RDMA-Based Distributed Data Structures for Large-Scale Parallel Systems

Benjamin Brock

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2022-93
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-93.html

May 13, 2022

Copyright © 2022, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

RDMA-Based Distributed Data Structures for Large-Scale Parallel Systems

by

Benjamin Brock

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Katherine Yelick, Co-chair
Adjunct Assistant Professor Aydın Buluç, Co-chair
Professor Joseph Hellerstein
Associate Professor Zachary Pardos

Spring 2022

RDMA-Based Distributed Data Structures for Large-Scale Parallel Systems

Copyright 2022
by
Benjamin Brock

Abstract

RDMA-Based Distributed Data Structures for Large-Scale Parallel Systems

by

Benjamin Brock

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Katherine Yelick, Co-chair

Adjunct Assistant Professor Aydın Buluç, Co-chair

Running programs across multiple nodes in a cluster of networked computers, such as in a supercomputer or commodity datacenter system, is increasingly important across multiple domains, including data science, machine learning, and scientific computing. This is brought on by a combination of increasing data sizes, which push beyond the memory capacity of a single node, and increasing computational demands from new, more elaborate simulations, models, and applications.

However, writing parallel programs for clusters of computers remains a difficult task, particularly for programs that are irregular in terms of data distribution or access pattern. Many parallel programs today are still written using communication libraries like MPI or OpenSHMEM, which require users to explicitly manage low-level details. While high-level parallel programming languages and libraries do exist, and these can make implementing certain types of programs much easier, developers often have to expend significant effort building custom infrastructure and data structures for their applications.

This thesis argues that a large part of the reason why parallel programming remains difficult is a lack of high-level *distributed data structures* analogous to the data structures that have become ubiquitous in sequential programming environments like C++ and Python. These especially include irregular data structures like hash tables and queues that may require fine-grained memory accesses along with synchronization. This thesis examines techniques for building high-level, cross-platform distributed data structures using one-sided remote memory operations like remote put, remote get, and remote atomics. These memory access primitives allow for a high degree of asynchrony, enabling better performance by removing synchronization bottlenecks and allowing a high degree of overlap between communication and computation. They can also be efficiently executed directly by the network hardware in

modern supercomputer and commodity datacenter networks, removing the need to synchronize with remote processes.

This thesis examines several RDMA-based distributed data structures, including hash tables, Bloom filters, queues, and dense and sparse matrices. We provide a performance model for evaluating the cost of RDMA-based distributed data structure methods in terms of their component remote memory operations, and demonstrate how this model can be extended to support GPUs in addition to conventional CPUs.

To Grandmom, who let my cousin and me take apart her computer.

Contents

Contents **ii**

List of Figures **iv**

List of Tables **vi**

1 Introduction **1**

2 Background and Related Work **4**
2.1 Execution Models for Parallel Programs 4
2.2 Communication Models for Parallel Programs 6

3 Building Distributed Data Structures **13**
3.1 RDMA-Based Communication Primitives 13
3.2 Backends 19
3.3 BCL ObjectContainers 20
3.4 Extension to GPUs 22
3.5 Building Distributed Data Structures 22
3.6 Concurrency Promises 23
3.7 Buffers 24

4 Experimental Setup and Methodology **25**

5 RDMA-Based Queues **28**
5.1 Asynchronous All-to-All Pattern 33
5.2 Concurrency Promises 34
5.3 Experimental Evaluation 35

6 Set Data Structures **42**
6.1 Hash Tables 42
6.2 Bloom Filters 47
6.3 Experimental Evaluation 49

7 Matrix Data Structures **55**
7.1 Introduction 55
7.2 Distributed Matrix Data Structure 56
7.3 Distributed Sparse Matrix Data Structure 61
7.4 Sparse Matrix Multiplication 70
7.5 Background 74
7.6 Performance Model 78
7.7 Implementation 81
7.8 Evaluation 83
7.9 Related Work and Conclusions 89

8 Comparing RDMA and RPC-Based Distributed Data Structures **92**
8.1 Introduction 92
8.2 Background 94
8.3 Experimental Design 99
8.4 Results 102
8.5 Conclusions 108

9 Conclusion **109**

Bibliography **111**

List of Figures

3.1 Organizational diagram of BCL. 19

5.1 Process for pushing values to a BCL `FastQueue`. First (1) a `fetch_and_add` operation is performed, which returns a reserved location where values can be inserted. Then (2) the values to be inserted are copied to the queue. 33
5.2 Our bucket sort implementation in BCL for the ISx benchmark. 34
5.3 Performance comparison on the ISx benchmark on three different computing systems. All runs measure weak scaling with 2^{24} items per process. 36
5.4 Microbenchmarks for CircularQueue. Benchmarks with "many" in the title are performed with one queue per node, while benchmarks without "many" are performed with all processes accessing a single queue. The label before the underscores indicates the operation performed, while the label after the underscore indicates the concurrency promise used. For example, "pushpop" indicates the concurrency promise `push | pop`. 39
5.5 Microbenchmarks for FastQueue. Benchmarks with "many" in the title are performed with one queue per node, while benchmarks without "many" are performed with all processes accessing a single queue. 40
5.6 Microbenchmarks for ChecksumQueue. Benchmarks with "many" in the title are performed with one queue per node, while benchmarks without "many" are performed with all processes accessing a single queue. ChecksumQueue does not support concurrency promises, since the checksum must always be calculated and written for a push to complete successfully. However, the label after the underscore indicates the concurrency promise level that could be supported by the implementation for easier comparison with the CircularQueue plots in Figure 5.4. 41

6.1 A small change to user code—inserting into the `HashMapBuffer` instead of the `HashMap`—causes inserts to be batched together. 47
6.2 Performance comparison on the Meraculous benchmark on the *chr14* dataset. . 49
6.3 Performance on the Meraculous benchmark 50
6.4 Strong scaling for our k-mer counting benchmark using dataset *chr14*. 52
6.5 Microbenchmarks for the hash table. 53

7.1 A collection of matrix tile grids and distributions. 60

7.2 Illustration of the local matrix multiplications necessary to compute one block of the output matrix. All submatrices in the block row of A are multiplied by the corresponding submatrices in the block column of B. 64
7.3 Total (end-to-end) vs. per-stage load balance multiplying a R-MAT model-generated sparse matrix with a sparse 2D algorithm. Simulated on a 16×16 process grid. 74
7.4 *Inter-node* roofline plots for SpMM and SpGEMM with a 2D distribution. SpMM plot models performance for different widths of the dense B matrix at a fixed scale (24 GPUs), while SpGEMM models performance at different scales. Dashed horizontal lines represent *local roofline peaks* for SpMM and SpGEMM operations, while vertical lines represent *inter-node* roofline peaks for particular problems. . 80
7.5 Single-node runtimes for SpMM, with different numbers of columns N in the dense matrix B. 85
7.6 Multi-node runtimes for SpMM, with different numbers of columns N in the dense matrix B. 87
7.7 SpGEMM strong scaling experiments. 91

8.1 Modifying a hash table using one-sided RDMA operations. 95
8.2 Modifying a hash table using an RPC. 96
8.3 The component latencies for RDMA operations and AMs on Cori. 104
8.4 Latencies for RDMA- and RPC-based queue push operations. 105
8.5 Latencies for RDMA- and RPC-based hash table operations. 106
8.6 Measuring the cost of a queue insertion as remote processes become less attentive due to intermixed computation. 107

List of Tables

4.1 Summary of systems used in evaluation throughout this work. 25

5.1 A selection of methods from BCL queues. Costs are best case, using implementation chosen with no concurrency promises. R is the cost of a remote read, W the cost of a remote write, A the cost of a remote atomic memory operation, B the cost of a barrier, ℓ the cost of a local memory operation, and n the number of elements involved. c is the number of bytes in a checksum divided by the number of bytes per element (usually ≤ 1). 31
5.2 Implementations of data structure operations for BCL's circular queue data structure. 31

6.1 Implementations of data structure operations for BCL's hash table data structure. 45

7.1 Matrix block descriptors, which describe different tiling strategies for a distributed matrix. 57
7.2 A selection of methods from our dense and sparse distributed matrix data structures. R is the cost of a remote read, W the cost of a remote write, A the cost of a remote atomic memory operation, B the cost of a barrier, ℓ the cost of a local memory operation, and n is the number of elements involved in the data structure operation. m_{tile}, n_{tile}, and nnz_{tile} are the number of rows, columns, and nonzeros per tile. 60
7.3 Sparse accumulators. 63
7.4 Matrices used in our experiments. "load imb." lists the imbalance in number of nonzeros when split amongst 100 processes on a 10×10 2D process grid. 84
7.5 Component breakdown for selected matrices. 89

8.1 Latency of various RDMA operations, measured on Cori with 64 nodes. 97
8.2 RDMA-based hash table method implementations considered in this paper. . . . 98
8.3 Implementations for circular queue methods. 99

Acknowledgments

First and foremost, I would like to thank my advisors, Kathy Yelick and Aydın Buluç, for their mentorship and support throughout my PhD.

Kathy was always quick to make time for me, despite her impossibly busy schedule, and she was always eager to connect me with potential collaborators and contacts, both within the university and at Lawrence Berkeley National Laboratory. Her depth and breadth of expertise, on an array of topics, including parallel computing, compilers, domain science, and more, was invaluable to me, particularly as a fledgling graduate student. Kathy was always gracious enough to recognize of her students that, "While they didn't always seem to do what [she] said, they tended to end up doing good things anyway," and for that freedom I owe her a large debt of gratitude.

Aydın joined on as my advisor during my second year, and was instrumental in helping me find a central direction for my research, and what eventually became the core of this thesis. His passion and enthusiasm during our research discussions provided motivation both to start new and exciting research projects, but also to push through the grind in order to finish them. Aydın's practical experience implementing large-scale sparse linear algebra codes with CombBLAS was extremely useful to me as I set off on my own journey to implement RDMA-based data structures, and his deep understanding of algorithms on a theoretical level as well as performance analysis has shaped how I think about writing, implementing, and evaluating my own data structures and algorithms.

I must also extend my thanks to the other members of my dissertation committee, Joe Hellerstein and Zachary Pardos, whose feedback has greatly improved the quality of this thesis. In particular, Joe's insightful comments during my qualifying exam and dissertation writing process helped me view my work through a much wider and clearer lens, and as a result this thesis has been significantly improved.

I have had the great fortune throughout my PhD to work with a number of excellent collaborators and co-authors, and much of the work in this thesis, as well as the researcher I have become today, would not be possible without them. Yuxin Chen has been a longtime colleague and collaborator, and much of the work that I have done with building data structures for GPUs would not have been possible without the many discussions we have had about building communication libraries for GPUs. Jiakun Yan, who interned with us during the Fall of 2019, worked with me on an aggregation runtime for RPCs, and our many research discussions, as well as much of the work that Jiakun did with us, helped shape the work in Chapter 8. I must extend a deep thanks to Nathan Pemberton, Howard Mao, and Jenny Huang, my colleagues from the computer architecture lab. They were gracious enough to collaborate with me on a number of projects combining my distributed data structures work with experimental hardware they were developing, and I also learned a great deal from our interactions. I also owe a great deal of thanks to Michael Driscoll, my academic older brother, who took me under his wing as a second year graduate student. Working with Michael taught me a great deal about writing good software and good papers, as well as brewing good beer.

I had the opportunity to work with a number of collaborators and colleagues at Lawrence Berkeley National Laboratory, and special thanks go out to Taylor Groves, John Bachan, Paul Hargrove, Dan Bonachea, and others.

I also had the pleasure of working with the GraphBLAS Languages Committee on the GraphBLAS C and C++ APIs during my PhD. Many enlightening conversations on sparse and irregular data structures as well as API design helped shape my views on how to build high-level data structures, so my thanks go out to Aydın Buluç, Tim Mattson, Scott McMillan, and José Moreira.

I have received support, both academic and personal, from so many graduate students throughout my PhD that they are too numerous to all name here. I have been lucky enough to have Kevin Läufer as both a friend and colleague from the start of my PhD. Kevin and I would always bounce ideas off of each other, and he did not hesitate to bounce many of my less brilliant ideas back in my direction. A number of older graduate students gave me wonderful advice and helped orient me to graduate school life, and my thanks go out to Marquita Ellis, Michael Driscoll, Yang You, Penporn Koanantakool, Evangelos Georganas, Grace Dinh, Albert Magyar, Orianna DeMasi, Becca Roelofs, Nathan Pemberton, and others. I was also blessed to have a lively cohort of PhD students to go through the experience of grad school with, and my thanks go out to Rohan Bavishi, Michael Dennis, Cristina Teodoropol, Alex Reinking, Kristina Monakhova, Rachel Chen, Erin Grant, Esther Rolf, David Gaddy, Giulia Guidi, Alok Tripathy, and Vivek Bharadwaj, and many others. Specific thanks, also, to Eric Love, author of the Ressort programming language, for his priceless advice to "perhaps not make a new programming language during your PhD, if you can avoid it."

Special thanks go to my girlfriend Caroline Lemieux for her companionship throughout both of our PhD journeys. Graduate school life can be hectic and frustrating at times, and it has been made so much more pleasant by having you there to share it with. Outside of grad school, we climbed mountains, went on long runs and bike rides, and visited new countries and continents together. For all of the adventures and trivialities, for all the hills and bayous, for cutting my hair during the pandemic, and for teaching me French, you have my sincerest gratitude. *C'était, en fait, le mieux d'être avec toi.*

A sincere thank you must go to my family, without whom none of this would have been possible. My parents have always been an inspiration to me, both in their own academic pursuits and in the attitudes they encouraged in me and my sister toward knowledge and personal growth. From when I was very young, learning was not an activity only performed in the classroom, and we were surrounded by good books and by interesting places and people. From nature walks in the Great Smoky Mountains to trips to bustling markets in the most densely populated city on Earth, my sister and I were always surrounded with the best learning environment. Thanks to my father, who passed on to me his love of reading and his gift of gab, as well as my mother, who homeschooled my sister me while we lived abroad, and who passed on to me her love for music and her love of learning. Thanks to my sister, who has always been there for me, and whose will to follow her dreams and adventurous spirit have always served as an inspiration. And finally, a big thanks goes to my grandparents, who always nurtured my curiosity, particularly toward computers.

The work presented in this thesis was supported in part by the National Science Foundation Graduate Research Fellowship Program under Grant No. GE 1752814, as well as under National Science Foundation Award No. 1823034. It used resources of the National Energy Research Scientific Computing Center at Lawrence Berkeley National Laboratory, as well as the Oak Ridge Leadership Computing Facility at Oak Ridge National Laboratory, which are supported by the Department of Energy's Office of Science under Contract Nos. DE-AC02-05CH11231 and DE-AC05-00OR22725, respectively. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation Grant No. ACI-1548562. This work used the Extreme Science and Engineering Discovery Environment (XSEDE) Bridges-2 at the Pittsburgh Supercomputing Center through allocation ASC180051. This work also used computational resources obtained through the AWS Cloud Credits for Research Program.

Chapter 1

Introduction

Due to a number of trends across data science, machine learning, scientific computing, and computer architecture, parallel computing has become more and more ubiquitous and important. Ever-growing datasets are pushing data and graph analytics workloads beyond the limits of a single computer's memory [104]. More and more sophisticated machine learning models require more computational resources as well as more memory in order to be trained and deployed in reasonable amounts of time [119]. In scientific computing, computational demands in fields like genomics, climate science, and medical imaging mean that new scientific advances often require ever-increasing amounts of computational resources [8, 46, 116]. At the same time, the breakdown of important trends in computer architecture like Dennard scaling and Moore's law mean that we can no longer depend on exponential gains in the computational performance of individual conventional processors [109]. All of this means that parallel computing has become an increasingly important tool across much of computer science and adjacent fields [8, 75, 100]. However, writing *parallel programs*, or programs that run on clusters of computers, remains notoriously difficult, particularly for applications which have irregular data layouts or access patterns.

The software tools at the disposal of the average parallel programmer today are little different than they were a decade ago. The most commonly used communication libraries, such as the Message Passing Interface (MPI) [110] and OpenSHMEM [20], all have interfaces that were crystallized in the late 90s or early 2000s. While these low-level communication libraries can serve as excellent platforms for applications that are closely compatible to the interfaces they supply, such as applications that can easily conform to *bulk synchronous parallel* communication model [37], where discrete phases of local computation are separated from communication operations by global synchronization. For applications that are *irregular*, either because their data layouts do not conform to a regular block distribution or because their data access patterns require sparse or random access, implementation can be particularly difficult, with users having to explicitly manage the details of synchronization and memory management across nodes [83, 117].

Dedicated programming languages and libraries do exist for writing parallel programs, and these can somewhat ease the burden of parallel programming by introducing dedicated

abstractions for communication and data distribution. However, these dedicated parallel programming environments have seen relatively little uptake in the broader HPC community. One of the primary reasons for this is that it can be difficult for developers to take the leap to a new programming language, with new, often immature toolchains. This may be exacerbated by the fact that developers often have pre-existing code in conventional programming languages like C, C++, Python, or Fortran that they need to integrate with, adding further barriers to adopting new languages. In addition, when developers are able to use dedicated parallel programming languages, they often must spend time implementing application-specific data structures, since most parallel programming environments lack expressive data structures that have become ubiquitous in sequential programming environments. Before a parallel programmer can implement their application, it is not uncommon to first have to implement the necessary data structures such as distributed hash tables, matrices, or queues. While parallel programming environments may simplify the process of writing these data structures, users are generally left to implement custom data structures on their own. This leads to a situation in which application developers must spend significantly more time implementing their parallel application than developing their parallel algorithms.

This thesis argues that one of the primary reasons that writing parallel programs for supercomputers and commodity datacenter environments remains difficult is because of this lack of high-level data structures. High-level data structures such as arrays, hash tables, and matrices have become ubiquitous in sequential programming environments like C++ and Python and as a result have greatly reduced programmer burden. In parallel programming environments and communication libraries, however, *distributed data structures* have remained largely absent, even in many high-level parallel programming environments like UPC [41] and UPC++ [17]. When they have provided data structures, most high-level parallel programming environments have focused on regular data structures, such as multi-dimensional dense arrays or regular grids. While these regular data structures are no doubt useful for applications which fit them, they are of limited use for applications that are irregular, meaning that either their data is not structured in a way that allows it to be fit into a data structure like a multi-dimensional array, hierarchical grid, or structured sparse matrix, or their data access patterns are fine-grained or irregular. In fact, Kennedy, et al. [72] cite a lack of support for irregular and sparse data distributions as one of the primary reasons for the decline of High Performance Fortran (HPF), saying that the data distribution mechanisms in HPF were not sufficiently expressive to efficiently handle sparse or irregular data. This caused many performance-conscious users to switch to low-level libraries like MPI, which gave them more granular control over performance [73].

In this thesis, I argue that providing high-level distributed data structures can reduce the burden on parallel application developers, particularly for irregular applications. I present a design for building distributed data structures using *one-sided* communication operations, remote get, remote put, and remote atomic operations that allow processes to access and manipulate data resident on a remote process. These operations can be executed directly by the network hardware on modern supercomputer and commodity datacenter platforms using remote direct memory access (RDMA). This allows processors to access and manipulate re-

mote data at very low latencies, since the operations can be completed in hardware and also there is no need to wait on a remote process to handle requests. This RDMA-based design also allows for a greater level of asynchrony, since processes can retrieve data independently, without needing to synchronize with other processors. This model of RDMA-based access maps well onto the data structure interfaces that users are familiar with from sequential programming environments, where inserting an element into a hash table or writing to an element in an array is a fundamentally one-sided operation. I discuss the design and implementation of several high-level distributed data structures, including hash tables, Bloom filters, queues, and dense and sparse matrices, along with algorithmic and implementation benefits that can be achieved by using an RDMA-based design. I demonstrate how this design can be extended to support GPUs. Finally, I build a performance model to evaluate the cost of various RDMA-based data structure operations in terms of their component RDMA operations, comparing these costs to a remote procedure call (RPC) implementation.

The rest of the thesis is as follows. Chapter 2 discusses background and related work. Chapter 3 presents our general framework for building cross-platform, high-level distributed data structures in C++ that can be used natively inside applications written using different communication libraries. Chapter 4 discusses the supercomputer systems and experimental setup used in experiments throughout this thesis. Chapter 5 presents and analyzes a number of different designs for RDMA-based queues. Chapter 6 discusses the design of distributed set-based data structures, including hash tables and Bloom filters. Chapter 7 discusses the design of RDMA-based dense and sparse matrix data structures, along with new RDMA-based algorithms for sparse matrix multiplication that are enabled by one-sided access. Finally, Chapter 8 presents a performance model for RDMA-based data structure operations, comparing the costs to RPC-based implementations, and Chapter 9 presents overall conclusions.

Chapter 2

Background and Related Work

We begin this chapter with a brief overview of background necessary for the rest of this thesis, beginning with *execution models* for parallel programs, which define how a program can be executed across many nodes in a supercomputer or cluster. We then discuss different *communication models*, along with their advantages and disadvantages. Finally, we discuss related work in parallel programming environments, including dedicated programming languages as well as libraries, before providing an overview of related work in distributed data structures and high-level programming models for parallel programs in general.

2.1 Execution Models for Parallel Programs

In order to execute a program in parallel across multiple computers, we need some model for how the code will be executed across multiple computers. In a distributed memory context, execution models roughly fall into two categories, single program multiple data (SPMD), and task-based models. In this thesis, we assume a SPMD execution model, although the RDMA-based data structures discussed here are applicable regardless of execution model.

SPMD Execution Model

In the SPMD execution model, a single program is run by multiple processes. Typically, when users execute a program, they select the number of processes that it should be started with (often stylized as `nprocs`). `nprocs` copies of the program will then be started on different processes, with different processes possibly being executed on different nodes in a cluster. Parallel execution frameworks will usually provide programmers with a mechanism for each process to obtain its process ID (often stylized as `rank`) as well as the total number of processes `nprocs`. These processes may then communicate with each other using a variety of mechanisms, depending on the communication model in use.

A simple example of a parallel program written using the SPMD execution model is shown in Figure 1. When the user starts the program, as with `mpirun -n 4 ./spmd_example` in the

execution printout, multiple copies of the program will be executed on different processes based on the number of processes specified by the user. Each statement in the program will be executed by each process, as with the `printf` statement, unless an explicit branch prevents it, as with the **if** statement. Each process has access to its process ID and the total number of processes. The program in Figure 1 is written using BCL [26], the framework described in this thesis, but in most other communication frameworks, including MPI [110], OpenSHMEM [34], GASNet-EX [24], and others, the corresponding programs would be written with only slightly modified syntax.

Algorithm 1 Simple example of a program using the SPMD execution model, with an observed execution output in the comment below. Note that the lines could print in different orders, depending on the order in which processes execute the program.

```
#include <bcl/bcl.hpp>

int main(int argc, char** argv) {
  BCL::init();

  printf("Hello, World! I am rank number %lu out of %lu\n",
         BCL::rank(), BCL::nprocs());

  if (BCL::rank() == 0) {
    printf("Special hello from rank %lu\n",
           BCL::rank());
  }

  BCL::finalize();
        return 0;
}

/*
  Execution output:
  $ mpirun -n 4 ./spmd_example
  Hello, World! I am rank number 1 out of 4
  Hello, World! I am rank number 2 out of 4
  Hello, World! I am rank number 3 out of 4
  Hello, World! I am rank number 0 out of 4
  Special hello from rank 0
*/
```

The SPMD execution model is the dominant execution model used for batch computing jobs on supercomputers. This is likely due to scalability concerns, since as the number

of processes in a distributed execution increases, the overhead associated with spawning additional tasks increases. The SPMD model reduces the overhead of spawning tasks inside the actual program execution by taking care of this with one large execution at program startup.

Task-Based Models

In task-based execution models, programs are expressed as a collection of discrete tasks, where each task is to be executed by a single processor. These tasks may either be dynamically spawned on demand at runtime, as is the case in systems like Parallel Virtual Machine (PVM) [49] and Ray [81], or, if the complete task graph is known ahead of time, they may be spawned by a runtime system with an intelligent scheduler, as is the case with systems like Legion [21]. The core difference between the SPMD execution model and task-based models is how users express their programs, with users needing to segment their programs into tasks in a task-based model and users needing to think about different processes or groups of processes in the SPMD model. One of the advantages of task-based models is that the user does not necessarily have to explicitly define how their program should be distributed, but a static or dynamic scheduler can instead make the decision about how to partition tasks across nodes in a cluster. An intelligent scheduler can then make optimal decisions for any particular number of processors, partitioning tasks in such a way as to maximize parallelism.

Some frameworks that support remote procedure calls (RPCs), such as UPC++ [17], can be viewed as a hybrid approach that blend aspects of task-based models into a SPMD execution model. For example, while UPC++ programs are executed across a set number of processes chosen by the user at runtime, UPC++ allows users to describe their programs in terms of tasks, and users can compose these tasks into arbitrarily complex task graphs using futures and continuations. Unlike traditional task-based models however, the user must explicitly specify on which process each RPC task should be executed, which requires the user to determine how the computation should be split up amongst the available processes in the distributed execution.

2.2 Communication Models for Parallel Programs

Given a particular execution model, there are multiple mechanisms that can be used to communicate between processes. These are roughly split into variants of the bulk synchronous model, which separates communication and local computation into discrete phases partitioned by global synchronization, and models that use asynchronous forms of one-sided communication with primitives like remote memory operations and RPC.

Before we dive into the different communication models, let's first discuss the underlying mechanisms that can be used to send messages over the network. Message passing, also called *two-sided communication*, is one of the primary methods used for sending data between processes. When using two-sided communication, processes have the ability to send or receive

messages using send or receive calls. In order for a process to send a message to another process, the corresponding process must issue a matching receive call in order to complete the message. Blocking send and receive calls inherently incur some synchronization cost, since two processes must synchronize together in order for the message to be passed between them. If one of the processes is delayed getting to its send or receive call, the other process will be forced to wait, incurring some overhead. Asynchronous versions of send and receive calls do exist, and these can be used to implement asynchronous runtime systems, as is the case with YGM [93] or indeed MPI's own implementation of one-sided remote memory access primitives when RDMA hardware is not available [52].

One-sided communication, by contrast, requires the involvement of only one process to complete. Primary examples include remote get, remote put, and remote atomic operations that are supported by RDMA hardware. In order for a process to access or manipulate another process' shared memory segment, the remote process need not be involved. Instead, the only requirement is that that process' shared segment have been made available to the origin process through some registration process before the remote operation is issued. On systems without native network hardware support for RDMA, one-sided remote operations can be emulated within the communication library in software, although this may incur a runtime cost [44]. In terms of the user-facing communication model, operations are still one-sided, since the user does not need to direct the remote process to take part in the operation. Instead, a runtime system will ensure that the remote process performs the remote memory operation at some point. In this vein, remote procedure calls (RPCs) may be considered another form of one-sided operation, since a remote process' involvement will at some point be required to actually run the RPCs, with this typically taking place inside a runtime system. However, this is generally invisible to the user, with the RPC library's runtime system ensuring that each process handles any waiting RPC requests, either using dedicated progress threads that handle requests in the background, or by checking for waiting RPC requests each time a library function is invoked [14].

Bulk Synchronous Model

One of the most dominant paradigms for distributed memory communication in use today is the bulk synchronous parallel model [37]. The bulk synchronous model typically segments applications into alternating phases of communication and computation. These phases are separated from each other by barrier operations which ensure global synchronization between all the processes. During a computation phase, processes independently perform local computation on local data. During a communication phase, these processes work together to communicate data using collective communication operations such as broadcast, allreduce, or all-to-all. Collective communication operations have a number of advantages, such as that they provide a clear boundary for the programmer between local computation and global communication. Not least of collective communication operations' advantages is that they provide a standardized interface for performing communication. This allows communication library implementers to implement a set of standard communication collectives, possibly

with different implementations for different network architectures, messages sizes, and data distributions, and for users to automatically reap the benefits of new algorithms and optimizations once they are implemented. These collective communication operations can be implemented in a topology-aware manner, which allows for them to achieve better performance on low-radix network architectures, where latency between pairs of nodes that are far away from each other may be quite high, and bandwidth quite low. In addition, library implementations of collectives are able to use techniques like recursive doubling and pipelining to achieve asymptotically better performance compared to point-to-point versions [106]. In a high-performance computing context, the MPI communication library [110, 54] is commonly used to implement programs in a bulk synchronous style. MPI implements a number of fundamental primitives for performing bulk synchronous communication, including broadcast, reduce, allreduce, and many others. While these collective operations may be implemented using either one-sided or two-sided operations, the primitives themselves follow the bulk synchronous model, with the corresponding advantages and disadvantages.

It is important to note that while there do exist asynchronous versions of collective operations such as asynchronous broadcast and asynchronous allreduce, and these can allow for greater overlap of communication and computation, these do not allow the same degree of asynchrony as a fully one-sided approach. In general, it is not possible for a process to exit an asynchronous broadcast until at least some other processes have entered the broadcast. In broadcast operations that use low radix topologies along with aggressive pipelining, as is done in collective operations for GPUs in nccl [69], which uses a ring topology, no process can exit the collective operation until all processes have entered it, since the collective's performance depends on pipeline parallelism gained from having all processes send messages simultaneously. This is also true regardless of topology for collectives like reduce and allreduce, where inputs from multiple processes must be accumulated together.

Partitioned Global Address Space Model

In contrast to the bulk synchronous model, the Partitioned Global Address Space (PGAS) communication model is based around allowing asynchronous one-sided communication operations. In the PGAS model, each process is equipped with a special region of memory called a *shared segment*. The shared segments are typically allocated once at program startup, with each process having an equally sized shared segment, where the size of the shared segments is configurable at runtime. Processes can use *global pointers*, which reference memory that lives in the shared segment of some process, to issue one-sided remote get, put, and atomic operations to access and manipulate the memory of other processes. In this way, the shared segments together form a globally addressable memory space. However, this memory space is partitioned across all of the different processes. The PGAS model can allow for more asynchronous algorithms than the bulk synchronous model, since processes do not necessarily need to synchronize with other processes in order to access data. For instance, if a process needs to access part of a distributed data structure, using the PGAS model, the process can simply access that data directly using remote memory operations into the memory of the

remote process. One of the other advantages of the PGAS model is that its communication primitives correspond directly to RDMA primitives that are available in the hardware. This means that when PGAS programs are executed on modern supercomputer or commodity datacenter systems, these one-sided communication operations can be completed with very low latencies, and without requiring the involvement of remote processes.

Examples of parallel programming environments that provide support for PGAS include parallel programming languages such as UPC [41] and Chapel [111], as well as communication libraries such as OpenSHMEM [34], MPI's remote memory access primitives [52], and GASNet [23]

RPC-Based Models

As discussed above, remote procedure calls (RPCs) may be viewed as an extended form of remote memory access that is able to handle more complex control flow. In fact, this was the original vision for *active messages* (AMs) [47], which are a restricted form of RPC that sends messages carrying a fixed-size data payload along with a pointer to a user space function to be executed. Because of their low-latency design, active messages come with restrictions, namely that their payloads are fixed size and that they cannot issue additional AMs [47, 23]. Active messages can be a very powerful tool, allowing programmers and library developers to implement new, low-latency operations that include more complex control flow than is possible with the current generation of RDMA operations.

Unlike active messages, remote procedure calls in general do not have these restrictions. In most frameworks, RPCs are free to issue additional RPC operations and to carry payload arguments of unlimited size. This creates additional latency due to the additional overhead associated with maintaining an additional RPC queue of potentially unbounded length with variable-size arguments [58].

