# Automatic Adaptive Prefetching for Fine-grain Communication in Chapel

Thomas B. Rolinger
tbrolin@cs.umd.edu
University of Maryland
College Park, MD, USA

Alan Sussman
als@cs.umd.edu
University of Maryland
College Park, MD, USA

## ABSTRACT

Applications that operate on large, sparse graphs and matrices exhibit fine-grain irregular memory accesses patterns, leading to both performance and productivity challenges on today's distributed-memory systems. The Partitioned Global Address Space (PGAS) model attempts to address these challenges by combining the memory of physically distributed nodes into a logical global address space, simplifying how programmers perform communication in their applications. Chapel is an example of a programming language that implements a PGAS. However, while Chapel and the PGAS model can provide high developer productivity, the performance issues that arise from irregular memory accesses are still present. In this talk, we will discuss an approach to improve the performance of Chapel programs that exhibit fine-grain remote accesses while maintaining the high productivity benefits of the PGAS model. To achieve this goal, we designed and implemented a compiler optimization that performs *adaptive prefetching* for remote data. Specifically, the compiler performs static analysis to identify irregular memory access patterns to distributed arrays in parallel loops and then applies code transformations to prefetch remote data that will be needed in future loop iterations. Our approach is adaptive because the prefetch distance (i.e., how many iterations ahead to prefetch) is automatically adjusted as the program executes to ensure the prefetches are not issued too early or too late. Furthermore, the optimization is fully automatic and requires no user intervention. We demonstrate runtime speed-ups as large as 3.2x via adaptive prefetching when compared to unoptimized baseline implementations of different irregular workloads across three different distributed-memory systems.

## KEYWORDS

PGAS, Chapel, fine-grain communication, compiler optimizations, prefetching

## 1 INTRODUCTION

Applications that exhibit sparse and irregular memory access patterns have become a crucial component in today's data analytics workflows and scientific computing applications. Furthermore, many of these applications require the use of distributed-memory systems due to the large memory footprint of their input data. Using conventional two-sided communication models like MPI is often challenging for distributed-memory applications that exhibit irregular memory access patterns. This is because the access pattern is often not known until runtime due to its data dependent nature. The Partitioned Global Address Space (PGAS) model [7] is one approach that can address the productivity challenges that distributed irregular applications face by providing a shared-memory style of programming on a distributed-memory system. Under the PGAS model, all of the physically distributed memory of a system can be viewed logically as a single global address space, where each compute node "owns" a partition of the address space. Chapel [4] is an example of a programming language that implements a PGAS.

However, while high developer productivity can be achieved via the PGAS model and Chapel, the performance of irregular applications is often hindered by *fine-grain remote communication* [5, 10–12], which is a direct result of the shared-memory style of communication provided by the PGAS model. Fine-grain remote communication arises from irregular memory accesses to distributed data and is at odds with modern high performance network interconnects, like Infiniband, which provide the best performance when sending fewer but larger messages. One solution to improve the performance of PGAS/Chapel programs that exhibit irregular memory access patterns is to apply optimizations. However, because irregular memory access patterns cannot be inferred at compile time, the task of applying optimizations is often left entirely to the programmer. This can dramatically reduce developer productivity by requiring additional lines of code to be added to the program, which leads to a significant increase in the amount of time required to write the program. Additionally, applying such optimizations often requires expert knowledge of irregular optimizations and an in-depth understanding of the specific system/architecture that the program is executing on.

In this talk, we will present an approach to improve the runtime performance of irregular applications in Chapel by automatically applying an optimization via the compiler. By applying the optimization automatically, programmers are not required to modify their code. Instead, the optimization is enabled by simply turning on a compiler flag. The optimization performs *adaptive prefetching* for irregular memory access patterns present in `forall` loops. Specifically, static analysis is used within the compiler to identify irregular memory access patterns of the form **A[B[i]]** where **A** is a distributed array. Then, transformations are applied to insert the necessary code to prefetch remote data that will be needed in future loop iterations. The optimization is adaptive because the prefetch distance (i.e., how many loop iterations ahead to prefetch) is automatically adjusted as the program executes. This ensures that the prefetches are not being issued too early or too late, which can degrade performance. We demonstrate that the performance of different irregular workloads can be improved by as much as 3.2x across three different systems. We also provide insight into the effectiveness of adapting the prefetch distance throughout the program's execution by conducting experiments where the distance is held at a fixed value.

