# Caching Puts and Gets in a PGAS Language Runtime

Michael P. Ferguson
*Cray Inc.*
*Seattle, Washington*
*mferguson@cray.com*

Daniel Buettner
*Laboratory for Telecommunication Sciences*
*College Park, MD*
*dbuettner@ltsnet.net*

***Abstract*—We investigated a software cache for PGAS PUT and GET operations. The cache is implemented as a software write-back cache with dirty bits, local memory consistency operations, and programmer-guided prefetch. This cache supports programmer productivity while enabling communication aggregation and overlap. We evaluated an implementation of this cache for remote data within the Chapel programming language. The cache provides a 2x speedup for several distributed memory application benchmarks written in Chapel across a variety of network configurations. In addition, we observed that improvements to compiler optimization did not remove the benefit of the cache.**

***Keywords*-PGAS; cache; remote data; communication aggregation; communication overlap; prefetch; Chapel**

## I. Introduction

Partitioned Global Address Space (PGAS) languages and libraries often support communication overlap and aggregation. Such support can include explicit user calls, compiler optimizations, or runtime support. Each approach includes trade-offs between program complexity, mechanism applicability, and resulting performance.

The PGAS programming model is built upon the idea of one-sided PUT and GET operations. A PUT is a possibly hardware-supported message that writes to a memory location on another node in a distributed memory system. Similarly, a GET reads from a memory location in another node. Communication overlap and aggregation are key to reducing the impact of the latency of these PUT and GET operations. PGAS systems have primarily used three techniques to support communication overlap and aggregation: relaxed memory models, explicit non-blocking PUT and GET calls, and compiler optimization. However, each of these approaches has drawbacks (see Section IV).

At the same time, the existing Chapel compiler and runtime produce programs with too much fine-grained communication. While we expect the Chapel compiler to improve in this regard, fine-grain communication is in inherent to many PGAS programs. These fine-grain PUT and GET operations can really limit performance if the communication is not optimized. At the same time, a compiler alone cannot optimize these operations in some common situations (see Section IV-B). In order to address this problem, we implemented a runtime cache for remote data.

This paper provides the following contributions: a description of the new cache, a discussion of different methods to overlap and aggregate PUTs and GETs, and an initial experimental evaluation of the cache. In addition, we show that such a cache has value even with improved compiler optimization. While we worked with the Chapel language, this cache could offer benefits to other PGAS languages and libraries including UPC [1], UPC++ [2], Fortran 2008 [3], and OpenSHMEM [4] to name a few.

## II. Chapel Background

Chapel [5], [6] is a parallel programming language designed for productive scalable computing. It supports parallelism within and across nodes via high-level abstractions for data parallelism and task parallelism. Chapel was designed from first principles rather than by extending an existing language. It is imperative and uses block-structured syntax like C, C++, Fortran, Java, and Python.

As a PGAS language, Chapel allows programmers to specify the placement of both data and computation. Chapel programs can run across many nodes in a compute cluster or supercomputer. These nodes are called *locales* in Chapel. One important feature of Chapel is distributed arrays [7], [8]. Array distributions can be specified when an array is declared. Code working with the distributed array will perform implicit communication to access the array. In this way, a parallel algorithm can be separated from potentially platform-specific communication details. In cases where this separation of concerns strategy breaks down, Chapel's multi-resolution design philosophy allows developers to incrementally add architectural detail to realize full performance.

## A. Chapel Memory Consistency

When adding communication overlap — which can take the form of *prefetch* and *write behind* — it is important to understand the memory consistency model in order to know when these optimizations can be applied. Although Chapel's memory model is a work in progress [9], it is similar to the memory models of C11 [10], C++11 [11], and UPC [1]. These memory models all use the idea of sequential consistency for data-race-free programs. This model is robust enough to apply in a distributed memory context. In particular, the rules for loads and stores can be straightforwardly extended into rules for GETs and PUTs.

The Chapel, C11, and C++11 memory models provide some lower-level features in addition to sequential consistency for data-race-free programs. In particular, they support *acquire*, and *release* operations. An *acquire* indicates that any read or write that is requested after the *acquire* cannot be started before the *acquire*. A *release* indicates that any read or write that is started before the *release* cannot complete after the *release*. Note that programs in Chapel, C11, and C++11 do not typically require explicit *acquire* or *release* operations. Instead, operations such as acquiring a lock, starting a thread, or using an *atomic* variable imply *acquire* and/or *release* semantics. See the article by Boehm [12] for an introduction to memory consistency models.

As a concrete example, consider the psuedo-code in Listing 1 which contains two tasks that are run in parallel. Without *acquire* and *release*, there is nothing to prevent the compiler or the processor from reordering these instructions. Task A could initiate the write to the `notify` variable before the write to `x` has completed. Task B could *prefetch* the value of `x` and use a value from before the while loop. Or, task B could read `notify` into a register once and loop forever since the register will not change in the while loop. The fundamental problem is that the compiler and processor have no way to know that the `notify` variable is actually being used for inter-task communication. These systems are trying to improve performance with *prefetch* and *write-behind*. These issues come up for programs in many PGAS languages whether or not the relevant variables are remote.

To prevent the problematic optimization, this program must indicate that `notify` is being used to synchronize the two threads. Listing 2 shows the *acquire* and *release* operations that will ensure correctness. Under Chapel's memory model, adding *acquire* and *release* will ensure correct execution even when task A, task B, `notify`, and `x` are on different locales. The *release* operation in task A will cause the write to `x` to occur before `notify` is updated. At the same time, the *acquire* operation in task B will cause the value of `x` to be read after `notify` is read. Note that a Chapel program would typically use `sync` or `atomic` variables to communicate between these two tasks. The `sync` and `atomic` variables default to sequential consistency and so imply both *acquire* and *release*. We used explicit *acquire* and *release* operations here in order to introduce these ideas since our cache will need to differentiate between *acquire* and *release*. As long as our optimization respects *acquire* and *release*, the memory model allows *prefetch* and *write-behind* for local and remote memory locations.

