain reasonable execution times, and to exceed the size of LLC. We have also conducted some studies with increasing data sizes but did not observe any noteworthy difference. We report results

[2]This diverges from the PRK specification [40], which suggests column-major arrays and column-wise decomposition. However, there is no support for column-major arrays in Chapel as of today.

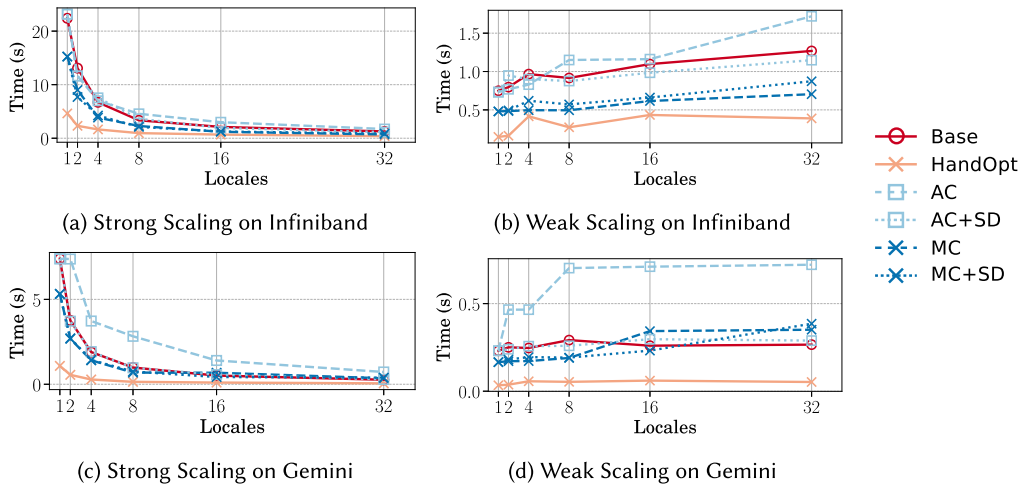


Fig. 11. Sobel edge detection performance results (*AC:* automatic consistency; *MC:* manual consistency; *SD:* static domain optimization).

of both strong and weak scaling in all benchmarks, except PRK-Sparse. The way in which the PRK-Sparse benchmark creates the sparse matrix prohibits precise control over the data size.

We experimented with both serialized and deserialized data. However, we did not observe any significant difference between the two, in alignment with our observation in Section 6.2.1. Therefore, we omitted the performance curves of using deserialized data from the plots. We use the following abbreviations/terms: *Base*: unoptimized version; *HandOpt*: hand-optimized version where data movements identical to LAPPS are implemented by the programmer; *AC*: auto-consistent LAPPS; *MC*: manually consistent LAPPS; *SD*: static domain optimization (Section 5.3.2).

Figure 11 shows the results of the Sobel benchmark. We observed that *MC* prefetching improved performance up to 2×, whereas *AC* prefetching showed some degradation. We believe that this behavior is due to synchronization and the low remote access ratio in the benchmark. In Section 6.2.5, we have shown that prefetching improves performance significantly if the remote access ratio is more than 5%. In Sobel, this ratio is below 0.05%. We also see that *Handopt* outperformed prefetching. Recall that access to the local copy created in the application space is type 1 access, which can be two times faster than access to prefetched data (Section 6.2.1). Also observe that the performances of *Base*, *MC*, and *HandOpt* are significantly different even with a single locale, emphasizing the effect of performance differences of access types. Even though we observed speedups by a factor of 2 on Infiniband, the benefit from prefetching is less on Gemini, due to better remote access performance of Gemini/uGNI. In the weak scaling tests, we see some performance degradation with increased number of locales. We attribute this to the locales that have a central chunk of the image and therefore need to communicate with eight different locales. Moreover, prefetching from corner neighbors involves a single pixel (1 byte) of the image that is not reused. Arguably, prefetching such small amount of data with no reuse has more overhead than issuing a GET.