## 2 OPTIMIZATION APPROACH

A performance issue specific to fine-grain remote reads (i.e., `gets`) is that they are usually performed in a blocking manner, which prevents tasks from proceeding with computation until the data has arrived over the network. One approach to address this issue is to hide communication latency by *prefetching* the remote data that will be needed in the future, where these prefetches are non-blocking and useful computation is performed while the prefetch is being executed. Ideally, once that data is needed by a task, the prefetch has completed and the data will be accessible at a lower cost. Lower cost means that the prefetched data is presumably moved into some sort of cache memory that is physically closer to the task that needs the data. Prefetching has been widely used in both hardware [6, 14–16] and software [1–3]. In this talk, we focus on software prefetching since it is more easily leveraged for distributed-memory systems and irregular memory access patterns.

In this section, we provide an overview of Chapel's remote cache and our adaptive remote prefetching optimization that is integrated into Chapel's compiler and leverages the remote cache. The optimization uses static analysis to identify candidate memory accesses to prefetch and then performs code transformations to enact the prefetches. The prefetch distance is adjusted automatically as the program executes to improve the timeliness of the prefetches. Adaptive software prefetching approaches have been studied in the past [9, 13], but our approach specifically targets remote memory accesses within the PGAS model.

### 2.1 Chapel's Remote Cache

Chapel's remote data cache is a software write-back cache with dirty bits, local memory consistency operations and programmer-guided prefetches. As the name suggests, the remote cache is designed to store remote data. The initial design and implementation of the remote cache was introduced by Ferguson and Buettner [8], and has since been improved and integrated into Chapel's runtime[1]. Their design of the remote cache is not necessarily specific to Chapel, as it considers the underlying `puts` and `gets` that are ultimately issued by a PGAS program. As such, the high level design of our adaptive remote prefetching optimization is not specific to Chapel.

Each compute core on a locale (i.e., node) has its own remote data cache that operates independently from the other cores/caches. As a task in Chapel is ultimately mapped to a compute core on a locale, these caches can be considered to be per task. Each cache entry corresponds to a 1024 byte cache page, and each cache line is 64 bytes. Remote reads are automatically rounded up to fill cache lines, and remote writes to nearby data will be aggregated, if possible. However, as with traditional CPU caches, there are limitations to the benefits that can be provided by the remote cache for irregular data access patterns that exhibit poor spatial and temporal locality. Additionally, Chapel provides a low-level interface to perform prefetches for the remote cache, which issue non-blocking reads from specific remote addresses. These prefetches form the basis for our adaptive prefetching optimization.

[1]The remote cache is enabled by default for all programs since version 1.24 of Chapel.

### 2.2 Static Analysis

The static analysis for the adaptive remote prefetching optimization has the following three steps: (1) identifying candidate memory access patterns in `forall` loops, (2) determining whether the loop that contains the candidate has a valid form and (3) detecting whether the data to be prefetched is a record and if so, which fields will be accessed. The details of each of these steps will be described below.

*2.2.1 Identifying Candidate Memory Access Patterns.* Static analysis begins by identifying potential candidates for prefetching, where these candidates must be contained inside of `forall` loops. However, the candidate can be nested within a `for` loop that is itself nested in a `forall` loop. The optimization specifically targets irregular reads and looks for indirect access patterns of the form **A[B[i]]**, where **A** and **B** are arrays and **A** is a distributed array. More generally, the optimization operates on distributed arrays that are accessed by an expression that involves another array access. As prefetching attempts to look ahead some number of loop iterations, the compiler must be able to guarantee that the prefetches will be issued to valid memory locations. To address this restriction, the loop index variable for the loop that contains the prefetch candidate must be used within the array access expression. If the candidate access does not use the loop index variable, then it cannot be prefetched because it is not known how the access pattern progresses with respect to the loop iterations. Furthermore, static analysis must ensure that any other variables used within the candidate's access expression can be reasoned about with respect to how the loop progresses. As a result, such variables are required be read-only within the scope of the loop body, which greatly simplifies the analysis that is required to ensure the prefetches are issued to valid array locations.