**Listing 1** Racy program with incorrect synchronization

```
// global variables
var x=0;
var notify=0;

begin { // task A: a producer task
  x=42;
  notify=1; // INCORRECT synchronization!
}
begin { // task B: a consumer task
  // INCORRECT synchronization!
  while 0 == notify {
    /* wait */
  }
  compute_with(x);
}
```

**Listing 2** Correct synchronization

```
// global variables
var x=0;
var notify:atomic int;

begin { // task A: a producer task
  x=42
  notify.write(1,
         order=memory_order_release);
}
begin { // task B: a consumer task
  // wait for notification from A
  while 0 == notify.read(
                     memory_order_acquire) {
    /* wait */
  }
  compute_with(x);
}
```

## III. Implementation

This section describes the cache for remote data that we have designed and implemented. Source code is available in the open-source Chapel project repository[1]. It is a software write-back cache with automatic sequential prefetch and optional programmer-guided prefetch.

### A. Implementation Overview

At a high level, the cache for remote data works by adding a layer between Chapel statements that refer to remote data

[1]The open source Chapel repository is available at http://github.com/chapel-lang/chapel.

and the underlying PGAS library that handles the network requests. The cache stores *cache pages* that contain *cache lines*. GETs are always rounded up to whole *cache lines*. Data structures keep track of *valid* or *dirty* areas inside each cache page. Valid regions are those that have been read and dirty regions are those that have been written to but for which a network PUT operation has not yet started. Thus, it is a *write-back* cache. This design allows the cache to aggregate writes. Besides PUT and GET operations, the cache supports *acquire* and *release* operations, programmer-guided prefetch, and automatic sequential read-ahead.

Where Chapel statements would cause a GET to occur, the cache for remote data rounds up the size of the GET to whole cache lines and stores the fetched data in the cache. This approach works with spatial and temporal locality: accesses to nearby memory or repeated accesses to the same memory can often be completed without communication.

Where Chapel statements would cause a PUT to occur, the cache for remote data stores the new value in the cache and records which bytes were written (by marking those bytes as dirty) so that adjacent memory locations are not inadvertently updated when the PUT completes. It then defers initiating the PUT until it either encounters a *release* or the number of dirty pages is greater than a configurable maximum. By putting off starting the PUT, the cache is able to aggregate PUTs to adjacent memory locations. Note that in order for sequential programs to work correctly when these PUTs are deferred, the PUT value must be stored in the cache and returned when the same task reads the value it just wrote. Note that unlike GETs, PUTs do not round up to cache lines since without a traditional concurrency protocol (such as those described in [13]) we cannot safely overwrite any memory other than what the programmer requested.

The cache supports a prefetch hint and automatic sequential prefetch. These prefetch operations are implemented with non-blocking GETs. If the prefetched region of the cache is read, the implementation will wait for these non-blocking operations. The automatic prefetch is triggered when two cache lines within the same cache are separately read. In that case, the implementation reads the rest of the cache page and marks that cache page with a prefetch trigger. When a page with the prefetch trigger is read, an asynchronous prefetch for the next region needing prefetch is started. Note that neither form of automatic prefetch requires that the prefetched data be valid memory from the application's point of view. The implementation avoids prefetching memory that might be outside of addressable memory. For the GASNet `fast` segment configuration, the implementation considers any address within the pre-allocated heap to be addressable. For the `everything` segment, the implementation does not prefetch data outside of a requested system page.

We chose a conservative memory consistency protocol that does not add messages beyond the original PUTs and GETs. An *acquire* has the effect of invalidating everything in the cache and a *release* causes the implementation to complete any pending operations.

### B. Cache Data Structures

We implemented a 2Q cache since they are reported to perform better than LRU under real work loads [14]. Cache entries are organized in a *pointer tree*: a two-level hashtable with distinct hashing functions that select disjoint parts of the remote address at each of the two levels. Each hash table bucket stores a linked list of *cache entries*. Each cache entry points to a cache page, which stores 1024 bytes. The cache lines within that page are 64 bytes each. Since network GETs are rounded up to fill entire cache lines, all GETs are aligned to the cache line size. Each cache entry stores 1 bit per cache line to record which cache lines are valid. Each cache entry also records which areas have pending writes using dirty bits. It uses one bit per byte to record the modified addresses with a byte-level granularity. In order to minimize the size of each cache entry, the cache entry contains a pointer to a dirty bits structure. The implementation supports a limited number of dirty pages at a time.

Since several tasks might be executed by the same OS thread and all share a cache, each task stores a sequence number that indicates last time an *acquire* was run on that task. We also keep track of the sequence numbers for the first and last read and write to each cache page. These sequence numbers allow the implementation to detect an *acquire* in the requesting task that occurred after the *cache page* was populated. We also use these sequence numbers to handle conflicting accesses to the same cache page.

This cache was initially developed on a small cluster with an InfiniBand network, and the sizes of the cache elements were determined experimentally using this network. We chose 64 bytes for the size of a cache line for two reasons. First, it is the largest request size which has no significant increase in latency from an 8 byte request. Second, it is a typical cache line size in current microprocessors, and so data structures that are designed to work well with 64-byte cache lines to better use L1 and L2 cache will also work well in a distributed memory setting. We chose 1024 bytes for the size of a cache page because it is the smallest request size that allows close to peak bandwidth. A larger page size enables larger contiguous network transfers and thus higher bandwidth, but larger cache pages also increase the memory overhead of the cache in situations where just a few bytes per page are accessed.

Our implementation uses fixed size and pre-allocated memory for all cache structures. For the experiments reported in Section V, each thread-local cache can store up to 1MiB of data and consumes about 1.75MiB of memory. We used a maximum of 32 dirty pages in the experiments.

## C. Cache Operations

Having described the structures used by the remote cache to perform its bookkeeping, let us now turn our attention to exactly how the cache for remote data handles GET, PUT, *release*, and *acquire* operations.

When processing a GET, the implementation first checks to see if the needed cache page already has the relevant data from a PUT or GET operation after the task's last *acquire*. If so, it returns the cached data. If not, it will start by waiting for any conflicting operations such as a pending write or prefetch to the same cache page. If no cache page was found, it will find an unused cache page, evicting one if necessary. Then it will immediately start a non-blocking GET to read the relevant cache lines. While the GET is ongoing, the implementation adjusts the pointer tree and the 2Q queues as necessary and marks the relevant cache lines as valid. Finally, it waits for the GET to complete and returns the requested data.

When processing a PUT, the implementation searches the pointer tree for the relevant cache page. If none is found, it finds an unused cache page, evicting one if necessary. Since PUTs also require dirty bits, it finds an unused dirty bits entry if the cache entry does not already have one, cleaning a dirty page if necessary (as described below). The implementation invalidates the relevant cache lines if they were read before the task's last *acquire*. Then it waits for any network operation that could overlap with regions of the cache page that it is about to overwrite. Finally, it copies the data into the appropriate part of the cache page and updates the dirty bits accordingly.

