ABSTRACT

Title of Dissertation:	COMPILER OPTIMIZATIONS FOR IRREGULAR MEMORY ACCESS PATTERNS IN THE PGAS PROGRAMMING MODEL
	Thomas B. Rolinger Doctor of Philosophy, 2023
Dissertation Directed by:	Professor Alan Sussman Department of Computer Science

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. However, while the PGAS model can provide high developer productivity, the performance issues that arise from irregular memory accesses are still present. This dissertation aims to bridge the gap between high productivity and high performance for irregular applications in the PGAS programming model.

To achieve that goal, I designed and implemented COPPER, a framework that performs Compiler Optimizations for Productivity and PERformance. COPPER automatically performs static analysis to identify irregular memory access patterns to distributed data within parallel loops, and then applies code transformations to perform optimizations at runtime. These optimizations perform small message aggregation, adaptive prefetching and selective data replication. Furthermore, they are applied without requiring user intervention, thereby improving performance and developer

productivity. I demonstrate the capabilities of COPPER by implementing it within the Chapel parallel programming language and conducting performance evaluations across a variety of irregular workloads and hardware platforms. These evaluations show that COPPER can achieve runtime speed-ups of 1.08 – 87x for small message aggregation, 0.78 – 3.2x for adaptive prefetching and 1.2 – 444x for selective data replication.

COMPILER OPTIMIZATIONS FOR IRREGULAR MEMORY ACCESS PATTERNS IN THE PGAS PROGRAMMING MODEL

by

Thomas B. Rolinger

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2023

Advisory Committee:
Professor Alan Sussman, Chair/Advisor
Professor Donald Yeung, Dean's Representative
Professor Leonidas Lampropoulos
Professor Abhinav Bhatele
Professor Laxman Dhulipala
Professor Michelle M. Strout

© Copyright by
Thomas B. Rolinger
2023

Acknowledgements

First, I would like to thank my committee members for taking the time to review this dissertation. I would like to thank my advisor, Alan Sussman, for his continued support, encouragement, patience and guidance throughout the work that is represented in this dissertation. Additionally, Chris Krieger was instrumental in my early career at the Laboratory for Physical Sciences and if it were not for his advice, I most likely would not have started my PhD or found the particular area of research that I am involved with today. Both Chris and Alan have been co-authors on a majority of the work that is represented in this dissertation, so for that I would like to thank them for their efforts; I believe that the work has only been as successful as it has been with their help.

I would also like to thank the Chapel team for their support during my involvement with the Chapel programming language. Specifically, Vass and Engin for getting me started within Chapel's compiler infrastructure (especially Vass who would answer my late night questions without complaining), Elliot for helping me with runtime and performance issues, Michael for answering my questions about Chapel's remote cache and Brad for providing overall expertise within Chapel. I would also like to thank Michelle for facilitating access to the Cray XC system used throughout this dissertation.

The people and facilities at the Laboratory for Physical Sciences (LPS) played an important role during my PhD. First off, they not only allowed but supported

my desire to pursue a PhD while working full time. Furthermore, I have been able to use the high performance computing resources available at LPS to conduct my performance evaluations.

Last, I would like to thank my family for their unconditional support. Specifically, my wife for listening to hours and hours of my ideas, frustrations and overall ramblings for the past 6 years, my brother for guiding me into computer science many years ago, and my mom, dad and sisters for their encouragement. And also my three cats for their much needed interruptions when working from home (I neither confirm nor deny that some portion of this dissertation was inadvertently authored by them walking across my keyboard).

Table of contents

Acknowledgements **ii**

Table of contents **iv**

1 Introduction **1**
- 1.1 Problem Statement 4
- 1.2 Thesis Statement 4
- 1.3 COPPER: Compiler Optimizations for Productivity and Performance . 5
  - 1.3.1 Purpose and Design Features 5
  - 1.3.2 Optimizations 7
- 1.4 Performance Evaluations Across Systems 12
- 1.5 Comparison to Prior Work 12
- 1.6 Summary 13

2 Background **16**
- 2.1 Defining and Measuring Developer Productivity 16
- 2.2 Irregular Memory Access Patterns 18
- 2.3 Partitioned Global Address Space Model 20
- 2.4 The Chapel Parallel Programming Language 24
  - 2.4.1 Design Philosophy and Goals 24
  - 2.4.2 Syntax for Parallel and Distributed Programming 26

3 Applications and Kernels **34**
- 3.1 Graph Analytics 35
  - 3.1.1 Breadth First Search 36
  - 3.1.2 Single Source Shortest Path 40
  - 3.1.3 K-core Decomposition 43
  - 3.1.4 PageRank 47
  - 3.1.5 Triangle Counting 51
  - 3.1.6 Other Graph Analytics 54
- 3.2 Scientific Computing 55
  - 3.2.1 Conjugate Gradient 56
  - 3.2.2 Moldyn 60
- 3.3 Mini-kernels 62
  - 3.3.1 Histogram 63

3.3.2 Graph Construction . . . 65

4 COPPER: Compiler Optimizations for Productivity and Performance 70
4.1 Design . . . 70
4.2 Implementation . . . 72
4.2.1 Existing Chapel Compiler Passes . . . 73
4.2.2 COPPER's Workflow . . . 77

5 Remote Write Aggregation 83
5.1 Approach . . . 84
5.1.1 Static Analysis . . . 85
5.1.2 Code Transformations . . . 91
5.2 Performance Evaluation . . . 96
5.2.1 Experimental Setup . . . 97
5.2.2 Results . . . 98
5.2.3 Summary of Results . . . 105
5.3 Limitations . . . 105
5.3.1 Additional Operations . . . 106
5.3.2 Memory Usage . . . 107
5.4 Related Work . . . 107

6 Adaptive Remote Prefetching 110
6.1 Chapel's Remote Data Cache . . . 112
6.2 Approach . . . 113
6.2.1 Static Analysis . . . 114
6.2.2 Code Transformations . . . 118
6.2.3 Prefetch Distance Adjustment . . . 125
6.3 Performance Evaluation . . . 131
6.3.1 Experimental Setup . . . 132
6.3.2 Results . . . 132
6.3.3 Summary of Results . . . 144
6.4 Limitations . . . 145
6.4.1 Multiple Prefetch Streams . . . 145
6.4.2 Selecting Thresholds . . . 146
6.4.3 Branch-dependent Prefetches . . . 147
6.5 Related Work . . . 148

7 Selective Data Replication 151
7.1 Approach . . . 153
7.1.1 High Level Inspector-Executor Approach . . . 154
7.1.2 Static Analysis . . . 155
7.1.3 Code Transformations . . . 163
7.1.4 Supporting Writes to Replicated Data . . . 171
7.2 Performance Evaluation . . . 173
7.2.1 Experimental Setup . . . 173

7.2.2 Results . . . 174
7.2.3 Summary of Results . . . 183
7.3 Limitations . . . 183
7.3.1 Multiple Candidates in the Same Loop . . . 183
7.3.2 Memory Usage Increases . . . 184
7.3.3 Overlapping Inspector and Executor . . . 185
7.3.4 Accurate Prediction of Optimization Profitability . . . 187
7.4 Related Work . . . 188

8 Interoperability Between Optimizations 191
8.1 Approach . . . 192
8.1.1 Single Candidate for Multiple Optimizations . . . 192
8.1.2 Multiple Candidates in the Same Loop . . . 193
8.2 Performance Evaluation . . . 195
8.2.1 Experimental Setup . . . 195
8.2.2 Results . . . 196

9 Performance and Productivity Portability Across Systems 201
9.1 Description of Systems . . . 202
9.2 Experiments and Results . . . 203
9.2.1 Remote Data Aggregation: BFS and SSSP . . . 203
9.2.2 Adaptive Remote Prefetching: SSSP and PR . . . 205
9.2.3 Selective Data Replication: NAS-CG . . . 210
9.3 Summary of Results . . . 211

10 Conclusions and Future Work 213

Bibliography 219

Chapter 1

Introduction

Applications that exhibit sparse and irregular memory access patterns have become a crucial component in today's data analytics workflows. Such applications include graph analytics [Pea17], tensor factorization [Smi+15; RSK19] and even some aspects of machine learning [Gal+20]. Furthermore, several scientific computing applications are characterized by irregular memory accesses, including molecular dynamics simulations [MWK99; BE06], solving partial differential equations (PDEs) [GZ99] and sparse linear solvers [Dav06]. Rather than issuing sequential or linear memory accesses, which can often be predicted and optimized for traditional cache-memory hierarchies, irregular applications exhibit indirect access patterns with little to no spatial and temporal locality. Furthermore, these irregular access patterns are data dependent and not known at compile time, which hinders static optimizations. As a result, achieving high performance for irregular applications is a major challenge. To illustrate this, consider systems ranked in the TOP500 List [1]. The TOP500 benchmark measures the performance that a system achieves when performing dense linear algebra, which does not exhibit any irregular memory access patterns. While the top ranked system in November 2022, Frontier, achieved 65% of its theoretical maximum floating-point performance for the TOP500 benchmark, it only achieved a mere 0.8%

[1] https://www.top500.org/lists/top500/2022/11/

of its theoretical maximum performance when running the High Performance Conjugate Gradient (HPCG) benchmark [2], which is dominated by irregular memory access patterns.

To complicate matters, many irregular applications operate on large, sparse data sets that can exceed the memory capacity of a single server. As a result, distributed-memory systems and programming models are often required to execute irregular workloads, adding an additional layer of complexity to achieving high performance. Specifically, irregular memory accesses to distributed data can lead to *fine-grain remote communication*, where many small messages (often only 8 bytes in size) are sent across the network to access data. This is at odds with modern high performance network interconnects, like Infiniband, which provide the best performance when sending fewer but larger messages.

Developer productivity, which is the amount of effort required to write a program that achieves some goal (e.g., good performance, correct results, etc.), is also a challenge for these distributed workloads. Because irregular memory access patterns are not known until runtime, it is difficult for programmers to orchestrate the necessary communication. The Partitioned Global Address Space (PGAS) model [De +15] is one approach that can address the productivity challenges that distributed irregular applications face. 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. Through this abstraction, remote data is accessed via *one-sided* communication calls, which provide a shared-memory style of programming on a distributed-memory system. Rather than use sends and receives like in the two-sided Message Passing Interface (MPI), the PGAS model provides abstractions for *gets* and *puts*, which are one-sided remote reads and writes, respectively. When a process performs a `get` to read some remote data, the

[2] https://www.top500.org/lists/hpcg/2022/11/

owner of that data does not need to be aware, programmatically, that another process needs the data. Shared-memory implementations of programs can often be adapted to distributed-memory systems via the PGAS model by simply declaring the relevant arrays as distributed, letting the PGAS model handle the required low level details regarding communication. Languages and libraries that implement the PGAS model include UPC [El-+05], GlobalArrays [NHL96] and Chapel [Cha+18].

However, while high developer productivity can be achieved via the PGAS model, *the performance of irregular applications is often hindered by fine-grain remote communication* [CIY05; RKS21b; Rol+21], which is a direct result of the one-sided communication provided by the PGAS model. Specifically, the shared-memory style of programming provided by the PGAS model encourages the use of `gets` and `puts` to only access data that is needed, which will result in fine-grain remote communication in the presence of irregular memory access patterns. One solution to improve the performance of PGAS programs that exhibit irregular memory access patterns is to manually apply *runtime optimizations*. Runtime optimizations rely on information that is only known once the program is executing, such as the organization of data in a sparse matrix or graph. This is in contrast to static optimizations that can only reason about the program at compile-time, at which point the details of irregular memory access patterns are not known. With runtime information, optimizations can use techniques like computation and data reordering [DK99; SCF03], replication [Das+94] and prefetching [AJ17] to address irregular memory access patterns.

Through manual application of these optimizations, programmers are effectively peeling back the abstractions provided by the PGAS model, resulting in programs that look similar to traditional two-sided communication MPI programs. Furthermore, these optimizations often increase the number of lines of code in the program several fold and require a significant amount of time and expertise to apply correctly.

1.1 Problem Statement

Irregular memory access patterns pose both performance and productivity challenges in the PGAS model. Because the access patterns are not known until runtime, traditional compiler optimizations are not effective, placing the burden of applying optimizations onto the programmer. To achieve good performance, programmers have to consider complex runtime optimizations such as coalescing small messages to avoid fine-grain communication, overlapping communication with computation and replicating remotely accessed data. The challenges in applying these optimizations include:

- Locating statements in a program that exhibit irregular memory access patterns and are likely to lead to fine-grain remote communication.
- Determining which optimizations can be applied without producing incorrect program behavior.
- Determining which optimizations to apply to provide improved performance.
- Modifying an existing program to perform complex runtime optimizations.

Addressing these challenges by hand decreases developer productivity by increasing the amount of code that a programmer would have to write, as well as increasing the amount of time required to produce optimized code. As a result, achieving good performance comes at the cost of diminishing the productivity benefits provided by the PGAS model.

1.2 Thesis Statement

This dissertation aims to demonstrate the following:

> *Through the design and implementation of a framework for automatic runtime optimizations, good performance and developer productivity can*

be achieved for PGAS programs that exhibit irregular memory access patterns.

1.3 COPPER: Compiler Optimizations for Productivity and Performance

The main contribution of this dissertation is COPPER, a framework that performs Compiler Optimizations for Productivity and PERformance in the PGAS programming model. COPPER specifically targets irregular memory access patterns to distributed data within parallel loops. Through the use of COPPER, programmers can more effectively use the PGAS model for irregular applications. In the following subsections, I summarize the purpose and design features of COPPER, as well as the different runtime optimizations that it performs. A more in-depth description of COPPER is provided in Chapter 4.

1.3.1 Purpose and Design Features

The purpose of COPPER is to bridge the gap between high performance and developer productivity within PGAS programs that exhibit irregular memory access patterns. COPPER automatically identifies irregular memory access patterns within PGAS programs and performs code transformations to enact different runtime optimizations, which include aggregation, prefetching and replication. Therefore, users can achieve significant performance improvements for their applications without having to rewrite their code to perform manual optimizations. In fact, programmers need not be aware that their application even exhibits irregular memory accesses or fine-grain communication, as COPPER will automatically identify such patterns without requiring the user to add code annotations to guide the optimizations.

I implement COPPER within the compiler for the Chapel parallel programming

language [Cha+18]. Chapel is a high-level language that provides constructs for parallel loops and distributed arrays, and implements the PGAS model. More details regarding Chapel are provided in Chapter 2. The performance issues that COPPER addresses are not specific to Chapel, but rather they are based in the PGAS model itself.

Static Analysis

COPPER uses static analysis to identify indirect accesses to distributed arrays within parallel loops, specifically of the form $A[B[i]]$. Such access patterns are likely to be irregular, and preclude static optimizations because the values in B are often not known until runtime. By focusing on indirect accesses to distributed arrays, COPPER will hone in on memory access patterns that are likely to exhibit fine-grain remote communication. The $A[B[i]]$ access pattern is present in a variety of key kernels across different application domains (see Chapter 3). Beyond identifying irregular memory access patterns, COPPER performs interprocedural analysis, alias analysis and dynamic call path analysis to ensure the validity of the optimizations that are applied. *These analyses did not exist in the Chapel compiler previously.*

Code Transformations

Because the irregular memory accesses cannot be fully optimized at compile time, the purpose of the code transformations performed by COPPER is to set up the optimizations that will be performed at runtime. These code transformations are centered around loop specialization techniques, but also include inserting calls to Chapel procedures that will be executed at runtime to carry out crucial components of the optimizations. These procedures consist of the optimization tasks that do not require compile-time specialization, such as determining whether an array access results in remote communication. By abstracting such tasks to Chapel procedures rather than

having the compiler construct the necessary code, the design and implementation of COPPER is more modular and modifiable. *These code transformations and optimization routines did not exist in Chapel previously.*

Interoperability Between Optimizations

Additionally, COPPER is invoked through a single compiler flag to the program, where all the optimizations are applied as needed. This means that the user does not need to be cognizant of which optimization(s) should be considered for a given program. Instead, COPPER performs the required analysis to determine which optimization can and should be applied for a given irregular memory access. This interoperability feature for the optimizations further reduces the burden on the programmer.

1.3.2 Optimizations

The core components of the COPPER framework are the runtime optimizations it applies. These optimizations include *aggregation*, *prefetching* and *replication*. Within the realm of the PGAS model, as well as distributed-memory parallelism, these three optimizations have been shown to be effective techniques to reduce fine-grain communication [Alv+13], hide remote communication latency [KFE18] and exploit remote data reuse [Das+94]. *My contributions are adapting these optimizations into the PGAS model and Chapel, as well as integrating the optimizations into a single framework.*

Aggregation

A contributing factor to the performance issues that arise due to fine-grain communication is the overhead of preparing individual messages to be sent over the network. When performing fine-grain communication that consists of many small messages, each message incurs the setup cost to prepare it for delivery across the network. A

better approach is to delay sending the small messages and coalesce them locally into a single message, which only incurs the overhead of preparing one message for delivery. This approach is often referred to as *small message aggregation*, and represents a widely used method to improve the performance of fine-grain communication [Kay+21; Alv+13; Pri+19; MD19; Pau+21]. The challenge with applying small message aggregation to programs is that the programmer must be aware of whether a given operation can and should be aggregated, and they must be able to write the code to perform the aggregation. COPPER addresses these challenges by performing static analysis to identify operations that meet the criteria for aggregation and then applies code transformations to perform the aggregation at runtime.

Prefetching

The purpose of prefetching is to hide memory access latency, which in the case of PGAS programs usually means remote communication latency. Through static analysis and code transformations, COPPER can identify irregular memory accesses to a distributed array within a parallel loop and insert procedure calls to prefetch elements of the array that will be accessed in future loop iterations. The act of prefetching refers to performing an asynchronous (i.e., non-blocking) `get` to a specified element of the array. While the prefetched element is being transferred over the network, the program continues execution of the loop. Ideally, when the program reaches the loop iteration that needs to access the prefetched element, the data transfer is complete and the element is now locally accessible. A challenging task for any prefetching optimization is determining how far "ahead", or how many loop iterations in the future, to issue prefetches. This value is referred to as the *prefetch distance*, and can be influenced by the irregular memory access pattern, the overall computation/communication workload performed within a loop iteration and the latency of the network interconnect. COPPER approaches this problem at runtime by dynamically adapting

the prefetch distance for a given memory access pattern, utilizing system performance counters to evaluate the effectiveness of the prefetches. In this way, the prefetch distance automatically adapts to not only the underlying memory access pattern and loop iteration workload, but also to the interconnect latency and load on the network from other programs/users.

Replication

In some applications, processes executing a parallel loop need repeated access to remote elements of a distributed array. Examples of these applications include conjugate gradient solvers [DHL16], the PageRank graph analytics algorithm [BGS05] and molecular dynamics simulations [MWK99; BE06]. Rather than issue repeated fine-grain remote accesses to the distributed array, each process can be given a copy of the remote data prior to the loop, thereby providing local access to the data within the loop. This is a common technique used in traditional two-sided MPI programs and can exploit the data reuse present in the loop. However, with the information only available at compile time, it is not possible to determine which elements need to be replicated for each process. Replicating the entire array on each process can lead to unnecessary communication and memory usage. COPPER addresses this by performing *selective data replication*, where only the elements that are accessed by a process are replicated. This is done via the *inspector-executor* technique [SMC91; Das+94; SCF03]. The inspector is a routine inserted by the compiler to “inspect” the memory accesses in the loop, determining which accesses are remote for each process. The executor is an optimized version of the loop that leverages the information from the inspector to perform the replication and execute the loop using the replicated data.

Related Work

Optimizations for prefetching, aggregation and replication have been developed in prior research, but COPPER brings them together under one framework and applies them automatically to PGAS programs via the compiler. Furthermore, there are several differences between the optimizations implemented within COPPER versus these prior studies, which make the work presented in this dissertation novel. These differences are summarized below, and a more detailed discussion is presented in each optimization's respective chapter.

Remote data aggregation has been studied and implemented in several prior works [Kay+21; Alv+13; CIY05; FB15; Pau+21; Pri+19], most of which focus on PGAS languages and libraries. Some approaches perform aggregation at the communication layer [FB15; CIY05], where only individual `gets` and `puts` are considered with very little context related to the higher level source program. This prevents the aggregation of more complex operations that cannot be easily represented as a sequence of `gets` and `puts`. Other work performs aggregation with the source program in mind [Kay+21; Alv+13], relying on static analysis to identify what can and should be aggregated. However, that approach to aggregation is limited to what can be statically determined at compile-time, such as what operations are order independent. Another approach to aggregation is to require the user to specify all the details of the aggregation [Pau+21], which provides the most flexibility of the noted approaches but also places the highest burden on the user. The aggregation optimization in COPPER is performed automatically on the source program and requires no user intervention. This approach is most similar to the work by Kayraklioglu et al. [Kay+21], but significantly extends that work to support more operations and loop structures.

Prefetching for both hardware [Smi82; DS96] and software [AJ17; CKP91; Bad+04] has been widely studied, with particular emphasis on determining the prefetch dis-

tance to use. The adaptive prefetching employed by COPPER is based on the work by Heirman et al. [Hei+18], specifically their approach to adjusting the distance over time. However, I significantly extend their work by applying adaptive prefetching automatically via the compiler, and do so for distributed-memory systems and the PGAS model.

Using replication to exploit remote data reuse is a common technique used in distributed-memory applications. Das et al. [Das+94] implemented an inspector-executor optimization that performs selective data replication and is very similar to the approach taken by COPPER. However, their work predates the PGAS model, and as a result, there are many differences between their assumed programming model that lead to significantly different approaches to static analysis and code transformations. Within the PGAS model, there has been prior work to implement data replication as a compiler optimization. Su and Yelick [SY05] developed an inspector-executor optimization for replication within the PGAS language Titanium and Alvanos et al. [Alv+12] perform replication via the compiler for the PGAS language UPC. While Titanium and UPC are PGAS languages, they differs significantly from the target language of this dissertation (Chapel) in its programming and parallel execution model. Specifically, UPC and Titanium follow the Single Program Multiple Data (SPMD) model while Chapel uses a fork-join style of parallelism where execution begins with a single thread/task executing the program. Additionally, UPC requires more explicit information from the programmer regarding parallelism and where threads execute their work. Such information is often important for compiler optimizations that aim to address fine-grain communication, as the mapping of threads to hardware resources (i.e., compute nodes) ultimately determines communication patterns. Because of these differences with Chapel, COPPER requires a different approach to static analysis and code transformations which have to infer information that is not exposed directly within the source code.

Table 1.1: Comparison between prior works and COPPER. The "Compiler Opt" column refers to whether the optimization is applied automatically by the compiler. For the "Opt" column, *aggr* refers to aggregation, *pf* refers to prefetching and *repl* refers to replication. The languages Chapel, UPC and Titanium are PGAS languages. OpenSHMEM is a PGAS communication library but is listed as a language for convenience

Study	Compiler Opt	Language	Opts	Target Arch
Kayraklioglu [Kay+21]	Yes	Chapel	aggr	Distr-memory
Paul [Pau+21]	No	C++/OpenSHMEM	aggr	Distr-memory
Conveyors [MD19]	No	C/UPC	aggr	Distr-memory
YGM [Pri+19]	No	C++	aggr	Distr-memory
Ainsworth [AJ17]	Yes	C/C++	pf	Multi/Many-core
Prodigy [Tal+21]	Yes	C/C++	pf	Multi-core
Saavedra [SP96]	No	C	pf	Distr-memory
Heirman [Hei+18]	No	C/C++	pf	Many-core
Alvanos [Alv+13]	Yes	UPC	repl	Distr-memory
Das [Das+94]	Yes	Fortran	repl	Distr-memory
Su [SY05]	Yes	Titanium	repl	Distr-memory
LAPPS [KFE18]	No	Chapel	repl	Distr-memory
COPPER	**Yes**	**Chapel**	**aggr,pf,repl**	**Distr-memory**

1.4 Performance Evaluations Across Systems

In addition to the design and implementation of the COPPER framework, this dissertation also provides an extensive performance evaluation of the optimizations in COPPER across a range of different irregular applications, data sets and hardware platforms. It is crucial to run experiments across different systems, as differences in on-node architecture and the network interconnect can lead to different performance behavior with and without the optimizations. Such differences can severely hinder the portability of compiler optimizations. However, results show that COPPER can achieve runtime speed-ups of 1.08 – 87x for aggregation, 0.78 – 3.2x for prefetching and 1.2 – 444x for replication across three different systems.

1.5 Comparison to Prior Work

Table Table 1.1 summarizes several prior works as they relate to the optimizations that COPPER provides, namely aggregation (aggr), prefetching (pf) and replication

(repl). The different optimizations are labeled within each work with respect to how COPPER defines the optimization. For example, the authors of LAPPS [KFE18] consider their optimization to be prefetching, but it is more similar to replication when compared to COPPER because LAPPS' prefetches are blocking. These prior works are described in more detail within each optimization's respective chapter (Chapters 5 – 7). While many of these works automatically apply some of these optimizations via the compiler, only COPPER provides compiler support for all three within the PGAS programming model.

1.6 Summary

This dissertation aims to demonstrate that both good performance and developer productivity can be achieved for PGAS programs that exhibit irregular memory access patterns. By doing so, programmers can more effectively use the PGAS model for irregular applications. Towards this goal, I make the following contributions:

1. Detailed descriptions of the design and implementation of a variety of irregular applications and kernels in Chapel. These implementations leverage the high productivity features of Chapel and the PGAS model, and represent the baselines that are used to evaluate the effectiveness of COPPER and its performance optimizations. (Chapter 3)

2. Design and implementation of a suite of static analyses and code transformation routines in Chapel's compiler. The static analyses identify irregular memory access patterns, and perform interprocedural and alias analysis. The code transformations modify a program to enable runtime optimizations. These analyses and transformations did not previously exist in Chapel.

3. The COPPER framework for automatically optimizing irregular memory accesses within PGAS programs written in the Chapel programming language [Cha+18].

COPPER consists of the above mentioned static analyses and code transformations. (Chapter 4)

4. A remote data aggregation optimization that is applied automatically by COPPER. The optimization targets irregular access patterns that modify (write) data and reduces fine-grain communication by buffering the writes locally and then performing them as one large message. Runtime speed-ups of 1.08 – 87x can be achieved via aggregation. (Chapter 5)

5. An adaptive software prefetching optimization that is applied automatically by COPPER. This optimization performs non-blocking remote reads a number of loop iterations in the future for a given irregular memory access pattern. Periodically, the prefetch distance is adapted (increased or decreased) to optimize the performance of the prefetches. Runtime speed-ups of 0.78 – 3.2x can be achieved via prefetching. (Chapter 6)

6. A selective data replication optimization that is applied automatically by COPPER. This optimization employs the inspector-executor technique to determine which elements of a distributed array are accessed remotely within a loop. The remotely accessed elements are replicated so they can be read locally during the loop execution, thereby exploiting data reuse by avoiding repeated remote communication. Runtime speed-ups of 1.2 – 444x are possible via selective data replication. (Chapter 7)

7. Discussion of the interoperability of the above-mentioned optimizations within COPPER. Specifically, I discuss how the optimizations can be applied together to a single program without requiring the user to specify which optimization to apply in a given scenario. Runtime speed-ups of 1.94 – 2.7x are possible on applications that perform several different kernels that are optimized separately via COPPER. (Chapter 8)

8. Extensive performance evaluation of COPPER across a range of different high performance computing systems. Each system exhibits different characteristics that impact performance, such as the number of CPU cores and the type of network interconnect between the compute nodes. Through this evaluation, I demonstrate that the optimizations provide performance advantages across the different systems. (Chapter 9)

Chapter 2

Background

In this chapter I present an overview of the topics that are central to this dissertation, namely developer productivity, irregular memory access patterns, the Partitioned Global Address Space (PGAS) programming model and the Chapel programming language. For each topic, I will provide brief background information as it relates to this dissertation, as well as motivation for the problem that this dissertation addresses: bridging the gap between performance and productivity for programs that exhibit irregular memory accesses written with the PGAS programming model.

2.1 Defining and Measuring Developer Productivity

As described in Section 1.2, this dissertation aims to demonstrate that both good performance and developer productivity can be achieved for PGAS programs that exhibit irregular memory access patterns. While performance can be measured with well-established metrics, such as time to solution or resource utilization, developer productivity is a more abstract concept. At a high level, developer productivity refers to the cost required to produce software that achieves some desired quality, where "cost" and "quality" can take on a variety of meanings. For example, cost could mean the number of lines of code (LOC) that had to be written and quality

could refer to the program producing the correct answer (e.g., numerical output). In another case, cost could refer to the amount of time required by the developer to write the program and the quality could refer to the program's execution time being less than one minute.

Beyond defining the meaning of developer productivity, another challenge is effectively measuring productivity. Regardless of the definition, productivity involves a human aspect (i.e., the programmer), which introduces significant variability into any measurement. For example, consider productivity as defined as the amount of time required by a developer to produce a program that yields the correct answer. If a programmer is an expert with the programming language being used and the problem being coded, their observed productivity will be significantly different from a programmer who is not an expert.

There have been studies conducted that attempt to measure productivity in a variety of settings, specifically within the realm of parallel programming models. Hochstein et al. [Hoc+05] developed parallel programming assignments to be given in graduate-level introductory high performance computing (HPC) courses at universities. The quality metric was the runtime speed-up achieved when compared to the serial implementations and the cost metric was the amount of effort (in hours) to plan, write, test and tune the code. They found that LOC alone is not a good enough proxy for developer effort. Ebcioglu et al. [Ebc+06] conducted a study to compare the productivity benefits of C with the Message Passing Interface (MPI), Unified Parallel C (UPC) [El-+05] and X10 [Cha+05]. User subjects were asked to parallelize a sequential version of the Smith-Waterman local sequence matching algorithm, which is a bioinformatics application. The metrics gathered included the amount of time until the first correct parallel solution was developed and the overall execution time of the program. Their results showed that the development time for subjects using X10 was significantly lower than those using MPI and UPC. Karlin et al. [Kar+13] conducted a

study that looked at different implementations of the LULESH hydrodynamics proxy application. The implementations include traditional parallel programming models like OpenMP and MPI, Charm++ [KK93] and (at the time of their study) emerging models such as Chapel [Cha+18]. Their measurements of productivity included LOC and subjective impressions on the experience of porting LULESH to the different programming models. The authors found that languages like Chapel require fewer LOC than the serial C++ implementation of LULESH. However, the user subjects in this study were experienced parallel programmers and even included some of the creators of the programming models evaluated.

In this dissertation, productivity is measured in terms of the amount of effort (e.g., LOC and time spent writing code) required to tune the performance of a program. One of the contributions of this dissertation is the COPPER framework, which automatically optimizes PGAS programs that exhibit irregular memory accesses. COPPER drastically improves the runtime and scalability of the program without requiring additional LOC. Furthermore, COPPER is implemented as part of the compiler, so the time spent improving the program's performance is equivalent to re-compiling the code. More details on COPPER will be presented in Chapter 4.

2.2 Irregular Memory Access Patterns

When considering a memory access pattern within a loop, it is often described as *regular* or *irregular*. Other terms used to describe regular and irregular access patterns are *affine* and *non-affine*, respectively. A regular/affine memory access pattern is of the form $A[i * a + b]$, where i is the loop index and a and b are symbolic constants within the loop. Such access patterns include purely sequential, non-unit strided and linear. Regular array access patterns often enable compiler optimizations since much of the pattern can be determined statically [Bag+19]. They also lend themselves to

high utilization of a cache memory hierarchy and hardware prefetching, as their access patterns exhibit spatial locality and can be predicted. Examples of regular memory access patterns are shown on lines 2–5 in Listing 2.1 and are commonly found in scientific applications and linear algebra kernels [Law+79].

```
// Regular access patterns
for i in 0..N {
  A[i] = ...;    // trivially sequential
  A[i*2] = ...; // non-unit strided
}

// Irregular access patterns
for i in 0..N {
  A[B[i]] = ...; // elements of B not known at compile time
  A[f(i)] = ...; // evaluation of f() not known at compile time
  list->next = ...; // pointer to "random" memory location
}
```

Listing 2.1: Examples of regular and irregular memory access patterns.

On the other hand, irregular/non-affine memory access patterns are often described as random or indirect and cannot be easily inferred at compile time. Examples of irregular memory access patterns are shown on lines 8–12 in Listing 2.1. In this dissertation, I focus on access patterns of the form $A[B[i]]$, where the offset into A is dependent on the values in B. These irregular memory access patterns are commonly found in sparse matrix computations [Dav06], graph analytics [Pea17] and even some forms of machine learning [Gal+20]. For many irregular applications that exhibit sparse, indirect memory access patterns, the internal representation of the data structures are compressed (i.e., dense). In other words, the sparse matrix or graph is not stored as a sparse structure, but instead uses formats such as Compressed Sparse Row (CSR) and adjacency lists. These compressed formats allow one to ignore zeros within the data, which saves memory. The irregular nature of the applications arise from the memory access pattern to these compressed data structures.

Irregular memory access patterns pose significant performance challenges on modern systems due to poor utilization of the cache memory hierarchy, as well as their inability to be statically optimized by the compiler. These non-affine access patterns

exhibit little to no spatial locality, which translates to under utilization of cache lines and overall wasted data movement. Aananthakrishnan et al. [Aan+20] empirically show this by measuring the number of bytes within a 64-byte cache line that are actually used by the CPU for a variety of irregular workloads. They observed that in most cases, zero or 8 bytes were used. These two scenarios represent when a cache line is prefetched but never used and when only a single 8-byte value is needed (i.e., sparse memory accesses). Furthermore, irregular memory access patterns often cannot be inferred until runtime, as they are data dependent. For example, in the indirect access pattern $A[B[i]]$, the values in B may refer to the non-zero column indices in a sparse matrix that is provided to the application at runtime. Because of this, static compiler optimizations that target affine array references are largely not applicable for irregular memory access patterns.

Worth noting is that many applications with irregular memory accesses operate on large, sparse data sets (e.g., graphs) that easily exceed the memory capacity of a single server, and therefore require distributed memory systems. When the irregular memory access patterns described above are issued to data distributed across a cluster, they result in *fine-grain remote communication*. In some ways, this is significantly more problematic than poor cache utilization on a shared-memory system because the latency of a network interconnect (such as Infiniband) is much higher than the latency between a processor and its memory hierarchy (i.e., microseconds vs. nanoseconds). For this dissertation, I specifically target irregular memory accesses on distributed-memory systems.

2.3 Partitioned Global Address Space Model

As noted in Section 2.2, this dissertation focuses on irregular applications executing on distributed-memory systems, where multiple servers (referred to as nodes) are

connected over a high-speed network and multiple processes across the nodes work together to execute an application. One of the reasons to target distributed-memory systems is because the amount of memory needed by an application exceeds a single server. For example, the amount of memory required to store the raw edge data for the 2012 Web Data Commons (WDC) hyperlink graph [Meu+] is more than 8 TB and the graphs evaluated by the Graph500 benchmark [Bad+22] are as large as 9 PB.

A key principle of developing distributed applications is *data distribution*, and by extension, *data ownership*. Because the data structures are so large, they are partitioned and distributed across the nodes in a system. The processes/threads executing on a node will have ownership of their partition of the data, but a given process will often need to access data that is located on another node, which will require communication over the network. The standard message-passing model (i.e., MPI) requires explicit communication calls to be added to the application to perform the communication, as each process in the application has its own address space. Figure 2.1(a) illustrates how this looks between two processes, where process 0 wants to access data owned by process 1. For message passing, both processes must participate in the communication, which is referred to as *two-sided communication*. This is a challenge for productivity, as it requires the programmer to be aware of whether data is local or remote with respect to each process. Furthermore, programmers must implement the logic in their code that ensures process 0 performs a `receive` for the data and process 1 performs a matching `send` for that data. However, this presents opportunities for high performance, as it forces the programmer to consider the higher cost of remote versus local accesses (i.e., locality), which naturally leads to techniques such as aggregation and replication to reduce communication costs. This can be contrasted with Figure 2.1(c), which shows the shared-memory model. In this case, there is a single address space that multiple processes/threads share, so accessing data does not require any explicit communication calls. But, as previously noted, the applications

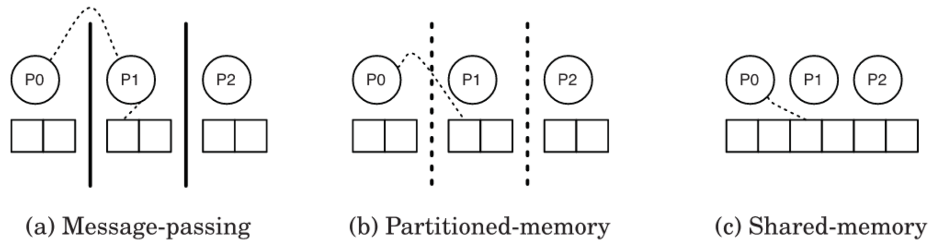

Figure 2.1: Different memory models and how data is accessed. Circles present processes and rectangles represent memory. The dashed lines represent accesses from processes to memory. The vertical bars between processes and memory represent the partitioning of the memory. This figure was taken from De Wael et al. [De +15].

of interest in this dissertation are those that execute on distributed-memory systems.

The Partitioned Global Address Space (PGAS) model attempts to provide a shared-memory style of programming for distributed-memory systems while also achieving high performance [De +15]. The PGAS model provides an abstraction to the application developer where the distributed memory in a system is viewed as a single logical global address space. The PGAS model has been implemented as part of libraries (GlobalArrays [NHL96] and OpenSHMEM [Cha+10]), extensions to existing languages (UPC [El-+05] and UPC++ [Bac+19]) and completely new languages (Chapel [Cha+18]). The PGAS model can lead to performance benefits since the global address space is partitioned amongst the nodes to allow for locality to be exploited, where memory accesses exhibit different costs depending on their source and destination (i.e., local versus remote). However, programmer productivity can also be improved via the PGAS model by simplifying how communication is performed, since the application developer can perform *one-sided communication* that does not require both processes to be involved. This is illustrated in Figure 2.1(b). The dotted vertical line represents the fact that the memory is still physically distributed as in the message-passing model, but the access from process 0 to the memory owned by process 1 does not require participation by process 1 (i.e., the address space is logically shared). In other words, process 0 can simply *get* the memory it needs, and it

can similarly *put* data to memory local to another process. It is worth noting that version 2 of the Message Passing Interface (MPI-2) standard provides a similar concept of one-sided communication, but it does not provide the rich abstractions that many PGAS languages/libraries provide. For example, use of one-sided communication in MPI-2 requires explicit allocation/registration of the memory which will be used by data structures involved in the communication. In contrast, PGAS languages like Chapel and UPC automatically handle the registration of memory to be used for communication by leveraging libraries such as GASNet [BH17].

The PGAS model is very appealing for irregular applications that need to perform fine-grain communication, as it provides a shared-memory style of programming that is significantly more productive than standard two-sided message-passing. However, the PGAS model does not fully address the performance challenges of irregular memory access patterns that are described in Section 2.2. The same types of optimizations that are necessary for standard message-passing applications, such aggregation and replication, are still necessary for irregular applications that use the PGAS model [CIY05; Rol+21; RS22]. As a result, a trade-off arises between achieving high performance via such optimizations and maintaining high developer productivity, since applying these optimizations often peels back the abstraction layers that the PGAS model provides. In the end, a high performance irregular application within the PGAS model may closely resemble a message-passing implementation. My thesis is that users can obtain both high productivity and high performance for irregular applications in the PGAS model by automatically applying optimizations via the compiler. This is demonstrated through the COPPER framework (see Chapter 4).

2.4 The Chapel Parallel Programming Language

In this section, I will provide a summary of the Chapel parallel programming language [Cha+18]. Chapel is one of the outcomes from DARPA's High Productivity Computing Systems program (2002-2010) [Don+08] and is still actively developed as an open-source project at the time of this writing[1]. Chapel implements the Partitioned Global Address Space (PGAS) model and provides many high-level features such as built-in syntax for parallel loops and data distributions, as well as implicit remote communication. These features enable high developer productivity when writing distributed codes that exhibit irregular memory access patterns, such as graph analytics. However, these features can lead programmers to develop codes that exhibit fine-grain communication, which degrades performance.

Chapel is the primary vehicle for the COPPER framework and its optimizations (Chapters 4 – 7). This section will describe the overall design philosophy of Chapel, as well as syntax and program construct details. Readers can refer to the Chapel documentation[2] for a more in-depth overview of the language.

2.4.1 Design Philosophy and Goals

The overall goal of Chapel is to provide a high productivity parallel programming language that is also high performing and scalable to large distributed-memory systems[3]. As a result, Chapel allows for domain scientists to write parallel applications without expert knowledge of data distribution, communication, synchronization, etc. Chapel implements a style of parallel programming where a programs begin with a single thread of execution and additional parallelism is created through high-level constructs (see Section 2.4.2). Chapel also provides mechanisms to achieve high performance by

[1] https://github.com/chapel-lang/chapel

[2] https://chapel-lang.org/docs/index.html

[3] While Chapel can also run on shared-memory systems, my work is focused on distributed-memory systems.

exposing computation and data locality to the user, allowing programmers to specify where computation should execute and where data should be placed. In an effort to provide a parallel language that is approachable by users of more mainstream languages, Chapel provides object oriented features (data encapsulation and classes), polymorphism and static type inference.

When designing a distributed-memory application, developers must consider not only how the algorithm should be structured, but also how to perform the algorithm in a distributed/parallel manner. To achieve high productivity, Chapel attempts to separate the details of data distribution and communication from the algorithm itself. Chapel achieves these goals largely through high-level parallel programming constructs and the Partitioned Global Address Space (PGAS) model. Of particular relevance to this dissertation, Chapel performs *implicit remote communication*. This means that issuing accesses to remote elements of a distributed data structure will appear identical (in the application code) to accesses to the local elements. Chapel's runtime system will determine if the access is remote, and if so, perform the one-sided communication to access the data. This provides a significant advantage for distributed irregular applications, whose communication pattern cannot be inferred at compile time, as the resulting code will closely resemble the equivalent shared-memory implementation. However, irregular applications in Chapel often suffer from fine-grain remote communication, which is a direct result of Chapel's productivity advantages (i.e., implicit communication) and the PGAS model. One of the goals of Chapel is that programmers should be able to develop and test their application on a single server, and then simply recompile it for execution across a cluster. While Chapel does provide the productivity features to allow such portability, high performance will most likely not be achieved in the presence of irregular memory access patterns.

2.4.2 Syntax for Parallel and Distributed Programming

In this section, I will use code examples to present the basic syntax of Chapel's parallel and distributed programming environment. This is not intended to be a comprehensive summary of the entirety of the Chapel language, but rather focus on the specific topics that are most relevant for this dissertation: terminology, arrays and domains, distributed arrays and parallel loops.

Terminology

In Chapel, *tasks* are computations that can conceptually execute concurrently. For example, the body of a loop may be assigned to a task, where the statements in the body are executed sequentially by the task. However, multiple tasks (each given their own iteration of the loop) can execute concurrently. Tasks are implemented via threads by a tasking layer provided to Chapel, where Qthreads [WMT08] is the default tasking layer used. Tasks execute on *locales*, which are units of machine resources that include memory and compute cores. Unless stated otherwise, a locale will be equated to a compute node in a cluster within this dissertation. Tasks running on a locale will have faster access to that locale's memory when compared to accessing another locale's memory, which would require remote communication. When a Chapel program is launched, the number of locales to use is specified on the command line and can be queried within the program through the global variable `numLocales`. Within a program, Chapel provides mechanisms to query and interact with locales by defining a `locale` type. The built-in variable `here` is a `locale` variable and is used to query where a task is currently executing via `here.id`, which returns an ID between 0 and `numLocales-1`. Many other variables contain a `locale` type as part of their underlying construction, allowing programmers to query where a given variable is located (e.g., `foo.locale.id`).

Chapel uses a style of fork-join parallelism, where programs begin with a single

task executing on locale 0. Language constructs are then used to create additional tasks, distribute data and control where tasks execute. This is in contrast to the Single Program Multiple Data (SPMD) model that is used in other PGAS languages/models, such as UPC, OpenSHMEM and GlobalArrays. Listing 2.2 shows examples of controlling where tasks execute, as well as querying where a given task is currently executing. Worth noting is the built-in `Locales` variable, which is an array that contains `locale` variables that represent each locale. The `Locales` array can be iterated over, as shown on lines 4–8, to execute a task on different locales. This is ultimately accomplished via the `on` clause, which will migrate the current task to execute on the locale specified by `loc`. The operand given to the `on` clause can be any expression, where the task will execute on the locale where the result of the expression is stored. For example, `on func()` will execute the task on the locale where the return value of the procedure `foo()` is located. If `foo()` returns an element of an array that is stored on locale 4, then that is where the task will execute.

```
1  writeln("The number of locales is ", numLocales);
2  writeln("Program always starts on locale ", here.id);
3
4  for loc in Locales {
5    on loc {
6      writeln("Migrated task to locale ", here.id);
7    }
8  }
9
10 /*
11   Output when running on 4 locales:
12
13   The number of locales is 4
14   Program always starts on locale 0
15   Migrated task to locale 0
16   Migrated task to locale 1
17   Migrated task to locale 2
18   Migrated task to locale 3
19 */
```

Listing 2.2: Examples of interacting with Chapel locales.