*2.2.2 Determining Valid Loop Forms.* A majority of the static analysis performed for the prefetching optimization involves ensuring that the loop that contains the prefetch candidate has a valid form. In this case, valid means that the compiler can statically reason about how the loop iterations progress. This is important for prefetching because the optimization needs to ensure that prefetches will not be issued to invalid memory addresses (e.g., beyond the end of the array). A valid loop has one of the following forms: (1) the loop iterates over an array (`forall elem in Arr`), (2) the loop iterates over a domain (`forall i in Arr.domain`), or (3) the loop iterates over a range (`forall i in 0..4`). For each of these three forms, the optimization also supports loops with explicit strides (`forall i in 0..4 by 2`). Such forms are required because they enable the optimization to determine the first and last value of the loop index, the stride of the loop and the number of loop iterations. This information is crucial for the code transformations that will be discussed in Section 2.3.

*2.2.3 Detecting Record Field Accesses.* A particular use-case that appears often in irregular Chapel programs is performing a remote read like **A[B[i]]** where **A** stores records (i.e., C structs). In this case, different fields of the record could be accessed in subsequent statements. This complicates prefetching because a single prefetch is not guaranteed to fetch all of the accessed fields if they are not adjacent to each other in the record's definition. While the number of bytes to prefetch can be specified when issuing the prefetch, it is not practical to do so in some cases where there are many fields (i.e., bytes) between the desired fields. A better approach is to issue separate prefetches for each field that is accessed in the loop. To enable separate prefetches, static analysis is used to detect whether **A** stores records, and which fields are accessed in the loop. Once it is determined that **A** stores records, the static analysis will look for any expression that uses the retrieved element and determine which field is accessed. Interprocedural analysis is used to identify field accesses where the retrieved element is passed in as an argument to a procedure. A list of these fields is stored internally within the compiler and used during the code transformation to generate the prefetch calls.

## 2.3 Code Transformations

The code transformations performed by the compiler for the adaptive remote prefetching optimization have the following steps: (1) creating prefetch-related variables, (2) creating compile time checks, (3) creating the prefetch distance adjustment check and (4) creating the prefetch procedure call. Listing 1 presents an example `forall` loop that contains a valid prefetch candidate (line 3), followed by code that is equivalent to the output of the code transformations[2]. As each core on a locale has its own remote cache, prefetching is performed per-task, resulting in each task receiving its own prefetch distance that is adjusted independently from the other tasks created by the `forall` loop. In this section, `B.domain` and `i` from Listing 1 will be referred to as the *loop iterand* and *loop index variable*, respectively.

```
1  // Original forall loop
2  forall i in B.domain {
3    C[i] = A[B[i]];
4  }
5
6  // Transformed forall loop
7  reset_prefetch_counters();
8  const last = getLastLoopIndex(B.domain);
9  const first = getFirstLoopIndex(B.domain);
10 const stride = getLoopStride(B.domain);
11 const adjIters = getAdjIters(first, last, stride);
12 forall i in B.domain with (var d=8, var cnt=0) {
13   if isPrefetchSupported(B.domain, A, B) {
14     if cnt < adjIters {
15       cnt += 1;
16     }
17     else {
18       adjustPrefetchDistance(d);
19       cnt = 0;
20     }
21     if (i + (d * stride)) <= last {
22       prefetch(A[B[i+(d*stride)]]);
23     }
24   }
25   C[i] = A[B[i]]; // original access
26 }
```

**Listing 1: Example of a prefetch candidate and the resulting code transformations applied by the compiler. `C[i] = A[B[i]]` is the prefetch candidate.**

[2]We do not perform source-to-source compilation. The code transformations modify the internal representation of the program before the executable is created

*2.3.1 Create Prefetch Variables.* The first step for the code transformations is to create several variables that are used to compute the bounds of the prefetches and when to perform the prefetch distance adjustments. These variables are on lines 8–11 in Listing 1. All of these variables are declared as `const`, since they do not change throughout the loop iterations, and they rely on extracting information from the loop's iterand at runtime via Chapel procedures that were written for this optimization. Rather than explaining their definitions and uses up front, it will be easier to provide details within the context of their use throughout the code transformations below.