Figure 12 shows the performance of the heat diffusion. Interpreting the results is difficult as we observed a performance degradation going from four to eight locales. This difference is more pronounced in Infiniband and it affects *Base* and *HandOpt* as well as *SD* versions. LAPPS versions without *SD* remained unaffected by the degradation and scaled well. It should be noted that this behavior has previously been observed [27]. However, in an attempt to understand the issue better, we implemented a synthetic benchmark (results not shown). We run 1D/two-point,

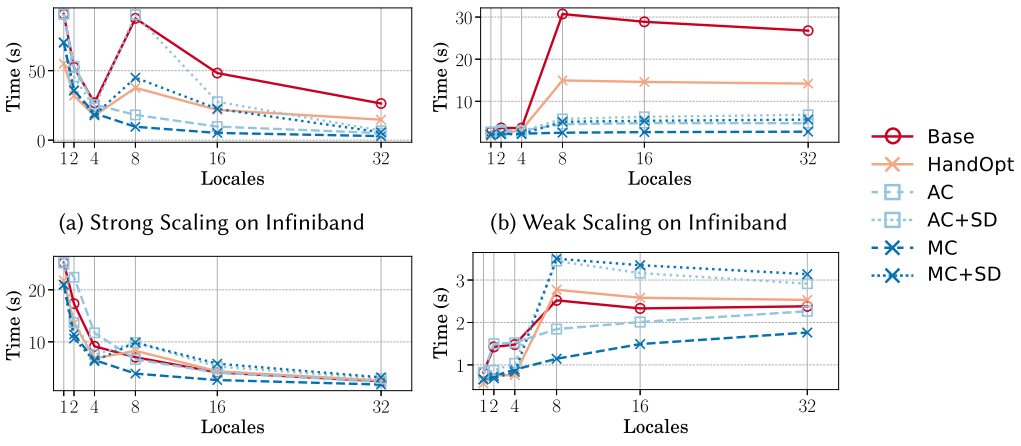


Fig. 12. Heat diffusion performance results (*AC:* automatic consistency; *MC:* manual consistency; *SD:* static domain optimization).

2D/four-point, and 3D/six-point stencils with one iteration and no synchronization (heat diffusion benchmark runs in synchronized iterations until convergence) in a 3D grid. For 1D and 2D stencils, we alternated the dimension of the operator. Results showed that the degradation happens in the 3D stencil only. We observed the degradation with single thread per locale, as well. This eliminates the possibility of network contention in central locales which communicate with different neighbors in parallel. Moreover, as discussed above, Sobel requires locales to communicate with eight neighbors and does not show similar performance degradation. Interestingly, we did not observe the degradation with runs that does not use *SD* optimization. Recall that without this optimization, owner locales create serial buffers and transfer the data to the prefetchers using PUTs. This changes the schedule of events that may be causing the performance degradation. Another possible reason for the performance issue is a design and/or implementation problem in multidimensional distributed arrays. With a small number of locales, we see that *MC* prefetching performs similarly to *HandOpt*, whereas *AC+SD* prefetching performs slightly worse yet delivers around 1.5× speedup. Note that, in this benchmark, maximum remote access ratio is 4%.

PRK-DGEMM results are shown in Figure 13. First, on both systems LAPPS outperformed *Base* by around two orders of magnitude. Moreover, almost all LAPPS versions outperformed *HandOpt*, especially with higher numbers of locales. We attribute this to the difference in initialization and update costs. We see that the *MC* version performed relatively poorly on Gemini compared to other LAPPS versions. We believe that this is due to additional synchronization required by the low-level implementation of PRK-DGEMM [26]. `coforall`-based implementation to support the blocking logic requires special update methods. This synchronization on top of the on statement-based protocol required by the lack of *SD* caused noticeable idling of processors. Moreover, PRK-DGEMM prefetches both input arrays, increasing the amount of synchronization further. Second, in strong scaling tests we observed that *MC+SD* scaled well, gaining more than 20× performance with 32 locales. In weak scaling tests, this version showed only very little degradation with four locales because of the change in communication patterns due to distribution of the matrices.

PRK-Transpose results are depicted in Figure 14. First, we observe up to 20× and 50× speedups on Infiniband and Gemini. The difference is mainly due to the worse *Base* performance with high number of locales on Gemini. In separate tests (results not shown), we observed that Gemini