While both RPC and the more restricted active messages have advantages over RDMA operations in terms of expressiveness, they suffer from performance degradation due to lack of attentiveness. When a process issues an RPC to be executed on a remote process, it must wait until that process finishes any local computation or other communication operations it is involved with, enters a progress function, and dequeues and runs the RPC. When RPCs are interspersed with computation, this may cause a dramatic increase in latency unless resources are dedicated to maintain progress threads. We discuss this tradeoff in further detail in Chapter 8.

High-Level Parallel Programming Environments

A significant amount of the high-performance computing literature, as well as engineering effort in the HPC space, has been devoted to alleviating the burden on programmers who develop and optimize applications that run in large-scale distributed memory supercomputer as well as commodity datacenter environments.

Parallel Programming Languages

Most early approaches to developing high-level programming environments for parallel application developers focused on developing new programming languages. A number of dedicated programming languages or language extensions provide language-level support for one-sided remote memory access primitives, almost all using a Partitioned Global Address Space (PGAS) programming model. UPC [41], Titanium [118], X10 [36], High Performance Fortran [72], and Chapel [111, 32] are parallel programming languages which offer a PGAS distributed memory abstraction. There are a number of advantages associated with using a dedicated programming language that has a semantic understanding of a distributed memory space. Chief among them is the ability to perform static optimizations using global program knowledge. This can allow an optimizing compiler to automatically relax synchronization, to convert remote operations into local ones, or to aggregate many small messages together to improve network performance [71]. Language builders also have the ability to restrict the syntax that users have access to, in order to encourage them to write easily parallelizable programs. However, due to the complexities associated with developing and implementing a new programming language, a compiler, and the libraries needed to make it usable, developing parallel programming languages can be a difficult task, and they have historically had difficulties increasing adoption [72, 73].

High-Level Parallel Programming Libraries

UPC++ is a C++ library which offers a PGAS distributed memory programming model as well as the ability to issue remote procedure calls (RPCs) [121, 16, 17]. UPC++ has a heavy focus on asynchronous programming that is absent from BCL, including futures, promises, and callbacks. UPC++'s RPCs can be used to create more expressive atomic operations, since all RPCs are executed atomically in UPC++. However, these operations require interrupting the remote CPU, and thus have slower throughput than true RDMA atomic memory operations. The current version of UPC++ also lacks a library of data structures, and UPC++ is closely tied to the GASNet communication library, instead of supporting multiple backends.

Global Arrays provides a portal shared memory interface, exposing globally visible array objects that can be read from and written to by each process [86]. While many application-specific libraries have been built on top of Global Arrays, it lacks the kind of high-level generic data structures that are the focus of this work.

Distributed Data Structures

DASH is another C++ library that offers a PGAS distributed memory programming model [48]. DASH has a large focus on structured grid computation, with excellent support for distributed arrays and matrices and an emphasis on providing each process with fast access to its local portion of the distributed array. While DASH's data structures are generic, they do

not support storing objects with complex types. DASH is tied to the DART communication library, which could potentially offer performance portability through multiple backends, but is currently limited to MPI for distributed memory communication.

HPX is a task-based runtime system for parallel C++ programs [70]. It aims to offer a runtime system for executing standard C++ algorithms efficiently on parallel systems, including clusters of computers. Unlike BCL, which is designed to use coordination-free RDMA communication, HPX's fundamental primitives are remote procedure calls used to distribute tasks.

STAPL, or the standard adaptive template library, is an STL-like library of parallel algorithms and data structures for C++ [105]. STAPL programs are written at a much higher level of abstraction than BCL, in a functional style using special higher-order functions such as map, reduce, and for-each which take lambda functions as arguments. From this program description, STAPL generates a hybrid OpenMP and MPI program at compile time. Some versions of STAPL also include a runtime which provides load balancing. The current version of STAPL is only available in a closed beta and only includes array and vector data structures [103].

PETSc, Chombo, and AMReX provide data structures for sparse and dense matrices and structured and unstructured grids, but do not focus on the types of irregular, generic data structures discussed here [1, 40, 120].

Aguilera, et al. have proposed various hardware extensions to RDMA specifically to allow for the efficient execution of operations on remote data structures in NIC hardware [3]. These include an indirect access primitive that can be used to access a value at an offset from a pointer on a remote node, allowing for a dynamically resizing remote vector; various scatter and gather primitives, and a form of notifications. We believe that the performance model presented here, in Chapter 8, is good fit for evaluating potential new RDMA instructions, and that similar microbenchmark analysis can help hardware and software developers to design and evaluate new hardware and data structures.

RPC-Based Programming Models

AM++ is C++ library built on top of MPI that provides a high-level active message API similar to that discussed in this paper [113]. Active Pebbles extends AM++ by adding support for message aggregation and more sophisticated termination detection mechanisms [112]. You've Got Mail (YGM) is an MPI-based system that provides an active message-like API with message- and node-level aggregation [93].

The Multipol library provided a set of concurrent data structures on top of active messages, including dynamic load balancing and optimistic task schedulers [31]. However, it was non-portable and did not have the rich set of hash table data structures discussed here nor the notion of concurrency promises.

The Python-based parallel programming frameworks Ray [81] and Dask [95] expose parallelism using RPCs. However, both of them use a scheduler to automatically determine

where to place RPCs within a cluster, rather than requiring users to explicitly declare where tasks should be executed.

Libraries for Aggregation

Message aggregation is an important technique for optimizing the execution of parallel applications that perform a large number of fine-grained operations. By bundling up many small messages together into larger aggregated messages, the latency costs associated with sending many small messages over the network can be eliminated. Aggregation systems that automatically bundle communication operations together can thus transform what would otherwise be latency-bound problems into bandwidth-bound ones.

Significant work has been done on using aggregation within some of the parallel programming environments already discussed, such as UPC and UPC++ [51, 17], but some libraries have been specifically developed to support aggregation. Bale is a C library designed to support aggregation, offering the core abstraction of *conveyors*, which are dedicated streams shuttling data between processes [79]. Each process can pull data from a conveyor or push data to a particular process using a conveyor. The Bale system will automatically aggregate pieces of data that need to be transferred across the conveyor, and parameters such as target message size can be tuned independently after a program has been written.

Many RPC-based programming environments also offer tools to automatically aggregate RPC requests together, sending them over the network in larger bundled messages. The previously mentioned Active Pebbles [112] implements aggregation automatic aggregation of RPCs on top of the AM++ [113] programming environment, while YGM [93] implements an RPC-like with node-level aggregation directly over MPI. The library ARL [114] implements a future-based RPC interface with automatic node-level aggregation on top of GASNet.

Chapter 3

Building Distributed Data Structures

This chapter introduces the overall design used to develop RDMA-based distributed data structures in this work, starting with a discussion of the hardware capabilities of modern RDMA-based networks in regards to building distributed data structures. We follow with a detailed introduction of the *BCL Core*, which is the cross-platform communication API, based on global pointers, that we use to implement distributed data structures. We then discuss how this API can be augmented to support complex user-defined types and extended to support addressing other types of remote memory, such as memory resident on GPUs. We then continue with an overview of the general user-side interface for constructing and manipulating distributed data structures as well two mechanisms for allowing user-guided control of data structure optimizations, *concurrency promises* and *buffers*.

3.1 RDMA-Based Communication Primitives

This chapter discusses the BCL Core, which is a set of RDMA-based communication primitives for implementing distributed data structures. The BCL Core provides a core abstraction of *global pointers*, which are pointer-like C++ objects that can be used to reference memory residing in the shared segment of another process. Users can interact with global pointers in a similar manner to how they would interact with regular pointers, such as dereferencing them to access individual data elements, issuing bulk data copies, or invoking atomic operations such as *compare-and-swap*. Operations on global pointers correspond directly to Remote Direct Memory Access (RDMA) primitives supported by most modern supercomputer interconnects, as well as in many commodity datacenter environments. This means that operations on global pointers can be executed directly in the network hardware, which helps reduce latency and prevents the need for direct synchronization between remote processes. In addition, the one-sided nature of RDMA network communication corresponds directly to the one-sided nature of most traditional data structures, which typically involve one thread of executing manipulating a data structure using operations like insertions, retrievals, pushes or pops.

RDMA-Based Network Communication Capabilities

Before discussing a specific software API, it is important to consider the RDMA hardware capabilities of modern Network Interface Cards (NICs). While the precise capabilities of different RDMA NICs varies, we can identify a common subset of RDMA operations supported by the vast majority of RDMA-supporting networks, as well as their general synchronization models and performance characteristics. When describing RDMA operations on remote memory, we will refer to two processes, the *local* or origin process that is issuing the RDMA operation and the *remote* process whose memory is being accessed. *Remote get* and *remote put* are the two most fundamental operations supported by RDMA.

Remote put operations copy a block of data from a local buffer into a region of a destination process' memory. Typically, one or more network packets are sent across the network to the remote NIC. As these packets are received at the destination, a DMA (Direct Memory Access) engine on the NIC copies them directly into DRAM memory. To signal to the origin process that the remote put operation has completed, the destination NIC must send a signal back to the origin, completing the roundtrip. The origin process can wait on this message in order to synchronize on the completion of the put operation. In order to access remote memory using RDMA operations, the memory must be registered with the NIC, allowing the NIC to directly read or write into that memory when it receives an RDMA request over the network. It is also important to note that before a put operation is completed, RDMA-based networks do not necessarily check the validity of data that arrives in the buffer, nor do they guarantee the order in which data may arrive in the buffer. This is because it is possible for packets to arrive out of order or even with corrupted data when they are initially copied into the destination buffer. Only when the operation fully completes is the data guaranteed to be fully copied.

Remote get operations copy data from the memory of a remote process into a local buffer on the origin process. Remote get operations typically operate by signaling to the remote NIC to issue a remote put operation copying the desired remote data back into a buffer on the local process. In order for the data to be copied back onto the origin process, the local receive buffer must be registered with the NIC. This can either happen on-demand at the time the remote get is issued, in which case the registration of the local output buffer will incur a fixed cost, or it can be done beforehand using a *bounce buffer* of pre-registered memory [78]. Typically, the bounce buffer is used for small, fine-grained transfers where the cost of an additional copy is minimal, while the local output buffer is registered for larger transfers where the cost of the registration is amortized by the time to copy the data over the network [77].

In addition to remote get and remote put operations, the vast majority of RDMA networks offer remote *atomic operations*, which perform a restricted set of read-modify-write operations atomically to locations in remote memory. These atomic operations include atomic operations common in the CPU domain such as compare-and-swap on 32 and 64 bit value, as well as basic integer arithmetic, including fetch-and-add, fetch-and-AND, fetch-and-OR, etc. on 32 or 64-bit integers. Some networks support a broader set of atomic or other synchro-

nization primitives, such as extended size atomics, which can operate on larger word sizes, or signaling puts, which signal when a put has completed by updating an additional memory location. However, due to a lack of portability, these extended atomics are not widely supported by communication libraries. Thus, in this work, we largely restrict ourselves to remote get, remote put, and 64-bit atomic operations.

Since RDMA get, put, and atomic operations can all be executed directly in the NIC, they can be completed with very low latency, typically on the order of 2 to 4 microseconds on modern supercomputer and commodity Ethernet networks. Atomic operations sometimes take slightly longer, particularly if many processes are modifying the same value. However, even under high contention, a single remote atomic operation is unlikely to take longer than 6-8 microseconds, even at high concurrencies. Note that this hardware latency of a single atomic operation is distinct from the overall latency of, for example, obtaining a lock if multiple retries over the network are required.

Small RDMA transfers, for example an RDMA put of 512 bytes or smaller, are entirely bound by latency. As the transfer size grows, however, RDMA transfers quickly achieve better performance, becoming bound by network bandwidth once reaching a transfer size of around 4 KB on Cray Aries networks, although the optimal message sizes do vary by network [24]. In microbenchmarks, RDMA-based communication libraries like GASNet-EX are able to reach peak bandwidth at lower message sizes compared to two-sided communication primitives [24].

An API for Building RDMA-Based Data Structures

One of the main goals of this work is to provide a collection of high-level data structures that are *cross-platform*, meaning that they can be used within programs written using a variety of different communication libraries. To accomplish this, BCL uses a small cross-platform internal API called the BCL Core. The BCL Core is supported by a number of different communication backends which implement a limited number of fundamental communication operations, typically using low-level communication libraries. These include an initialize function, which initializes the communication library including a shared segment of globally accessible memory on each process, as well as basic communication functions manipulating these shared segments such as remote puts, gets, and atomic operations. One of these communication backends is chosen at compile time using compiler directives. There are three primary advantages to this approach. First, supporting a variety of communication libraries aids integration of high-level data structures into an already-existing library. Since the BCL library only consists of code that calls functions in the backend communication library, BCL typically can be included as a header-only library without modifications to the compiler toolchain. Second, this allows more opportunity for portability and performance tuning for programs that only use BCL programs, since they can be compiled with any of the supported communication libraries. This is particularly useful when dealing with a collection of varied supercomputer or commodity cluster environments whose optimized communication libraries may not intersect. Third, implementing data structures on top of an internal DSL

allows for new communication libraries to be supported more easily, including both traditional distributed memory communication libraries as well as communication libraries that target new or experimental hardware, such as communication between GPUs or with custom hardware.

While the BCL Core is a full-fledged distributed programming environment and can be used to implement complex distributed programs, such as the data structures discussed in this work, users need not learn or use any of the BCL Core functionality discussed here—outside of the `BCL::initialize()`, `BCL::finalize()`, and `BCL::barrier()` functions—in order to use them. The main contribution of the BCL Core is that it provides a minimal set of communication functionality for implementing RDMA-based distributed data structures.

The BCL Core provides a high-level PGAS memory model based on *global pointers*, which are C++ objects that allow the manipulation of globally accessible regions of remote memory. During initialization of the BCL library, each process creates a *shared memory segment* of a fixed size, and throughout the program every process can read and write from or to any location within any process's shared segment using global pointers.

Global pointers are C++ objects that keep track of (1) the *rank*, or process ID number of the process on which the memory being pointed to is located, as well as (2) a particular offset within that process' shared memory segment. Together, these two parameters uniquely identify an address in global memory. Global pointers are regular, trivially copyable C++ objects and can be passed around between BCL processes using communication primitives, or indeed themselves be stored using global memory. Global pointers support pointer arithmetic operations, comparison, dereference, and other operators and methods, making them analogous to local pointers. They also fulfill the `std::random_access_iterator` concept in the C++ standard library.

An important limitation of global pointers is that they may only be used to reference *trivially copyable* objects, which are objects which can be copied by copying their byte level representation, as would be done with `memcpy`. If a type has a trivial copy constructor, it means we can byte copy that object into and out of remote memory. Types that do not have trivial copy constructors often use dynamic memory management or perform other operations that would cause their internal representations to be invalid if byte copied over the network. We discuss a low-overhead mechanism for storing non trivially copyable types using serialization in Section 3.3.

Global pointers can be used in a number of basic communication functions that implement primitives such as *remote get* and *remote put*, which read from or write to remote memory. They can also be used with atomic operations such as *fetch-and-op* (with support various operators such as addition, subtraction, and bitwise operations) as well as the more fundamental *compare-and-swap*.

Communication Primitives

Writing and Reading

The BCL Core's primary memory operations involve writing and reading to and from global pointers. *Remote get* operations read from a global pointer and copy the result into local memory, and *remote put* operations write the contents of an object in local memory to a shared memory location referenced by a global pointer. Remote gets and remote puts can be done with a number of different communication functions, including `BCL::memcpy()`, which functions similarly to the standard `std::memcpy()`, as well as various versions of `BCL::rput()` and `BCL::rget()`, which can transfer individual elements as well as arrays of data.

Memory Allocation

The BCL Core provides methods that allow each process to allocate and deallocate memory within its own shared segment. This is implemented by the BCL Core itself using the shared segment allocated by the BCL backend. *Remote memory allocation* is the ability for processes to asynchronously allocate memory resident in the shared segment of another process. In PGAS environments that also support remote procedure calls (RPC), remote memory allocation can be trivially supported by invoking an RPC on a remote process that then allocates memory in the desired shared segment, returning a global pointer to the newly allocated memory [16]. However, in the absence of RPC, adding support for remote memory allocation using only RDMA primitives must rely on remote atomics. This would require either adding a significant performance penalty for allocation operations performed in local shared segments or bifurcating memory to allow remote allocations using remote atomics in a special region, reducing the size of the largest possible memory allocation and fragmenting memory. In addition, freeing and compacting remote memory introduces significant challenges. Thus, this work does not assume remote memory allocation as a fundamental primitive, and instead relies on processes using coordination to communication pointers to remote memory. This is usually done using broadcast operations to communicate pointers to newly allocated memory during collective data structure construction or resize operations.

Synchronization and Memory Model

The BCL Core's communication primitives follow a relaxed consistency model shared by most low-level communication libraries, which allows remote put operations to be completed lazily in a "nonblocking" manner. When a remote put operation returns, the local buffer whose data is to be copied to a remote location is free to be modified without interrupting the operation. However, the data is not necessarily finished copying in place at the remote location until a barrier or flush operation has been completed, which serves as a memory fence and ensures that all previous remote memory operations are completed before the

program continues. Remote get operations are completed synchronously—the data is copied from the remote memory location to the local memory buffer immediately, and when the call returns the local memory buffer with the copied data is available to be read. Remote atomic operations are completed immediately, but do not necessarily force completion of any previous memory operations, necessitating the addition of a flush if an atomic is to signal the completion of some other remote memory operation.

Asynchronous Operations

BCL also supports asynchronous remote get, put, and atomic operations, which must be synchronized using individual request objects to ensure completion. Unlike regular remote put operations, when an asynchronous put operation returns, the local input buffer cannot be modified without interfering with the operation until after waiting on the operation's completion handle. Similarly, for remote get operations and atomic operations, the local output buffer is not guaranteed to be ready with the results of the operation until after waiting on the corresponding completion request. Request objects have two methods, `check`, which returns whether or not the request has been completed, and `wait`, which waits until the request has been completed. Since each communication library may have different synchronization handles and mechanisms, the request class is implemented by the backend. Each request handle implementation typically requires around 20 lines of code.

Building on top of asynchronous communication and these generic request handles, *futures* provide a way to return an object asynchronously without having to wait for all of the communication or computation involved in constructing that object to finish. In BCL, some data structures have asynchronous versions of some of their data structure methods, which return futures. When the object associated with a future is required, the method `get` can be invoked, which performs any necessary synchronization and returns the object. BCL provides a generic future implementation that can be used to return future to objects being asynchronously read over the network. This future implementation takes in a pointer to the object that is being read, along with any number of BCL requests that need to be completed before the object will be ready to be read. Some data structures also implement custom future types incorporating data structure-specific logic. For example, our hash table provides a future implementation that will asynchronously probe until the desired key is found.

Barrier and Flush

BCL applications can enforce synchronization using BCL barriers, which are both barriers and memory fences, forcing ordering of remote memory operations. In order for a process to enter a barrier, all of its memory operations must complete, both locally and at the remote target. In order for a process to exit a barrier, all other processes must have entered the barrier. Flush operations force the completion of any outstanding remote memory operations

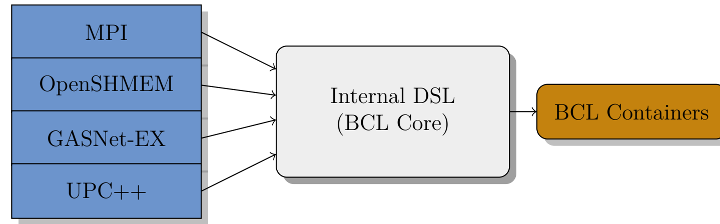


Figure 3.1: Organizational diagram of BCL.

issued by the calling process. Thus, a barrier contains an implied flush operation on each process.

Collectives

BCL includes the broadcast and allreduce collectives. Depending on the backend, these may be implemented using raw remote put and remote get operations, or, more likely, may map directly to high-performance implementations offered by the backend communication framework. In the work presented here, performance of these collectives is not critical, as they are mainly used for organizing and transporting pointers and control values.

Atomics

BCL's data structures avoid direct avoid coordination between CPUs, instead relying on *remote memory atomics* to maintain consistency. BCL backends must implement at least the atomic compare-and-swap operation, since all other atomic memory operations (AMOs) can be implemented on top of compare-and-swap [61]. However, backends will achieve much higher performance by directly including any atomic operations available in hardware. Other atomic operations provided by current BCL backends and utilized by BCL data structures include atomic fetch-and-add and atomic-fetch-and-or. We depend on backends to provide high quality interfaces to atomic operations as implemented in hardware now commonly available in both high-end supercomputers and commodity datacenters. When hardware atomics are not available, backends often provide atomic operation support through active messages or progress threads. This allows the application to run, but likely with higher overhead due to the slower performance of atomics.

3.2 Backends

BCL backends implement a limited number of communication primitives to provide support for the full BCL Core. These include an `init()` function which allocates symmetric shared memory segments of a requested size, the `barrier()`, `read()` and `write()` operations that

perform variable-sized reads and writes to global memory, at least an atomic compare-and-swap, and broadcast and reduce operations. For asynchronous operations, backends must implement a basic `request` class that provides a completion handle with a `wait` method. The BCL Core then implements futures using this basic `request` object. BCL backends typically require around 300 to 700 lines of code [1].

MPI

For our MPI backend, we use the MPI-3 one-sided communication interface [52]. Depending on the MPI implementation and available hardware, these remote memory access routines may be implemented using RDMA or shared memory, or emulated using two-sided communication if hardware support is unavailable. Remote get and remote put operations can be directly implemented using the `MPI_Get` and `MPI_Put` functions. The flush operation can be implemented using `MPI_Win_flush_all`, which forces the completion of all outstanding remote memory operations. A barrier consists of a flush followed by an MPI barrier. Atomic operations can all also be implemented using corresponding one-sided MPI functions.

OpenSHMEM

OpenSHMEM is a standardized interface for distributed memory communication offering one-sided remote memory operations, atomics, and collectives [34]. SHMEM operations map very closely to the low-level remote memory operations available in hardware. High-performance SHMEM implementations that comply with the OpenSHMEM standard include Cray SHMEM, Sandia OpenSHMEM, and Open MPI's SHMEM implementation [98].

GASNet-EX

GASNet-EX is a high-performance communication framework for supercomputers and has been used as a backend for many high-level PGAS libraries and languages, including UPC, UPC++, Co-Array Fortran, Legion, and Chapel [23].

3.3 BCL ObjectContainers

While global pointers themselves cannot be directly used to store non trivially copyable types in remote memory, the BCL Core includes a mechanism called BCL ObjectContainers that allows non trivially copyable types to be stored in distributed memory. ObjectContainers provide a transparent abstraction for storing complex data types in distributed memory with low overhead. BCL ObjectContainers are necessary because not all data types can be stored in distributed memory by byte copying. The common case for this is a struct or class, such as

[1] As reported by David A. Wheeler's 'SLOCCount,' which reports "physical lines of source code," not including blank lines and comments.

the C++ standard library's `std::string`, which contains a pointer. The pointer contained inside the class is no longer meaningful once transferred to another node, since it refers to local memory that is now inaccessible, so we must use some other method to serialize and deserialize our object in a way that is meaningful to remote processes. At the same time, we would like to optimize for the common case where objects *can* be byte copied and avoid making unnecessary copies.

Implementation

BCL ObjectContainers are implemented using the C++ type system. A BCL ObjectContainer is a C++ struct that takes template parameters `T`, a type of object that the ObjectContainer will hold, and `TSerialize`, a C++ struct with methods to serialize objects of type `T` and deserialize stored objects back to type `T`. BCL ObjectContainers themselves are of a fixed size and can be byte copied to and from shared memory. An ObjectContainer has a *set* method, which allows the user to store an object in the ObjectContainer, and a *get* method, which allows the user to retrieve the object from the container.

BCL includes a number of `TSerialize` structs for common C++ types, and these will be automatically detected and utilized by the BCL data structures at runtime. Users will usually not have to write their own serialization and deserialization methods unless they wish to use custom types which use heap memory or other local resources.

A finer point of BCL serialization structs is that they may serialize objects to either *fixed length* or *variable length* types. This is handled automatically at compile time by looking at the return type of the serialization struct: if the serialization struct returns an object of any trivially copyable type, then the serialized object is taken to be fixed size and is stored directly as a member variable of the serialization struct. If, however, the serialization struct returns an object of the special type `BCL::serial_ptr`, this signifies that the object is *variable length*, even when serialized, so we must instead store a global pointer to the serialized object inside the ObjectContainer.

User-Defined Types

To store user-defined types in BCL data structures, users can simply define serialization structs for their type and inject the struct into the BCL namespace. For trivially copyable types, this is unnecessary, since it will be automatically detected through the type system that a byte copy is sufficient for serialization. Users can then either serialize their objects into a trivially copyable object, as would be the case with a fixed-size list or other types that do not use dynamic memory allocation, or serialize their object into a dynamically sized `BCL::serial_ptr` buffer.

Copy Elision Optimization

An important consideration when using serialization is overhead in the common case, when no serialization is actually required. In the common byte-copyable case, where the serializa-

tion struct simply returns a reference to the original object, intelligent compilers are able to offer some *implicit* copy elision automatically. We have observed, by examining the assembly produced, that the GNU and Clang compilers are able to optimize away the unnecessary copy when a ObjectContainer object is retrieved from distributed memory, `get()` is called to retrieve the item lying inside. However, when an array of items is pulled together from distributed memory and unpacked, the necessary loop complicates analysis and prevents the compiler from performing this copy elision.

For this reason, BCL data structures perform *explicit* copy elision when reading or writing from an array of ObjectContainers stored in distributed memory when the ObjectContainer inherits from the `BCL::identity_serialize <T>` struct, which signifies that it is byte copyable. This is a compile-time check, so there is no runtime cost for this optimization.

3.4 Extension to GPUs

This model of pointers to remote memory can be extended to include other types of memory, such as memory residing on GPUs. In BCL, this is supported through the addition of *CUDA* pointers, which point to a location in *device memory*, that is memory residing on a GPU in CUDA C++. In BCL, we have implemented an additional, optional backend that uses the NVSHMEM communication library, which is an extension of the OpenSHMEM API to support GPU communication. NVSHMEM enables direct GPU-to-GPU communication using GPUDirect RDMA over Infiniband between GPUs on different nodes, or between GPUs on the same node over an in-node fabric like NVLink or PCIe.

3.5 Building Distributed Data Structures

Using global pointers that support asynchronous, one-sided access into remote memory, which can be executed using in hardware using RDMA, along with ObjectContainers, which augment remote pointers by allowing us to store complex, user-defined types, we have all the necessary components to build distributed data structures.

Data structures in this work are split into two categories: *remote* and *distributed*. Both types of data structures are globally visible within the context of a distributed execution; however, a remote data structure stores data only on a single process, and a distributed data structure distributes data amongst multiple processes. The primary example of a remote data structure within BCL is a queue, where multiple processes may push or pop data from the queue, but the queue itself lives on a single process.

Both remote and distributed data structures rely upon processes having *global knowledge* about the location and distribution of data within a data structure. Typically, data will be stored using a collection of global pointers, which point to different segments of data. In order to allow each process to independently manipulate the data inside the data structure, it will need access to those global pointers, which typically means each process having a local

array with copies of the global pointers to each segment of data. This helps reduce latency, as it allows a process to directly issue remote memory operations to manipulate the data structure, instead of first loading in global pointers from remote memory.

In order to ensure that each process has a global view of the data structures, data structures must be constructed and destructed *collectively*, meaning that every process within the distributed execution must call the constructor at the same time. Inside the constructor, the different segments of the distributed data structure are allocated, and remote pointers are broadcast to all the processes. In addition to construction and destruction, any data structure operation that reallocates data or modifies the layout of the data structure must be a collective operation. This is due to the global knowledge that processes must have. Since processes could attempt to access any part of the data structure at any time, the location of data cannot be changed without informing all processes. This includes, for example, `resize` or `reshape` operations, which must often reallocate and redistribute global pointers to newly expanded data segments.

All other data structure operations, including primary data structure operations such as inserts, lookups, pushes, or pops, are *noncollective*, and can be called by any process individually at any time. To perform a data structure operation, a process can directly issue remote memory operations that access remote data, and data structure methods will use atomics or locks to synchronize access where necessary.

3.6 Concurrency Promises

Many data structure operations have multiple possible implementations that use different levels of atomics or locking synchronization, trading off between the cost of coordination and the consistency level that can be guaranteed. If we have appropriate global knowledge about the *context* in which a particular data structure operation is issued (i.e., whether a particular invocation of `insert` may be called concurrently with `find`, or will only be coincide with other calls to `insert`), we can use that knowledge to select the fastest implementation that is guaranteed to be correct. This can allow for constant factor increases in performance, since an individual data structure operation may, for example, be reduced from 3 to 2 remote memory operations if a single synchronization operation can be elided.

Since we are implementing distributed data structures in the context of a software library, where we do not have the ability to perform static analysis or other compiler optimizations, it is not possible for us to automatically select the optimal implementations. However, it is possible to provide the user with the ability to provide *concurrency promises*, which are optional pieces of information about the context in which a particular data structure operation is being invoked. These are given in the form of a list of data structure methods that may be invoked concurrently by other processes during a method invocation. Generally, concurrency promises specify whether a data structure will undergo only reads, only writes, or both reads and writes during a particular phase. For example, with the remote queue data structure CircularQueue, users can optionally specify an extra parameter containing `push`,

to indicate only push operations will occur concurrent with an operation, `pop`, to indicate only pop operations, or `push | pop` to indicate both may occur. By default, data structure methods will use the implementation that supports the highest level of concurrency (and thus, typically the highest cost). Upon recognizing that a data structure invocation occurs within a specific context, such as that it occurs in a part of the program in which only other `insert` operations are being issued, or even in which only local `insert` operations are being issued, users can add concurrency promises as an additional optional argument. Concurrency promises are discussed in the chapters for their respective data structures, in Chapter 5 for queues and Chapter 6 for hash tables. Additional performance modeling of the different data structure methods for each promise level are discussed in Chapter 8.

3.7 Buffers

While concurrency promises can allow users to achieve constant factor speedups by removing unnecessary synchronization operations based on the context in which data structure operations are issued, reducing the latency of individual operations, the overall performance of an application that uses them will still be bound by network latency. While there is an important class of applications which require the lowest latency possible in order to perform asynchronous, real-time lookups, many applications also have phases during which operations do not necessarily have to complete immediately. One example of this is an *insertion* phase, in which all processes insert values into a distributed data structure in order to fill it with data, followed by a barrier before another phase begins. This type of pattern allows for the use of *aggregation*, where operations that need to be completed, instead of being issued in separate operations that would correspond to separate, latency-bound roundtrips over the network, can be bundled together in larger messages that can saturate the network bandwidth. When the operations are processed on the remote side, they can be performed using local memory operations, which are much higher performance. We discuss buffer data structures later along with the specific data structures that they support.

Chapter 4

Experimental Setup and Methodology

In this work, we examine the performance characteristics of various distributed data structures using a collection of applications and benchmarks. These applications and benchmarks are executed across a variety of distributed computing systems, each of which is equipped with different networks, processors, and accelerators. Here, we include a brief description of each of the compute systems used in benchmarks in this work, a summary of which is presented in Table 4.1.

Cori Phase I

Cori Phase I is a supercomputer system at the National Energy Research Center (NERSC) at Lawrence Berkeley National Laboratory. Each node of Cori Phase I is equipped with two Intel Xeon E5-2698v3 processors, each of which has 16 cores using the Haswell microarchitecture [84]. Each Cori node is equipped with 128 GB of DDR4 memory.