Arrays and Domains

An array in Chapel has two distinct parts: the array itself that stores the data and the *domain* that represents the index set of the array. Domains, and hence arrays, can be multi-dimensional, represent negative indices and store non-contiguous indices. While Chapel provides a rich set of features for domains, this dissertation mostly uses standard rectangular domains (one- or two-dimensional) with integer-based indices. However, I also utilize *associative domains*, which provide dictionary-style functionality. With a domain declared, users can define arrays over the domain. Listing 2.3 presents examples of declaring and using domains and arrays in Chapel. Line 2 declares a domain `D`, where the `#` operator is the count operator and can be used when defining ranges. In this case, it specifies a range with `N` elements starting at 0. Line 3 declares an array `data` over the domain `D`, where the elements in the array will be integers. Multiple arrays can be declared over the same domain, where updating the domain (e.g., resizing it) will automatically update all arrays declared over that domain. Lines 7–19 show multiple ways to access elements of an array, such as manually iterating over the index space (lines 7–9), iterating directly over the domain variable (lines 10–12), iterating specifically over the array's domain (lines 13–15), iterating over the array's elements in a for-each style (lines 16–18) or using whole array assignment when the programmer wants to give every element the same value (line 19). Worth noting is that whole array assignments/operations are automatically performed as parallel operations.

```
var N = 5; // static type inference used to infer N is an int
var D : domain(1) = {0..#N};
var data : [D] int;

// All of the following are equivalent approaches to
// setting all elements of data to 1.
for i in 0..#N {
  data[i] = 1;
}
for i in D {
  data[i] = 1;
}
for i in data.domain {
  data[i] = 1;
}
for elem in data {
  elem = 1;
}
data = 1;

// Associative domain/array
var C : domain(string);
C += "foo";
C += "bar";
var dict : [C] real;
dict["foo"] = 1.0;
dict["bar"] = 2.0;
```

Listing 2.3: Examples of Chapel domains and arrays.

Listing 2.3 also shows an example of declaring and using associative domains. An associative domain can be used to represent an arbitrary set, or it can be used to define a key-value style array. Line 22 declares an associative domain `C` that will contain strings and lines 23 and 24 append keys to `C` via the `+=` operator. An array `dict` is declared over the associative domain on line 25, where the values in the array will be reals (i.e., 64-bit floating point). Values can then be associated with the keys in the associative domain (lines 26 and 27). A key property of associative domains that is leveraged in this dissertation is that multiple tasks can concurrently update the domain without any explicit synchronization. This parallel-safety feature is enabled by default when creating an associative domain, but can be disabled to improve performance when it is known that updates will not be issued by multiple tasks. However, any array declared over the associative domain should not be accessed while

the domain is being modified, which is also true for all other types of domains/arrays. Furthermore, associative domains do not enforce any particular ordering of their elements and they automatically ignore duplicate updates (i.e., the same key/element will not be added to the associative domain twice).

Distributed Arrays

The domains and arrays presented in Listing 2.3 would be allocated on the locale where the task is currently executing. In other words, they are not distributed across multiple locales. Chapel provides built-in mechanisms to create domains and arrays with a variety of distributions over the locales, such as block, cyclic and block-cyclic. The distribution used primarily in this dissertation is the block distribution, where the array elements are partitioned into contiguous chunks and each locale receives one chunk. Listing 2.4 illustrates how a block distributed domain is defined and then used to declare a block distributed array. Line 1 defines the index space for 16 elements and line 2 defines a block distributed domain `D` over the index space. The array `data` is declared and distributed over the block distributed domain on line 3. Figure 2.2 shows an illustration of how the block distributed array would be distributed across four locales. The `for` loop on lines 7–9 sets each array element to the ID of the locale on which it is allocated. The iterations of this loop will execute on locale 0, meaning that any iteration that accesses elements of `data` that are not on locale 0 will cause remote communication. However, the difference between a local versus remote access is abstracted from the user due to Chapel's implicit remote communication. Specifically, users do not need to reason about distributed arrays in terms of data partitions that exist on locales, where each partition has its own index set that starts at 0 (which is often the case for standard message-passing programming models). However, this can pose a challenge for optimizing communication performance, as it is not immediately clear from the source code which accesses are remote. Mechanisms to control where

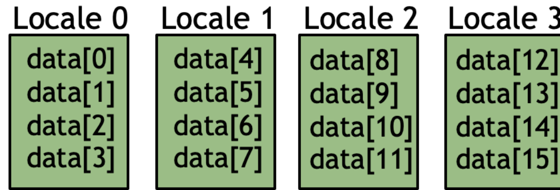

Figure 2.2: Illustration of how the 16-element block distributed array `data` from Listing 2.4 is distributed across four locales.

loop iterations are executed, which avoids the unnecessary communication shown in this example, will be discussed next.

```
var s = {0..#16};
var D = newBlockDom(s);
var data : [D] int;
// can also directly create the array via newBlockArr(s, int)

// Implicit remote communication
for i in 0..#16 {
  data[i] = data[i].locale.id;
}
```

Listing 2.4: Chapel block distributed arrays.

Parallel Loops

Chapel provides mechanisms to enable both task and data parallelism. The main construct used in this dissertation for task parallelism is the `coforall` loop. The `coforall` loop spawns a separate task for each iteration of the loop, where the task is responsible for executing its iteration of the loop body. Tasks spawned via a `coforall` loop execute concurrently, with an implicit barrier at the end of the loop. For data parallelism, Chapel provides the `forall` loop construct, where the iterations of the loop are divided into chunks and the chunks are executed concurrently by separate tasks. This is more or less equivalent to the `omp parallel for` directive in OpenMP [DM98]. When programmers use `forall` loops, they are asserting that the loop iterations are order independent, and thus have no loop carry dependencies. While there is a burden placed on the user to ensure that this is true, Chapel can statically detect simple data races and raise a compiler error, such as writes issued

to non-array variables that are declared outside the scope of the `forall`. In addition to the `forall` loop, Chapel also provides the `foreach` loop. The `foreach` loop also indicates that the programmer is asserting order independent loop iterations, but it does not result in the creation of parallel tasks like the `forall` loop. Instead, it serves as a hint to the compiler that this loop is a candidate for vectorization or offloading to a graphics processing unit (GPU).

```
// Distributed-memory parallelism
coforall loc in Locales do on loc {
  writeln("Hello from Locale ", loc.id);
}

// Shared-memory parallelism
forall i in 0..#64 {
  writeln("Iteration number ", i);
}

var data = newBlockArr({0..#64}, int);

// Explicit distributed- and shared-memory parallelism
coforall loc in Locales do on loc {
  forall i in data.localSubdomain() {
    data[i] += 1;
  }
}

// Implicit distributed- and shared-memory parallelism
forall d in data {
  d += 1;
}
```

Listing 2.5: Chapel coforall and forall loop constructs.

Listing 2.5 shows examples of the `coforall` and `forall` loop constructs. A common idiom in Chapel programs is to spawn one task on every locale to execute the body of a loop, which achieves a form of distributed-memory parallelism. This is shown on lines 2–4, where each task spawned will print out its locale ID in a non-deterministic order, as each task is executing concurrently. Note the use of `do on loc`, which is a shorthand for the more explicit `on` clause from Listing 2.2. Lines 7–9 show an example of a `forall` loop that prints out the iteration index of the loop. In this case, the 64 iterations are broken up into chunks and executed concurrently, where the number of compute cores available on the locale determines the number of tasks

spawned and the chunk size. To achieve both distributed- and shared-memory parallelism, the `coforall` and `forall` constructs can be used in conjunction, as shown on lines 14–18. In this case, the `coforall` loop will spawn one task per locale, where that task then executes a `forall` loop to access the elements of the distributed array `data` in parallel, utilizing that locale's compute cores. The `localSubdomain()` function returns the portion of `data`'s domain that is located on the given locale. However, an implicit form of distributed- and shared-memory parallelism can achieved, as shown on lines 21–23, which is equivalent to the code on lines 14–18. Chapel's `forall` loops will automatically execute tasks on the locales where the iterations are mapped to when iterating over a distributed array/domain. I will refer to this as controlling the `forall` loop's *locale affinity*.

It is worth defining some of the terminology used by Chapel to describe various portions of loops, specifically iterators, iterands and loop index variables. These terms will be used extensively when describing COPPER's optimizations and how they are applied to Chapel programs. An *iterator* in Chapel is a procedure that can yield multiple values (consecutively or in parallel). Arrays and domains have default iterators associated with them, which determine how their values are yielded when they are iterated over in loop. Users can also define their own iterators to specify custom behavior (see Section 6.2.2 for examples). Chapel's default iterator for distributed arrays and domains is what provides the mechanism that automatically distributes a `forall` loop's task across the locales. The *iterand* of a loop is the portion that appears after the `in` keyword and can either be an iterator or iterable expression. For example, `Locales`, `data` and `0..#64` are loop iterands in the form of iterable expressions. Finally, the *loop index variable* is simply the variable that "holds" the value yielded by the loop's iterator (e.g., `loc`, `i` and `d` in Listing 2.5).

Chapter 3

Applications and Kernels

To evaluate the effectiveness of the COPPER framework (Chapter 4) and the optimizations it provides, I performed several experiments across different high performance computing (HPC) systems and distributed-memory irregular applications. In this chapter, I provide detailed descriptions of the irregular applications and kernels that are used in the performance evaluation of COPPER. Each application is placed into one of three categories: graph analytics, scientific computing and mini-kernels. In each application, performance is negatively impacted due to fine-grain remote communication. I also provide baseline implementations of these applications written in Chapel [Cha+18], since COPPER is implemented within the Chapel compiler.

There are an abundance of application-specific optimizations that can be manually applied to these codes. However, the purpose of these experiments is to evaluate COPPER, which automatically applies optimizations without domain-specific knowledge of the application. Therefore, the baseline implementations presented in this chapter represent straightforward approaches to each algorithm that leverage the productivity features of Chapel and the Partitioned Global Address Space (PGAS) model. To this end, many of the Chapel implementations are based on "textbook" algorithms and/or existing shared-memory parallelized implementations. These existing shared-memory

codes can often be adapted to distributed-memory systems by simply declaring the relevant arrays as distributed, letting Chapel and the PGAS model handle the required low level details regarding communication.

Each application is placed into one of three categories: graph analytics, scientific computing and mini-kernels. It should be noted that COPPER is not specifically optimized for any one of these categories. Instead, COPPER is designed to recognize and optimize common patterns in programs that lead to fine-grain remote communication. The purpose of evaluating applications that span these categories is to demonstrate the versatility of COPPER.

3.1 Graph Analytics

Graphs are a natural way to represent data and the connection/relationship amongst elements in the data. For example, a social network can be represented as a graph where the vertices represent people and edges between vertices represent a social interaction (e.g., friendship, communication, belonging to the same group, etc.). However, any given person in a social network is unlikely to be connected to all other people in the network. As a result, graphs often model sparse and irregular data, which give rise to irregular memory accesses when operations are performed on graphs. Furthermore, the amount of memory required to represent a graph can easily exceed the available memory on a single server. Figure 3.1 shows examples of the sparsity structure for some of the graphs that will be evaluated in this dissertation. As can be seen, there is very little structure that can be exploited within these graphs, which presents challenges for performance when executing graph analytics applications.

Graph analytics are a rich source of irregular memory access patterns to distributed data and serve as a valuable way to evaluate the effectiveness of the COPPER framework, which specifically targets such access patterns. Furthermore, graph an-

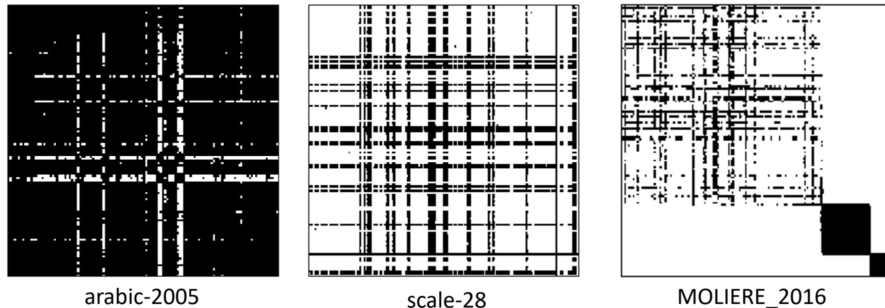

Figure 3.1: Sparsity patterns for some of the graphs that will be evaluated in this dissertation. These graphs were obtained from the SuiteSparse Matrix Collection [DH11] or generated according to the Graph500 benchmark specification [Bad+22] in the case of the scale-28 graph.

alytics are typically straightforward to implement when compared to applications within the scientific computing domain. For these reasons, graph analytics form a majority of the applications and kernels that are used to evaluate COPPER. The graph analytics that are evaluated by COPPER are Breadth First Search (BFS), Single Source Shortest Path (SSSP), K-core decomposition, PageRank (PR) and Triangle Counting (TC). In the following sections, I describe each analytic at a high level and provide baseline Chapel implementations. I also discuss other graph analytics that were studied but not included in the performance evaluations due to duplication of their overall structure in the other analytics or COPPER not providing a performance gain or loss when applied.

3.1.1 Breadth First Search

High-level Description

The Breadth First Search (BFS) algorithm traverses a graph from a start vertex referred to as the *root*, visiting all vertices adjacent to the root before visiting each of their neighbors. This level-by-level traversal continues until all vertices reachable from the root have been visited. BFS is a fundamental graph analytic and forms the basis for many other graph algorithms, as will be shown later.

Typically, two queue data structures are used to implement BFS, one to hold the vertices to visit at the current level and another for the vertices to be visited at the next level. These will be referred to as the current-queue and next-queue, respectively. Implementing BFS in a distributed-memory setting can be challenging due to the need to distribute the graph and queues across the nodes, which leads to many fine-grain remote writes. There is also the complication of handling concurrent updates from multiple tasks to the queues. This can arise because multiple vertices may share the same neighbor, and as a result attempt to add that shared neighbor to the respective queue at the same time.

Data Structures

Listing 3.1 presents the high productivity Chapel implementation of BFS that serves as the baseline for the experiments. The graph `G` is represented as a block-distributed array of records (i.e., C structs), where each record corresponds to a vertex. The vertex records store information such as the distance from the root vertex (`dist`) and an array of IDs that correspond to the vertex's neighbors in the graph (`neighbors`). A vertex's distance is initialized to -1 and is used to indicate that the vertex has not been visited yet.

The typical approach to performing BFS in a distributed manner is to give each locale its own current- and next-queue that only hold vertices located on the locale. Lines 1–2 in Listing 3.1 declare the current-queues (`currQs`) and next-queues (`nextQs`) as block-distributed arrays, where each locale is given one element of the distributed array. This is specified by the `LocaleSpace` argument to `newBlockArr()`, which is a built-in domain in Chapel that is simply defined as `{0..#numLocales}`. Each element of these distributed arrays is an associative domain, which is the underlying data structure used to represent the queues. Associative domains are a natural choice for the queues in this scenario because they provide parallel-safe insertions and ignore

duplicate entries, both of which are necessary to support the BFS algorithm. Each queue (i.e., associative domain) stores integers, which represent the IDs for the vertices within the queue.

```
1  var currQs = newBlockArr(LocaleSpace, domain(int));
2  var nextQs = newBlockArr(LocaleSpace, domain(int));
3  currQs[G[root].locale.id] += root;
4  var curr_lvl = 1;
5
6  while true {
7    var work : bool = false;
8    coforall cq in currQs with (|| reduce work) do on cq {
9      forall u in cq with (|| reduce work) {
10       if G[u].dist == -1 { // not visited yet
11         G[u].dist = curr_lvl;
12         work = true;
13         foreach v in G[u].neighbors {
14           nextQs[G[v].locale.id] += v;
15         }
16       }
17     }
18   }
19   coforall (cq, nq) in zip(currQs, nextQs) do on cq {
20     cq <=> nq; // <=> is the swap operator
21     nq.clear();
22   }
23   if !work then break;
24   curr_lvl += 1;
25 }
```

Listing 3.1: Baseline implementation of BFS.

Algorithm Implementation

Line 3 in Listing 3.1 begins the BFS algorithm by adding the specified root vertex, `root`, to the current-queue on the locale which "owns" the root vertex. Recall that updates/insertions to associative domains are performed via the overloaded `+=` operator. Lines 6–25 execute the main BFS loop, which iterates until there are no vertices left to visit. Lines 8–18 perform a single "round" of visits across all of the locales, where the `coforall` loop creates one task on each locale to process that locale's current-queue (`cq`) in parallel via the `forall` loop. If a given vertex `u` has not been visited before, it is marked as visited (lines 10-11). Lines 13–15 then add each neighbor vertex `v` to the next-queue of the locale that owns `v`.

After a round of visits is finished, each locale's next-queue becomes its current-queue, and the next-queues are cleared (lines 19–22). If no vertices were visited in the round, then the algorithm is finished and breaks out of the `while` loop (line 23). The `work` flag is used to indicate whether any vertices were visited in a given round. Each locale/task may update this flag at the same time, but as long as one of them sets it to true, that is enough to conclude that the algorithm needs to continue. To implement this behavior, a copy of `work` is given to each task implementing the `coforall` and `forall` loops via the `with` clause. The `with` clause allows for the creation of task-private variables within a loop. Combined with the `reduce` clause, all task-private copies of `work` are "combined" when the loops end, where the operation performed when combining copies is logical OR (`||`). This has the effect of setting the original copy of `work` to true as long as one of the private copies was set to true.

Fine-grain Remote Communication

Line 14 in Listing 3.1 is the source of the fine-grain remote communication in the BFS algorithm, and is the target of COPPER's aggregation optimization. This optimization will be discussed in detail in Chapter 5. For this operation, a task executing an iteration of the `forall` loop on a given locale will perform a remote write to add the neighbor vertex `v` to the next-queue of another locale (this write may be local if the given locale owns `v`).

With respect to the rest of the kernel, the fine-grain remote write on line 14 in Listing 3.1 will dominate the overall kernel execution time. This is because line 14 will execute many more times than any other memory access in the kernel, as it is nested within a `foreach` loop. Furthermore, all other memory accesses in the kernel are local and do not result in communication. As a result, optimizing line 14 is expected to provide large performance gains.

3.1.2 Single Source Shortest Path

High-level Description

Single Source Shortest Path (SSSP) is an algorithm that finds the shortest path from a specified root vertex to all other reachable vertices in the graph. In this context, "shortest" refers to the path with minimum edge weight. A path in a graph is defined as the sequence of edges that join some series of vertices, where each edge in the graph is assigned a weight value (typically between 0 and 1). Therefore, the edge weight of a path between the root vertex and some other vertex is the sum of the weights along the path's edges. This path edge weight is often referred to as the *distance* from the root.

A common sequential approach to perform SSSP is Dijkstra's algorithm. In this approach, the tentative distance of a given vertex from the root is updated iteratively until the algorithm terminates and the shortest path is determined. The distances are updated via *edge relaxations*, which change the distance if it is determined that there is a "shorter" path to the vertex in question. A common method to parallelize SSSP is the delta (Δ)-stepping algorithm [MS03]. The Δ-stepping algorithm is considered a distance-correcting algorithm, which allows for the graph to be processed in parallel. Specifically, vertices are binned into buckets depending on their current tentative distance. Each bucket represents a range of distance values that are of size Δ, which is a tunable input parameter to the algorithm. Each phase of the algorithm concurrently removes the vertices in the first non-empty bucket and performs edge relaxations on the edges whose weight is less than Δ; these edges are referred to as *light edges*. Edges with higher weight are relaxed only when their respective starting vertex has a non-tentative distance and are referred to as *heavy edges*.

Data Structures

Listing 3.2 presents the baseline implementation of the main SSSP kernel in Chapel, which is based on the Δ-stepping algorithm [MS03]. This is the kernel that iterates over the light edges in the graph. While there are other kernels performed within SSSP, this kernel contains the target memory accesses for COPPER and contributes the most to the overall runtime performance of the algorithm. The graph `G` is represented with the same data structure used for Breadth First Search (BFS), as described in Section 3.1.1. However, it is slightly modified so that each vertex record holds an array of edge weights (`weights`). Like BFS, the SSSP algorithm also makes use of per-locale associative domains to represent queues (`currQs` and `nextQs`).

```
1  var currQs = newBlockArr(LocaleSpace, domain(int));
2  var nextQs = newBlockArr(LocaleSpace, domain(int));
3  ...
4  coforall cq in currQs do on cq {
5    forall u in cq {
6      foreach i in G[u].neighbors.domain {
7        const v_weight = G[u].weights[i];
8        if (v_weight < delta) {
9          const w = G[u].dist + v_weight;
10         ref v = G[neighbors[i]];
11         if (v.dist < 0 || v.dist > w) {
12           v.dist = w;
13           if (!v.visited && w < max_delta) {
14             v.visited = true;
15             nextQs[G[v.id].locale.id] += idx;
16           }
17         }
18       }
19     }
20   }
21 }
```

Listing 3.2: Baseline implementation of the light-edge kernel in SSSP.

Algorithm Implementation

The code in Listing 3.2, which implements one of the kernels in the Δ-stepping algorithm for SSSP, is more or less a BFS traversal as presented in Listing 3.1. However, the processing that is performed for each vertex in the current queue is significantly

more complicated. This is because the SSSP algorithm does more than simply traverse the graph and visit vertices. It must consider the weight of the edges in the graph and keep track of tentative distances between the root and each vertex.

The `forall` loop on line 5 processes all of the vertices in a given locale's current-queue (`cq`) in parallel. The `foreach` loop on line 6 looks at each neighbor `v` of the vertex `u`, specifically the edge weight `v_weight` between `v` and `u` (line 7). If this edge is a light edge (i.e., `v_weight` is less than Δ), then additional processing is performed. Line 9 computes a tentative distance `w` between the root vertex and `v` by adding `u`'s distance to `v_weight`. If `v` has not been processed yet (its distance is less than 0) or its current distance is larger than the tentative distance `w`, then its distance is set to `w` (lines 11–12). Finally, if `v` has not been visited yet and its new distance `w` is less than `max_delta`, then `v` is added to the next-queue on the locale where `v` is located. The value of `max_delta` is initially set to Δ, but increases by Δ after each iteration of the SSSP algorithm.

While not shown in Listing 3.2, there are two other kernels performed as part of the Δ-stepping SSSP algorithm. One of the kernels processes the heavy edges, which are edges with weight greater than Δ, and the other kernel checks for the algorithm's termination condition. The former does involve remote communication, but much less than the kernel shown in Listing 3.2. The latter does not result in any remote communication and the access patterns are entirely regular.

Fine-grain Remote Communication

Lines 10 and 15 in Listing 3.2 are the sources of fine-grain remote communication. Both accesses may result in remote communication, where line 10 performs a read to access the vertex `v` and line 15 performs a write to `nextQs`. As noted in Section 3.1.1 for BFS, line 15 is a candidate for COPPER's remote data aggregation optimization (Chapter 5). Line 10 is a candidate for COPPER's adaptive remote prefetching op-

timization (Chapter 6). With respect to the rest of the kernel in Listing 3.2, lines 10 and 15 will dominate the overall kernel execution time because they are executed just as many times as any other memory access in the kernel. The other memory accesses, such as on lines 7 and 9, will not induce remote communication. As a result, optimizing the fine-grain remote communication on lines 10 and 15 should provide significant performance gains.

3.1.3 K-core Decomposition

High-level Description

The k-core of a graph is the largest induced sub-graph with minimum degree k (i.e., all vertices in the sub-graph have at least k neighbors). The core number of a given vertex is the largest k where that vertex belongs to a k-core. Determining the k-core of a graph has applications in social network analysis, bio-informatics, and network visualization [Mal+20]. The k-core *decomposition* of a graph finds all the k-cores of the graph (i.e., determines the core number of all vertices in the graph).

A popular algorithm to perform k-core decomposition is presented by Batagelj et al. [BMZ99], which is often referred to as the BZ algorithm. The BZ algorithm sorts the vertices in increasing order by their degree, and then in that order recursively removes vertices whose degree is less than k. Throughout this pruning process, the degrees of the vertices decrease until the core number for each vertex is determined. However, the BZ algorithm is not well suited for parallelization due to synchronization overheads that are required to keep the data structures coherent [DDZ14].

The ParK algorithm [DDZ14] is a parallel algorithm that performs k-core decomposition and is the basis for the Chapel implementation described below. The full details of the ParK algorithm are not needed for this discussion but can be found in the original paper [DDZ14]. At a high-level, the algorithm processes the vertices in a level-by-level fashion, where a level is a set of vertices and the next level consists

of the neighbors of those vertices. When processing a level, the algorithm scans the vertices and finds those whose core number is equal to the current level. It then performs a BFS-like traversal and processes these vertices and their neighbors. It is worth noting that while this algorithm was designed for multi-core environments and not specifically for distributed-memory systems, it is relatively straightforward to incorporate distributed-memory parallelism via Chapel due to Chapel providing a PGAS model.

Data Structures

Listing 3.3 presents the Chapel implementation of the main kernel in the ParK k-core decomposition algorithm. This code is functionally equivalent to the code on lines 10–26 in Figure 5 in the ParK paper [DDZ14]. In terms of data structures, the graph `G` is represented as a block-distributed array of vertex records (much like what is used for BFS and SSSP). However, the `deg` field is added to serve as the current degree of the vertex. As the implementation performs a BFS-like traversal, it also makes use of the same `currQs` and `nextQs` data structures as BFS to represent per-locale queues. However, unlike BFS and SSSP, the next-queues store tuples of integers rather than single integers. In BFS and SSSP, multiple tasks could attempt to add vertex v to the same next-queue. As associative domains will ignore duplicates, v only appears in the queue once, which is the desired outcome for an algorithm like BFS or SSSP (i.e., vertices do not need to be visited more than once). However, the intended behavior of the k-core decomposition algorithm is for the vertex v to be added multiple times to the next-queue. Further details are provided in the algorithm implementation description below.

```
var currQs = newBlockArr(LocaleSpace, domain(int));
var nextQs = newBlockArr(LocaleSpace, domain((int,int)));
...
while notEmpty(currQs) {
  coforall cq in currQs do on cq {
    forall v in cq {
      const ref neighbors = G[v].neighbors;
      foreach u in neighbors {
        nextQs[G[u].locale.id] += (u,v);
      }
    }
    cq.clear();
  } /* end of coforall */

  coforall (cq, nq) in zip(currQs, nextQs) do on nq {
    forall (v_idx, _) in nq {
      ref v = G[v_idx];
      if v.deg.read() > level {
        const a = v.deg.fetchSub(1);
        if a == level+1 {
          cq += v_idx;
        }
        if a <= level {
          v.deg.add(1);
        }
      }
    }
    nq.clear();
  } /* end of coforall */
} /* end of while */
```

Listing 3.3: Baseline implementation of the sub-level processing kernel in k-core decomposition.

Algorithm Implementation

The code in Listing 3.3 represents the process sub-level kernel in the ParK k-core decomposition algorithm. This kernel takes almost all of the runtime in the entire algorithm due to fine-grain remote communication; the other kernels do not induce communication or have irregular memory access patterns. Lines 5–13 perform a step/iteration of a BFS-like traversal with all the vertices in the current queue (`currQs`). Vertices were placed into the current queue during a step prior to the `while` loop on line 4 if their degree (`deg`) was equal to the current level in the algorithm. This step is referred to as the scan kernel in the ParK algorithm and requires

no remote communication or irregular memory accesses.

To allow for a given vertex `u` to be added multiple times to the next-queue (which is the intended behavior), `u` is added to a tuple that also contains its neighbor `v`. This makes the entry into the queue unique so it cannot be ignored as a duplicate. Lines 15–29 process the vertices that were added to the next-queue via the code on lines 5–13. For each vertex `v` in the next-queue `nq`, if its degree is greater than the current level, then its degree is decreased by 1, and if its degree is now equal to the current level then `v` is added to the the current-queue `cq`. The process of performing the BFS-like traversal and then processing the next-queue is repeated until the current-queue is empty (the `while` loop condition on line 4).

Lines 18–26 in Listing 3.3 address a race condition that would exist in the naive parallelization of the sequential ParK algorithm, as multiple tasks could attempt to read and modify a given vertex's degree at the same time. The ParK paper and the Chapel implementation in Listing 3.3 address this by performing the operations atomically. Chapel allows for variables to be declared as `atomic`, which can then support atomic operations, which enforce that only one task/thread is allowed to perform the operation at a time. However, it is still possible for multiple tasks to perform the atomic decrement on line 19 when the intended behavior is for only one task to do so. For example, two tasks could atomically read the degree as greater than the level (line 18) before any atomic decrements are performed (line 19), thus allowing both tasks to proceed. The solution to this is to look at the degree value prior to the atomic decrement (`a`), which is returned as part of the `fetchSub()` procedure that performs an atomic read and decrement. By looking at this value, a task can determine whether it has incorrectly decremented the degree, since it cannot be less than or equal to the current level (lines 23-25), and make corrections as needed (line 24).

The Chapel code shown in Listing 3.3 is slightly different from the pseudo-code

shown in Figure 5 in the ParK paper [DDZ14]. In the ParK paper, the BFS-like traversal (lines 5–13 in Listing 3.3) is combined with the degree-processing (lines 15–29 in Listing 3.3). Specifically, before the vertex `u` is added to the next-queue, its degree is checked and decremented. However, ParK was not designed for distributed-memory systems, and as a result, this degree-processing would induce fine-grain remote communication (specifically remote reads). Remote reads are blocking operations, so it is often a better strategy to avoid them if the implementation allows that. In this case, the Chapel code postpones the degree processing until after the vertices have been added to their respective next-queue. This may result in unnecessary remote writes to the queue (since vertices that do not meet the level criteria may be added).

Fine-grain Remote Communication

The fine-grain remote communication present in the k-core decomposition application is on line 9 in Listing 3.3. As with SSSP and BFS, this is a fine-grain remote write to the next-queues array and can be optimized via COPPER's aggregation optimization (Chapter 5). The entirety of the `coforall` loop on lines 15–29 executes without requiring remote communication. Therefore, the fine-grain remote write on line 9 will dominate the kernel execution time.

3.1.4 PageRank

High-level Description

PageRank (PR) [BGS05] is an iterative analytic that provides a ranking of the vertices in a graph, where a vertex's rank is determined by the ranks of its incoming neighbors. The original purpose of PR was to provide a ranking of websites for a search engine, where a given site is determined to have a high ranking if many other high-ranking sites have hyperlinks to it. PR outputs a probability distribution that provides the likelihood that a user randomly clicking hyperlinks will land on any given website.

The ranking given to each vertex, which will be referred to as the *pr-value*, is a probability and the higher this probability is, the more "important" the vertex is deemed. PR has shown to be a useful analytic beyond the web domain [Gle15].

Each iteration in PR visits every vertex's neighbors to update the pr-values based on the prior iteration's results. This neighbor processing is what leads to irregular memory accesses. PR terminates when there are no significant changes in the pr-values between two iterations, where "significant" is determined by an input threshold (e.g., 1*e*-7). As a result, the number of iterations required to converge depends on the underlying structure of the graph and the convergence threshold. A key property that can be exploited via optimizations is that each iteration of PR performs the same traversal pattern over the graph. Analytics like BFS, SSSP and k-core decomposition can be considered as iterative algorithms, where each iteration explores a different portion of the graph. However in PR, each iteration does an entire pass over all vertices in the graph and the way in which this is performed does not change between iterations. This allows for optimizations like the inspector-executor to be used (see Chapter 7).

Implementing PR can be done via *pull-based* or *push-based* updates. In the pull-based approach, a given vertex reads the pr-values of its incoming neighbors and uses those to update its own pr-value. In the push-based approach, a given vertex "sends" its pr-value to its outgoing neighbors. To parallelize PR, the most straightforward approach is to pull updates from incoming neighbors. This is because it can be performed without any synchronization amongst the tasks. Specifically, each vertex can be processed in parallel and read the pr-value of its incoming neighbors to compute its new pr-value. However, there could be a race condition if another task is reading that now-changing pr-value as part of its own updates. To address this without synchronization, each vertex can store two pr-values, one used exclusively for reading and one used exclusively for writing. Then, when a vertex pulls updates from its

incoming neighbors, it uses the read version of the pr-values and then makes updates to its own write version of the pr-value. If the push-based approach was used instead, synchronization would need to be performed for the concurrent writes to the pr-values, and that cannot be addressed by using two versions for the pr-values.

Data Structures

Listing 3.4 presents the baseline Chapel implementation of the PR kernel. Like the previous graph analytics, PR uses the same block-distributed array of vertex records to represent the graph. Each vertex record provides a read-only pr-value (`pr_read`) and a write-only pr-value (`pr_write`). Additionally, each vertex stores an array of its incoming neighbors (`neighbors`) and the number of out-going neighbors (`out_degree`).

```
1 forall v in G {
2   var val = 0.0;
3   const ref neighbors = v.neighbors;
4   for i in neighbors.domain {
5     ref t = G[neighbors[i]];
6     val += t.pr_read / t.out_degree;
7   }
8   v.pr_write = (val * d) + ((1.0-d)/num_vertices) + sink_val;
9 }
```

Listing 3.4: Baseline implementation of the PageRank kernel.

Algorithm Implementation

Each iteration the PR algorithm executes the code in Listing 3.4. While not shown, there is an outer loop that runs the kernel until convergence is met. Between executions of the kernel, the `pr_write` values for each vertex are copied to the `pr_read` values. The entirety of the kernel is a `forall` loop that processes all the vertices in the graph `G` in parallel. Since `G` is distributed across the locales, the iterations of the `forall` loop on line 1 will execute across the locales as well. For each vertex `v`, the pr-values of its incoming neighbors (`pr_read`) are accumulated into a temporary `val`. Then, `v`'s new pr-value (`pr_write`) is computed as shown on line 8. The variable `d`

is an algorithmic constant set to 0.85 and known as the *damping factor*. The value of 0.85 is commonly used in literature as well as in Neo4j [Web12], an open-source graph database. The damping factor represents the probability that the hypothetical user who is clicking hyperlinks will continue to do so, while the probability that they will jump to a completely random page is `1-d`.

The variable `sink_val` is computed each iteration and allows for the correct handling of vertices with no out-going edges. Such vertices are referred to as *sinks* and can complicate the PR algorithm. To address this, the common approach is to assume that such sink vertices can link to all other vertices. As a result, their pr-values are added to all of the other vertices (hence the inclusion of `sink_val` in the calculation on line 8). The sink value needs to be computed each iteration as the sink vertices pr-values change between iterations.

Fine-grain Remote Communication

The fine-grain communication present in the PR algorithm is on line 5 in Listing 3.4. Specifically, this leads to potentially fine-grain remote reads. However, it can be optimized by COPPER through the adaptive remote prefetching optimization (Chapter 6) or the selective data replication optimization (Chapter 7). It is worth noting that the code in Listing 3.4 is nearly identical to the C++ OpenMP version of PR provided in the GAP Benchmark Suite [BAP15]. However, that implementation is only shared-memory parallelized while the Chapel code can execute across multiple nodes in a cluster. The fact that the OpenMP code could be adapted to a distributed-memory environment by simply block-distributing the graph illustrates the productivity benefits of the PGAS model. Likewise, the disadvantage of the PGAS model is also demonstrated because the ease of developing this code is the direct cause of the fine-grain communication on line 5.

With respect to the rest of the kernel in Listing 3.4, the fine-grain remote read on

line 5 will dominate the kernel's execution time. This is because all other accesses in the kernel do not require remote communication, and the access on line 5 is performed just as many times as these other accesses. As a result, optimizing line 5 is expected to provide large performance gains with respect to the overall runtime.

3.1.5 Triangle Counting

High-level Description

Within a graph, the most basic sub-graph that can be formed is a *triangle*, which consists of three mutually adjacent vertices. A common graph analytic is determining, or *counting*, the number of triangles present in a graph. Triangle counting (TC) can be used in community detection, link prediction and spam filtering [AD18]. While a triangle is a simple concept in a graph, it can be computationally expensive to perform triangle counting [Sam+17].

A straightforward approach to counting triangles is to iterate over each vertex u in the graph and its corresponding neighbor vertices v. Then for each neighbor v, iterate over its neighbors w and check whether w is also a neighbor of u. This effectively discovers the existence of the edges (u, v), (v, w) and (w, v), which forms a triangle and will be denoted as $\Delta(u, v, w)$. However, in the case of an undirected graph, this approach will count the triangle $\Delta(u, v, w)$ six times: $\Delta(u, v, w)$, $\Delta(u, w, v)$, $\Delta(v, u, w)$, $\Delta(v, w, u)$, $\Delta(w, u, v)$ and $\Delta(w, v, u)$. To address this, a common optimization is to ensure that a given vertex's neighbors are sorted by ID and only count a triangle if $u > v > w$. In this way, the algorithm can avoid searching through a given neighbor list once the IDs become too large to satisfy the above inequality.

Data Structures

Listing 3.5 presents the baseline Chapel implementation of the TC algorithm. As with the other graph analytics presented, TC uses a block distributed array of vertex

records to represent the graph. Each vertex record contains its assigned ID and an array of its sorted neighbor IDs. Worth noting is that in the case of the TC algorithm as described above, duplicate edges are required to be removed from the graph prior to the kernel.

```
1  forall u in G with (+reduce num_tri) {
2    const ref u_neighbors = u.neighbors;
3    for i in u_neighbors.domain {
4      const ref v = G[u_neighbors[i]];
5      if (v.id > u.id) then break;
6      const ref v_neighbors = v.neighbors;
7      var u_idx = 0;
8      for j in v_neighbors.domain {
9        const ref w = G[v_neighbors[j]];
10       if (w.id > v.id) then break;
11       while (G[u_neighbors[u_idx]].id < w.id) {
12         u_idx += 1;
13       }
14       if (w.id == G[u_neighbors[u_idx]].id) {
15         num_tri += 1;
16       }
17     }
18   }
19 }
```

Listing 3.5: Baseline implementation of the Triangle Counting kernel.

Algorithm Implementation

The code in Listing 3.5 is virtually equivalent line-for-line to the C++/OpenMP implementation of TC provided in the GAP Benchmark Suite [BAP15]. The `forall` loop on line 1 processes all vertices `u` in the graph `G` in parallel. Since `G` is distributed across the locales, the `forall` loop also executes its iterations across the locales. Each task that is created to execute the `forall` loop is given its own copy of the variable `num_tri`, which is initialized to 0. The `+ reduce` clause states that after the `forall` loop ends, each task's copy of `num_tri` will be sum-reduced into a single value, which represents the total number of triangles found. The `for` loop on line 3 iterates over the neighbors of `u` and the check on line 5 will break out of this loop if a given neighbor `v` has a larger ID than `u`. Otherwise, the `for` loop on line 8 iterates over `v`'s neighbors, where a similar check is performed on line 10 to break out if the neighbor

`w` has a larger ID than `v`. The `while` loop on line 11 iterates through `u`'s neighbors until it reaches a neighbor ID that is at least as large as `w`. Finally, if this neighbor ID is equal to `w`, then the triangle $\Delta(u, v, w)$ has been found.

Fine-grain Remote Communication

There are four sources of fine-grain remote communication in the TC implementation shown in Listing 3.5, namely lines 4, 9, 11 and 14. These irregular memory accesses can result in fine-grain remote reads and lead to performance issues. The accesses on lines 4 and 9 can be optimized via COPPER's adaptive remote prefetching optimization (Chapter 6). However, the accesses on lines 11 and 14 cannot be optimized via COPPER. To understand why requires explaining the lower level details of the adaptive remote prefetching and selective data replication optimizations, which will be provided in Chapters 6 and 7, respectively.

With respect to the rest of the kernel in Listing 3.5, the optimization candidates on lines 4 and 9 do not make up a majority of the memory accesses executed in the kernel. For a single iteration of the outer `forall` loop on line 1, line 4 is executed N times where N is the number of neighbors for a given vertex `u` and line 9 is executed $N{\cdot}M$ times where M is the number of neighbors for a given vertex `v`. However, the memory access on line 11 (`G[u_neighbors[u_idx]]`) is executed $N{\cdot}M{\cdot}O$ times, where O is the number of iterations of the `while` loop on line 11 (which can vary as the algorithm progresses). The end result is that the remote access on line 11, which cannot be optimized by COPPER, will execute more times than the accesses that COPPER can optimize (lines 4 and 9). This means that the overall performance gains that COPPER can provide may not be significant when compared to the other graph analytics presented.

As with the PR implementation in Listing 3.4, it was straightforward to start from a shared-memory parallelized TC algorithm and implement distributed-memory

parallelization by block-distributing the graph across the nodes. However by doing so, fine-grain communication naturally occurs, which underscores the conflict between the PGAS model's productivity benefits and their impact on performance.

3.1.6 Other Graph Analytics

In addition to the graph analytics described above, I also explored Connected Components (CC), Prim's Minimal Spanning Tree algorithm (MST) and Maximal Independent Set (MIS). These analytics are not included in the overall performance evaluation of COPPER because they either exhibit very similar memory access patterns with respect to the other analytics or performance is neither improved nor degraded via COPPER's optimizations.

CC as implemented via the Shiloach-Vishkin algorithm [SV80] provided by the Graph Based Benchmark Suite (GBBS) [Dhu+20] and can be optimized via adaptive remote prefetching (Chapter 6). However, CC consists of the same type of memory access pattern as PageRank (PR), where both analytics consider each vertex in parallel and access their respective neighbors. Prim's MST algorithm has a candidate for remote data aggregation (Chapter 5), but the parallel loop that contains the candidate iterates over a given vertex's neighbors rather than all of the graph's vertices. Such a loop will have relatively few iterations, which does not present much of an opportunity for aggregating many small messages. As a result, the aggregation optimization only provides a minimal performance gain. MIS has a candidate for adaptive remote prefetching, but performance gains are minimal when the optimization is applied. Each iteration of MIS (via Luby's algorithm [Lub85]) significantly reduces the number of active vertices to consider, which reduces the amount of remote communication to perform. With decreasing amounts of remote communication, prefetching remote data does not provide a performance advantage.

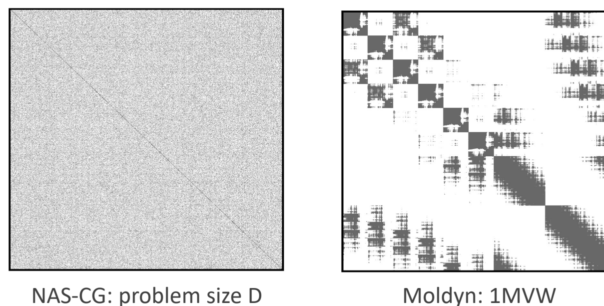

Figure 3.2: Sparsity patterns for some of the data sets that will be evaluated in this dissertation within the scientific computing domain. NAS-CG is a conjugate gradient benchmark and Moldyn is a molecular dynamics simulation.

3.2 Scientific Computing

Scientific computing refers to solving complex mathematical problems with advanced computing resources, such as modeling and simulation of systems with a large number of variables that far exceed what can be computed by hand. Some of the earliest applications to leverage high performance computing (HPC) resources included those performing scientific computation. As a result, scientific computing still makes up a majority of the HPC application landscape, spanning domains such as astrophysics [Boy+19], chemistry [Ken+00] and quantum computing [Doi+19]. Despite appearing vastly different from the graph analytic applications discussed in Section 3.1, irregular memory accesses also exist in scientific computing codes and pose the same types of performance challenges. This is largely due to the underlying nature of the problems being solved, which can often be thought of as graph or sparse matrix problems. As such, the optimizations COPPER provides can be applied. Figure 3.2 shows examples of the sparsity structure of these graphs/sparse matrices, which differ greatly from those shown in Figure 3.1.

In this section, I will describe two applications that will be studied in COPPER's performance evaluation, namely Conjugate Gradient (CG) and a molecular dynamics simulation (Moldyn). The difficulty in evaluating additional scientific computing

applications is that even benchmark or proxy codes for these applications can be thousands of lines of code. Furthermore, many of these codes have been developed over several decades, and as a result, are usually written in C or Fortran. Such codes do not align well with the goal of this dissertation, which is to improve both the performance and developer productivity for PGAS programs with irregular memory accesses. This is because these codes have been heavily optimized by hand, leaving little room for improvement via automatic compiler optimizations.