Note that the technique of waiting for conflicting cache operations will cause a write to a cache page followed by a read to a different portion of a cache page to wait for the write to complete and then do a new GET of the read region. In the case that the read is to the same values that were written, the written values are returned without delay.

To clean a dirty page, the cache needs only to start PUTs for any region marked dirty. It is not necessary to wait for those PUTs to complete to clean a page. To start the PUTs, the implementation finds each contiguous section of dirty memory locations within a single cache page and creates a request for it. In this manner, PUTs to adjacent memory locations are aggregated into fewer and larger PUTs, up to the cache page size.

On a *release* operation, the implementation cleans all dirty pages and then waits for all operations to complete.

On an *acquire* operation, the cache conceptually flushes all data. It sets the task-local acquire sequence number to the cache's current sequence number. Cache data will be refreshed when the task accesses any existing cache page.

There are Chapel communication operations that do not reduce to simple blocking put or get, such as strided get and put operations. In our implementation, we handle those by wrapping them with *release* and *acquire* fences. Chapel also supports network atomic operations for read-modify-write activity, which are supported by the fences added by the compiler as described in the next section.

**Listing 3** Allocation, freeing, and the cache

```
// global variables
class Point { var x,y: int; }
p = new Point(1,2);
on Locales[1] {
  p.x = 5; p.y = 10;
  free(p);
}
```

## D. Memory Consistency

*Acquire* and *release* are necessary for correct program behavior and in some cases are implied by the Chapel memory model for sequential programs. We modified the Chapel compiler to add *acquire* and *release* fences to the `on` statement, task creation, task join operations, `sync` variable use, and `atomic` variable use. The task operations need a fence in our implementation because we chose to create one cache per OS thread. In contrast, if there were one cache per locale, the task creation and join operations would not require fences. However, different tasks performing fences on the global cache could cause significant interference.

The `on` statement requires a fence since each locale has a separate cache. The concept of the `on` statement is to migrate a sequential execution context to another locale. In other words, the `on` statement is not a parallel operation: a serial Chapel program can use `on` statements. A fence ensures that program order is preserved even as the program moves to another locale and starts to work with a different cache.

Let us consider two example programs to illustrate the memory model we are supporting and to explain the implications of the cache. Recall the corrected program in Listing 2. When `notify` is atomic and written to with a *release* in task A, the cache will complete all operations — including the write to `x` — before updating `notify`. In task B, reading `notify` with *acquire* causes the cache to discard any cached values, including any cached value of `x` from before the *acquire* operation.

The second example program, shown in Listing 3, will help to demonstrate how the cache interacts with the memory allocator. First, after the Point `p` is allocated, the fence implied by the `on` statement causes the current value of `p` to be communicated to the portion of the program running on Locale 1. For the portion of the program deallocating the memory, the writes setting the point value to $(5, 10)$ will be completed before the memory is freed because the free function is implemented with an `on` statement and thus implies a fence. Note that if Chapel supported remote heap allocation and deallocation without an `on` statement, it would be necessary to include fences in those operations.

### E. Implementation Challenges

It took significant effort to implement this cache efficiently enough to see improved performance. The implementation described in this paper is the fourth design we tried. The basic challenge is that the network is already very fast. For a low-latency network, a small PUT or GET completes in approximately 2 $\mu s$. That provides roughly enough time for 1 call to `malloc`; 20 processor synchronizations; 20 processor cache misses; or 2000 machine instructions.

These numbers strongly influenced our implementation. All memory needed by the cache is pre-allocated. The cache starts a non-blocking GET as soon as possible and performs bookkeeping while the GET is underway. Using one cache per OS thread avoids processor synchronization. We did our best to minimize the memory reads and writes while processing a GET or PUT by avoiding repeated enqueuing and dequeuing of pending operations.

## IV. Discussion

In this section, we will compare our approach to four other strategies: weak memory consistency, compiler optimization, explicit non-blocking communication, and explicit communication aggregation.

First, note that it is not necessary to modify Chapel programs to take advantage of the cache. To achieve good performance with the cache, a Chapel program needs to have fairly long running tasks without too much synchronization. The most common idiom for parallelism in Chapel, the `forall` loop, typically creates such tasks.

### A. Preserving Program Order

Our cache for remote data provides Chapel programs with some of the performance benefits normally limited to programming models with a weaker memory consistency model. We will use OpenSHMEM [4] as a concrete example in order to understand the benefits and drawbacks of weaker memory consistency. OpenSHMEM is a PGAS programming library that requires explicit fences in order to guarantee that two PUTs complete in program order. See Listing 4 for an example sequential program in which an OpenSHMEM implementation could reorder the PUTs. Note that it would not matter if all of the PUTs and GETs were implemented with OpenSHMEM atomics (such as `shmem_int_fadd` for example) since the OpenSHMEM atomics guarantee only atomicity and do not have any implication for the order of operations.

The weak ordering in OpenSHMEM is attractive because it can easily be implemented to achieve high performance. There are two reasons for this property. First, OpenSHMEM programs can take advantage of networks that reorder messages in order to achieve higher performance. An example might be a network that load balances across multiple network links. PUTs can be buffered and load balanced across the different network links even if that will result in a change in delivery order. Second, an implementation is free to buffer PUTs and send them off in a “fire and forget” manner. When remote completion of PUT operations is important, programmers must use an OpenSHMEM fence or quiet operation. Note that this weak memory model does not help overlap GETs with computation since the result of the GET must be returned. To enable overlap, OpenSHMEM extensions provide non-blocking GET operations such as `shmemx_int_get_nb` that use explicit handles. We will discuss this explicit handle approach in Section IV-C.

**Listing 4** Example of OpenSHMEM’s weak memory consistency. Without an explicit fence, the PUT operations on lines 2 and 3 can be reordered. As a result, when this program completes, `y` could have any of the values 0, 2, or 3.

```
1 x is a global variable initialized to 0
2 PUT 2 into x
3 PUT 3 into x
4 y = GET x
```

The performance benefits of OpenSHMEM PUTs come at the cost of weak memory consistency semantics. This trade-off might not be appropriate for a PGAS language. We have already discussed a totally sequential program that can have potentially misleading outcomes in Listing 4. These misleading outcomes are an acceptable trade-off for OpenSHMEM because the C or Fortran program calling the OpenSHMEM library will still have reasonable memory consistency for local and sequential operations. PGAS languages like UPC and Chapel seek to unify programming for local and remote data, so that the same algorithm can operate on distributed or local arrays, depending on variable declarations. Since the expression of the algorithm is meant to be the same whether or not the data is local, these languages need to have the same consistency semantics for local and remote data. At the same time, it would be confusing not to have program order semantics for sequential and local programs. Therefore, these languages need to extend program order semantics to operations on remote data.