Cori nodes are connected with a Cray Aries network, with each node having an injection bandwidth of approximately 9.25 GB/s. The Cray Aries network uses a Dragonfly topology, which is a high-radix network that closely mimics the performance characteristics of a fully connected network topology, such as a fat tree [6]. Like most Cray Aries installations, Cori Phase I's network is missing some global fiber optic links to save cost, which means that at large scales the *routing bandwidth*, or bandwidth available to route messages between the two endpoints, is lower than the aggregate injection bandwidth. This means performance may be somewhat degraded at large scales for irregular all-to-all communication. On Cori

Name	CPUs	GPUs	Interconnect Type	Injection BW
Cori Phase I	Intel Xeon E5-2698v3 x2	-	Cray Aries	9.25 GB/s
AWS c3.4xlarge	Intel Xeon	-	Commodity Ethernet	1.25 GB/s
Summit	IBM POWER9 x2	Nvidia Tesla V100 x6	Mellanox EDR Infiniband	25 GB/s
Summitdev	IBM POWER8 x2	Nvidia Tesla A100 x6	Mellanox EDR Infiniband	25 GB/s

Table 4.1: Summary of systems used in evaluation throughout this work.

Phase I, the *global bandwidth*, which is the sum of all of the routing links, is equal to 2.41 GB/s per node. This means that for very large jobs with random communication patterns, we would expect achieved bandwidth per node to decrease below the injection bandwidth as the scale of the job increases. This is because unlike in a full fat tree network, there is not enough routing bandwidth to sustain the injection bandwidth at scale for all communication patterns.

Amazon Web Services

Amazon Web Services Elastic Compute (AWS EC2) is a cloud service providing on-demand access to commodity datacenter servers. AWS servers are available in a number of different configurations, including different processor, memory, accelerator, and network configurations. In some of our experiments, we use AWS EC2 c3.4xlarge servers, which are advertised as compute-oriented servers connected by a commodity Ethernet network. Each node is equipped with an Intel Xeon E5-2680 v2 processor with 16 cores [7]. The nodes are connected by a commodity Ethernet network, with each node's network interface card (NIC) capable of 1.25 GB/s of injection BW.

Amazon does not publish the topology of the network used by c3.4xlarge or most other AWS EC2 server types. However, upon personal communication with AWS staff, we were informed that users should expect the network to exhibit fat tree topology-like behavior with a high amount of routing bandwidth, assuming all servers are requisitioned using the same placement group using the *cluster* placement strategy, which attempts to place all servers in an allocation close together. This was done for all of our AWS experiments.

Summit

Summit is a supercomputer system at the Oak Ridge Leadership Computing Facility (OLCF) at Oak Ridge National Laboratory (ORNL). Each node of Summit is equipped with two 22 core IBM POWER9 processors, along with 6 Nvidia Tesla V100 GPUs [87]. Each node's CPUs share 512 GB of DDR4 memory, while each GPU is equipped with 16 GB of DDR5 memory.

Summit's nodes are connected by a Mellanox EDR Infiniband network, with each node equipped with two Network Interface Cards (NICs), each with an injection bandwidth of 12.5 GB/s for a combined injection bandwidth of 25 GB/s per node. Summit's nodes are connected using a fully connected fat tree topology.

Summitdev

Summitdev was an experimental supercomputer system at the Oak Ridge Leadership Computing Facility (OLCF) at Oak Ridge National Laboratory (ORNL). Summitdev served as a precursor system to Summit, with very similar hardware, but a smaller scale. Each

node of Summitdev was equipped with two IBM POWER8 processors, along with 6 Nvidia Tesla P100 GPUs [9].

Like Summit, Summitdev's nodes were connected by Mellanox EDR Infiniband network in a fully connected fat tree topology. Each node was equipped with two NICs, each with an injection bandwidth of 12.5 GB/s for a combined injection bandwidth of 25 GB/s per node.

Chapter 5

RDMA-Based Queues

Queues are important data structures for a variety of applications. In a parallel computing context, they are commonly associated with a number of different computational patterns, including producer-consumer task management, where each process has a queue of tasks it needs to complete and new tasks are pushed onto the appropriate process's queue when new work is produced [4]; frontier-based graph algorithms [38], where queues are used to control the ordering of vertices visited; and data redistribution, where processes can redistribute data to other processes by pushing values onto their queues [50]. One of the key advantages of using RDMA-based queues is that they can be highly asynchronous. A process can reserve a spot in the queue to write its data and then allow that data to be sent over the network while it attends to other computation. This allows for overlap between the computation that is generating the data to be redistributed and the queue communication, unlike a bulk synchronous all-to-all operation, where all the data must be ready at the start of the operation. Section 5.1 discusses more details of using queues for all-to-all communication patterns.

When designing a distributed memory queue, there are two families of queue designs to consider from the shared memory programming literature, linked list-based queues and fixed-size ring buffer queues [62]. The first design has the advantage of supporting queues of unbounded length. However, linked list-based queues have several disadvantages when it comes to a distributed memory implementation.

First, linked list-based queues traditionally rely on the ability to perform atomic compare-and-swap on pointers, which can be used to atomically append a new element to the queue. While this is relatively easy and inexpensive to perform on just about any shared memory CPU architecture, since pointers are usually 64 bits, within the word limitations of CPU atomic operations, remote pointers are typically larger than 64 bits. There are methods to enable compare-and-swap operations in distributed memory, but they all come with compromises. Some recent RDMA NICs support extended size atomics, enabling compare-and-swap operations on values larger than 64 bits [68]. However, these operations are non-standard and are currently not supported by major communication libraries, leading to a loss of portability. Some PGAS programming environments, such as UPC and UPC++, have special modes

that shrink the size of global pointers to 64 bits, thus enabling atomic compare-and-swap operations on global pointers, but this also requires shrinking both the maximum amount of memory and the maximum number of processes that can be supported [15, 58]. It is possible to emulate compare-and-swap operations using a lock, but this comes with significant performance limitations, since it quadruples the latency in the best case (requiring four roundtrips instead of one: lock, read, write, unlock) and in the presence of contention results in even worse slowdowns.

The second disadvantage of link list-based queues is that, without the ability to allocate memory in the shared segment of other processes, the actual nodes of the queue will likely be located on a number of different processes. While this haphazard placement could potentially be seen as an advantage for some use cases, in practice, it prevents queues from being used to redistribute data to a particular process. It also introduces difficulties when it comes to memory management, as a process that pops a node off of the queue may now need to deallocate memory located in the shared segment of another process.

For this reason, we focus on fixed-size ring buffer queues in this chapter. In a ring buffer queue, queues have a fixed maximum capacity, the size of the buffer, and the actual range of elements within the buffer that hold queue elements are marked with integers: `head`, which holds the index of the first element stored in the queue, and `tail`, which points to one slot past the last element held in the queue. In order to push elements onto the queue, `tail` must be incremented in order to allocate space for them, and then a remote put operation will write the actual data into the buffer. In order to pop elements from the queue, `head` must be incremented, and then a remote get will read the popped elements from the queue.

One of the central issues around maintaining correctness with distributed ring buffer queues concerns how to ensure that data that is being written to the queue will not be read until the write has fully completed. If two processes were simultaneously pushing and popping data to and from a queue using only the reservation process described above, the popping process would be free to read data from the queue as soon as it has finished its fetch-and-add operation. Using the `head` and `tail` pointers, the popping process can only see whether a push has begun, not whether the data is fully written in place. This could result in the reading process reading invalid, half-written data.

As previously discussed, different applications have different demands from remote queues. When using queues to perform work stealing, work pushing, or other queue-based task management, it is crucial for a queue to support concurrent push and pop operations correctly and efficiently [38]. In contrast, when queues are being used to redistribute data in dedicated phases, where barriers are commonly used to ensure completion, simultaneous push and pop operations need not be supported, and the synchronization operations in a queue implementation that does support simultaneous pushes and pops only introduce unnecessary overhead. In addition, there are multiple techniques that can be used for ensuring synchronization between pushing and popping processes that may be more suitable for different use cases.

All queues in BCL are implemented as ring buffer queues. There is one large data buffer, which holds the data that will be inserted into the queue, and this data buffer sits on a

single process, which is designated when the queue is created. Queues are thus *remote* data structures in BCL, so while a queue is globally visible to all processes, it is resident on only one process at a time. In order to signal where data is written in the queue and to ensure synchronization, a globally accessible *head* and *tail* integers mark the locations in the buffer where data elements are stored. To push and pop data from the queue, processes perform atomic operations on these integers and remote get and put operations on the data buffer. Depending on the specific queue implementation, there may be additional requirements for synchronization.

BCL contains three different types of remote queues suitable for different situations. `CircularQueue` supports concurrent push and pop operations from multiple processes. To support this, it requires a higher level of synchronization, which comes at a higher cost, although this can be somewhat mitigated using concurrency promises, discussed in Section 5.2, which can disable some of the synchronization when it is not needed. By contrast, `FastQueue` supports only simultaneous pushes *or* concurrent pops. This queue specifically targets data redistribution, where data is sent in distinct phases partitioned by global barriers. It minimizes synchronization by relying on barrier synchronization to ensure data is finished writing before any pop operations are issued. As a result, it is able to achieve significant performance improvements over `CircularQueue`. Finally, `ChecksumQueue`, is able to offer most of the same guarantees as `CircularQueue`, but at the cost of increased space and computation to maintain a checksum value for each element pushed to queue.

`FastQueue`

We begin our detailed discussion of queue implementations with `FastQueue`, since it is the simplest, and is built upon by the other queues, which essentially add additional synchronization to the same framework. `FastQueue` uses three shared objects: a data segment, where queue elements are stored; a shared integer that stores the head of the queue; and a shared integer that stores the tail of the queue. To insert, a process first increments the tail using an atomic fetch-and-add operation, checks that this does not surpass the head pointer, and then inserts its value or values into the data segment of the queue. An illustration of a push operation is shown in Figure 5.1. In general, the head overrun check is performed without a remote memory operation by caching the position of the head pointer, so an insertion requires only two remote memory operations. We similarly cache the location of the tail pointer, so pops to the queue usually require only one atomic memory operation to increment the head pointer and one remote memory operation to read the popped values.

`CircularQueue`

To support concurrent reads and writes, `CircularQueue` has an additional set of head and tail pointers which indicate which portions of data in the queue are ready to be read. As previously mentioned, there are multiple implementations of push and pop for a `CircularQueue` data structure, as listed in Table 5.2.

Data Structure	Method	Collective	Description	Cost
`BCL::CircularQueue`				
	`bool push(const T &val)`	N	Insert item into queue.	$2A + W$
	`bool pop(T &val)`	N	Pop item into queue.	$2A + R$
	`bool push(const std::vector <T> &vals)`	N	Insert items into queue.	$2A + nW$
	`bool pop(std::vector <T> &vals, size_t n)`	N	Pop items from queue.	$2A + nR$
	`bool local_nonatomic_pop(T &val)`	N	Nonatomically pop item from a local queue.	ℓ
	`void resize(size_t n)`	Y	Resize queue.	$B + \ell$
	`void migrate(size_t n)`	Y	Migrate queue to new host.	$B + nW$
`BCL::FastQueue`				
	`bool push(const T &val)`	N	Insert item into queue.	$A + W$
	`bool pop(T &val)`	N	Pop item into queue.	$A + R$
	`bool push(const std::vector <T> &vals)`	N	Insert items into queue.	$A + nW$
	`bool pop(std::vector <T> &vals, size_t n)`	N	Pop items from queue.	$A + nR$
	`bool local_nonatomic_pop(T &val)`	N	Nonatomically pop item from a local queue.	ℓ
	`void resize(size_t n)`	Y	Resize queue.	$B + \ell$
	`void migrate(size_t n)`	Y	Migrate queue to new host.	$B + nW$
`BCL::ChecksumQueue`				
	`bool push(const T &val)`	N	Insert item into queue.	$A + W(1 + c)$
	`bool pop(T &val)`	N	Pop item into queue.	$A + R(1 + c)$
	`bool push(const std::vector <T> &vals)`	N	Insert items into queue.	$A + nW(1 + c)$
	`bool pop(std::vector <T> &vals, size_t n)`	N	Pop items from queue.	$A + nR(1 + c)$
	`bool local_nonatomic_pop(T &val)`	N	Nonatomically pop item from a local queue.	ℓ
	`void resize(size_t n)`	Y	Resize queue.	$B + \ell$
	`void migrate(size_t n)`	Y	Migrate queue to new host.	$B + nW(1 + c)$

Table 5.1: A selection of methods from BCL queues. Costs are best case, using implementation chosen with no concurrency promises. R is the cost of a remote read, W the cost of a remote write, A the cost of a remote atomic memory operation, B the cost of a barrier, ℓ the cost of a local memory operation, and n the number of elements involved. c is the number of bytes in a checksum divided by the number of bytes per element (usually ≤ 1).

Method	Concurrency Promise	Description	Cost
`push`			
(a)	`push | pop`	Fully Atomic	$2A + W$
(b)	`push`	Only Pushes	$2A + W$
(c)	`local`	Local Push	ℓ
`pop`			
(d)	`push | pop`	Fully Atomic	$2A + R$
(e)	`pop`	Only Pops	$2A + R$
(f)	`local`	Local Pop	ℓ

Table 5.2: Implementations of data structure operations for BCL's circular queue data structure.

Push and Pop Operations

The default fully atomic implementation used to push (Table 5.2a) to a `CircularQueue` data structure involves 2 atomic operations and a remote put operation with a flush. First, we issue a fetch-and-add operation to increment the tail pointer, then write the data to the queue and flush it. Finally, we must perform a compare-and-swap operation to increment the "tail ready" pointer, indicating that the pushed data is ready to be read. A compare-and-swap (CAS) operation is specifically necessary for the final step because we must ensure that all previous pushes are also complete before we modify the "tail ready" pointer. A fetch-and-add operation could increment the ready pointer to mistakenly mark other processes' writes as ready to be read. In the case where no pop operations will be performed concurrently with pushes, we may perform the final atomic increment using a fetch-and-add (Table 5.2b). Analogous implementations are used for pop operations (Table 5.2d and 5.2e).

Both queues support resizing as well as migrating to another host process, both as collective operations. We evaluate the performance of our circular queue data structures in Section 5.3.

`ChecksumQueue`

An alternative mechanism to using an atomic compare-and-swap operation to signal that the pushed data is ready to read—since using CAS operations in this way can cause contention when the operation must be repeatedly retried—is to use a checksum. Instead of using an additional atomic operation to signal that the pushed data is ready to be read, processes write a checksum alongside each element being pushed. When the popping process desires to pop an element, it must check the checksum corresponding to the element being popped. This use of the checksum removes the need for an additional atomic operation, at the cost of (1) additional data movement and computation, since a checksum must now be computed and written for every data element, and (2) increased cost of popping, since the popping process must now perform a checksum to verify each element has been successfully written before popping. However, most importantly, the checksum-based queue is able to reduce the cost associated with repeated compare-and-swap attempts in the circular queue, which greatly reduces the cost of a push operation in the presence of contention.

Since a ring buffer design is being used, if the queue wraps around, popping processes may see old data that was previously written, including a correct checksum. In order to prevent popping processes from perceiving this data as ready to be popped, it is necessary to use a different hash function each time the queue is rounded. This may be done by dividing the slot number by the size of the queue to calculate the round, and if the round is odd, hashing the value twice. This results in old data always having an incorrect checksum value, meaning that popping processes will never verify old data as correct.

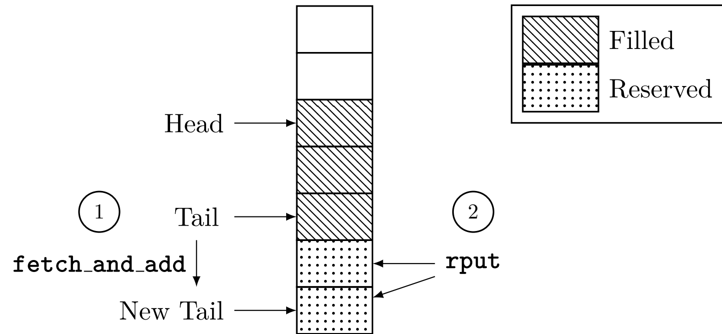


Figure 5.1: Process for pushing values to a BCL `FastQueue`. First (1) a `fetch_and_add` operation is performed, which returns a reserved location where values can be inserted. Then (2) the values to be inserted are copied to the queue.

Advantages of FastQueue

`FastQueue` can only be used in cases where there are phases of only pushes and only pops. As discussed previously, this special case is particularly important when it comes to data redistribution. While limited in this way, `FastQueue` has the advantage of requiring one fewer atomic memory operations per push or pop. While `CircularQueue` does support disabling some of its synchronization when it is not needed, allowing the final pop to be a single non-blocking fetch-and-add operation, it still introduces additional, unnecessary coordination overhead. We felt that phasal pushes and pops were important enough to warrant a separate version of the data structure, since queues that support only multi-reader and multi-writer *phases* are crucial to multiple algorithms.

5.1 Asynchronous All-to-All Pattern

BCL circular queues provide a straightforward, object-oriented way to asynchronously redistribute data in a distributed setting. This can be accomplished by placing a queue on each node, then reading and writing into the appropriate queues to redistribute data to other nodes. This achieves an effect similar to MPI's all-to-all collective operation, but is asynchronous and avoids all-to-all's bulk synchronous coordination, instead using one-sided operations. Newer versions of MPI do include asynchronous all-to-all operations; however, this new version still requires coordination: all processes must enter the all-to-all operation to initiate it, and a process may not introduce new data in the middle of an asynchronous all-to-all operation already underway. Performing data redistribution with BCL queues allows for greater overlap of communication and computation, since processes can overlap asynchronous queue insertions with sorting operations and other computation that may create new data to push to other processes. Figure 5.2 demonstrates how BCL queues can be used to overlap communication and computation in the data redistribution phase of a bucket sort.

```
1 auto sort(const std::vector<int>& data) {
2   std::vector<std::vector<int>> buffers(BCL::nprocs());
3
4   std::vector<BCL::FastQueue<int>> queues;
5   for (size_t rank = 0; rank < BCL::nprocs(); rank++) {
6     queues.push_back(BCL::FastQueue<int>(rank, queue_size));
7   }
8
9   for (auto& val : data) {
10    size_t rank = map_to_rank(val);
11    buffers[rank].push_back(val);
12    if (buffers[rank].size() >= message_size) {
13      queues[rank].push(buffers[rank]);
14      buffers[rank].clear();
15    }
16  }
17
18  for (size_t i = 0; i < buffers.size(); i++) {
19    queues[i].push(buffers[i]);
20  }
21
22  BCL::barrier();
23
24  std::sort(queues[BCL::rank()].begin().local(),
25            queues[BCL::rank()].end().local());
26  return queues[BCL::rank()].as_vector();
27 }
```

Figure 5.2: Our bucket sort implementation in BCL for the ISx benchmark.

Later, we will show how this leads to better performance on the ISx bucket sort benchmark.

5.2 Concurrency Promises

As we have discussed specifically for `CircularQueue`, distributed data structure operations often have multiple alternate implementations, using varying levels of coordination. Depending on the *context* in which an operation is issued, meaning which other types of operations may occur concurrently with it, more or less coordination may be required. The common example of this which we have already discussed is *phasal* operations, where data structures are manipulated in separate phases, each of which is separated by a barrier. Fig-

ure 5.2 demonstrates a phasal queue operation, and other phasal operations are discussed in Chapter 6. Crucially, the barriers associated with phasal operations provide a guaranteed synchronization point between different types of operations that. For example, in a push phase, we are guaranteed that a barrier will take place before any pop operations are issued. This often allows the use of implementations that use less synchronization. As we have discussed, `CircularQueue` is able to convert a compare-and-swap operation to a more contention-tolerant fetch-and-add operation, depending on the concurrency promise given. In Chapter 6, we also discuss how find operations in a hash table can be executed with a more optimized implementation—a single remote get operation, rather than 2 AMOs and a remote get—when we are guaranteed that only find operations will be executed in the same barrier region.

To take advantage of optimized implementations using concurrency promises, users optionally provide a list of data structure operations that can take place concurrently with the operation being issued. For example, if the user is calling the `push` method on `CircularQueue`, they could specify the concurrency promise `push` to specify that only push operations will occur simultaneously with this call to `push`, or they could specify `push | pop` to specify that either push or pop operations may occur. The data structure will then automatically select the best implementation that will be correct based on the user-provided concurrency promise.

Within a dedicated programming language that can perform compiler passes to apply optimizations using global program knowledge, many of these optimizations could in principle be applied automatically. Indeed, some parallel programming languages such as Chapel [71] do provide static optimizations of this kind, although they are currently focused on lower-level operations like reads and writes into dense arrays. Since BCL is a library and not a programming language, it is not possible to perform automatic optimizations based on global code knowledge that would determine static analysis to discover. Thus, we require user input to give us this static knowledge directly, which we can then use to apply the optimizations.

5.3 Experimental Evaluation

We evaluated the performance of BCL's queue data structures using ISx, an integer sorting mini-application, as well as a collection of microbenchmarks. We tested these benchmarks across three computing systems, including Cori Phase I, Summitdev, and AWS. The details of these system's configurations are discussed in Chapter 4. On Cori, experiments are performed up to 512 nodes. On Summitdev, experiments are performed up to 54 nodes, which is the size of the whole cluster. On AWS, we provisioned a 64 node cluster and performed scaling experiments up to its full size.

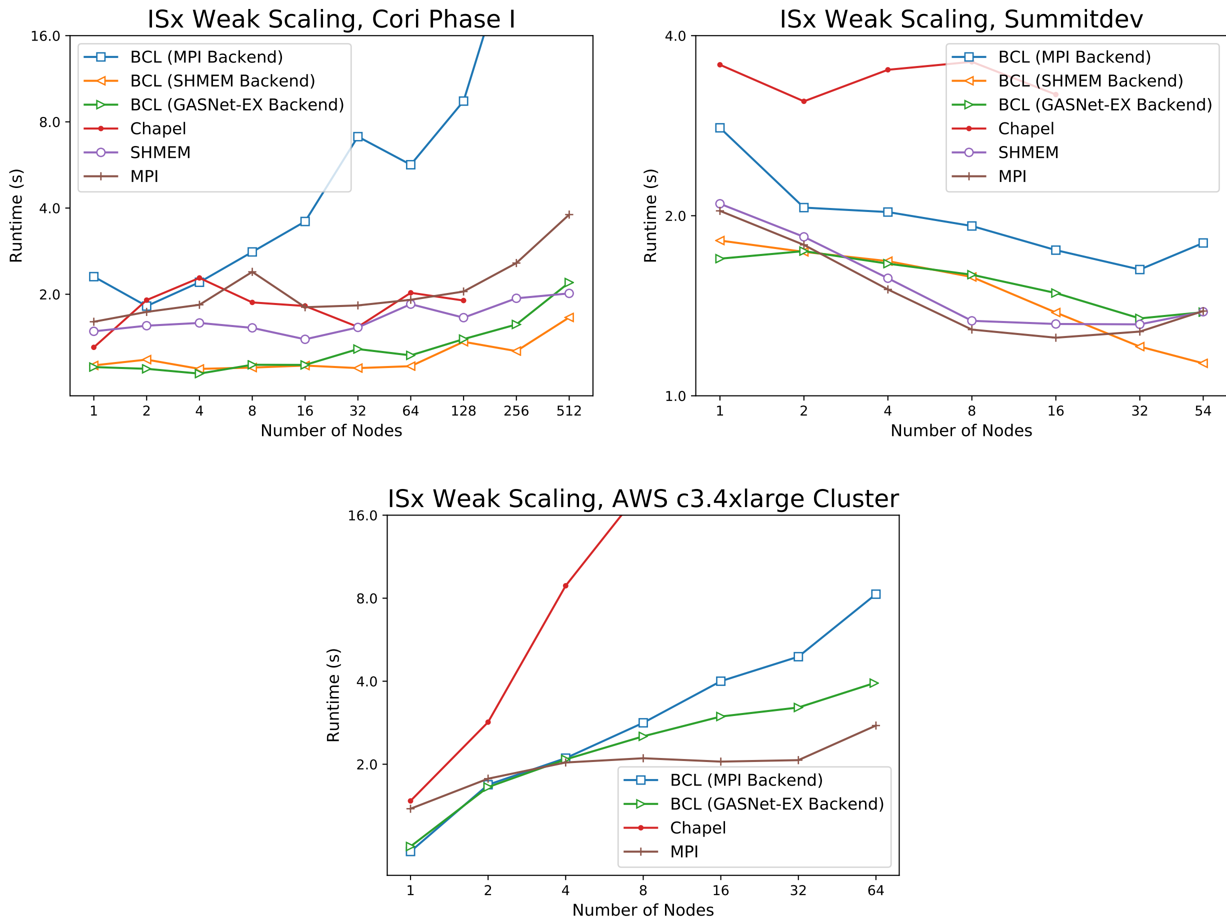


Figure 5.3: Performance comparison on the ISx benchmark on three different computing systems. All runs measure weak scaling with 2^{24} items per process.

ISx Benchmark

To test out our queue's performance, we implemented a bucket sort algorithm to execute the ISx benchmark [57]. The ISx benchmark is a simple bucket sort benchmark performed on uniformly distributed data. It consists of two stages, a distribution stage and a local sort stage. In the distribution stage, processes use pre-existing knowledge about the distribution of the randomly generated data to assign each key to be sorted to a bucket, where there is by default one bucket on each node. After this stage, each process simply performs a local sort on its received data. The original ISx benchmark comes with an MPI implementation, which uses an all-to-all collective for the distribution stage and an OpenSHMEM implementation, which sends data asynchronously. An implementation in Chapel, a high-level parallel programming language, has also been published [60, 33].

BCL Implementation

We implemented our bucket sort in BCL using the circular queue data structure. First, during initialization, we place one circular queue on each process. During the distribution phase, each process pushes its keys into the appropriate remote queues. After a global barrier, each node performs a local sort on the items in its queue. During the distribution phase, we perform *aggregation* of inserts to amortize the latency costs of individual inserts. Instead of directly pushing individual items to the remote queues, we first place items in local buffers corresponding to the appropriate remote queue. Once a bucket reaches a set message size, we push the whole bucket of items at once and clear the local bucket. It's important to note that this push is *asynchronous*, meaning that the communication involved with pushing items to the queue can be overlapped with computation involved with sorting the items. The fact that BCL circular queue's push method accepts a vector of items to insert simultaneously makes adding aggregation very straightforward. Even with this optimization, our full BCL sorting benchmark code, including initialization and timing, is only 72 lines long, compared to the original MPI and SHMEM reference implementations at 838 and 899 lines, and the Chapel implementation at 244 lines. A slightly abbreviated version of our implementation is listed in Figure 5.2.

Analysis

As shown in Figure 5.3, our BCL implementation of ISx performs competitively with the reference and Chapel implementations.

On the Cray Aries systems, BCL outperforms the other implementations. This is because BCL is able to overlap communication with computation: the asynchronous queue insertions overlap with sorting the values into buckets. This is an optimization that would be much more complicated to apply in a low-level MPI or SHMEM implementation (the reference implementation uses all-to-all patterns), but is straightforward using BCL's high-level interface.

There is an upward trend in the BCL scaling curves toward the high extreme of the graph on Cori. This is because as the number of processes increases, the number of values sent to each process decreases. At 512 nodes with 32 processes per node, each process will send, on average, 1024 values to each other process. With our message size of 1024, on average only one push is sent to each other process, and the potential for communication and computation overlap is much smaller, thus our solution degenerates to the synchronous all-to-all solution, and our performance matches the reference SHMEM implementation. Note that performance with the MPI backend is poor on Cori; we believe this is because the MPI implementation is failing to use hardware atomics.

The performance on Summitdev is similar, except that there is a slight downward trend in all the scaling lines because of cache effects. As the number of processes increases, the keyspace on each node decreases, and the local sort becomes more cache efficient.

Historically, PGAS programs have not fared well on Ethernet clusters, since PGAS pro-

grams often rely on fast hardware RDMA support. With our BCL implementation, we are able to increase the message size to amortize the cost of slow atomic operations. While our performance on AWS does not scale as well as the reference MPI implementation, we consider the performance acceptable given that it is a high-level implementation running in an environment traditionally deemed the exclusive domain of message-passing. On the Ethernet network, the GASNet-EX backend, which is using the UDP conduit, performs better than the MPI backend, which is using Open MPI. The native MPI implementation likely performs better than our BCL version for two reasons. The first reason is that on the Ethernet network, which has significantly less bandwidth than the high-performance networks, communication is much more expensive, which means that the bucketing computation constitutes a much smaller portion of the runtime. This means that the computation/communication overlap enabled by the asynchronous RDMA queues has less of a benefit. Second, the Ethernet network has a higher inherent latency, and since it does not support RDMA, our communication library emulates it, which introduces even more latency. This generally results in higher overheads for using an RDMA-based approach.

Microbenchmarks

We prepared a collection of microbenchmarks to compare (1) different backends' performance across data structure operations and (2) the relative performance of different implementations of data structure operations. Each benchmark tests a single data structure operation.

The queue microbenchmarks presented in Figures 5.4, 5.5, and 5.6 are split into two categories: (1) benchmarks in which all all processes operate upon a single queue, living on a single process, and (2) benchmarks in which there are many queues, specifically one every process, (latter benchmarks labeled "many"). In general, we expect the single queue benchmarks to achieve lower throughput than the "many" benchmarks. This is primarily because of synchronization overheads. In the single queue benchmarks, all processes are vying to perform pushes on a single queue. This increased contention results in greater overheads, particularly when the level of synchronization used is high, as in the `*pushpop` benchmarks. When operations are spread out over many queues, contention is lowered, and with it the associated synchronization overheads.

In the `CircularQueue` benchmarks, we see that fully atomic operations (`pop_pushpop` and `push_pushpop`) are quite expensive when all processes are inserting into a single queue, compared to `pop_pop` and `push_push`, which perform less coordination. This is unsurprising, since the final compare-and-swap (CAS) operation in an atomic `CircularQueue` push or pop essentially serializes the completion of operations. Processes must continuously attempt to update the `reserve_tail` pointer using a CAS operation, which can only be completed once the processor before it has completed its push. When pushing to multiple queues, this contention is greatly decreased, although using the version of the operation that performs less coordination results in a factor of two performance improvement (`push_many_pushpop` vs. `push_many_push`).