With respect to Chapel [Cha+18], which is a relatively new language and what COPPER is implemented in, there are only a few large-scale scientific computing applications written in Chapel. Examples include Chapel Multi-Physics Simulations (CHAMPS), which performs fluid dynamics simulations [Par+21], and ChplUltra [Pad+20], which performs astrophysics simulations for ultralight dark matter. However, CHAMPS has also been heavily optimized by hand and ChplUltra does not appear to exhibit the types of irregular memory access patterns that COPPER targets[1]. While the two applications presented below are relatively simple in comparison to other scientific codes, they contain the type of kernels that can be commonly found in other scientific applications.

3.2.1 Conjugate Gradient

High-level Description

The conjugate gradient (CG) method solves the equation $Ax = b$ for x, where A is a symmetric positive-definite matrix and x and b are vectors. The CG method can be used to solve unstructured optimization problems and partial differential equations. The iterative approach to CG is used when A is large and sparse. The bottleneck for CG is repeated sparse matrix-vector multiplies (SpMVs), where the memory access pattern remains the same across each SpMV (i.e., the structure of A remains

[1]CHAMPS and ChplUltra do not appear to be open-source projects.

fixed). There are existing benchmarks that use CG as a means to evaluate system performance, specifically in the presence of irregular memory access patterns. Examples include the NAS Parallel Benchmark suite [Bai+95] and the High Performance Conjugate Gradient (HPCG) benchmark [DHL16].

In the evaluation of COPPER, the NAS Parallel Benchmark suite is used. The NAS benchmark for CG, which will be referred as as NAS-CG, specifies the matrix A, the number of iterations to execute and the expected numerical answer. There are different problem sizes that can be evaluated, where problem size S refers to a small sample size, W for a single workstation size and then sizes A through F that dramatically increase in size. For example, problem size A specifies that the matrix A consists of 14,000 rows and 2 million non-zeros, while problem size F specifies that A consists of 54 million rows and 55 billion non-zeros. Worth noting is that the sparsity pattern for matrix A is generated to be uniformly random, where each row and column consists of roughly an equal number of non-zeros (see Figure 3.2). While this may not realistically model all problems, it provides useful insights into the performance of COPPER's optimizations, which will be discussed in Chapter 7.

Data Structures

Listing 3.6 presents the baseline Chapel implementation of the SpMV kernel in NAS-CG. This kernel represents the computational bottleneck of NAS-CG and is the target of COPPER's optimizations. The sparse matrix A is represented as a block-distributed array (`Rows`) of records, where each record represents a row in the matrix. Within a row record is a domain that stores offsets (`offsets`) into two block-distributed arrays, `values` and `col_idx`. The `values` array stores the values for the non-zero elements in the matrix and the `col_idx` array stores the column indices of those non-zero elements. Both arrays have the same length, which is equal to the number of non-zeros in A. Therefore, A is represented in a format similar to Compressed

Sparse Row (CSR). Rather than storing an explicit array of row pointers/offsets, as in traditional CSR, the offsets are stored individually within each row record as the `offsets` domain. The arrays x and b used in the CG method are represented as block-distributed arrays (`x` and `b` in Listing 3.6). Since A is a square matrix by definition, both `x` and `b` have the same length, which is equal to the number of rows in A.

```
1 forall row in Rows {
2   var accum : real = 0;
3   for k in row.offsets {
4     accum += values[k] * x[col_idx[k]];
5   }
6   b[row.id] = accum;
7 }
```

Listing 3.6: Baseline implementation of the Sparse Matrix-Vector Multiply (SpMV) kernel in NAS-CG.

Algorithm Implementation

The Chapel implementation of SpMV as shown in Listing 3.6 is virtually identical to the C++ OpenMP version of the NAS-CG algorithm [Löf+21]. The `forall` loop on line 1 iterates over all of the rows in the matrix in parallel, as each row can be processed independently. Since the array `Rows` is distributed across the locales, the `forall` loop will execute its iterations across the locales as well. For each row record `row`, the `for` loop on line 3 iterates over its non-zeros and performs a multiply-accumulate operation, storing the result into a temporary `accum`. The value in `accum` after the `for` loop is the dot-product between `row` and the input vector `x`. Line 6 stores the dot-product result into the output vector `b` at the index specified by `row`'s ID. The elements of `b` written to by a given `row` are guaranteed to be located on the locale processing `row` because `b` and `Rows` have the same length and distribution.

There are two potential sources of sequential/regular remote communication on line 4 in Listing 3.6. While each row record `row` is processed on the locale where it is stored, the `values` and `col_idx` arrays are block-distributed across the locales and

have different lengths from the distributed array Rows. As a result, the values and column indices accessed by a given row on line 4 may not be co-located on the locale where the row is. However, the potentially remote accesses to values and col_idx are sequential and are unlikely to lead to fine-grain communication. This is because Chapel's runtime is able to recognize the sequential access pattern and prefetch/read-ahead the remote data, leading to negligible overhead in the overall SpMV kernel. More details on this feature of Chapel's runtime are provided in Section 6.1.

Fine-grain Remote Communication

The accesses to values, col_idx and b in Listing 3.6 are either local or induce regular communication. However, the accesses to the input vector x on line 4 induce fine-grain remote communication. Specifically, the values in col_idx lead to sparse/irregular accesses to x, which is distributed across the locales. The performance bottleneck of the CG algorithm can be entirely attributed to this single indirect access pattern. COPPER can optimize this memory access pattern via adaptive remote prefetching (Chapter 6) or selective data replication (Chapter 7).

As observed with the PageRank and Triangle Counting kernels in Section 3.1, the C++ OpenMP code for SpMV was able to be parallelized for distributed-memory execution in Chapel by simply distributing the arrays. However, their distribution is what leads to the fine-grain remote communication on line 4. This highlights the way in which the PGAS model can provide productivity benefits when developing distributed codes with irregular memory accesses, but also naturally lead to performance issues.

3.2.2 Moldyn

High-level Description

A common scientific computing application that requires HPC resources is molecular dynamics simulations. Molecular dynamics simulations analyze how a system of particles evolves over time, where particles interact with other particles to influence their positions, forces and velocities. Particles can be atoms, molecules, or even large astronomical objects. The underlying kernel for molecular dynamics simulations is essentially operating on a graph, where an edge exists between two particles that interact with each other. The simulation performs iterations, which are referred to as time steps, that recompute each particle's forces, velocity and position based on the results from the prior time step. Updates to a particle's forces depend on the particles that interact with it and how close together in space they are. Typically, a simulation continues for some number of time steps before the particle interactions are updated, which changes the underlying graph representation of the system. In evaluating COPPER, I implemented the Moldyn proxy/benchmark in Chapel. Moldyn is derived from the CHARMM molecular dynamics application [Bro+09], which has been used in prior works for data/computation reordering optimizations [HT00; SCF03].

Data Structures

Listing 3.7 presents the baseline Chapel implementation of the main kernel in Moldyn, which updates each particle's forces based on their interactions with each other. The other kernels in Moldyn do not contribute much to the overall application runtime since they do not require remote communication. For Moldyn, the system of particles is represented as a block-distributed array of records (`Particles`), where each record stores the data associated with a given particle. This data includes the particle's

forces, velocities and position in the x, y and z directions. The record field that represents the forces is stored as an `atomic` variable because of the concurrent updates issued on lines 6 and 7 in Listing 3.7. Furthermore, each record has an array of particle IDs that it interacts with (`neighbors`). Each particle p only stores interactions with particle q if $p < q$ in terms of their IDs (i.e., only one direction of the undirected edge is stored).

```
1  forall p in Particles {
2    for j in 0..#p.num_neighbors {
3      ref q = Particles[p.neighbors[j]];
4      const rd = // distance between p and q
5      if rd < cutoff {
6        p.forces.add(..); // atomic add
7        q.forces.add(..); // atomic add
8      }
9    }
10 }
```

Listing 3.7: Baseline implementation of the interaction force updates kernel in Moldyn.

Algorithm Implementation

The interaction force update kernel shown in Listing 3.7 is essentially a single parallel `forall` loop that iterates over each particle `p` in the system. Since the array of particles (`Particles`) is distributed across the locales, the `forall` loop on line 1 also executes its iterations across the locales. For each particle `p`, the `for` loop on line 2 iterates over the neighbors of `p`, which are the particles that interact with `p`. For each neighbor particle `q`, line 4 computes the distance between `p` and `q`. If this distance is within the predetermined cutoff value (line 5), then the forces for `p` and `q` are updated (lines 6–7). This cutoff value is a constant set by the Moldyn benchmark. Because two particles may interact with the same neighbor particle, the updates on lines 6 and 7 must be atomic. Several lines of numerical computation are omitted from Listing 3.7 for brevity, but they do not cause remote communication.

Not shown in Listing 3.7 is the outer time step loop that executes the interaction force update kernel multiple times. Along with this kernel, there are other routines

that update the position and velocities of each particle. However, these updates are embarrassingly parallel and do not require any communication. After the specified number of time steps are completed, the simulation recomputes the interactions between the particles.

Fine-grain Remote Communication

The array access on line 3 in Listing 3.7 is the source of fine-grain communication in the Moldyn benchmark. This access is similar to what is observed in some of the graph analytics from Section 3.1, which is not surprising as the underlying representation of the particle system is a graph. This indirect array access can be optimized by COPPER's selective data replication optimization (Chapter 7). All other accesses in this kernel are to local data, so the fine-grain remote read on line 7 constitutes a majority of the runtime overhead of the Moldyn benchmark.

The complication of this particular scenario is that the data that will be replicated (`q`) is written to on line 7. Writing to replicated data presents a challenge, as the replicated copies must be "combined" together at the end of the kernel to reconstruct what the original numerical result would have been without replication. Details on how this is performed will be provided in Section 7.1.4.

3.3 Mini-kernels

Beyond the graph analytics and scientific computing applications described in Sections 3.1 and 3.2, I also use mini-kernels to evaluate COPPER's optimizations. These kernels represent relatively simple operations that are typically performed as part of larger applications. Note that the term "simple" in this context does not necessarily mean the amount of code required to implement the kernel is small. Instead, it refers to the scope of the kernel with respect to the overall application. However, despite

their simplicity, these mini-kernels do not achieve high performance due to fine-grain remote communication.

In this section, I present and describe two mini-kernels: histogram and graph construction. As in Sections 3.1 and 3.2, I present baseline Chapel implementations of these kernels and briefly describe where COPPER's optimizations apply.

3.3.1 Histogram

High-level Description

The term "histogram" in statistics refers to approximating the distribution of some numerical data. The act of *histogramming* data means to place individual elements into bins that correspond to some range of values. However, the underlying operation of histogramming appears often in parallel programming, such as when multiple processes need to send updates to some shared data structure. In the use cases that are of interest to this dissertation, the shared data structure is a distributed table (i.e., array). As histogramming does not require any particular ordering for the binning of data, the distributed updates can be performed in any order.

A histogram mini-kernel is used within the Bale package [2], which provides various mini-kernels and operations that are designed to drive the discussion of productivity in parallel programming. Bale specifies the size of the distributed array and the number of updates that each process will generate and perform. Updates are in the form of incrementing a random element of the distributed array in an atomic fashion. These updates are generated according to a uniform random distribution. Bale's histogram implementation forms the basis for the mini-kernel used to evaluate COPPER.

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

Data Structures

Listing 3.8 presents the baseline implementation of the histogram mini-kernel in Chapel. The array `table` is distributed in a cyclic manner across the locales, where the first element is placed on locale 0, the second on locale 1, etc. Each element in `table` is an `atomic int` and the number of elements is specified as `tableSize`. The array `index` is a block-distributed array of integers and represents the random updates that each process issues to `table`. In this context, a locale is considered a process, so `index` stores `numUpdates` elements per locale. The distribution of `table` and `index` as being cyclic and block, respectively, is specified by Bale.

```
1 var table = newCyclicArr({0..#tableSize}, atomic int);
2 var index = newBlockArr({0..#(numLocales*numUpdates)}, int);
3 index = // random ints between 0 and tableSize
4 forall i in index.domain {
5   table[index[i]].add(1);
6 }
```

Listing 3.8: Baseline implementation of the histogram mini-kernel.

Algorithm Implementation

The implementation of histogram via Listing 3.8 is straightforward and only consists of a single `forall` loop with one statement in its body. The `forall` loop iterates over the `index` array in parallel, and for each `index[i]` element, issues the atomic increment to `table[index[i]]` (line 5). The atomic nature of the updates prevents race conditions, and the fact that incrementing (i.e., addition) is a commutative operation ensures that the updates can be performed in any order.

Fine-grain Remote Communication

Despite the histogram kernel only consisting of a few lines of code in Listing 3.8, it contains an irregular memory access that will produce fine-grain remote communication, namely line 5. This memory access is a candidate for the remote data aggregation optimization that COPPER provides (Chapter 5), and in fact represents

the most primitive form of what COPPER can recognize for the aggregation optimization. As the histogram kernel does not perform any other memory accesses besides in line 5, significant performance gains are expected from the aggregation optimization.

3.3.2 Graph Construction

High-level Description

An important step that was omitted from the graph analytics that were presented in Section 3.1 is constructing the graphs that are used in the analytics. The task of graph construction is often omitted from performance studies and is considered a pre-processing step (much like reading in data from disk). Graph construction starts with a list of edges, where each edge is typically a tuple that specifies the two vertices at either end of the edge. This is often referred to as the *raw edge list* and represents the underlying structure of the graph. As only having the raw edge list is not very useful for performing operations on the graph, the construction process will produce a usable graph data structure such as an adjacency matrix or adjacency list.

However, when the graph is very large and requires a distributed data structure, the construction can be a significant performance bottleneck for the entire analytic. This can be seen by considering the Graph500 Benchmark [Bad+22], which specifies that one of the kernels to be evaluated is graph construction (denoted as kernel 1 in the Graph500 specification). However, despite being designated as its own kernel, the Graph500 results that are presented twice a year do not include the graph construction kernel. Instead, the Graph500 results only focus on Breadth First Search (BFS) and Single Source Shortest Path (SSSP). Furthermore, the reference C/MPI code that the Graph500 provides consists of an optimized graph construction kernel that leverages manual remote data aggregation. This indicates the level of difficulty, as well as necessity, to implement a high performance graph construction kernel.

Data Structures

Listing 3.9 presents the baseline Chapel implementation for the graph construction mini-kernel. The data structure that will hold the constructed graph `G` is a block-distributed array of records, where each record represents a vertex in the graph. The raw edge list `edges` is a block-distributed array of tuples, where the elements in the tuple refer to the end-points of an edge. Throughout the code in Listing 3.9, various temporary and auxiliary distributed arrays are created (`atomic_degrees`, `degrees`, etc.). The array `neighbors`, which is created on line 18, is a block-distributed array that stores neighbor IDs. Each vertex record in `G` has a local domain (`offset_dom`) that consists of offsets into `neighbors` that point to the vertex's neighbors. So the combination of `G` and `neighbors` represents the fully constructed graph. This is very similar to the data structure used to represent the sparse matrix for Conjugate Gradient, as described in Section 3.2.1.

```
var G = newBlockArr({0..#num_vertices}, Vertex);
var edges = // block-distributed array of edge tuples

// Phase 1: determine vertex degrees
var atomicDegrees : [G.domain] atomic int;
forall i in edges.domain {
  if edges[i](0) != edges[i](1) {
    atomicDegrees[edges[i](0)].add(1);
    atomicDegrees[edges[i](1)].add(1);
  }
}
// Copy out the atomicDegrees to a non-atomic array
var degrees : [G.domain] int;
...

// Phase 2: create vertex offset domains
var neighbors = newBlockArr({0..#num_edges}, int);
var tmpoffset : [G.domain] atomic int;
forall (v, i) in zip(G, G.domain) {
  v.id = i;
  v.num_neighbors = degrees[i];
}
var tempScan = (+ scan degrees);
G[0].offset = 0;
G[0].offset_dom = {0..#degrees[0]};
forall i in G.domain do if i != 0 {
  ref v = G[i];
  v.offset = tempScan[i-1];
  v.offset_dom = {v.offset..#v.num_neighbors};
  tmpoffset[i].write(v.offset);
}

// Phase 3: populate neighbors array
forall i in edges.domain {
  const u = edges[i](0);
  const v = edges[i](1);
  if u != v {
    neighbors[tmpoffset[u].fetchAdd(1)] = v;
    neighbors[tmpoffset[v].fetchAdd(1)] = u;
  }
}
```

Listing 3.9: Baseline implementation of the graph construction mini-kernel.

Algorithm Implementation

The graph construction mini-kernel in Listing 3.9 starts by creating the empty block-distributed array of records `G` on line 1, where each record represents a vertex. The raw edge list `edges` is created on line 2, where the exact method of how this is

performed is not shown. The raw edge list could be read in from a file or generated in-memory.

The first phase of the mini-kernel is to determine each vertex's degree (i.e., number of neighbors). Note that the graph being constructed is undirected. As a result, if the edge (u, v) exists in the raw edge list, then the edge (v, u) must also be included in the constructed graph. The `forall` loop on line 6 iterates over each edge-tuple in `edges` in parallel. As multiple tasks could attempt to update the degree for the same vertex at the same time, the update is performed atomically and the degree values themselves are stored as `atomic int` within the `atomicDegrees` block-distributed array (line 5). Line 7 will ignore self-edges, where both end-points of the edge are the same. Lines 8 and 9 atomically increment the current degree for each vertex associated with the given edge. After the `forall` loop, line 13 will create a non-atomic version of `atomicDegrees`, since it is not necessary for the degree values to be atomic throughout the rest of the mini-kernel.

Phase 2 of the mini-kernel involves constructing each vertex's offset into the block-distributed array `neighbors`. The `forall` loop on line 19 sets up each vertex `v`'s fields with its ID and number of neighbors. Line 23 performs a parallel prefix sum on the `degrees` array via the `+ scan` clause/operation. As a result, `tempScan[0]` is equal to `degrees[0]` and `tempScan[i]` is equal to `tempScan[i-1]` + `degrees[i]`. Lines 24 and 25 explicitly set up vertex 0's offsets. The `forall` on line 26 iterates over all the vertices in `G` except vertex 0. For each vertex `v`, the offset domain `offset_dom` is created on line 29. A given vertex's offset into `neighbors` starts where the "previous" vertex's offset ended and continues for however many neighbors the vertex has. Line 30 stores the starting offset for each vertex as an atomic integer within `tmpoffset`. The `tmpoffset` array is used during phase 3, which is described below.

The final phase of the graph construction mini-kernel is to populate the `neighbors` array with vertex IDs. This is accomplished with the `forall` loop on line 34, which

iterates over each raw edge tuple in parallel. For a given edge with end-points u and v, where u and v are not equal (i.e. not a self-edge), lines 38 and 39 place u and v into their respective location in neighbors. Initially, tmpoffset[u] stores the starting offset into neighbors for vertex u's neighbors. Therefore, the correct location for v within neighbors is neighbors[tmpoffset[u]], assuming the edge (u, v) is being processed. After this placement, the location for u's next neighbor will be one element after tmpoffset[u], so the value at tmpoffset[u] needs to be incremented. As multiple tasks may attempt to add u's neighbors in parallel, this increment must be atomic.

Fine-grain Remote Communication

The memory accesses on lines 8, 9, 38 and 39 are the only ones that generate remote communication. Specifically, they generate fine-grain remote communication in the form of irregular/indirect writes. These memory accesses are candidates for COPPER's remote data aggregation optimization (Chapter 5). Because no other memory accesses in the mini-kernel generate remote communication, optimizing the accesses on lines 8, 9, 38 and 39 would be expected to provide large performance gains.

Chapter 4

COPPER: Compiler Optimizations for Productivity and Performance

The main contribution of this dissertation is COPPER, a framework that performs Compiler Optimizations for Productivity and PERformance within the PGAS programming model. COPPER specifically addresses fine-grain remote communication that arises from irregular memory accesses to distributed data by applying three different runtime optimizations: aggregation, prefetching and replication. These optimizations are applied automatically via the compiler, so users are not required to modify their original program. This provides high performance and improves developer productivity by not burdening the user with the task of identify fine-grain communication in their program and manually applying optimizations. In this chapter, I will discuss the design and implementation of COPPER. Each of the three optimizations will be presented in their own respective chapters (Chapters 5 – 7).

4.1 Design

The goal of COPPER is to provide good performance for PGAS programs that exhibit irregular memory access patterns while not burdening the programmer by requir-

ing them to manually apply optimizations and/or rewrite their code. As described in Section 2.2, irregular memory access patterns to distributed data lead to fine-grain remote communication and present a significant challenge towards achieving high performance. Techniques such as aggregation and replication can be applied to improve performance, but at the cost of developer productivity. Specifically, programmers must identify where the fine-grain remote accesses are within their application and then manually perform the specific optimization.

One of the goals of the PGAS model is to improve developer productivity for distributed-memory applications by providing abstractions that present a shared-memory view of a distributed-memory system. However, manually applying optimizations to PGAS programs to achieve good performance undermines these productivity benefits. To achieve good performance for irregular PGAS programs and maintain the original productivity benefits of the PGAS model, COPPER performs a combination of static analysis and code transformations via the compiler to automatically optimize the code without any user intervention.

Static analysis is used to identify irregular memory access patterns to distributed data, which are likely candidates for fine-grain remote communication. For each optimization that COPPER supports, it checks whether a given irregular memory access pattern is "valid" for that optimization. The definition of valid is specific to each optimization and is discussed in detail within the chapters that present the optimizations. While COPPER supports three different optimizations, users are not required to know which optimization to apply in a given scenario. Instead, the interface to COPPER is a single compiler flag that is toggled on or off. The static analysis performed by COPPER will determine which optimization is suitable for a given irregular memory access. When an irregular memory access is a candidate for multiple optimizations, COPPER will attempt to apply the more powerful optimization first (i.e., the optimization which usually provides the most performance improvement). For example,

replication (Chapter 7) usually provides a larger benefit than prefetching (Chapter 6) due to effectively turning all remote accesses to local accesses, while prefetching only attempts to overlap remote communication latency with computation. More details on how these decisions are made will be presented in Chapter 8.

Because all details of irregular memory access patterns are not known at compile time, the optimizations that COPPER applies are *runtime optimizations*. These optimizations leverage information that only becomes known to the program once it runs, such as the sparsity structure of a sparse matrix or graph. To this end, COPPER applies code transformations at compile time to set up the optimizations, which will be enacted when the program executes. The code transformations are mostly centered around loop specialization, such as creating clones of a loop and modifying them to perform the optimization. However, COPPER also inserts several procedure calls to library functions that perform tasks that do not require specialization. Examples of such procedures include determining whether an array is distributed or whether a given access to an array results in remote communication. This reduces the complexity of the compiler infrastructure that COPPER is built on, as well as provides a means to extend and modify the framework in a modular way.

4.2 Implementation

The COPPER framework and accompanying optimizations are implemented as part of the Chapel compiler, which is written in C++. As described in Section 2.4, Chapel is a high productivity parallel programming language that implements the PGAS model. In this section, I describe Chapel's compiler workflow and how COPPER is integrated into it. Specific design and implementation details of each optimization will be presented in their respective chapters (Chapter 5 – 7).

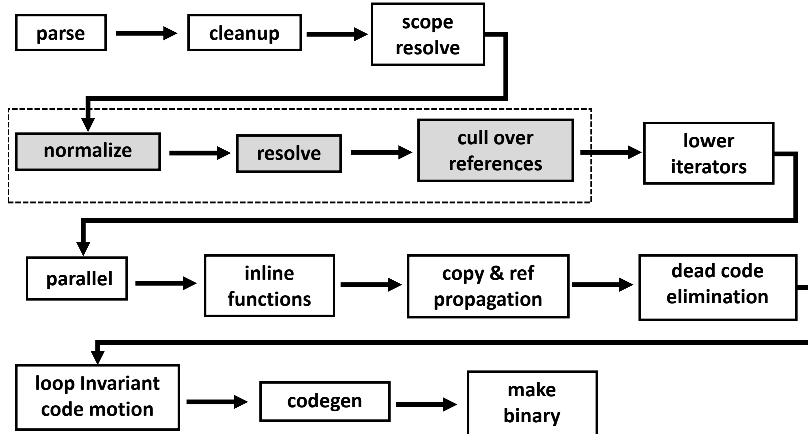

Figure 4.1: High-level overview of the Chapel compiler, where each box represents a pass performed by the compiler. The shaded passes within the dotted box represent those where COPPER performs code transformations and static analysis.

4.2.1 Existing Chapel Compiler Passes

To better understand how COPPER is integrated within Chapel's compiler, it is necessary to describe Chapel's compiler workflow. Figure 4.1 presents a high-level overview of the Chapel compiler at the time of this writing. Note that there are roughly 40 passes in Chapel's compiler, but most are omitted in Figure 4.1 for brevity. The Chapel compiler uses its own intermediate representation (IR) of the program and performs a majority of its optimizations on this IR. However, the code is eventually translated to the LLVM IR [LA04] during the codegen pass. By the time the code is lowered to LLVM, much of the Chapel/PGAS-specific context cannot be inferred. As a result, COPPER performs all of its static analysis and code transformations on Chapel's IR before it is translated to LLVM IR. The compiler passes where COPPER is implemented are *normalize, resolve* and *cull-over references*.

Normalize

The *normalize* pass is considered to be an early pass in the compiler, where a majority of the abstract syntax tree (AST) representation of the program has not been modified. Therefore it is much easier to reason about code structures when compared to

later passes, which begin to insert compiler-generated temporary variables that store intermediate values. Listing 4.2 shows the AST for the `forall` loop from Listing 4.1 during the normalize pass. It is straightforward to associate each line in the source code in Listing 4.1 to a portion of the AST in Listing 4.2. For example, lines 2–6 in Listing 4.2 declare the loop index variable `i` and assign it to the result of accessing `B`'s domain.

```
forall i in B.domain {
  C[i] += A[B[i]];
}
```

Listing 4.1: Example `forall` loop in Chapel.

```
ForallStmt
  DefExpr "index var" "insert auto destroy" unknown i:_unknown
  CallExpr
    UnresolvedSymExpr '.'
    SymExpr 'unknown B:_unknown'
    SymExpr "_dom" 'val _cstr_literal:c_string'
  BlockStmt
    CallExpr
      UnresolvedSymExpr '+='
      CallExpr
        SymExpr 'unknown C:_unknown'
        SymExpr 'unknown i:_unknown'
      CallExpr
        SymExpr 'unknown A:_unknown'
        CallExpr
          SymExpr 'unknown B:_unknown'
          SymExpr 'unknown i:_unknown'
```

Listing 4.2: AST of the `forall` loop from Listing 4.1 during the normalize pass.

However, a challenge with performing static analysis during the normalize pass is that variable types have not been resolved yet. Of particular relevance to COPPER are arrays, and during the normalize pass, array accesses look no different from procedure calls. One reason for this is that Chapel arrays are not primitive data types like an array in C/C++. Instead, they are implemented as objects/classes and accessing an array is implemented as a procedure call on the array object. Therefore during the normalize pass, an array access like $A[B[i]]$ is indistinguishable from $A(B(i))$, where A and B in this case are functions. This can be seen in Listing 4.2, where all of the variables (`A, B, C` and `i`) have unknown types.

The way that COPPER addresses this issue is by looking for candidate accesses that have the AST structure of a call expression whose argument is another call expression (e.g., lines 13–17 in Listing 4.2). This access is flagged within the AST as a candidate that can be further analyzed in later passes when the data types have been resolved.

```
ForallStmt
  DefExpr "const" "index var" "insert auto destroy" const i:int(64)
  CallExpr move
    SymExpr const i : int
    SymExpr unknown chpl__followIdx:int
  DefExpr "maybe param" "temp" ref call_tmp[1] : _ref(int)
  CallExpr move
    SymExpr ref call_tmp[1] : _ref(int)
    CallExpr
      SymExpr fn this : int
      SymExpr const-val _mt : _MT
      SymExpr unknown C:[BlockDom()] int
      SymExpr const i : int
  DefExpr "maybe param" "temp" ref call_tmp[2] : _ref(int)
    CallExpr move
    SymExpr ref call_tmp[2] : _ref(int)
    CallExpr
      SymExpr fn this : int
      SymExpr const-val _mt : _MT
      SymExpr unknown B:[BlockDom()] int
      SymExpr const i : int
  DefExpr "maybe param" "temp" ref call_tmp[3] : _ref(int)
  CallExpr move
    SymExpr ref call_tmp[3] : _ref(int)
    CallExpr
      SymExpr fn this : int
      SymExpr const-val _mt : _MT
      SymExpr unknown A:[BlockDom()] int
      SymExpr ref call_tmp[2]
  CallExpr
    SymExpr fn += : void
    SymExpr ref call_tmp[1] : _ref(int)
    SymExpr ref call_tmp[3] : _ref(int)
```

Listing 4.3: AST of the `forall` loop from Listing 4.1 during the resolve pass.

Resolve

The *resolve* pass is where data types and function calls are resolved. This can be seen in Listing 4.3, which shows the AST for the `forall` in Listing 4.1 during the resolve pass. This stage of the AST is much more complicated and more difficult to reason about in terms of code structure, as various temporary variables are created

to store intermediate results. The simple operation `C[i] += A[B[i]]` is broken into four separate stages: (1) move `C[i]` into temporary `call_tmp[1]` (lines 6–13), (2) move `B[i]` into temporary `call_tmp[2]` (lines 14–21), (3) move `A[call_tmp[2]]` into temporary `call_tmp[3]` (lines 22–29) and (4) perform the `+=` operation on the temporaries `call_tmp[1]` and `call_tmp[3]`.

While the AST is more complicated during the resolve pass, it provides type information for variables in the program. This enables more detailed static analysis, as an optimization could now determine whether A and B from $A[B[i]]$ are actually arrays. Such a query is crucial to the static analysis performed by COPPER, which exclusively applies optimizations to arrays.

Cull-over References

The *cull-over references* pass resolves the *intents* of function arguments, where an argument's intent refers to whether it is passed by value or reference. This pass statically sets the intent of the argument to different values depending on whether the procedure reads or writes to the argument. Accesses to arrays are carried out as function calls in Chapel (specifically the `this` function that can be seen in Listing 4.3). Therefore, the cull-over references pass will provide information about whether an array is modified or read-only within a given operation. For example, the `unknown` qualifier for the arrays `A`, `B` and `C` on lines 12, 20 and 28 in Listing 4.3 indicate that the compiler has not yet resolved their intents. During the cull-over references pass, these intents would be resolved to `ref`, `const ref` and `const ref`, respectively. The `ref` intent means that the reference to the array is modified and `const ref` indicates that the array reference is read-only. The importance of this information will be made more clear in Chapter 7 when the replication optimization is described.

4.2.2 COPPER's Workflow

This section presents a high-level description of how COPPER is integrated within the Chapel compiler passes that were described in Section 4.2.1. The optimizations that COPPER supports have unique implementation details in terms of static analysis and code transformations, and will be presented in their respective chapters (Chapters 5 – 7). It is worth noting that Chapel's existing compiler does not provide a majority of the infrastructure needed to perform the type of static analysis and code transformations needed by COPPER, such as recognizing non-affine array accesses, interprocedural analysis and alias analysis.

Identifying Candidate Memory Accesses

COPPER begins by performing static analysis during the normalize pass to identify optimization candidates that are within `forall` loops. The exact meaning of "within" is that the candidate must be directly within the `forall` loop's body (i.e., not within a procedure which is then called from the loop body). These candidates are memory access patterns that are likely to result in fine-grain communication, and `forall` loops are likely performance hot spots in applications where a majority of the runtime is spent. However, it is often not possible to statically determine whether a given memory access will result in remote communication, let alone fine-grain remote communication. Such access patterns are data dependent and only become known once the program executes.

To address this, COPPER looks for specific structures within the source code that are likely to produce fine-grain communication when the program runs. Specifically, all of the optimizations that COPPER supports rely on the same underlying non-affine memory access pattern, which is of the form $A[B[i]]$, where A and B are arrays, and A is a distributed array. More complex access patterns are supported and will be described in each optimization's chapter (Chapters 5 – 7). Requiring that the array

A is distributed increases the likelihood that there will be remote communication. Furthermore, the values in the array B are often not known until runtime which then requires runtime-based optimizations. Such access patterns are found in graph analytics [BGS05], sparse linear algebra [DHL16] and other scientific computing applications [MWK99]. This fact is made clear in Chapter 3, which presents a variety of applications and kernels that exhibit the $A[B[i]]$ access pattern.

Replacing Candidates with Compiler Primitives

As noted in Section 4.2.1, accesses to arrays cannot be inferred during the normalize pass in Chapel's compiler. The AST that represents $A[B[i]]$ where A and B are arrays will look identical to the AST that represents $A(B(i))$ where A and B are functions. To handle this issue, COPPER will "flag" the candidate access within the AST by replacing it with a *compiler primitive*. A compiler primitive is a user defined AST node that can be customized to store various information, where "user" in this context is the programmer developing the compiler analysis/optimization. Each of the optimizations that COPPER supports has its own compiler primitive defined, which stores the original candidate access information (i.e., the symbols for A and B) as well as additional information that is specific to the optimization. These primitives will be processed during the resolve pass, where the information needed to perform additional static analysis will be available. Note that if a given memory access pattern is a candidate for multiple optimizations, then the order in which the optimizations are applied becomes crucial. This is because the first optimization applied would replace the candidate with its own primitive, which would prevent subsequent optimizations from detecting the candidate. Chapter 8 discusses the details of how COPPER deals with the ordering of optimizations.

Code Transformations

Most of the code transformations that COPPER applies are performed during the normalize pass. It is at this stage in Chapel's compiler that it is easiest to reason about the original program's structure, and therefore it is easiest to make changes to the program's structure. The code transformations that COPPER performs are centered around loop specialization. An example is cloning a `forall` loop that contains a candidate memory access and modifying it in some way. COPPER will also insert calls to Chapel procedures that were written to support the optimizations or certain aspects of static analysis. These procedures are fairly general and do not require on-the-fly generation via the compiler. An example would be checking whether a given access is local or remote at runtime. Because of this, many aspects of COPPER are modular and easily modified without changing the C++ compiler implementation code.

```
1  if isArray(A) && isArray(B) { // resolved at compile time
2    forall i in B.domain {
3      // optimized code here
4    }
5  }
6  else {
7    forall i in B.domain {
8      // original, unoptimized code here
9    }
10 }
```

Listing 4.4: COPPER compile time control flow code transformation example.

Another common code transformation that COPPER performs is setting up control flow logic that will be resolved at compile time. Because Chapel's compiler passes operate in stages, where certain information (e.g., data types) is not known at a given point, it is necessary to proceed with optimization-specific code transformations during the normalize pass, but "undo" those changes if something is found to invalidate the optimization. Again, performing code transformations, like loop cloning, is significantly easier during the normalize pass when compared to doing it during the resolve pass. Listing 4.4 shows a simple example of a compile time control flow transforma-

tion, where an if-statement will check whether `A` and `B` are both arrays. If that check is true, the optimized version of a `forall` loop will execute; otherwise the original `forall` loop will execute. However, the checks performed by the conditional are compile time checks, meaning that they can be resolved statically once the types of `A` and `B` are determined during the resolve pass. Therefore, what actually happens in the example in Listing 4.4 is that by the time Chapel's compiler begins the resolve pass, the conditional will be known to be true or false (i.e., it will literally be `if true` or `if false`). Chapel's compiler can then remove the branch that is not taken, thereby keeping the optimized `forall` if the check is true or the original `forall` if the check is false. This allows for COPPER to perform code transformations for an optimization before it can be known whether the optimization is valid to apply.

Complex Static Analysis

The resolve and cull-over references passes are where COPPER performs a majority of its complex static analysis. It is during these passes that data types and read-write intents have been resolved, as well as the creation of the program's call graph. With this information known, COPPER can determine with more certainty the validity of an optimization. Beyond determining that the memory access pattern involves array accesses, there are other static checks that involve both alias and interprocedural analyses. For example, a given optimization may require that there are no modifications (writes) to a given array within the `forall` loop. This check must also identify modifications to aliases of that array (i.e., pointers/references). Furthermore, the `forall` loop may issue a call to a procedure that modifies the array, so the analysis must work across procedure calls. More details of these analyses will be provided when discussing each optimization (Chapters 5 – 7).

Relatively simple checks, such as those in Listing 4.4, can be performed at compile time, where Chapel's existing compiler infrastructure will "clean up" the unreachable

code. However, some checks require significantly more complicated analysis that cannot be performed as a Chapel procedure call like the `isArray()` procedure. The scenario given above of looking for modifications to an array across aliases and procedure calls is one example. Such analysis requires inspecting data flow information as well as the program's call graph, which can only be performed within the compiler itself, and only during the resolve and cull-over references passes. As a result, COPPER must perform code removal for an invalid optimization's code transformations. While adding new code structures during the later passes in Chapel's compiler is very complicated, removing code is relatively straightforward.

Additionally, some of the checks needed by optimizations to determine validity cannot be easily resolved at compile time, such as requiring that the `forall` loop is nested within an outer loop along all possible call paths. As a concrete example, consider the code in Listing 4.5 which has a procedure `func` that contains a `forall` loop that is being optimized via COPPER. The procedure `func` is called from within the `for` loop on lines 8–10, as well as outside of the `for` loop on line 11. In this case, it is too conservative to invalidate the entire optimization because of the invalid call path to `func` on line 11. To address this, COPPER must essentially "turn off" the optimization along the invalid call path while keeping the optimization "on" along the valid call path (lines 8–10). While this can be done with loop/procedure cloning at compile time, it quickly becomes very complicated and leads to a large increase in the program's code size. Instead, COPPER will insert checks that are performed at runtime that can toggle the optimization on or off depending on the call path taken. An example of this runtime control flow analysis and transformation is shown in Listing 4.6, which uses the code from Listing 4.5 as an example. In this case, COPPER performs static analysis to walk the call graph and identify the outer loop that is on the path which leads to the `forall` loop in the procedure `func`. COPPER then inserts procedure calls in the outer loop to toggle a flag at runtime (lines 16

and 18). This flag is checked at runtime to determine whether the optimized `forall` should execute or not (line 3).

```
proc func()
{
  forall i in B.domain {
    ...
  }
}

for i in 0..#10 {
  func();
}
func();
```

Listing 4.5: Example code for COPPER's runtime control flow analysis and transformation.

```
proc func()
{
  if outerLoopFlagIsSet() { // checked at runtime
    forall i in B.domain {
      // optimized code here
    }
  }
  else {
    forall i in B.domain {
      // original, unoptimized code here
    }
  }
}

for i in 0..#10 {
  setOuterLoopFlag();
  func();
  unsetOuterLoopFlag();
}
func();
```

Listing 4.6: Example output of COPPER's runtime control flow analysis and transformation for the code in Listing 4.5.

Chapter 5

Remote Write Aggregation

One of the major performance issues with fine-grain remote communication is the cost of sending many small messages. Preparing a message to be sent over the network and processing the message on the receiver side incurs a non-negligible amount of overhead. As a result, programmers with in depth knowledge of the application being developed strive towards sending fewer larger messages to amortize the overhead of sending data over the network. This is often referred to as *aggregation* and is a common optimization technique for applications that exhibit fine-grain remote communication [Kay+21; Alv+13; Mor+14]. Aggregation can be applied to PGAS programs by "delaying" individual `puts` and `gets` to a given destination node until enough of them can be performed within a single message. Essentially, these small messages are buffered locally until the buffer is full, at which point the buffer is *flushed* to the remote destination in a single message. However, this requires knowledge of "how long" the `puts` and `gets` can be delayed before another statement in the program requires that they have been completed.

Applying aggregation by hand places a burden on the programmer to know when and how to apply aggregation effectively. Determining which operations lead to remote communication and when those operations must be completed to ensure program

correctness is difficult and usually requires expert knowledge of the algorithm/application. The problem can be complicated by PGAS languages like Chapel [Cha+18], where remote communication is performed implicitly by the runtime system. This means that a programmer is not exposed to the lower level `puts` and `gets`, which makes it difficult to know which statements in the program are producing remote communication. Additionally, the remote communication pattern (when identified) can be data dependent, which is often the case for irregular memory access patterns. Challenges also arise when considering the underlying system and network architecture. The number of compute nodes being used for an application, the communication latency between these nodes and the available network bandwidth all influence the effectiveness of aggregation. In the end, these application and system-level factors contribute to the difficulty of manually applying aggregation.

In this chapter, I present an optimization that performs aggregation specifically for remote writes. This optimization is applied automatically to Chapel programs via the COPPER framework described in Chapter 4. Aggregation is performed at the application level by identifying high level operations that can be aggregated. I present details on the approach taken by COPPER to perform aggregation as well as performance results across a suite of irregular applications that were described in Chapter 3. The results show that runtime speed-ups as large as 87x can be achieved via aggregation without requiring any user intervention. Much of the work presented in this Chapter was also presented in prior papers that were published [Rol+21] or under submission [RS] at the time of this writing.

5.1 Approach

In this section, I describe the overall approach taken by COPPER to perform automatic aggregation for remote writes. At a high level, COPPER employs a combination of

static analysis and code transformations to identify valid candidates for aggregation and modify the original program to perform aggregation. Aggregation candidates must be within `forall` loops, and aggregation is performed on a per-task basis. This means that each task that is executing some portion of the `forall` iterations has its own aggregation buffers that are independent from the other tasks. This will be discussed in more detail in Section 5.1.2. In this way, aggregation is performed with respect to a specific statement/operation within the `forall` loop. This is in contrast to how aggregation is usually performed within the communication runtime, which considers arbitrary `puts` that arise from any number of different statements. Such low-level forms of aggregation can have limited effect due to the lack of knowledge of the high-level operations which produced the `puts`.

5.1.1 Static Analysis

The static analysis performed by COPPER for the aggregation optimization has three steps: (1) identifying candidate memory access patterns in `forall` loops, (2) determining whether the operation can be aggregated and (3) using data dependency analysis to ensure that no other operation in the `forall` loop relies on the outcome of the operation to be aggregated. The details and reasoning behind these steps will be discussed below. All of the static analysis is performed during the normalize pass in Chapel's compiler (see Section 4.2.1). It is worth noting that while there is an existing aggregation optimization in Chapel's compiler [Kay+21], COPPER extends it by providing new capabilities, which will be highlighted below.

Identifying Candidate Memory Access Patterns

Static analysis begins by identifying potential candidates for aggregation, where these candidates must be contained inside of `forall` loops. Such loops are usually where a majority of an application's runtime is spent. But more importantly, as noted

in Section 2.4.2, the iterations of a `forall` loop are order independent. Note that the candidate must be directly within the `forall` loop's body (i.e., not within a procedure which is then called from the loop body). Additionally, candidates can be within `foreach` loops that are nested in `forall` loops, since `foreach` loops also imply that iterations are order independent. This order independence property is crucial for COPPER's static analysis, as it asserts that there are no data dependencies between loop iterations. The importance of this property will be expanded upon when discussing the remaining two static analysis steps.

The optimization specifically targets irregular writes and looks for non-affine access patterns of the form $A[B[i]]$ where A and B are arrays and A is a distributed array. These access patterns are likely to result in fine-grain remote communication, and their access pattern cannot be inferred at compile time. More generally, the optimization operates on distributed arrays that are accessed by an expression that involves another array access, where the expression can be arbitrarily complex and contain constants, variables inside and outside the scope of the loop, procedure calls, etc. (e.g., $A[B[i + func(j)] + k]$). For brevity, I will consider the simplest form, $A[B[i]]$, for the rest of this discussion. By enforcing that the array A is distributed, there is the possibility of remote communication.

Determining Order Independent Operations

In addition to identifying candidate access patterns, the static analysis must also consider the overall operation being performed and whether it is valid for aggregation. In general, an operation is valid for aggregation if a sequence of these operations can be performed in any order. In other words, the operation can be performed as part of a generalized reduction [LM98]. This simplifies the static analysis that is required to ensure that aggregation will be valid, as maintaining the original order of the operations is not necessary. However, statically inferring whether an arbitrary

operator is order independent is not possible, unless the language provides information about the operator's properties.

COPPER maintains a list of operators that are known to be order independent and uses that list to determine whether a given operation can be aggregated. There are limitations imposed on the optimization due to having to rely on such a list, which is discussed in more detail in Section 5.3. The list of operators includes known order independent arithmetic operators (addition, multiplication), addition between atomic variables, updates to associative domains via the `+=` operator (see Listing 2.3) and assignment (`=`).