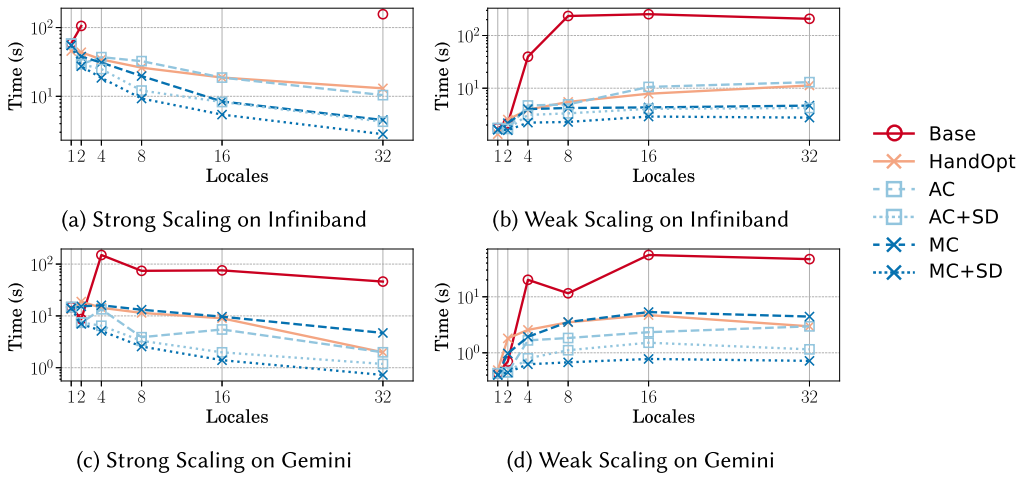


Fig. 13. PRK-DGEMM performance results. Missing data due to timed out jobs. (*AC:* automatic consistency; *MC:* manual consistency; *SD:* static domain optimization).

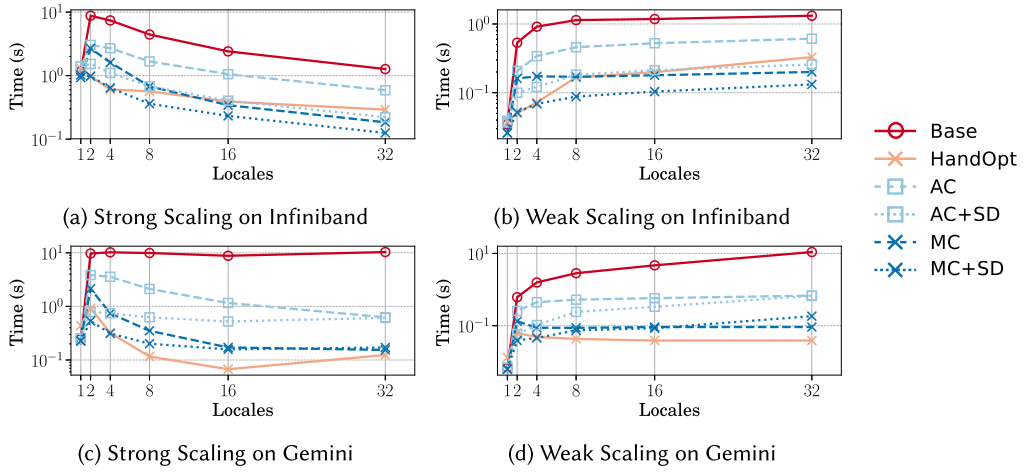
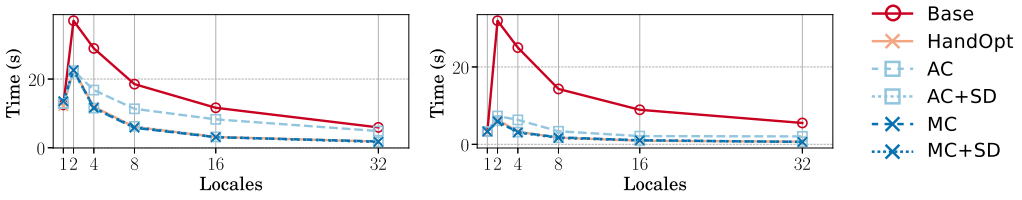


Fig. 14. PRK-transpose performance results (*AC:* automatic consistency; *MC:* manual consistency; *SD:* static domain optimization).

performs better with less number of threads per locale. We attribute this behavior to possible network contention on each locale due to simultaneous all-to-all communication with 16 threads. This is also in alignment with what we observe in weak scaling tests. On Gemini, weak scaling performance degraded as a greater number of locales were used, although the amount of total communication per locale is fixed. In contrast, Infiniband performance stabilized after some point. Second, we see that *SD* versions do not scale well on Gemini. This is due to the lower performance of *SD* for initialization and update (Sections 6.2.2 and 6.2.3). Finally, *HandOpt* outperformed LAPPS on Gemini, whereas on Infiniband, most LAPPS versions performed competitively to *HandOpt*. We believe the reason is twofold: (1) access type 1 is faster on Gemini than it is on Infiniband, whereas prefetch access types (5–10) have similar latencies (Section 6.2.1). Since *HandOpt* creates a local



(a) Strong Scaling on Infiniband (b) Strong Scaling on Gemini

Fig. 15. PRK-sparse performance results (*AC:* automatic consistency; *MC:* manual consistency; *SD:* static domain optimization).