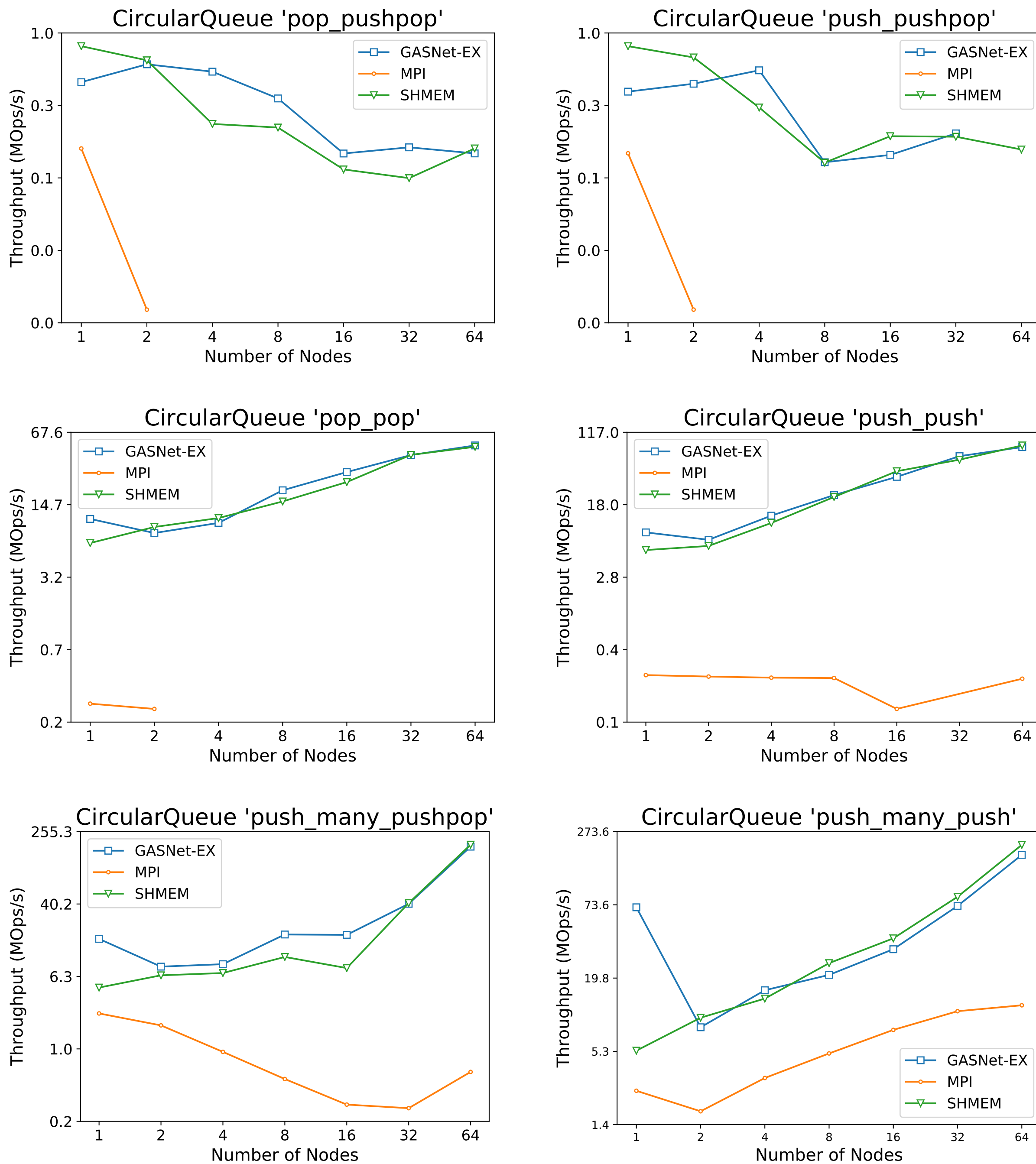


Figure 5.4: Microbenchmarks for CircularQueue. Benchmarks with "many" in the title are performed with one queue per node, while benchmarks without "many" are performed with all processes accessing a single queue. The label before the underscores indicates the operation performed, while the label after the underscore indicates the concurrency promise used. For example, "pushpop" indicates the concurrency promise `push | pop`.

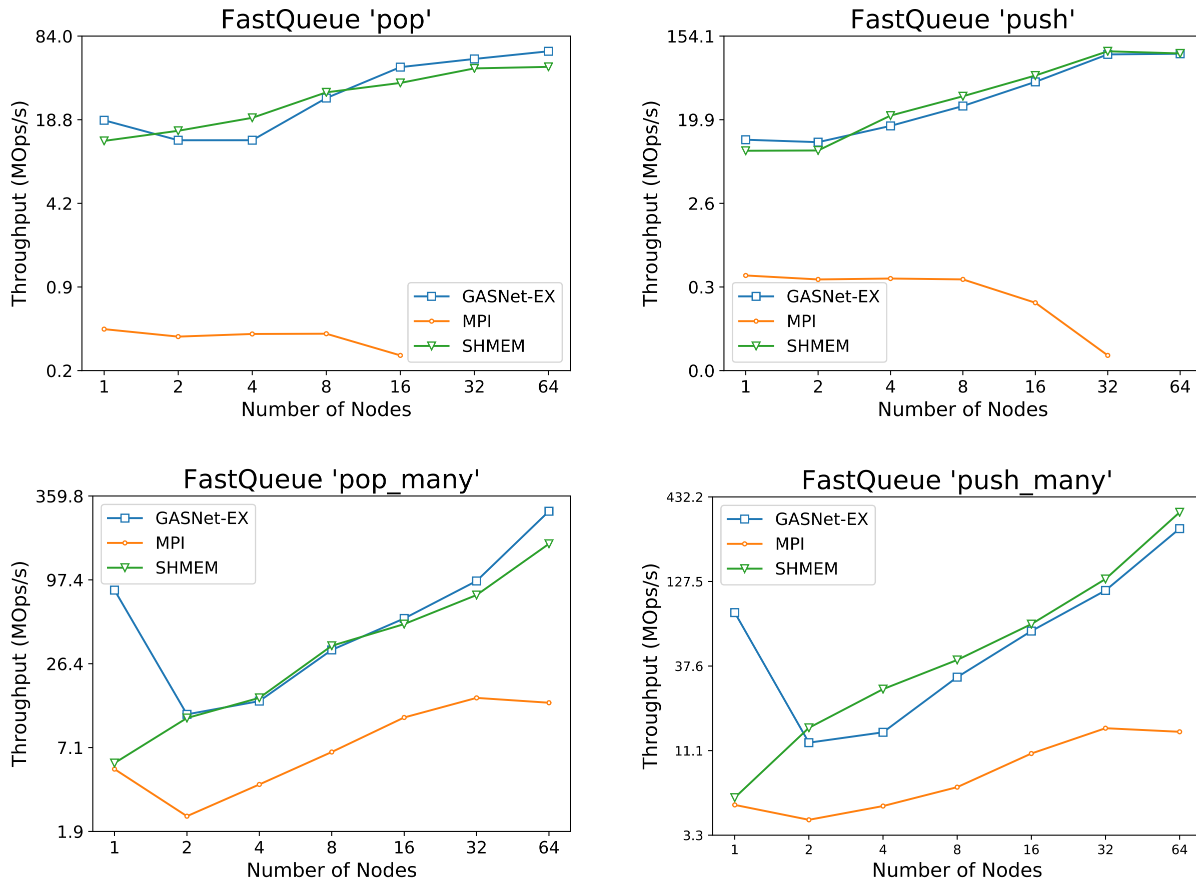


Figure 5.5: Microbenchmarks for FastQueue. Benchmarks with "many" in the title are performed with one queue per node, while benchmarks without "many" are performed with all processes accessing a single queue.

Looking at the `FastQueue` benchmarks, we see that this data structure does achieve a significant performance improvement even over the less-atomic implementations of data structure operations in `CircularQueue`. In particular, `FastQueue` achieves reasonable performance even when all processes are operating on a single queue, compared to `CircularQueue`, which slows two orders of magnitude when faced with that much contention. `ChecksumQueue` strikes a balance between the two queues. It is faster than `CircularQueue` across the board, but, like `FastQueue`, makes the best performance improvements on the single queue benchmarks. `ChecksumQueue` has slightly lower performance than `FastQueue`, but comes close to its performance on most benchmarks.

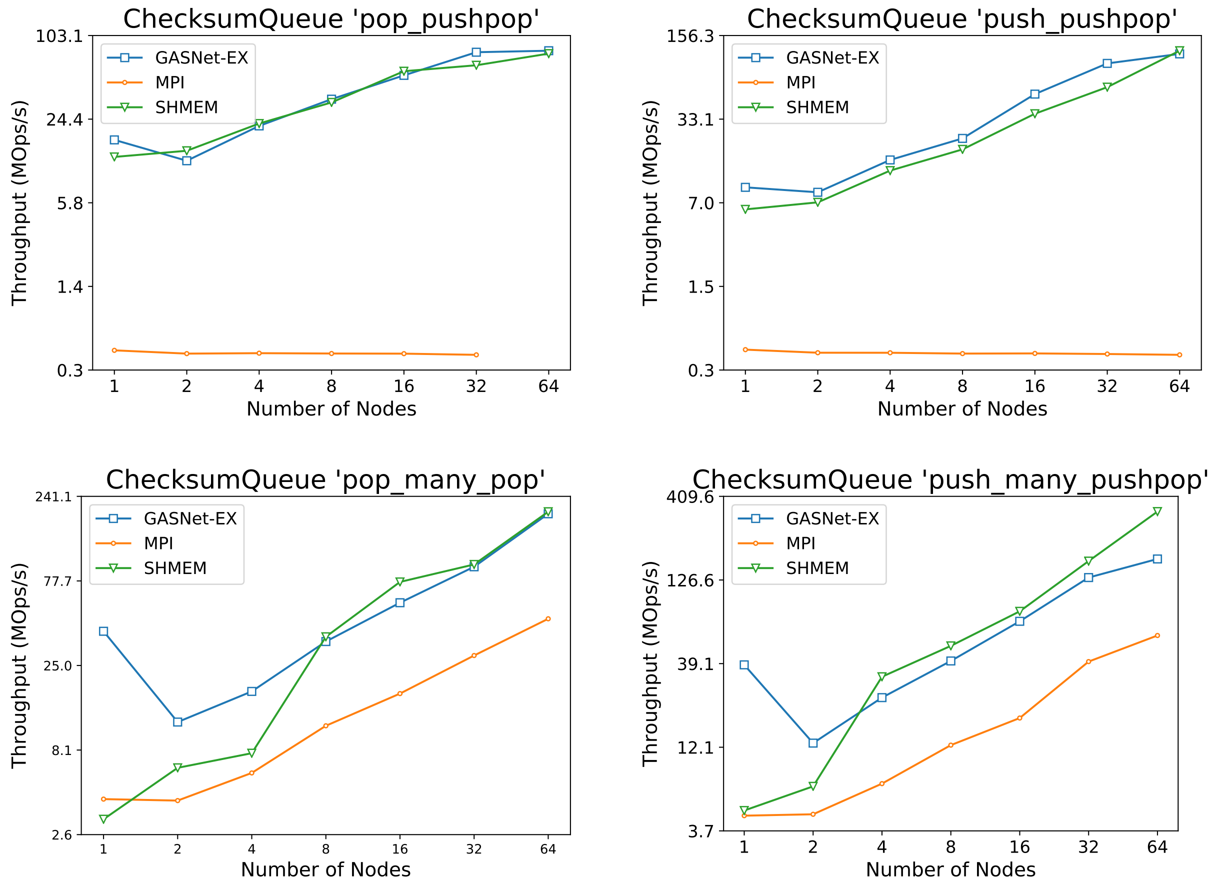


Figure 5.6: Microbenchmarks for ChecksumQueue. Benchmarks with "many" in the title are performed with one queue per node, while benchmarks without "many" are performed with all processes accessing a single queue. ChecksumQueue does not support concurrency promises, since the checksum must always be calculated and written for a push to complete successfully. However, the label after the underscore indicates the concurrency promise level that could be supported by the implementation for easier comparison with the CircularQueue plots in Figure 5.4.

Chapter 6

Set Data Structures

In this chapter, we discuss set data structures. Set data structures keep track of a set of unique keys and can typically insert or remove keys from the set as well as answer queries about set membership. In this chapter, we explore two kinds of set data structures: hash tables, which store a collection of keys, each with a matching value, and Bloom filters, which store a collection of keys and can answer questions about set membership with some false positive rate. Both hash tables and Bloom filters are able to make extensive use of remote memory atomics to achieve very low latency in operations like insert and find. Our hash table and Bloom filter design are both centered around providing the lowest latency possible in the best and average case. In cases where consistency can be relaxed, the user can provide information about the context in which an operation is issued to elide some of the remote memory operations. When consistency can be relaxed even further for bulk operations, we demonstrate that high-level buffer objects can use queues to aggregate fine-grained operations that would each require a small, latency-bound roundtrip message into large bulk messages bundling together multiple requests.

6.1 Hash Tables

Hash tables are associative data structures used in a broad variety of applications, including graph analytics, data analysis, and genomics. Hash tables are often used when data will need to be queried in an irregular or unsorted fashion. In a distributed context, hash tables can be used to create a balanced distribution of data elements to processes in a distributed execution, relying on the statistical properties of the hash function to ensure a balanced distribution. Hash tables are associative data structures that map keys onto values using a hash function, which generates a deterministic, pseudorandom integer value for each key. This integer value corresponds to a particular location in an array of stored values, which allows hash tables to achieve order constant time lookups and insertions. While a large variety of hash table designs exists, they broadly fit into two categories: open addressing and closed addressing. In both designs, the hash table is stored in an array of *buckets*

which can store key value pairs, and we use a hash function to map a particular entry into a particular bucket. The different between the two techniques is in how they deal with *collisions*, which occur when two entries produce hashes mapping onto the same bucket. In closed addressing, if a key hashes to a particular bucket, the key value pair *must* be stored in that bucket (thus the addressing scheme is *closed*). Since collisions are likely to occur, mapping multiple elements into the same bucket, this means that each bucket must able to hold multiple entries. This is typically accomplished using a linked list or other variable size array. Because each hash bucket holds a variable size array with space for multiple entries, hash table that use closed addressing can be expanded incrementally, even if every bucket has at least one entry. In order to preserve constant time access, however, the hash table must still *eventually* be expanded to include more buckets, as otherwise lookup time will degrade to take order linear time as the linked lists grow in size.

In open addressing, by contrast, collisions are dealt with by moving to the next available bucket, according to some probing scheme. Linear and quadratic probing are two common schemes for deciding the next available bucket. When inserting an entry, the key value pair are inserted into the first available bucket. When performing a lookup, probing continues until either an entry with the desired key is found or an empty bucket is encountered. As a hash table fills up, it becomes more and more costly to insert new elements into the hash table, until eventually no new elements can be inserted. Hash tables that support dynamic resizing, including those that use both closed and open addressing, typically aim for a particular *load factor*, which is the ratio of filled buckets to total buckets. In order to prevent slowdowns due to too many collisions, the hash table is expanded and all entries are rehashed into the new table.

When it comes to implementing a distributed hash table, open addressing and closed addressing have different advantages and disadvantages. Closed addressing has the obvious advantage of supporting incremental resizing, since new entries can be inserted into a bucket's list even if all buckets have at least one element. However, as with sequential implementations of closed addressing hash tables, they will still eventually need to be resized in order to preserve order constant lookups and insertions. Distributed hash tables that use closed addressing with linked lists for the buckets may also introduce an additional issue of *locality*. As discussed in Section 3.1, BCL does not assume the ability to remotely allocate memory. As such, allocating a new piece of global memory to store a node of the linked list must be done on the *local* process, not the process where the remote list is located. That is to say, the linked list inside a closed addressing hash table's bucket would likely contain linked list nodes allocated on processes scattered throughout the execution, rather than on the specific processes that own the portion of the hash table the bucket is located in. This can prevent the hash table design from being used for *redistribution* of data elements, since many hash table entries will simply be stored in the memory of the process that inserted them. In addition, the use of a linked list will always require pointer chasing to deal with collisions. Using static capacity arrays for the buckets can fix both of these problems, allowing a single remote get operation to retrieve all entries in the bucket. However, rehashing will still be required whenever one of the static capacity buckets has been completely filled.

In BCL, we implement our distributed hash table using open addressing, with a single logically contiguous array of hash table buckets distributed block-wise among all processes. Each bucket is a struct including space for a single hash table entry, including key and value, as well as a status flag for synchronization. Our hash table uses open addressing with a user-configurable probing strategy to resolve hash collisions. As our hash table is implemented with remote get, put, and atomic operations, neither insert nor find operations to our hash table require any direct coordination with remote ranks, and where hardware support is available, hash table operations will take place purely with RDMA operations.

Interface

BCL's `BCL::HashMap` is a distributed data structure. Users can create a `BCL::HashMap` by calling the constructor as a collective operation among all ranks. BCL hash tables are created with a fixed key and value type as well as a fixed size. BCL hash tables use Object-Containers, discussed in Section 3.3, to store keys and values of any type. BCL hash tables also use the standard C++ STL method for handling hash functions, which is to look for a `std::hash <K>` template struct in the standard namespace that provides a mechanism for hashing key objects.

The hash table supports two primary methods, `insert` and `find`. Section 6.3 gives a performance analysis of our hash table.

Atomicity

By default, hash table insert and find operations are atomic with respect to one another, including simultaneous insert operations and find operations using the same key. In addition to this default level of atomicity, users can pass a concurrency promise as an optional argument at each callsite that can allow the data structure to select a more optimized implementation with less strict atomicity guarantees. All the available implementations for insert and find operations are shown in Table 6.1.

Our hash table uses a lightweight, per-bucket locking scheme. Each hash table bucket has a 32-bit *used* flag that ensures atomicity of operations. The lowest 2 bits of this flag indicate the reservation status of the bucket. There are three possible states: (1) free, (2) reserved, and (3) ready. The free state represents an unused bucket, the reserved state represents a bucket that has been reserved and will immediately be written to by a process, and the ready state indicates that a bucket is ready to be read. The remaining 30 bits are *read flag* bits, and they indicate, if flipped, that a process is currently reading the hash table entry, and prevent another process from writing to the entry before the other process has finished reading.

Insert Operations

The default, fully atomic process for inserting (Table 6.1a) requires two atomic memory operations (AMOs) and a remote put with a flush. First, the inserting process computes the

Method	Concurrency Promise	Description	Cost
`insert`			
(a)	`find | insert`	Fully Atomic	$2A + W$
(b)	`local`	Local Insert	ℓ
`find`			
(c)	`find | insert`	Fully Atomic	$2A + R$
(d)	`find`	Only Finds	R

Table 6.1: Implementations of data structure operations for BCL's hash table data structure.

appropriate bucket. Then it uses a compare-and-swap (CAS) operation to set the bucket's status to reserved, a remote put to write the correct key and value to the reserved bucket, followed by a flush to ensure completion of the put, then finally an atomic XOR to set the status of the bucket to ready. Pseudocode for this implementation follows.

1. CAS bucket state from free $\rightarrow$ reserved. If state is discovered to be reserved/ready, instead CAS ready $\rightarrow$ reserved.
2. Once successful, if the previous state was free, write the key and value to the bucket. If the previous state was ready, check the key. If key is incorrect, reset status, move to the next bucket, and begin at (1).
3. Set status bits to ready using an atomic XOR. (Flipped read bits from attempted reads could interfere with a CAS, which is unnecessary.)

In some special cases, we may wish to have processes perform local insertions into their own portions of the hash table. This may be done with only local CPU instructions, not involving the NIC. Crucially, this cannot be done when other operations, such as general find or insert operations, might be executed, since CPU atomics are not atomic with respect to NIC atomics. This implementation requires the concurrency promise `local` (Table 6.1b).

Find Operations

The default, fully atomic implementation of the find operation (Table 6.1c) again involves two AMOs and a remote read. First, the process uses a fetch-and-or to set a random read bit. This keeps other processes from writing to the hash bucket before the process has finished reading it. Then, it reads the value, and, after reading, unsets the read bit. Pseudocode for this implementation follows.

1. Atomic fetch-and-or to set a random read bit. If bit was previously set, start again at (1). If bit was unset and status was reserved, atomically fetch until status is ready.
2. Once read bit is set and status is ready, read the key and value.

3. Unset read bit using atomic fetch-and-and. If key was incorrect, move on to next bucket.

In the common case of a *traversal phase* of an application, where no insert operations may occur concurrent with find operations, we may use an alternate implementation that requires no atomic operations (Table 6.1d), but just a single read operation to read the whole bucket including the reserved flag, key, and value.

Hash Table Size

A current limitation of BCL is that, since hash tables are initialized to a fixed size and do not dynamically resize, an insertion may fail. In the future, we plan to support a dynamically resizing hash table. Currently, the user must call the collective `resize` method themselves when the hash table becomes full.

A dynamically resizing implementation could potentially have each process detect when the hash table becomes too full and activate a collective resize operation automatically. Whenever any process enters a hash table function, they would need to check whether a resize operation had been triggered, which could be done using a low-latency RDMA signaling data structure. The main software complexity to overcome with this approach is how to ensure that all processes will periodically enter the hash table's progress function to check whether a resize needs to be performed. This would likely require allowing data structures to register progress functions that must be checked every time certain BCL functions are called, such as barriers or other functions that could initiate coordination. Since such a signaling system does not fundamentally affect the operation of an RDMA-based hash table, we leave this extension to future work.

Buffering Hash Table Insertions

Many applications, such as the Meraculous benchmark [85, 51], discussed in detail in Section 6.3, exhibit *phasal* behavior, where there is an insert phase, followed by a barrier, followed by a read phase. We anticipate that this is likely to be a common case, and so have created a hash table *buffer* data structure that accelerates hash table insertion phases. An application programmer can create a new `BCL::HashMapBuffer` on top of an existing hash table. The user then inserts directly into the hash map buffer object using the same methods provided by the hash table. This simple code transformation is demonstrated in Figure 6.1. While the hash table interface ensures ordering of hash table insertions, insertions into the hash table buffer are non-blocking, and ordering is no longer guaranteed until after an explicit flush operation. The hash table buffer implementation creates a `FastQueue` on each node as well as local buffers for each other node. When a user inserts into the hash table buffer, the insert will be stored in a buffer until the buffer reaches its maximum size, when it will be pushed to the queue lying on the appropriate node to be staged for insertion. At the end of an insert phase, the user calls the `flush()` method to force all buffered insertions

```
1  BCL::HashMap<int, int> map(size);
2
3  BCL::HashMapBuffer<int, int> buffer(map, queue_size,
4                                      message_size);
5
6  for (...) {
7   buffer.insert(key, value);
8  }
9
10 buffer.flush();
```

Figure 6.1: A small change to user code—inserting into the `HashMapBuffer` instead of the `HashMap`—causes inserts to be batched together.

to complete. Insertions into the actual table will be completed using a local, fast hash table insertion (Table 6.1b). The hash map buffer results in a significant performance boost for phasal applications, as discussed in Section 6.3.

6.2 Bloom Filters

A Bloom filter is a space-efficient, probabilistic data structure that answers queries about set membership [22]. Bloom filters can be used to improve the efficiency of hash tables, sets, and other key-based data structures. Bloom filters support two operations, *insert* and *find*. To insert a value into the Bloom filter, we use k hash functions to hash the value to be inserted k times, resulting in k locations in a bit array that will all be set to one. To check if a value is present in a Bloom filter, the value is hashed k times, and if each of the corresponding bits is set, the value is said to be present. Because of hash collisions, a Bloom filter may return false positives, although it will never return false negatives.

Distributed Bloom Filter

A simple Bloom filter can be implemented in distributed memory as an array of integers distributed block-wise across all processes. To insert a value into this Bloom filter, the value to be inserted is hashed k times to determine which bits in the distributed Bloom filter must be set. Then, the appropriate bits are set using atomic fetch-and-or operations to the corresponding integers. To check whether a value is in the Bloom filter, the value can be hashed k times and the corresponding locations in the array checked.

There is, however, a fundamental limitation in implementing a distributed Bloom filter this way, as it results in a loss of atomicity for insert operations. For many applications, it

may be useful to have an atomic insertion operation that inserts a value into a Bloom filter and atomically returns a boolean value indicating whether the value was already present in the Bloom filter. In a regular serial Bloom filter, we define a value as already present if all k bits for our value were already set in the filter, and not already present if we were required to flip any bits in the filter. Since our insertion operation consists of multiple AMOs which cannot be guaranteed to be executed together atomically, we cannot guarantee that two processes which attempt to insert the same value simultaneously into the Bloom filter will not both believe that they were the first process to insert that value into the Bloom filter. This violates the invariant that a Bloom filter will return no false negatives, so the Bloom filter described above cannot provide this information.

There is also a disadvantage in terms of communication cost for this implementation, since performing an insert requires flipping k bits, generally resulting in k independent atomic operations.

Blocked Bloom Filter

Instead of being comprised of a single bit array, *blocked Bloom filters* are composed of many smaller Bloom filters [94]. To insert a value into a blocked Bloom filter, the value is first hashed to determine which Bloom filter the value should be stored in. The item will then be hashed k times to determine which bits in the smaller Bloom filter need to be set. In shared memory systems, blocked Bloom filters are used to improve the cache performance of large Bloom filters with a block size that is a multiple of the cache line size.

In BCL, we implement a distributed blocked Bloom filter as `BCL::BloomFilter` to solve both of the issues with distributed Bloom filters raised above. Our distributed blocked Bloom filter consists of a number of 64-bit Bloom filters. Inserting an item into our blocked Bloom filter now requires, in all cases, a single atomic memory operation, and the operation is fully atomic. Checking if an operation is present in the blocked Bloom filter requires one remote read memory operation.

Blocked Bloom filters come with the disadvantage of requiring more space to maintain the same false positive rate as a regular Bloom filter. While theoretically, under the assumption that values will be uniformly distributed by a hash function, a blocked Bloom filter should require no more space, in reality values may become clustered, requiring more Bloom filters than theoretically necessary for a particular false positive rate [94]. Empirical experiments with our distributed blocked Bloom filter have shown that an extra $\ln p$ factor of 64-bit Bloom filters are necessary for our distributed blocked Bloom filter to maintain the same false positive rate as a regular Bloom filter, where p is the false positive rate. In the practical false positive range of 10-0.001%, this constitutes a 2-12x increase in space required. Since Bloom filters, even for large sets of data, are very space-efficient (for example, a regular Bloom filter requires only 172 MB to maintain a 0.1% false positive rate with a set of 100M unique items), this is likely an acceptable overhead for the atomicity and performance gains.

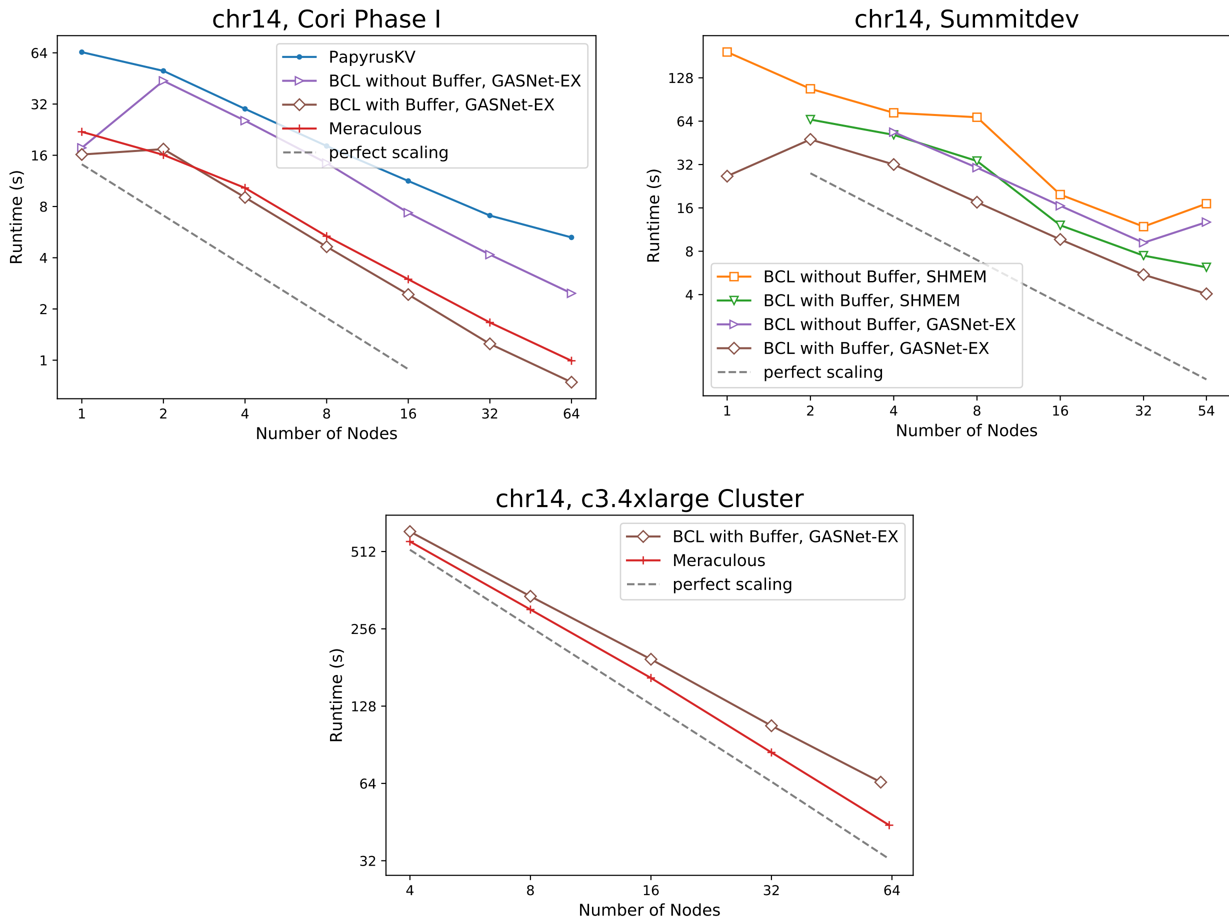


Figure 6.2: Performance comparison on the Meraculous benchmark on the *chr14* dataset.

6.3 Experimental Evaluation

To evaluate the performance of our set-based distributed data structures, we used a number of benchmarks, including two mini-applications, Meraculous, which performs contig generation, and k-mer counting, both of which are from large-scale genome assembly, as well as a collection of microbenchmarks.

We tested these benchmarks across three computing systems, including Cori Phase I, Summitdev, and AWS. The details of these system's configurations are discussed in Chapter 4. On Cori, experiments are performed up to 512 nodes. On Summitdev, experiments are performed up to 54 nodes, which is the size of the whole cluster. On AWS, we provisioned a 64 node cluster and performed scaling experiments up to its full size. For reasons of space, the microbenchmarks are presented only on Cori up to 64 nodes.

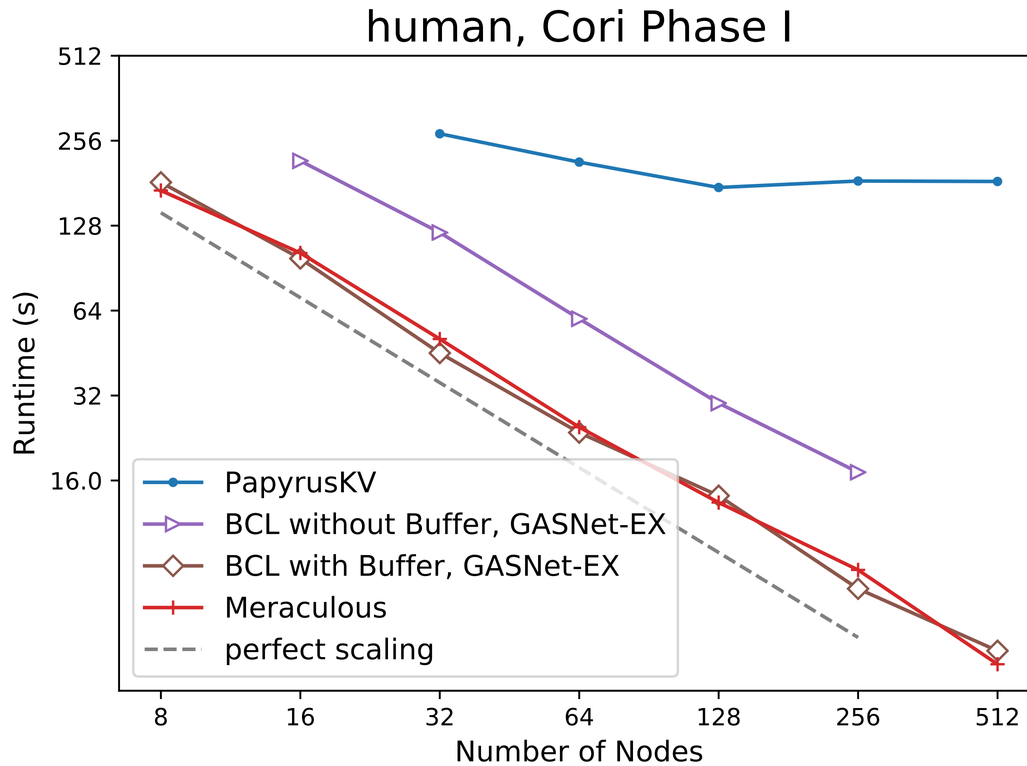


Figure 6.3: Performance on the Meraculous benchmark

Genome Assembly

We evaluated BCL's generic hash table using two benchmarks taken from a large-scale scientific application, a de novo genome assembly pipeline: contig generation and k-mer counting. Both use a hash table, contig generation to traverse a de Bruijn graph of overlapping symbols, and k-mer counting to compute a histogram describing the number of occurrences of each k-mer across reads of a DNA sequence.