*2.3.2 Create Compile Time Check.* Line 13 in Listing 1 performs a compile time check to ensure that prefetching is valid. The procedure `isPrefetchSupported()` returns true if the loop iterand is not an associative domain/array and if `A` and `B` are arrays. Associative arrays/domains do not store their indices in any particular order, since they function more or less like dictionaries/maps. Therefore, prefetches cannot be issued with respect to a loop that is yielding seemingly random values for the loop index variable (i.e., the optimization cannot infer the progression of the loop index variable). As this check is performed at compile time, the compiler will automatically remove the branch that is not taken. In other words, if prefetching cannot be supported, then all of the code transformations that appear within the then-branch of the if statement will be removed.

*2.3.3 Create Prefetch Distance Adjustment Check.* Lines 14–20 in Listing 1 perform the check that determines when to adjust the prefetch distance. Each task executing the `forall` loop is given its own variables `cnt` and `d` (line 12). The `cnt` variable is initialized to 0 and is used to determine when to adjust the task's prefetch distance `d`, which is initialized to 8. The choice of the initial prefetch distance value is important for non-adaptive approaches, but our optimization will adjust the distance as the program executes. Therefore, an initial value of 8 is chosen as a reasonable starting point (i.e., not too small or too large). For each loop iteration that a task executes, it increments `cnt` (line 15) until it reaches the value of `adjIters`, at which point the optimization will adjust the prefetch distance (line 18). The actual calculation of `adjIters` will be described in Section 2.4, but it relies on knowing the number of iterations that the loop will execute, which itself relies on knowing the first and last valid value of the loop index variable and the stride of the loop. These are extracted by the procedure calls on lines 8–10 in Listing 1.

*2.3.4 Create Prefetch Call.* The final step of the code transformations is to generate the call to actually perform the prefetching, as well as the bounds check to ensure that the prefetch will not be beyond the bounds of the array. One way to perform bounds checking is to check whether the desired index is contained within the array's domain, which can be performed via the `.contains()` method on the array's domain. However, this is unnecessarily expensive. A better approach is to simply check whether the desired index is greater than the last valid value of the loop index variable. Assuming that every value that the loop index variable takes on throughout the iterations yields a valid array access (which is the responsibility of the programmer), this check is less costly. Line 21 in Listing 1 performs this bounds check. After the bounds check, the prefetch call itself can be constructed (line 22), which takes in the address/element to prefetch and then issues a call to Chapel's runtime to perform the remote cache prefetch. Finally, if the static analysis determines the target array `A` stores records, then the compiler will transform the code to issue prefetches specifically to the record fields that are accessed inside the loop. It is worth noting that the index into the target array (e.g., `B[i]` in Listing 1) is not explicitly prefetched by our optimization. Chapel's remote cache will automatically read ahead for this type of sequential access pattern.

*2.3.5 Modifying Loop Iterator.* While not demonstrated in Listing 1, our optimization can also correctly handle `forall` loops that iterate over arrays, such as `forall elem in Arr`. In order to do so, the loop iterator must be modified to ensure that it yields a loop index variable that can be used for prefetching. In the case of iterating over an array, the goal of prefetching is to "compute" the address for a future element of the array that the loop will yield. However, this cannot be done by offsetting `elem` by the prefetch distance. Instead, we need to determine the corresponding index of `elem` within `Arr` and offset that by the prefetch distance. In order to do this, the compiler will modify the `forall` loop's iterator to yield both `elem` and its index in `Arr`. With this index, proper prefetches can be issued.