While operations like addition and multiplication are known to be order independent, their direct use in parallel loops (i.e., `forall` loops) could lead to data races, which would be the responsibility of the programmer and exist with or without aggregation applied. For example, consider the statement `A[B[i]] += k` that is in a `forall` loop, where `i` is the loop index variable. Multiple tasks could end up performing the addition to the same element of `A` concurrently (e.g., `B[i]` and `B[i+i]` could hold the same value), which would lead to a race condition. As a result, the more common arithmetic operations to find in parallel loops are *atomic operations*. Chapel allows for variables to be declared as `atomic`, which can then support atomic operations. For example, `A[B[i]].add(k)` atomically adds `k` to the value stored at `A[B[i]]`. This means that only one task/thread is allowed to perform the addition at a time.

Updating an associative domain via the overloaded `+=` operator is order independent as well, since associative domains do not enforce any ordering on the elements. Additionally, updates to associative domains are parallel-safe by default. However, users can declare associative domains with parallel safety disabled. It would be their responsibility to deal with the race conditions that could arise when performing parallel updates in a `forall` loop. Again, such a race condition would be present with

or without aggregation applied.

The reason why statements involving the assignment operator can be aggregated will be made more clear when the data dependency analysis is discussed. To summarize, as long as the array being assigned to is not accessed after the assignment within the scope of the `forall` loop body, then the assignment can be performed in any order. This is because the `forall` loop iterations are asserted to be order independent, so it is irrelevant to COPPER whether another loop iteration that is executing in parallel could be assigning to the same array element. Such a data hazard is the responsibility of the programmer and would exist whether aggregation was applied or not. This is the same reasoning why non-atomic addition and multiplication are valid.

Data Dependency Analysis to Detect Data Hazards

A majority of the static analysis that COPPER performs for aggregation focuses on ensuring that there are no data dependencies present in the loop that would lead to incorrect program behavior when aggregation is applied. Specifically, aggregation cannot be applied if the out-of-order execution of the operation would lead to data hazards (read-after-write and write-after-write) within the scope of a loop iteration. To address this restriction, COPPER performs data dependency analysis to ensure that no operations in the loop that follow the aggregation candidate read or write the data array (i.e., A in $A[B[i]]$). When the aggregation candidate is within a `foreach` loop that is nested in a `forall` loop, the analysis considers the operations following the candidate within the body of the `foreach` loop, as well as the operations that come after the `foreach` loop itself. This is because aggregation is always applied with respect to the `forall` loop.

In addition to looking at the statements in the loop that follow the aggregation candidate, COPPER performs interprocedural analysis to detect accesses to the data

array across procedure calls, as well as alias analysis to detect accesses that are made to pointers/references to the data array. Recall that COPPER performs all static analysis for aggregation during the normalize pass in Chapel's compiler. During this pass, details such as argument intents are not known yet, which provide information about whether a variable is accessed via a read or write. However, the analysis required for aggregation is simply to detect whether the data array is accessed, which could be a read or write. As a result, knowledge of argument intents is not needed.

One exception to the requirement that the data array is not accessed after the aggregated statement is when the subsequent statement is valid for aggregation itself, and the two candidates involve atomic operations (e.g., atomic addition, updating an associative domain). In such a case, the two statements are order independent with respect to each other, since they are both valid for aggregation and are performed atomically with respect to the data array. If instead there were two assignment statements that are considered for aggregation, only the one that appears later in program order could be aggregated. Note that in order for this type of analysis to be done efficiently, COPPER considers aggregation candidates in a given `forall` loop in reverse order (i.e., it starts at the last statement in the loop and searches backwards).

As discussed previously, when users write a `forall` loop they are asserting to the compiler that the loop iterations are order independent. As a result, it is assumed that there are no loop carried dependencies. This significantly simplifies the data dependency analysis that must be performed, as it allows for COPPER to only consider the data dependencies that exist within the scope of a single loop iteration. This also applies to `foreach` loops, which also assert order independent loop iterations.

Listing 5.1 illustrates scenarios when aggregation is valid. The listing uses the atomic add operator as an example, where `A[B[i]].add(i)` is the aggregation candidate. However, any other supported operator discussed earlier would be applicable as well. The `forall` loop on lines 1–4 in Listing 5.1 is the simplest example of valid

aggregation, as there are no statements after the aggregation candidate (note that `.write()` is how to assign to atomic types in Chapel). The `forall` loop on lines 5–9 contains a valid aggregation candidate because the array `A` is not accessed after the candidate. While the array `B` is modified and `B` is used when accessing `A`, it is valid because when aggregation is ultimately performed, the expression `B[i]` within `A[B[i]]` is resolved in normal program order. In other words, aggregation is delaying the potential remote addition of `i` to the address pointed to by `A[B[i]]`, not the evaluation of sub-expressions within the statement. Likewise, the `forall` on lines 10–17 is contains a valid candidate since `A` is not accessed after the aggregation candidate within the `foreach` loop nor in the remainder of the `forall` body. Finally, the `forall` loop on lines 18–21 has a valid candidate because both statements in the loop can be aggregated and performed in any order with respect to each other.

```
1  forall i in B.domain {
2    A[B[i]].write(i);
3    A[B[i]].add(i);
4  }
5  forall i in B.domain {
6    A[B[i]].add(i);
7    C[i] = B[i];
8    B[i] = i-1;
9  }
10 forall i in B.domain {
11   foreach j in C.domain {
12     C[j] = i;
13     A[B[i]].add(i);
14     B[i] = j;
15   }
16   B[i] = i-1;
17 }
18 forall i in B.domain {
19   A[B[i]].add(i);
20   A[C[i]].add(i);
21 }
```

Listing 5.1: Examples of valid aggregation candidates, where `A[B[i]].add(i)` is the aggregation candidate.

Listing 5.2 illustrates scenarios when aggregation is not valid. The listing also uses the atomic add operator as an example, where `A[B[i]].add(i)` is the aggregation candidate. The `forall` loop on lines 1–4 is contains an invalid candidate because `A`

is accessed after the aggregation candidate (note that `.read()` is how to read atomic types in Chapel). Likewise, the `forall` on lines 5–10 has an invalid candidate because `A` is accessed after the aggregation candidate, which is in a `foreach` loop.

```
1 forall i in B.domain {
2   A[B[i]].add(i);
3   B[i] = A[B[i]].read();
4 }
5 forall i in B.domain {
6   foreach j in C.domain {
7     A[B[i]].add(i);
8   }
9   C[i] = A[B[i]].read();
10 }
```

Listing 5.2: Examples of invalid aggregation candidates, where `A[B[i]].add(i)` is the aggregation candidate.

5.1.2 Code Transformations

The code transformations performed by COPPER for the aggregation optimization have the following steps: (1) creating per-task aggregation objects, (2) generating a procedure that performs the operation to be aggregated and (3) replacing the original operation with a call to the aggregation object. Almost all of the code transformations are performed during the normalize pass within Chapel's compiler, where modifying the structure of the program is less challenging when compared to later compiler passes. Listing 5.3 presents an example `forall` loop that contains a valid aggregation candidate (line 3), followed by code that is equivalent to the output of COPPER's code transformations. Explanations of each step of the code transformations are provided below.

```
// Original forall loop
forall i in B.domain {
  A[B[i]].add(i);
}


// Code equivalent to the output of the code transformations
inline proc op(ref lhs, const rhs) {
  lhs.add(rhs);
}
forall i in B.domain with (var agg = newAggr(A, op)) {
  param check : bool = isAggregationSupported(A);
  if check {
    agg.bufferOp(A[B[i]], i);
  }
  else {
    A[B[i]].add(i); // original operation
  }
}
```

Listing 5.3: Examples of an aggregation candidate and the resulting code transformations applied by COPPER. `A[B[i]].add(i)` is the aggregation candidate.

Creating a Per-task Aggregation Object

The first code transformation step creates a per-task aggregation object, which is denoted as `agg` on line 11 in Listing 5.3. The `with` clause in Chapel enables the creation of per-task variables within the scope of a `forall` loop, so each task has its own copy of `agg` that is independent from the others. Because the iterations of a `forall` loop are asserted to be order independent, and the supported aggregation operators are known to be order independent, each task can consider its chunk of iterations independently from the other tasks. To this end, each task has its own aggregation object, and tasks do not communicate or share aggregation-related data amongst each other. The lifetime of `agg` does not extend beyond the scope of the `forall` loop. The procedure `newAggr()` is implemented as part of a generalized aggregation module/library in Chapel and allows for the creation of aggregation objects based off of the data array (`A` in Listing 5.3) and the operation to aggregate, which will be discussed in more detail below. The main interface to the aggregation object is the `bufferOp()` method (line 14), which adds the specified operation to the task's

aggregation buffers.

Generating Aggregation Procedures

As noted in Section 5.1.1, COPPER supports a list of operators that can be aggregated. Given this fact, it would be possible to develop code for a finite set of aggregation objects that are specialized for each operator. For example, we could create an aggregation object for atomic additions and an aggregation object for assignments. This is what the existing aggregation optimization in Chapel's compiler does, where assignment is the only operator supported [Kay+21]. However, this approach can quickly become unscalable, as supporting new operators requires the manual creation of new aggregation object implementations. Also, the only difference between the code that implements these specialized aggregation objects is which operation is performed when flushing a buffer.

To address this issue, COPPER's code transformations will automatically generate a procedure that performs a generalized version of the operation being aggregated. Lines 8–10 in Listing 5.3 represent this generated procedure, `op()`, for the atomic addition operator. The procedure's arguments are a modifiable reference to the left-hand side (LHS) of the operation and a read-only right-hand side (RHS) value. In the case of `A[B[i]].add(i)`, the LHS is `A[B[i]]` and the RHS is `i`. The body of the procedure performs the specified operation (`.add()`) on the LHS and RHS. This can be generalized to any of the supported aggregation operators: `LHS = RHS`, `LHS += RHS`, etc.

There are syntactic details related to these aggregation procedures that are worth discussing. Ideally, these procedures would be generated as shown in Listing 5.3 and passed into the aggregation object constructor `newAggr()` as first-class functions (i.e., function pointers). However, Chapel's support for first-class functions at the time of this writing is not sufficient for what is required in the case of aggregation.

To address this limitation, what COPPER actually generates is syntactically different from the code shown in Listing 5.3. Specifically, COPPER creates a record (i.e., struct) that contains a special method that performs the aggregated operation. This method is referred to as a *type* method, as it is invoked on the record type, not an instance of the record itself. As this is a minor detail that only affects syntax, which is never observed by the programmer since the optimization is applied automatically, the remainder of the discussion will assume that the aggregation procedure is passed in as a first-class function.

Replacing the Original Operation

The final code transformation step is to replace the original operation/statement with a procedure call that performs aggregation. However, the code transformations and static analysis performed for this optimization occur during the normalize pass in Chapel's compiler. Because of this, some of the information that is needed to determine whether aggregation is valid is not available yet. Specifically, it is not known whether the data array `A` is in fact an array and whether it is a distributed array. These properties are necessary for COPPER to apply aggregation.

To address this issue, COPPER applies a compile time control flow transformation, which was described in general terms in Section 4.2.2. Lines 12–18 in Listing 5.3 show this transformation specifically for the aggregation optimization. Line 12 declares a *param* variable called `check` that holds the boolean (true/false) return value of the procedure `isAggregationSupported()`, which simply checks whether `A` is an array and is distributed. The param qualifier specifies that `check` will be resolved at compile time rather than runtime. Specifically, by the time Chapel's compiler moves to the resolve pass, `check` will be known to the compiler as either true or false. As a result, the conditional on lines 13–18 can be resolved at compile time, which will either remove the else-branch if `check` is true or the then-branch if `check` is false. This

has the effect of "undoing" the code transformations if it turns out that aggregation cannot be supported, leaving the original operation in its place.

Assuming that aggregation is valid, the original operation is replaced with what is on line 14 in Listing 5.3. This statement invokes the `bufferOp()` method on the aggregation object `agg`, passing in the LHS and RHS arguments of the operation to aggregate. Within the library code that implements the aggregation objects, `bufferOp()` determines at runtime which locale the LHS argument is on (i.e., which locale `A[B[i]]` is on). With this information, the library can add the tuple (LHS, RHS) to the appropriate buffer that is destined for the specified locale. Note that what is included in the tuple is the address of the LHS, which does not require remote communication to determine. After the tuple is added, the library checks whether the buffer for the specified locale is full. If it is, it invokes the flushing procedure, which performs a bulk-copy of the buffer's data to the specified locale and then performs the original operation on the data. This effectively performs the operation within a single message rather than several small messages.

Listing 5.4 presents the library code that is executed whenever a given buffer is flushed. The locale to flush to (`loc`) and the number of operations to flush (`bufferSize`) are specified when the library invokes this routine. Line 20 is where the `op()` procedure is called, which COPPER has generated to perform the original operation (lines 8–10 in Listing 5.3). Note that this buffering and flushing of data is performed per-task and is blocking, which means that the task is not performing any other computation while flushing occurs. Also, multiple tasks may be flushing to the same destination locale at the same time, which further requires order-independent operations. Finally, there is an implicit flush of all the buffers within an aggregation object when it is deinitialized/destroyed. Since the aggregation objects are created as per-task variables within the `forall`, they are deinitialized at the end of the `forall` loop's scope. This ensures that all outstanding operations have been flushed and

completed when the `forall` loop completes.

```
proc flushBuffer(const loc : int, ref bufferSize)
{
  // Buffer is empty, so nothing to flush
  if bufferSize == 0 then return;

  // Set up the remote buffer on locale loc
  // rBuffers is the array of remote buffers, one-per locale.
  ref rBuffer = rBuffers[loc];
  const rBufferPtr = rBuffer.cachedAlloc();

  // Do a bulk copy of local buffered data to the remote buffer.
  // lBuffers is the array of local buffers, one-per locale.
  // PUT is an optimized remote-copy routine.
  rBuffer.PUT(lBuffers[loc], bufferSize);

  // On the destination local, process the remote buffer.
  // For each item in the buffer, call op().
  on Locales[loc] {
    for (dstAddr, srcVal) in rBuffer {
      op(dstAddr.deref(), srcVal);
    }
  }
  // Logically empty the buffer
  bufferSize = 0;
}
```

Listing 5.4: Library code that is called to flush an aggregation buffer. The argument `loc` specifies which locale to flush to and `bufferSize` is the current size of the buffer.

5.2 Performance Evaluation

This section presents a performance evaluation of COPPER's aggregation optimization. The goal of the evaluation is to demonstrate that significant performance improvements can be achieved via COPPER on baseline implementations of the irregular applications described in Chapter 3. These baseline codes fully leverage the productivity advantages of the PGAS model and Chapel, but suffer severe performance issues due to fine-grain communication. As the runtime improvements are achieved without modifying the original code, these results demonstrate that performance gains can be obtained without sacrificing user productivity, enabling users to develop irregular applications without having to deal with the issues posed by fine-grain communication.

5.2.1 Experimental Setup

The experiments presented in this section were executed on a 32-node FDR Infiniband cluster. Each node consists of two 10-core Intel Xeon E5-2650 CPUs and 512 GB of DDR4 memory. Each node in the cluster is considered a single locale within the context of a Chapel program, and the terms “locale” and “node” will be used interchangeably. Given that each node in the cluster has 20 compute cores, all experiments use up to 20 tasks per locale when executing parallel `forall` loops. All applications were compiled using Chapel version 1.28.0, which has been modified to implement the COPPER framework and the aggregation optimization.

Chapel allows users to specify the communication layer to be used, which provides the implementation for how communication is performed between locales. For all experiments, GASNet [BH17] was used as the communication layer, which is a one-sided communication and active message library. Additionally, Chapel allows users to specify the underlying GASNet conduit to use when GASNet is selected as the communication layer. GASNet provides different conduit implementations that are optimized for specific network interconnects. All experiments used the `ibv` GASNet conduit, which is optimized for Infiniband networks.

The applications and kernels evaluated for the aggregation optimization are Breadth First Search (BFS), Single Source Shortest Path (SSSP), k-core decomposition and histogram. The descriptions and implementations of these codes are presented in Chapter 3. The primary metrics reported for each experiment are application runtime and the runtime speed-up achieved by COPPER relative to the unoptimized baseline implementations. For each experiment, multiple trials were performed and the average of these trials is presented. The variation in runtime across all trials did not exceed 5%.

Table 5.1: Graphs used in aggregation performance evaluation.

Name	# Vertices	# Edges	Density (%)
scale-24	16M	536M	1.9e−4
scale-25	33M	1B	9.5e−5
scale-26	67M	2B	4.8e−5
scale-27	134M	4B	2.4e−5
scale-28	268M	8.5B	1.2e−5

5.2.2 Results

The following experiments involving graph analytics operate on undirected graphs generated according to the Graph500 benchmark specifications [Bad+22]. These graphs are synthetically generated but are characteristic of graphs whose degree distributions follow a power law. The size of the graphs are determined by two values: the *scale* and *edge factor*. The scale of a graph is the $\log_2$ number of vertices in the graph. The edge factor of the graph is defined as the ratio of the graph's edge count to its vertex count, which is half of the average degree of each vertex. The default edge factor used by the Graph500 is 16 (i.e., each vertex has an average degree of 32). For example, a scale 26 graph with an edge factor of 16 has 2^{26} vertices and at most $2^{26} \cdot 32$ edges. Table 5.1 presents the range of graph scales that are evaluated, which all use an edge factor of 16.

Breadth First Search

The Breadth First Search (BFS) graph analytic, as described in Section 3.1.1, can be optimized via COPPER's remote data aggregation, specifically the remote write performed on line 14 in Listing 3.1. Table 5.2 presents the BFS runtime speed-ups achieved by COPPER when compared to the unoptimized baseline implementation on the scale 26, 27 and 28 graphs from Table 5.1. Additionally, Tables 5.3 and 5.4 present the execution runtimes (in minutes) for the unoptimized and optimized codes, respectively.

Table 5.2: Runtime speed-ups achieved by COPPER's remote data aggregation optimization on BFS for the graphs in Table 5.1 relative to the unoptimized implementation.

Locales	scale-26	scale-27	scale-28
2	4.1	2.8	4
4	4.3	2.6	4.1
8	4.2	2.5	4.3
16	5.5	3.7	4.9
32	7.6	7.2	6.3
geomean	5	3.4	4.7

Table 5.3: Unoptimized BFS baseline execution runtime in minutes.

Locales	scale-26	scale-27	scale-28
2	73	147	294
4	51	103	202
8	30	63	120
16	19	35	69
32	14	26	46

Table 5.4: Aggregation-optimized BFS execution runtime in minutes.

Locales	scale-26	scale-27	scale-28
2	18	37	74
4	12	24	49
8	7	14	28
16	4	7	14
32	1.8	3.5	7

As can be seen from Table 5.2, COPPER's aggregation optimization provides runtime speed-ups as large as 7.6x when compared to the unoptimized code. When considering the raw execution times from Tables 5.3 and 5.4, we see that these speed-ups are not merely saving a handful of seconds but multiple hours. For example, the scale-28 graph requires almost 5 hours to complete on two locales with the unoptimized code. However, COPPER reduces this runtime to just over one hour. Furthermore, the optimized code achieves better scalability than the unoptimized code. This is because as the number of locales increase, the number of fine-grain remote writes increases, leading to performance issues for the unoptimized code. However, COPPER can handle the increase in communication by performing more aggregation, providing better overall scalability. For example, the optimized code achieves a speed-up of 10.6x from two to 32 locales on the scale-28 graph while the unoptimized code only achieves a speed-up of 6.4x.

Despite the significant speed-ups that COPPER provides for BFS, the overall run-

time performance of the code still falls short of hand-tuned C/C++ code that use OpenMP and MPI. This is because the unoptimized Chapel implementation for BFS is designed to be as simple as possible and leverage the productivity features of Chapel and PGAS. Specifically, an OpenMP+MPI implementation of BFS, as presented in prior work [Rol+21], that performs manual aggregation is roughly 28x faster than the optimized Chapel code. For example, performing BFS on the scale-28 graph on 16 locales requires 69 minutes for the unoptimized Chapel code, 14 minutes for the COPPER-optimized Chapel code and 30 seconds for the OpenMP+MPI code. While the OpenMP+MPI and COPPER-optimized Chapel codes use aggregation, the Chapel code suffers greatly from the use of high productivity data structures, which is the root cause of the large performance difference. Specifically, the associative domains that are used to represent the queues incur large overheads because of their built-in support for concurrent updates, and the fact that operations on associative domains are much more expensive than those on normal domains/arrays. This is in contrast to the OpenMP+MPI code which uses thread-private C++ vectors to accumulate updates before performing global reductions.

When evaluating a version of BFS implemented in Chapel that matches the manual aggregation approach of the OpenMP+MPI code and uses normal arrays, it was observed that the performance difference is diminished. In fact, the Chapel code is 3.7x faster, on average, than the OpenMP+MPI code due to using a more efficient memory allocator for multi-threaded applications, namely jemalloc [Eva06]. When using jemalloc within the OpenMP+MPI code instead of the standard C++ memory allocator, we observe a 2x improvement in runtime. This results in the Chapel code being 1.3x faster, on average, than the OpenMP+MPI code. These results highlight the limitation of what COPPER's automatic aggregation optimization can achieve, as fine-grain communication may not be the only performance issue present in the code.

Table 5.5: Runtime speed-ups achieved by COPPER's remote data aggregation optimization on SSSP for the graphs in Table 5.1 relative to the unoptimized implementation.

Locales	scale-24	scale-25	scale-26
2	1.4	1.4	1.4
4	1.5	1.7	1.5
8	1.4	1.4	1.5
16	1.3	1.4	1.4
32	1.1	1.2	1.2
geomean	1.3	1.4	1.4

Table 5.6: Unoptimized SSSP baseline execution runtime in minutes.

Locales	scale-24	scale-25	scale-26
2	28	64	149
4	13	29	65
8	8	16	37
16	4	9	20
32	2	4.3	10

Table 5.7: Aggregation-optimized SSSP execution runtime in minutes.

Locales	scale-24	scale-25	scale-26
2	20	44	107
4	9	18	42
8	5	11	25
16	3	6	14
32	1.9	3.7	8

Single Source Shortest Path

Section 3.1.2 presented the Single Source Shortest Path (SSSP) graph analytic and Listing 3.2 described the baseline Chapel implementation of the main kernel of the SSSP algorithm. The remote write on line 15 can be aggregated via COPPER and creates a similar memory access pattern to BFS. Table 5.5 presents the SSSP runtime speed-ups achieved by COPPER when compared to the unoptimized baseline implementation on the scale 24, 25 and 26 graphs from Table 5.1. Additionally, Tables 5.6 and 5.7 present the execution runtimes (in minutes) for the unoptimized and optimized codes, respectively.

COPPER's aggregation optimization provides as much as a 1.7x speed-up over the unoptimized code. These speed-ups are noticeably smaller than those for BFS shown in Table 5.2. This is because SSSP is a more complicated analytic and exhibits other irregular memory access patterns that are not optimized via aggregation. Specifically, line 10 in Listing 3.2 can lead to fine-grain remote communication but it is not in

the form of a write, so aggregation cannot be applied. However, COPPER's adaptive remote data prefetching optimization can be applied (Chapter 6).

Similar to BFS, the optimized SSSP performance falls short of manually optimized C code that uses MPI for communication. The Graph500 benchmark [Bad+22] provides an MPI implementation of SSSP that uses a custom active message library to optimize the fine-grain communication. Rather than only aggregating the remote write on line 15 in Listing 3.2, the Graph500 code essentially aggregates the entirety of lines 8–18 by encapsulating the computation into an active message. However, this requires the programmer to write the active message handler that can process these messages, which means that programmer must be aware of when, where and how aggregation can be applied. Due to these differences, the MPI code is roughly 100x faster than the optimized Chapel code. Part of this performance disparity is also due to the Chapel code using associative domains for the queues, as described above for the BFS results.

K-core Decomposition

Section 3.1.3 presented the k-core decomposition graph analytic and Listing 3.3 provided the unoptimized Chapel baseline implementation of the main k-core decomposition kernel. Line 9 in Listing 3.3 is an irregular memory access pattern that can induce fine-grain remote communication and is a candidate for COPPER's aggregation optimization.

Aggregation via COPPER provides speed-ups as large as 5.4x over the unoptimized baseline, which can be seen in Table 5.8. The kernel from Listing 3.3, which contains the aggregated statement, represents the only portion of the k-core decomposition algorithm that issues remote communication. As a result, optimizing this kernel via aggregation provides significant performance gains. It can be seen in Tables 5.9 and 5.10 that the optimized code exhibits better runtime scalability as the number

Table 5.8: Runtime speed-ups achieved by COPPER's remote data aggregation optimization on k-core decomposition for the graphs in Table 5.1 relative to the unoptimized implementation.

Locales	scale-26	scale-27	scale-28
2	2.2	2	1.8
4	3	2.9	2.7
8	3.8	3.6	3.3
16	4.8	4	3.9
32	5.4	5.1	5.3
geomean	3.7	3.4	3.2

Table 5.9: Unoptimized k-core decomposition baseline execution runtime in minutes.

Locales	scale-26	scale-27	scale-28
2	83	166	348
4	58	118	237
8	37	72	143
16	27	47	92
32	18	34	73

Table 5.10: Aggregation-optimized k-core decomposition execution runtime in minutes.

Locales	scale-26	scale-27	scale-28
2	38	81	193
4	20	41	88
8	10	20	43
16	5.5	12	24
32	3.4	6.8	14

of locales increases. For example, the optimized code achieves a 14x speed-up from two to 32 locales on the scale-28 graph while the baseline code only achieves a 4.8x speed-up from two to 32 locales. Because of this, the optimized runtime speed-ups relative to the baseline in Table 5.10 improve as the number of locales increase.

Histogram

The histogram mini-kernel is described in Section 3.3.1 and the baseline Chapel implementation is presented in Listing 3.8. Histogram consists of a single `forall` loop with a single statement that can be aggregated via COPPER. Table 5.11 provides the runtime speed-ups achieved by COPPER relative to the unoptimized implementation. The "data sets" used in this experiment refer to the number of updates that each locale issues to a constant-size distributed table/array, which has 2^{24} elements distributed across the locales. Therefore, as the number of locales increase for a given number of updates, more fine-grain remote communication will be issued to the distributed

Table 5.11: Runtime speed-ups achieved by COPPER's remote data aggregation optimization on histogram relative to the unoptimized implementation. The distributed table/array has 2^{24} elements. Each column refers to the number of updates that each locale issues to the table.

Locales	2^{31}	2^{32}	2^{33}
2	83	79	87
4	57	56	57
8	58	59	59
16	60	60	67
32	61	58	56
geomean	63	63	64

Table 5.12: Unoptimized histogram baseline execution runtime in minutes. Each column refers to the number of updates that each locale issues to the table

Locales	2^{31}	2^{32}	2^{33}
2	97	194	393
4	47	94	187
8	51	103	207
16	57	114	253
32	60	114	224

Table 5.13: Aggregation-optimized histogram execution runtime in minutes. Each column refers to the number of updates that each locale issues to the table

Locales	2^{31}	2^{32}	2^{33}
2	1.2	2.5	4.5
4	0.8	1.7	3.3
8	0.9	1.8	3.5
16	0.9	1.9	3.8
32	1	2	4

table. Tables 5.12 and 5.13 present the runtimes in minutes for the unoptimized and optimized codes, respectively.

COPPER provides as much as an 87x speed-up over the unoptimized baseline, which is equivalent to a reduction of 6.5 hours of execution time. Indeed, Table 5.12 shows that a majority of the execution times for the unoptimized histogram code are an hour or more while Table 5.13 shows that the aggregation optimization keeps runtimes below five minutes. As histogram is a mini-kernel that performs no other operation besides fine-grain remote writes, the speed-ups observed due to aggregation are significantly higher than for the graph analytics algorithms presented earlier. However, despite its simplicity, the histogram mini-kernel is still representative of real-world communication patterns in parallel programs [MD19].

5.2.3 Summary of Results

The performance evaluation of COPPER's remote data aggregation optimization demonstrates that significant speed-ups can be achieved across different graph analytics and mini-kernels: 7.6x for BFS, 1.7x for SSSP, 5.4x for k-core decomposition and 87x for histogram. Aggregation improves performance by reducing the number of fine-grain messages that are sent over the network. It is important to note that these speed-ups were obtained without modifying the baseline implementations. Instead, COPPER applies the aggregation optimization automatically, removing the burden on the programmer to detect when, where and how aggregation can be used. Therefore, these results show that COPPER can successfully improve the performance of PGAS programs that exhibit irregular memory access patterns and provide good developer productivity.

While the runtime speed-ups achieved by COPPER are significant, the absolute execution times of the optimized codes are generally not competitive with hand-tuned C/C++ codes that use OpenMP and MPI. However, these codes require application-specific tuning such as active message handlers, which arguably reduces developer productivity. COPPER's goal is to strive for high performance without requiring user intervention, which can limit the complexity of the optimizations that are applied, and by extension, the degree of performance improvements. The performance gains that COPPER achieves in this evaluation highlight the trade off that programmers must consider in terms of performance and the amount of effort required to achieve that performance.

5.3 Limitations

In this section I will discuss the limitations of COPPER's remote data aggregation optimization and possible approaches to address these limitations.

5.3.1 Additional Operations

To begin with, the most obvious limitation of the optimization is support for additional operations that can be aggregated. As described in Section 5.1.1, COPPER maintains a list of operations that are known to be order-independent and can be aggregated. This limits what can be aggregated, both in terms of the size and complexity of the operation/computation as it can only optimize a single statement in the program. An approach like active messages [Eic+92; Pau+21; Pri+19; MD19] allows users to encapsulate arbitrarily complex computations within the message to be aggregated. However, this requires deep understanding of the specific application being developed, as well as knowledge of how aggregation can be applied. As a result, while COPPER can be limited in the scope of what it will aggregate, it does provides fully automatic aggregation with no user intervention.

One approach to strike a balance between performance and productivity for aggregation would be to allow for any given operation to be aggregated if it is known to be order-independent or has been marked as such by a user. This would allow for custom-defined or overloaded operations to be aggregated automatically, provided that a developer has indicated that such operations are order-independent. This order-independent property could be specified via a pragma, as shown in Listing 5.5. Supporting this pragma would not require any significant changes to COPPER's static analysis or code transformations, as it is straightforward to check pragmas within the compiler.

```
1 pragma "order-independent"
2 inline operator /= (arg1, arg2)
3 {
4   // custom operation for /=
5 }
```

Listing 5.5: Example of how a `pragma` could be used to specify that an operation is order-independent.

5.3.2 Memory Usage

Another limitation to COPPER's aggregation, as well as the existing aggregation that Chapel's compiler provides, is memory usage. Each task that performs aggregation for an operation in a `forall` loop allocates its own aggregation buffers. These buffers are of a fixed size and hold the addresses/values involved in the aggregation. This can lead to fairly significant memory usage increases, especially when a program is running on a system with many locales (nodes) and tasks (cores per node). In the experiments performed in Section 5.2, memory usage with aggregation never approached the point of exceeding the available memory per node, even for large graphs. However, the system used in the evaluation has a large amount of memory per node (512 GB).

One solution to reduce memory usage, which would be important for systems with less available memory, would be to leverage *multi-level* aggregation. Rather than having each task store buffers for each destination locale, all tasks on the same locale share a single set of buffers, which are then flushed to the destination locales. The downside of this approach is that it requires synchronization amongst the tasks on the same locale to update the shared buffers. The Chapel team has expressed interest in providing multi-level aggregation, and it would not be expected to require significant changes to COPPER to support once implemented.

5.4 Related Work

The most closely related prior work to COPPER's approach to aggregation is the existing support for aggregation in Chapel's compiler as described by Kayraklioglu et al. [Kay+21]. There are significant differences between COPPER's aggregation optimization and the aggregation optimization that Chapel's compiler provides. To begin with, Chapel's existing optimization can only apply aggregation to assignment (`=`) operations, and only when the assignment is the last statement in the `forall` loop.

By enforcing that the statement is at the end of the `forall`, data dependency analysis is not required to detect data hazards. By extension, the existing optimization cannot optimize statements that are in `foreach` loops, even if the statement is at the end of the `foreach` loop and that loop is at the end of the `forall` loop. In contrast, COPPER supports aggregation on multiple types of operators and provides extensive data dependency analysis to allow for aggregation to be applied to statements that are not at the end of the `forall` loop body.

However, the existing aggregation optimization can perform aggregation on both read and write operations, whereas COPPER specifically focuses on write operations. COPPER has other optimizations that specifically target remote reads (Chapter 6 and 7), which are not applicable to remote writes. Furthermore, the existing aggregation optimization enforces more strict requirements on the structure of the candidate memory access. Specifically, one side of the operation (in this case, the only operation supported is `=`) must be local with respect to the executing task and the other side is not local. This locality property must hold for every iteration of the `forall` loop. This is a reasonable requirement, as aggregation should only be applied when local data can be buffered and then "flushed" to the remote destination. However, because of this requirement the cases where aggregation can be performed can be very limited when considering the irregular memory access patterns that are the focus of COPPER. Since the underlying mechanisms that perform aggregation in COPPER will work whether the data to aggregate is local or remote, COPPER does not attempt to infer the locality of the operations, which often cannot be determined due to the irregular nature of the memory access pattern.

Beyond the work of Kayraklioglu et al. [Kay+21], aggregation has been performed within the scope of active messages. Examples include Conveyors [MD19], You Got Mail (YGM) [Pri+19] and the actor-based programming system [Pau+21]. COPPER's aggregation optimization differs from these because it is performed as part of

a compiler pass that automatically identifies candidates and applies aggregation. In contrast, the active message approaches rely on programmer guidance/intervention to both identify what can be aggregated and to apply/implement the aggregation mechanism.

Chapter 6

Adaptive Remote Prefetching

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 the 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 (i.e., 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 [Smi82; DS96; Sri+07; Tal+21] and software [AJ17; CKP91; Bad+04]. Hardware prefetchers rely on predicting memory access patterns as they appear across the memory bus, irrespective of the application that is issuing the accesses. In this way, hardware prefetchers work well when the access patterns are sequential or strided, as those can be predicted with high accuracy (i.e., the data that is prefetched is likely to be accessed by the application). However, when an access pattern is truly irregular and does not exhibit any predictable spatial or temporal locality, then hardware prefetchers will not perform well. This is because

of the loss in accuracy in predicting the data that will be needed, which leads to wasted data movement and cache pollution.

On the other hand, software prefetching is well-suited for irregular memory access patterns because it removes the "guess work" from prefetching by issuing the prefetches for a specific access pattern in the application. In the context of distributed-memory systems, prefetches are issued to remote data to hide remote communication latency, which is expensive. For example, consider a loop that issues an access of the form $A[B[i]]$, where i is the loop index variable and accesses to A may be remote. A simple approach to software prefetching would be to insert a call to do a non-blocking prefetch for $A[B[i + d]]$ before or after the original $A[B[i]]$ access in the loop. The value d is the number of loop iterations to "look ahead" and is referred to as the *prefetch distance*. Ideally, by the time iteration $i + d$ is executed, the prefetch will have completed and the data will be available locally, thus hiding the communication latency.

A challenge with software prefetching is how far ahead to issue the prefetches, which equates to the value of the prefetch distance. If the prefetch distance is too small, the prefetches will not have completed by the time the data is needed, and if the distance is too large, then the data that was prefetched may be evicted from the cache before it can be used. The former is referred to as prefetches that are *late* and the latter as prefetches that are *early*. Determining a good prefetch distance is dependent on multiple factors, such as memory access latency across the system and the structure/workload of a loop iteration. These parameters can vary across multiple hardware platforms, applications and the input data that the application runs on.

In this chapter, I describe an adaptive remote prefetching optimization that is integrated into the COPPER framework (Chapter 4). The optimization uses static analysis to identify candidate memory accesses to prefetch and then performs code transformations to perform the prefetches. As COPPER targets programs written in

the Chapel programming language (Section 2.4), the prefetches are issued with respect to Chapel's runtime managed software cache that works specifically for remote data. As a result, large performance gains should be possible because the cost of remote communication is significant. This is in contrast to traditional prefetching for on-node CPU caches, where the memory access latencies are orders of magnitude smaller when compared to remote communication. The optimization is referred to as *adaptive* because the prefetch distance is adjusted as the program executes rather than relying on a predetermined fixed distance. Adaptive software prefetching approaches have been studied in the past [SP96; Hei+18], but COPPER's approach specifically targets remote memory accesses within the Partitioned Global Address Space (PGAS) model.

I also present a performance evaluation of the optimization across a suite of irregular applications that were described in Chapter 3. These results show that runtime speed-ups as large as 2.6x can be achieved via adaptive remote prefetching without requiring any user intervention. Much of the work presented in this chapter was also presented in a paper that is under submission at the time of this writing [RS].

6.1 Chapel's Remote Data Cache

Before describing the adaptive remote prefetching optimization, it is necessary to provide details on Chapel's remote data cache, since COPPER targets Chapel programs. 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 specifically for remote data. The initial design and implementation of the remote cache was introduced by Ferguson and Buettner [FB15]. 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 COPPER's 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 COPPER's adaptive prefetching optimization.

6.2 Approach

In this section, I describe the overall approach taken by COPPER to perform adaptive remote prefetching for remote reads. At a high level, COPPER employs a combination of static analysis and code transformations to identify valid candidates for prefetching and modify the original program to perform various checks necessary to do the prefetches. Prefetching candidates must be within `forall` loops, and prefetching is performed on a per-task basis (i.e., each task has its own prefetch distance). While virtually any access can be prefetched to some degree, COPPER focuses exclusively on array-based accesses where the loop index variable is used to index into the array. In this way, prefetching involves "looking ahead" some number of loop iterations and prefetching the array value that would be accessed in that iteration. Another key aspect of the optimization is adjusting the prefetch distance(s) as the loop executes. Adjusting the prefetch distances is performed by reading performance counters that

I added to Chapel's remote cache, which provide details about the timeliness of the prefetches. Using these counters, I designed heuristics that adjust the distances to provide better timed prefetches.

6.2.1 Static Analysis

The static analysis performed by COPPER for the adaptive remote prefetching optimization has 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. Most of the static analysis is performed during Chapel's normalize compiler pass (Section 4.2.1), but there are some key tasks performed during the resolve and cull-over references passes.

Identifying Candidate Memory Access Patterns

Static analysis begins during the normalize compiler pass by identifying potential candidates for prefetching, where these candidates must be contained inside `forall` loops. However, the candidate can be nested within a `for` loop that is itself nested in a `forall` loop. Note that the candidate must be directly within the loop's body (i.e., not within a procedure which is then called from the loop body). The optimization specifically targets irregular reads and looks for non-affine access patterns of the form $A[B[i]]$, where A and B are arrays and A is a distributed array. These access patterns are likely to result in fine-grain remote communication, and their access pattern cannot be inferred at compile time. More generally, the optimization operates on distributed arrays that are accessed by an expression that involves another array access.

The expressions used in the candidate array access can be more complex than

$A[B[i]]$, but not as general as what is supported for COPPER's aggregation optimization (Chapter 5). As prefetching attempts to look ahead some number of loop iterations, COPPER must be able to guarantee that the prefetches will be issued to valid memory locations. To this end, COPPER requires that the loop index variable for the loop that contains the candidate is 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. This check for read-only variables is performed during the cull-over references pass, which is when the read/write intents of variables can be determined.

For example, consider a candidate of the form $A[B[i] + foo]$, where i is the loop index variable. If foo is not modified as the loop executes, then it can be considered a constant and prefetching can be performed by offsetting i with the prefetch distance. However, if foo is updated within the loop (e.g., $foo \mathrel{+}= bar()$), then it cannot be easily determined how to issue prefetches that are guaranteed to not go beyond the bounds of A. There are simple cases, such as if foo is modified by a constant amount (e.g., $foo \mathrel{+}= 1$). However, foo could be modified by another variable, a procedure call, etc. In the end, it was determined that such access patterns were not common enough in the irregular applications studied for this dissertation to justify the effort to support them.

Last, the array B in the prefetch candidate $A[B[i]]$ cannot be modified within the loop. Consider the case where COPPER issues a prefetch for $A[B[i + d]]$, where d is the prefetch distance. Through various checks that will be described below,

COPPER ensures that the prefetch will be to a valid element of A (i.e., an element of A that will be accessed in some future iteration). However, if $B[i + d]$ happens to be modified in some way within the loop after the prefetch, then the prefetch that was issued may not reflect a true future access to A. This is because when issuing the prefetch, the value of $B[i + d]$ is computed/determined. To avoid this issue, COPPER will not apply the optimization if B is modified within the loop, which can be determined during the cull-over references pass in the compiler. This check will perform interprocedural and alias analysis to track modifications to B across procedure calls and pointers/references.

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, which can be done entirely during the normalize pass. 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, COPPER 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 6.2.2. As noted earlier regarding the supported memory access patterns for the candidates, the loop index variable must be used within the candidate access. Therefore, the form of the loop determines how the loop index variable progresses,

and hence how the array access progresses. Additionally, because COPPER allows for prefetch candidates to be within `for` loops that are nested within `forall` loops, the same static analysis is applied to the `for` loop.

Detecting Record Field Accesses

A particular use-case that appears often in the irregular applications described in Chapter 3 is performing a remote read like $A[B[i]]$ where A stores records (i.e., structs). In this case, different fields of the record could be accessed in subsequent statements. This can be seen in Listing 3.2 on lines 10–15, where `v` is the remotely accessed record and the fields `dist`, `visited` and `id` are accessed. Similar to languages like C, Chapel stores the fields of a record in a contiguous block of memory according to the order that they are defined in the record. However, arrays are not stored as simple pointers to the memory that contain their elements. Since Chapel arrays are high-level objects, there is various metadata associated with the array that is stored within the record's memory. As a result, fields of a record can be separated by a number of bytes that can be difficult to determine statically.

By the design of Chapel's remote cache, prefetches will be rounded up to fill 64-byte cache lines. However, the fields of a record that are ultimately accessed within the loop may be separated by more than 64 bytes. While the number of bytes to prefetch can be specified when issuing the prefetch, it is not practical to do in some cases where there are many fields of varying data types/sizes between the desired fields. In that case, it would be wasteful to round up the prefetch to encapsulate all of the fields that are needed.

A better approach is to issue separate prefetches for each field that is accessed in the loop. The code transformation to issue these prefetches is described in Section 6.2.2. Static analysis is used to detect whether A stores records, and which fields are accessed in the loop. This analysis is performed during the resolve compiler pass,

which is when data types have been determined. 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.

6.2.2 Code Transformations

The code transformations performed by COPPER for the adaptive remote prefetching optimization have the following steps: (1) modifying the loop iterator, (2) creating prefetch-related variables, (3) creating compile time checks, (4) creating the prefetch distance adjustment check and (5) creating the prefetch procedure call. A significant number of these transformations are performed during Chapel's normalize compiler pass, where modifying the structure of the program is less challenging when compared to later compiler passes. Listings 6.1 and 6.2 present an example `forall` loop that contains a valid prefetch candidate (line 3 in both listings), followed by code that is equivalent to the output of COPPER's code transformations. Each listing illustrates how the transformations are performed under different scenarios.

```
// Original forall loop
forall i in B.domain {
  C[i] = A[B[i]];
}

// Code equivalent to the output of the code transformations
reset_prefetch_counters();
const lastIdx = getLastLoopIndex(B.domain);
const firstIdx = getFirstLoopIndex(B.domain);
const stride = getLoopStride(B.domain);
const sampleIters = getSampleSize(firstIdx, lastIdx, stride);
forall i in B.domain with (var d = 8, var cnt = 0) {
  if isPrefetchSupported(B.domain, A, B) { // compile time check
    if cnt < sampleIters {
      cnt += 1;
    }
    else {
      adjustPrefetchDistance(d);
      cnt = 0;
    }
    if (i + (d * stride)) <= lastIndex {
      prefetch(A[B[i+(d*stride)]]);
    }
  }
  C[i] = A[B[i]];
}
```

Listing 6.1: Example of a prefetch candidate and the resulting code transformations applied by COPPER when the loop's iterand is known to be a domain. `C[i] = A[B[i]]` is the prefetch candidate.