Contig Generation in De Novo Genome Assembly

During the *contig generation* stage of de novo genome assembly, the many error-prone reads recorded by a DNA sequencer have been condensed into k-mers, which are short error-free strands of DNA guaranteed to overlap each other by exactly $k-1$ bases. The goal of contig generation is to process these k-mers to produce *contigs*, which are long strands of contiguous DNA [35, 51, 35].

Assembling these k-mers into longer strands of DNA involves using a hash table to traverse the de Bruijn graph of overlapping k-mers. This is performed by taking a k-mer,

computing the next overlapping k-mer in the sequence, and then looking it up in the hash table. This process is repeated recursively until a k-mer is found which does not match the preceding k-mer or a k-mer with an invalid base is discovered.

A fast implementation for contig generation is relatively simple in a serial program, since using any of a large number of generic hash table libraries will yield high performance. However, things are not so simple in distributed memory. The reference solution for the NERSC-9 Meraculous benchmark, written in UPC, is nearly 4,000 lines long [1] and includes a large amount of boilerplate C code for operations like reading and writing to memory buffers [85].

The implementation of the contig generation phase of a genome assembly pipeline is greatly simplified by the availability of a generic distributed hash table in BCL. As described above, the contig generation benchmark is really a simple application split into two phases, an insert phase, which builds the hash table, and a traversal phase, which uses the hash table to traverse the de Bruijn graph of overlapping symbols. Because of this phasal behavior, we are able to optimize the performance of the hash table using BCL's hash map buffer, which groups together inserts by inserting them all at once into a local queue on the node where they will likely be placed, then inserting them all using a fast local insert when a flush operation is called on the hash map buffer. Our implementation of the Meraculous benchmark is only 600 lines long, 400 of which consist of code for reading, parsing, and manipulating k-mer objects. In contrast, the NERSC-9 Meraculous benchmark, written in UPC, contains just over 4,000 lines of code, of which a similar number are devoted to I/O.

We implemented the contig generation phase of a genome assembly pipeline using the Meraculous algorithm [51, 50, 35]. Our implementation is similar to the high-performance UPC implementation [51], but (1) uses our generic hash table, instead of a highly specialized hash table, and (2) uses a less sophisticated locking scheme, so sometimes processes may redundantly perform extra work by reconstructing an already constructed contig. We should note that the Meraculous UPC benchmark is based on the HipMer application, which may have higher performance [50].

We benchmarked our hash table across the same three HPC systems described in Table 4.1 using the *chr14* dataset, which is from sequencing data of human chromosome 14. We compared our implementation to the high-performance UPC reference Meraculous benchmark implementation provided on the NERSC website, which we compiled with Berkeley UPC with hardware atomics enabled [85, 51]. We also compared our hash table to PapyrusKV, a high-performance general-purpose hash table implemented in MPI which has a Meraculous benchmark implementation available [74]. All performance results were obtained by running one process per core. Benchmarks for the UPC implementation are not available on Summitdev because the code fails on POWER processors due to an endianness issue. We also used the Meraculous benchmark prepared by the developers of PapyrusKV [74]. As shown in Figure 6.2, our BCL implementation of the contig generation benchmark matches or exceeds the performance of both the reference high-performance implementation

[1] As measured by David A. Wheeler's 'SLOCCount'.

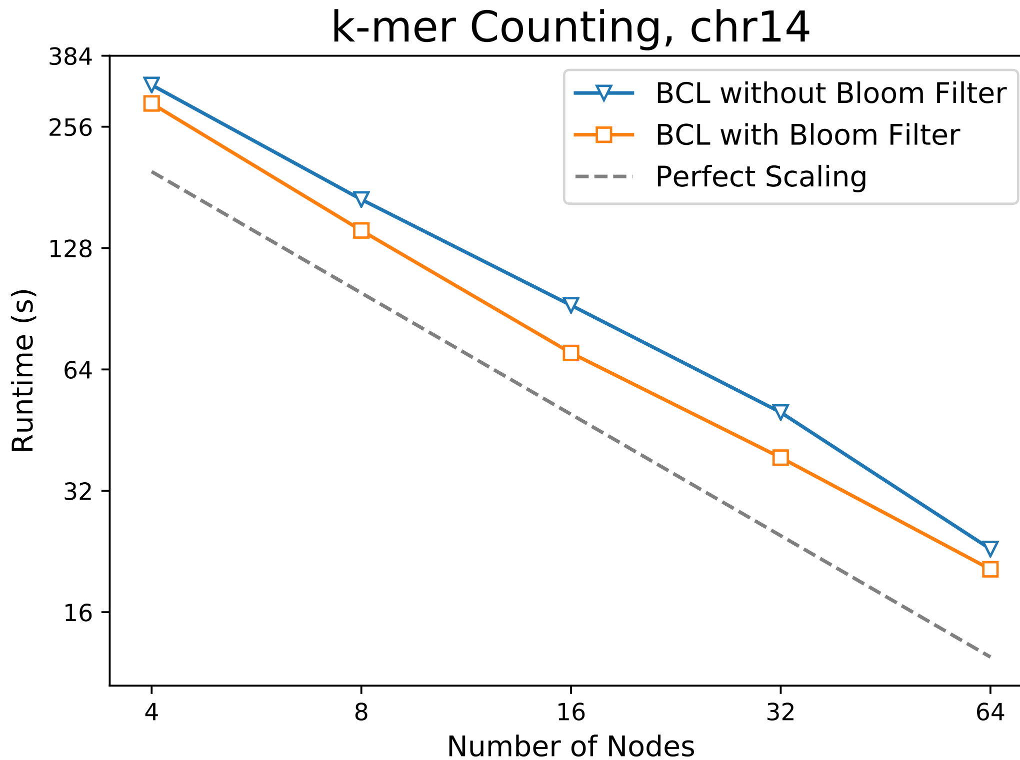


Figure 6.4: Strong scaling for our k-mer counting benchmark using dataset *chr14*.

of Meraculous and the general PapyrusKV hash table. In Figure 6.3, we also show a larger scaling experiment using the human genome as a dataset on Cori. Our BCL implementation appears to be both scalable to high numbers of nodes and portable across different architectures and interconnects.

k-mer Counting

k-mer counting is another benchmark from de novo genome assembly. In k-mer counting, we examine a large number of lossy reads of DNA sequences and split these reads up into small, overlapping chunks of length k. We then use a hash table to calculate a histogram, accumulating the number of occurrences of each individual k-mer, to try to eliminate errors. A large number of k-mers will be erroneous, and, as a result, will appear only once in the counting.

To speed up this histogram calculation, we can avoid unnecessary hash table lookups for items which appear only once in the hash table by using our distributed blocked Bloom filter as discussed in Section 6.2. With this optimization, we first atomically insert a k-mer into the Bloom filter, then only update its histogram value if the k-mer was already

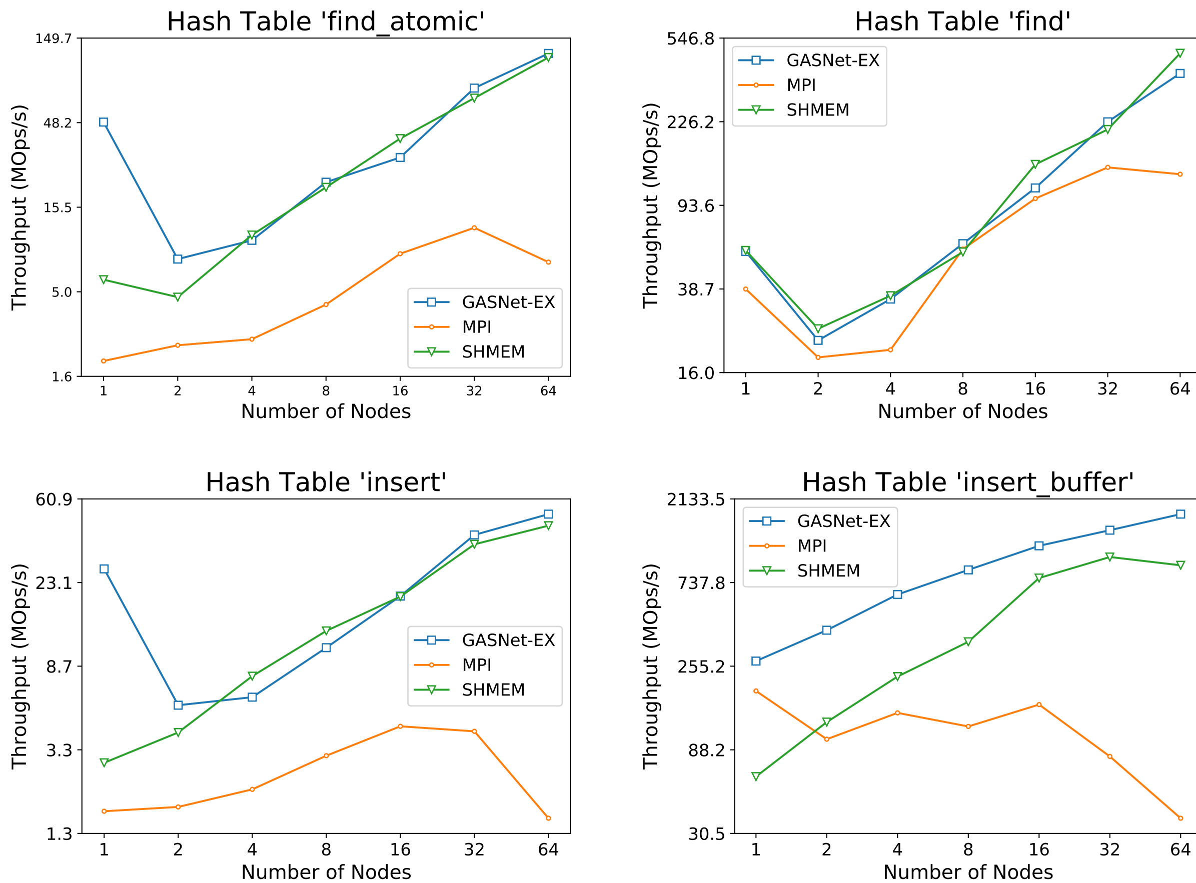


Figure 6.5: Microbenchmarks for the hash table.

present in the Bloom filter. This optimization also significantly reduces the overall memory consumption of k-mer counting because a high portion of the unique k-mers occur only once due to sequencing errors. Consequently, Bloom filters are now common in single node k-mer counters [80]. However, it is harder to efficiently take advantage of Bloom filters in distributed k-mer counting. In the absence of an efficient distributed Bloom filter that keeps global information about the k-mers processed so far, all the occurrences of a k-mer had to be localized in and counted by the same process for local Bloom filters to produce accurate counts [51]. BCL's distributed Bloom filter avoids this localization and the expensive all-to-all exchange of all k-mers associated with it.

As shown in Figure 6.4, our k-mer counting benchmark shows excellent strong scaling, along with a slight performance boost when using our distributed blocked Bloom filter. Aside from this performance boost, lower memory footprint is another benefit of BCL's distributed Bloom filter, since k-mers which appear only once need not be inserted into the hash table.

Microbenchmarks

We prepared a collection of microbenchmarks to compare (1) different backends' performance across data structure operations and (2) the relative performance of different implementations of data structure operations. Each benchmark tests a single data structure operation. In the `HashMap` microbenchmarks in Figure 6.5, we see clear differences between fully atomic versions of data structure operations (`find_atomic` and `insert`) and versions offering fewer atomicity guarantees or buffering (`find` and `insert_buffer`). We see that buffering offers an order of magnitude increase in performance, which we would expect from transforming a latency-bound operation into a bandwidth-bound operation, while the optimized `find` operation offers a factor of 2-3x improved performance, as we would expect from the relative best-case costs ($2A + R$ and R).

Across these benchmarks, GASNet-EX is most effective at automatically utilizing local atomics when only running on one node, while MPI lags behind on most benchmarks, particularly those which make heavy use of atomic operations.

Chapter 7

Matrix Data Structures

7.1 Introduction

Dense and sparse matrices are important data structures for a wide variety of applications across different domains, from machine learning to genomics, graph algorithms, and more. Matrices serve as fundamental building blocks for a wide array of algorithms, and as such high-level matrix libraries such as Matlab [63] and NumPy [59] have become ubiquitous tools for productive programming in sequential and shared memory environments. Since matrix primitives such as matrix multiplication can be quite expensive, and since matrices are likely to grow to a size too big to store in any one node's memory as applications and data sizes scale, there is a rich literature in distributed matrix algorithms and in different schemes for partitioning matrices across nodes in a distributed execution. While there is a large body of work in distributed memory linear algebra, both sparse and dense, the majority of this work has been directed at distributed data structures and algorithms that use *two-sided* communication [61]. Two-sided or message passing communication provide primitives based on matching send and receive operations or on collective communication operations like broadcast and reduce. While two-sided-based linear algebra (and two-sided communication) has been very successful, there are a number of limitations in two-sided algorithms and systems that can be addressed using *one-sided* communication, which allows processes to directly read and write into each others' memory.

Lack of Asynchrony Classical distributed matrix algorithms, like SUMMA [108] or Cannon's algorithm [76] for matrix multiplication, rely on a bulk-synchronous, lock-step execution, and are not easily extended to support asynchronous execution. While this is not an issue for dense problems, in sparse problems where load balancing issues arise, this lack of asynchrony may lead to performance issues.

New Architectures and Systems Some novel architectures and systems, such as GPUs, cannot natively use send and receive operations themselves. Many of these systems, can,

however, use one-sided communication primitives.

Programming Model There are numerous distributed linear algebra libraries that use two-sided communication, and some of them, such as the Cyclops Tensor Framework [102] and Elemental [92], even offer high-level matrix data structures. However, there is a lack of *intermediate-level* primitives in such data structures, such as the ability to retrieve a local copy of a submatrix, that can make development of sparse and asynchronous algorithms cumbersome. This lack of intermediate-level operations is not an oversight of library developers, but a limitation of the programming model— with two-sided communication, retrieving a remote part of a data structure will require send/receive calls from two, three, or even more processes.

In this chapter, we build RDMA-based distributed dense and sparse matrix data structures. These data structures allow processes to retrieve or modify any part of the matrix asynchronously, without coordinating with other processes. This allows us to implement RDMA-based sparse matrix algorithms that are completely asynchronous, with processes free to enter and leave a matrix multiply operation without coordinating with any other processes in some algorithms. Compared to bulk synchronous approaches, which require coordination inside the inner loop, this can help with load balance issues that arise when processes have varying amounts of work due to nonuniform sparsity.

We begin by introducing our cross-platform distributed dense and sparse matrix data structures, which support various ways of distributing the matrix. We then discuss the various data structure primitives offered by our distributed matrices, including a new slicing primitive for one-sided dense matrix data structures. We then discuss different RDMA-based algorithms for matrix multiplication and other essential matrix operations, presenting a performance model that can be used to evaluate their performance. We also demonstrate how this data structure model can be extended to sparse matrices before discussing a collection of different RDMA-based sparse matrix multiplication algorithms, including novel work-stealing algorithms enabled by our RDMA-based data structure. We also discuss implementations of our data structures for GPUs and evaluate the performance of various sparse matrix algorithms on GPUs.

7.2 Distributed Matrix Data Structure

A standard way to store a matrix as a distributed object is to split it up into a collection of *tiles*, where each tile is a single contiguous block of the matrix. In order to operate on our distributed matrix data structure using one-sided operations, we will store each of these tiles in the shared segment of a processor and will distribute a global pointer that references the tile to each process. This way, every process can access any part of the matrix by using index arithmetic to compute the appropriate global pointers and tile offsets, then performing one-sided remote memory operations on the appropriate global pointers.

Name	Description	Optional Parameters
`BlockRect`	Rectangular tiles	None
`BlockRow`	Block row distribution—each tile spans a whole row	Number of rows
`BlockColumn`	Block column distribution—each tile spans at least a whole column	Number of columns
`BlockSquare`	Force a square processor grid	Tile dimensions
`BlockRow` `BlockCustom`	User provides explicitly controls distribution	Tile dimensions, processor grid dimensions

Table 7.1: Matrix block descriptors, which describe different tiling strategies for a distributed matrix.

Tile Distribution

In order to partition a matrix into multiple tiles, we need two facilities: (1) a strategy for splitting up a matrix into a collection of tiles, and (2) a function that, given a particular tile coordinate and grid dimensions, picks the process that will own that particular tile. To do this, we've chosen a a distributed block cyclic layout scheme similar to that used by ScaLAPACK [39], along with a collection of user-friendly *Block* data structures that describe different tile layouts. A block cyclic data layout is a good choice because it can be configured with only a few straightforward parameters and also because other important data layouts, such as block row and column layouts, are special cases of a block cyclic layout. Given a matrix, our block cyclic data layout requires two parameters to distribute that matrix to a series of tiles, (1) a *tile size*, which defines the size of an individual tile, and (2) a *processor grid* which provides a mapping from the tile grid to a particular processor. Given a tile size, a matrix is partitioned into a *tile grid*, which is a collection of smaller tiles that together make up the matrix. An example of a 4x3 tile grid is shown in Figure 7.1a.

The second parameter required, which assigns these tiles to processes, is the shape of the *processor grid*, which is a rectangular grid of processes. In general, our tile grid may be larger than our processor grid, in which case our processor grid is duplicated over the tiles of the matrix. We can compute a tile's corresponding process by taking its grid coordinates and mapping them onto the processor grid with modular arithmetic. Figures 7.1b and 7.1c show distributed matrices with 2x2 and 2x3 processor grids, respectively. Each tile is labeled with the process that owns it, and the processor grid in the top left corner is in bold.

Controlling Matrix Distribution

In our distributed matrix data structure design, users determine how a matrix should be split up using *block descriptors*, which are parameterizable objects that describe different strategies for tiling a matrix. Examples include block row, block column, and rectangular blocked, and are listed in Table 7.1. The block descriptors describe a method for splitting up the matrix into different rectangular tiles and for assigning those tiles to be located on different processors.

The `BlockRect` distribution produces tiles that are as square as possible and equally distributed amongst all processors. By contrast, the `BlockSquare` distribution enforces a

square tile size, at the cost of a possibly unequal tile distribution. `BlockRow` and `BlockColumn` produce block row and block column distributions, such that each tile contains all elements in the corresponding row or column dimension. `BlockOpt`, which uses a block cyclic layout, using a heuristic to pick optimal tile sizes; Finally, the `BlockCustom` distribution allows users to directly supply a tile size and processor grid, giving them explicit control over the grid distribution. All the block distribution objects follow a standard interface, and users can supply their own block distributions if desired, for example by using `BlockCustom` as a base class.

In order to allow easy configuration of the parameterizable block distribution classes, many of the block distributions allow users to optionally pass in parameters that will modify their behavior. Typically, these modify the tile size, for example by setting the number of rows to include in each block in `BlockRow` to achieve a cyclic block row distribution. Anywhere that a dimension is required, users can pass in the constant `div` in order to signify a tile size should be used that will evenly divide the dimension among processors in that grid dimension. If the user does not provide any parameters, any free parameters are automatically picked to be `div`. For example, with the default `BlockRow` distribution with no configuration parameters given, the number of block row tiles is set to evenly divide the number of processors, thus evenly distributing the matrix amongst all the processors.

Data Structure Operations

Every process has a global view of the matrix in an RDMA-based distributed matrix data structure and can access and manipulate different parts of the matrix in a one-sided manner. Our dense matrix distributed data structure has three primary mechanisms for accessing the matrix data: access to a particular matrix element, access to a particular matrix tile, and access to an arbitrary submatrix.

Elementwise Access The bracket operator provides a mechanism for accessing individual elements of the matrix. Invoking the bracket operator along with a matrix index will return a global reference object that can be written to or read from in order to write or read to that element of the remote matrix. (For example, `matrix[{1, 2}]` will return a global reference to the element row index 1 and column index 2.) While every process is free to use elementwise access to manipulate the matrix, it is not a particularly efficient method for implementing matrix algorithms, as each individual element access will result in a message over the network, incurring a latency penalty.

Tile Access Users can also access individual tiles of the matrix. The dimensions of the tile grid, which is determined upon matrix construction and is generally fixed over the lifetime of the distributed matrix, are accessible using the function `grid_shape`. Users can retrieve individual tiles by passing the grid coordinates of the desired tile into the method `get_tile`, which returns a local matrix object containing the contents of the tile. The method `arget_tile` retrieves a tile of the matrix asynchronously. Upon calling `arget_tile`,

the method immediately returns a future object. When the tile is needed, the user can call the `get` method on the future in order to return the tile. A global pointer to a particular tile can be accessed using `get_tile_ptr`.

Slicing One limitation of accessing individual matrix tiles is that tiles are blocks of a fixed dimension located in a fixed tile grid. If a user needs to retrieve an arbitrary submatrix, data from multiple tiles may be required. Using remote get operations, we can also implement a *slicing* operation that retrieves an arbitrary submatrix. Unlike `get_tile`, which can only retrieve blocks of the matrix that are pre-determined by the tile distribution, *slicing* provides a general mechanism for obtaining a local copy of an arbitrary submatrix. Slicing is a useful operation for more complicated algorithms that require access to non-tile-aligned submatrices as well as for algorithms in which the user does not have control over data partitioning. Two matrices which would otherwise require an expensive repartitioning in order to multiplied together can instead be multiplied using what we call a *slicing matrix multiply*, which can multiply two matrices together regardless of the layouts of their respective tile grids.

Slicing is performed by iterating through the rows of the required submatrix, and, for each row, iterating through the tiles that contain that row. Each portion of a tile row included in the slice is copied into a local matrix data structure using an asynchronous remote memory operation. These independent remote memory operations are completed before returning in the blocking `slice` method. In the non-blocking `arslice` method, a future object to the submatrix is returned, and synchronizing on the future will synchronize the corresponding communication operations and return the final submatrix output. This is done using BCL futures, described in detail in Section 3.1. The future contains a pointer to the object that is being constructed along with several request handles, associated with each network transfer needed to retrieve the slice, that must be competed in order for the slice to be ready. Once the user calls the `get` method, all necessary synchronization will be performed, and the slice object will be returned.

Optimizing Remote Get Using C++ Allocators

One caveat of using remote get operations to retrieve objects stored in remote memory is that the local receive buffer into which the remote data will be copied must be registered (or "pinned") with the network interface card (NIC). This is necessary so that the NIC on the remote process will be able to copy the payload data over the network into the receive buffer. (More discussion of the primitives supported by RDMA hardware can be found in Section 3.1.) Remote get operations work by instructing the remote process to perform a put operation into the local receiving buffer in the memory of the process that issued the get operation. In order to perform this remote put, the local receive buffer must be registered. It is important to note that this *hardware* limitation is not present in most *software* communication libraries, which automatically handle remote get operations using one of two methods. For large remote get operations into an unpinned local receive buffer, the receive buffer will be registered with the NIC inside the remote get operation, which

Data Structure	Method	Description	Cost
BCL::DMatrix			
	GlobalRef<T> operator(size_t i, size_t j)	Read or write to an individual of the matrix.	$W\|R$
	std::vector<T> get_tile(size_t i, size_t j)	Synchronously copy tile of matrix to local memory.	$R * m_{\text{tile}} n_{\text{tile}}$
	BCL::future<std::vector<T>> get_tile(size_t i, size_t j)	Asynchronously copy tile of matrix to local memory.	$R * m_{\text{tile}} n_{\text{tile}}$
	std::pair<size_t, size_t> tile_shape(size_t i, size_t j)	Get the shape of a particular tile.	0
	std::pair<size_t, size_t> grid_shape(size_t i, size_t j)	Get the shape of the submatrix grid.	0
	std::pair<size_t, size_t> shape(size_t i, size_t j)	Get the shape of the matrix.	0
BCL::SPMatrix			
	CSRMatrix<T> get_tile(i, j)	Synchronously copy tile of matrix to local memory.	$R * (2nnz_{\text{tile}} + m_{\text{tile}})$
	BCL::future<CSRMatrix<T>> arget_tile(i, j)	Asynchronously copy tile of matrix to local memory.	$R * (2nnz_{\text{tile}} + m_{\text{tile}})$
	size_t tile_nnz(size_t i, size_t j)	Get the number of nonzeros in the sparse matrix.	0
	void set_tile(size_t i, size_t j, CSRMatrix<T> matrix)	Assign a sparse matrix tile to a matrix.	ℓ
	void rebroadcast_tiles()	Propagate new tile updates	B

Table 7.2: A selection of methods from our dense and sparse distributed matrix data structures. R is the cost of a remote read, W the cost of a remote write, A the cost of a remote atomic memory operation, B the cost of a barrier, ℓ the cost of a local memory operation, and n is the number of elements involved in the data structure operation. m_{tile}, n_{tile}, and nnz_{tile} are the number of rows, columns, and nonzeros per tile.

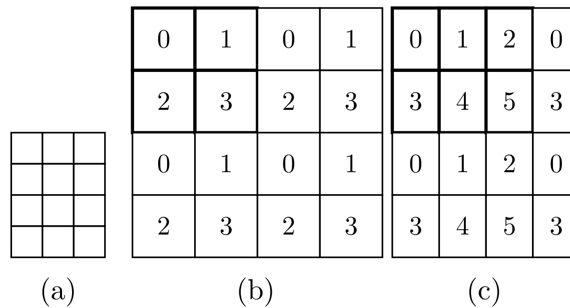


Figure 7.1: A collection of matrix tile grids and distributions.

adds the overhead of an operating system call in addition to the latency of communicating with the NIC over the PCIe bus. For smaller transfers, typically the data will first be copied into a pre-allocated *bounce buffer* of pinned memory, after which it will be copied into the receive buffer [78].

In order to eliminate the overhead associated with performing a remote get with an unpinned local receive buffer, matrix data structure methods that retrieve data, such as `get_tile()` and `slice()` by default use a custom C++ allocator that allocates memory inside the BCL shared memory segment, which is already registered. The allocator is configurable via a template parameter, so users can use other allocators if they have another source of registered memory or they prefer to use unpinned memory. One example of a source of registered memory is the MPI-specific `MPI_Alloc_mem()` function, which provides fast access to pre-registered memory using an internal cache. A C++ allocator that allocates memory using `MPI_Alloc_mem()` is available in BCL's MPI backend.

7.3 Distributed Sparse Matrix Data Structure

Similar to the previously described dense matrix data structure, our sparse matrix data structure splits up a matrix into tiles, each of which stores elements that lie in its corresponding submatrix. In the sparse case, instead of storing a dense tile with a single global pointer, we store a sparse matrix for each tile. In our implementation, we chose to store tiles using the CSR format, but this design is straightforwardly extendable to other sparse storage formats.

Each tile is stored as a globally-accessible CSR matrix, consisting of four values that are copied to each process: a global pointer to a row pointers array that marks the beginning and end of each row, a global pointer to the stored values, a global pointer to the corresponding column indices, and an integer indicating the number of stored values. Any process can retrieve a tile by copying these values locally, and then operate on them as a local sparse matrix. Processes can also calculate the exact shape of each tile using the shape of the overall matrix and the shape of matrix tiles, both of which are globally known.

Tile Access Similarly to our distributed dense matrix data structure, our sparse matrix data structure supports a `get_tile` method, which retrieves a tile of the matrix. In the sparse case, `get_tile` fetches the corresponding CSR matrix data arrays and returns them in a local `CSRMatrix` data structure. An asynchronous version, `arget_tile`, returns a future to a sparse matrix data structure.

Writing to a distributed sparse matrix tile, however, is somewhat more complex than in the dense case: while processes can simply read and write directly to the dense global array used to hold a tile of a dense matrix, modifications to a tile of a sparse matrix will usually change the nonzero structure of the underlying sparse matrix, necessitating the allocation of a new space in remote memory for the tile. Since other processes rely on global knowledge of the size and location of the tiles in remote memory to access the matrix, modifying a tile of the sparse matrix will ultimately require some communication in order to update other processes with the location of the new tile. Other sparse matrix formats could potentially allow for more dynamic, element-wise updates. For example, using a coordinate format sparse matrix for each tile implemented as a distributed queue could potentially allow users to append new values to a particular tile. However, building sparse matrix data structures that are incrementally updatable in normal sequential programs is still an active area of research [43, 89], so we leave this to future work.

Writing to a tile of a sparse matrix happens in two phases. First, the process calls `set_tile` with a local CSR matrix that holds the new values for the tile. `set_tile` allocates a new buffer in the shared segment if necessary, then copies the local matrix over into the shared segment, staging the new global pointer address so that it can be updated. To update all the other processes with the new tile location, the user can issue a collective call to `rebroadcast_tiles`, which will update all of the global pointers resident on each process and will cause the new updated tiles to be visible to all processes.

Sparse Slicing Accessing an arbitrary submatrix of the distributed sparse matrix is once again slightly more complicated. In order to retrieve an arbitrary submatrix, we first retrieve all of the tiles that the submatrix intersects. After the tiles are retrieved, we collect all of the values that are part of the submatrix into a single output matrix using one of our sparse accumulators, which we will introduce shortly. This process can be completed asynchronously using non-blocking methods to retrieve each of the necessary tiles and then accumulating them into the output matrix as they arrive.

The slicing problem that we have defined here can be seen as a special case of sparse matrix indexing as outlined by Buluç and Gilbert [27], who present a bulk synchronous algorithm for sparse matrix indexing based on sparse matrix multiplication. Our RDMA-based algorithm has an asymptotic improvement in communication volume compared that technique, whose communication cost scales with the size of the whole matrix A as $O(nnz(A)/\sqrt{p})$ [27]. By contrast, given a fixed rectangular tile size, the cost of retrieving a submatrix using our RDMA-based algorithm depends only on the size of submatrix being retrieved. So, given a fixed submatrix and tile size, the cost of our slicing algorithm remains constant as the size of the matrix increases. This is because having one-sided remote access to the matrix allows processes to retrieve only the parts of the matrix that they need to access.

Note that this slicing algorithm will sometimes retrieve more than just the specific nonzeros that must be transferred, particularly in the case where a small submatrix is retrieved that intersects many larger tiles. With row and column compressed storage, such as CSR and CSC, it is possible to reduce communication volume in exchange for a higher latency cost. For example, in the case of CSR, an RDMA-based slicing algorithm could first retrieve part of the row pointer array in order to determine which parts of the values and column indices arrays contain rows that are part of the slice. The specific range of elements within each sorted row that needs to be retrieved could then be determined using a binary search algorithm. The high latency costs associated with such an algorithm make it unlikely to compete favorably with the algorithm we have implemented that is outlined above, except in extreme cases, so we leave exploration of alternate sparse slicing algorithms to future work.

The sparse matrix data structure also includes methods that describe the matrix, tile, and grid shape in a manner identical to those in the dense matrix data structure, and the sparse matrix data structure also supports all the block distribution classes supported by the dense matrix data structure. See Table 7.2 for a list of methods supported by the sparse matrix data structure.