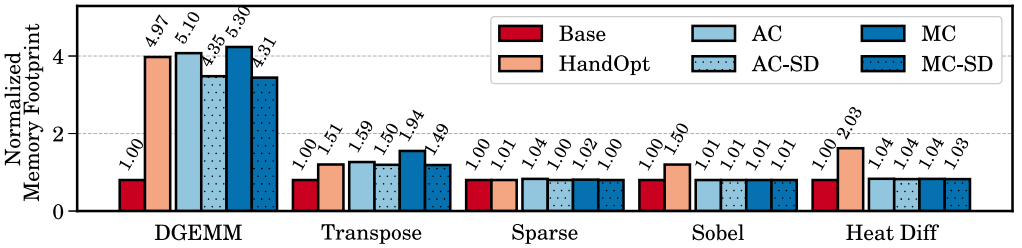


Fig. 16. Normalized memory footprint of application benchmarks (*AC:* automatic consistency; *MC:* manual consistency; *SD:* static domain optimization).

array to store the remote data, it does type 1 access. (2) Initialization cost of hand copying data on Gemini is not as bad as it is on Infiniband (Section 6.2.2).

PRK-Sparse results are shown in Figure 15. PRK-Sparse is different than other benchmarks as it does random remote accesses, and the remote data to be prefetched is more straightforward (input vector as a whole). We observed a significant difference in behavior of *AC* compared to other prefetch versions. Albeit it outperforms *Base* version by 1.5× and 4× on Infiniband and Gemini, for other versions these speedups are 3.5× and 9×. This is expected because the *AC* version is the least optimized prefetch version as it does synchronization per access *and* lacks *SD* optimization. However, other versions performed very similarly to *HandOpt* on both architectures.

6.4 Memory Footprint

As prefetching creates local replicas of remote data, it increases the memory usage. However, it is important to keep the memory cost of prefetching as low as possible in order to be able to use it on systems with limited memory or applications with large data. Figure 16 depicts the average per-locale memory footprint of different prefetch techniques. In this context, we use the term "memory footprint" to denote the largest amount of memory allocated at any given point during program execution (i.e., high-water mark). We normalized memory footprint to the *Base*. All results are obtained with 32 locales. Locale 0 is excluded as it allocates more data.

First, we observed that without *SD*, LAPPS has at most 4% memory overhead over the *HandOpt*. Moreover, in stencil operations like Sobel and Heat Diffusion, our design has less overhead, because *HandOpt* replicates the local and remote data to create a local array that stores all the data accessed by the locale. Arguably, a hand optimization can copy the data in a more precise fashion to reduce the footprint. However, this involves managing many different arrays to store the data replicated from different neighbors. Implementing such optimization is very arduous, especially with high-order stencil operations. In contrast, LAPPS allocates and manages space for remote data internally, without any programmer effort. Second, *SD* has a positive effect on the footprint. This is due to

additional space allocated on the owner node to create serialized buffers, when *SD* is not used. As discussed in Section 5.3.2, *SD* optimization avoids that step. Thirdly, we observed that *MC* can have a bigger footprint than *AC*. We believe this is due to the different timings of updates. In *MC*, all locales issue updates at the same time; whereas, in *AC* updates are issued as needed. Therefore, in the latter updates are staggered, reducing the likelihood of using more memory due to simultaneous data serialization on the owner locales. Note that there is no such difference in *SD* versions where the owner is passive. Finally, we see the footprint change similarly between different versions in all benchmarks. However, the amplitude of change depends on the remote access characteristic of the benchmark. For example, where PRK-DGEMM versions have about five times bigger memory footprint, *SD* versions do not even increase the memory footprint in PRK-Sparse. This is due to the initialization costs of sparse array initialization costs exceeding the memory cost of LAPPS.