## 2.4 Adjusting Prefetch Distance

Beyond static analysis and code transformations, a crucial component of the adaptive prefetching optimization is adjusting the prefetch distance(s) as the loop executes. There are several motivating reasons for automatically adjusting the prefetch distance. First, the "best" prefetch distance is determined by a variety of factors that change across different systems, applications and even input data. Examples of these factors include the latency of the system interconnect, the nature of the irregular memory access patterns in the application and the underlying sparsity structure of the input data. Additionally, within a single execution of an application on a system, the prefetch distance that provides the best performance may vary depending on how the memory access pattern changes over time. Finally, within Chapel each core/task has its own remote cache, and therefore its own prefetch distance. Even on systems with a modest number of nodes and cores per node, this results in an overwhelming number of distances to manage by hand.

To address these issues, our optimization employs a heuristic that adjusts the prefetch distances based on the observed timeliness of the prefetches. This is accomplished by adding per-core counters to Chapel's remote cache that keep track of the number of prefetches issued, how many were late (not completed by the time the data was needed) and how many were early (evicted from the cache before being used). The existing `CommDiagnostics` counters for these metrics are not sufficient because they are per-locale and atomic, resulting in unacceptable overhead when the program issues a significant number of prefetches[3].

There are two main considerations for the prefetch distance adjustment heuristic: (1) how often to adjust the distances, and (2) how to adjust the distances. For (1), this is determined by the calculation of `adjIters` on line 11 in Listing 1. The procedure `getAdjIters()` considers how many iterations of the loop will be assigned to each task and then computes a fraction of those iterations using a tunable parameter referred to as the *sample ratio*. Through sensitivity studies across different workloads and systems, we found a sample ratio of 0.01 performs best. This means that if a task is assigned 10,000 iterations then it will attempt to adjust its prefetch distance every 100 iterations.

With respect to (2), the heuristic looks exclusively at the percentage of prefetches that were marked as late. If this percentage is above a threshold, then the prefetch distance is increased by one to allow for more time to elapse between the prefetch call and the data access. If the percentage is below the threshold, and has remained as such for a specified number of adjustment intervals (i.e., calls to `adjustPrefetchDistance()`), then the distance is decreased by one. This prevents the prefetches from being issued too early, which can lead to cache pollution. The late prefetch tolerance and the number of adjustment intervals before decreasing the distance are also tunable parameters that we chose via sensitivity studies. Their values were chosen as 10% and 8, respectively.

Focusing on late prefetches rather than early prefetches allows for more accurate adjustments to the prefetch distance. This was noted by Heirman et al. [9], which is the work we based our adjustment heuristic on. A late prefetch still represents a useful prefetch, in the sense that some of the communication latency is hidden. Furthermore, a small adjustment to the distance is likely to improve the timeliness of the prefetch. On the other hand, early prefetches are harder to respond to without specific information about *how* early the prefetches were. Instrumenting the remote cache to provide that level of information would be difficult and likely require the use of timestamps.

## 3 PERFORMANCE EVALUATION

To evaluate the performance of our adaptive prefetching optimization, we implemented a set of irregular workloads in Chapel, namely Single Source Shortest Path (SSSP), PageRank, Sparse Matrix Vector Multiply (SpMV) and index gather [4]. All of these workloads are written using the high productivity features of Chapel/PGAS. We execute these workloads on three different systems, each with a different network interconnect and different CPUs/number of cores.

Our results show that runtime performance improvements as large as 3.2x can be achieved on these workloads across the different systems, demonstrating the effectiveness of the optimization as well as its portability across different hardware platforms. Furthermore, since the optimization is applied automatically via the compiler, developer productivity is drastically improved by not requiring any modifications to the programs. Additionally, we compared the performance of the adaptive optimization versus choosing an initial distance and keeping it fixed throughout the program. We found that performance when not adapting the distance can be up to 1.44x slower when compared to adapting the distance. In regards to compilation time, we do not observe any noticeable overhead due to our static analysis and code transformations.

[3]It is noted that the first author of this proposal implemented and contributed the per-locale atomic counters to Chapel in preparation for this optimization. However, they had to be modified as described to be practical for the optimization.