A cache for remote data allows a PGAS language implementation to have both the performance benefits of the OpenSHMEM approach and a more understandable memory consistency model. In particular, the cache can distinguish between conflicting and non-conflicting operations. The cache handles two conflicting operations by explicitly waiting for the completion of the first operation. Any non-conflicting operations can be freely reordered by the network.

### B. Interaction with Compiler Optimization

The cache for remote data is not a replacement for good compiler optimization. Instead, it enhances the reach of compiler optimizations that might otherwise be limited in applicability by the need to preserve program order semantics. Consider the case of compiler optimizations for UPC programs targeting a network that performs better if it can reorder messages. Suppose a UPC program contains a loop that contains a PUT. We would like the compiler to produce a program that performs many PUTs at a time. However, if the compiler's alias analysis infrastructure is unable to prove that each PUT will go to a different memory location, the compiler will have to arrange for the PUTs to be ordered. That might mean waiting a network round-trip for each PUT to complete. A cache for remote data improves the situation. In the case that the PUTs overlap, the cache itself will ensure memory consistency. Even for a network that reorders messages, it will not be necessary to wait for each PUT unless the cache identifies a conflicting access.

In addition, the cache for remote data makes easier it for a compiler to optimize GETs. Prefetch hints are much easier for a compiler to emit than non-blocking GET calls since the non-blocking calls would require buffer management, completion handles, and possibly successful alias analysis. Alias analysis is necessary to ensure that that any PUTs in the loop do not overlap with the remote address of the GETs. In contrast, prefetch hints allow the compiler to overlap communication while ignoring the possibility that PUTs or GETs alias since the the cache correctly handles the conflicting accesses at run-time.

Lastly, our experiments show that significant improvements to compiler optimization did not remove the performance benefit of the cache. See Section V-C.

### C. Communication Overlap

With support for explicit non-blocking GETs and PUTs, a Chapel programmer could manually arrange communication overlap. However, this manual optimization increases program complexity. Using the prefetch hint provided by the cache offers a promising alternative that should be familiar to performance-minded programmers. At the same time, the cache will automatically overlap PUT operations with no intervention from the programmer.

Suppose we have a program that adds values from a computed index in an array but where the function computing the indices does not depend on previous array values. Listing 5 shows such a program in Chapel. Here we assume that `f` is computationally lightweight and has no side effects.

The access of `A[f(i)]` will become a GET request for 8 bytes. The Chapel runtime will issue each of these GET requests in a blocking manner. Thus, the program will spend a lot of time waiting for each GET request to complete. While there are many ways to improve upon the Chapel compiler and runtime, this irregular access pattern would make compiler-based communication optimization unlikely. At the same time, even though the computation could be parallelized, each task would still wait for each GET.

**Listing 5** Reading from a computed index

```
var A:[1..n] int;
on Locales[1] {
  var sum:int;
  for i in 1..n do sum += A[f(i)]
}
```

**Listing 6** Using non-blocking GET

```
var A:[1..n] int;
on Locales[1] {
  var sum:int;
  var h: [0..k] handles;
  var bufs: [0..k] int;
  // Warm up loop
  for i in 1..k {
    // nonblocking GET A[f(i)]
    //   into bufs[i%k]
    h[i%k] = get_nb(bufs[i%k], A[f(i)])
  }
  for i in 1..n {
    wait (h[i%k]);
    sum += bufs[i%k];
    if i+k <= n {
      // nonblocking GET A[f(i+k)]
      //   into bufs[(i+k)%k]
      h[(i+k)%k] =
        get_nb(bufs[(i+k)%k],A[f(i+k)])
    }
  }
}
```

Listing 6 shows an example implementation using explicit non-blocking calls. Our loop now has a startup loop and a more complicated loop body that explicitly performs the non-blocking GETs $k$ loop accesses before they are used. We needed to introduce two new arrays: an array of handles to manage the completion of the nonblocking operations and an array of buffer space in which to store the result of these nonblocking operations.

Since the Chapel language aspires to separate concerns of communication and data distribution from the description of an algorithm, explicit communication calls are not desirable because they embed the communication into the algorithm.

**Listing 7** Using prefetch

```
var A:[1..n] int;
on Locales[1] {
  var sum:int;
  // Optional warm up
  for i in 1..k do prefetch(A[f(i)]);
  for i in 1..n {
    if i+k <= n then prefetch(A[f(i+k)]);
    sum += A[f(i)]
  }
}
```

**Listing 8** Sequential PUTs

```
for i in 1..n do B[i] = compute(i);
```

In addition, they reduce portability since the non-blocking calls will not help with non-distributed programs.

Compare with the version using a prefetch hint shown in Listing 7. This program is much simpler because it does not need to keep track of buffers or communication handles. It is more portable because the prefetch hint is relevant whether or not the A array is distributed since it can prefetch into processor cache when the data is local. One issue is that the prefetch distance $k$ might depend on the distribution of the array A, the network, and upon the runtime. We believe that future work enabling adaptive prefetch or auto-tuning could vary $k$ at runtime to find the optimal value.

Communication overlap for PUTs is also easier with the cache. The cache automatically overlaps PUTs using write-behind without any program changes. In contrast, explicit non-blocking PUTs would have similar complexity to Listing 6 and suffer from the same portability problems.

### *D. Communication Aggregation*

Besides overlapping communication, a Chapel programmer might want to aggregate communications. We will explore the issues around performing aggregation in Chapel with two example programs.

In our first example, shown in Listing 8, `B` is a distributed array of integers, and `compute` is a computationally-intensive procedure. It might be more important to the program's performance to control where `compute` is run and so this program writes to `B` stored remotely. In the current Chapel implementation, the running task will wait for the network round-trip latency for each PUT to `B[i]` to complete.

Similarly, we might have a distributed array `A` that we wish to read to provide inputs for a computationally-intensive procedure `consume`, Listing 9 shows such an example. The read of `A[i]` performs a blocking GET and so waits for a network round-trip.