6.5 Impact on Programmer Productivity

Programmer productivity is not easily quantifiable. In this section, we qualitatively discuss the matter in light of our experiences. First, the most important benefit of our design is to achieve complete separation of application logic and performance concerns. As discussed in Section 1, with hand optimization, the end programmer has to manage copies of remote data in separate arrays. This approach has a twofold impact on programmer productivity: (1) the programmer needs to use different variables for prefetched data. This gets complicated as data from different locales are prefetched. (2) The need for storing per-locale data enforces using `coforall`s instead of `forall` loops. `coforall` loops create a task per iteration and thus can be used to create SPMD regions. Whereas, `forall` loops create parallelism in a data-driven fashion. Therefore, using `coforall`s requires a lower-level approach in distributing the work across locales adding more burden to the programmer [26]. This implies that hand optimizations have indirect costs on programmer productivity that are side effects of the prefetch logic. In contrast, with our design neither of the above is a problem. Programmers can implement their application logic without communication optimization concerns, and then add necessary prefetch calls with no change to the application logic. This also implies that prefetching strategies can be swapped easily without modification to the application code. Without LAPPS, such changes would be tedious as they would require significant implementation changes.

Secondly, the interface for the end-programmer is object-oriented and simple. Three methods need to be used at most: a prefetch method (i.e., prefetch pattern), an update method that is used with *MC* only, and an evict method that is optionally used to reclaim memory. With the help of the small interface, the amount of additions to the application code is also minimal. For instance, Listing 4(c) shows all three methods in use in heat diffusion benchmark. As another example, in PRK-DGEMM, up to 40× speedup was obtained with *AC* compared to *Base*. This took adding two method calls (one per input matrix). Moreover, *MC* can be used with one additional method (update) call per matrix to achieve 70× speedup over *Base*.

7 CONCLUSION

The PGAS model provides a productive mechanism to program parallel architectures by creating a shared memory view, while supporting data locality awareness to be leveraged by programmers. Some overhead, however, is still generated due to fine-grained communication associated with remote data accesses. These overheads can be greatly eliminated by efficient remote data prefetching. However, implementing automatic/compiler-based techniques that can uncover access patterns in applications is very challenging as static code analysis and transformation opportunities can be limited due to conservatism in the compiler. On the other hand, relying on hard-coded

optimizations by the programmer greatly reduces the productivity, which is inconsistent with the premise of the PGAS model.

We argue that the PGAS model presents opportunities for more productive and efficient optimizations. These can be achieved via a semi-automatic technique that is user-directed but leverages locality awareness coupled with language libraries and the runtime system, thereby striking a balance between the two extremes: compiler-based automatic and user hard-coded optimizations. Thus, we propose the design and implementation of LAPPS for PGAS languages.

Our approach is to implement remote data prefetching as a language feature that inherently uses the full software stack of the language, thereby reducing the user effort while maintaining efficiency. Based on the runtime system, LAPPS augments the standard language libraries, and extends the compiler with the capability to generate appropriate data types that interact with the prefetch subsystem. This enables the full exploitation of locality awareness for the sake of productivity.

We prototyped LAPPS in the Chapel software stack to demonstrate its efficacy. Tests on two common interconnection networks and communication middleware showed that LAPPS can achieve up to orders of magnitude speedup over non-optimized counterparts and perform competitively to similar hard-coded optimizations. More importantly, such improvements can be achieved by adding a few method calls in the application, without disrupting the programmer experience.

To conclude, our work explores possibilities of achieving scalable performance in PGAS languages with minimal programmer effort. We believe that our findings can help understand the power of locality awareness in programming systems.

APPENDIX

A LAPPS API REFERENCE

The high-level interface is implemented by the domain maps and used by the programmer.

- **`stencilPrefetch(depth=1, corners=true, consistent=true, staticDomain=false)`**: See Figure 2(a) . `depth` argument sets the stencil depth; `corners` flag determines whether the Jacobi operator is plus- or square-shaped. Arguments with default values can be omitted.
- **`rowWisePrefetch(consistent=true, staticDomain=false)`**: See Figure 2(b). **`colWisePrefetch`**, **`transposePrefetch`** and **`allToAllPrefetch`** have identical arguments.
- **`customPrefetch(descTable, consistent=true, staticDomain=false)`** `descTable` is a list of domains that represent the set of indices that needs to be prefetched. It may contain local indices or non-existing indices in the array as they are filtered out internally.
- **`updatePrefetch()`**: Update prefetch entry and data.
- **`evictPrefetch()`**: Evict prefetch entry and data.