Sparse Accumulation Interface

A number of different algorithms, including sparse matrix multiplication as well as the sparse slicing algorithm discussed above, require efficient *sparse accumulation*, or the ability to accumulate a collection of sparse values into a sparse output matrix. Depending on the density and distribution of sparse values being accumulated, as well as the underlying hardware architecture and amount of memory available, there are many different methods for

Name	Description	Needs Sorted Rows?
MKL Accumulator	Calls MKL to greedily sum matrices.	N
Heap Accumulator	Lazily uses a heap to accumulate each row.	Y
Merge Accumulator	Lazily calls a merge-based algorithm implemented in CombBLAS.	Y
Hash Accumulator	Lazily uses a hash table to sum each row.	N
SPA Accumulator	Lazily uses a sparse accumulator to sum each row.	N

Table 7.3: Sparse accumulators.

accumulating sparse matrices that might achieve good performance. In order to allow algorithms to flexibly select from a number of different sparse matrix accumulation methods, we created a generic `SparseAccumulator` interface along with a number of different implementations. Classes that fulfill the sparse accumulator interface must provide two methods: (1) `accumulate`, which accepts a sparse matrix to be accumulated, along with an optional offset and bounds offset and bound the region being accumulated, and (2) `get_matrix`, which returns the sparse matrix being accumulated. Some sparse accumulators are lazy, storing matrices internally to only be accumulated once `get_matrix` is called, and some are greedy, maintaining an internal representation of the accumulated matrix that is updated with each call to `accumulate`.

Sparse accumulator interfaces that we explored include an accumulator based on Intel MKL's matrix summation kernel that calls MKL to greedily sum sparse matrices together, an accumulator that uses a heap-based algorithm to lazily accumulate matrices, an accumulator that calls a lazy merge-based algorithm in the Combinatorial BLAS [12], an accumulator that uses a hash table to sum each row of the resultant sparse matrix on demand, and an accumulator that uses a SPA sparse accumulator [53] to sum each row of the sparse matrix on demand. These accumulators are listed in Table 7.3.

RDMA-Based Matrix Multiplication

In this section, we provide a brief introduction to RDMA-based distributed matrix multiplication before introducing an *inter-node roofline model* that can be used to model the performance of RDMA-based matrix multiply algorithms. We describe how the inter-node roofline model can be used to evaluate different variants of distributed matrix multiplication, such as Stationary A, B, and C algorithms, and to select the optimal algorithm for a particular problem. We then show how these algorithms can be extended to the sparse case and to support execution on GPUs in Section 7.4.

Distributed matrix multiplication can be viewed as a collection of smaller matrix multiplications that compute each block of the output result matrix. Here, we refer to the result matrix as $AB = C$. To compute one output block of the C matrix, we must multiply a *row block* of A by a *column block* of B. This can be calculated incrementally as the accumulation of a series of matrix multiplications of the tiles in the corresponding rows of A and columns of B. Figure 7.2 illustrates the component matrix multiplications necessary to compute one

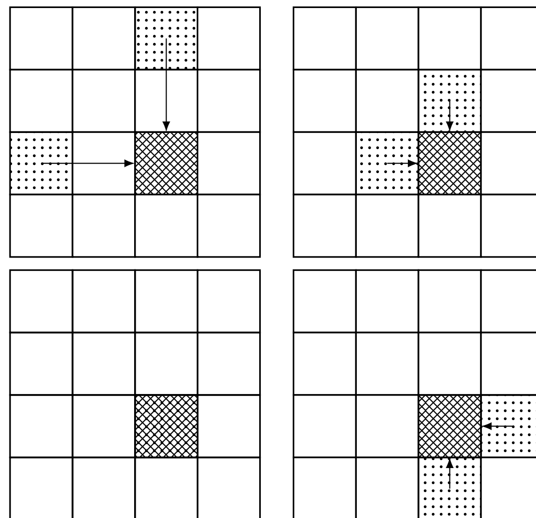


Figure 7.2: Illustration of the local matrix multiplications necessary to compute one block of the output matrix. All submatrices in the block row of A are multiplied by the corresponding submatrices in the block column of B.

block of the C matrix. Each of these four matrix multiplications pictured is independent, and can be computed in any order, but the results must be accumulated together to arrive at the resulting output block of the C matrix.

Stationary C Algorithm

Given this decomposition of matrix multiplication into local matrix multiplication operations whose outputs must be accumulated together in order to produce one block of the output, a distributed matrix multiplication algorithm must use some strategy to split up this work amongst all the processors in the distributed execution. One of the most common strategies for this is a *Stationary C* approach, which dictates that the process that owns a particular tile of the output C matrix should perform each of the component matrix multiplications needed to produce the output. This technique typically results in a need to communicate the tiles of A and B to each of the processors that require them as inputs, while the tiles of C remain in place. Thus, the C matrix remains stationary.

In a traditional bulk synchronous approach, communication occurs using collective broadcast operations in the tile rows of A and the tile columns of B. Pseudocode for such a SUMMA implementation is shown in Algorithm 2. As we discuss in more detail in Section 7.4, this collective communication forces a global synchronization inside the inner iteration, which can cause performance degradation when there is load imbalance due to differences in computation or communication volume, as commonly occurs when the matrices are sparse.

An RDMA-based version of this algorithm instead allows processes to directly access tiles

Algorithm 2 Bulk-synchronous SUMMA implementation.

```
i,j = C.my_block()
for k in 0..K-1:
  # Broadcast k'th tile of A in row
  broadcast(local_a, k, A.row_comm(i))
  # Broadcast k'th tile of B in col
  broadcast(local_b, k, B.col_comm(j))
  # C is local
  local_c = C.tile_ref(i, j)
  local_c += local_a*local_b
```

of the matrix using remote memory operations. Thus, in a Stationary C algorithm, a process that owns a tile of C can iterate through the corresponding tile row of A and tile column of B and directly retrieve the tiles of A and B needed to perform the component local matrix multiplications without synchronizing with any remote processes. This decouples the iterations of the inner loop, allowing for greater asynchrony. Algorithm 3 shows pseudocode for our RDMA-based algorithm, while Algorithm 6 shows C++ code for a naive implementation of the algorithm using the `get_tile` method to retrieve each tile.

Algorithm 3 RDMA-based 2D *C*-Stationary SpMM. M, N, and K represent tile dimensions.

```
for i in 0..M-1:
  for j in 0..N-1:
    if C.owner(i, j) == rank():
      for k in 0..K-1:
        local_a = A.get_tile(i, k)
        local_b = B.get_tile(k, j)
        local_c = C.tile_ref(i, j)
        local_c += local_a*local_b
barrier()
```

Optimizations

Two important optimizations are necessary in order to achieve good performance for the RDMA-based algorithms described here. First, we use the non-blocking version of `get_tile` to asynchronously retrieve a remote tile of the matrix. This allows us to prefetch the tiles needed for the next iteration before entering into the local matrix multiply operation, creating overlap of communication with computation. Second, we apply an iteration offset to the inner loop, which ensures two things: (1) it spaces processes apart, so that not all processes will request the same tile in their row and column at the same time, and (2) it generally ensures[1]

[1]For square block tile grids, both remote gets in the first iteration will be to a local tile, while for rectangular block tile grids, one of them will be to a remote tile.

that the first remote get issued is to a local tile, which helps jumpstart communication and ensures that almost all communication can be overlapped with computation. For the stationary C algorithm, we apply an iteration offset of $i + j$, while in the stationary A and stationary B algorithms, we apply an iteration offset of $i + k$ and $k + j$, respectively. Pseudocode which demonstrates these optimizations applied to a stationary C algorithm is shown in Alg 4.

Algorithm 4 Optimized RDMA-based 2D C-Stationary SpMM. M, N, and K represent tile dimensions.

```
for i in 0..M-1:
  for j in 0..N-1:
    if C.owner(i, j) == rank():
      k_offset = i + j
      buf_a = A.async_get_tile(i, k_offset % K)
      buf_b = B.async_get_tile(k_offset % K, j)
      for k_ in 0..K-1:
        k = (k_ + k_offset) % K
        local_a = buf_a.get()
        local_b = buf_b.get()
        local_c = C.tile_ref(i, j)

        if (k_ + 1 < K):
          buf_a = A.async_get_tile(i, (k+1) % K)
          buf_b = B.async_get_tile((k+1) % K, j)
        local_c += local_a*local_b
barrier()
```

Performance Model

In this section, we develop an inter-node roofline model that we can use to model and evaluate the performance of RDMA-based distributed matrix multiplication algorithms.

Let us consider a distributed matrix multiply computing the product $C = AB$. Here, each of the matrices C, A, and B is split up into tiles which live on different processes. These tiles could be dense or sparse, but we first assume a dense matrix multiplication. Say that C is distributed into a grid of $M \times N$ tiles, A is distributed among a grid of $M \times K$ tiles, and B is distributed into a grid of $K \times N$ tiles. As outlined above, in this work we assume that tiles will be regular, meaning that each tile will have the same size, dimensions, modulo differences due to the tile size not dividing the matrix size evenly. A different, more irregular tiling mechanism might be more appropriate for matrices from certain applications such as those involving linear-scaling quantum-chemical calculations [25], where the near-sightedness creates natural block-sparse structure. However, the vast majority of application targets (Graph Neural Networks, graph algorithms, eigenvector computations, and tensor decompositions) do not have this natural block-sparse structure, so the added complexity of irregular tiling is not justified for us.

As outlined above, to compute a single tile of the C output matrix $C_{i,j}$, we must compute the result $C_{i,j} = C_{i,j} + \sum_{k=0}^{K} A_{i,k}B_{k,j}$, where the indices i, j, and k index into tiles of the matrices. This computation space can be thought as a 3D cube where each block corresponds to a single local matrix multiply. These pieces of work can be assigned to processors in different ways, resulting in 1D, 2D, and 3D structured algorithms. We are then left with the choice of how to assign pieces of the computation to processors, as well as how to communicate tiles of the matrices necessary to execute the matrix multiply. In the *stationary C* method outlined above, the A and B matrices will be communicated, while tiles of the C matrix will be available locally. For each tile of the matrix $C_{i,j}$, the process that owns that tile will be responsible for calculating the output block $C_{i,j} = C_{i,j} + \sum_{k=0}^{K} A_{i,k}B_{k,j}$.

Similarly, we could organize the computation so that the owner of each tile of A will be responsible for executing each of the local matrix multiplication operations in which it is used. In this *stationary A* method, tiles of A will thus be available locally, while tiles of B will be fetched through some method, and output updates to C will be sent and accumulated at their final location. This means that for each tile of A $A_{i,k}$, the process that owns that tile will be responsible for executing $C_{i,j} = C_{i,j} + A_{i,k}B_{k,j}$ for all j. Note that this means that the tile of B $B_{k,j}$ must be somehow retrieved from remote memory, while the result of each local matrix multiply $C_{i,j}$ must be sent and accumulated to the corresponding block of C. The *stationary B* method is similar, except that B remains stationary, while A must be communicated.

As in our performance model for distributed dense matrix multiply, assume a two-dimensional grid of matrix tiles, where the matrices A $(m \times k)$, B $(k \times n)$ B, and C $(m \times n)$ are distributed amongst p processors laid out in $\sqrt{p} \times \sqrt{p}$ tile grids. Tile dimensions for A, B, and C are then $\frac{m}{\sqrt{p}} \times \frac{k}{\sqrt{p}}$, $\frac{k}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$, and $\frac{m}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ respectively, with $\sqrt{p}$ tile columns and $\sqrt{p}$ tile rows in each matrix. For a dense matrix, the total number of elements in each tile is the product of its dimensions. Our RDMA-based stationary C algorithm performs a remote get operation to retrieve a dense tile of A of size $\frac{mk}{p}$ and a dense tile of B of size $\frac{kn}{p}$. The total communication cost is the sum of these two sizes.

If we offset the order of iteration of the k loop by $i + j$, as discussed in Section 7.3, we can also ensure that communication will be balanced with each process sending exactly two tiles per iteration, one of the A matrix, and one of the B matrix. To prove this, assume tile (i, j) is stored on processor $i(N + 1) + j$, where $i \in [0, N)$, $j \in [0, N)$, and $N = \sqrt{p}$.

Proof. Towards a contradiction, assume that $i(N+1)+j$ is not unique. Then, $i(N+1)+j = i'(N + 1) + j'$. Assume $i' > i$. Then $j = (i' - i)(N + 1) + j'$. Since $i' - i \geq 1$, $j > N + 1$, which is a contradiction. □

With the offset, each process sends and receives exactly $\frac{mk}{p}\frac{kn}{p}$ elements over the network in each iteration. In a fully connected network, each of these transfers is independent and will not interfere with each other, in the sense that they will not inhibit other transfers from achieving full link bandwidth. Many modern datacenter networks, such as that used by

Summit, are indeed fully connected fat trees where this assumption holds completely. Many other network topologies, such as Dragonfly, closely approximate a fully connected network.

In the case of Cannon's algorithm [76], which reaches the theoretical lower bound for distributed matrix multiplication without replication [67], each process also sends $2(N/\sqrt{p})^2$ elements in each iteration (for $N = m = k = n$), which includes one tile of A, continually passed on to the right, and one tile of B, continuously passed down. The popular SUMMA algorithm also achieves within a log factor of this lower bound.

Next, we present an *inter-node* roofline model that predicts the performance of each iteration of the distributed matrix multiplication by examining the ratio of data that must be communicated over the network and computation that must be performed. The roofline model predicts the maximum possible performance of a computational kernel depending on its *arithmetic intensity*, or number of operations performed per byte that must be fetched from memory. Typically, the roofline model is used to characterize serial or multi-threaded compute kernels in terms of how compute or bandwidth intensive they are, where the limit on "compute" is defined by the processor's arithmetic peak, and "bandwidth" is determined by memory bandwidth. Here, we develop an *inter-node* roofline model to characterize performance of our distributed memory, RDMA-based matrix multiply algorithms. In this instance, the bandwidth involved is network communication, and the roofline peak of local operations serves as our "arithmetic peak." In our *inter-node* roofline model, we compute the inter-node arithmetic intensity of each iteration as the number of flops performed divided by the number of bytes that must be communicated over the network. The flat portion, providing the "roof" on performance of our inter-node roofline, is the *local* roofline peak for our local dense matrix multiply (GEMM) calls.

We first build a regular *local* roofline model to characterize the performance of our local matrix multiply operations. For distributed matrix multiply, we can calculate the arithmetic intensity as the number of flops that need to be performed ($O(n^3)$, based on the three tile dimensions) divided by the amount of data that needs to be read from memory ($O(n^2)$, based on the sum of the dimensions of A, B, and C). We use the variable w to represent the number of bytes in an element. This allows us to compute the arithmetic intensity (AI) for a local GEMM operation.

$$local\ GEMM\ AI = \frac{\frac{2mkn}{p\sqrt{p}} + \frac{mn}{p}}{w\frac{mk+kn+mn}{p}}$$

To calculate the local roofline peak, which in our inter-node roofline model is the new "roof" on performance, we multiply this arithmetic intensity by the memory bandwidth of each local processing unit B_l, followed by a max operation with the arithmetic peak of the machine. This gives us the maximum possible performance that each individual processor could achieve for a particular problem size, assuming all network communication can be overlapped.

Now, for our *inter-node* roofline model, to compute the inter-node arithmetic intensity, we divide the number of flops performed, which is the same expression from the numerator of the

local GEMM arithmetic intensity, by the total number of bytes that must be communicated over the network, which in the Stationary C algorithm is equal to the number of bytes in the tiles of A and B. Note that the formula for the inter-node arithmetic intensity is similar to the local arithmetic intensity, except that the denominator does not include the output C matrix, which need not be communicated over the network in this algorithm variant.

$$\textit{inter-node GEMM AI} = \frac{\frac{2mkn}{p\sqrt{p}} + \frac{mn}{p}}{w\frac{mk+kn}{p}}$$

To calculate the inter-node roofline peak, we multiply this inter-node arithmetic intensity with the inter-node bandwidth B_i, which will give us a flop rate, and then max this output with the local roofline peak calculated above. We can use this roofline to characterize the performance of distributed dense matrix multiply on the Cori supercomputer.

Network Balance

The theoretical calculation using the injection bandwidth above assumes a *balanced* network, by which we mean a network in which the network switches have enough bandwidth that they will not be overly congested, resulting in lower achieved bandwidth than the injection bandwidth. With modern high radix, low diameter HPC networks such as the Cray Aries and Mellanox EDR Infiniband networks, this is often a fair assumption: some systems like Summit are built with full global bandwidth, meaning they can always sustain the injection bandwidth no matter the distribution of nodes given to a running job. On some systems, however, this is not the case. For example, on Cori Phase I, the global bandwidth per node is 2.41 GB/s, while the injection bandwidth is 9.25 GB/s. This means that smaller jobs that are locally clustered will operate at the injection bandwidth; however, as the number of nodes in a job increases, bandwidth achieved per node will decay to the global bandwidth per node as messages are forced to travel or more congested global links, meaning that performance will be somewhat degraded. On fully-connected Fat Tree networks like on Summit, this effect is not present.

Alternate Work Distributions

The *Stationary C* algorithm, which we have outlined above and analyzed using our inter-node roofline modes, provides one possible way of dividing work amongst processors, namely by scheduling the process which owns a given block of C to perform all computation that goes into that output block. Conveniently, this schedule simplifies data movement, since the output blocks are all accumulated locally, instead of needing to be accumulated to a remote process. This is a side effect of the fact that, with the *Stationary C* algorithm, the needed block of C is local, and can be obtained with no communication cost.

While this algorithm lends itself to a straightforward implementation, since it does not require updates to be applied remotely over the network, it is not always the most optimal algorithm. Problems where A or B are of much larger sizes, may benefit from distributing

work such that the A or B matrix remains stationary, while updates to C are applied over the network. In the *Stationary A* algorithm, each process iterates through each tile of A, and, for each tile that the process owns, iterates through the corresponding tiles of the B matrix which need to be multiplied with that tile. For each component matrix multiplication, the process retrieves a local copy of the tile of the B matrix, computes a local matrix product, then writes the component update to one layer of k copies of the C matrix, where k is the number of rows in the tile grid of A. Algorithm 5 shows pseudocode for the *Stationary A* algorithm. Here, we use a remote queue on each process to allow tiles to be remotely accumulated. When an updated is produced that needs to be applied to a tile of the C matrix, the process that produces it performs an insert operation on the remote queue on the process where it needs to be applied. At some point, the remote process will pull the update off of the queue and apply it.

This algorithm, in exchange for the overhead associated with queue insertions, has the advantage of reading tiles from B and sending updates to C instead of reading from the A and B. In the case that A is much larger than C, this can greatly increase the inter-node arithmetic intensity by avoiding the need to read large tiles of A. Instead of the equation from earlier, this algorithm now transfers

$$w\frac{kn + mn}{p}$$

bytes. We can compare the amount of data transfer necessary with the Stationary A, B, and C algorithms in order to select the optimal version at runtime based on the matrix dimensions involved.

Algorithm 5 RDMA-based 2D A-Stationary SpMM. M, N, and K represent tile dimensions.

```
for i in 0..M-1:
  for k in 0..K-1:
    if A.owner(i, k) == rank():
      for j in 0..N-1:
        local_a = A.tile_ref(i, k)
        local_b = B.get_tile(k, j)
        local_c = local_a*local_b
        queue[C.owner(i, j)].push(get_ptr(local_c))

while local_c_ptr = queue[rank()].pop():
  C.my_tile() += get_tile(local_c_ptr)
barrier()
```

7.4 Sparse Matrix Multiplication

Sparse matrix multiplication is an important computational primitive that arises in simulation, data analysis, and machine learning applications. Typically limited by memory and

Algorithm 6 Matrix multiplication using our distributed matrix data structure and the *Stationary C* algorithm.

```
1 template <typename T>
2 void multiply(BCL::DMatrix<T>& a, BCL::DMatrix<T>& b,
3               BCL::DMatrix<T>& c) {
4   for (size_t i = 0; i < c.grid_shape()[0]; i++) {
5     for (size_t j = 0; j < c.grid_shape()[1]; j++) {
6       if (c.tile_ptr(i, j).is_local()) {
7         for (size_t k = 0; k < a.grid_shape()[1]; k++) {
8           auto local_a = a.get_tile(i, k);
9           auto local_b = b.get_tile(k, j);
10          T* local_c = c.tile_ptr(i, j);
11
12          cblas_gemm_wrapper_(local_a, local_b, local_c,
13                              ...);
14        }
15      }
16    }
17  }
18 }
```

network performance, these computations are especially challenging for unstructured matrices, such as those occurring in graph neural networks, genomics, graph analytics, and other data intensive problems. Sparse matrix primitives provide a convenient and important set of operations for performance tuning that will impact a wide array of applications, and there is a large body of prior work optimizing sparse matrix kernels for multicore [88], manycore [96], GPU [115], and distributed memory environments [97, 2, 64, 101, 55]. Two of the most important kernels are sparse times dense matrix multiplication (SpMM), where the dense matrix is tall and skinny, i.e., representing a set of vectors, and sparse times sparse matrix multiplication (SpGEMM). Sparse matrix-vector multiplication is also an important kernel in many applications, and has been studied even more extensively, although our focus here is on the matrix-matrix operations.

The large majority of prior work on distributed-memory sparse matrix kernels focuses on bulk synchronous implementations based on SUMMA [108], using collective operations, typically broadcast and/or reduce, in the inner loop of the algorithm to move data to the processes that need it for their local matrix multiplications. While SUMMA-like algorithms have the advantages of being straightforward to implement and generally scaling well, for sparse problems, they have the disadvantage of forcing processes to proceed in lockstep in the inner loop of the algorithm. When there is load imbalance, either in the amount of computation that must be performed locally by each process, the amount of data that must be transferred to each process, or both, this can potentially result in decreased performance. This can occur even when the total amount of work performed by each process is the same,

Algorithm 7 Matrix multiplication using our distributed matrix data structure and the *Stationary A* algorithm.

```
1 template <typename T>
2 void multiply(BCL::DMatrix<T>& a, BCL::DMatrix<T>& b,
3               BCL::DMatrix<T>& c) {
4   std::vector<BCL::DMatrix<T>> acc_c;
5   for (size_t i = 0; i < a.grid_shape()[1]; i++) {
6     acc_c.push_back(c.copy());
7   }
8   for (size_t i = 0; i < a.grid_shape()[0]; i++) {
9     for (size_t k = 0; k < a.grid_shape()[1]; k++) {
10      if (a.tile_ptr(i, k).is_local()) {
11        for (size_t j = 0; j < c.grid_dim()[1]; j++) {
12          T* local_a = a.get_tile(i, k);
13          auto local_b = b.get_tile(k, j);
14          std::vector<T> local_c(c.tile_size(i, j));
15
16          cblas_gemm_wrapper_(local_a, local_b, local_c,
17                              ...);
18          acc_c[k].put_tile(i, k, local_c);
19        }
20      }
21    }
22  }
23  BCL::barrier();
24
25  for (size_t i = 0; i < c.grid_shape()[0]; i++) {
26    for (size_t j = 0; j < c.grid_shape()[1]; j++) {
27      if (c.tile_ptr(i, j).is_local()) {
28        for (size_t k = 0; k < acc_c.size(); k++) {
29          c.tile_view(i, j) += acc_c[k].tile(i, j);
30        }
31      }
32    }
33  }
34 }
```

since processes may have differing amounts of work in each iteration. To illustrate, Figure 7.3 shows two kinds of load balance for squaring a randomized sparse matrix generated by the R-MAT model [30], with parameters $a = 0.6$, $b = c = d = 0.4/3$, edgefactor 8, and scale 17. The total (end-to-end) computation has only 20% load imbalance whereas the synchronization points in between stages amplify the load imbalance to $\approx 2.3\times$. The load imbalance is defined as the max/avg ratio, the ratio of maximum number of flops performed by any processor to the average number of flops per processor.

Load imbalance is often addressed by randomly permuting the rows and/or columns of the sparse matrices, but this has disadvantages, the foremost being that the required permutation of the inputs, followed by permutation of the outputs, can be expensive. Furthermore, permuting the matrix may possibly remove structural locality, resulting in decreased performance of local matrix multiply operations. For example, Slota et al. [99] showed that the loss of locality due to random permutations can hurt the performance of graph algorithms significantly where parallel PageRank performs $2-5\times$ slower on a randomly permuted graph compared to other orderings. Even if we ignore the loss of locality or try to recover it by local reordering after a global random permutation, there are theoretical limits on how much load balance random permutations can achieve. For matrices with dense rows or columns, the load balance can deteriorate with $\sqrt{p}$ for large p in the worst case, where p is the number of processors, per Theorem 5.4 of Azad et al. [11].

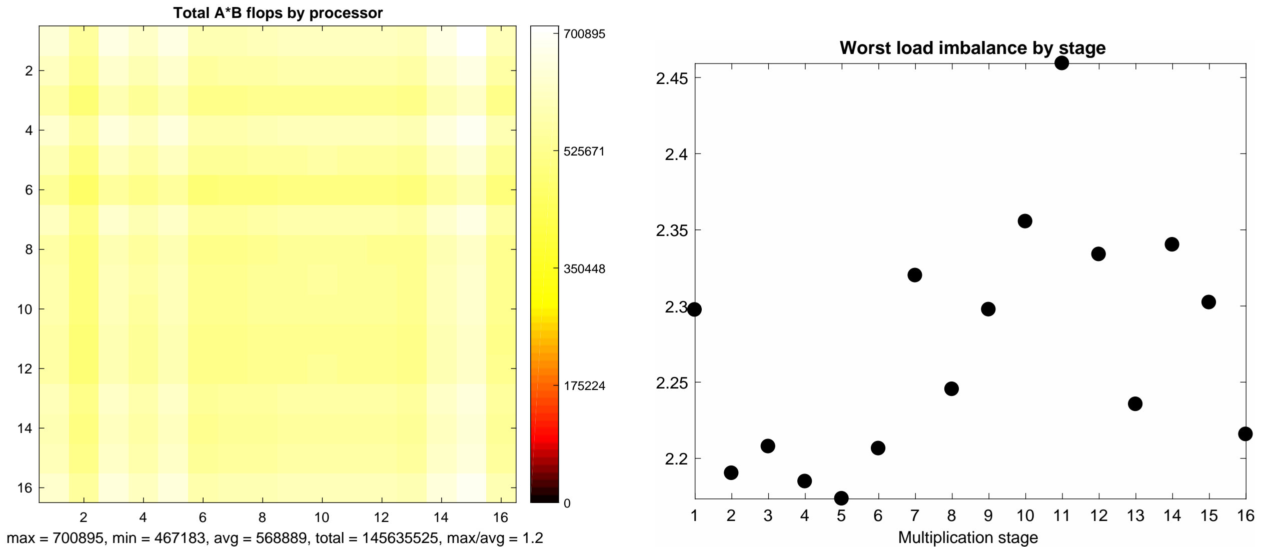
Asynchronous algorithms, which do not require synchronization or coordination between processes as in bulk synchronous algorithms, are an approach to dealing with this load imbalance in distributed sparse matrix multiplication. In this chapter, we develop dense and sparse matrix data structures that use RDMA, meaning that a process can manipulate remote parts of the matrix using one-sided remote put and get operations, without requiring synchronization or coordination with the remote process. Using these dense and sparse matrix data structures, we then design and implement a number of different *RDMA-based, asynchronous* matrix multiply algorithms, which remove synchronization from the inner loop, allowing each process to proceed independently. In addition, we develop and implement an RDMA-based workstealing algorithm, which allows processes that have finished early to steal work from processes that are overburdened using a lightweight reservation scheme.

Furthermore, we examine the performance of these different sparse matrix multiply algorithms running across multiple GPUs in a distributed memory environment. We examine the challenges presented by the highly compute dense nodes of modern GPU-based supercomputers, as well as the benefits of direct GPU-to-GPU transfers offered by new network technologies like NVIDIA GPUDirect RDMA and NVLink.

The main contributions of this chapter are:

1. Design and implementation of RDMA-based distributed dense and sparse matrix data structures for GPUs.

2. New asynchronous, RDMA-based algorithms for SpMM and SpGEMM, including work-stealing algorithms.



(a) End-to-end load imbalance when processors do not synchronize across 16 stages. The heat plot has $256 = 16 \times 16$ boxes, which demonstrate the max/avg load imbalance is ≈ 1.2.

(b) Per stage max/avg load imbalance of the same algorithm. An implementation that synchronizes in between stages would consequently incur load imbalance of ≈ 2.3.

Figure 7.3: Total (end-to-end) vs. per-stage load balance multiplying a R-MAT model-generated sparse matrix with a sparse 2D algorithm. Simulated on a 16×16 process grid.

3. A roofline-based performance model, to characterize the performance of our RDMA-based implementations.

4. An in-depth performance analysis of our RDMA-based implementations, along with a comparison to traditional bulk synchronous methods.

7.5 Background

Sparse times dense matrix multiply (SpMM) and sparse times sparse matrix multiply (SpGEMM) are two important sparse linear algebra primitives. SpMM is used in a variety of blocked iterative methods, graph algorithms, as well as graph neural networks (GNNs) [66, 107, 65]. In these contexts, SpMM typically involves multiplication of a sparse matrix with a tall and skinny dense matrix, where the number of columns of the sparse matrix varies between 32 and 1024. SpGEMM is also used in graph algorithms, including betweenness centrality, triangle counting, cycle detection, and Markov clustering, as well as in applications such as in genomics [28, 10, 45, 29, 42].

Distributed Matrix Algorithms

Distributed matrix algorithms allow multiple processes to work together to execute a single matrix multiply operation. Let's first consider a distributed matrix multiply computing the product $C = AB$. Here, each of the matrices C, A, and B is split up into tiles which live on different processes. The matrix tiles will be sparse or dense, depending on what kind of matrix multiplication is being performed. Let's assume that C is distributed among a grid of $M \times N$ tiles, A is distributed among a grid of $M \times K$ tiles, and B is distributed among a grid of $K \times N$ tiles. The algorithms we consider here rely on a tiling scheme that is regular on the indices. In other words, each tile has the same dimensions, modulo differences due to the tile size not dividing the matrix size evenly. A different, more irregular tiling mechanism might be more appropriate for matrices from certain applications such as those involving linear-scaling quantum-chemical calculations [25], where the near-sightedness creates natural block-sparse structure. However, the vast majority of application targets (GNNs, graph algorithms, eigenvector computations, and tensor decompositions) do not have this natural block-sparse structure, so the added complexity of irregular tiling is not justified for us.

Bulk Synchronous SUMMA

In bulk-synchronous SUMMA, collective broadcast operations are used to communicate tiles of the matrix. In SUMMA, communicators are created for each tile row of the A matrix and each tile column of the B matrix. In each iteration of the algorithm, a block of the A matrix is broadcast in each row communicator and a block of the B matrix is broadcast in each column communicator. Each process will compute its corresponding local matrix multiplication, accumulating into its local block of C. Stationary A (or B) algorithms are also possible with SUMMA, by replacing one of the broadcasts with a collective reduction operation to accumulate each block of C into the correct place. Pseudocode for SUMMA is shown in Algorithm 2.

As discussed in Section 7.4, variation in the number of nonzeros, as well as the sparsity patterns, of individual tiles of sparse input matrices can create load imbalance in computation due to the differing numbers of flops that must be performed in each local matrix multiply, as well as load imbalance in communication, due to the different amounts of data that must be transferred by each process. Bulk synchronous SUMMA-like algorithms can suffer due to these load imbalances.

Modifying Remote Tiles

Modifying a local tile is straightforward, since we can use local pointers which point to a local tile of the matrix to directly modify the data. Modifying a remote tile, which is only required for some of the RDMA algorithms, is somewhat more complex, particularly when the remote tile happens to be sparse. To deal with this, we use a system of remote queues

to issue asynchronous updates to the remote tiles. Each process has a globally visible queue, and other processes can push updates to this queue in order to send updates that the remote process needs to apply to tiles that it owns. During algorithms that require accumulations to remote tiles, each process will periodically check to see if there are elements waiting in the queue, and if so, dequeue the elements and accumulate the tile. The element inserted into the queue is a lightweight pointer to the submatrix that needs to be accumulated, so the dequeueing process will issue get operations to retrieve the data before accumulating. Push and pop operations are performed atomically to ensure no updates are lost.