Both of these programs could straightforwardly be optimized using whole-array assignment as shown in Listing 10 and Listing 11. Although whole-array assignment aggregates communication in both cases, there are two problems with this approach. First, the solution listed here is not completely general since it might run out of memory. In particular, if A is a large distributed array, a copy of it may not fit on a single locale. Second, it detracts from the Chapel goal of separating algorithm from data distribution since this modification must be made in the algorithm code itself. We would prefer to have algorithm code that could perform well whether or not the arrays `A` and `B` are distributed.

Note that it is also possible to write versions of these programs using array slices to work with a subset of the

**Listing 9** Sequential GETs

```
for i in 1..n do consume(A[i]);
```

**Listing 10** Optimizing Sequential PUTs

```
var localB:[1..n] int;
for i in 1..n do localB[i] = compute(i);
B = localB;
```

array at a time, but such a modification significantly increases the complexity. It presents portability problems since the programmer has to specify what size chunk should be communicated at a time. Lastly, in our experiments, the array slice approach did not improve performance because slices are relatively heavyweight in the current implementation.

The cache for remote data is able to perform communication aggregation for the original programs in Listing 8 and Listing 9. It provides these simpler programs with communication optimization so that the increased complexity shown in Listing 10 and Listing 11 is not required to improve network performance.

## V. Experimental Evaluation

In our experiments with the cache, we sought the answers to several questions. What impact does the cache have to the overhead of GET operations? How effective are the write-behind and prefetch supported by the cache? How does the cache impact the performance of applications? Have improvements to compiler optimizations reduced the benefits of the cache?

We studied the performance of the cache on three systems. One is a Cray XC30™ system with approximately 50 nodes connected with the Aries network. Each node in this Cray XC system 128 GB of memory and dual 2.7 GHz Ivy Bridge E5-2697 processors. Each processor has 12 cores and 24 hardware threads. The second is a Cray CS400™ cluster system with approximately 200 nodes connected with an FDR InfiniBand network. Each node in the Cray CS400 system has 128 GB of memory and dual 2.3 GHz Haswell E5-2698 processors. Each processor has 16 cores and 16 hardware threads. The third system is a small cluster consisting of 10 nodes with dual 6-core Xeon X5670 at 2.93 GHz and 24 GiB of memory, connected with 1 Gigabit Ethernet and 10 Gigabit Ethernet.

Table I shows the evaluated GASNet [15] network configurations, including the GASNet GET and PUT latency for 8-byte blocking requests and the bandwidth for single 1024-byte blocking requests. On the Cray XC, we also measured performance with Chapel's native uGNI communication support [16]. This Cray uGNI support works very closely with the Aries network and is only available in the pre-built Chapel module for Cray XC and XE series systems. It is an alternative to GASNet configured with the Aries conduit.

**Listing 11** Optimizing Sequential GETs

```
var localA:[1..n] int = A;
for i in 1..n do consume(localA[i]);
```

Table I
NETWORKS STUDIED

| Network (conduit) | latency $\mu s$ for 8b GET | latency $\mu s$ for 8b PUT | bandwidth MB/s for 1024b GET | bandwidth MB/s for 1024b PUT |
|---|---|---|---|---|
| 1 Gb Ethernet (udp) | 49 | 49 | 11 | 11 |
| 10 Gb Ethernet (udp) | 16 | 16 | 52 | 53 |
| 56 Gb FDR InfiniBand (ibv) | 2.1 | 1.6 | 405 | 430 |
| 90+ Gb Aries [17, p. 21] (aries) | 1.7 | 1.3 | 415 | 632 |

Lastly, while we studied 4 different networks, we focus here on the high performance InfiniBand and Aries networks.

All benchmarks were compiled with the `--fast` flag to the Chapel compiler and we compared performance with and without the `--cache-remote` compiler flag.

To help understand the impact of compiler improvement on the cache, we evaluated two software versions in these experiments: v1.9+ is revision `5ba6639` which includes GASNet 1.22.0; v1.11+ is revision `6c635a1` which includes GASNet 1.24.0. The cache itself did not change significantly over this time period. Experiments with Ethernet were done with the older version. The cache initially only supports Chapel's `fifo` tasking layer and so we used that in these experiments. We used the `fast` segment in the GASNet configurations.

## A. *Synthetic Benchmarks*

We measured four synthetic benchmarks. The first of these benchmarks, copy, measures the performance of GETs and PUTs with a sequential access pattern. It sequentially copies $10,000$ integers stored contiguously in arrays, using a loop making individual array accesses. The second and third benchmarks, rand-puts and rand-gets use low-level routines to perform PUTs or GETs to unpredictable locations. In particular, each is hand optimized and performs $30,000$ GET or PUT operations within a remote array storing $10,000,000$ 64-bit integers. The fourth benchmark, prefetch, studies the impact of varying the prefetch distance $k$ in the prefetch example described in Section IV-C. All four of these benchmarks were run with two locales.

The left-hand side of Figure 1 shows the impact of the cache on the copy benchmark. This benchmark compiles down to at least 1 GET and 1 PUT per loop iteration. Since this benchmark represents a best-case scenario for the cache, enabling the cache gives good performance. Speedups range from $26\times$ for the Cray uGNI configuration to $98\times$ for the latest InfiniBand configuration. Note that the speedup is lower for the GASNet Aries and the Cray uGNI configurations because they were already more efficient: with the cache disabled, benchmark ran $1.7\times$ and $3\times$ faster