The low-level interface is implemented by the internal libraries and used by the domain maps.

- **`requestPrefetch(sourceIdx, sourceHook, sliceDesc, wholeDesc, consistent=true, staticDomain=false):`** `sourceIdx` and `sourceHook` denote the source node ID and the corresponding hook, respectively. `sliceDesc` is optional and used to prefetch only a slice of data. If omitted, source's data is prefetched as a whole. `wholeDesc` is the local domain of the array on the source. It is used to determine if a slice of data is suitable for static domain optimization. This argument is mandatory only if `sliceDesc` is passed and `staticDomain` is `true`.
- **`accessPrefetch(localeIdx, index, out prefetched):`** Return a prefetch reference to data with `index` originated from `localeIdx`. `prefetched` is a Boolean and set to `true` if

the data is prefetched, `false` otherwise. Return value is undefined if `prefetched` is set to `false`.
- —**`updatePrefetch():`** Update the prefetched data associated with this prefetch hook.
- —**`evictPrefetch():`** Evict the prefetched data associated with this prefetch hook.

Utility functions must be implemented by the domain maps and are used by the backend.

- —**`serializeData()`**: Yield serial chunk(s) of data.
- —**`serializeMetadata()`**: Yield serial chunk(s) of metadata.
- —**`getMetadataSize()`**: Return the size of metadata created for serial buffer in bytes.
- —**`getSerializedSize()`**: Return the size of data created for serial buffer in bytes.
- —**`getIdxFromByteOffset(data, offset)`**: Given a byte `offset`, compute the potentially multi-dimensional index of the element stored in `data+offset`.
- —**`getByteOffsetFromIdx(data, idx)`**: Given a potentially multi-dimensional index, compute the byte offset, whose index resides in the serial buffer `data`. `data` passed to these two methods is the data that is yielded by `serializeMetadata` and `serializeData`.
- —**`getDeserializationContainer(data)`**: Return the deserialization container for storing `data`.
- —**`getDataStartAccess(idx)`**: Return a C pointer to the element in position `idx`.

ACKNOWLEDGMENTS

We would like to thank Bradford L. Chamberlain, Vikram Narayana, Abdullah Kayi, Olivier Serres and Ahmad Anbar for many useful insights that contributed to the quality of this article.

REFERENCES

[1] 2017. Chapel Language Spefications - Version 0.984. Retrieved February 02, 2018 from https://chapel-lang.org/spec/spec-0.98.pdf.

[2] 2018. prefetch/noprefetch | Intel Software. Retrieved February 20, 2018 from https://software.intel.com/en-us/node/524554.

[3] 2018. Using the GNU Compiler Collection (GCC): Other Builtins. Retrieved February 20, 2018 from https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html.

[4] Michail Alvanos, Montse Farreras, Ettore Tiotto, José Nelson Amaral, and Xavier Martorell. 2013. Improving communication in PGAS environments: Static and dynamic coalescing in UPC. In *Proceedings of the 27th International ACM International Conference on Supercomputing*. ACM, 129–138.

[5] Michail Alvanos, Gabriel Tanase, Montse Farreras, Ettore Tiotto, Josè Nelson Amaral, and Xavier Martorell. 2013. Improving performance of all-to-all communication through loop scheduling in pgas environments. In *Proceedings of the 27th International ACM International Conference on Supercomputing*. ACM, 457–458.

[6] Gene M. Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the Spring Joint Computer Conference (AFIPS'67 (Spring))*. ACM, New York, 483–485. DOI: http://dx.doi.org/10.1145/1465482.1465560

[7] Ahmad Anbar, Olivier Serres, Engin Kayraklioglu, Abdel-Hameed A. Badawy, and Tarek El-Ghazawi. 2015. PHLAME: Hierarchical locality exploitation using the PGAS model. In *Proceedings of the 2015 9th International Conference on Partitioned Global Address Space Programming Models (PGAS'15)*. 82–89. DOI: http://dx.doi.org/10.1109/PGAS.2015.16

[8] Ahmad Anbar, Olivier Serres, Engin Kayraklioglu, Abdel-Hameed A. Badawy, and Tarek El-Ghazawi. 2016. Exploiting hierarchical locality in deep parallel architectures. *ACM Transactions on Architecture and Code Optimization (TACO)* 13, 2 (2016), 16.