Modifying Loop Iterator

The first code transformation that is performed is modifying the loop iterator for the loop that contains the candidate access to ensure that it yields the proper loop index variable for prefetching. Prefetching will look ahead some number of loop iterations by adding the prefetch distance to the loop index variable, where this adjusted value will refer to the value of the loop index variable in a future iteration. When the loop's iterand is known to be a domain during the normalize pass, as in Listing 6.1, then this transformation is not necessary. In this case, the loop index variable `i` can be adjusted by the prefetch distance to yield a future value. When the iterand is an explicit range, such as `0..10`, the loop index variable can also be adjusted in the same way.

```
// Original forall loop
forall e in Foo {
  C[i] = A[B[e]];
}

// Code equivalent to the output of the code transformations
reset_prefetch_counters();
const lastIdx = getLastLoopIndex(Foo);
const firstIdx = getFirstLoopIndex(Foo);
const stride = getLoopStride(Foo);
const sampleIters = getSampleSize(firstIdx, lastIdx, stride);
forall (e,idx) in prefetchIter(Foo) with (var d = 8, var cnt = 0) {
  if isPrefetchSupported(Foo, A, B) { // compile time check
    if cnt < sampleIters {
      cnt += 1;
    }
    else {
      adjustPrefetchDistance(d);
      cnt = 0;
    }
    if (idx + (d * stride)) <= lastIndex {
      if isArray(Foo) { // compile time check
        prefetch(A[B[Foo[idx+(d*stride)]]]);
      }
      else {
        prefetch(A[B[idx+(d*stride)]]);
      }
    }
  }
  C[i] = A[B[e]];
}
```

Listing 6.2: Example of a prefetch candidate and the resulting code transformations applied by COPPER when the loop's iterand could be an array. `C[i] = A[B[e]]` is the prefetch candidate.

However, consider the case in Listing 6.2 where the loop iterand `Foo` could be an array or domain, as such information is not known until types are resolved during the resolve pass. If `Foo` is an array, then adding the prefetch distance to the loop index variable `e` is incorrect, as that would not yield a future value that `e` would hold. This is because `e` is an element of an array, not an index from its domain. Instead, what is needed is not only `e` but its corresponding index in `Foo`. Using this index, the optimization can generate the proper prefetch address, which will be described later. To generate this additional loop index variable, COPPER modifies the loop's iterator to use a custom iterator, `prefetchIter` (line 12 in Listing 6.2). This custom iterator

yields a tuple that consists of the array's value `e` and its corresponding index `idx`.

If the loop's iterand turns out to be a domain and not an array, then the custom iterator `prefetchIter` is not necessary, since the domain will already yield a proper loop index variable for prefetching. However, as discussed above, determining whether the loop iterand is an array or domain cannot be performed during the normalize pass. To account for this issue, the `prefetchIter` will yield a tuple that consists of the original loop index variable repeated twice when it is provided with a domain instead of an array. In other words, `e` and `idx` on line 12 in Listing 6.2 would be the same value, and COPPER will then use `idx` within the rest of the code transformations. This allows for the same code transformation to be used in either case of the iterand being an array or a domain.

Create Prefetch Variables

The next step in 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 Listings 6.1 and 6.2. 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 COPPER. 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.

Create Compile Time Checks

Line 13 in Listings 6.1 and 6.2 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 with other compile time checks described in Chapters 4 and 5, this check will resolve to true or false after the normalize pass. The end result is that if prefetching cannot be supported, then all of the code transformations that appear within the then-branch of the if statement will be removed.

There is an additional compile time check on line 22 in Listing 6.2, which is specific to the case where the loop iterand could be an array or domain. The compile time check determines whether the loop iterand is an array, and if so, it will issue a modified version of the prefetch (line 23). More details on how this is performed will be provided later.

Create Prefetch Distance Adjustment Check

Lines 14–20 in Listings 6.1 and 6.2 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 in the case of COPPER, the distance will be adjusted as the program executes. Therefore, the initial value of 8 is chosen as a reasonable starting point (i.e., not too small or too large). An alternative approach to selecting the initial prefetch distance would be to analyze the instructions/operations in a loop iteration. For example, if there are several reads/loads to potentially distributed data then a smaller initial distance could be used. The presence of these loads could mean that each iteration would be expensive, so the prefetches can be issued "closer" together without running the risk of being too early. As will be shown in Section 6.3.2, using an initial value of 8 leads

to good performance, so the analysis described above is left as future work.

For each loop iteration that a task executes, the task increments `cnt` (line 15) until it reaches the value of `sampleIters`, at which point the optimization will adjust the prefetch distance (line 18). The actual calculation of `sampleIters` will be described in Section 6.2.3, 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. If the loop's iterand is a domain `dom`, then the last valid loop index will be `dom.high`, which is a built-in field within Chapel's implementation of domains. If the loop's iterand is an array `arr`, then the last valid loop index (as yielded by `prefetchIter`) will be `arr.domain.high`. Finally, if the loop iterand is an explicit range `start..end`, then the last valid loop index will be `end`. The procedure `getLastLoopIndex()` (line 8) extracts this information from the loop's iterand. The first value of the loop index variable can be found in a similar way via `getFirstLoopIndex()` (line 9). The loop stride refers to the amount that the loop index variable progresses between each iteration. The stride of an array or domain can be accessed via the `.stride` field within Chapel's implementation of domains, and the default stride of an explicit range is 1. However, Chapel allows for loops to have explicit strides via the `by` clause. For instance, `forall i in B.domain by 2` will yield indices from `B.domain` that are two apart between subsequent iterations. In either case, COPPER can correctly extract the loop's stride via the `getLoopStride()` procedure (line 10). With these values, the number of iterations is equal to `((lastIdx - firstIdx)+1)/stride`.

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 faster approach is to simply check whether the desired index is greater than 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 6.1 and 6.2 perform this bounds check. Specifically, the expression `d * stride` computes how many loop iterations ahead to prefetch with respect to the current iteration. By considering the loop stride, the optimization can account for loops that do not necessarily increment the loop index variable by one. Then, by adding this value to the current loop index variable (`i` in Listing 6.1 and `idx` in 6.2), the optimization can check whether it is less than or equal to the last value that the loop index can take.

After the bounds check, the prefetch call itself can be constructed. Line 22 in Listing 6.1 shows the call to `prefetch()`, which passes in the original `A[B[i]]` access but with `i` replaced with `i+(d*stride)`, so effectively looks ahead `d` iterations with respect to the loop. The procedure `prefetch()` takes in a single argument, which is the address to prefetch. Within the procedure, a lower-level call is made to Chapel's runtime to perform the remote cache prefetch, which requires additional information such as the locale that hosts the data to be prefetched. However, in the case of Listing 6.2 where the loop iterand could be an array, additional code transformations are necessary to perform the prefetch. As noted before, if the loop iterand `Foo` is an array, then the loop index variable `e` is an element of the array, which cannot be used directly in the prefetch address calculation. Instead, a future value of `e` must be computed by accessing `Foo` by `idx+(d*stride)`, where `idx` is the corresponding index of `e` within `Foo` yielded by the custom iterator `prefetchIter`. The compile time check on line 22 in Listing 6.2 checks whether `Foo` is an array, and if so, the call

to `prefetch()` on line 23 will execute, which uses `Foo[idx+(d*stride)]` to compute a future value of `e`. Otherwise, the call to `prefetch()` on line 26 will execute, which directly uses `idx` as the loop index variable since `Foo` is a domain.

Finally, if the static analysis determines during the resolve pass that the target array `A` stores records, then COPPER will transform the code to issue prefetches specifically to the record fields that are accessed throughout the loop. Chapel provides the `Reflection` module[1] that allows programmers to perform code introspection. Of particular relevance to COPPER is the ability to query information about records, such as returning the value of a field as specified by its name (as a string) or index/position in the record. COPPER provides an alternative version of the `prefetch()` procedure that takes in an additional argument, which is the index/position of a record field to prefetch for. The static analysis pass builds an internal list of the record fields that are accessed, including their position in the record. In this way, it is straightforward for COPPER to adjust the `prefetch()` calls created during the normalize pass to instead issue prefetches to the specific record fields.

It is worth noting that the index into the target array (e.g., `B[i]` in Listing 6.1) is not explicitly prefetched by COPPER. While this is often performed in other software prefetching techniques [Bad+04; CKP91; AJ17], Chapel's remote cache will automatically read ahead for sequential access patterns. As the traversal of the index array is often sequential or has a known stride, it is not necessary to explicitly prefetch the index, and doing so would most likely lead to an increase in network traffic and degrade performance.

6.2.3 Prefetch Distance Adjustment

Sections 6.2.1 and 6.2.2 described how COPPER performs static analysis and code transformations to perform the adaptive remote prefetching optimization. Beyond

[1] https://chapel-lang.org/docs/modules/standard/Reflection.html

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.

In this section, I will describe the details of how often the prefetch distances are adjusted and how the optimization leverages performance counters to make informed decisions about adjusting the prefetch distances.

Prefetch Distance Adjustment Frequency

Section 6.2.2 noted the `sampleIters` variable that COPPER's code transformation creates (line 11 in Listings 6.1 and 6.2), which specifies how many iterations of the loop must elapse before a given task's prefetch distance is adjusted. It is based on the total number of iterations of the loop and a configurable constant referred to as the *sample ratio*. As prefetching is performed per task, the frequency of the prefetch distance adjustment should be per task. There are two scenarios to consider for the calculation of `sampleIters`: (1) the prefetch candidate is directly nested in a `forall` loop (as in Listings 6.1 and 6.2), and (2) the prefetch candidate is nested in a `for` loop that is nested within a `forall` loop. Each of these cases result in a different

approach to computing how many iterations of the loop each task will execute.

For scenario (1) above, the iterations of the `forall` loop are divided amongst the tasks executing the loop. If the `forall`'s iterand is a distributed array or domain, then tasks will be created across all of the locales. Assuming that there are N tasks across the locales and I iterations of the loop, then each task will get roughly I/N iterations of the `forall` loop. If the `forall`'s iterand is not a distributed array or domain, then the iterations will only be divided amongst the tasks on a single locale. In either case, COPPER can determine the number of iterations that each task will be given and use that number when computing the value of `sampleIters`. For scenario (2) above, each task executing the `forall` will perform all iterations of the `for` loop that contains the prefetch candidate. COPPER treats the `for` loop the same as a `forall` loop (i.e., the loop bounds and stride are computed with respect to the `for` loop). Since the iterations of the `for` loop are not divided up amongst tasks, the total number of iterations of the `for` loop is used to compute the value of `sampleIters`.

With the proper number of iterations per task computed, the final step in computing `sampleIters` is factoring in the sample ratio. This is a configurable constant that is between 0 and 1, so that when multiplied together with the number of iterations per task, it will yield some fraction of the number of iterations. To determine a "good" value to use for the sample ratio, I performed various experiments across different systems, applications and data sets to see which value yielded the best performance. Based on these experiments, a value of 0.01 will be used in the experiments presented in Section 6.3. A sensitivity analysis for this value will be provided in Sections 6.3.2 and 9.2. For example, if the number of iterations per task is determined to be 10,000 then the value of `sampleIters` will be 100, indicating that the prefetch distance should be adjusted after every 100 iterations. If the computed value for `sampleIters` is less than 1, then COPPER will not attempt to adjust the distance.

The value of 0.01 for the sample ratio favors loops with a large number of iterations,

which provide more feedback to the optimization at runtime to make adjustments to the prefetch distance. However, there could be situations where a loop has a significant number of iterations and each iteration is expensive. In this case, the number of iterations that must elapse before adjusting the distance would be high and the adjustments may not happen frequently enough. Therefore, it may be necessary to enforce a minimum value for `sampleIters`, though the experiments performed in Section 6.3 did not call for such a requirement.

Remote Cache Performance Counters

A crucial aspect of COPPER's approach to adaptive remote prefetching is gathering information to make informed decisions about how to adjust the prefetch distances. As discussed at the beginning of this chapter, prefetches can be issued too early or too late, where early prefetches run the risk of being evicted from cache before being used and late prefetches do not complete by the time a task needs to access the data. With knowledge of how many prefetches are late or early, heuristics can be developed to make adjustments to the distance to produce better timed prefetches. To achieve these goals, I added performance counters to Chapel's remote cache that keep track of the number of prefetches issued by a task, how many of those prefetches were evicted before being accessed and how many of those prefetches had to be "waited on" before being accessed.

There are existing counters for Chapel's remote cache that can provide some level of insight into the timeliness of prefetches and are provided within Chapel's `CommDiagnostics` module[2]. These counters are per locale, and as a result, must be atomic to account for modifications by multiple concurrent tasks. The overhead of these counters is relatively low for applications that do not issue prefetches to the remote cache. However, the applications optimized via COPPER will issue potentially

[2]https://chapel-lang.org/docs/modules/standard/CommDiagnostics.html

millions of prefetches. As a result, the atomic nature of the existing counters introduce so much overhead that any benefits from prefetching are negated. It is worth noting that I contributed these atomic counters to Chapel's runtime[3] in an attempt to support the adaptive remote prefetching optimization, and their atomic nature was driven by the need to integrate them with Chapel's existing `CommDiagnostics` remote cache counters, which are also atomic and per locale.

The two issues with the existing remote cache counters described above are that they do not provide metrics that are isolated to a given remote cache (i.e., task) and they introduce too much overhead to be used in a runtime/adaptive optimization. To make the remote cache counters more practical for COPPER, I implemented additional counters that are local/private to a given remote cache. As a result, they provide the granularity of detail necessary for COPPER to adjust each task's prefetch distance and they do not require atomic operations, and hence, introduce a negligible amount of overhead in real application executions. As these counters are designed to be used by COPPER as part of its automatic optimization framework, they are not user-facing via the `CommDiagnostics` module. Instead, I added low-level procedures that COPPER can use to reset the counters (e.g., line 7 in Listings 6.1 and 6.2), query individual counters, etc.

Adjustment Heuristics

With efficient performance counters available to COPPER to characterize the timeliness of the prefetches, the final step to the adaptive remote prefetching optimization is to adjust a given prefetch distance to improve performance. There are two main approaches that were evaluated: (1) attempt to minimize early prefetches and (2) attempt to minimize late prefetches. In both of these approaches, the goal is to improve the timeliness of prefetches to hide remote communication latency, which

[3]https://github.com/chapel-lang/chapel/pull/19969

serves as a proxy for application runtime. The overall idea behind the heuristics is based on the work of Heirman et al. [Hei+18]. However, our approach differs from their because we target distributed memory systems and the PGAS model, and we implement the adjustment heuristic as part of a compiler optimization. On the other hand, Heirman et al. provide an accompanying hardware design with their approach and target many-core architectures. While they also describe how software prefetching could be utilized in their design, it is demonstrated with manually inserted prefetch instructions rather than via a compiler optimization.

For approach (1), the optimization reads the remote cache counters to compute the percentage of prefetches that were issued but evicted before being used. If this percentage is greater than some threshold, then the prefetch distance is decreased by one. The desired outcome of decreasing the distance is to have the prefetched data remain in cache long enough to be accessed. Otherwise, if the percentage is less than the threshold, and has remained so for some number of adjustment intervals (i.e., calls to `adjustPrefetchDistance()` on line 18 in Listings 6.1 and 6.2), the prefetch distance is increased by one. This has the effect of ensuring that the prefetch distance does not grow too small, leading to unnecessarily late prefetches. For approach (2), the same logic is used but it considers the percentage of prefetches that were marked as late. In that case, the prefetch distance is increased by one, allowing more time for the prefetches to complete before being accessed. If the percentage of late prefetches has remained less than the threshold for some number of adjustment intervals, the prefetch distance is decreased by one to discourage unnecessarily early prefetches. For both approaches, the remote cache counters are reset after each adjustment.

Heirman et al. [Hei+18] postulate that adaptive heuristics should target late prefetches rather than early prefetches. Their reasoning is that late prefetches are still "useful" prefetches, in that they represent data that will be accessed at some reduced latency cost. Furthermore, small adjustments to the prefetch distance are

sufficient to lead to improved performance. In contrast, early prefetches are ones where the entire cost of the memory access latency is incurred (since the data has to be communicated again), and without extensive profiling, it is difficult to quantify “how early” the prefetches were. Because of this, it is difficult to determine how much to adjust the prefetch distance. I evaluated both heuristics and found that targeting late prefetches always performed significantly better than targeting early prefetches, thereby confirming the approach taken by Heirman et al. Specifics results are provided in Section 6.3.

For the heuristic that targets late prefetches, there are two configurable values that must be discussed: the threshold for the percentage of late prefetches and the number of adjustment intervals that elapse before decreasing the distance (assuming that the percentage of late prefetches is less than the threshold). From sensitivity studies across different applications and systems, I found that a late prefetch threshold of 10% and an adjustment interval of 8 provides good performance (see Sections 6.3.2 and 9.2).

6.3 Performance Evaluation

This section presents a performance evaluation of COPPER's adaptive remote prefetching optimization. The goal of the evaluation is to demonstrate that significant performance improvements can be achieved via COPPER on baseline implementations of the irregular applications described in Chapter 3. The memory access patterns that are optimized in this evaluation are strictly those that are not candidates for the aggregation optimization (Chapter 5), since the prefetching optimization targets reads while the aggregation optimization targets writes. These baseline codes fully leverage the productivity advantages of the PGAS model and Chapel, but suffer severe performance issues due to fine-grain communication. As the runtime improvements

are achieved without modifying the original code, these results demonstrate that performance gains can be obtained without sacrificing user productivity, enabling users to develop irregular applications without having to deal with the issues posed by fine-grain communication.

6.3.1 Experimental Setup

The experiments presented in this section were executed on the same 32-node FDR Infiniband cluster from Section 5.2 and use the same Chapel installation and configuration settings. The applications and kernels evaluated for the adaptive remote prefetching optimization are PageRank (PR), Single Source Shortest Path (SSSP) and Triangle Counting (TC). The descriptions and implementations of these codes are presented in Chapter 3. The primary metrics reported for each experiment are application runtime and the runtime speed-up achieved by COPPER relative to the unoptimized baseline implementations. For each experiment, multiple trials were performed and the average of these trials is presented. The variation in runtime across all trials did not exceed 5%. For the evaluations of PR, SSSP and TC, COPPER uses the heuristic that targets late prefetches when adjusting the prefetch distance and uses a late prefetch tolerance of 10%, an adjustment frequency of 8 and an iteration sample ratio of 0.01. However, I also present results from a sensitivity study that justifies these values. Last, I provide results from evaluating different prefetch adjustment heuristics and a comparison between COPPER's adaptive approach for the prefetch distance and an approach that keeps the prefetch distance fixed throughout the program's execution.

6.3.2 Results

All of the following experiments involve graph analytics and operate on undirected graphs generated according to the Graph500 benchmark specifications [Bad+22] or

Table 6.1: Graphs used in prefetching performance evaluation.

Name	**# Vertices**	**# Edges**	**Density (%)**
scale-18	262K	7.6M	0.01
scale-19	524K	15M	0.005
scale-24	16M	536M	1.9e−4
scale-25	33M	1B	9.5e−5
scale-26	67M	2B	4.8e−5
arabic-2005	23M	631M	1.2e−4
webbase-2001	118M	992M	7.1e−6
GAP-twitter	61M	1.5B	3.9e−5
sk-2005	50M	2B	7.5e−5
MOLIERE_2016	30M	6.6B	7.3e−4

Table 6.2: Runtime speed-ups achieved by COPPER's adaptive remote prefetching optimization on PR for the graphs in Table 6.1 relative to the unoptimized implementation.

Locales	scale-26	GAP-twitter	MOLIERE_2016
2	0.85	1.01	1.03
4	1.31	1.01	1.37
8	1.27	1.27	2.1
16	1.34	1.36	2.3
32	1.32	1.43	2.6
geomean	1.2	1.2	1.78

real-world graphs obtained from the SuiteSparse Matrix Collection [DH11]. Section 5.2.2 provides an explanation of the Graph500 graphs and their parameters. Table 6.1 presents the graphs evaluated, where the "scale-" graphs are the synthetic Graph500 graphs. The arabic-2005, sk-2005 and webbase-2001 graphs are from the Laboratory for Web Algorithmics (LAW) [BV04] and represent web crawl data. The GAP-twitter graph is from the GAP Benchmark Suite [BAP15] and represents a graph of the Twitter social network topology. Finally, MOLIERE_2016 represents documents/keywords that were used in a hypothesis generator for biomedical research [SSS17].

Table 6.3: Unoptimized PR baseline execution runtime in minutes.

Locales	scale-26	GAP-twitter	MOLIERE_2016
2	186	190	1733
4	94	78	1833
8	56	61	1850
16	31	50	1590
32	18	31	1488

PageRank

The PageRank (PR) graph analytic, as described in Section 3.1.1, can be optimized via COPPER's adaptive remote prefetching optimization, specifically the remote read performed on line 5 in Listing 3.4. Table 6.2 presents the PR runtime speed-ups achieved by COPPER when compared to the unoptimized baseline implementation on the scale-26, GAP-twitter and MOLIERE_2016 graphs from Table 5.1. Additionally, Tables 6.3 and 6.4 present the execution runtimes (in minutes) for the unoptimized and optimized codes, respectively. A convergence tolerance value of 1*e*-7 was used for these experiments, which is the same value used by Neo4j [Web12], an open-source graph database.

Table 6.2 shows that speed-ups as large as 2.6x are achieved via COPPER's adaptive remote prefetching optimization. Speed-ups generally improve as the number of locales increase, where using only two locales provides minimal performance gains, or in the case of the scale-26 graph, worse performance with respect to the baseline implementation. The prefetching optimization does incur some amount of overhead to perform the bounds checking, and when there are only two locales the amount of remote communication present in the PR algorithm can be rather low. Because of this, prefetching may not provide performance gains if there are not enough remote accesses to amortize the cost of the bounds checking.

Not present in Tables 6.2, 6.3 and 6.4 are results for the arabic-2005, webbase-2001 and sk-2005 graphs. All three of these graphs do not benefit from the prefetching

Table 6.4: Prefetch-optimized PR execution runtime in minutes.

Locales	scale-26	GAP-twitter	MOLIERE_2016
2	220	188	1682
4	72	77	1342
8	44	48	897
16	23	36	697
32	13	22	571

optimization, and in most cases exhibit worse performance when compared to the baseline. As alluded to above, this is because these graphs do not induce enough remote communication to make prefetching profitable. Specifically, less than 10% of the accesses made when executing these three graphs lead to remote communication. This is due to a combination of the graphs' structure and the block-distributed representation of the graph within the Chapel code, resulting in most of a given vertex's neighbors being co-located on the same locale as the vertex itself. On the other hand, more than 70% of the accesses issued for the GAP-twitter, MOLIERE_2016 and scale-26 graphs are remote, which provides ample opportunities for the optimization to amortize the cost of repeated bounds checking.

There is an approach that can be employed to determine the profitability of the prefetching optimization, which can avoid applying the optimization if there is not enough remote communication. Because PR is an iterative algorithm, where each iteration performs the same sequence of memory accesses to the graph, the *inspector-executor* technique can be applied [SMC91]. The *inspector* is a routine inserted before the PR kernel by the compiler that computes the number of remote accesses in the loop, which can be determined without actually performing the remote accesses. This keeps the inspector very lightweight and relatively inexpensive. For example, the overhead of the inspector does not exceed 1% of the total runtime of the PR kernel on the graphs in Table 6.1. The *executor* is the prefetch-optimized kernel that will run if the number of remote accesses is deemed sufficient, otherwise the original unoptimized loop is executed. In the case of these experiments, "sufficient" was determined to be

Table 6.5: Runtime speed-ups achieved by COPPER's adaptive remote prefetching optimization on SSSP for the graphs in Table 6.1 relative to the unoptimized implementation.

Locales	scale-24	scale-25	scale-26
2	0.78	0.78	0.77
4	1.2	1.32	1.26
8	1.4	1.38	1.45
16	1.35	1.45	1.41
32	1.23	1.31	1.34
geomean	1.17	1.22	1.22

Table 6.6: Unoptimized SSSP baseline execution runtime in minutes.

Locales	scale-24	scale-25	scale-26
2	28	64	149
4	13	29	65
8	8	16	37
16	4	9	20
32	2	4.3	10

Table 6.7: Prefetch-optimized SSSP execution runtime in minutes.

Locales	scale-24	scale-25	scale-26
2	36	82	193
4	11	22	51
8	5.4	11	26
16	2.9	6	14
32	1.6	3.3	7.6

60% or more.

The downside of this inspector-executor approach is that it is not suitable for all applications. Specifically, the kernel being optimized must be nested in an outer loop, which allows for the overhead of the inspector to be amortized over multiple executions of the optimized kernel. Additionally, if the memory access pattern for the prefetch candidate changes in some way, the inspector must be executed again, which leads to additional overhead. COPPER's selective data replication optimization is based on the inspector-executor technique and is described in detail in Chapter 7. Virtually all of the analyses and transformations performed for selective data replication's inspector and executor can be applied for prefetching as well.

Single Source Shortest Path

Section 3.1.2 presented the Single Source Shortest Path (SSSP) graph analytic and Listing 3.2 described the baseline Chapel implementation of the main kernel of the

SSSP algorithm. The remote read on line 10 can be prefetched via COPPER. Table 6.5 presents the SSSP runtime speed-ups achieved by COPPER when compared to the unoptimized baseline implementation on the scale 24, 25 and 26 graphs from Table 6.1. Additionally, Tables 6.6 and 6.7 present the execution runtimes (in minutes) for the unoptimized and optimized codes, respectively. While Chapter 5 showed that SSSP can also be optimized via remote data aggregation, specifically line 15 in Listing 3.2, the results presented here only show the performance gains from prefetching (i.e., aggregation was not enabled). Chapter 8 will present results when prefetching and aggregation are used at the same time.

The results in Table 6.5 show that speed-ups as large as 1.45x are achieved via adaptive remote prefetching. As observed with PR, prefetching does not perform well when only using two locales. When compared to the performance achieved by COPPER's aggregation optimization on SSSP (Section 5.2.2), prefetching provides smaller performance gains. This is because while aggregation is a much more specialized optimization (i.e., can only be applied to certain operations), it is more powerful. The best case scenario for prefetching is that a majority of the communication latency for remote accesses is masked. However, it may not hide all of the latency and it requires a non-negligible amount of overhead to perform bounds checking. Furthermore, prefetching attempts to optimize at a very fine-grain level, where each remote access is individually prefetched. On the other hand, aggregation requires very little overhead and operates at a more coarse-grain scale, as it locally buffers many small messages together to send as one large message.

Triangle Counting

Section 3.1.5 presented the Triangle Counting (TC) graph analytic and Listing 3.5 described the baseline Chapel implementation of the TC kernel. The remote reads on lines 4 and 9 can be prefetched via COPPER. As there are two prefetch candidates

Table 6.8: Runtime speed-ups achieved by COPPER's adaptive remote prefetching optimization on TC for the graphs in Table 6.1 relative to the unoptimized implementation.

Locales	scale-18	scale-19
2	0.96	0.97
4	1.02	1.03
8	1.03	1.12
16	1.1	1.1
32	1.05	1.06
geomean	1.03	1.05

Table 6.9: Unoptimized TC baseline execution runtime in minutes.

Locales	scale-18	scale-19
2	37	135
4	21	69
8	13	39
16	7.6	26
32	5.6	17

Table 6.10: Prefetch-optimized TC execution runtime in minutes.

Locales	scale-18	scale-19
2	39	140
4	20	67
8	12.8	35
16	7	24
32	5.3	16

in this kernel, COPPER will apply the prefetching optimization separately, creating distinct prefetch distances for each candidate. While COPPER is able to optimize two different prefetch candidates within the same `forall` loop, there are some limitations when it comes how the remote cache counters are used for both prefetch streams. This will be expanded upon further in Section 6.4.

Table 6.8 presents the TC runtime speed-ups achieved by COPPER when compared to the unoptimized baseline implementation on the scale 18 and 19 graphs from Table 6.1. Additionally, Tables 6.9 and 6.10 present the execution runtimes (in minutes) for the unoptimized and optimized codes, respectively. The performance gains that COPPER provides via prefetching for TC are noticeably smaller than those seen for PR and SSSP. While there are two remote reads that can be prefetched for TC, a majority of the remote accesses made during the TC kernel are those that COPPER cannot optimize. Specifically, the accesses on lines 11 and 14 in Listing 3.5 do not use the loop index variable within their array access expression, so COPPER will not apply

the prefetching optimization. The access on line 11 is nested within the inner most loop of the kernel, so it tends execute more times than the prefetched accesses on lines 4 and 9. Furthermore, not all elements prefetched by COPPER will be accessed. As described in Section 3.1.5 and Listing 3.5, the TC algorithm can break out of an iteration if the vertex IDs being analyzed are too large (lines 5 and 10). COPPER currently does not attempt to detect such branch-dependent accesses, so it may issue prefetches for elements that will not be accessed. Because of these reasons, the remote reads on lines 4 and 9 that are prefetched only lead to minimal improvement over the baseline code. Despite this, the prefetch optimization still provides a net improvement in most cases.

Sensitivity of Thresholds

There are three important thresholds used by COPPER's adaptive prefetching optimization, namely the late prefetch tolerance (as a percentage), the adjustment interval and the iteration sample ratio. If the percentage of prefetches marked as late exceeds the late prefetch tolerance, then the prefetch distance will be increased. If the percentage of late prefetches is consistently below the tolerance value for N consecutive calls to `adjustPrefetchDistance()` then the prefetch distance will be decreased, where N is the adjustment interval threshold. The iteration sample ratio is a parameter that determines the number of loop iterations that must elapse before the prefetch distance can be adjusted.

For the results presented thus far, a late prefetch tolerance of 10%, an adjustment interval of 8 and an iteration sample ratio of 0.01 was used. I evaluated the sensitivity of these thresholds for both PR and SSSP when executing on 8 locales for the scale-24 graph. For a given threshold evaluated, the other thresholds were set to the values noted above. Figure 6.1 presents the late prefetch tolerance sensitivity, Figure 6.2 presents the adjustment interval sensitivity and Figure 6.3 presents the it-

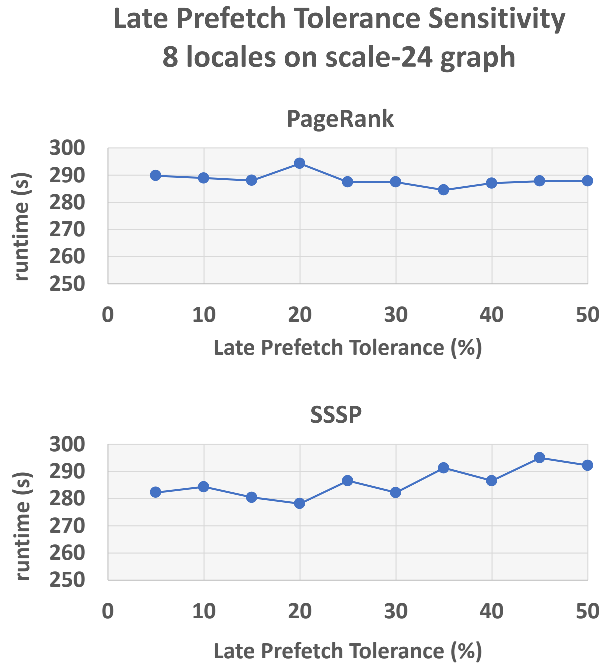

Figure 6.1: Runtime sensitivity (in seconds) when varying the late prefetch tolerance (%) for PageRank and SSSP on 8 locales and the scale-24 graph from Table 6.1. For PageRank, only 5 iterations were executed.

eration sample ratio sensitivity. As can be seen, the runtimes for both PR and SSSP are only affected slightly with different values for late prefetch and adjustment interval thresholds. For the iteration sample ratio, a value of 0.01 always provides better performance. In Section 9.2, I present results from this sensitivity study on another system with a different network interconnect than the FDR Infiniband system used in these experiments.

Prefetch Adjustment Heuristics

Section 6.2.3 described two heuristics that were considered for adjusting the prefetch distance as the program executes. One approach targets late prefetches and the other targets early prefetches, where the former is what COPPER uses by default (and what was used in the evaluations of PR, SSSP and TC). Tables 6.11 and 6.12 show the results of evaluating PR and SSSP on the scale-24 graph from Table 6.1 when using the two different adjustment heuristics.

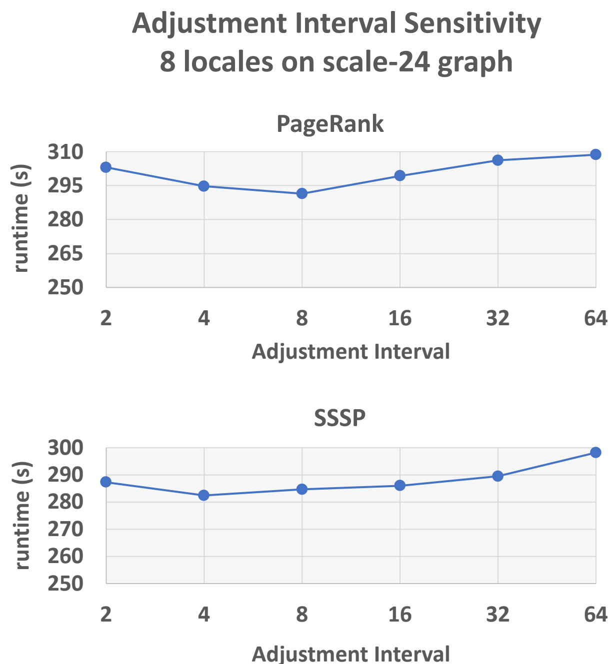

Figure 6.2: Runtime sensitivity (in seconds) when varying the adjustment interval for PageRank and SSSP on 8 locales and the scale-24 graph from Table 6.1. For PageRank, only 5 iterations were executed.

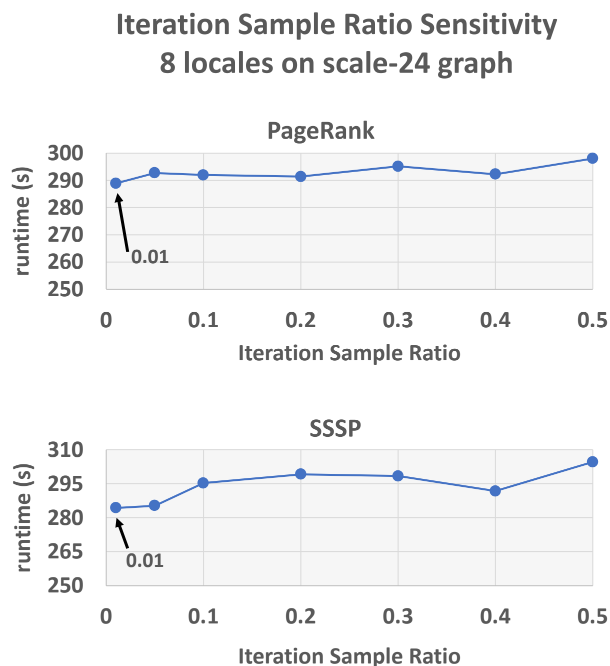

Figure 6.3: Runtime sensitivity (in seconds) when varying the iteration sample ratio for PageRank and SSSP on 8 locales and the scale-24 graph from Table 6.1. For PageRank, only 5 iterations were executed.

Table 6.11: Runtimes in minutes for PR on the scale-24 graph from Table 6.1 when using different prefetch adjustment heuristics.

Locales	early	late
2	54	48
4	23	18
8	13	10.5
16	7.5	5.8
32	4	3.4

Table 6.12: Runtimes in minutes for SSSP on the scale-24 graph from Table 6.1 when using different prefetch adjustment heuristics.

Locales	early	late
2	49	36
4	14	11
8	7.7	5.4
16	4.3	2.9
32	2.4	1.6

Across both PR and SSSP, these results show that targeting late prefetches is always the better heuristic to use. On average, the late heuristic provides a 23% improvement over the early heuristic for PR and a 40% improvement for SSSP. As noted in Section 6.2.3, by targeting late prefetches COPPER is able to make small adjustments to the distance to better mask the remote communication latency. While the remote cache performance counters provide information about how many prefetches were issued too early, it does not provide insight into *how* early the prefetches were. Without that information, a heuristic cannot make well-informed decisions about how much the distance needs to be adjusted. It would be very difficult to instrument the remote cache to measure the "earliness" of a prefetch, as that would require the use of timestamps which would be prone to measurement inaccuracy.

Adaptive Versus Fixed Prefetch Distances

A key component of COPPER's prefetch optimization is that the prefetch distances adapt as the program executes. As discussed earlier, this addresses many of the challenges that software prefetching techniques face, such as different network latencies across systems and varying irregular memory access patterns present across different applications. While the performance results for PR, SSSP and TC thus far have demonstrated that COPPER's adaptive remote prefetching optimization can provide performance gains, they do not provide insight into the effectiveness of using an adap-

tive approach versus fixing the prefetch distance to a constant value. To address this issue, I performed two sets of experiments to quantify the benefits of adapting the prefetch distance.

The first experiment compares the performance of using the adaptive heuristic that adjusts the prefetch distance based on late prefetches to the performance of holding the prefetch distance fixed at its initial value of 8. For PR on the scale-24 graph, holding the prefetch distance at a value of 8 yields performance that is 3% worse when compared to adapting the distance over the program's execution. For SSSP on the scale-24 graph, a fixed distance of 8 is 27% worse compared to adapting the distance. The SSSP kernel is more sensitive to the value of the prefetch distance because it is a much more complicated analytic. In comparing the code listings for SSSP (Listing 3.2) and PR (Listing 3.4), it is clear that SSSP's `forall` loop contains more memory accesses and branches. This leads to more variation in iteration runtimes, which means that a fixed prefetch distance is unlikely to perform well. On the other hand, PR is a fairly straightforward `forall` loop that contains a single remote access (which is prefetched) and no branches.

The second experiment directly measures how each task's prefetch distance changes over the execution of the program. Specifically, it counts the number of prefetches that were issued using a specific distance. Figure 6.4 presents heat maps for PR and SSSP when executing on the scale-26 graph across 8 locales. Each row in the map represents a task and the columns are the prefetch distances used, where lighter colors correspond to more prefetches and darker colors indicate fewer prefetches. These maps show that PR and SSSP exhibit very different prefetch distance usage patterns, despite running on the same system and using the same input graph. Chapter 9 expands this experiment to show that the prefetch distance usage pattern varies across different systems as well.

Using the results gathered from Figure 6.4, the most used prefetch distance can be

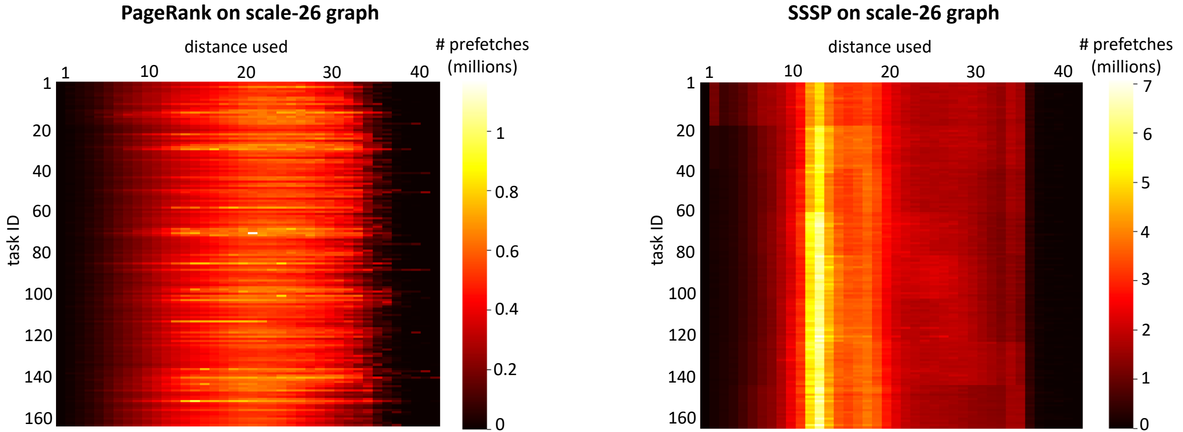

Figure 6.4: Heat maps that show each task's prefetch distance usage pattern when running SSSP and PageRank on the scale-26 graph with 8 locales (20 tasks per locale). The horizontal rows in a map represent individual tasks and the vertical columns represent prefetch distances used, which range from 1 to 40. Values in a map represent how many prefetches were issued using a given distance, where lighter colors correspond to more prefetches and darker colors correspond to fewer prefetches.

determined for each application: 22 for PR and 12 for SSSP. I then executed PR and SSSP using those distances as the initial prefetch distance and disabled the adaptive heuristic, which will hold the distances fixed. The results show that even when picking the most used prefetch distance, performance is 3% worse for PR and 36% worse for SSSP when compared to using the adaptive heuristic that targets late prefetches. When considering the real-world graphs GAP-twitter and MOLIERE_2016 for PR, the adaptive heuristic provides 16% and 44% improvements over fixing the distance to the most used value, respectively. These results directly demonstrate the effectiveness of COPPER's adaptive approach to prefetching.

6.3.3 Summary of Results

The performance evaluation of COPPER's adaptive remote prefetching optimization demonstrates that runtime speed-ups can be achieved across different graph analytics: as much as 2.6x for PR, 1.45x for SSSP and 1.12x for TC. It is important to

note that these speed-ups were obtained without modifying the baseline implementations. Instead, COPPER applies the prefetch optimization automatically, removing the burden on the programmer to detect when, where and how prefetching can be used. Furthermore, COPPER automatically adjusts the prefetch distances as the program executes, providing up to 44% better performance when compared to fixing the distance to a constant value. These results show that COPPER can successfully improve the performance and developer productivity for PGAS programs that exhibit irregular memory access patterns.

6.4 Limitations

In this section I discuss the limitations of COPPER's adaptive remote prefetching optimization and possible approaches to address these limitations.

6.4.1 Multiple Prefetch Streams

The Triangle Counting (TC) kernel that was evaluated in Section 6.3.2 contains two irregular memory accesses that can be optimized via prefetching. COPPER's static analysis and code transformations are well equipped to handle this. However, the remote cache counters that keep track of the timeliness of the prefetches are not specific to any given prefetch stream (i.e., candidate access pattern). Instead, the counters gather data for the entire remote cache for a given task/core on the system. This means that if two prefetch candidates issue prefetches with respect to the same remote cache, the counters will not be able to distinguish between them. While this is a limitation, it does not completely invalidate the optimization in the case of TC, as small performance gains are still possible. However, it does mean that the accuracy of the prefetch adjustments is not as good as it could be.

One way to address this problem is to provide per-candidate counters for each

remote cache. A given candidate can be given a unique tag that would be passed to Chapel's runtime when issuing prefetches. The data that is prefetched will then be associated with that tag. When it is time to adjust the prefetch distance for the candidate, only metrics related to its tag can be reported. However, the same data could be prefetched/accessed by different candidates, which complicates the approach of associating a tag with the access. For example, two candidates can both be issuing prefetches to the same array.

The above approach is feasible to implement for the case where the different candidates are not accessing the same data. However, it would be significantly more challenging to provide robust performance counters that can differentiate between multiple prefetch streams accessing the same data. Additionally, such modifications to Chapel's remote cache data structures to keep track of this information are likely to negatively impact its performance. This is because the remote cache implementation is highly tuned for the current design/size of the data structures. For these reasons, more robust support for multiple prefetch streams is left as future work.

6.4.2 Selecting Thresholds

Section 6.2 describes the approach and implementation of the adaptive remote prefetching optimization. An important aspect of the optimization are three thresholds that are used to control how the prefetch distance is adjusted as the program executes, namely how often to adjust the distance, the percentage of late prefetches allowed and the number of adjustment intervals before decreasing the distance. These thresholds are considered limitations because they require some degree of tuning.

The sensitivity studies I performed indicated that the chosen values for these thresholds used in the performance evaluation presented in Section 6.3 worked well across different systems and applications (performance results for prefetching on different systems is presented in Chapter 9). However, there could still be portability

issues with respect to other systems and applications not studied. To help address this issue, it would be possible to construct a set of benchmarks that could be used to set these thresholds. These benchmarks would be executed when the Chapel compiler is built/installed on a system for the first time. A similar approach is used within the ATLAS project [WPD01], which will run various micro-benchmarks to determine cache sizes to better optimize linear algebra kernels. More generally speaking, there are a variety of approaches to *auto-tuning* applications and libraries for specific hardware platforms [Bal+18]. In the context of the adaptive remote prefetching optimization, the threshold values can be set at runtime, so an auto-tuning approach does not have to generate multiple implementations of the code to select from. In this way, the selection of values is performed via an empirical search, where the best values evaluated will be set as the default.

6.4.3 Branch-dependent Prefetches

While COPPER can ensure that the prefetches it issues will be to valid memory locations, it cannot always guarantee that the data being prefetched will be accessed in a future loop iteration. Specifically, if the prefetch candidate is nested within an if-statement or otherwise controlled by the result of a branch, the elements prefetched may not be accessed. This is the case for the triangle counting (TC) kernel, as described in Section 3.1.5 and Listing 3.5. It was shown in Section 6.3.2 that performance gains are still possible via prefetching, but they were minimal.