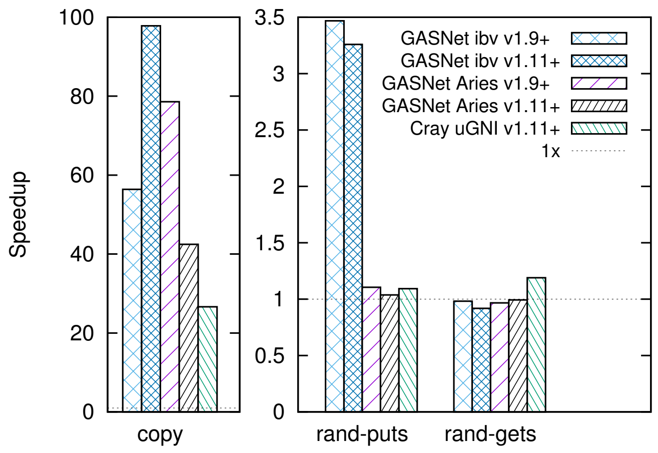

Figure 1. Synthetic Benchmark Performance

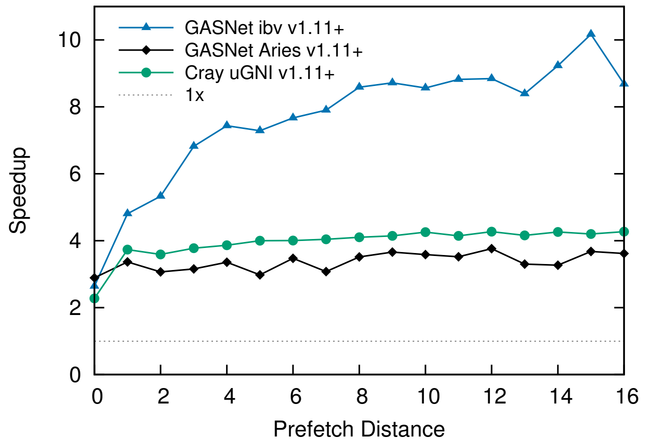

Figure 2. Prefetch Performance

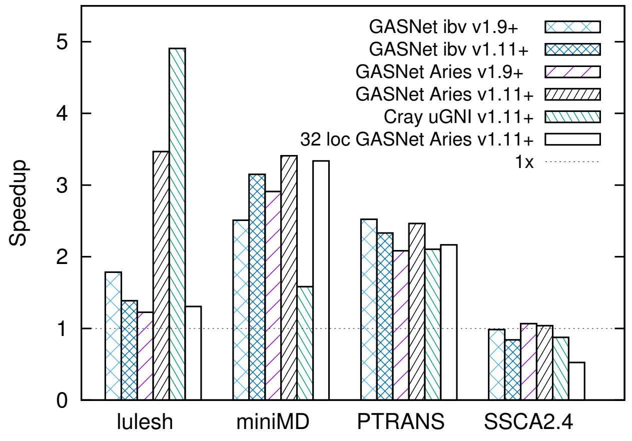

Figure 3. Application Performance

than the InfiniBand in these configurations. In addition, we observed 350× speedup for 1 Gb Ethernet and 480× speedup with 10 Gb Ethernet.

We created the rand-gets benchmark to measure the GET overhead of the cache. Since this benchmark is performing reads to random addresses, each access is not likely to be cached. Thus, we expect the cache to add overhead. We measured the cache hit rate at less that 0.1% which indicates that this benchmark does in fact measure the overhead of the cache. In the worst case, the cache added close to 10% overhead in the Cray uGNI and 1 Gb Ethernet configurations.

With the rand-puts benchmark, we sought to measure the performance benefit of write-behind without aggregation. If the cache is enabled, the task can perform PUTs without waiting for each one. We see about 3× speedup on InfiniBand and only modest speedups with the GASNet Aries and Cray uGNI configurations. Note that both of these Aries networking layers are currently performing more network ordering than is necessary: the PUTs with GASNet are ensuring local completion and the PUTs with Cray uGNI are currently blocking. For the Ethernet configurations, we observed significant speedup as with InfiniBand: 3.7× for 1 Gb Ethernet and 2.0× for 10 Gb Ethernet.

Figure 2 shows how performance of the prefetch example (Listing 7) varies with the prefetch distance $k$. Recall that this program accesses computed indices of an array. Here, it performed $30,000$ reads into a remote array of $10,000,000$ 64-bit integers. This benchmark gains 2-3× performance when the cache is enabled but the prefetch hint is not used (shown in the graph as prefetch distance 0) because array header accesses no longer require communication. We observed additional speedup for optimizing the prefetch distance. On InfiniBand, the best prefetch distance of 14 was approximately 4× faster than having the cache enabled and not prefetching at all. With GASNet on Aries, the best prefetch distance of 12 was 1.3× faster, and with Cray uGNI, the best distance of 12 was 1.9× faster. These improvements combine to make the program about 10× faster on InfiniBand and 4× faster with Aries. The Ethernet configurations were similar to InfiniBand for this benchmark and also showed a total 10× speedup.

## B. Application Benchmarks

To understand how the cache impacts the performance of applications, we experimented with four benchmark programs. We chose these benchmarks because they are well-known benchmarks that already existed as Chapel programs. They are available in the Chapel `examples/benchmarks` directory. We did not modify or tune these benchmarks. We ran all of the configurations with 8 locales and in addition measured the GASNet Aries configuration with 32 locales. Figure 3 shows the result. In this section, we will discuss only the results for v1.11+. We will discuss the impact of compiler improvement in the next section.

LULESH is the Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics benchmark [18]. We ran LULESH with the `sedov15oct.lmesh` input. LULESH showed good performance improvements from enabling the cache, including a 4.9× speedup in the Cray uGNI configuration, 3.5× for GASNet Aries, and 1.3× with InfiniBand. The 32-locale run still showed a 1.2× speedup.

MiniMD is a proxy application for molecular dynamics from Sandia's Mantevo group [19]. We compiled miniMD with `-snoRefCount -suseStencilDist=true -sdefaultDoRADOpt=false -sdisableAliasedBulkTransfer=false` and ran it with a problem size of 10. In most configurations, MiniMD had about a 3× speedup. That speedup continued when scaling to 32 locales. However, the Cray uGNI configuration showed less improvement - only 1.6× - because the Cray uGNI configuration was much faster than the other configurations: 3.2× faster than the GASNet Aries configuration. For the Cray uGNI configuration, the communication was already more efficient, and so there was less benefit to optimizing it.

PTRANS is a parallel matrix transpose benchmark [20]. We ran PTRANS with `--numrows=500 --rowBlkSize=3 --colBlkSize=5`. This benchmark showed good speedups around 2× in all configurations. As with miniMD, the Cray uGNI configuration ran significantly faster than the GASNet Aries configuration and so less speedup was available.

SSCA2 is from one of the HPCS benchmarks, Scalable Synthetic Compact Applications [21]. We analyzed kernel 4, which computes the betweenness centrality for all vertices. We used flags to disable the torus versions, to make the runs reproducible, and to enable timing statistics. We ran it with scale 8. SSCA2 kernel 4 is fundamentally different from the other applications we studied because it performs many atomic operations without doing much other work. Performance suffers in the InfiniBand, Cray uGNI, and 32 locales configurations because of the overhead of flushing the cache with each atomic operation. Work remains in optimizing the cache for such applications. In addition, it would help to use relaxed atomic operations to avoid the cache flushes.

## C. Impact of Compiler Improvements

The Chapel compiler improved significantly between v1.9+ and v1.11+. We used these improvements to try to understand the value of the cache even as the compiler performs better communication optimization.