[9] Rajkishore Barik, Jisheng Zhao, David Grove, Igor Peshansky, Zoran Budimlic, and Vivek Sarkar. 2011. Communication optimizations for distributed-memory X10 programs. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium (IPDPS'11)*. 1101–1113. DOI: http://dx.doi.org/10.1109/IPDPS.2011.105

[10] Dan Bonachea. 2002. *GASNet Specification, v1.1*. Technical Report UCB/CSD-02-1207. EECS Department, University of California, Berkeley.

[11] Bradford L. Chamberlain. 2001. *The Design and Implementation of a Region-Based Parallel Programming Language*. University of Washington.

[12] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. 2007. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications* 21, 3 (Aug. 2007), 291–312. DOI: http://dx.doi.org/10.1177/1094342007078442

[13] Bradford L. Chamberlain, Sung-eun Choi, Steven J. Deitz, David Iten, and Vassily Litvinov. 2011. Authoring user-defined domain maps in chapel. In *Proceedings of Cray Users Group*.

[14] Satish Chandra, Vijay Saraswat, Vivek Sarkar, and Rastislav Bodik. 2008. Type inference for locality analysis of distributed data structures. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*. ACM, New York, 11–22. DOI: http://dx.doi.org/10.1145/1345206.1345211

[15] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. 2010. Introducing OpenSHMEM: SHMEM for the PGAS community. In *Proceedings of the 4th Conference on Partitioned Global Address Space Programming Model (PGAS'10)*. ACM, New York, 2:1–2:3. DOI: http://dx.doi.org/10.1145/2020373.2020375

[16] Wei-Yu Chen, Costin Iancu, and Katherine Yelick. 2005. Communication optimizations for fine-grained UPC applications. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*. IEEE, 267–278.

[17] Sung-Eun Choi and L. Snyder. 1997. Quantifying the effects of communication optimizations. In *Proceedings of the 1997 International Conference on Parallel Processing (Cat. No.97TB100162)*. 218–222. DOI: http://dx.doi.org/10.1109/ICPP.1997.622647

[18] Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, François Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanti, Yiyi Yao, and Daniel Chavarría-Miranda. 2005. An evaluation of global address space languages: Co-array Fortran and unified parallel C. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'05)*. ACM, New York, 36–47. DOI: http://dx.doi.org/10.1145/1065944.1065950

[19] Tarek El-Ghazawi and François Cantonnet. 2002. UPC performance and potential: A NPB experimental study. In *Proceedings of the ACM/IEEE 2002 Conference on Supercomputing*. IEEE, 1–26.

[20] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. 2003. *UPC: Distributed Shared-Memory Programming*. Wiley-Interscience.

[21] Tarek El-Ghazawi and Sébastien Chauvin. 2001. UPC benchmarking issues. In *Proceedings of the International Conference on Parallel Processing, 2001*. IEEE, 365–372.

[22] Michael P. Ferguson and Daniel Buettner. 2015. Caching puts and gets in a PGAS language runtime. IEEE, 13–24. DOI: http://dx.doi.org/10.1109/PGAS.2015.10

[23] Riyaz Haque and David Richards. 2016. Optimizing PGAS overhead in a multi-locale chapel implementation of CoMD. In *Proceedings of the F1st Workshop on PGAS Applications*. IEEE, 25–32. https://e-reports-ext.llnl.gov/pdf/838618.pdf.

[24] Akihiro Hayashi, Jisheng Zhao, Michael Ferguson, and Vivek Sarkar. 2015. LLVM-based communication optimizations for PGAS programs. ACM, 1–11. DOI: http://dx.doi.org/10.1145/2833157.2833164

[25] Costin Iancu, Wei Chen, and Katherine Yelick. 2008. Performance portable optimizations for loops containing communication operations. In *Proceedings of the 22nd Annual International Conference on Supercomputing*. ACM, 266–276.

[26] Engin Kayraklioglu, Wo Chang, and Tarek El-Ghazawi. 2017. Comparative performance and optimization of Chapel in modern manycore architectures. In *Proceedings of the 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW'17)*. 1105–1114. DOI: http://dx.doi.org/10.1109/IPDPSW.2017.126

[27] Engin Kayraklioglu, Olivier Serres, Ahmad Anbar, Hashem Elezabi, and Tarek El-Ghazawi. 2016. PGAS access overhead characterization in Chapel. In *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW'16)*. IEEE, 1568–1577.