One solution to address this issue is to simply not attempt to apply the prefetching optimization if the candidate is nested within an if-statement or if the loop iterations could be stopped via a break statement. More robust solutions would require extensive static analysis to determine the “bounds” of the branch and only issue prefetches that will be within those bounds. This can be complicated when the conditional of the branch involves variables other than constants and the loop index variable. In such a

case, it is difficult to reason about the evaluation of the branch in some future iteration of the loop. The situation presented in TC is particularly challenging because the conditional relies on the data that would be prefetched/accessed (i.e., `v.id` on line 5 and `w.id` on line 10 in Listing 3.5). Additionally, adding more bounds checking is likely to add to the existing overhead of the bounds checking that the optimization already performs.

6.5 Related Work

Hardware and software prefetching techniques have been in use for many years [Smi82; DS96; AJ17; CKP91; Bad+04]. Prefetching techniques are commonly applied to shared-memory systems and programming models, where the data being prefetched is placed into a local CPU cache to lower access latencies. While the overall techniques can be applied to prefetches for remote data on a distributed-memory system, the large cost difference between on-node memory latency and network latency is significant, which can lead to different approaches for prefetching remote data.

Ainsworth and Jones [AJ17] present a software prefetching optimization specifically for indirect memory access patterns on shared-memory systems. Their compiler pass to identify prefetch candidates is similar to COPPER's, as it looks for array accesses where "future" accesses can be determined by offsetting a loop index variable. They also do similar bounds checking to ensure that the prefetches will be issued to valid array elements. However, their static analysis does not employ interprocedural or alias analysis. Furthermore, while they calculate the prefetch distance based on characteristics of the program (e.g., number of loads in the loop), they do not adapt the distance throughout program execution. In their performance studies, they found that the prefetch distance is insensitive to different architectures and applications, where a value of 64 generally provided the best results. However, their approach tar-

gets shared-memory systems while COPPER focuses on distributed-memory systems. As a result the prefetch distance calculation is very different for COPPER, and as shown in Section 6.3.2, there can be large performance losses when not adapting the distance.

Talati et al. [Tal+21] proposed Prodigy, a hardware-software co-design for intelligent prefetching on shared-memory systems. Prodigy has a compact representation of data traversal patterns for irregular workloads and a programming model/LLVM [LA04] compiler pass to analyze a given program and extract the necessary information to construct the data traversal patterns. Prodigy then relies on hardware that is specifically designed for this traversal pattern representation. For the prefetch distance, Prodigy uses an initial value that is dependent on the length of the "path" in their traversal pattern representation, where a longer path indicates more work and a smaller distance is used. Rather than insert prefetch calls into the application, Prodigy's hardware design snoops for loads within an address range that has been designated by their traversal representation as a candidate for prefetching. When such a load is detected, a prefetch is issued with the distance described above. However, Prodigy does not provide support for adapting/throttling the prefetch distance over time. Prodigy differs from what COPPER provides in several ways. First, COPPER is not designed for, nor does it require, specific hardware. Instead, COPPER can be used on any distributed memory-system. Also, Prodigy (as presented in the paper [Tal+21]) is evaluated via a simulator and focuses specifically on shared-memory systems. Second, COPPER uses the compiler to insert prefetch calls into an application while Prodigy relies on a hardware mechanism to detect when loads are issued to prefetch-able addresses. Last, COPPER's provides support for adapting the prefetch distance over time.

Kayraklioglu et al. presented Locality-Aware Productive Prefetching Support for PGAS (LAPPS), which specifically targets Chapel programs that run on distributed-

memory systems [KFE18]. LAPPS provides a user-interface that requires programmers to specify when/where prefetching is to be applied. Furthermore, prefetching is supported through pre-defined routines/patterns that are specific to distributed arrays, such as stencil, row-wise, column-wise, transpose and all-to-all. Additionally, programmers can manually write their own prefetch patterns to be used. However, the communication induced by LAPPS to perform prefetching is blocking and does not leverage Chapel's remote cache. Instead, prefetched data and its coherency is handled by the LAPPS library and Chapel runtime (which was modified to support LAPPS). Also, their approach to prefetching is more similar to what COPPER does for data replication (Chapter 7). In contrast, COPPER's approach to prefetching is fully automated via a compiler optimization, as it statically finds candidates for prefetching without requiring user guidance. COPPER also issues non-blocking prefetches to the remote cache, which makes it more similar to traditional prefetching optimizations.

Alvanos et al. [Alv+12] present an inspector-executor approach to performing an optimization similar to prefetching for UPC [El-+05], which is a PGAS language. However, like LAPPS, it is more similar to replication than prefetching. Specifically, the target loop is strip mined so that some pre-determined number of iterations can be processed by an inspector routine. The inspector determines which accesses are remote within those iterations, and then creates local copies of the remote elements to be accessed later. This replication/prefetching appears to be performed in a blocking manner, but it is not clear from their publications. The prefetch factor, as they refer to it in the paper, is the number of iterations to look ahead and is determined by the number of iterations in the original loop, the number of accesses in the loop to distributed data and the number of processes (locales) used. On the other hand, COPPER's adaptive remote prefetching optimization will dynamically change the prefetch distance as the loop executes to automatically adjust to the performance of the prefetches themselves. Furthermore, COPPER issues non-blocking prefetches.

Chapter 7

Selective Data Replication

A common theme that has been emphasized throughout this dissertation is that remote communication is expensive, especially in the case of fine-grain remote reads that result in blocking communication. Chapter 6 showed how the COPPER framework can be used to perform adaptive prefetching for irregular memory accesses to remote data. While prefetching is a useful technique to improve performance, *replication* can provide significantly larger performance gains. Rather than issuing repeated remote accesses to some data items, replication can be used to make copies of the data on each processing element (i.e., locale/node), thereby providing continued local access. In this way, replication can exploit remote data reuse, and do so much more effectively than prefetching. Even though prefetching can be considered a form of replication, where a copy of the prefetched data is stored in a cache, prefetching must rely on the data remaining in the cache without being evicted to be reused.

However, the challenge with replication is knowing *what* to replicate. One approach is to replicate the entire set of data in question (e.g., array) across the system. Another approach is to only replicate the data elements that are accessed remotely in the portion of code being optimized (e.g., a loop). The advantage of the first approach is that it requires no additional analysis to determine what should be repli-

cated. However, it can result in replicating data that is never accessed, which will increase data movement costs when updating the replicated copies, as well as increase memory usage. The second approach requires runtime analysis to determine which elements are accessed remotely and need replication, but will be more efficient in terms of data movement and memory usage. The target applications studied in this dissertation (Chapter 3) often operate on very large input data, where full replication is not always feasible. Because of this, I focus on the second approach to replication, referred to as *selective replication*.

While selective replication can provide large performance gains, it cannot be applied as easily to programs when compared to other optimizations like prefetching. There are two reasons for this: (1) selective replication requires maintaining data coherency across the replicated copies, and (2) selective replication requires runtime analysis to determine what to replicate. For the prefetching optimization, data coherency was automatically handled via Chapel's runtime managed remote cache (and this would also be true for any prefetching approach that utilizes some form of software/hardware cache). In the case of selective replication, the data is being replicated in "user space", which requires a more manual approach to coherency. Additionally, because the replication is performed selectively, runtime analysis is needed to determine which elements are accessed remotely. This analysis is performed via the *inspector-executor* technique [SMC91; Das+94; SCF03]. The use of the inspector-executor technique places several restrictions on the types of programs that can be optimized with selective replication.

In this chapter, I describe a selective data replication optimization that is integrated into the COPPER framework (Chapter 4). The optimization uses static analysis to identify candidate memory accesses for replication and code transformations to implement the inspector-executor technique. When compared to the aggregation (Chapter 5) and prefetching (Chapter 6) optimizations presented thus far, selective

data replication is significantly more complicated in terms of both the static analysis required and the code transformations performed. However, as noted above, selective replication often provides orders of magnitude better performance than aggregation and prefetching. Specifically, the performance evaluation I present in this chapter shows that runtime speed-ups as large as 272x can be achieved via selective data replication without requiring any user intervention. Much of the work presented in this chapter was also presented in a prior papers [RKS21b; Rol+21; RS22].

7.1 Approach

In this section, I describe the overall approach taken by COPPER to perform selective data replication. At a high level, COPPER employs a combination of static analysis and code transformations to identify valid candidates for replication and generate the inspector and executor routines. As with the other optimizations that COPPER provides, selective data replication is applied automatically without any user intervention. Replication candidates must be within `forall` loops and the data to be replicated must be either (1) read-only throughout the loop, or (2) modified by an operation that can be done as a reduction. Replication is performed on a per-locale basis, where the elements of the target array that are accessed remotely from a given locale will be replicated on that locale. Because COPPER uses the inspector-executor technique for this optimization, there are additional requirements that must be satisfied regarding the candidate access, specifically its location in the program and how its memory access pattern can change throughout the program's execution. It is worth noting that there may be conflicts between the replication optimization and the adaptive remote prefetching optimization (Chapter 6), in that they both may view a particular memory access as a candidate for their respective optimization. Chapter 8 provides details about how such conflicts are resolved.

7.1.1 High Level Inspector-Executor Approach

Before explaining the details of the static analysis and code transformations that COPPER uses for selective data replication, it is worth presenting the high-level approach to the inspector-executor technique that is applied. Doing so will make the explanation of the static analysis and code transformations more clear.

The goal of selective data replication is to determine which elements of some array are accessed remotely from a given locale, and then to make local copies of those elements (i.e., replicate) on that locale. The *inspector* is a routine that executes prior to the kernel and gathers information that can be used to optimize the kernel. In the case of the replication optimization, the kernel is a `forall` loop with a candidate memory access pattern and the inspector will gather information about the remote accesses issued as part of the candidate memory access. Specifically, the optimization keeps track of which elements are accessed remotely from each locale. I will refer to this remote communication information as a *communication schedule* [ASS95]. The *executor* is an optimized version of the `forall` that uses the communication schedule provided by the inspector to replicate the remotely accessed elements prior to the loop. Within the executor loop any remote accesses to the target array will be redirected to the replicated copies. In this way, each replicated element requires one remote access to replicate/update but can then be accessed locally within the loop as many times as needed, assuming the elements are read-only in the loop. Handling writes to the replicated data will be discussed in more detail in Section 7.1.4.

As the inspector gathers data that pertains to the candidate memory access pattern in the `forall` loop (i.e., the communication schedule), it must be kept up-to-date. If the candidate memory access pattern changes in some way, that could result in different elements being accessed remotely in the loop, or the same elements being accessed from different locales. In both cases, if the executor is not provided with this new communication schedule then the optimization will produce incorrect program

behavior. To address this issue, the inspector must run again if the memory access pattern may have changed. A major role of COPPER is using static analysis to determine when it is necessary to run the inspector and then use code transformations to insert procedure calls to toggle a flag on/off at runtime that will enable/disable the inspector. As will be described in Section 7.1.2, there are several factors that can contribute to altering the memory access pattern for some candidate.

Another crucial consideration for the inspector-executor technique is the fact that the inspector routine will incur a non-negligible amount of overhead. As a result, the performance gains provided by the executor must be large enough to offset the inspector's cost. In most cases, this cannot be achieved if the `forall` loop is only run once throughout the program's execution (i.e., the executor is only executed once). Instead, a common requirement for any inspector-executor application is that the kernel is executed several times as part of some outer loop in the program. In this way, the cost of the inspector is amortized over multiple executions of the executor. The more times that executor can run without having to run the inspector, the larger the performance gains will be.

7.1.2 Static Analysis

The static analysis performed by COPPER for the selective data replication optimization has four steps: (1) identifying candidate memory access patterns in `forall` loops, (2) determining if the optimization can or cannot be applied, (3) determining if the optimization should or should not be applied and (4) determining when to run the inspector routine. The details of each of these steps will be described below. Unlike COPPER's other optimizations (Chapters 5 and 6), the static analysis for selective data replication is more evenly divided between Chapel's compiler passes rather than being mostly performed during the normalize pass. This is because the static analysis for replication is much more complicated than what is employed for aggregation

and prefetching. For example, replication requires that modifications (writes) to a given array/domain can be found, which can only be performed during the cull-over references pass.

Identifying Candidate Memory Access Patterns

The first step of the static analysis is to look for candidate memory access patterns within `forall` loops. The candidate must be directly within the `forall` loop's body (i.e., not within a procedure which is called from the loop body). To recognize candidate access patterns that are in procedures called from within the `forall` loop, the program's call graph needs to be available to perform interprocedural analysis. However, the call graph is not available until after the resolve pass in Chapel's compiler, at which point performing the necessary code transformations for the optimization is difficult. As with the previous optimizations, candidates are generally defined as a distributed array that is indexed by another array access, such as $A[B[i]]$ where elements of A are the target of the replication. These non-affine access patterns are likely to result in fine-grain remote communication, and their access pattern cannot be inferred at compile time.

However, a crucial difference between the selective data replication optimization and COPPER's other optimizations is that replication relies on the inspector-executor technique. Because of this, the candidate memory access pattern must be structured in such a way that COPPER can reason about *when* and *how* its access pattern could change. Such a change would require the inspector to be executed to update the communication schedule used by the executor. More details regarding the supported memory access structure will be provided below.

Determining When the Optimization Cannot Be Applied

A majority of the static analysis that COPPER applies for replication is used to determine if the optimization is invalid and cannot be applied. In this regard, the optimization cannot be applied if it would lead to incorrect program behavior with respect to the original code. Incorrect program behavior can arise if the inspector is not executed when it should be, leading to out of date communication schedules provided to the executor. It is not satisfactory to simply run the inspector each time the `forall` is encountered since the goal of the optimization is to amortize the cost of the inspector over multiple executions of the executor. Therefore, COPPER must be able to statically determine when the candidate memory access pattern could change to know when to run the inspector. The following situations describe when the optimization cannot be applied for correctness reasons:

1. The `forall` loop that contains the candidate access is nested within another `forall`, `coforall` or any other structure that can create parallel tasks.

2. The `forall` loop that contains the candidate access does not iterate over a distributed array/domain.

3. The variables used in the candidate access array expression are not elements from an array/domain.

4. The array B or its domain from the candidate $A[B[i]]$ is modified in the `forall` loop that contains $A[B[i]]$.

(1) is problematic because it could lead to concurrent modifications by parallel tasks to the communication schedule (see Section 7.1.3). To determine that there are no enclosing `forall`/`coforall` loops requires fairly involved analysis if the candidate memory access is in a `forall` that is part of a procedure. In this situation, there could be multiple paths to the procedure within the program's call graph, where

some paths may be invalid according to (1). Rather than simply cancelling the entire optimization, COPPER uses call path analysis to determine which paths are valid/invalid, and then enables/disables the optimization along those paths. This analysis is performed via a depth first search on the call graph that starts from the procedure that contains the target `forall` loop and looks for invalid enclosing structures. Then COPPER uses runtime control flow analysis and transformations (Section 4.2.2) to insert procedure calls that will toggle flags on/off at runtime to indicate when a call path is valid or invalid. More details regarding the code transformations performed to accomplish this analysis are provided in Section 7.1.3.

The importance of (2) is subtle, as it deals with the implicit locale affinity of the `forall` loop (i.e., which locales the iterations are executed on). The locale affinity is important because replication is performed on a per-locale basis, and the locale affinity dictates which locales issue remote accesses to the target array. By requiring the `forall` loop to iterate over a distributed array/domain, it becomes possible to statically determine when the `forall` loop's locale affinity could change by looking for modifications to the array/domain. The `forall` loop must iterate over a distributed array/domain because that overrides any other "outer" locale affinity control constructs, like an `on` statement.

(3) is required so that COPPER can statically determine when a different index will be used to access the target array. As long as any variable used in the candidate access array expression is an element of an array/domain, COPPER can track modifications to those arrays/domains to know when the index expression could change (and hence, require the inspector to run). If the index variables are not derived from an array/domain then it becomes very difficult to statically determine when their values could change. For example, they could be the return value of some procedure call that has non-deterministic behavior. To this end, the allowed expressions within the candidate memory access are those that involve binary operations (i.e., operations

with two operands) between immediates and variables as described above. By only allowing binary operations, that excludes procedure calls, record field accesses, etc., which are complicated to statically track for changes.

(4) is required because changes to B or its domain directly inside of the loop would be at odds with the inspector's role. The inspector analyzes the remote memory access pattern of the $A[B[i]]$ candidate within the `forall` loop so that the executor can be created to perform replication. The requirement is that what the inspector observes about the loop is true when the executor is performed; otherwise, the inspector's analysis is irrelevant. The requirement that B's domain cannot be modified is somewhat subtle. Modifying an array's domain can lead to modifications to the array elements themselves. For example, if the domain is grown to a larger size then new elements are added to the array declared over the domain.

Determining When the Optimization Should Not Be Applied

While it is important for COPPER to know when replication cannot be performed for correctness reasons, it is also important for COPPER to know when replication should not be applied for performance reasons. The end goal of the selective data replication optimization is to improve the runtime performance of some application. However, as discussed in Section 7.1.1, the inspector routine creates overhead that must be amortized by multiple executions of the optimized version of the loop (i.e., the executor). COPPER can perform a few checks to identify situations in a program where performance gains are unlikely due to the inspector having to be executed too many times. These situations where performance gains are not likely are described below:

(i) the `forall` that contains the candidate access is not nested in an outer serial loop.

(ii) the array B or its domain from the candidate access $A[B[i]]$ is modified directly

outside of the `forall` that contains $A[B[i]]$.

(iii) the domain of the array A from the candidate $A[B[i]]$ is modified directly outside of the `forall` that contains $A[B[i]]$.

Case (i) would result in only one execution of the executor, and therefore almost always lead to performance loss because the overhead of the inspector is not amortized. By ensuring that the `forall` loop with the candidate memory access is nested in an outer serial loop (e.g., `for`, `while`, etc.), it at least presents the possibility that the executor will be executed multiple times. As with check (1) regarding program correctness, looking for an outer loop is complicated when the target `forall` is nested within a procedure. However, the same static analysis and runtime control flow analysis/transformations can be used to evaluate the program's call graph and look for call paths with outer loops. More details on the code transformations for this step will be provided in Section 7.1.3.

If case (ii) is detected, that would result in the inspector running every time the executor is performed. Recall that that inspector is executed any time the candidate memory access could have changed, which includes when the array B or its domain is modified. If this modification happens directly outside of the `forall` loop but inside of the outer serial loop (if present), it would result in the inspector being executed each time the `forall` loop runs. This situation will most likely lead to a performance loss due to not amortizing the inspector's cost over multiple executions of the executor.

Case (iii) is not as obvious as the other two cases, but it can also lead to the inspector being executed every time the executor is performed. If A's domain is modified, that creates the possibility of altering the mapping of array elements to locales and/or introducing new elements through domain resizing. Such a change would lead to a different access pattern, and require the communication schedule to be updated. As with case (ii), if this modification occurs directly outside of the target `forall`, it will result in the inspector being performed each time the `forall`

is encountered. It is worth noting that the array A itself (i.e., the elements) can be modified without any performance concerns. This is because one of the roles of the executor is to replicate/update the remotely accessed elements of A prior to the `forall`, as will be discussed in more detail in Section 7.1.3.

Determining When to Run the Inspector

A crucial part of the inspector-executor technique as applied to selective data replication is determining when the inspector needs to run. If the inspector does not run when it needs to, the executor will have out-of-date communication schedules and potentially lead to incorrect program behavior. Generally speaking, the inspector must run the very first time it is encountered and any other time afterwards if the candidate memory access pattern could have changed. COPPER uses static analysis to determine when the memory access pattern could be altered and then inserts procedure calls that are executed at runtime to enable/disable the inspector. Below are the situations that result in potential changes to the candidate memory access pattern $A[B[i]]$, resulting in the inspector being executed:

(a) The value of i has changed due to the loop iterator which yields i being modified.

(b) The arrays/domains that yield any other variables used in the candidate array access expression are modified.

(c) The contents of the array B or its domain are modified.

(d) A's domain is modified.

(e) The distributed array/domain that the `forall` iterates over is modified.

Cases (a)–(d) will result in different values being used to index into A or the accesses being issued to different locales, which will likely lead to a different remote access pattern. Note that these modifications are assumed to be "valid" in the sense that

they do not match any of the cases described earlier regarding when the optimization cannot and should not be applied. Case (e) has the possibility to alter the locale affinity of the `forall`, which can result in the remote accesses to A being issued from different locales. All of the above situations require information that is not provided to Chapel's compiler until the cull-over references pass, which is when argument intents are known. As discussed in Section 4.2.1, argument intents can be used to determine if a particular access is a read or write. The role of the static analysis pass for this step is to locate these modifications. COPPER will then use code transformations to enable a runtime flag that indicates the inspector needs to run (Section 7.1.3).

Interprocedural and Alias Analysis

As described thus far, a large portion of COPPER's static analysis is used to locate both valid and invalid modifications (writes) to the various arrays and domains that are part of the selective data replication optimization. However, the methods described so far are not adequate to fully support replication and would only provide correct program behavior in simple cases. For example, one requirement that COPPER enforces is that the array B from the candidate $A[B[i]]$ cannot be modified within the `forall` loop being optimized. The current analysis will look for accesses made to B within the loop and determine if any are writes. However, the analysis could not determine if there is a procedure call within the loop that takes in B as an argument and then modifies B. Similarly, if an alias was made to B (i.e., `ref foo = B`) and then modified, the analysis would fail to detect the modification to B.

COPPER addresses this by employing both interprocedural and alias analysis, which are crucial to extending selective data replication to real-world applications. The role of interprocedural analysis is to track the accesses to the arrays/domains across procedure calls and determine whether those accesses are writes. The analysis can work across arbitrarily long procedure call paths. Alias analysis is used to deter-

mine the existence of references made to/from the arrays/domains, and then track modifications to those aliases. COPPER's alias analysis is able to handle arbitrarily long alias "chains". It is worth noting that Chapel's existing compiler infrastructure does not provide pre-built passes to perform this type of interprocedural and alias analysis.

Supporting Records

A consideration for any replication technique is memory usage, which can increase significantly when replicating large amounts of data. Additionally, the data type of the array being replicated may not be primitive (e.g., `int`, `real`, `bool`), but a record (i.e., struct). The fields of a record can store domains, arrays, other records, classes, etc. Naively replicating the entire record can lead to unnecessary memory usage increases and unnecessary data movement when performing replication. COPPER addresses this by only replicating the fields of a record that are accessed within the `forall` being optimized.

Using a similar analysis technique as described for the adaptive remote prefetching optimization (Section 6.2.1), COPPER determines the data type of the target array during the resolve compiler pass. If the data type is a record, static analysis is used to locate accesses issued to the array and internally log which fields are accessed. These fields are then used within the code transformations to create a routine that will only replicate/update those fields of the record (see Section 7.1.3). This record field analysis also leverages both interprocedural and alias analysis to find field accesses across procedure calls and references made to the target array.

7.1.3 Code Transformations

The code transformations performed by COPPER for the selective data replication optimization have the following steps: (1) creating compile time checks, (2) creating

the inspector, (3) creating the executor, (4) turning the inspector on, and (5) enabling dynamic call path analysis. Unlike the aggregation and prefetching optimizations described in Chapters 5 and 6, replication requires extensive loop cloning that can only be performed during normalization. However, replication also requires some code transformations to be performed during the resolve and cull-over references passes. Listing 7.1 presents an example `forall` loop that contains a valid replication candidate (line 3), followed by code that is equivalent to the output of COPPER's code transformations.

Internal Data Structures for Replication

Before explaining the details of the code transformations, I describe the internal data structures used to store the replicated data. For the distributed array A from the candidate $A[B[i]]$, a new record is created on each locale that contains the remote communication information about the $A[B[i]]$ access on that locale. Specifically, these records are implemented as fields in the internal class that implements the distributed array object in Chapel. The records store the communication schedule for the optimization. A communication schedule is essentially a set of associative arrays that map the index $B[i]$ to the value $A[B[i]]$ when $A[B[i]]$ is a remote access issued from a given locale. If A is associated with multiple `forall` loops that are being optimized for replication, COPPER will create different communication schedules for A that are linked to each `forall` loop via a hash/unique identifier.

Associative arrays are used for the communication schedules because they can represent non-contiguous indices. This means that the original "global" index $B[i]$ can be directly used to get the replicated copy of $A[B[i]]$ from the per-locale communication schedule array. However, accessing an associative array is about 2 - 3x slower than accessing a default Chapel array. To use default arrays, the optimization becomes much more complicated, as COPPER would need to construct a new mapping of the

original global $B[i]$ indices to per-locale local arrays, which have their own domain indices. To accomplish this, B would need to be modified in such a way that $B[i]$ now contains a new index to the communication schedule arrays. This would require either modifying the contents of B or making a full copy of B. The former is risky, as it modifies the user's data and requires extensive transformations to "undo" the changes when the optimization is not being performed. The latter is more feasible, but can lead to even more memory usage increases on top of the replication of the elements of A. For these reasons, COPPER relies on associative arrays instead.

```
1  // Original forall loop
2  forall i in B.domain {
3    C[i] = A[B[i]];
4  }
5
6  // Code equivalent to the output of the code transformations
7  if isCallPathValid(funcID) { // runtime check
8    if replicationIsValid(A, B) { // compile time check
9      if doInspector(A, B) {
10       inspectorPreamble(A);
11       forall i in inspectorIter(B.domain) {
12         inspectAccess(A, B[i]);
13       }
14       inspectorPostamble(A);
15       inspectorOff(A, B);
16     } /* end doInspector() */
17
18     executorPreamble(A);
19     forall i in B.domain {
20       C[i] = executeAccess(A, B[i]);
21     }
22   } /* end if replicationIsValid() */
23   else {
24     forall i in B.domain {
25       C[i] = A[B[i]];
26     }
27   }
28 } /* end if isCallPathValid() */
29 else {
30   forall i in B.domain {
31     C[i] = A[B[i]];
32   }
33 }
```

Listing 7.1: Example of a replication candidate and the resulting code transformations applied by COPPER. `C[i] = A[B[i]]` is the replication candidate.

Create Compile Time Checks

As with the other optimizations that COPPER performs, a compile time check is added during the normalize compiler pass to check whether replication can be performed (line 8 in Listing 7.1). The code transformations begin by cloning the original `forall` loop, where one version will be used in the optimization while the other will be a "fall back" in case the optimization cannot be applied. The routine `replicationIsValid()` checks for basic properties, like whether `A` and `B` are arrays and whether `A` is a distributed array. If these compile time properties are not satisfied, the routine will return false and result in the entire then-branch (lines 9–22) being removed after the normalize pass, effectively eliminating the optimization. In such a case, the original `forall` loop is left to be executed (lines 23–27).

For the more complicated checks that are performed after the normalize pass (e.g., looking for invalid modifications to the arrays/domains), a different approach is required. Chapel's automatic dead-code removal that is leveraged between the normalize and resolve passes does not apply between the resolve and cull-over references passes (note that more traditional dead code elimination is still applied by Chapel in later passes). Therefore, for the static checks that are performed during the resolve and cull-over references passes, COPPER must manually remove the optimization (lines 9–22) if something is found that invalidates the optimization.

Create Inspector

The first major step in COPPER's code transformations is creating the inspector routine. The inspector consists of a conditional check, a few procedure calls and a clone of the original `forall` that is modified to perform the remote memory access analysis. This clone of the `forall` loop was created when the compile time check was inserted. The inspector is naturally parallelized since it is created from a `forall` loop. Line 9 in Listing 7.1 shows the conditional that checks whether the inspector should be

performed. Each replication candidate is associated with a unique flag that is used to determine whether its associated inspector should run. The `doInspector()` routine simply checks this flag.

Note that a replication candidate is defined by more than just the array `A`, but also the array `B`. Specifically, COPPER will compute a unique hash from `A`, `B` and the other arrays/domains that define the candidate. This allows COPPER to support multiple replication optimizations (i.e, `forall` loops) that use the same target array. Furthermore, this allows COPPER to correctly handle cases where the `forall` is in a procedure with multiple call sites, where `A` and `B` are passed in as arguments. As each call site does not necessarily pass in the same values of `A` and `B`, it is necessary to associate each call site with a unique hash that allows COPPER to know which communication schedule to use.

The `inspectorPreamble()` procedure call is inserted on line 10 to initialize the communication schedules (e.g., clears the associative arrays so the new schedule can be added). Lines 11–13 comprise the inspector loop, which is a modified clone of the original `forall` loop. The loop's iterator is modified to instead use the custom `inspectorIter` iterator. This custom iterator creates one task on each locale via a `coforall` loop to execute that locale's portion of the original `forall` loop serially. This is similar to lines 14–18 in Listing 2.5 from Section 2.4.2, but instead of a `forall` loop iterating over the elements of a local subdomain, it is a serial `for` loop. While this does reduce the total amount of parallelism that the default iterator provides, it enables turning off parallel-safety for the underlying associative arrays in the communication schedule, which generally provides performance improvements when accessing the communication schedules during the executor.

The body of the inspector loop is identical to the original `forall` loop to begin with, but COPPER removes any statement that is not relevant to the candidate memory access (`A[B[i]]` in this case). This keeps the inspector's overhead to a minimum.

The original candidate access is then replaced with a call to `inspectAccess()` (line 12), which performs a simple check of whether the index `B[i]` is a remote access to `A` from the current locale. This check can be performed without actually performing the remote access, since Chapel provides the `.contains()` method on domains to check whether a value is present in the domain. If the access is determined to be remote, then an entry for `B[i]` is created in the communication schedule. Regardless of how many times an element is accessed remotely from a locale, it is only replicated and stored once in the communication schedule. Associative domains automatically ignore duplicate entries, so this is easy to support. Worth noting is that the actual replication has not occurred at this point.

The last two steps of the inspector routine perform post processing and turn off the inspector flag. Posting processing is performed via a call to `inspectorPostamble()` on line 14. The main task performed here is creating a sorted version of the indices stored in the communication schedules. Recall that the communication schedules are implemented as associative arrays, and that associative arrays/domains do not store their indices in any particular order (i.e., they are more or less unordered maps). As will be seen below, the executor must iterate through the communication schedules to perform replication, and doing this on the unordered indices will severely degrade performance. To address this, COPPER creates a sorted array of each locale's associative array indices to be used by the executor. Finally, line 15 turns off the inspector flag associated with this candidate so that the inspector will not execute the next time the `forall` loop is encountered (unless the flag is turned on again).

Create Executor

The executor consists of two parts: the executor preamble and the optimized version of the `forall` loop. The executor preamble is performed by `executorPreamble()` on line 18 in Listing 7.1. Its role is to actually perform the replication according to the

communication schedule that the inspector created. To do this, a `coforall` loop is used to create a task on each locale. Each task then uses a `forall` loop to iterate over all of the `B[i]` indices that were logged by the inspector on the given locale. For each `B[i]` index, a remote read is performed to `A[B[i]]` to replicate the corresponding element of `A` on the locale. As noted above regarding the inspector postamble, this `forall` loop iterates over the sorted indices rather than the associative arrays directly. Because the accesses to replicate elements of `A` are strictly remote accesses, using the sorted indices significantly improves remote data locality within Chapel's remote cache (see Section 6.1). Furthermore, because replication is performed as simple assignment in a `forall` loop, aggregation can be used to improve performance. This form of aggregation is separate from what COPPER applies automatically (Chapter 5), as it deals with remote reads rather than remote writes, and is applied manually since the `executorPreamble()` procedure is hidden behind a library/module.

The `forall` loop on lines 19–21 is another clone of the original `forall` loop but with a single modification, namely replacing `A[B[i]]` with `executeAccess(A, B[i])`. The `executeAccess()` procedure checks whether `B[i]` will be a remote access into `A`, and if so, it will return the current locale's local copy of `A[B[i]]` that was replicated by `executorPreamble()`. If the access would already be local, the original `A[B[i]]` is returned. In this way, COPPER incurs the cost of one remote access per replicated element when performing `executorPreamble()`, but can amortize that cost over multiple local accesses during the `forall`, thereby exploiting remote data reuse. Not shown in Listing 7.1 is how the executor is created when dealing with writes to replicated data. This will be described in Section 7.1.4.

When `A` stores records, COPPER will insert a special call to `executorPreamble()` that also takes in a list of the record fields that are accessed in the loop. These fields were determined by COPPER during static analysis. The specialized version of `executorPreamble()` will only replicate those specified fields, which reduces unnec-

essary memory usage and data movement that would occur by replicating the entire record.

Turn Inspector On

During the cull-over references compiler pass, COPPER's static analysis detects all the locations within the program that issue modifications (writes) to the various arrays/domains involved in the replication optimization. Modifications to these arrays/domains require the inspector to run to provide an updated communication schedule to the executor. Using the program locations determined during static analysis, COPPER inserts procedure calls to turn on the inspector flag for the relevant replication candidate. Then, when `doInspector()` executes on line 9 in Listing 7.1, it will find that the flag is set and perform the inspector.

Enable Dynamic Call Path Analysis

As noted in Section 7.1.2, COPPER performs static analysis to look for "valid" call paths for a given replication candidate. A call path is valid if there exists an outer serial loop somewhere on the path, and there are no enclosing `forall`/`coforall` loops along the path. Rather than rely only on static analysis to find these valid call paths, COPPER uses information that is available at runtime. The role of the static analysis is locate the program locations of all outer serial loops and enclosing `forall`/`coforall` loops along each call path. COPPER then uses code transformations (at compile time) to insert procedure calls to toggle flags on/off at each of the program locations identified during the static analysis.

When the program executes and traverses the call path to the target `forall` loop with the replication candidate, the call to `isCallPathValid()` on line 7 in Listing 7.1 is executed. This procedure is parameterized by a unique function ID provided by COPPER and checks the corresponding flags for the presence of an outer serial loop

and the absence of enclosing `forall`/`coforall` loops. If a call path is determined at runtime to be invalid, then the check on line 7 will be false and the else-branch will execute, which contains the unoptimized/original `forall` loop. A simple example of how this runtime control flow analysis and transformation looks is provided in Listing 4.6 in Section 4.2.2.

7.1.4 Supporting Writes to Replicated Data

When the target array A from the candidate memory access $A[B[i]]$ is read-only throughout the entire `forall` loop being optimized, replication can be applied as described thus far. Data coherency of the array elements across replicated copies is handled before the executor by the `executorPreamble()`. However, there could be programs where elements of A are modified within the `forall` loop. This leads to a problem because such elements may be replicated on different locales, leading to multiple copies of the same elements with potentially different values. Without properly "combining" these partial writes to compute the intended final value, the optimization will result in incorrect program behavior. Therefore, to support modifications to replicated data the following must be true: (1) the modification must be a reduction operation, and (2) the data being modified cannot be accessed within the loop after the modification. These requirements are nearly identical to those used by COPPER's remote data aggregation optimization (Chapter 5).

With respect to (1), a modification is a reduction operation if the modifications to the replicated copies can be combined together after the `forall` loop to compute what the value would have been without replication. Such operations can be referred to as *reducible* operations. For example, addition (`+`) is a reducible operation because partial sums can be combined (in any order) to produce a final sum. Likewise, multiplication (`*`) is a reducible operation, but division (`/`) is not. (2) is then necessary to ensure that the partial modification results will not be accessed throughout the

rest of the loop body. As with aggregation, because COPPER focuses on `forall` loops that are asserted by the programmer to have order-independent iterations, this data dependency analysis only needs to consider a single iteration of the loop (i.e., it is assumed that there are no loop-carried dependencies).

However, another complication with supporting replicated writes is ensuring that the initial values of the data are compatible with the type of reduction that will be performed. For example, in the simple example of a parallel sum reduction, it is often assumed by the programming model (e.g., OpenMP) that the initial value of the sum is set to 0. In general, each reducible operator has an identity value: 0 for sum, 1 for multiplication, etc. Because COPPER is applying replication in an automated fashion without user intervention or guidance, it cannot rely on assumptions about the initial values of the replicated elements prior to the `forall` loop.

To address this, COPPER inserts a runtime check within the `executorPreamble()` procedure that ensures the to-be-replicated elements have the correct identity value. This check is significantly easier to perform at runtime, and incurs relatively little overhead. Additionally, COPPER inserts a call to the `executorPostamble()` procedure to perform the reductions across each replicated copy to form the final result. This procedure call is inserted right after the executor-optimized `forall` loop. As COPPER requires that the write operations be order independent, aggregation can be performed when carrying out the reductions, which reduces the overhead incurred by `executorPostamble()`.

COPPER currently supports writes to replicated data specifically in the case of atomic operations, such as atomic adds (`.add()`). Prior works have shown that automatic reductions can be applied by the compiler to more operators [RP99], but their use cases involved automatically parallelizing loops. In the applications that COPPER targets, the loops are already parallelized. So while non-atomic adds are reducible, their presence in `forall` loops leads to race conditions and non-reproducible

behavior, which is the responsibility of the programmer. Because of this, COPPER only focuses on scenarios where reproducible behavior can be achieved. This same reasoning is used for the aggregation optimization (Section 5.1.1).

7.2 Performance Evaluation

This section presents a performance evaluation of COPPER's selective data replication optimization. The goal of the evaluation is to demonstrate that significant performance improvements can be achieved via COPPER compared to baseline implementations of the irregular applications described in Chapter 3. Some of the memory access patterns that are optimized in this evaluation are also candidates for adaptive remote prefetching and were evaluated in Section 6.3. As a result, direct comparisons can be made between the performance gains provided by replication and those provided by prefetching. The baseline codes fully leverage the productivity advantages of the PGAS model and Chapel, but suffer severe performance issues due to fine-grain communication. As the runtime improvements are achieved without modifying the original code, these results demonstrate that performance gains can be obtained without sacrificing user productivity, enabling users to develop irregular applications without having to deal with the issues posed by fine-grain communication.

7.2.1 Experimental Setup

The experiments presented in this section were executed on the same 32-node FDR Infiniband cluster from Section 5.2 and Section 6.3, and use the same Chapel installation and configuration settings. The applications and kernels evaluated for the selective data replication optimization are conjugate gradient from the NAS Parallel benchmark suite (NAS-CG), PageRank (PR), and molecular dynamics simulation (Moldyn). The descriptions and implementations of these codes are presented in

Table 7.1: Problem sizes for NAS-CG. Each problem size refers to the properties of the sparse matrix A from $Ax = b$.

Name	# Rows	# Nonzeros	Density (%)	# of SpMVs
C	150K	39M	0.17	1950
D	150K	73M	0.32	2600
E	9M	6.6B	0.008	2600
F	54M	55B	0.002	2600

Chapter 3. The primary metrics reported for each experiment are application runtime and the runtime speed-up achieved by COPPER relative to the unoptimized baseline implementations. For each experiment, multiple trials were performed and the average of these trials is presented. The variation in runtime across all trials did not exceed 5%.

7.2.2 Results

Each of the following experiments leverage different data sets for input, since the applications evaluated are widely different. As a result, each of the following sections will describe the application evaluated, the data sets used and their results. A summary of the results will be presented in Section 7.2.3.

NAS-CG

The NAS-CG benchmark, as described in Section 3.2.1, can be optimized via COPPER's selective data replication optimization, specifically the remote read to `x` performed on line 4 in Listing 3.6. NAS-CG's data sets are shown in Table 7.1 and are generated according to the benchmark specifications and represent the sparse matrix A from $Ax = b$. Different problem sizes also specify the total number of iterations to perform, which affect the number of sparse matrix-vector multiplies (SpMVs) that are executed. These SpMVs are the target of the replication optimization.

Table 7.2 presents the NAS-CG runtime speed-ups achieved by COPPER when compared to the unoptimized baseline implementation on the benchmark problem

Table 7.2: Runtime speed-ups achieved by COPPER's selective data replication optimization on NAS-CG for the problem sizes in Table 7.1 relative to the unoptimized implementation. Missing values indicate that too much memory was required to execute the unoptimized implementations.

Locales	C	D	E	F
2	5.8	4.6	273	—
4	7.5	5.3	215	—
8	42	50	230	—
16	97	130	143	166
32	68	100	225	224
geomean	26	28	213	193

Table 7.3: Unoptimized NAS-CG baseline execution runtime in minutes. Missing values indicate that too much memory was required to execute the problem size.

Locales	C	D	E	F
2	6.7	12.8	1.5e+5	—
4	6	10.5	7e+4	—
8	22	53	4.2e+4	—
16	40	100	2.3e+4	1.9e+5
32	27	67	1.2e+4	9.9e+4

Table 7.4: Replication-optimized NAS-CG execution runtime in minutes. Missing values indicate that too much memory was required to execute the problem size.

Locales	C	D	E	F
2	1.2	2.8	557	—
4	0.8	2	327	—
8	0.5	1.1	181	—
16	0.4	0.8	159	1141
32	0.4	0.7	52	442

sizes specified in Table 7.1. Additionally, Tables 7.3 and 7.4 present the execution runtimes (in minutes) for the unoptimized and optimized codes, respectively. Because problem sizes E and F are very large and the baseline implementation of NAS-CG is severely hindered by fine-grain remote communication, the results reported for sizes E and F were projected by running a single iteration of the NAS-CG benchmark. Each iteration of NAS-CG performs identical computation and communication, so there is little variance in runtime between multiple iterations. Last, problem size F requires too much memory to be executed on fewer than 16 locales, even for the baseline implementation which does not perform replication. In regards to memory storage overhead due to replication, there is an average increase in memory usage of only 6%.

Table 7.2 shows that speed-ups as large as 273x can be achieved via selective data replication. Indeed, super linear speed-ups can be seen over the unoptimized

baseline code. This is due to the fact that the optimized code is significantly faster in total execution time and because the baseline code generally does not scale as the number of locales increase while the optimized code achieves rather good scalability (see Tables 7.3 and 7.4). The optimized code is significantly faster because it is able to exploit the large amount of remote data reuse that is present in the SpMV kernel, which comes from the sparsity pattern of the matrices generated by the benchmark. As noted before, COPPER can amortize the single remote access needed per replicated element over multiple local accesses within the executor. Without COPPER, such data reuse is unobtainable due to the irregular memory access pattern, leading to the slow execution times. Furthermore, remote communication generally increases as the number of locales increase, leading to the poor scalability of the baseline code.

Within NAS-CG, the SpMV kernel is executed many times during the NAS-CG benchmark and only requires the inspector to be performed once because the target memory access pattern does not change. This allows for the inspector overhead to be amortized to the point of only contributing to 3% or less of the total execution time. This further contributes to the significant performance gains of COPPER over the baseline code. While the synthetic matrices used by the NAS-CG benchmark may not reflect real-world applications, they provide useful insight into the performance of the replication optimization when presented with ideal circumstances (i.e., significant amounts of data reuse and many outer iterations).

It is worth noting that COPPER's adaptive remote prefetching optimization (Chapter 6) can also be applied to the same memory access pattern in NAS-CG that replication targets. As will be discussed in Chapter 8, the two optimizations cannot be applied to the same access pattern at the same time, since replication effectively eliminates any remote accesses that could be prefetched. However, I evaluated NAS-CG when optimized via prefetching and found that speed-ups as large as 1.4x could be achieved, which is significantly less than what replication achieves. As alluded to at

Table 7.5: Graphs for PageRank.

Name	# Vertices	# Edges	Density (%)	# of Iterations
arabic-2005	23M	631M	1.2e−4	52
webbase-2001	118M	992M	7.1e−6	33
GAP-twitter	61M	1.5B	3.9e−5	18
scale-26	67M	2B	4.8e−5	11
MOLIERE_2016	30M	6.6B	7.3e−4	59

the beginning of this chapter, replication can be a much more powerful optimization than prefetching due to its ability to exploit remote data reuse.

Last, the unoptimized baseline code has such poor runtime performance, as shown in Table 7.3, that it can be considered impractical for the large problem sizes. For example, problem size F on 16 locales is projected to require 131 days to complete. By comparison, COPPER's selective data replication can reduce this runtime to 19 hours.

PageRank

The PageRank (PR) graph analytic, as described in Section 3.1.1, can be optimized via COPPER's selective data replication optimization, specifically the remote read performed on line 5 in Listing 3.4. PR uses a subset of the graphs from the adaptive remote prefetching evaluation (Section 6.3) and those graphs are included in Table 7.5 for easier reference during this evaluation. The number of iterations that are required to converge using a tolerance of 1*e*-7 is also presented in Table 7.5, which is relevant for the inspector-executor technique used by COPPER for selective data replication. The tolerance value of 1*e*-7 was also used in the prefetching experiments and is used by Neo4j [Web12], an open-source graph database. The memory access pattern within each iteration does not change throughout PR, so the inspector only needs to run once.