First, note that the performance of the rand-puts and rand-gets benchmarks did not change significantly between v1.9+ and v1.11+. These benchmarks are hand-optimized and so do not leave much room for compiler optimization.

Figure 1 shows that the cache had a consistent impact for these benchmarks across these versions.

In contrast, the copy benchmark leaves quite a lot available for compiler optimization. Without the cache enabled, this benchmark became about $2.5\times$ faster for the InfiniBand configuration and $3.5\times$ faster for the GASNet Aries configuration between the studied compiler versions. The cache enabled large speedups for both versions.

For LULESH, the cache improved application run times even more after optimization improvements to the Chapel compiler between the tested versions. We ran the newest version of the LULESH benchmark with both versions of the compiler. The run-time without `--cache-remote` improved by about $3.2\times$ for the GASNet Aries configuration between v1.9+ and v1.11+. Improvements in applying loop invariant code motion in Chapel version 1.10 are the primary reason. As Figure 3 shows, the improved optimization allowed the cache to be even more effective: the speedup went from $1.2\times$ to $3.5\times$.

Lastly, we investigated how enabling experimental LLVM communication optimizations with `--llvm-wide-opt` changed the performance impact of enabling the cache. With the cache disabled, these LLVM optimizations gave LULESH a $1.2\times$ speedup, miniMD a $1.4\times$ speedup, and PTRANS a $1.7\times$ speedup in the GASNet Aries configuration. Even though the speedup from the cache was reduced, the cache still offered $3.5\times$, $2.5\times$, and $1.7\times$ speedups for LULESH, miniMD, and PTRANS respectively. In all of these cases, the fastest configuration combined the improved compiler optimization with the cache.

## VI. Related Work

Previous work with UPC includes HP UPC and MuPC (described by Zhang [22], [23]) and a class prototype for Berkeley UPC described in [24]. Like our work, these systems do not use explicit coherency messages but instead complete pending operations and invalidate the cache on a fence. However, our system caches both reads and writes (unlike HP UPC) and can support hardware RDMA (unlike MuPC).

The present work with Chapel allows for more latency hiding than other implementations with UPC. Zhang suggested that a major reason that the cache systems for UPC did not achieve wider adoption is that they increased throughput but did not help with latency [22]. Since Chapel supports tasks, the runtime can overlap communication with task execution. The Grappa [25] and GMT [26] projects have investigated this kind of latency hiding. In addition, the prebuilt Chapel module available on Cray systems has some capabilities in this regard [16].

Besides using write-behind to hide PUT latency, our work also includes an alternative means for GET latency hiding: a prefetch hint similar to processor prefetch instructions. For some applications with a programmer predictable access pattern, hints of what to prefetch can improve performance and cover the latency of the GET requests. To our knowledge, our work is the first time a PGAS language supported a prefetch hint for remote memory.

Several previous works make use of a cache in a PGAS library. DeSouza and Kalé describe Multiphase Shared Arrays which make use of programmer demarcated access patterns [27]. For example, in one phase of computation a distributed array is read-only; while in a second phase it is write-only. The MSA implementation takes advantage of these phases to avoid all cache invalidations and to buffer all writes. Memory consistency is handled during the large-scale barriers indicating phase change between parts of the computation. Larkins et al. discuss Global Trees which includes a caching strategy for tree nodes, with strict and relaxed consistency modes [28]. Lastly, Niu et al. investigated how a Global Arrays Read-Only Cache can improve the the scalability of quantum Monte Carlo applications [29]. In contrast to these approaches, our work focuses on a PGAS language and does not require program modification.

Marc Snir has proposed an alternative PGAS system with a cache. In his work, programs specify explicitly to which variables they will have exclusive, shared read-only, or transactional access [30]. In this way, a cache coherence protocol is embedded into the program itself. In contrast, the present work does not require that the original PGAS program be written with a caching system in mind.

Distributed shared memory research, including Dash, Plus, Munin, and TreadMarks [31], [32] include caches for remote data but have a more aggressive coherency protocol and do not have PGAS language integration.

Lastly, a compiler-supported software cache has been studied for the Cell processor in [33]. This work included compiler integration like our work, but had different constraints since it was working within a processor.

## VII. Future Work

There are many ways to build upon this work in order to improve communication performance of PGAS languages. First, there are many cache parameters that would benefit from further study, including the number of dirty pages, the number of intermediate nodes in the pointer tree, the size of each of the 2Q queues, the cache line size, the cache page size, and the size of the cache overall.

The implementation could be optimized further to reduce the overhead of repeated *acquire* and *release* operations and to improve the performance of large PUTs and GETs. In addition, the cache could be optimized for specific networks.

There are several ways that the prefetch capability could be improved. In some cases, compile-time optimizations could automatically add a prefetch hint in order to hide the latency of GET requests. In addition, the prefetch hint would benefit greatly from an adaptive prefetch mechanism.

Adaptive prefetch would automatically choose $k$ in Listing 7.

Finally, it would be interesting to evaluate this cache in the context of other PGAS languages and libraries that preserve sequential program order such as UPC [1], UPC++ [2], or Fortran 2008 [3].

## VIII. Conclusion

We have created a system that caches remote data for Chapel programs. We believe that this cache is a useful tool for programmers seeking good performance and productivity for applications in a distributed memory setting.

There are several ways in which the cache for remote data supports programmer productivity. First, programs do not need to be modified to take advantage of this cache since it works with Chapel's memory consistency semantics. Second, the design supports communication aggregation and overlap for straightforward programs, and it does not require that programs include explicit aggregation, or nonblocking communication calls. Lastly, we included a prefetch hint that functions in a manner similar to processor cache prefetch instructions and so should be familiar to performance-minded programmers. Programs using the prefetch hint are simpler than those using non-blocking GETs with operation handles.

Our evaluation of the of this cache shows that it has a promising impact on performance. It provided solid performance improvements on the order of 2× or better for most of the benchmarks. Lastly, the cache still provided these speedups after optimization significantly improved within the Chapel compiler.

## Acknowledgment

The authors would like to thank Greg Titus, Sung-Eun Choi, and Bradford Chamberlain from the Chapel team for their help in reviewing the cache implementation and this paper.

## References

[1] *UPC Language Specifications Version 1.3*, The UPC Consortium, November 2013, Lawrence Berkeley National Lab Tech Report LBNL-6623E. [Online]. Available: http://upc.lbl.gov/publications/upc-spec-1.3.pdf

[2] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick, "UPC++: A PGAS extension for C++," in *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium*. Washington, DC, USA: IEEE Computer Society, 2014, pp. 1105–1114.