[28] Charles H. Koelbel and Mary E. Zosel. 1993. *The High Performance FORTRAN Handbook*. MIT Press, Cambridge, MA.

[29] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*.

[30] Paul E. McKenney and John D. Slingwine. 1998. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*. 509–518.

[31] John M. Mellor-Crummey and Michael L. Scott. 1991. Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'91)*. ACM, New York, 106–113. DOI: http://dx.doi.org/10.1145/109625.109637

[32] Matthias M. Müller. 1999. KaHPF: Compiler generated data prefetching for HPF. In *High Performance Computing in Science and Engineering 99*. Springer, Berlin, 474–482. DOI: 10.1007/978-3-642-59686-5_46.

[33] Matthias M. Müller, Thomas M. Warschko, and Walter F. Tichy. 1998. Prefetching on the cray-T3E. In *Proceedings of the 12th International Conference on Supercomputing (ICS'98)*. ACM, New York, 361–368. DOI: http://dx.doi.org/10.1145/277830.277919

[34] Robert W. Numrich and John Reid. 1998. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum* 17, 2 (Aug. 1998), 1–31. DOI: http://dx.doi.org/10.1145/289918.289920

[35] Arun Raman, Greta Yorsh, Martin Vechev, and Eran Yahav. 2011. Sprint: Speculative prefetching of remote data. In *ACM SIGPLAN Notices* 46, 10 (2011), 259–274.

[36] John Reid. 2008. The new features of Fortran 2008. *SIGPLAN Fortran Forum* 27, 2 (Aug. 2008), 8–21. DOI: http://dx.doi.org/10.1145/1408643.1408645

[37] Alberto Sanz, Rafael Asenjo, Juan López, Rafael Larrosa, Angeles Navarro, Vassily Litvinov, Sung-Eun Choi, and Bradford L. Chamberlain. 2012. Global data re-allocation via communication aggregation in Chapel. In *Proceedings of the 2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'12)*. 235–242. DOI: http://dx.doi.org/10.1109/SBAC-PAD.2012.18

[38] Kyle B. Wheeler, Richard C. Murphy, and Douglas Thain. 2008. Qthreads: An API for programming with millions of lightweight threads. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS'08)*. 1–8. DOI: http://dx.doi.org/10.1109/IPDPS.2008.4536359

[39] Rob F. Van der Wijngaart, Abdullah Kayi, Jeff R. Hammond, Gabriele Jost, Tom St John, Srinivas Sridharan, Timothy G. Mattson, John Abercrombie, and Jacob Nelson. 2016. Comparing runtime systems with exascale ambitions using the parallel research kernels. In *High Performance Computing*. Springer, Cham, 321–339. http://link.springer.com/chapter/10.1007/978-3-319-41321-1_17 DOI: 10.1007/978-3-319-41321-1_17.

[40] Rob F. Van der Wijngaart, Tim Mattson, Jeff Hammond, Srinivas Sridharan, and Evangelos Georganas. 2017. Parallel Research Kernels. Retrieved September 11, 2017 from https://github.com/ParRes/Kernels/blob/master/doc/par-res-kern-report-v1.3.pdf.

[41] Rob F. Van der Wijngaart and Tim G. Mattson. 2014. The parallel research kernels. In *Proceedings of the 2014 IEEE High Performance Extreme Computing Conference (HPEC'14)*. 1–6. DOI: http://dx.doi.org/10.1109/HPEC.2014.7040972

[42] Rob F. Van der Wijngaart, Srinivas Sridharan, Abdullah Kayi, Gabriele Jost, Jeff Hammond, Tim G. Mattson, and Jacob E. Nelson. 2015. Using the parallel research kernels to study PGAS models. In *Proceedings of the 2015 9th International Conference on Partitioned Global Address Space Programming Models*. 76–81. DOI: http://dx.doi.org/10.1109/PGAS.2015.24

[43] Yili Zheng, Amir Kamil, Michael B. Driscoll, Hongzhang Shan, and Katherine Yelick. 2014. UPC++: A PGAS extension for C++. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 1105–1114. DOI: http://dx.doi.org/10.1109/IPDPS.2014.115

Received November 2017; revised March 2018; accepted June 2018