Table 7.6 presents the PR runtime speed-ups achieved by COPPER compared to the unoptimized baseline implementation on the graphs from Table 7.5. Additionally,

Table 7.6: Runtime speed-ups achieved by COPPER's selective data replication optimization on PageRank for the graphs in Table 7.5 relative to the unoptimized implementation.

Locales	arabic-2005	webbase-2001	GAP-twitter	scale-26	MOLIERE_2016
2	2.4	3.1	33	5.7	36
4	1.5	2.8	9.5	3.5	16
8	1.3	2.4	7	3.6	16
16	1.4	2.3	5.8	3.2	14
32	2	2.1	5.8	2.9	12
geomean	1.7	2.5	9.4	3.7	17

Table 7.7: Unoptimized PR baseline execution runtime in minutes.

Locales	arabic-2005	webbase-2001	GAP-twitter	scale-26	MOLIERE_2016
2	3.1	8.6	190	186	1735
4	1.9	4.7	78	94	1834
8	2	2.8	61	56	1842
16	1.7	2.2	50	31	1590
32	1.9	1.7	31	18	1488

Table 7.8: Replication-optimized PR execution runtime in minutes.

Locales	arabic-2005	webbase-2001	GAP-twitter	scale-26	MOLIERE_2016
2	1.3	2.8	5.7	33	42
4	1.3	1.7	8.2	27	99
8	1.6	1.2	8.6	15	100
16	1.2	1	8.5	9.6	100
32	1	0.8	5.4	6.1	113

Tables 7.7 and 7.8 present the execution runtimes (in minutes) for the unoptimized and optimized codes, respectively. Unlike NAS-CG, PR adds the complication of storing records in the array of interest rather than a base type data (i.e., `int`, `real`, etc.). COPPER recognizes this scenario and will only replicate the fields that are accessed in the `forall` loop, namely, `pr_read` and `out_degree`, rather than replicating the entire record.

The memory storage overhead of the optimization for PR is 40–80%, which is much larger than what we observed for NAS-CG (6%). In both applications, the array that is the target for replication has N elements, where N is the number of rows for NAS-CG and the number of vertices for PR. As can be seen from Tables 7.1 and 7.5, the PR data sets have much larger arrays than NAS-CG, with the exception of problem size F. However, the memory storage overhead incurred by replication for PR is significantly less than what full replication of the array would incur.

Table 7.6 shows that COPPER achieves speed-ups as large as 36x over the baseline codes. Overall, the PR speed-ups are significantly smaller when compared to those achieved on NAS-CG in Table 7.2. This is largely because the PR kernel is executed fewer times than the NAS-CG kernel. Indeed, Table 7.5 shows that the number of iterations that each graph requires is at most 59 and as little as 11, versus the thousands of iterations performed within NAS-CG. As a result, the average overhead of the inspector for PR is 28% (versus 3% for NAS-CG). Additionally, the PR graphs represent real-world data (with the exception of the scale-26 graph), which exhibit much less remote data reuse compared to NAS-CG. Specifically, a given remote element within NAS-CG for problem size D is accessed on average 85 times within a single execution of the SpMV kernel. By comparison, the average remote data reuse for the PR on the webbase-2001 graph is 5. The graphs that exhibit large speed-ups, such as MOLIERE_2016, have much higher remote data reuse. This highlights the impact of data reuse and how COPPER's replication optimization can exploit it.

Furthermore, due to the highly irregular nature of the graphs, where many have degree distributions that follow a power law, the runtime performance can fluctuate as the number of locales increases due to the partitioning of the graph across the system. This can change where the elements are located, which may lead to a once heavily accessed remote element now being local. This is more significant for the unoptimized code, as the replicatiom-optimized code would have only incurred the cost of the remote access once due to replication. An example is the GAP-twitter graph, where the baseline performance is severely hindered when using two locales but improves significantly at 4 locales (Table 7.7).

Last, as mentioned earlier, PR can also be optimized via COPPER's adaptive remote prefetching optimization (Chapter 6). The speed-ups achieved via prefetching (Table 6.2) are noticeably smaller when compared to those achieved via replication (Table 7.6). This is because replication is more effective at exploiting remote data reuse, and with PR being an iterative algorithm, the overhead of the inspector can be amortized. For these reasons, replication is a better suited optimization for PR. Chapter 8 will discuss how COPPER determines which optimization to apply in a given scenario.

Moldyn

The Moldyn application, as described in Section 3.2.2, can be optimized via COPPER's selective data replication optimization, specifically the remote read performed on line 3 and the remote write on line 7 in Listing 3.7. Because the replicated data (`q`) is modified via an atomic add, COPPER can still perform replication using the technique described in Section 7.1.4. Moldyn uses real-world data sets obtained from the Research Collaboratory for Structural Bioinformatics (RCSB) Protein Data Bank [Ber+00]. Each data set represents a set of particles and their initial positions (i.e., coordinates in space), which determine the interaction between particles. These

Table 7.9: Datasets for Moldyn.

Name	**# Particles**	**# Interactions**	**Density (%)**
4B2Q	70K	163M	6.6
4BOG	90K	310M	7.7
1M8Q	75K	468M	16.7
1MVW	92K	600M	14.1

Table 7.10: Runtime speed-ups achieved by COPPER's selective data replication optimization on Moldyn for the data sets in Table 7.9 relative to the unoptimized implementation.

Locales	4B2Q	4BOG	1M8Q	1MVW
2	0.8	2.2	7.4	5.9
4	5	2.4	4.3	3.4
8	2.3	2.2	2.8	2.1
16	1.5	1.4	1.7	1.5
32	1.6	1.2	1.4	1.2
geomean	1.9	1.8	2.9	2.4

data sets are shown in Table 7.9. Each data set is executed for 10 time steps, where the underlying interactions do not change during those time steps. This means that the inspector only needs to run once.

Table 7.10 presents the Moldyn runtime speed-ups achieved by COPPER when compared to the unoptimized baseline implementation on the data sets from Table 7.9. Additionally, Tables 7.11 and 7.12 present the execution runtimes (in minutes) for the unoptimized and optimized codes, respectively. Like the PR application, the array of interest in Moldyn stores records, so COPPER will only replicate the fields that are accessed throughout the target `forall` loop.

The memory storage overhead of the optimization for Moldyn is 0–13%. The 4B2Q data set is unique when compared to the other data sets, as its particle interactions can be perfectly partitioned across two locales. As a result, there is no remote communication when using two locales, leading to zero replication. Beyond that, the overall memory storage overhead is relatively low because the size of the target array for replication is small compared to the total memory footprint of the application.

The average runtime overhead of the inspector for Moldyn is 76%, with a mini-

Table 7.11: Unoptimized Moldyn baseline execution runtime in minutes.

Locales	4B2Q	4BOG	1M8Q	1MVW
2	0.027	2.5	9.1	12.7
4	2.3	1.9	4.7	6.8
8	1.6	1.9	3.2	4.5
16	0.9	1.3	2.5	2.8
32	0.6	0.8	1.5	1.7

Table 7.12: Replication-optimized Moldyn execution runtime in minutes.

Locales	4B2Q	4BOG	1M8Q	1MVW
2	0.034	1.1	1.2	2.2
4	0.47	0.8	1.1	2
8	0.68	0.86	1.14	2.1
16	0.6	0.89	1.5	1.9
32	0.4	0.7	1.1	1.5

mum of 23% and a maximum of 89%. Moldyn incurs significantly higher overheads compared to NAS-CG (3%) and PR (28%) because of the amount of remote communication required, as well as the relatively few number of iterations performed. The particle interactions are represented as an undirected graph, where only the upper triangle of the adjacency matrix is stored. Specifically, each particle i only stores interactions with particle j if $i < j$. For a given set of locales, the locales with higher IDs will be assigned fewer interactions, leading to less remote communication. In the case of the "last" locale, it will never require remote communication. As a result, there is not as much remote communication for COPPER to exploit, which means that the inspector overhead cannot be amortized as effectively as for NAS-CG and PR. Additionally, Moldyn is only executed for 10 time steps, which is less than all data sets evaluated for NAS-CG and PR.

Despite the high inspector overhead, COPPER still provides runtime speed-ups as large as 7.4x over the unoptimized baseline code. The performance loss on the 4B2Q data set on two locales is because there is no remote communication to exploit, which will incur the full cost of the inspector. It is worth noting that the Moldyn data sets are much less sparse and smaller in size when compared to the NAS-CG and PR

data sets. Nevertheless, COPPER can still provide performance gains, highlighting the versatility of COPPER on a range of applications and data sets.

7.2.3 Summary of Results

The performance evaluation of COPPER's selective data replication optimization demonstrates that runtime speed-ups can be achieved across different applications and data sets: 273x for NAS-CG, 36x for PR and 7.4x for Moldyn. It is important to note that these speed-ups were obtained without modifying the baseline implementations. Instead, COPPER applies the replication optimization automatically, removing the burden on the programmer to detect when, where and how replication can be used. Furthermore, these speed-ups are correlated with significant reductions in runtime (e.g., 131 days to 19 hours). These results show that COPPER can successfully improve the performance and developer productivity for PGAS programs that exhibit irregular memory access patterns.

7.3 Limitations

Despite the significant improvements that COPPER can provide via selective data replication, there are some limitations. In this section I will discuss these limitations and present possible approaches to address them.

7.3.1 Multiple Candidates in the Same Loop

The selective data replication optimization as presented in this chapter will not attempt to optimize more than one candidate in the same loop. Instead, the first candidate that is detected will be optimized. Consider two candidates in the same loop, $A[B[i]]$ and $C[D[i]]$. Complications arise because B and D may be modified at different points in the program, and thus may require the inspector to be executed

at different times. This could be supported by generating separate inspector loops for each candidate access. As a result, this limitation is not a fundamental challenge to the optimization, but rather an implementation detail. However, none of the applications and kernels presented in Chapter 3 have two candidates for replication in the same loop. Additionally, if there were multiple candidates in the same loop, COPPER will apply the adaptive remote prefetching optimization (if it is valid) to the other candidates, as the candidate requirements for replication and prefetching often overlap (see Chapter 8).

7.3.2 Memory Usage Increases

An obvious side-effect of replication is an increase in the total memory used by the application. For the performance evaluations presented in Section 7.2, it was observed that COPPER's selective data replication can lead to as much as an 80% increase in the total memory footprint of an application. Using a selective approach to replication is significantly more efficient than full replication, which sometimes cannot be performed without exceeding the amount of available memory [RKS21b]. However, selective replication can still present a challenge on systems with less memory per node. To address this, COPPER could perform an even more selective approach to replication where only some of the remotely accessed elements are replicated. For example, only elements that exhibit a sufficient amount of data reuse would be replicated.

While this approach is feasible to implement, it will add non-negligible amounts of overhead to both the inspector and executor. For the inspector, it would effectively require two passes: one to count all the remote memory accesses and another to determine which ones exhibit enough data reuse. The overhead added to the executor is more subtle. When the executor encounters the candidate $A[B[i]]$ in the loop, it checks whether $B[i]$ is remote, and if so, it will access the replicated copy. However, this assumes that any remote element has been replicated. With the ap-

proach described above for reducing memory usage, not all remote elements would be replicated. To perform the proper check then, the executor will have to "query" the communication schedule for the presence of the remote element. This query will be significantly more expensive than what the executor currently does, as it requires a look-up into the underlying associative array. Furthermore, this additional overhead is incurred for each remote access in the kernel. For these reasons, it likely that any performance gains will be negated with the approach described above.

For the system used in this chapter's evaluation, which has 512 GB of memory per node, there were no instances where the unoptimized version of the code had enough memory but the replication-optimized version did not. Additionally, the current trend for high performance computing (HPC) clusters appears to show that nodes will have 512 GB or more memory per node. For example, the Zaratan cluster at the University of Maryland [1] and the Frontier system at Oak Ridge National Lab both have nodes with 512 GB of memory.

7.3.3 Overlapping Inspector and Executor

One issue with the inspector-executor approach is what Alvanos et al. [Alv+13] refer to as the "pause" issue. Specifically, the inspector's analysis is a blocking operation, as it must complete before the executor can proceed. If the inspector is very expensive, this can lead to significant overhead. Furthermore, imbalance amongst the locales in terms of how many remote accesses they issue can lead to poor resource utilization. For example, if one locale issues a majority of the remote accesses, its portion of the inspector will run longer than on other locales. This results in all other locales waiting on the "slow" locale to finish.

A possible solution would be to overlap the inspector and executor. As each locale only needs to replicate the data that it accesses remotely, it does not rely

[1] https://hpcc.umd.edu/hpcc/zaratan.html

on any coordination with the other locales in terms of replication. Because of this, once a locale has executed its portion of the inspector loop, it could proceed to the executor without waiting for all other locales. This effectively overlaps the inspector and executor. To implement this solution, the code transformations performed by COPPER would need to be heavily modified to "merge" the inspector and executor loops. Listing 7.2 presents a sketch for how these merged loops would look for the same example `forall` loop on lines 1–4 in Listing 7.1. The custom `inspectorIter` iterator from Listing 7.1 is expanded in Listing 7.2 to reveal the underlying `coforall` loop (line 9), which allows each locale to perform its portion of the inspector (lines 11–18) and executor (lines 20–23) independently from the other locales.

```
// Original forall loop
forall i in B.domain {
  C[i] = A[B[i]];
}

// Code equivalent to the output of the code transformations
if isCallPathValid(funcID) { // runtime check
  if replicationIsValid(A, B) { // compile time check
    coforall loc in Locales do on loc {
      const ref indices = B.domain.localSubdomain();
      if doInspectorLocal(A, B) {
        inspectorLocalPreamble(A);
        for i in indices {
          inspectAccess(A, B[i]);
        }
        inspectorLocalPostamble(A);
        inspectorOffLocal(A, B);
      } /* end of doInspector */

      executorLocalPreamble(A);
      forall i in indices {
        C[i] = executeAccess(A, B[i]);
      }
    } /* end of coforall */
  } /* end if replicationIsValid() */
  else {
    forall i in B.domain {
      C[i] = A[B[i]];
    }
  }
} /* end if isCallPathValid() */
else {
  forall i in B.domain {
    C[i] = A[B[i]];
  }
}
```

Listing 7.2: Example of how the inspector and executor loops could be overlapped.

7.3.4 Accurate Prediction of Optimization Profitability

Because COPPER's replication optimization requires the inspector-executor technique, certain conditions must be met to ensure that performance gains are likely. One of these conditions is that the `forall` loop being optimized must be nested in an outer loop, allowing for the overhead of the inspector to be amortized over multiple executions of the executor. While COPPER can determine whether such a loop exists, it cannot always determine how many iterations the loop will execute. For instance,

the outer loop for the PageRank application relies on a convergence threshold being satisfied, which depends on a variety of factors that are often not known until runtime. If the outer loop happened to only run a handful of times, it is possible that the overhead of the inspector would not be fully amortized, leading to a net performance loss.

To address this issue, COPPER could restrict the optimization to cases where it can statically (or dynamically) determine the minimum number of times the kernel will execute and ensure that there are no break statements to interrupt the outer iterations. Assuming a sufficient number of iterations were determined, then COPPER can proceed with the optimization. However, this would exclude applications like PageRank and others that rely on more complicated outer loops. This challenge highlights the performance-productivity trade-offs that COPPER encounters due the desire to not require user intervention to guide the optimizations.

7.4 Related Work

The inspector-executor technique has been applied in a variety of ways to perform different types of optimizations [SMC91; DK99; SCF03]. Of particular relevance to COPPER's use of the technique, Das et al. [Das+94] presented an inspector-executor optimization specifically for remote data communication on distributed-memory systems. Their code transformations are similar to what COPPER performs, in that it replicates remotely accessed data on the processor that needs it. While COPPER leverages the overall idea of this optimization, their work predates the Partitioned Global Address Space (PGAS) model, and as a result, is not directly applicable due to fundamental differences in programming model design and implementation.

Su and Yelick [SY05] developed an inspector-executor optimization for the PGAS language Titanium that targets the $A[B[i]]$ memory access pattern. However, Tita-

nium differs from the PGAS language that I use in my evaluations (Chapel), specifically in its execution model and language constructs for parallel loops. Titanium uses a Single Program Multiple Data (SPMD) model and Su and Yelick's optimization targets `foreach` loops in Titanium, which are parallel loops that execute entirely on the local processor. In contrast, Chapel uses a fork-join style of parallelism and COPPER targets `forall` loops in Chapel, which will implicitly run across multiple nodes/locales when iterating over distributed arrays/domains. This leads to COPPER employing a fundamentally different approach to static analysis that must resolve locale affinity details when such details are not explicitly available in the program.

Alvanos et al. [Alv+12; Alv+13; Alv+16] described an inspector-executor framework for the PGAS language UPC [El-+05], which also differs from Chapel. UPC uses a SPMD model and requires explicit constructs to control the processor affinity of parallel loops. Like Titanium, these differences lead to a significantly different approach to the static analysis used by COPPER. Also, the UPC optimization targets both regular and irregular remote memory accesses, where the goal is to coalesce fine-grain communication into fewer, larger messages. Since Chapel's remote cache [FB15] offers similar capabilities for regular memory accesses, COPPER only targets irregular memory accesses. Additionally, Alvanos et al. implement their inspector in a very different way by "merging" it within the executor loop. Specifically, a given iteration of the executor loop will process some future chunk of the loop iterations to determine which data to replicate. In this way, it is similar to COPPER's approach to adaptive remote prefetching (Chapter 6). On the other hand, COPPER will process the entire memory access pattern within a given `forall` loop to determine what to replicate, and then perform the executor. A pipelined approach to the inspector and executor could improve performance if the communication and computation can be overlapped effectively. However, that will require more extensive analysis to determine how many loop iterations ahead to process via the inspector.

Kayraklioglu et al. presented Locality-Aware Productive Prefetching Support for PGAS (LAPPS), which specifically targets Chapel programs that run on distributed-memory systems [KFE18]. LAPPS was briefly described in Section 6.5 in the context of prefetching. However, the operations performed by LAPPS is more similar to replication, as it performs a blocking operation to gather remote data. Rather than using runtime information to determine which data to gather, LAPPS relies on pre-defined access patterns (e.g., row-wise, stencil, all-to-all) that are specified by the programmer. This is in contrast to COPPER, which uses the inspector-executor technique to selectively replicate remotely accessed data in a loop and requires no programmer intervention.

Chapter 8

Interoperability Between Optimizations

The goal of this dissertation is to improve performance of PGAS programs with irregular memory accesses patterns without negatively impacting developer productivity. In Chapters 5–7, I presented three different optimizations that the COPPER framework can apply automatically to PGAS programs. Individually, these optimizations have been shown to significantly improve performance, and they do not require any intervention on the part of the programmer.

However, it would be detrimental to productivity to provide programmers with three separate options when optimizing their program, even if those optimizations were applied automatically. In this way, users would either need to be aware of what each optimization does, or they would need to try each of them individually or in some combination to see if it provides a benefit. Furthermore, there are situations where a given irregular memory access pattern is a candidate for more than one of COPPER's optimizations. This complicates productivity, as users would need to be even more informed about what each optimization does. To address these issues, COPPER provides a single compiler flag that users can turn on for their program, where

this flag controls all three optimizations. No further action from the programmer is needed to utilize COPPER.

In this chapter, I will describe the overall approach to how COPPER chooses which optimizations to apply in a program. Additionally, I will present a performance evaluation of applications where multiple optimizations are applied together rather than in isolation.

8.1 Approach

The COPPER framework begins its process by considering each `forall` loop in a Chapel program during the normalize pass in Chapel's compiler (Section 4.2.1). It will then attempt to find candidate irregular memory accesses within the `forall`, where each of the three optimizations (aggregation, prefetching and replication) are considered sequentially. In this way, the order in which the optimizations are applied is crucial to how COPPER prioritizes the optimizations. The ordering is as follows: replication, prefetching and aggregation.

8.1.1 Single Candidate for Multiple Optimizations

Aggregation targets irregular writes and the other two optimizations specifically target reads, so aggregation candidates will not be considered for replication or prefetching. As a result, the order in which aggregation is applied with respect to the other optimizations is not important. However, applying replication before prefetching is important. As we saw in Section 7.2.2, a given application that could be optimized by prefetching or replication (e.g., PageRank) will most likely benefit significantly more from replication. This is because replication is much better at exploiting remote data reuse than prefetching. However, replication is a more restrictive optimization and cannot always be applied.

As discussed in Section 7.1.2, there are several scenarios where replication cannot be applied. These cases may not be known to COPPER until after the normalize pass. To address this issue, COPPER uses both compile time and runtime checks that will "revert" to the unoptimized `forall` if replication cannot be performed. Because these unoptimized `forall` loops were created by the replication optimization during the normalize pass, they are also processed by COPPER. Therefore, the prefetching optimization will be applied to those loop clones if prefetching is valid. As a result, if the `forall` cannot be optimized via replication then the `forall` loop that does end up executing will be optimized for prefetching. This effectively applies prefetching as a "back up" to replication. If prefetching was applied first, then the irregular memory access candidate would not be seen by COPPER when considering replication, as it would have been replaced by the compiler primitive used by the prefetching optimization (Section 4.2.2). Listing 8.1 shows an example of how a single `forall` loop with a candidate for both replication and prefetching is transformed. The `forall` loops on lines 23–34 and lines 39–50 are the unoptimized clones created by replication, which are then recognized by COPPER as candidates for prefetching.

8.1.2 Multiple Candidates in the Same Loop

For each optimization COPPER has different ways of supporting multiple memory access candidates in the same loop. As described in Section 5.1, multiple aggregation candidates can be optimized in the same `forall` loop if they use different target arrays, or if their original operations are order independent with respect to each other. For prefetching, multiple candidates can be optimized without any constraints, but the overall accuracy of the adaptive heuristic may be impacted due to the candidates sharing remote cache counters (Section 6.4). Finally for replication, only one candidate can be optimized in a loop (currently) due to the complications of dealing with multiple inspectors, as described in Section 7.3.

```
// Original forall loop
forall i in B.domain {
  C[i] = A[B[i]];
}

// Code equivalent to the output of the code transformations
if isCallPathValid(funcID) {
  if replicationIsValid(A, B) {
    if doInspector(A, B) {
      inspectorPreamble(A);
      forall i in inspectorIter(B.domain) {
        inspectAccess(A, B[i]);
      }
      inspectorPostamble(A); inspectorOff(A, B);
    } /* end doInspector() */
    executorPreamble(A);
    forall i in B.domain {
      C[i] = executeAccess(A, B[i]);
    }
  } /* end if replicationIsValid() */
  else {
    // prefefetch variables
    forall i in B.domain with (var d = 8, var cnt = 0) {
      if isPrefetchSupported(B.domain, A, B) {
        if cnt < sampleIters { cnt += 1; }
        else {
          adjustPrefetchDistance(d); cnt = 0;
        }
        if (i + (d * stride)) <= lastIndex {
          prefetch(A[B[i+(d*stride)]]);
        }
      }
      C[i] = A[B[i]];
    }
  }
} /* end if isCallPathValid() */
else {
  // prefefetch variables
  forall i in B.domain with (var d = 8, var cnt = 0) {
    if isPrefetchSupported(B.domain, A, B) {
      if cnt < sampleIters { cnt += 1; }
      else {
        adjustPrefetchDistance(d); cnt = 0;
      }
      if (i + (d * stride)) <= lastIndex {
        prefetch(A[B[i+(d*stride)]]);
      }
    }
    C[i] = A[B[i]];
  }
}
```

Listing 8.1: Simplified example of how COPPER applies replication and prefetching to the same candidate in a `forall` loop.

Beyond applying the same optimization to multiple candidates in a loop, COPPER can also apply different optimizations to separate candidates in the same loop. An example is the Single Source Shortest Path (SSSP) graph analytic described in Section 3.1.2, which contains candidates for both prefetching and aggregation within the same `forall`. COPPER was designed so that optimizations will not conflict with each other when applied to distinct candidates in the same loop. Specifically, the code transformations applied to the `forall` loop by each optimization will not cancel each other out or cause other issues. This feature somewhat addresses the limitation that multiple replication candidates in the same loop cannot be optimized. COPPER will optimize the first candidate via replication and then remaining candidates can often be optimized via prefetching, since the two optimizations have several overlapping requirements.

8.2 Performance Evaluation

In this section, I present a brief performance evaluation of two applications that showcase COPPER's ability to provide interoperability between its optimizations. The goal is to further demonstrate that COPPER can provide significant performance gains for irregular PGAS programs without burdening the user by requiring manual modifications to their code. Specifically, I show that users do not need to be aware of which optimizations COPPER can apply, and as a result, do not need to be aware of the order that the optimizations are applied.

8.2.1 Experimental Setup

The experiments presented in this section were executed on the same 32-node FDR Infiniband cluster from Sections 5–7, and use the same Chapel installation and configuration settings. The applications and kernels evaluated are SSSP (Section 3.1.2) and

Table 8.1: Runtime speed-ups achieved by COPPER's remote data aggregation and adaptive remote prefetching optimizations on SSSP for the graphs in Table 5.1 relative to the unoptimized implementation.

Locales	scale-24	scale-25	scale-26
2	1.4	1.4	1.4
4	1.7	1.9	1.8
8	1.8	1.8	1.9
16	1.7	1.8	1.8
32	1.4	1.5	1.7
geomean	1.6	1.7	1.7

an example workflow for graph analytics that performs multiple kernels in a single program. Both applications use the scale-24, scale-25 and scale-26 graphs as described in Table 5.1. The primary metrics reported for each experiment are application runtime and the runtime speed-up achieved by COPPER relative to the unoptimized baseline implementation. For each experiment, multiple trials were performed and the average of these trials is presented. The variation in runtime across all trials did not exceed 5%.

8.2.2 Results

Single Source Shortest Path

Section 3.1.2 presented the Single Source Shortest Path (SSSP) graph analytic and Listing 3.2 described the baseline Chapel implementation of the main kernel of the SSSP algorithm. The remote write on line 15 can be aggregated and line 10 can be prefetched via COPPER. Sections 5.2.2 and 6.3.2 evaluated SSSP when aggregation and prefetching were applied in isolation, respectively. Table 8.1 presents the SSSP runtime speed-ups achieved by COPPER when applying both optimizations compared to the unoptimized baseline implementation on the scale 24, 25 and 26 graphs from Table 5.1. Additionally, Tables 8.2 and 8.3 present the execution runtimes (in minutes) for the unoptimized and optimized codes, respectively.

Table 8.2: Unoptimized SSSP baseline execution runtime in minutes.

Locales	scale-24	scale-25	scale-26
2	28	64	149
4	13	29	65
8	8	16	37
16	4	9	20
32	2	4.3	10

Table 8.3: Aggregation- and prefetch-optimized SSSP execution runtime in minutes.

Locales	scale-24	scale-25	scale-26
2	20	47	109
4	7.6	15	35
8	4.3	8.7	20
16	2.3	4.8	11
32	1.4	2.8	6.1

When comparing the speed-ups from Table 8.1 to those achieved by aggregation and prefetching in isolation (Tables 5.5 and 6.5, respectively), it can be seen that applying both optimizations together provides better performance than the individual optimizations in all but two cases. These two cases are for utilizing two locales, which was shown to be generally harmful for prefetching due to the overhead of the prefetch bounds check. Other than those cases, applying both optimizations can improve performance over prefetching and aggregation alone by as much as 1.8x and 1.3x, respectively. When compared to the unoptimized baseline, Table 8.1 shows that COPPER can improve performance by as much as 1.9x These results emphasize that the performance for irregular PGAS programs can be significantly improved via techniques such as aggregation and prefetching. Furthermore, they show COPPER's versatility in targeting different memory access patterns within the same loop.

Graph Analytic Workflow

To demonstrate COPPER's ability to apply all three optimizations within the scope of a single program, I implemented a graph analytic workflow that is designed to be representative of a real-world use case. To this end, the workflow performs the following kernels: (1) graph construction from a raw edge list (Section 3.3.2), (2) PageRank (Section 3.1.4) and (3) SSSP (Section 3.1.2). The graph construction kernel can be optimized via aggregation, specifically lines 8, 9 and 39 in Listing 3.9. Note that line 38 cannot be optimized via aggregation since line 39 accesses the same target

Table 8.4: Runtime speed-ups achieved by COPPER on the graph analytic workflow for the graphs in Table 5.1 relative to the unoptimized implementation.

Locales	scale-24	scale-25	scale-26
2	2.1	2.4	1.9
4	2.3	2.3	2.3
8	2.3	2.3	2.5
16	2.35	2.3	2.6
32	2.5	2.5	2.7
geomean	2.3	2.4	2.4

array and the two operations are not order independent with respect to each other. PageRank can be optimized by both prefetching and replication, but as described in Section 8.1.1, replication will be given priority. Finally, SSSP is optimized by both prefetching and aggregation, as shown above.

The workflow begins by generating a list of raw edges (i.e., tuples) that are transformed into a distributed graph data structure. This kernel specifically generates edges according to the Graph500 benchmark specification [Bad+22] and the scale-24, scale-25 and scale-26 graphs as described in Table 5.1 will be evaluated. Once the graph is constructed, it is passed into the PageRank (PR) kernel to compute the rankings of each vertex. Using the PR results, the top three ranked vertices will be passed into the SSSP kernel, where each of those vertices will serve as a root vertex for SSSP. In this way, the SSSP kernel consists of three consecutive executions of the SSSP algorithm. This workflow represents an exploratory data analysis, where an analyst is interested in the shortest paths starting from the three most "important" vertices in the graph.

Table 8.4 presents the total runtime speed-ups achieved by COPPER compared to the unoptimized implementation of the workflow. As can be seen, speed-ups as large as 2.7x can be achieved through the use of COPPER's aggregation, prefetching and replication optimizations. Tables 8.5 and 8.6 present the total execution runtimes (in minutes) for the unoptimized and optimized codes, respectively. While these results are based on the total execution time of the workflow, each of the three individual

Table 8.5: Unoptimized graph analytic workflow baseline execution runtime in minutes.

Locales	scale-24	scale-25	scale-26
2	164	393	823
4	76	157	319
8	46	96	205
16	30	64	128
32	22	46	97

Table 8.6: Optimized graph analytic workflow execution runtime in minutes.

Locales	scale-24	scale-25	scale-26
2	78	163	424
4	33	69	128
8	20	41	81
16	13	28	49
32	8.8	18	36

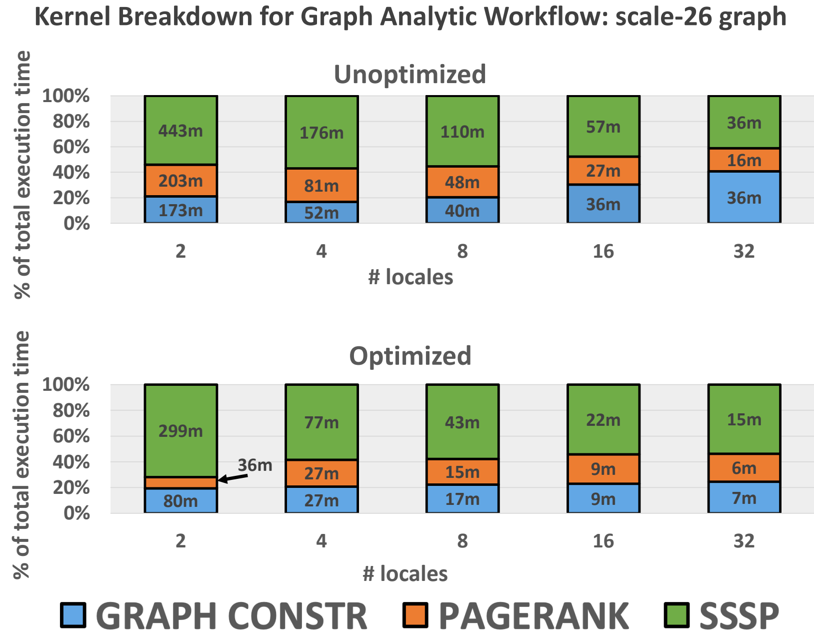

Figure 8.1: Percentage of the total execution time devoted to each of the three kernels in the graph analytic workflow on the scale-26 graph from Table 5.1. Values within the bars represent the execution time (in minutes) of each kernel.

kernels in the workflow contributes different amounts to the total execution time and COPPER provides different degrees of improvement for each kernel.

Figure 8.1 shows the percentage of time devoted to each kernel in the workflow on the scale-26 graph for both the unoptimized and optimized implementations, as well as the individual runtimes of each kernel in minutes. For both the unoptimized and optimized codes, the SSSP kernel contributes the most to the total runtime, as it is performed three times (once for each of the top three vertices as determined by PR). However, COPPER improves the SSSP kernel by as much as 2.6x over the baseline. The next largest bottleneck for the unoptimized version is the PR kernel, which takes up more time than graph construction up until 16 locales. At 16 locales and beyond, the graph construction kernel stops scaling due to the large amount of remote communication required. In contrast, the PR kernel achieves good scalability all the way up to 32 locales. For the optimized code, the PR kernel is the least expensive kernel due to COPPER's replication optimization providing as much as a 5.6x speed-up over the baseline. For the graph construction kernel, COPPER reduces the runtime by as much as 5.3x via aggregation and provides scalability up to 32 locales.

Chapter 9

Performance and Productivity Portability Across Systems

Performance portability refers to the ability to achieve good performance from an application across a variety of different environments, such as the system being used [ETS14]. Performance portability is closely related to developer productivity, since if a programmer must spend significant effort to achieve good performance from their codes when migrating to a different system, that negatively impacts their overall productivity. With respect to automatic compiler optimizations, such as those presented in this dissertation (Chapters 4–7), their productivity advantages can be undermined by a lack of performance portability across different systems and architectures.

In this chapter, I present several performance evaluations of the COPPER framework across different distributed-memory systems. These systems have different CPUs, number of cores, amounts of memory and network interconnects. The results from these experiments demonstrate that COPPER can achieve significant performance gains on irregular PGAS programs across a variety of different systems. Therefore, the performance and productivity advantages provided by COPPER are portable.

9.1 Description of Systems

In addition to the 32-node FDR Infiniband cluster that is used in the performance evaluations in Chapters 5–8, I evaluated COPPER on two other distributed-memory systems. All three systems are described in Table 9.1. FDR-IB refers to the system used in the prior evaluations. The FDR Infiniband network provides 56 Gb/s of bandwidth and a latency of 0.7μs. All 32 nodes are connected to an Infiniband switch.

HDR-IB is the Zaratan cluster hosted at the University of Maryland[1], where each node has 128 cores and 512 GB of memory. However, as indicated in Table 9.1 the number of cores per node utilized in the evaluation is 16 and the memory per node is 64 GB. The GASNet communication library used by Chapel on Infiniband systems serializes the initiation of communication across threads. As a result, Infiniband systems with high core counts exhibit performance issues when running Chapel programs that issue many small messages, which characterizes the target applications for COPPER. I found that using 16 cores per node provides the best performance across the applications evaluated. The memory per node was adjusted to match the reduction in cores. The HDR Infiniband network provides 100 Gb/s of bandwidth and a latency of less than 0.6μs. As this system consists of hundreds of nodes, the network topology follows a fat-tree.

The Cray XC system is hosted by Hewlett Packard Enterprise (HPE) and access was granted by the Chapel team. This system uses the Aries interconnect in a Dragonfly topology [Kim+08], which provides significantly better performance for fine-grain communication compared to Infiniband, especially within the context of Chapel programs. Chapel's runtime has been optimized to better utilize the Aries interconnect for fine-grain communication compared to Infiniband. For example, communication initiation is not serialized across threads on the Aries interconnect as it

[1]https://hpcc.umd.edu/hpcc/zaratan.html

Table 9.1: Systems Used in Performance Evaluation

Name	CPUs	# Cores/node	Memory/node	Interconnect
FDR-IB	Intel Xeon E5-2650	20	512 GB	FDR Infiniband
HDR-IB	AMD EPYC 7763	16	64 GB	HDR Infiniband
Cray XC	Intel Xeon E5-2699	44	128 GB	Cray Aries

is for Infiniband.

9.2 Experiments and Results

For each of the three systems described in Table 9.1, I executed a subset of the applications from the prior evaluations in Chapters 5–7. These applications are Breadth First Search (BFS), Single Source Shortest Path (SSSP), PageRank (PR) and NAS-CG. BFS and SSSP can be optimized via remote data aggregation. Additionally, SSSP has a candidate for adaptive remote prefetching. PR and NAS-CG can both be optimized by either prefetching or replication, but I will only focus on evaluating PR with prefetching and NAS-CG with replication. The same data sets used in the prior evaluations will be used in these experiments, and they will be referred to when presenting each application's results.

9.2.1 Remote Data Aggregation: BFS and SSSP

Figure 9.1 presents the runtime speed-ups achieved when applying COPPER's remote data aggregation optimization to BFS and SSSP for the scale-28 and scale-26 graphs, respectively. These results are representative of what is observed on the other Graph500 graphs from Table 5.1. From the BFS results, it can be seen that speed-ups as large as 6.3x on the FDR-IB system, 11.2x on the HDR-IB system and 5.3x on the Cray XC system can be achieved. The scale-28 graph exceeded the available memory across two locales for the HDR-IB and Cray XC systems for both the baseline and aggregation implementations. Across the scale-26, scale-27 and scale-28 graphs for all

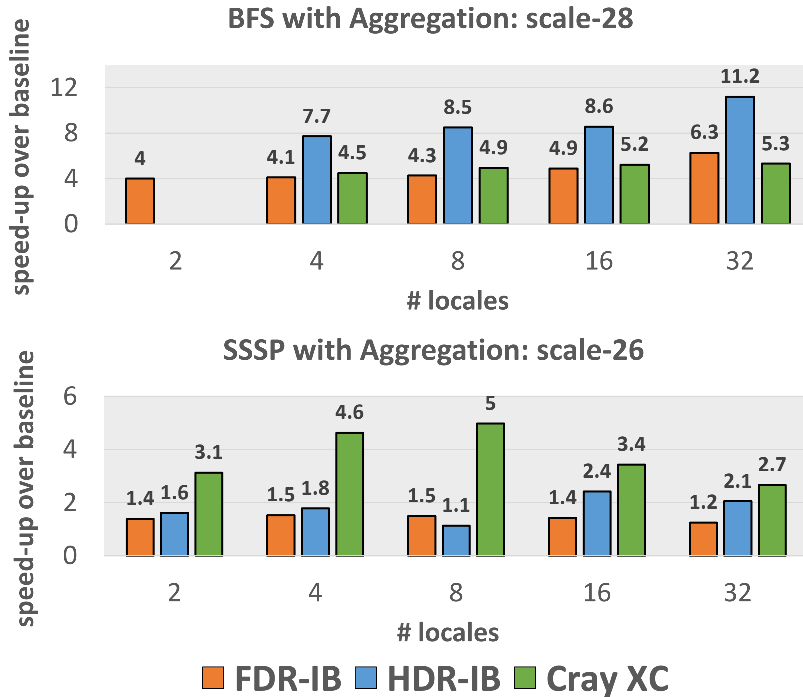

Figure 9.1: Runtime speed-ups achieved by COPPER's remote data aggregation optimization for BFS on the scale-28 graph and SSSP on the scale-26 graph from Table 5.1. Speed-ups are relative to the unoptimized implementation and higher bars represent better performance. The HDR-IB and Cray XC systems did not have enough memory on two locales for the scale-28 graph for BFS.

locale counts, the geomean speed-ups for the FDR-IB, HDR-IB and Cray XC systems are 4.3x, 7.9x and 4.3x, respectively.

The runtime speed-ups achieved by aggregation for SSSP on the scale-26 graph are as large as 1.5x on the FDR-IB system, 2.4x on the HDR-IB system and 5x on the Cray XC system. Across the scale-24, scale-25 and scale-26 graphs for all locale counts, the geomean speed-ups for the FDR-IB, HDR-IB and Cray XC systems are 1.4x, 2x and 3.5x, respectively. These speed-ups are lower than those achieved on BFS because the SSSP kernel contains another irregular memory access pattern that cannot be aggregated. However, this access pattern can be prefetched via COPPER (see below for results when applying both optimizations to SSSP).

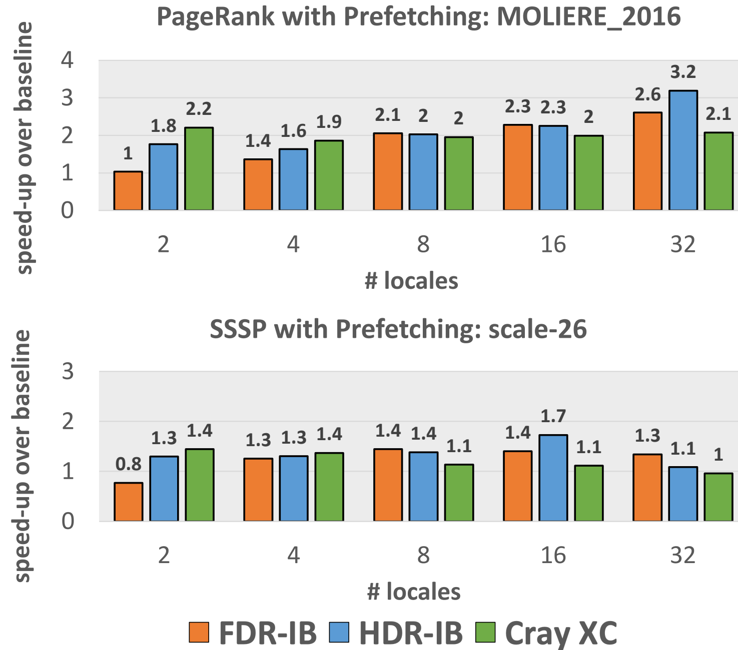

Figure 9.2: Runtime speed-ups achieved by COPPER's adaptive remote prefetching optimization for PageRank on the MOLIERE_2016 graph from Table 6.1 and SSSP on the scale-26 graph from Table 5.1. Speed-ups are relative to the unoptimized implementation and higher bars represent better performance.

9.2.2 Adaptive Remote Prefetching: SSSP and PR

Figure 9.2 shows that the runtime speed-ups achieved COPPER's adaptive remote prefetching optimization for PR on the MOLIERE_2016 graph are as large as 2.6x on the FDR-IB system, 3.2x on the HDR-IB system and 2.2x on the Cray XC system. A late prefetch tolerance of 10%, adjustment interval frequency of 8 and an iteration sample ratio of 0.01 were used on each system. Across all the real-world graphs and the Graph500 scale-26 graph from Table 6.1, the geomean speed-ups for the FDR-IB, HDR-IB and Cray XC systems are 1.4x, 1.4x and 1.7x, respectively. However, as described in Section 6.3.2 for the FDR-IB system, some of the real-world graphs are not suitable for the adaptive remote prefetching optimization because they do not induce enough remote communication, namely the arabic-2005, sk-2005 and webbase-2001 graphs. No performance losses are incurred for these graphs when processed with the inspector to determine whether there is enough communication to justify

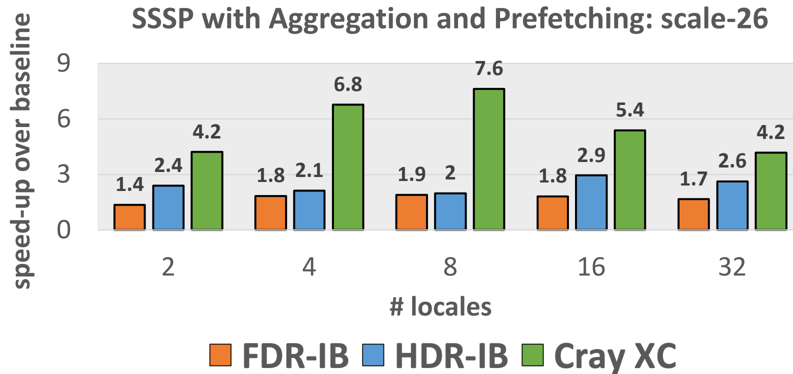

Figure 9.3: Runtime speed-ups achieved by COPPER when applying both remote data aggregation and adaptive remote prefetching to SSSP on the scale-26 graph from Table 5.1. Speed-ups are relative to the unoptimized implementation and higher bars represent better performance.

prefetching, as described in Section 6.3.2. From the SSSP results in Figure 9.2, it can be seen that speed-ups as large as 1.4x on the FDR-IB system, 1.7x on the HDR-IB system and 1.4x on the Cray XC system are achieved. Across the scale-24, scale-25 and scale-26 graphs for all locale counts, the geomean speed-ups for the FDR-IB, HDR-IB and Cray XC systems are 1.2x, 1.3x and 1.2x, respectively. Additionally, Figure 9.3 presents the results across the different systems for SSSP when both aggregation and prefetching are applied. As observed in Section 8.2.2 for the FDR-IB system, the other two systems achieve better performance via both optimizations compared to the optimizations in isolation.