[3] *Fortran 2008 Standard*, ISO/IEC Standard 1539-1:2010, October 2010.

[4] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, "Introducing OpenSHMEM: SHMEM for the PGAS community," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*. New York, NY, USA: ACM, 2010, pp. 2:1–2:3.

[5] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the Chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, Aug. 2007.

[6] B. L. Chamberlain, "Chapel," in *A Brief Overview of Parallel Programming Models*, P. Balaji, Ed. MIT Press, 2015 (expected).

[7] B. L. Chamberlain, S. J. Deitz, D. Iten, and S.-E. Choi, "User-defined distributions and layouts in Chapel: Philosophy and framework," in *Proceedings of the 2nd USENIX Conference on Hot Topics in Parallelism*. Berkeley, CA, USA: USENIX Association, 2010, pp. 12–12.

[8] B. L. Chamberlain, S. J. Deitz, D. Iten, S.-E. Choi, and V. Litvinov, "Authoring user-defined domain maps in Chapel," in *In CUG 2011*, 2011.

[9] M. Ferguson, S.-E. Choi, E. Ronaghan, and G. Titus, "The Chapel memory consistency model," CHIUW 2015 talk, 2015. [Online]. Available: http://chapel.cray.com/CHIUW/2015/hot-topics/01-Ferguson.pdf

[10] *C11 Standard*, ISO/IEC 9899:2001, C Standards Committee Std., December 2011.

[11] *C++11 Standard*, ISO/IEC 14882:2001, C++ Standards Committee Std., March 2014.

[12] S. V. Adve and H.-J. Boehm, "Memory models: A case for rethinking parallel languages and hardware," *Commun. ACM*, vol. 53, no. 8, pp. 90–101, Aug. 2010.

[13] D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*, 1st ed. Morgan & Claypool Publishers, 2011.

[14] T. Johnson and D. Shasha, "2Q: A low overhead high performance buffer management replacement algorithm," in *Proceedings of the 20th International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, pp. 439–450.

[15] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick, "Optimizing bandwidth limited problems using one-sided communication and overlap," in *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium*, April 2006.

[16] *Using Chapel on Cray Systems*, Chapel Developers, 2015. [Online]. Available: https://github.com/chapel-lang/chapel/blob/master/doc/release/platforms/README.cray

[17] B. Alverson, E. Froese, L. Kaplan, and D. Roweth, "Cray XC series network," Cray Inc., Tech. Rep. WP-Aries01-1112, 2012. [Online]. Available: http://www.cray.com/Assets/PDF/products/xc/CrayXC30Networking.pdf

[18] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. Devito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. H. Still, "Exploring traditional and emerging parallel programming models using a proxy application," in *Proceedings of the 27th IEEE International Symposium on Parallel and Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2013, pp. 919–932.

[19] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. N. Numrich, "Improving performance via mini-applications," Sandia National Laboratories, Tech. Rep. SAND2009-5574, September 2009. [Online]. Available: http://mantevo.org/MantevoOverview.pdf

[20] M. Berry, "Public international benchmarks for parallel computers: Parkbench committee: Report-1," *Scientific Programming*, vol. 3, no. 2, pp. 100–146, June 1994.

[21] D. A. Bader, J. Feo, J. Gilbert, J. Kepner, D. Koester, E. Loh, K. Madduri, B. Mann, and T. Meuse, "HPCS scalable synthetic compact applications #2 graph analysis," 2007. [Online]. Available: http://www.graphanalysis.org/benchmark/HPCS-SSCA2_Graph-Theory_v2.2.pdf

[22] Z. Zhang, "A performance model for Unified Parallel C," Ph.D. dissertation, Michigan Technological University, Houghton, MI, USA, 2006.

[23] Z. Zhang and S. Seidel, "A performance model for fine-grain accesses in UPC," in *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium*, April 2006.

[24] W. Chen, J. Duell, and J. Su, "A software caching system for UPC," 2003. [Online]. Available: http://crd-legacy.lbl.gov/~jcduell/cs265/project/UPC-cache.pdf

[25] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin, "Grappa: A latency-tolerant runtime for large-scale irregular applications," University of Washington, Tech. Rep. UW-CSE-14-02-01, 2014. [Online]. Available: http://sampa.cs.washington.edu/papers/grappa-tr-2014-02.pdf

[26] A. Morari, A. Tumeo, O. Villa, and M. Valero, "Scaling irregular apllications through data aggregation and software multithreading," in *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium*. Washington, DC, USA: IEEE Computer Society, 2014.

[27] J. DeSouza and L. V. Kalé, "MSA: Multiphase specifically shared arrays," in *Proceedings of the 17th International Workshop on Languages and Compilers for Parallel Computing*, West Lafayette, Indiana, USA, September 2004.

[28] D. B. Larkins, J. Dinan, S. Krishnamoorthy, S. Parthasarathy, A. Rountev, and P. Sadayappan, "Global Trees: A framework for linked data structures on distributed memory parallel systems," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 57:1–57:13.

[29] Q. Niu, J. Dinan, S. Tirukkovalur, L. Mitas, L. Wagner, and P. Sadayappan, "A global address space approach to automated data management for parallel Quantum Monte Carlo applications," in *Proceedings of the 19th International Conference on High Performance Computing*, December 2012, pp. 1–10.

[30] M. Snir, "Parallel Programming Language 1 (PPL1) v0.9 - draft," 2006. [Online]. Available: https://www.ideals.illinois.edu/bitstream/handle/2142/11217/Parallel%20Programming%20Language%201%20%28PPL1%29%20V0.9%20-%20Draft.pdf

[31] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "TreadMarks: Shared memory computing on networks of workstations," *Computer*, vol. 29, no. 2, pp. 18–28, Feb. 1996.

[32] B. Nitzberg and V. Lo, "Distributed shared memory: A survey of issues and algorithms," *Computer*, vol. 24, no. 8, pp. 52–60, Aug. 1991.

[33] M. Gonzàlez, N. Vujic, X. Martorell, E. Ayguadé, A. E. Eichenberger, T. Chen, Z. Sura, T. Zhang, K. O'Brien, and K. O'Brien, "Hybrid access-specific software cache techniques for the Cell BE architecture," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. New York, NY, USA: ACM, 2008, pp. 292–302.