[4]https://github.com/jdevinney/bale

## REFERENCES

[1] Sam Ainsworth and Timothy M Jones. 2017. Software prefetching for indirect memory accesses. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, Austin, TX, USA, 305–317.

[2] Abdel-Hameed Badawy, Aneesh Aggarwal, Donald Yeung, and Chau-Wen Tseng. 2004. The efficacy of software prefetching and locality optimizations on future memory systems. *Journal of Instruction-Level Parallelism* 6, 7 (2004).

[3] David Callahan, Ken Kennedy, and Allan Porterfield. 1991. Software prefetching. *ACM SIGARCH Computer Architecture News* 19, 2 (1991), 40–52.

[4] Bradford L Chamberlain, Elliot Ronaghan, Ben Albrecht, Lydia Duncan, Michael Ferguson, Ben Harshbarger, David Iten, David Keaton, Vassily Litvinov, Preston Sahabu, et al. 2018. Chapel Comes of Age: Making Scalable Programming Productive. https://cug.org/proceedings/cug2018_proceedings/includes/files/pap130s2-file1.pdf

[5] Wei-Yu Chen, C. Iancu, and K. Yelick. 2005. Communication optimizations for fine-grained UPC applications. In *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*. 267–278. https://doi.org/10.1109/PACT.2005.13

[6] Fredrik Dahlgren and Per Stenstrom. 1996. Evaluation of hardware-based stride and sequential prefetching in shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems* 7, 4 (1996), 385–398.

[7] Mattias De Wael, Stefan Marr, Bruno De Fraine, Tom Van Cutsem, and Wolfgang De Meuter. 2015. Partitioned Global Address Space Languages. *ACM Computing Surveys (CSUR)* 47, 4 (2015), 1–27.

[8] M. P. Ferguson and D. Buettner. 2015. Caching Puts and Gets in a PGAS Language Runtime. In *2015 9th International Conference on Partitioned Global Address Space Programming Models*. IEEE, Washington, DC, USA, 13–24.

[9] Wim Heirman, Kristof Du Bois, Yves Vandriessche, Stijn Eyerman, and Ibrahim Hur. 2018. Near-Side Prefetch Throttling: Adaptive Prefetching for High-Performance Many-Core Processors. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques* (Limassol, Cyprus) *(PACT '18)*. Association for Computing Machinery, New York, NY, USA, Article 28, 11 pages. https://doi.org/10.1145/3243176.3243181

[10] Thomas B. Rolinger, Joseph Craft, Christopher D. Krieger, and Alan Sussman. 2021. Towards High Productivity and Performance for Irregular Applications in Chapel. In *2021 SC Workshops Supplementary Proceedings (SCWS)*. IEEE, St. Louis, MO, USA, 1–11. https://doi.org/10.1109/SCWS55283.2021.00012

[11] Thomas B. Rolinger, Christopher D. Krieger, and Alan Sussman. 2021. Runtime Optimizations for Irregular Applications in Chapel. *The 8th Annual Chapel Implementers and Users Workshop (CHIUW'21)*.

[12] Thomas B. Rolinger and Alan Sussman. 2022. Compiler Optimization for Irregular Memory Access Patterns in PGAS Programs. In *Languages and Compilers for Parallel Computing*. to appear.

[13] R.H. Saavedra and Daeyeon Park. 1996. Improving the effectiveness of software prefetching with adaptive executions. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Technique*. IEEE, Boston, MA, USA, 68–78. https://doi.org/10.1109/PACT.1996.552556

[14] Alan Jay Smith. 1982. Cache memories. *ACM Computing Surveys (CSUR)* 14, 3 (1982), 473–530.

[15] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N Patt. 2007. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. IEEE, 63–74.

[16] Nishil Talati, Kyle May, Armand Behroozi, Yichen Yang, Kuba Kaszyk, Christos Vasiladiotis, Tarunesh Verma, Lu Li, Brandon Nguyen, Jiawen Sun, et al. 2021. Prodigy: Improving the memory latency of data-indirect irregular workloads using hardware-software co-design. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 654–667.