To further demonstrate the advantages of using an adaptive approach to prefetching, I monitored how many prefetches were issued with a given distance across all of the tasks for a given execution of SSSP and PR on the scale-26 graph using 8 locales. This same experiment was conducted in Section 6.3 but only on the FDR-IB system. Figure 9.4 shows the results of this experiment on all three systems from Table 9.1. The results from Section 6.3 showed that different applications can exhibit different prefetching usage patterns, even when using the same input data. However, the results in Figure 9.4 extend that observation by showing that the same application and

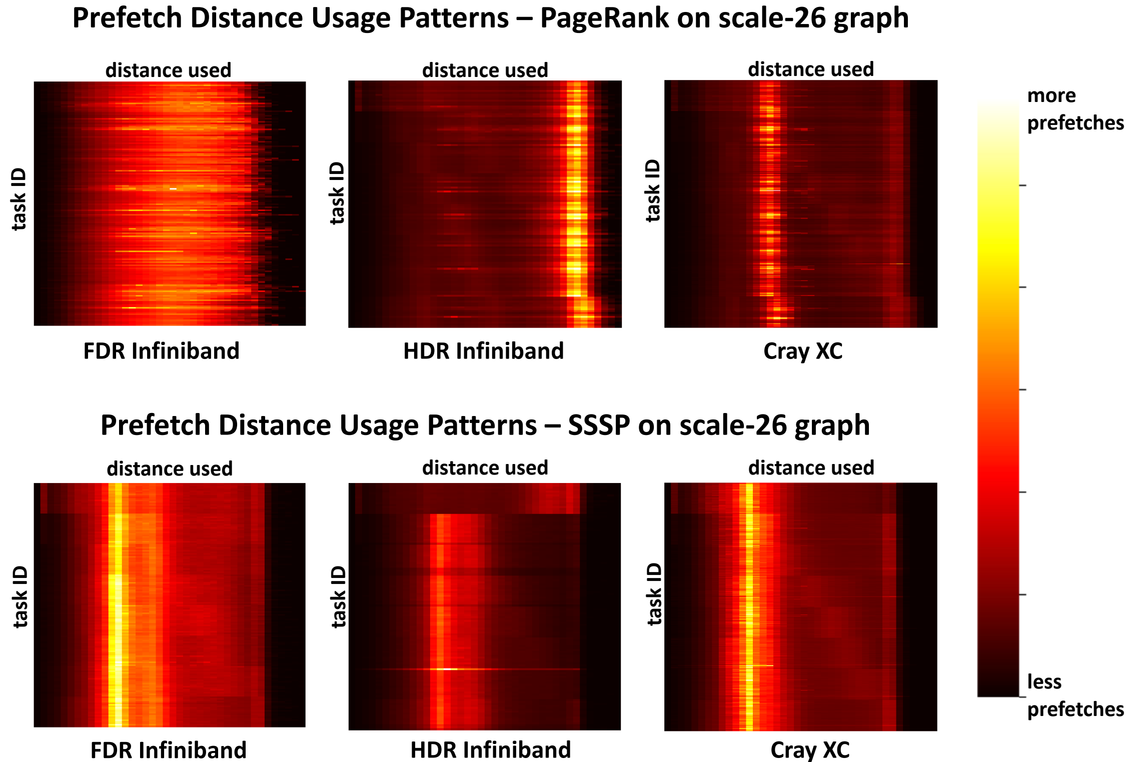

Figure 9.4: Heatmaps that show each task's prefetch distance usage pattern when running SSSP and PageRank on the scale-26 graph with 8 locales across the three systems from Table 9.1. The horizontal rows in a map represent individual tasks and the vertical columns represent prefetch distances used, which range from 1 to 40. Values in a map represent how many prefetches were issued using a given distance, where lighter colors correspond to more prefetches and darker colors correspond to fewer prefetches.

input data can exhibit very different prefetching usage patterns across different systems. These observations underscore the importance of using an adaptive approach.

Last, I performed the same sensitivity experiment presented in Section 6.3.2 for the late prefetch tolerance, adjustment interval and iteration sample ratio thresholds on the HDR-IB and Cray XC system. For a given threshold evaluated, the other thresholds were set to their "default" values (10% for late prefetch tolerance, 8 for the adjustment interval and 0.01 for the iteration sample ratio). The HDR-IB system is a shared system used by many users where multiple jobs can be executing on the same nodes, which can complicate gathering accurate results for sensitivity studies.

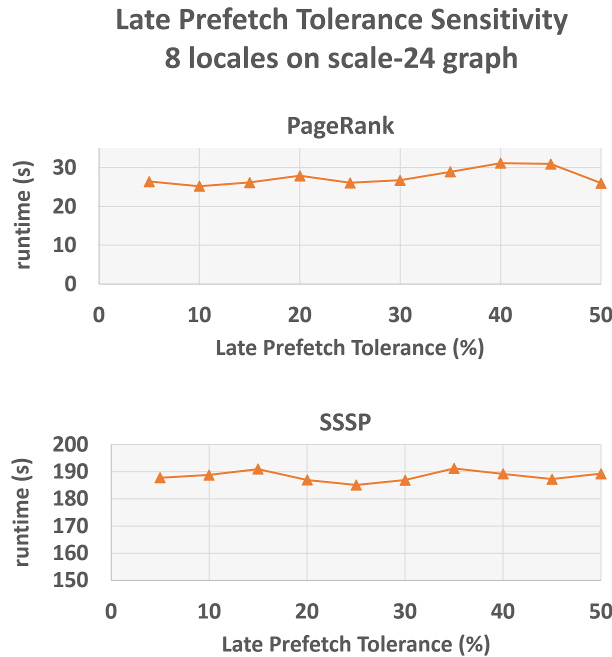

Figure 9.5: Runtime sensitivity (in seconds) when varying the late prefetch tolerance (%) on the Cray XC for PageRank and SSSP on 8 locales and the scale-24 graph from Table 6.1. For PageRank, only 5 iterations were executed.

Despite this issue, experiments on the HDR-IB system generally showed similar sensitivity behavior to the FDR-IB system, which is expected as both systems use the Infiniband interconnect. The main difference between the FDR and HDR networks is the available bandwidth, which does not significantly impact fine-grain prefetches that are latency-bound.

When compared to the Infiniband-based systems, the Cray XC's Aries interconnect is significantly different, as it is optimized for fine-grain communication. Figures 9.5, 9.6 and 9.7 present the runtime sensitivity for the three thresholds on the Cray XC for PR and SSSP when using 8 locales. For all three thresholds, the Cray XC generally exhibits more sensitivity compared to the FDR-IB system (Figures 6.1, 6.2 and 6.3). Despite this feature, the values of 10%, 8 and 0.01 used for the late prefetch tolerance, adjustment interval and iteration sample ratio, respectively, still provide good performance even on the Cray XC. This is demonstrated by the positive runtime speed-ups shown in Figure 9.2. However, auto-tuning approaches to determine these

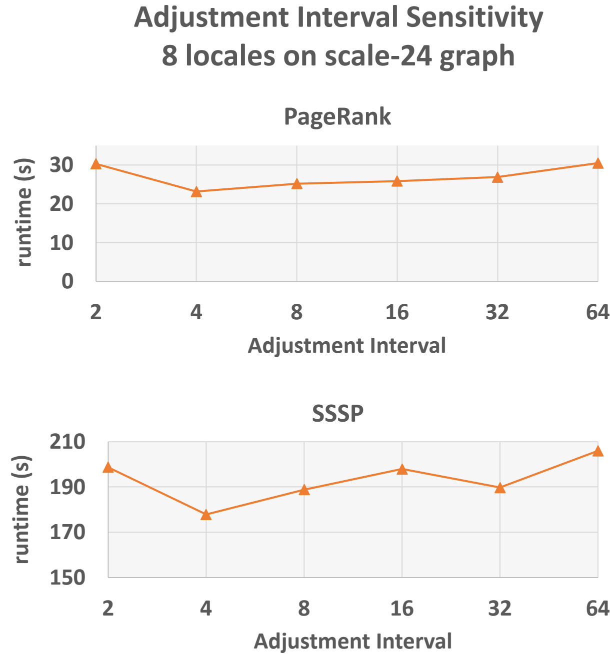

Figure 9.6: Runtime sensitivity (in seconds) when varying the adjustment interval on the Cray XC for PageRank and SSSP on 8 locales and the scale-24 graph from Table 6.1. For PageRank, only 5 iterations were executed.

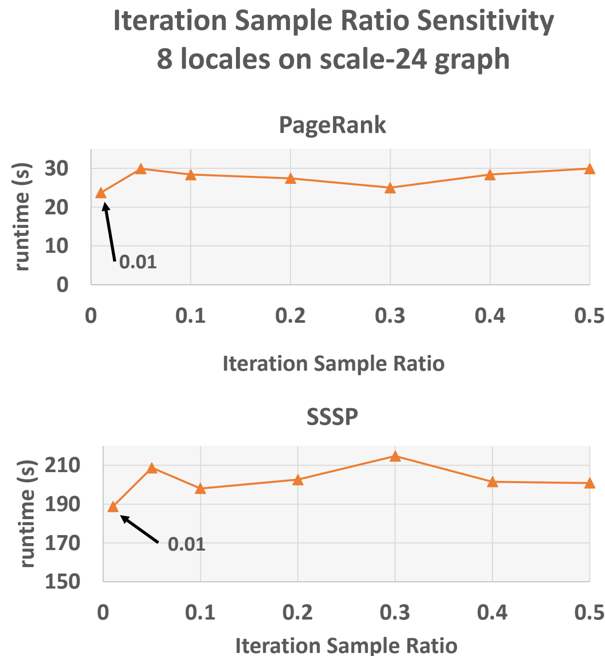

Figure 9.7: Runtime sensitivity (in seconds) when varying the iteration sample ratio on the Cray XC for PageRank and SSSP on 8 locales and the scale-24 graph from Table 6.1. For PageRank, only 5 iterations were executed.

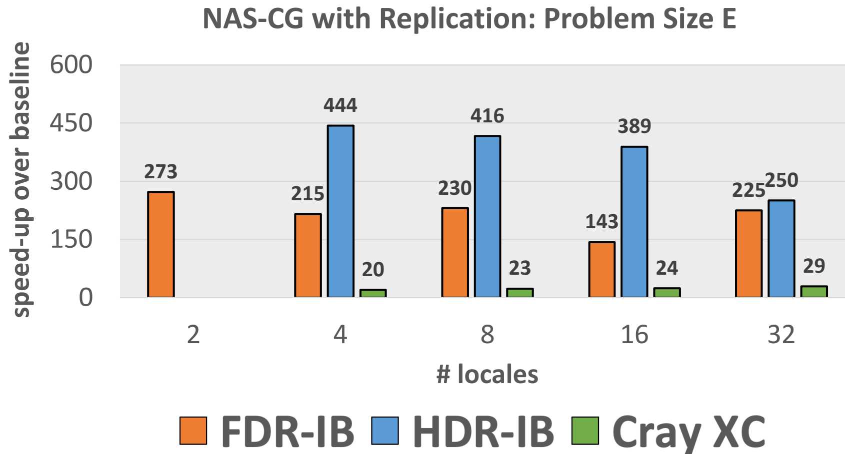

Figure 9.8: Runtime speed-ups achieved by COPPER when applying selective data replication to NAS-CG on problem size E, as described in Table 7.1. Speed-ups are relative to the unoptimized implementation and higher bars represent better performance. The HDR-IB and Cray XC systems did not have enough memory on two locales for problem size under the baseline implementation.

values could still be beneficial, as described in Section 6.4.

9.2.3 Selective Data Replication: NAS-CG

Figure 9.8 presents the results of applying COPPER's selective data replication optimization to the NAS-CG benchmark for problem size E, as described in Table 7.1. These results show the runtime speed-ups achieved by COPPER relative to the unoptimized baseline for each of the three systems in Table 9.1. Note that problem size E requires too much memory to be executed on the HDR-IB and Cray XC systems for both the unoptimized and optimized implementations. Runtime speed-ups as large as 273x, 444x and 29x can be achieved on the FDR-IB, HDR-IB and Cray XC systems, respectively. The speed-ups are significant on the FDR-IB and HDR-IB systems due to the poor performance of Infiniband for small fine-grain messages, and the fact that NAS-CG exhibits significant amounts of remote communication and data reuse. As a result, the baseline performance for NAS-CG on those systems is severely impacted. On the other hand, the Cray XC system achieves relatively small

speed-ups by comparison. This is due to the Aries interconnect being specifically optimized for fine-grain communication, which leads to much better baseline performance when compared to Infiniband. Despite this hardware advantage, COPPER still provides large speed-ups on the Cray XC.

9.3 Summary of Results

The results from Section 9.2 show that COPPER achieves significant performance improvements for irregular PGAS programs across three different distributed-memory systems. This underscores the importance of optimization techniques such as aggregation, prefetching and replication for fine-grain communication, which negatively impact performance on a variety of different hardware platforms. Even when considering the Aries interconnect on the Cray XC system, which is optimized specifically for fine-grain communication, COPPER is able to provide runtime speed-ups as large as 29x.

Additionally, COPPER's aggregation and replication optimizations can enable better resource utilization on systems with the Infiniband interconnect when executing applications using GASNet, which is a common choice for the communication library used for PGAS languages. As noted earlier, GASNet serializes communication initiation on Infiniband systems, leading to resource contention on platforms with high core counts. However, aggregation and replication reduce the total number of messages sent, which will allow for such systems to utilize more cores without increasing the contention to send messages.

These results demonstrate that the goal of this dissertation, which is to improve both performance and developer productivity, is further achieved across multiple platforms. As these results are achievable on each system without any fine-tuning or manual adjustments to the application or COPPER's optimizations, developer productivity

is maintained in a portable manner.

Chapter 10

Conclusions and Future Work

The goal of this dissertation was to show that through the design and implementation of a framework for automatic runtime optimizations, both good performance and developer productivity can be achieved for PGAS programs that exhibit irregular memory access patterns. To achieve this goal, I presented the COPPER framework that provides compiler optimizations to automatically perform remote data aggregation, adaptive remote prefetching and selective data replication. Additionally, COPPER automates the process of deciding which optimization should be applied in any given scenario and provides the necessary interoperability that allows multiple optimizations to be used in the context of a single program. To demonstrate the capabilities of COPPER, I performed extensive performance evaluations across a range of applications and kernels that exhibit irregular memory access patterns. These evaluations were also conducted on different hardware platforms, showcasing the portability of COPPER. Specifically, COPPER achieves runtime speedups of 1.08 – 87x for remote data aggregation, 0.78 – 3.2x for adaptive remote prefetching and 1.2 – 444x for selective data replication. The takeaway of this dissertation is that COPPER can enable programmers to more effectively use the PGAS model for applications with irregular memory access patterns.

Listed below are different directions that can be explored in the future to extend and improve the work presented in this dissertation.

Determining Profitability of Optimizations For any compiler optimization, not degrading performance is almost as important as improving performance. Given that such optimizations are applied automatically, if performance is severely degraded as a result of the optimization, it places an additional burden on the user to toggle the optimization on and off. For these reasons, a future direction for the work presented in this dissertation is to provide more advanced analysis to determine the profitability of the optimizations.

In the case of remote data aggregation (Chapter 5) and adaptive remote prefetching (Chapter 6), performance loss can occur when there is not enough remote communication performed within the optimized loop. When this happens, the overhead of storing messages in the buffers for aggregation or performing bounds checking for prefetching exceeds the benefit of the optimizations. One solution would be to employ static and dynamic analysis to determine whether there will be enough remote communication. This can be done via an inspector if the overhead could be amortized over multiple iterations of the loop, which was described in Chapter 6. A simpler, and more flexible, solution would be to analyze the loop bounds and see whether there are enough iterations, where "enough" would need to be quantified in some way. A sufficient number of iterations can suggest the possibility of enough remote communication for the optimizations. This approach may not be accurate in determining the actual amount of remote communication, but it would be rather lightweight and incur low overhead.

Addressing selective data replication (Chapter 7) is more complicated. In this case, performance loss can occur when the outer loop does not iterate enough times to fully amortize the cost of the inspector. Similar loop bounds analysis could be

performed as described above, but it is less likely to be successful as the outer loops in the applications studied in this dissertation are often `while` loops that iterate until some runtime convergence criterion is met.

In the approaches described above, a *cost model* would further improve the process of determining an optimization's profitability. A given set of code transformations that an optimization performs can be associated with an estimated *cost*, such as the cost of bounds checking or inspecting a memory access pattern. There would also be estimated *savings* provided by the optimization, such as reducing the number of small messages. These costs and savings can vary across different hardware platforms due to differences in network interconnects and node architecture. By integrating a cost model for a given platform with the static analysis and code transformations that COPPER performs, it may be possible to make more informed decisions about whether an optimization should or should not be applied (e.g., only apply the optimization if the savings outweigh the costs).

Formal Verification of Optimizations When designing and implementing the optimizations for COPPER, careful attention was given to ensuring that the optimizations, when applied, would not produce incorrect program behavior. Specifically, the results yielded by the optimized programs should match the results produced by the unoptimized program. Regardless of how successful the optimization is at improving performance, if it produces the incorrect behavior then it is a failed optimization.

However, while best efforts were made to maintain correct behavior, the code transformations applied by COPPER's optimizations are not formally verified. *Formal verification* is the process of mathematically proving that a program produces the intended behavior on all inputs, where the intended behavior is specified by some model of the program or directly from the semantics of the programming language itself. Formal methods have been used for a variety of application domains, ranging

from avionics software [Sou+09] and cyber-physical systems [MP16] to quantum computing [Hie+19]. Most relevant to this dissertation is CompCert, a formally verified optimizing compiler [Ler+16]. CompCert provides a verified C compiler that achieves performance comparable to `gcc` with optimization levels `-O1` and `-O3`.

While prior works have shown that formal methods can be used for compiler optimizations, it would require an enormous effort in the case of COPPER and the optimizations presented in this dissertation. The optimizations that COPPER performs do not drastically change the original program from a source code perspective, but rather they focus on modifying the underlying data movement. In this way, verification of the optimizations would not only need to be considered with respect to the semantics of the programming language, but also the complicated PGAS runtime system (i.e., Chapel's remote cache, the GASNet communication library, etc.). To the best of my knowledge, there are no existing formally verified PGAS languages or runtime systems.

Supporting More Irregular Memory Access Patterns For each optimization that COPPER performs, it targets memory access patterns of the form $A[B[i]]$, where A and B are arrays, and A is a distributed array. These access patterns are found commonly in graph analytics and scientific computing applications, as illustrated in Chapter 3. However, they are not representative of all possible types of irregular memory access patterns. Other examples include using the evaluation of a function as an index into an array and pointer referencing (lines 10 and 11 in Listing 2.1, respectively).

A more general approach to recognize and support different types of irregular memory access patterns is the Sparse Polyhedral Framework (SPF) [SHO18]. The SPF extends the well-known polyhedral framework [Bag+19; Bas04] by providing a means to represent sparse codes and transformations on those codes. In the context

of this dissertation, which used the Chapel programming language and compiler as a medium to demonstrate COPPER, leveraging the SPF would be a significant task. This is because Chapel's existing compiler does not make use of the polyhedral framework, or perform any affine/non-affine array access analysis. Indeed, all the analysis performed by COPPER to recognize and optimize irregular memory access patterns in Chapel had to be developed from scratch.

Targeting Different Hardware Platforms The systems evaluated in this dissertation were those that fall under the category of distributed-memory systems (i.e., collections of servers connected over a high speed network). However, there are other hardware platforms that suffer the same performance issues that affect distributed-memory systems with respect to irregular memory access patterns.

In the context of applications that require significant amounts of memory (e.g., graph analytics), a system like the Hewlett Packard Enterprise (HPE) Superdome Flex[1] is appealing. A Superdome Flex loosely resembles a traditional distributed-memory machine, in that it consists of multiple chassis connected over a network. However, the hardware is designed to provide a shared-memory view of the system to users, much like how the PGAS model provides software abstractions. This allows for users to write programs that have access to upwards of 48 TB of shared-memory across 1536 cores. Despite the hardware support for a PGAS-like model, memory access latency between different memories still exists. In this way, the optimizations provided by COPPER are likely to be relevant on a system like the Superdome Flex.

Graphics Processing Units (GPUs) are commonly used in many applications today, especially within the machine learning domain [SBS05]. However, they are difficult to leverage for applications that exhibit irregular memory access patterns. This is because GPUs have a Single Instruction Multiple Data (SIMD) architecture, which performs poorly in the presence of access patterns that are sparse and lack local-

[1] https://www.hpe.com/us/en/servers/superdome.html

ity. Nevertheless, there have been many efforts to utilize GPUs for graph analytics [Wan+16] and other sparse applications [RSK18; RSK17]. The domains that leverage GPUs are requiring ever increasing amounts of memory, which means that multiple GPUs are often necessary. As a result, the memory accesses issued between GPUs can be thought of as remote communication within a distributed system, which would present opportunities for a framework like COPPER to be applied.

Finally, there are also novel hardware platforms being explored, which rely on domain specific architectures to accelerate some application space (e.g., graphs, machine learning, etc.). An example of such an architecture is the Emu migratory thread system [Dys+16], which was designed to target irregular memory access patterns by migrating lightweight threads rather than moving data. As part of an overall evaluation effort of the Emu system, I implemented replication and data movement optimizations that share similarities with those provided by COPPER [RKS21a; RKS19; RK18]. Interestingly, while the architecture was designed to target irregular memory access patterns, the optimizations I developed were crucial to achieving good performance from the system. This demonstrates that the techniques used by COPPER to improve performance on traditional systems can also be utilized on novel architectures.

Bibliography

[Aan+20] Sriram Aananthakrishnan et al. "Piuma: Programmable integrated unified memory architecture". In: *arXiv preprint arXiv:2010.06277* (2020).

[ASS95] Gagan Agrawal, Alan Sussman, and Joel Saltz. "An integrated runtime and compile-time approach for parallelizing structured and block structured applications". In: *IEEE Transactions on Parallel and Distributed Systems* 6.7 (1995), pp. 747–754.

[AJ17] Sam Ainsworth and Timothy M Jones. "Software prefetching for indirect memory accesses". In: *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. Austin, TX, USA: IEEE, 2017, pp. 305–317.

[AD18] Mohammad Al Hasan and Vachik S Dave. "Triangle counting in large networks: a review". In: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 8.2 (2018), e1226.

[Alv+12] Michail Alvanos et al. "Automatic communication coalescing for irregular computations in UPC language". In: *Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research*. Toronto, Ontario, Canada: ACM, 2012, pp. 220–234.

[Alv+13] Michail Alvanos et al. "Improving Communication in PGAS Environments: static and Dynamic Coalescing in UPC". In: *Proceedings of the 27th International ACM Conference on Supercomputing (ICS'13)*. Eugene, ORE, USA: Association for Computing Machinery, 2013, pp. 129–138. ISBN: 9781450321303. DOI: 10.1145/2464996.2465006. URL: https://doi.org/10.1145/2464996.2465006.

[Alv+16] Michail Alvanos et al. "Using shared-data localization to reduce the cost of inspector-execution in Unified-Parallel-C programs". In: *Parallel Computing* 54 (2016), pp. 2–14. ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2016.03.002.

[Bac+19] John Bachan et al. "UPC++: A high-performance communication framework for asynchronous computation". In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2019, pp. 963–973.

[Bad+04] Abdel-Hameed Badawy et al. "The efficacy of software prefetching and locality optimizations on future memory systems". In: *Journal of Instruction-Level Parallelism* 6.7 (2004).

[Bad+22] David Bader et al. *Graph500 Benchmark*. https://graph500.org/. 2022.

[Bag+19] Riyadh Baghdadi et al. "Tiramisu: A polyhedral compiler for expressing fast and portable code". In: *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2019, pp. 193–205.

[Bai+95] David Bailey et al. *The NAS Parallel Benchmarks 2.0*. Tech. rep. Technical Report NAS-95-020, NASA Ames Research Center, 1995.

[Bal+18] Prasanna Balaprakash et al. "Autotuning in high-performance computing applications". In: *Proceedings of the IEEE* 106.11 (2018), pp. 2068–2083.

[Bas04] Cédric Bastoul. "Code generation in the polyhedral model is easier than you think". In: *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004*. IEEE. 2004, pp. 7–16.

[BE06] Ayon Basumallik and Rudolf Eigenmann. "Optimizing irregular shared-memory applications for distributed-memory systems". In: *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. 2006, pp. 119–128.

[BMZ99] Vladimir Batagelj, Andrej Mrvar, and Matjaž Zaveršnik. "Partitioning approach to visualization of large graphs". In: *International Symposium on Graph Drawing*. Springer. 1999, pp. 90–97.

[BAP15] Scott Beamer, Krste Asanović, and David Patterson. *The GAP Benchmark Suite*. 2015. DOI: 10.48550/ARXIV.1508.03619.

[Ber+00] Helen M. Berman et al. "The Protein Data Bank". In: *Nucleic Acids Research* 28.1 (Jan. 2000), pp. 235–242. ISSN: 0305-1048. DOI: 10.1093/nar/28.1.235. eprint: https://academic.oup.com/nar/article-pdf/28/1/235/9895144/280235.pdf. URL: https://doi.org/10.1093/nar/28.1.235.

[BGS05] Monica Bianchini, Marco Gori, and Franco Scarselli. "Inside PageRank". In: *ACM Transactions on Internet Technology* 5.1 (2005), pp. 92–128.

[BV04] Paolo Boldi and Sebastiano Vigna. "The WebGraph Framework I: Compression Techniques". In: *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. Manhattan, USA: ACM Press, 2004, pp. 595–601.

[BH17] Dan Bonachea and P Hargrove. *GASNet Specification, v1. 8.1*. Tech. rep. Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2017.

[Boy+19] Michael Boyle et al. "The SXS Collaboration catalog of binary black hole simulations". In: *Classical and Quantum Gravity* 36.19 (2019), p. 195006.

[Bro+09] Bernard R Brooks et al. "CHARMM: the biomolecular simulation program". In: *Journal of computational chemistry* 30.10 (2009), pp. 1545–1614.

[CKP91] David Callahan, Ken Kennedy, and Allan Porterfield. "Software prefetching". In: *ACM SIGARCH Computer Architecture News* 19.2 (1991), pp. 40–52.

[Cha+18] Bradford L Chamberlain et al. *Chapel Comes of Age: Making Scalable Programming Productive*. 2018. URL: https://cug.org/proceedings/cug2018_proceedings/includes/files/pap130s2-file1.pdf.

[Cha+10] Barbara Chapman et al. "Introducing OpenSHMEM: SHMEM for the PGAS Community". In: *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*. PGAS '10. New York, New York, USA: Association for Computing Machinery, 2010. ISBN: 9781450304610. DOI: 10.1145/2020373.2020375. URL: https://doi.org/10.1145/2020373.2020375.

[Cha+05] Philippe Charles et al. "X10: an object-oriented approach to non-uniform cluster computing". In: *Acm Sigplan Notices* 40.10 (2005), pp. 519–538.

[CIY05] Wei-Yu Chen, C. Iancu, and K. Yelick. "Communication optimizations for fine-grained UPC applications". In: *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*. 2005, pp. 267–278. DOI: 10.1109/PACT.2005.13.

[DM98] Leonardo Dagum and Ramesh Menon. "OpenMP: An Industry-Standard API for Shared-Memory Programming". In: *IEEE Comput. Sci. Eng.* 5.1 (Jan. 1998), pp. 46–55. ISSN: 1070-9924. DOI: 10.1109/99.660313. URL: https://doi.org/10.1109/99.660313.

[DS96] Fredrik Dahlgren and Per Stenstrom. "Evaluation of hardware-based stride and sequential prefetching in shared-memory multiprocessors". In: *IEEE Transactions on Parallel and Distributed Systems* 7.4 (1996), pp. 385–398.

[Das+94] Raja Das et al. "Communication Optimizations for Irregular Scientific Computations on Distributed Memory Architectures". In: *Journal of Parallel and Distributed Computing* 22.3 (1994), pp. 462–478.

[DDZ14] Naga Shailaja Dasari, Ranjan Desh, and M. Zubair. "ParK: An efficient algorithm for k-core decomposition on multicore processors". In: *2014 IEEE International Conference on Big Data (Big Data)*. 2014, pp. 9–16. DOI: 10.1109/BigData.2014.7004366.

[Dav06] Timothy A Davis. *Direct methods for sparse linear systems*. SIAM, 2006.

[DH11] Timothy A. Davis and Yifan Hu. "The University of Florida Sparse Matrix Collection". In: *ACM Trans. Math. Softw.* 38.1 (Dec. 2011), 1:1–1:25. ISSN: 0098-3500. DOI: 10.1145/2049662.2049663.

[De +15] Mattias De Wael et al. “Partitioned Global Address Space Languages”. In: *ACM Computing Surveys (CSUR)* 47.4 (2015), pp. 1–27.

[Dhu+20] Laxman Dhulipala et al. “The graph based benchmark suite (gbbs)”. In: *Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. 2020, pp. 1–8.

[DK99] Chen Ding and Ken Kennedy. “Improving Cache Performance in Dynamic Applications Through Data and Computation Reorganization at Run Time”. In: *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*. ACM, 1999, pp. 229–241. ISBN: 1-58113-094-5. DOI: 10.1145/301618.301670.

[Doi+19] Jun Doi et al. “Quantum computing simulator on a heterogenous hpc system”. In: *Proceedings of the 16th ACM International Conference on Computing Frontiers*. 2019, pp. 85–93.

[DHL16] Jack Dongarra, Michael A Heroux, and Piotr Luszczek. “High-performance Conjugate-gradient Benchmark: A New Metric for Ranking High-performance Computing Systems”. In: *The International Journal of High Performance Computing Applications* 30.1 (2016), pp. 3–10.

[Don+08] Jack Dongarra et al. “DARPA’s HPCS Program: History, Models, Tools, Languages”. In: *Advances in COMPUTERS*. Vol. 72. Advances in Computers. Elsevier, 2008, pp. 1–100. DOI: https://doi.org/10.1016/S0065-2458(08)00001-6. URL: https://www.sciencedirect.com/science/article/pii/S0065245808000016.

[Dys+16] Timothy Dysart et al. “Highly Scalable Near Memory Processing with Migrating Threads on the Emu System Architecture”. In: *Proceedings of the Sixth Workshop on Irregular Applications: Architectures and Algorithms*. Salt Lake City, Utah: IEEE Press, 2016, pp. 2–9. ISBN: 978-1-5090-3867-1. DOI: 10.1109/IA3.2016.7.

[Ebc+06] Kemal Ebcioglu et al. “An experiment in measuring the productivity of three parallel programming languages”. In: *Proceedings of the third workshop on productivity and performance in high-end computing*. 2006, pp. 30–36.

[ETS14] H Carter Edwards, Christian R Trott, and Daniel Sunderland. “Kokkos: Enabling manycore performance portability through polymorphic memory access patterns”. In: *Journal of parallel and distributed computing* 74.12 (2014), pp. 3202–3216.

[Eic+92] Thorsten von Eicken et al. “Active Messages: A Mechanism for Integrated Communication and Computation”. In: *SIGARCH Comput. Archit. News* 20.2 (Apr. 1992), pp. 256–266. ISSN: 0163-5964. DOI: 10.1145/146628.140382. URL: https://doi.org/10.1145/146628.140382.

[Eva06] Jason Evans. “A scalable concurrent malloc (3) implementation for FreeBSD”. In: *Proc. of the bsdcan conference, ottawa, canada*. 2006.

[FB15] M. P. Ferguson and D. Buettner. "Caching Puts and Gets in a PGAS Language Runtime". In: *2015 9th International Conference on Partitioned Global Address Space Programming Models*. Washington, DC, USA: IEEE, 2015, pp. 13–24.

[Gal+20] Trevor Gale et al. "Sparse GPU kernels for deep learning". In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2020, pp. 1–14.

[El-+05] Tarek El-Ghazawi et al. *UPC: Distributed Shared Memory Programming*. Vol. 40. Wiley Series on Parallel and Distributed Computing. NJ, USA: John Wiley & Sons, 2005.

[Gle15] David F Gleich. "PageRank beyond the Web". In: *siam REVIEW* 57.3 (2015), pp. 321–363.

[GZ99] Michael Griebel and Gerhard Zumbusch. "Parallel multigrid in an adaptive PDE solver based on hashing and space-filling curves". In: *Parallel Computing* 25.7 (1999), pp. 827–843.

[HT00] Hwansoo Han and Chau-Wen Tseng. "A comparison of locality transformations for irregular codes". In: *International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*. Springer. 2000, pp. 70–84.

[Hei+18] Wim Heirman et al. "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*. PACT '18. Limassol, Cyprus: Association for Computing Machinery, 2018. ISBN: 9781450359863. DOI: 10.1145/3243176.3243181. URL: https://doi.org/10.1145/3243176.3243181.

[Hie+19] Kesha Hietala et al. "A verified optimizer for quantum circuits". In: *arXiv preprint arXiv:1912.02250* (2019).

[Hoc+05] Lorin Hochstein et al. "Parallel programmer productivity: A case study of novice parallel programmers". In: *SC'05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. IEEE. 2005, pp. 35–35.

[KK93] L.V. Kalé and S. Krishnan. "CHARM++: A Portable Concurrent Object Oriented System Based on C++". In: *Proceedings of OOPSLA'93*. Ed. by A. Paepcke. ACM Press, 1993, pp. 91–108.

[Kar+13] Ian Karlin et al. "Exploring traditional and emerging parallel programming models using a proxy application". In: *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE. 2013, pp. 919–932.

[KFE18] Engin Kayraklioglu, Michael P. Ferguson, and Tarek El-Ghazawi. "LAPPS: Locality-Aware Productive Prefetching Support for PGAS". In: *ACM Trans. Archit. Code Optim.* 15.3 (Aug. 2018). ISSN: 1544-3566. DOI: 10.1145/3233299. URL: https://doi.org/10.1145/3233299.

[Kay+21] Engin Kayraklioglu et al. "Locality-Based Optimizations in the Chapel Compiler". In: *Languages and Compilers for Parallel Computing*. Ed. by Xiaoming Li and Sunita Chandrasekaran. NY, USA: Springer, 2021, pp. 3–17. ISBN: 978-3-030-99372-6.

[Ken+00] Ricky A Kendall et al. "High performance computational chemistry: An overview of NWChem a distributed parallel application". In: *Computer Physics Communications* 128.1-2 (2000), pp. 260–283.

[Kim+08] John Kim et al. "Technology-driven, highly-scalable dragonfly topology". In: *ACM SIGARCH Computer Architecture News* 36.3 (2008), pp. 77–88.

[LA04] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Proc. IEEE International Symposium on Code Generation and Optimization*. CGO '04. Palo Alto, California: IEEE Computer Society, 2004, p. 75. ISBN: 0769521029.

[Law+79] Chuck L Lawson et al. "Basic linear algebra subprograms for Fortran usage". In: *ACM Transactions on Mathematical Software (TOMS)* 5.3 (1979), pp. 308–323.

[Ler+16] Xavier Leroy et al. "CompCert-a formally verified optimizing compiler". In: *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*. 2016.

[Löf+21] Júnior Löff et al. "The NAS Parallel Benchmarks for evaluating C++ parallel programming frameworks on shared-memory architectures". In: *Future Generation Computer Systems* 125 (2021), pp. 743–757.

[LM98] Bo Lu and John Mellor-Crummey. "Compiler optimization of implicit reductions for distributed memory multiprocessors". In: *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*. IEEE. 1998, pp. 42–51.

[Lub85] Michael Luby. "A simple parallel algorithm for the maximal independent set problem". In: *Proceedings of the seventeenth annual ACM symposium on Theory of computing*. 1985, pp. 1–10.

[MD19] F. Miller Maley and Jason G. DeVinney. "Conveyors for Streaming Many-To-Many Communication". In: *2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. Denver, CO, USA: IEEE, 2019, pp. 1–8. DOI: 10.1109/IA349570.2019.00007.

[Mal+20] Fragkiskos D Malliaros et al. "The core decomposition of networks: Theory, algorithms and applications". In: *The VLDB Journal* 29.1 (2020), pp. 61–92.

[MWK99] John Mellor-Crummey, David Whalley, and Ken Kennedy. "Improving memory hierarchy performance for irregular applications". In: *Proceedings of the 13th international conference on Supercomputing*. 1999, pp. 425–433.

[Meu+] Robert Meusel et al. *Web Data Commons Webgraph*. http://webdatacommons.org/hyperlinkgraph/. Accessed: October 2021.

[MS03] Ulrich Meyer and Peter Sanders. "Δ-stepping: a parallelizable shortest path algorithm". In: *Journal of Algorithms* 49.1 (2003), pp. 114–152.

[MP16] Stefan Mitsch and André Platzer. "ModelPlex: Verified runtime validation of verified cyber-physical system models". In: *Formal Methods in System Design* 49 (2016), pp. 33–74.

[Mor+14] Alessandro Morari et al. "Scaling Irregular Applications through Data Aggregation and Software Multithreading". In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. Phoenix, AZ, USA: IEEE, 2014, pp. 1126–1135. DOI: 10.1109/IPDPS.2014.117.

[NHL96] Jaroslaw Nieplocha, Robert J Harrison, and Richard J Littlefield. "Global Arrays: A Nonuniform Memory Access Programming Model for High-performance Computers". In: *The Journal of Supercomputing* 10.2 (1996), pp. 169–189.

[Pad+20] Nikhil Padmanabhan et al. "Simulating ultralight dark matter in Chapel". In: *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2020, pp. 678–678.

[Par+21] Matthieu Parenteau et al. "Development of parallel CFD applications with the Chapel programming language". In: *AIAA Scitech 2021 Forum*. 2021, p. 0749.

[Pau+21] Sri Raj Paul et al. *A Scalable Actor-based Programming System for PGAS Runtimes*. 2021.

[Pea17] Roger Pearce. "Triangle counting for scale-free graphs at scale in distributed memory". In: *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 2017, pp. 1–4.

[Pri+19] Benjamin Priest et al. "You've got mail (ygm): Building missing asynchronous communication primitives". In: *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2019, pp. 221–230.

[RP99] L. Rauchwerger and D.A. Padua. "The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization". In: *IEEE Transactions on Parallel and Distributed Systems* 10.2 (1999), pp. 160–180. DOI: 10.1109/71.752782.

[RSK17] Thomas B Rolinger, Tyler A Simon, and Christopher D Krieger. "Performance challenges for heterogeneous distributed tensor decompositions". In: *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 2017, pp. 1–7.

[RSK19] Thomas B Rolinger, Tyler A Simon, and Christopher D Krieger. "Performance considerations for scalable parallel tensor decomposition". In: *Journal of Parallel and Distributed Computing* 129 (2019), pp. 83–98.

[RK18] Thomas B. Rolinger and Christopher D. Krieger. "Impact of Traditional Sparse Optimizations on a Migratory Thread Architecture". In: *2018 IEEE/ACM 8th Workshop on Irregular Applications: Architectures and Algorithms (IA3)* (2018), pp. 45–52.

[RKS19] Thomas B. Rolinger, Christopher D. Krieger, and Alan Sussman. "Optimizing Data Layouts for Irregular Applications on a Migratory Thread Architecture". In: *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. 2019, pp. 7–15. DOI: 10.1109/MCHPC49590.2019.00009.

[RKS21a] Thomas B. Rolinger, Christopher D. Krieger, and Alan Sussman. "Optimizing Memory-Compute Colocation for Irregular Applications on a Migratory Thread Architecture". In: *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2021, pp. 58–67. DOI: 10.1109/IPDPS49936.2021.00015.

[RKS21b] Thomas B. Rolinger, Christopher D. Krieger, and Alan Sussman. *Runtime Optimizations for Irregular Applications in Chapel. The 8th Annual Chapel Implementers and Users Workshop (CHIUW'21)*. 2021.

[RSK18] Thomas B. Rolinger, Tyler A. Simon, and Christopher D. Krieger. "An Empirical Evaluation of Allgatherv on Multi-GPU Systems". In: *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 2018, pp. 123–132. DOI: 10.1109/CCGRID.2018.00027.

[RS22] Thomas B. Rolinger and Alan Sussman. "Compiler Optimization for Irregular Memory Access Patterns in PGAS Programs". In: *Languages and Compilers for Parallel Computing*. To appear. 2022.

[RS] Thomas B. Rolinger and Alan Sussman. *Compiler Optimizations for Remote Data Aggregation and Prefetching in PGAS Programs*. In submission.

[Rol+21] Thomas B. Rolinger et al. "Towards High Productivity and Performance for Irregular Applications in Chapel". In: *2021 SC Workshops Supplementary Proceedings (SCWS)*. St. Louis, MO, USA: IEEE, 2021, pp. 1–11. DOI: 10.1109/SCWS55283.2021.00012.

[SP96] R.H. Saavedra and Daeyeon Park. "Improving the effectiveness of software prefetching with adaptive executions". In: *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Technique*. Boston, MA, USA: IEEE, 1996, pp. 68–78. DOI: 10.1109/PACT.1996.552556.

[SMC91] Joel H. Saltz, Ravi Mirchandaney, and Kay Crowley. "Run-Time Parallelization and Scheduling of Loops". In: *IEEE Transactions on Computers* 40.5 (1991), pp. 603–612.

[Sam+17] Siddharth Samsi et al. "Static graph challenge: Subgraph isomorphism". In: *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 2017, pp. 1–6.

[SV80] Yossi Shiloach and Uzi Vishkin. *An O (log n) parallel connectivity algorithm*. Tech. rep. Computer Science Department, Technion, 1980.

[Smi82] Alan Jay Smith. “Cache memories”. In: *ACM Computing Surveys (CSUR)* 14.3 (1982), pp. 473–530.

[Smi+15] Shaden Smith et al. “SPLATT: Efficient and parallel sparse tensor-matrix multiplication”. In: *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE. 2015, pp. 61–70.

[Sou+09] Jean Souyris et al. “Formal verification of avionics software products”. In: *FM 2009: Formal Methods: Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings 2*. Springer. 2009, pp. 532–546.

[Sri+07] Santhosh Srinath et al. “Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers”. In: *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. IEEE. 2007, pp. 63–74.

[SBS05] Dave Steinkraus, Ian Buck, and PY Simard. “Using GPUs for machine learning algorithms”. In: *Eighth International Conference on Document Analysis and Recognition (ICDAR'05)*. IEEE. 2005, pp. 1115–1120.

[SCF03] Michelle Mills Strout, Larry Carter, and Jeanne Ferrante. “Compile-time Composition of Run-time Data and Iteration Reorderings”. In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming language Design and Implementation*. San Diego, CA, USA: ACM, 2003, pp. 91–102.

[SHO18] Michelle Mills Strout, Mary Hall, and Catherine Olschanowsky. “The sparse polyhedral framework: Composing compiler-generated inspector-executor code”. In: *Proceedings of the IEEE* 106.11 (2018), pp. 1921–1934.

[SY05] Jimmy Su and Katherine Yelick. “Automatic Support for Irregular Computations in a High-Level Language”. In: *IEEE International Parallel and Distributed Processing Symposium*. Vol. 2. IEEE, 2005, 53b–53b.

[SSS17] Justin Sybrandt, Michael Shtutman, and Ilya Safro. “Moliere: Automatic biomedical hypothesis generation system”. In: *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*. 2017, pp. 1633–1642.

[Tal+21] Nishil Talati et al. “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. 2021, pp. 654–667.

[Wan+16] Yangzihao Wang et al. "Gunrock: A high-performance graph processing library on the GPU". In: *Proceedings of the 21st ACM SIGPLAN symposium on principles and practice of parallel programming*. 2016, pp. 1–12.

[Web12] Jim Webber. "A Programmatic Introduction to Neo4j". In: *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*. ACM, 2012, pp. 217–218. ISBN: 9781450315630. DOI: 10.1145/2384716.2384777.

[WPD01] R Clint Whaley, Antoine Petitet, and Jack J Dongarra. "Automated empirical optimizations of software and the ATLAS project". In: *Parallel computing* 27.1-2 (2001), pp. 3–35.

[WMT08] Kyle B Wheeler, Richard C Murphy, and Douglas Thain. "Qthreads: An API for Programming with Millions of Lightweight Threads". In: *IEEE International Parallel and Distributed Processing Symposium*. Miami, FL, USA: IEEE, 2008, pp. 1–8.