Algorithms

RDMA Stationary C Algorithm

The most straightforward RDMA-based algorithm is the stationary C algorithm, in which each processor iterates through its tiles of the C matrix, and, for each tile, iterates through the corresponding row block of A and column block of B, retrieving each of the remote tiles via `get_tile` and multiplying them together using a local matrix multiply operation. Pseudocode for a basic implementation of this algorithm is shown in Algorithm 3. Since the method `get_tile` does not require any coordination with remote processors, each process can execute its work independently, and does not need to wait on or synchronize with other processors. While the pseudocode includes a barrier statement at the end, unless the entire result is required immediately, processes are actually free to exit the distributed matrix multiply without synchronization and immediately proceed to do other work. In our actual matrix multiply implementations, matrix multiplications are performed lazily, and users can force a materialization if needed using a barrier statement.

In order to achieve good performance, a few optimizations are required, including prefetching tiles for the next iteration, which allows overlap of computation and communication, and an iteration offset in the inner loop, which balances which tile is requested within each row and column and generally ensures the first remote get is to a local tile. These optimizations are discussed in detail in Section 7.3.

RDMA Stationary A and B Algorithms

The RDMA-based stationary A algorithm is similar to the stationary C algorithm, except we assign work based on which processor owns which tile of A. Each process iterates through its tiles of A, and, for each tile, iterates through the corresponding tile row of B, pulling in each tile and performing a local matrix multiply. Each of these partial results must be accumulated into different tiles of C, each of which is potentially owned by a different process. In order to accumulate these partial results into their individual tiles of C, a global pointer to the local result is pushed to a queue on the remote process where it needs to be accumulated. At some point, the remote process will pop the pointer off of its queue, retrieve the remote partial result tile, and accumulate it into the correct local tile. Pseudocode for

the RDMA-based stationary A algorithm is shown in Alg 5. The stationary B algorithm is very similar, except work is assigned based on which processes own which tile of B.

The stationary B algorithm is very similar, except work is assigned based on which processes own which tile of B. Each process will iterate through its tiles of B, and for each tile of B, iterate through the corresponding tile column of A, retrieving the tiles of A, performing a local matrix multiply, and pushing a pointer to the partial result onto the remote queue of the process that owns the tile of C to which the partial result must be accumulated.

Workstealing Algorithms

In addition to supporting generally asynchronous implementations of distributed matrix multiply, RDMA also allows for more straightforward implementations of workstealing algorithms, in which processes that have finished their work can steal work from processes that are overburdened. In our workstealing algorithms, we use lightweight 2D and 3D reservation schemes to allow processors to claim work. First, processors attempt to perform their work as normal, iterating through each of the tiles that they own. However, before performing any work, they first issue a remote fetch-and-add operation, executed using RDMA, to reserve the work. If some of a process' work is stolen, it will move on to the next piece of work available. After it has completed all of its normal work, each process will iterate through a number of other processes, checking to see if they have work available to be stolen.

Random workstealing The simplest workstealing strategy is random workstealing, where work is stolen randomly without regard to locality. This has the advantage that many blocks are available to be stolen, with the disadvantage that performing stolen work will usually be more expensive than performing regular work, since the tiles of A, B, and C must all be communicated, as opposed to just two. Random workstealing uses a 2D work grid corresponding to the tiles of the stationary matrix. Each element in the work grid contains an integer that is initialized to zero. To claim a piece of work involving a tile, processes perform a fetch-and-add operation on the corresponding element of the work grid. The integer value returned corresponds to the piece of work that has been claimed. Pseudocode for this workstealing algorithm is shown in Alg 8 using the stationary A method.

Locality-Aware workstealing In locality-aware workstealing, instead of stealing randomly, each process will only steal pieces of work where it owns one of the components, ensuring that the cost of performing stolen work will be similar to if it had been performed by the origin process. Locality-aware workstealing requires a 3D work grid, where element i, j, k corresponds to the component matrix multiply $C[i, j] += A[i, k] * B[k, j]$. Processes will first perform work associated with the original stationary distribution, and then attempt to steal work items for which they have one of the three components.

Algorithm 8 Stationary A SpMM with random workstealing.

```
def attempt_work(i, k):
  # Remote atomic fetch-and-add to reserve work
  my_j = reserve_grid[i, k]++
  while my_j < N:
    local_a = A.get_tile(i, k)
    local_b = B.get_tile(k, j)
    local_c = local_a*local_b

    queue[C.owner(i, j)].push(get_ptr(local_c))
    my_j = reserve_grid[i, k]++

# Do work for my tiles
for i in 0..M-1:
  for k in 0..K-1:
    if A.owner(i, k) == rank():
      attempt_work(i, k)

# Attempt to steal work
for idx in 0..M-1:
  i = (rank()+idx) / M
  k = (rank()+idx) % M
  attempt_work(i, k)

while local_c_ptr = queue[rank()].pop():
  C.my_tile() += get_tile(local_c_ptr)
barrier()
```

7.6 Performance Model

In this section, we build a performance model to characterize and predict the performance of our RDMA-based matrix multiply algorithms, first computing the expected cost of communication before using this communication analysis to build an *inter-node* roofline model, which characterizes performance in terms of the ratio of network traffic to local work.

As in our performance model for distributed dense matrix multiply, assume a two-dimensional grid of matrix tiles, where the matrices A $(m \times k)$, B $(k \times n)$ B, and C $(m \times n)$ are distributed amongst p processors laid out in $\sqrt{p} \times \sqrt{p}$ tile grids. Tile dimensions for A, B, and C are then $\frac{m}{\sqrt{p}} \times \frac{k}{\sqrt{p}}$, $\frac{k}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$, and $\frac{m}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ respectively, with $\sqrt{p}$ tile columns and $\sqrt{p}$ tile rows in each matrix. For a dense matrix, the total number of elements in each tile is the product of its dimensions, while for a sparse matrix we approximate the number of nonzeros per tile as the matrix density d multiplied by the product of the tile dimensions.

For our stationary C RDMA-based algorithm for SpMM, in each iteration, each process will issue two remote get operations, one to retrieve a sparse tile of A, and one to retrieve a dense tile of B. Therefore, in each iteration, each process will retrieve exactly $\frac{kn}{p}$ elements in the dense tile of B, along with $\frac{dmk}{p}$ elements in a CSR data structure, for a total

communication cost of $\frac{kn}{p} + 2\frac{dmk}{p} + \frac{m}{\sqrt{p}} + 1$ elements.

As in the dense case, we also must apply an iteration offset of $i + j$ to the k loop, as demonstrated in Section 7.3. This ensures that in an evenly distributed matrix the communication does not suffer from imbalance due to processes within a tile row or tile column requesting tiles of the matrix using the same iteration order.

Next, we present an *inter-node* roofline model that predicts the performance of each iteration of the distributed matrix multiplication. The roofline model predicts the maximum possible performance of a computational kernel depending on its *arithmetic intensity*, or number of operations performed per byte that must be fetched from memory. Typically, the roofline model is used to characterize serial or multi-threaded compute kernels in terms of how compute or bandwidth intensive they are, where the limit on "compute" is defined by the processor's arithmetic peak, and "bandwidth" is determined by memory bandwidth. Here, we develop an *inter-node* roofline model to characterize performance of our distributed memory, RDMA-based matrix multiply algorithms. In this instance, the bandwidth involved is network communication, and the roofline peak of local operations serves as our "arithmetic peak." In our *inter-node* roofline model, we compute the inter-node arithmetic intensity of each iteration as the number of flops performed divided by the number of bytes that must be communicated over the network. The flat portion, providing the "roof" on performance of our inter-node roofline, is the *local* roofline peak for our local SpMM or SpGEMM calls.

We first build a regular *local* roofline model to characterize the performance of our local matrix multiplies. For local SpMM and SpGEMMs, precise roofline models can be difficult to compute, often requiring analysis of the actual matrices themselves, since they depend not only on the size of the matrices and the number of nonzeros, but also the sparsity patterns of the matrices involved. For SpMM, we can calculate an approximate upper bound on arithmetic intensity as the number of flops to be performed in the SpMM divided by the total size of the sparse and dense matrices in bytes. This upper bound assumes perfect cache performance for B and C, and depending on the nonzero pattern, values may have to be reloaded from memory, resulting in lower performance. For a local SpMM operation multiplying a tile of A times B, with the tile sizes derived above, and adding the variable w, which is the number of bytes per word, we compute the arithmetic intensity of our local SpMM operations.

$$\textit{local SpMM AI} = \frac{2(\frac{dmk}{p})(\frac{n}{\sqrt{p}})}{w(2\frac{dmk}{p} + \frac{m}{\sqrt{p}} + 1 + \frac{mn}{p} + \frac{kn}{p})}$$

That is, the number of flops to be performed, divided by the total number of bytes in A, B, and C. To calculate the local roofline peak, which is the "roof" on our inter-node roofline model, we multiply this arithmetic intensity by the memory bandwidth of the local processor B, followed by a max operation with the arithmetic peak.

To compute the arithmetic intensity of an iteration of our distributed SpMM algorithm using the *inter-node* roofline model, we divide the number of flops performed, which is the same expression from the numerator of the local SpMM arithmetic intensity, by the total

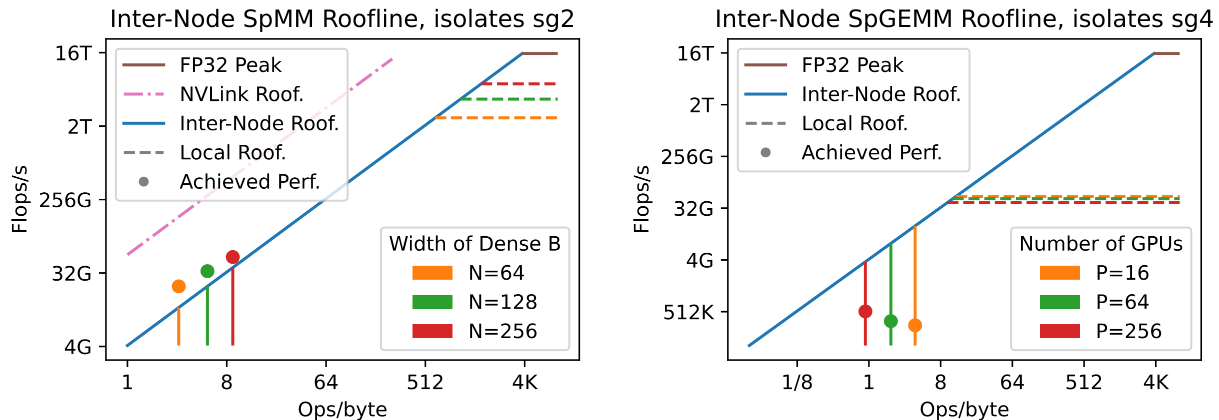


Figure 7.4: *Inter-node* roofline plots for SpMM and SpGEMM with a 2D distribution. SpMM plot models performance for different widths of the dense B matrix at a fixed scale (24 GPUs), while SpGEMM models performance at different scales. Dashed horizontal lines represent *local roofline peaks* for SpMM and SpGEMM operations, while vertical lines represent *inter-node* roofline peaks for particular problems.

number of bytes to be communicated over the network, equal to the total number of bytes in the tiles of A and B.

$$\textit{inter-node SpMM AI} = \frac{2(\frac{dmk}{p})(\frac{n}{\sqrt{p}})}{w(2\frac{dmk}{p} + \frac{m}{\sqrt{p}} + 1 + \frac{kn}{p})}$$

We can then use this roofline to characterize the performance of SpMM on the Summit supercomputer. Figure 7.4 shows the roofline plot for our inter-node SpMM roofline model, using the isolates subgraph2 sparse matrix, and evaluating performance for various numbers of columns in the dense matrix. In the plot, the blue sloping portion represents the region that is bound by network communication, and as such the bandwidth-bound portion has a slope of 3.83 GB/s, which is each GPU's share of injection bandwidth on Summit. The brown roofline at the top is the 32-bit floating point arithmetic peak, 16 TFlops/s for an Nvidia Tesla V100 GPU. Each of the other dotted horizontal lines represents a *local* roofline peak, which in the inter-node roofline model replaces the arithmetic peak. The vertical solid lines represent the roofline peak for a particular problem size, while the dots indicate achieved performance of our RDMA algorithm. Since all three of the problem sizes plotted are well into the bandwidth-bound portion of the inter-node roofline plot, we expect them to be bound by network communication. The SpMM roofline model was across 24 GPUs. Note that the achieved performance slightly exceeds the roofline bound based on network bandwidth. This is because of the high-speed NVLink interconnect providing higher bandwidth for intra-node transfers.

While we can also construct an inter-node roofline model for SpGEMM, it is not possible to write a general formula for FLOPs performed for a particular matrix size and density. This is because in SpGEMM, the number of FLOPs performed depends on the particular sparsity patterns of the matrices, and as such the number of FLOPs can vary significantly for matrices with the same dimensions and number of nonzeros. In light of this, we use the function FLOPS(A, B) to represent FLOPs performed, and in our roofline plot we use average FLOP values calculated experimentally. As such, we can represent the inter-node roofline model's arithmetic intensity as the number of flops performed divided by the size of A and B.

$$\textit{inter-node SpGEMM AI} = \frac{\text{FLOPS}(A, B)}{w(\frac{2dmk}{p} + \frac{m}{\sqrt{p}} + 1 + \frac{2dkn}{p} + \frac{k}{\sqrt{p}} + 1)}$$

For our local roofline model, we use the bound on local SpGEMM arithmetic intensity by Gu, *et al.*, which is representative of modern SpGEMM algorithms and expresses arithmetic intensity in terms of the compression factor *cf*, the number of flops performed per nonzero output, and the number of bytes to express each nonzero b.

$$\textit{local SpGEMM AI} = \frac{cf}{(3 + 2cf) * b}$$

Note that the roofline model is dependent on the sparsity structure, since *cf* depends on the particular sparse matrices being multiplied. Our inter-node SpGEMM roofline model is plotted in Figure 7.4. To obtain realistic values for *cf* and the number of flops performed *FLOPS(A, B)*, we performed each of the component local SpGEMM operations in the distributed SpGEMM operation for the isolates subgraph4 matrix, recording their values for *cf* and *FLOPS(A, B)* to compute average values for different numbers of GPUs P. In the SPGEMM plot, inter-node roofline peaks are much closer to their local roofline peaks than in the SpMM plot. Thus, our roofline model suggests SpGEMM is significantly less network communication-bound than SpMM, although still communication bound. We discuss more conclusions from our roofline model in Section 7.8.

7.7 Implementation

Communication Layer

As discussed earlier, the current generation of GPU-based supercomputers offers GPUDirect RDMA over Infiniband networks [91], which ensures that issuing one-sided operations will be efficiently executed in hardware. In GPUDirect RDMA, the CPU prepares an Infiniband request, which it sends to the local network interface card (NIC). When the NIC receives the request, it will copy the data directly over the network from GPU to GPU, without staging data through the CPU. When copies take place between GPUs within the same node, the high-bandwidth NVLink fabric is used.

All of our RDMA-based implementations use NVSHMEM for communicating data between GPUs. NVSHMEM uses GPUDirect RDMA over Infiniband to transfer data when the GPUs involved are on separate nodes, and NVLink to transfer data when the GPUs involved are within the same node [90].

Data Structures

We implement sparse and dense distributed matrix data structures using BCL, whose architecture is described in detail in Chapter 3. an RDMA-based distributed data structures library in written in C++. As discussed in Section 7.2, both dense and sparse matrix data structures split the distributed matrix up into evenly sized tiles, which are then assigned to processors using a processor grid, as in ScaLAPACK [39]. As discussed earlier, each process has a copy of a directory of remote pointers, which point to the matrix tiles stored in pinned memory, directly accessible through RDMA operations using NVSHMEM. In the dense case, a single remote pointer points to a dense tile data structure, and in the sparse case three remote pointers point to the values, row pointer, and column index arrays of a compressed sparse row (CSR) data structure. All data is stored directly on GPUs, and can be copied directly from GPU to GPU over the network using BCL's NVSHMEM backend.

Data Access Primitives

In terms of data access primitives, all direct manipulation of data is performed using put and get RDMA operations. To retrieve a tile of the dense matrix, a process will issue a remote get operation to retrieve all or part of a remote tile. Similarly, to write to the tile, a process can issue a remote put operation. In the sparse case, reading from a tile involves issuing remote get operations to retrieve each of the value, row pointer, and column index arrays in a CSR data structure. Due to the complexities involved in modifying a CSR data structure in place, only the process that owns a sparse tile is permitted to modify it. This is done by calling a function `replace_tile()` to replace the old tile with a new one, followed by a collective function `renew_tiles()` that will make all sparse tile modifications visible to other processes. As discussed in Section 7.5, when a process needs to send an update to a remote tile, as in the A and B stationary algorithms, we use a distributed queue to send a pointer to the update to the process that owns a particular tile, who will then perform an accumulation at the destination tile. For the update queue, we use BCL's `CheckSumQueue`, which enqueues a pointer using a single fetch-and-add operation and a remote put, while allowing simultaneous enqueues and dequeues.

As discussed earlier, the primary primitive for accessing remote tiles of the matrix is `get_tile()`, which fetches a remote tile of the matrix, and has the same API for both dense and sparse matrices. In the asynchronous version, we return a future object, which will return the local object when the method `get()` is called. In order to ensure the best possible performance and simplify memory management, we allocate most of each GPU's memory as shared memory that can be addressed remotely by NVSHMEM, and use our own memory

allocator to allocate memory when necessary. This (1) ensures that the local destination buffers in remote get operations are allocated in the shared memory segment, which is a requirement for NVSHMEM transfers over Infiniband, and (2) reduces the overhead of memory allocation during the algorithm, since `cudaMalloc` tends to be much less efficient than using a custom memory allocator.

MPI SUMMA Implementations

For one comparison benchmark, we also implement the bulk synchronous SUMMA algorithm using CUDA-aware MPI. In the MPI implementations, we use `MPI_Bcast` to broadcast tiles of the matrix using row and column communicators. We use the CUDA-aware API in IBM Spectrum MPI to perform our broadcasts directly on data that sits on the GPU, and IBM Spectrum MPI is configured to take advantage of GPUDirect RDMA to copy data directly from GPU-to-GPU over the Infiniband network. Note that the MPI SUMMA implementation only runs on square processor grids, so we run this implementation on perfect square numbers of processors.

7.8 Evaluation

We evaluated our RDMA-based sparse matrix multiply implementations in both single-node and multi-node environments. Multi-node experiments are run on the Summit supercomputer at the Oak Ridge Leadership Computing Facility (OLCF) at Oak Ridge National Laboratory. A Summit node has 6 Nvidia Tesla V100 GPUs, each with 16 GB of memory, and the nodes are connected with dual-rail EDR InfiniBand with a link bandwidth of 23 GB/s. Within a node, the GPUs are connected with Nvidia's NVLink interconnect. Single-node experiments are run on a DGX-2 system that is part of the Bridges-2 system at the Pittsburgh Supercomputing Center. The DGX-2 has 16 Nvidia Tesla V100 GPUs with 32 GB of memory, which are fully connected by Nvidia's NVLink interconnect. Both systems use NVLink 3.0, which provides 50 GB/s of link bandwidth. We ran experiments for matrices whose single-GPU runtime took less than a second in the single-node DGX-2 environment, while we used the multi-node environment for matrices which took longer.

On Summit, all codes were compiled with GCC 8.1.1, CUDA 11.2.0, and IBM Spectrum MPI 10.3.1.2. On the single-node DGX-2 system, codes were compiled with GCC 10.2.0, CUDA 11.1.0, and Open MPI 4.0.5. Our RDMA-based implementations use NVSHMEM 2.0.2. All performance experiments use CuSPARSE for local matrix computation. In addition to our own SpMM and SpGEMM implementations, we also benchmark CombBLAS's GPU SpMM and PETSc's GPU SpGEMM [28, 18]. We corresponded with CombBLAS's authors on how to build and execute the `gpu` branch of CombBLAS on Summit, and with OLCF staff for how to properly build PETSc with GPU support.

The sparse matrices in our experiments are detailed in Table 7.4. Load imbalance is defined as $max_{i,j}(\text{nnz}(A_{i,j}))/ave_{i,j}(\text{nnz}(A_{i,j}))$ where $A_{i,j}$ is the matrix assigned to a single

Sparse matrix (A)	kind	$m = k$	nnz(A)	load imb.
Mouse Gene	Biology	45.1K	29.0M	2.13
ldoor	Structural	952K	46.5M	8.23
reddit	GNN	233K	115M	1.08
nlpkkt160	NLP	8.3M	230M	9.46
Amazon Large	GNN	14.3M	230M	3.78
com-Orkut	NMF	3.1M	234M	8.15
Nm7	Eigen	5.0M	648M	6.38
Nm8	Eigen	7.6M	592M	6.48
Isolates, Subgraph4	Biology	4.4M	327M	1.00
Isolates, Subgraph2	Biology	17.5M	5.2B	1.00
Friendster	Graph	62.5M	3.4B	7.68

Table 7.4: Matrices used in our experiments. "load imb." lists the imbalance in number of nonzeros when split amongst 100 processes on a 10×10 2D process grid.

processor $P(i, j)$. We multiply these matrices against dense matrices with 128 and 512 columns. All matrices use 32-bit floating point values, and for indices, we use 32-bit integers except on "Isolates, Subgraph2" and "Friendster," which require 64-bit indices due to their size.

SpMM Experiments

To evaluate our RDMA-based algorithms for sparse times dense matrix multiplication (SpMM), we performed strong scaling experiments using a number of different sparse matrices multiplied by dense matrices of different widths.

Figures 7.5 and 7.6 show the results for a variety of different algorithms. The RDMA-based algorithms include a stationary C algorithm ("S-C RDMA"), a stationary A algorithm ("S-A RDMA"), stationary A algorithm with random workstealing ("R WS S-A RDMA"), and locality-aware workstealing with stationary A and C distributions ("LA WS S-C RDMA" and "LA WS S-A RDMA"). Bulk synchronous benchmarks include our own CUDA-aware MPI SUMMA implementation ("BS SUMMA MPI"), and CombBLAS's GPU SpMM kernel ("CombBLAS GPU"). We use matrix sizes representing typical workloads used in a range of real-world applications, such as Graph Neural Networks (GNNs), iterative methods, and graph algorithms[66, 107, 65].

The implementations generally scale well up to at least 8 GPUs on the single-node DGX-2 system in Figure 7.5, where each GPU has 50 GB/s of bandwidth. Some versions suffer a slowdown at 16 GPUs, especially on the smaller Nm7 and Nm8 matrices, where at 128 columns of B there is insufficient work to keep the GPUs fully occupied and still amortize

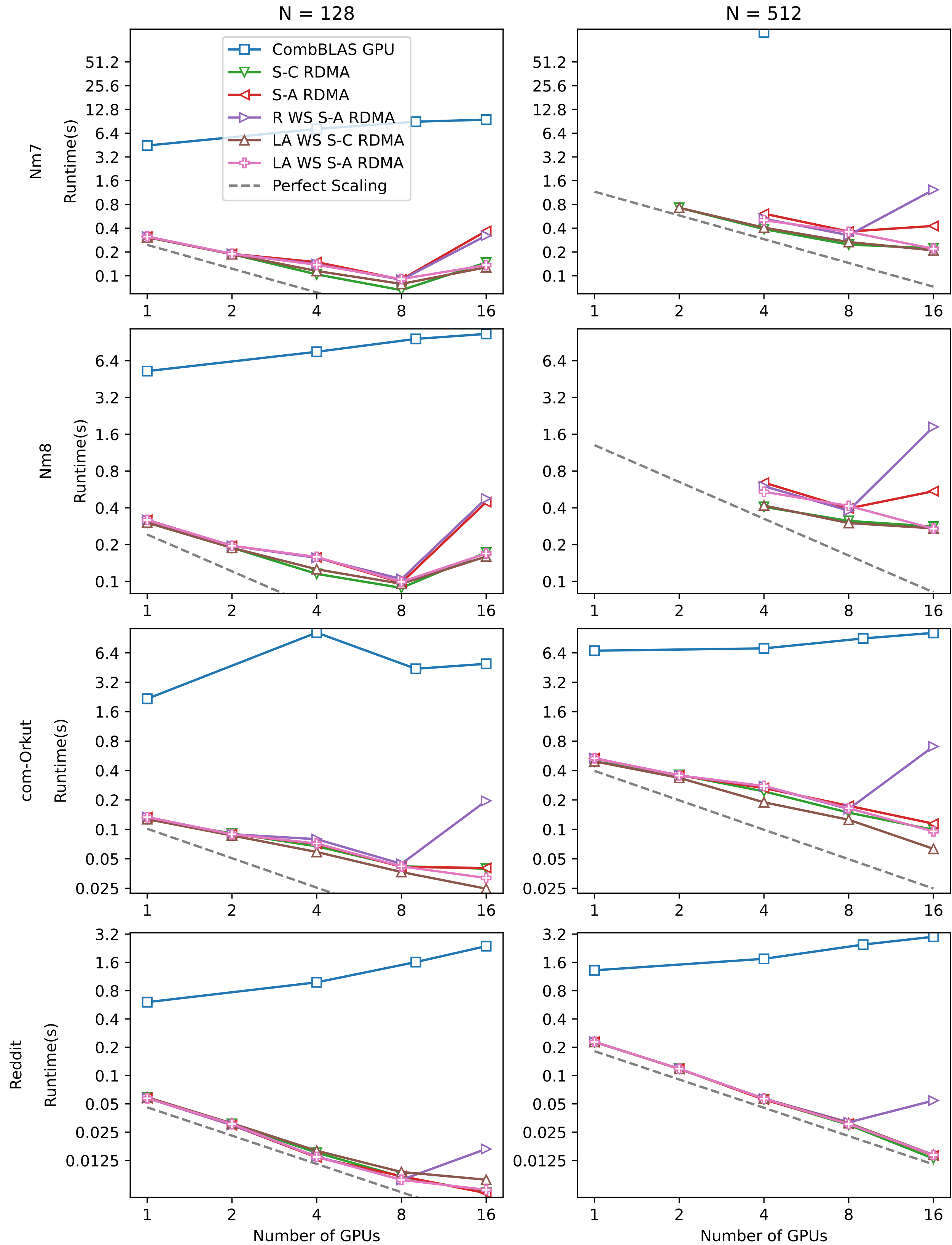


Figure 7.5: Single-node runtimes for SpMM, with different numbers of columns N in the dense matrix B.

data movement. This is particularly true for the stationary A algorithms because of an increase in the number of accumulation operations that become necessary as the number of tiles, and thus the number of local matrix multiply operations, increases.

For the larger matrices run on distributed memory in Figure 7.6, the shared links give each GPU only 3.8 GB/s on average. This significantly limits scalability, especially for the communication-bound case with a smaller dense B matrix. However the RDMA-based implementations outperform the bulk synchronous implementations for most of these configurations, and that benefit is greatest when the number of columns in the dense matrix B is lowest. This is likely because these problems have less computation and are therefore more sensitive to communication cost, both message latency and load imbalance from communicating unequal size blocks of A. A larger B matrix increases the both computation time and time for well-balanced communication of B. The stationary A algorithms have the same issue of increased accumulation cost as on the single node system, but due to the communication-bound nature of the multi-node problems, the overall effect is not as pronounced.

The multi-node scaling results are also influenced by the fact that some transfers happen over the local NVLink fabric, which is much faster than Infiniband. As the number of GPUs increases, the number of transfers that occur over local NVLink fabric decreases, and the lower bandwidth between nodes plays a larger role.

Scaling performance is best on larger matrices, such as isolates, and scaling is also better with wider B matrices. The reasons for this are twofold: (1) there is simply more work to be done with a larger dense matrix, and so scaling performance is better, and (2), as demonstrated by our inter-node roofline performance model in Figure 7.4, the wider the B matrix, the more arithmetically intense the operation becomes, and the less bound by network communication.

The relative performance of the RDMA SpMM algorithms generally depends on the relative sizes of the A, B, and C matrices. If one of the matrices is particularly large, then it is most efficient to keep that matrix stationary while communicating the two other, smaller matrices. When the sparse matrix is not too large in relation to the sizes of the dense matrices, stationary C performs best, which follows from the fact that it performs accumulations locally, removing the need for remote queues, which add some additional overhead. For problems where the sparse matrix is sufficiently larger than the dense matrices, the stationary A algorithm performs best. This trend can be seen in the multi-node plots for the matrix friendster, where stationary A performs better for $N = 128$ (meaning the dense B matrix is small), but that is reversed for $N = 512$ (meaning the dense B becomes bigger). As with most real-world graph applications, all of our sparse matrices are square, which means that the B and C matrices are the same size. This means there is no benefit in terms of communication volume to leaving B stationary instead of C, even while stationary B adds additional overhead due to remote queues. As such, we do not explore a stationary B implementation for SpMM.

Our RDMA-based workstealing algorithms are able to achieve performance improvements for some of the less load balanced matrices, such as com-Orkut and friendster, particularly at higher node counts, where the smaller amount of work per node is more likely to lead to some

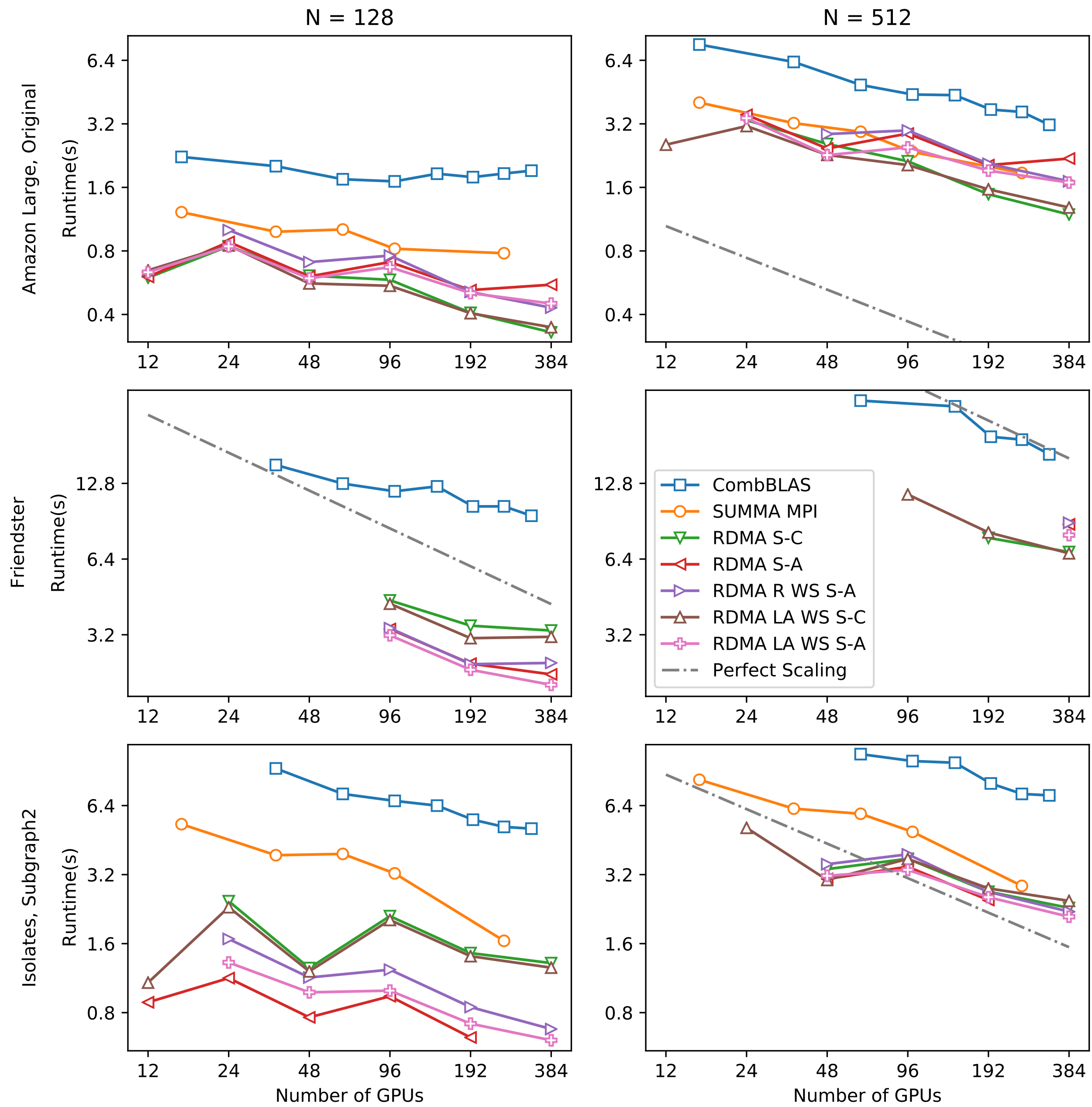


Figure 7.6: Multi-node runtimes for SpMM, with different numbers of columns N in the dense matrix B.

nodes being left with uneven amounts of work. The locality-aware workstealing algorithms tend to perform much better than the random workstealing algorithm, which often incurs a high cost when performing workstealing, since it is not guaranteed that any of the involved matrix tiles will be local. In Table 7.5a, we can see that the locality-aware workstealing algorithm is able to reduce the amount of time lost due to load imbalance compared to the other RDMA implementations. It should be noted that load imbalance numbers for the RDMA implementations cannot be directly compared to the bulk synchronous MPI implementation, since time lost to load imbalance is tied up in MPI's broadcast operations, which perform communication, but also enforce synchronization.

SpGEMM Experiments

To evaluate our RDMA-based SpGEMM algorithm, we perform the computation $C = AA$ for a number of different sparse matrices, representative of a number of different graph algorithms, such as Markov clustering [45], triangle counting [10], and transitive reduction [56]. We again benchmark RDMA-based stationary C and A algorithms, along with a work-stealing version, a SUMMA implementation in MPI, and PETSc's SpGEMM using cuSPARSE.

Strong scaling performance results are shown in Figure 7.7. Unlike SpMM, both the single- and multi-node environments are more likely to be compute bound. As a result, all the implementations have similar performance, because they are all performing the same computations. The only exception is PETSc, which is significantly slower, probably because it is not utilizing GPUDirect RDMA. On the large isolates matrix, our bulk synchronous CUDA-aware MPI implementation matches that of the RDMA implementation. On mouse gene, which is somewhat smaller and less load-balanced, we observe good scaling until very high concurrencies, but the RDMA-based implementations achieve higher overall performance as well as somewhat better scaling. On nlpkkt160, which has an uneven distribution of nonzeros, we observe poor performance in the multi-node experiments due to imbalance in both communication and computation. However, the RDMA implementations still perform better than the bulk synchronous implementations. Unlike SpMM, we do not meet our SpGEMM model's roofline bound on Summit, as shown in Figure 7.4. This is because the *local* cuSPARSE operations are not meeting the local roofline bound, evidenced by SpGEMM being compute bound in Table 7.5b. This likely indicates that there are opportunities for optimization in the local cuSPARSE SpGEMM operations.

For single-node experiments, we achieve better scaling on the highly load imbalanced matrices nlpkkt160 and ldoor, likely because communication imbalance is eliminated due to the significant increase in bandwidth, along with the fact that at lower concurrencies each GPU is left with a larger portion of the matrix, reducing the likelihood of load imbalance.

Env.	Matrix	Alg.	#GPUs	Comp.	Comm.	Acc.	Load Imb.
Summit	Amazon	S-C	24	0.02	1.3	-	0.4
			192	˜0	0.5	-	0.2
		S-A	24	0.02	1.1	0.4	0.3
			192	˜0	0.6	0.2	0.2
		S-C LW	24	0.01	1.2	0.03	0.3
			192	˜0	0.6	0.01	0.1
		MPI	16	0.02	2.1	-	-
			256	0	1.0	-	-
DGX-2	Nm-7	S-C	4	0.44	0.01	0.02	0.14
		S-C	16	0.19	˜0	0.09	0.10
		S-A	4	0.42	˜0	0.07	0.12
		S-A	16	0.09	0.00	0.02	0.10
		MPI	16	0.64	0.51	0.01	0.15
			256	0.08	0.15	0.01	0.08

(a) Component breakdown for SpMM, N = 256 columns.

Env.	Matrix	Alg.	#GPUs	Comp.	Comm.	Acc.	Load Imb.
Summit	Mouse Gene	S-C	24	0.02	1.3	-	0.4
			192	˜0	0.5	-	0.2
		S-A	24	0.02	1.1	0.4	0.3
			192	˜0	0.6	0.2	0.2
		S-C LW	24	0.01	1.2	0.03	0.3
			192	˜0	0.6	0.01	0.1
		MPI	16	0.02	2.1	-	-
			256	0	1.0	-	-
DGX-2	Mouse Gene	S-C	4	2.14	˜0	0.02	0.72
			16	0.55	˜0	0.01	0.21
		S-A	4	2.14	˜0	0.38	0.49
			16	0.55	˜0	0.11	0.13

(b) Component breakdown for SpGEMM.

Table 7.5: Component breakdown for selected matrices.

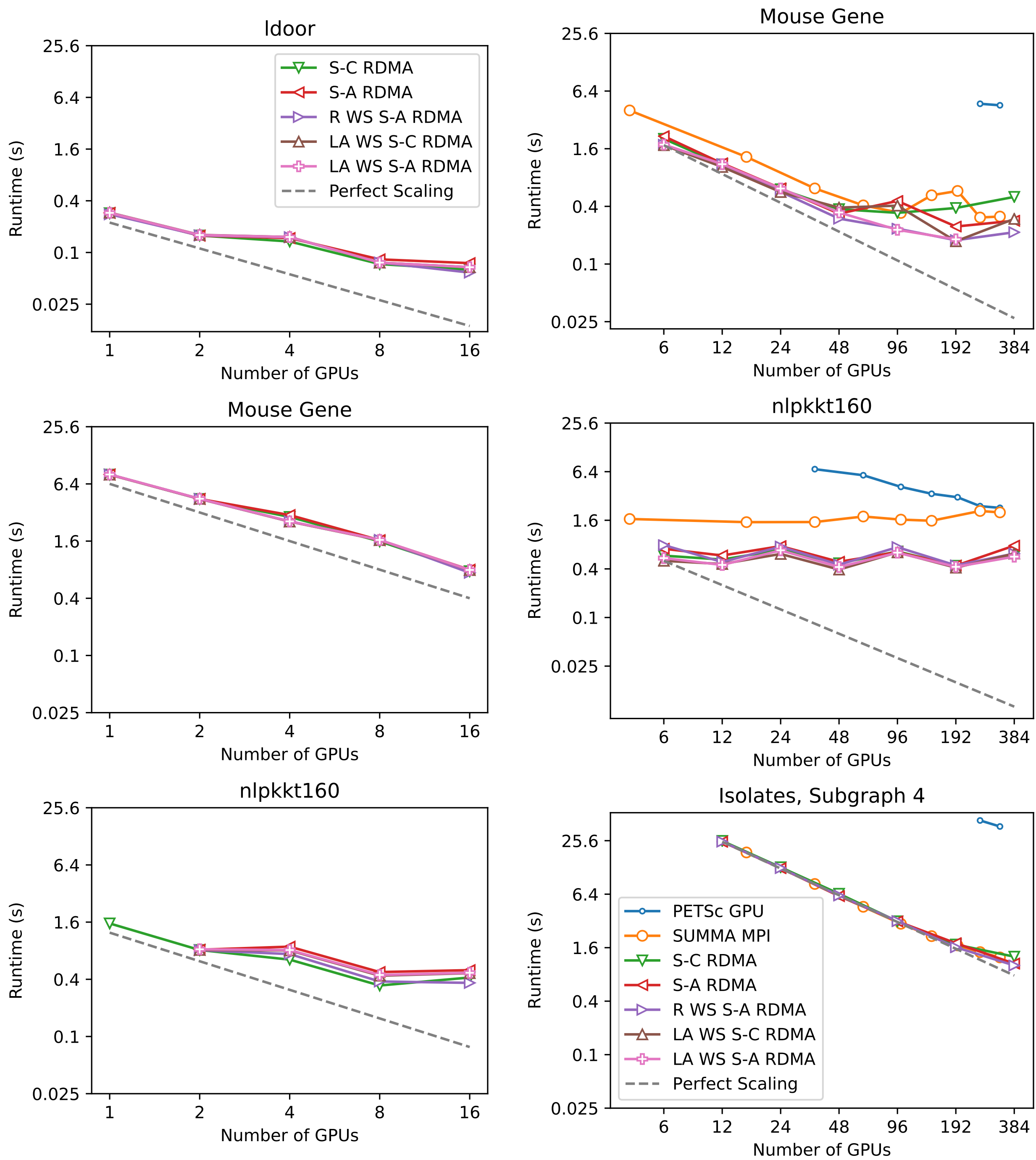
7.9 Related Work and Conclusions

The C++ PGAS library DASH, which uses RDMA through MPI's one-sided communication API, includes dense matrix and vector data structures as well as an implementation of stationary C dense matrix multiplication [48, 82]. However, DASH does not include any support for sparse matrices or GPUs.

The Combinatorial BLAS library (CombBLAS) [12] is an MPI-based library that provides distributed data structures for dense and sparse matrices, also implementing a number of distributed sparse matrix multiply algorithms. While focusing more on tensor computations, Cyclops Tensor Framework (CTF) [102] also supports several distributed sparse matrix multiplication algorithms.

Distributed matrix-matrix multiplication has seen a sustained interest among the HPC community. One of the most popular developments has been communication-avoiding (CA) algorithms [19] that replicate input, output, or intermediate data to asymptotically reduce communication. CA algorithms have been applied to SpGEMM [13] and SpMM within the context of graph neural network training [107]. Both CombBLAS and CTF implement CA algorithms for sparse matrices. Our work on asynchronous, RDMA-based implementations is orthogonal to CA algorithms, and we leave RDMA-based implementations of communication avoiding algorithms to future work.

This work explores a large space of possible communication and load balancing optimization for two sparse matrix operations, which are key to many simulation, analysis, and learning applications. We demonstrate that RDMA based communication can have a significant advantage over global collectives for some problems, and that dynamic work stealing can also contribute to improved scaling. Our extensive set of experiments on multiple matrices and two different machine configurations also highlight the importance of having high speed networking hardware that is balanced with the computational capabilities for this class of algorithms. Our inter-node roofline model provides a useful tool for evaluating performance and identifying opportunities for optimization.



(a) Single-node experiments.

(b) Multi-node experiments.

Figure 7.7: SpGEMM strong scaling experiments.

Chapter 8

Comparing RDMA and RPC-Based Distributed Data Structures

8.1 Introduction

As discussed in Chapter 1, many complex programs need to perform operations on abstract data structures, such as hash tables, queues, and arrays. While many mature, high quality libraries exist that provide implementations of abstract data structures for serial and multi-threaded programs, the development of techniques for high-level data structures for distributed programs is still an active area of research [3, 26, 48]. Of particular interest are distributed data structures for *irregular applications*, where data access patterns and volumes are not known in advance. These applications commonly use data structures which may be complex to implement using traditional message passing methods in a distributed memory setting, including graphs, trees, hash tables, and distributed queues. The primary focus of this work has been to develop distributed data structures for irregular applications using remote direct memory access (RDMA) primitives, which allow processes to directly manipulate regions of the memory of other processes over the network.

As discussed in detail in Chapter 3, RDMA operations include basic operations to read, write, or otherwise manipulate remote memory, including remote put, get, and atomic operations. These primitives can be executed directly by the network interface card (NIC) on most modern supercomputer and datacenter systems, which allows them to be executed in a very low latency and efficient manner. As discussed throughout this work, it is possible to implement distributed data structures using only these primitives for all essential data structure operations [26, 48]. This work uses the Partitioned Global Address Space (PGAS) model to build distributed data structures, which augments these one-sided remote memory access primitives with a model for how shared memory segments will be allocated and exposed, as a collection of partitioned, globally addressable address spaces [5].

However, there are other parallel programming models, discussed in detail in Chapter 2, that could potentially be used to implement distributed data structures. One such example

includes programming models based on remote procedure call (RPC) primitives [17]. Unlike the PGAS programming model, which exposes low-level memory access primitives like remote put, remote get, and remote atomics, RPC allows users to invoke functions on remote processes. Individual RPC invocations generally come at a higher latency cost than individual remote memory operations do in a PGAS environment, since RPCs require interrupting a remote CPU to handle the request. RDMA operations, on the other hand, can be executed directly by the NIC, which allows them to be executed with only a couple of microseconds latency. However, since it can execute an arbitrary function, a single RPC can capture complex control flow, such as searching through a hash table for an empty spot, that would require multiple round trips over the network to complete with only RDMA operations.

RDMA primitives can be (and often are) used to implement RPC runtimes. In fact, the buffering technique presented in Chapter 6.1, whereby hash table insertions are pushed onto queues, where they will later be locally inserted by the remote process, could be considered a type of specialized RPC runtime. In this chapter, we are concerned with a restricted set of latency-bound RDMA data structure operations in which the process manipulating the data structure does so solely using RDMA operations, without relying on the help of a remote process.

In this chapter, we evaluate some of the RDMA-based distributed data structure operations presented in this work and compare them with RPC-based implementations written using GASNet-EX. First, we build a performance model to analyze the cost of our RDMA-based data structure operations in terms of their component RDMA primitives, as well as the cost of the RPC-based implementations based on the cost of a GASNet-EX active message. We run microbenchmarks to experimentally determine these component costs. Next, we benchmark the actual costs of remote queue and hash table insertions, using different levels of coordination in the RDMA-based implementation. We then compare these results with those of our cost model to evaluate the costs of using our pure RDMA-based methods versus our RPC-based implementation.

By breaking down the cost of RDMA-based data structure operations in terms of an analytical cost model can help us determine why RDMA-based operations are expensive, when they are expensive. In particular, we identify locking operations that can require multiple retries over the network in the presence of contention as a particular pain point for RDMA. We then briefly discuss ways of addressing this, which could include avoiding this level of coordination when possible, using an RPC-based implementation, which can perform this coordination using cheaper local atomics, or investigating new types of hardware that could support more expressive RDMA operations. In summary, this chapter has three main goals:

- Evaluate the cost of various component operations for RDMA-based distributed data structures on a modern supercomputer network using a collection of microbenchmarks.
- To develop cost model that can be used to estimate the cost of RDMA-based distributed data structure operations, based on these component costs.

- To compare RDMA- and RPC-based distributed data structure performance for queue and hash table data structures at variable levels of concurrency requirements.

8.2 Background

Remote Direct Memory Access

As discussed in more detail in Chapters 1, 2, and 3, remote direct memory access (RDMA) provides an interface to manipulate remote data in a *one-sided* manner, meaning that an origin process can perform operations on the remote memory of a target process without any explicit coordination with the target. This is commonly executed by having the target's network interface card (NIC) directly communicate with its on-node memory, resulting in very low round-trip latency on the order of a microsecond. Low-latency RDMA primitives are now available on a number of supercomputer interconnects, including Cray Aries and Infiniband. RDMA is also increasingly available on datacenter commodity hardware through RDMA over converged Ethernet (RoCE).

As discussed in detail in Chapter 3, we consider a common set of RDMA operations available in most modern supercomputer and datacenter systems. This set includes remote put and remote get, which can be of variable size, along with the fixed-size 32 and 64-bit atomic memory operations (AMOs) compare-and-swap and fetch-and-op. Some have proposed an expanded set of RDMA operations to support various types of remote and distributed data structures, such as the Infiniband extended atomics API [68]. In addition, there are recently proposed API extensions to RDMA which would allow for more expressive RDMA operations [3]. These APIs are outside the scope of this chapter.

RDMA-Based Data Structures As discussed throughout this work, distributed data structures can be directly built on top of one-sided RDMA operations, so that all major data structure operations will be executed with RDMA. Examples of such partitioned global address space (PGAS) distributed data structure libraries include BCL, DASH, and Multipol [26, 48, 31]. Similar to shared memory concurrent data structures, these libraries are built to use a shared global memory space, with synchronization using atomics when necessary, to operate upon shared data. However, unlike shared memory data structures, the component costs and synchronization models of distributed programming frameworks can be quite different, so care must be taken to design data structures accordingly. As shown in Figure 8.1, libraries can use RDMA operations, which will be directly executed by the target process' NIC, to operate on remote data. There are two remote memory operations in this code example, `CAS`, which is a remote compare-and-swap operation, and `RPUT`, which is a remote put operation. In the best case, our inserting process will perform a remote compare-and-swap, succeed in reserving the first hash table slot, and then perform a remote put operation. This would have a cost of $A_{\text{CAS}} + W$, that is the cost of a compare-and-swap operation and a write. However, in the case of hash table collision, the algorithm will move

```
void insert(key, val) {
  slot = hash(key);
  while (!success) {
    rval = CAS(flags + slot,
               free_flag, taken_flag);
    success = (rval == free_flag);
    if (!success) {
      slot++;
    }
  }
  if (success) {
    RPUT(data + slot, {key, val});
  }
}
```

Figure 8.1: Modifying a hash table using one-sided RDMA operations.

on to the next available slot, and multiple round trips may be required to perform the insert operation. The particular hash table shown here is a hash table with open addressing and linear probing. Observant readers of Figure 8.1 will also notice that the listed code is not fully atomic. While the code is atomic with respect to concurrent insert operations, there is no guarantee that the remote put operation will finish before a remote find operation reads the half-written value. If we wish for our insert operation to be atomic with respect to concurrent find operations, we will require a second fetch-and-op operation to mark the slot as ready for reading after the remote put operation has finalized. This would increase the best case cost of the remote insert operation to $A_{\text{CAS}} + W + A_{\text{FAO}}$. So, depending on an application's atomicity requirements, data structure operations over RDMA may have different best-case costs. Also, depending on a particular execution of the application, the observed cost of a method may vary due to the number of round trips caused by contention.

Remote Procedure Calls

Remote procedure calls (RPCs), in contrast to RDMA operations, allow an origin process to remotely trigger the execution of a procedure on a target process. RPCs have the advantage of being more expressive than RDMA. While control flow in individual RDMA operations is limited to single-instruction atomics like compare-and-swap and fetch-and-op, RPCs can include complex control flow and arbitrary computation. This allows more complex data structure operations, such as inserting into a hash table, pushing a value onto a queue, or even modifying a dynamically sized data structure, to be performed with a single communication event. However, this added expressivity comes at a greater latency cost, since an RPC

```
void insert_handler(key, val) {
  local_hash.insert({key, val});
}

void insert(key, val) {
  node = hash(key) % nprocs();
  rpc(node, insert_handler, key, val);
}
```

Figure 8.2: Modifying a hash table using an RPC.

operation must wait for the target process to enter a progress function or interrupt the processor and make a function call to execute the procedure. It also changes load balancing across processors, moving away from the clearly defined SPMD model of execution in ways that can shift computational workload, intentionally or not.

In this chapter, we consider a restricted type of RPC called an active message (AM) [47]. AMs have the following restrictions: (1) active message handlers may not send additional active messages, except for a single response to the origin process and (2) active message handlers may not perform network communication. These restrictions allow for a high-performance, low-latency implementation of active messages with bounded buffer space [23, 24].

RPC Data Structures An implementation of a distributed data structure operation with RPCs requires two parts: (1) a handler function, which is the procedure that will be executed on the target process, and (2) a wrapper function, which is the function directly called by the user on the origin process and the code that will issue the RPC request. RPC data structure implementations can be quite simple, as shown in Figure 8.2. The wrapper function `insert` uses a hash function to map data to the appropriate nodes, then issues an RPC with the handler function `insert_handler`. The handler function in this case simply inserts the key and value into a local hash table. In contrast to the hash table implementation based on RDMA communication, this implementation will typically only require a single round trip over the network, since the origin node can push the RPC request onto the target node's RPC queue in a single network operation, then the target node can execute the necessary control flow to unpack and store the data. However, crucially, the number of network operations is unrelated to the control flow logic inside the data structure operation, which takes place on the target side inside the RPC function. Depending on the specific manner in which the handler function will be called (either serially or simultaneously with other threads), the handler function may require local atomic operations or other mechanisms for synchronization. However, these local mechanisms are significantly cheaper than remote memory operations.

Name	Notation	Latency (us)
put	W	3.0
get	R	3.7
compare-and-swap	A_{CAS}	3.8
fetch-and-op	A_{FAO}	3.9

Table 8.1: Latency of various RDMA operations, measured on Cori with 64 nodes.

One important detail not directly illustrated by the above code listing is that the execution of the handler function is dependent on the attentiveness of the target process, which must enter a progress function in order for its RPC queue to be serviced. While the liveness of RDMA operations is guaranteed by the network interface card, which will be constantly servicing instructions regardless of CPU state, RPC-based systems must either dedicate specific resources, such as a progress thread, to ensure attentiveness, or else pay the possible latency cost associated with waiting until the target process finishes its computation and enters a call to the RPC progress function.

The Berkeley Container Library

In this chapter, we compare the performance of RDMA-based implementations of distributed data structures to RPC-based implementations. For the RDMA-based implementations, we will benchmark data structures provided in the Berkeley Container Library (BCL), discussed throughout this work. We benchmark the queue and hash table data structures discussed in Chapters 5 and 6, respectively.

Performance Model Data structure operations in BCL can be characterized in terms of an analytical cost model, which characterizes the best-case costs of data structure operations in terms of the component RDMA operations. The component costs include remote get, remote put, compare-and-swap, and fetch-and-op operations. We do not distinguish different fetch-and-op operations in this performance model, since the operations involved are typically simple binary functions such as fetch-and-add or fetch-and-XOR, which have very low cost compared to the inherent network latency. A summary of these operations and the associated notation are shown in Table 8.1.

Alternate Implementations As discussed in Section 8.2, there are different levels of concurrency requirements with which RDMA-based data structure operations can be implemented, depending on the specific needs of an application. BCL exposes multiple implementations of data structure operations using a mechanism called *concurrency promises*, which allows users to optionally specify the operations that could occur concurrently with

Method	Concurrency Level	Description	Cost
`insert`			
(a)	Concurrent Read/Write (C_{RW})	Fully Atomic Insert	$A_{\text{CAS}} + W + A_{\text{FAO}}$
(b)	Concurrent Write (C_{W})	Phasal Insertions	$A_{\text{CAS}} + W$
`find`			
(c)	Concurrent Read/Write (C_{RW})	Fully Atomic Find	$A_{\text{FAO}} + R + A_{\text{FAO}}$
(d)	Concurrent Read (C_{R})	Phasal Finds	R

Table 8.2: RDMA-based hash table method implementations considered in this paper.

the operation being issued. To illustrate the different levels of concurrency requirements with which a data structure operation could be implemented, consider the case of a hash table insertion with arbitrarily large keys and values. Inserting an element into such a hash table will, in the general case, require at least two atomic memory operations and a write. The first atomic memory operation requests a lock on the bucket into which the element will be inserted, the write actually writes the value into the distributed hash table, and a final unlock operation signals that the bucket is ready to be read after the write hash completed. In this hash table implementation, without the final atomic memory operation, concurrent find operations might read halfway written data, resulting in an incorrect program execution. However, in the guaranteed absence of concurrent find operations within a barrier region, we can elide the final atomic memory operation, since the following barrier will ensure that the write completes before any find operations may be issued.

Similar levels of concurrency requirements exist for both hash table insert and find operations, as well as operations on queues. Tables 8.2 and 8.3 show some of the data structure implementations available in BCL's hash table and queue implementations, along with the associated best case costs. In the notation used in this paper, C_{W} indicates that an operation is allowable with concurrent writes (pushes or inserts), while C_{R} indicates that an operation is allowable with concurrent reads (pops or finds) and C_{RW} indicates the operation is allowable with either.

GASNet Active Messages

GASNet is a communication library that offers remote procedure call functionality in the form of active messages. Active messages are a restricted form of RPC, in that (1) active message handlers cannot require network communication, and (2) active message handlers cannot send additional active messages except for request handlers, which may send a single reply to the host. Since neither of these are necessary for the class data structure operations we consider in this paper and GASNet is known for having a high-quality, fast implementation of active messages, we use GASNet to implement a set of RPC-based distributed data structure implementations to compare against BCL's RDMA-based data structures.

Method	Concurrency Level	Description	Cost
`push`			
(a)	Concurrent Read/Write (C_{RW})	Fully Atomic	$A_{\text{FAO}} + W + A_{\text{CAS-P}}$
(b)	Concurrent Write (C_{R})	Only Pushes	$A_{\text{FAO}} + W$
(c)	Concurrent Local (C_{ℓ})	Local Push	ℓ
`pop`			
(d)	Concurrent Read/Write (C_{RW})	Fully Atomic	$A_{\text{FAO}} + RA_{\text{CAS-P}}$
(e)	Concurrent Read (C_{R})	Only Pops	$A_{\text{FAO}} + R$
(f)	Concurrent Local (C_{ℓ})	Local Pop	ℓ

Table 8.3: Implementations for circular queue methods.

GASNet-EX active messages consist of a fixed-size header, which includes an index referencing the desired handler function, and up to 64 bytes of arguments, along with an optional variable length payload. Active message handlers must be registered with the GASNet runtime before they can be used. When an origin process wishes to invoke an active message on a remote target process, it issues an active message request. A target process, inside the context of an active message handler, can optionally issue an active message reply to the origin process, which will result in the corresponding reply handler running on the origin process. To wait for the completion of an individual active message, an origin process must wait until a reply handler issued by the target process has finished running locally, writing some reply data or otherwise indicating that it has completed.

In order to service active messages, each process must enter the GASNet AM poll function, which can be called by the main process or from a progress thread that constantly checks the queue in order to provide attentiveness.

8.3 Experimental Design

In this section, we examine the performance of RDMA- and RPC-based designs for remote operations on distributed data structures. Our expectation is that there are some data structures, applications, and workloads for which RPC, with its greater expressiveness, will achieve higher performance, and some situations where RDMA, with its lower round trip latency and hardware-accelerated execution in the network interface card, will achieve higher performance. First, we perform microbenchmarks to measure the *component costs* for RDMA- and RPC-based distributed data structure implementations. That is, the cost of a remote put, remote get, compare-and-swap, and fetch-and-op operations, as well as the round trip cost of sending a GASNet active message request and receiving a reply.

Component Benchmarks

For the component benchmarks, we measure the cost of small-size remote put and remote get operations, along with the cost of performing the atomic compare-and-swap and fetch-and-op operations. In each of these microbenchmarks, we begin with a globally visible array located in the shared segment of each process in the program. To perform the benchmark, each process will continuously perform a remote memory operation to a random location in a random process' globally visible array. After the benchmark completes, we divide the total amount of time taken by the number of operations completed per process to arrive at the latency of the individual operations.

We also measure the cost of sending an active message. In this case, each process continually sends an active message to another random process and then waits for a reply to complete before proceeding.

The purpose of collecting these benchmarks is not only to evaluate the relative costs of the operations that make up RDMA-based distributed data structure methods, but also to evaluate BCL's analytical performance model. By plugging in the component method costs into the formulas for different data structure operations, we can evaluate to what extent observed performance deviates from theoretical best case performance. Performance could deviate for a number of reasons, including higher than optimal latency due to specific application workloads stressing the network hardware and high contention leading to many more round trips than would be necessary in the best case. This analysis will allow us to evaluate what makes up the cost of RDMA-based remote data structure operations, which can both allow data structure developers to better design data structure operations, prioritizing use of cheaper component operations, and allow hardware designers to identify which RDMA operations to optimize in order to increase RDMA-based data structure performance.

Data Structure Benchmarks

After collecting microbenchmark results, we ran a series of experiments where we benchmarked different distributed data structure operations in BCL and compared them with equivalent active message implementations. Full descriptions of these distributed data structure implementations can be found in the original BCL paper [26].

Hash Table

Distributed hash tables are an important data structure for many applications, including various data analysis problems such as genomics. In BCL, hash tables are implemented as a distributed array of buckets, where each bucket contains room for a key, a value, and a flag that will be used for synchronization.

$C_{\mathbf{RW}}$ **Insert** As discussed earlier, to achieve fully concurrent safety, a hash table insertion requires, in the best case, two atomic memory operations and a remote put operation. In

the BCL implementation, this is a compare-and-swap operation to request the hash table bucket, a remote put, and an atomic fetch-and-AND operation to mark the bucket as ready to read.

$C_{\mathbf{W}}$ **Insert** An insert operation without find concurrently occurring can be preformed using using only one atomic memory operation, setting the bucket's flag to ready, then performing a remote put. The collective barrier that must separate the inserts from any find operations will guarantee that the remote put has completed before any find operations can read the value.

$C_{\mathbf{RW}}$ **Find** To perform a fully concurrently safe find operation, the reading process must first obtain a read lock on a bucket before reading what is inside it. This is to prevent other processes from modifying the bucket while the process is reading it. In BCL's implementation, this is done with an atomic fetch-and-OR operation to set one of a number of read bits in the flag. After the lock is obtained, the process will read the bucket with a remote get operation, then unset the read bit with an atomic fetch-and-AND.

$C_{\mathbf{R}}$ **Find** A find without inserts concurrently occurring can be performed with a single remote get operation, since it is able to retrieve both the flag and the key and value in a single remote memory operation.

Queue

Queues are widely used data structures in many applications such as data redistribution and asynchronous all-to-all operations, producer-consumer problems, frontier based graph algorithms, and others. The most crucial operations for queues are pushing and popping. Since these operations are symmetric, we only consider push operations here for reasons of brevity. Queues in BCL are implemented as *remote* data structures, meaning they live on a single host process, but are visible to all processes. Applications may either require a single queue to be manipulated by all or a subset of processes, or, commonly, a queue on each process. Queues are implemented as a ring buffer, with sets of head and tail pointers marking the beginning and end of data stored within the queue.

$C_{\mathbf{RW}}$ **Push** Similar to the hash table insertion example, with this queue implementation, a fully concurrently safe push requires two atomic memory operations and a remote put operation. The first operation is a fetch-and-add operation, which requests space in the queue by advancing the tail pointer, followed by a remote put to the reserved space. Finally, a remote compare-and-swap operation is necessary to advance the ready tail pointer to indicate that the written segment of the queue is ready to be read. A fetch-and-add operation is not correct here, since it could advance the ready tail pointer past previous insertions which are not yet finished writing.

C_W Push A push without pops concurrently occurring can be completed with a single atomic fetch-and-add, to reserve space, followed by a remote put to write to the reserved spot in the queue. This is because a barrier will separate all the push operations from any pop operations, thus guaranteeing that the written data is ready to be read.

RPC Implementations

Our RPC implementations, based on GASNet-EX active messages, consist of a handler function, which performs the relevant operation on a local data structure then sends a reply. If the operation has no return value, the reply will simply increment a counter on the origin side, which can be used to ensure fine-grained completion of data structure operations. If the operation has a return value, it will also write the return value to a memory location passed in by the request AM.

8.4 Results

First we measured our set of *component costs*, which include individual operations that make up remote data structure operations. Measured operations include 32-bit remote get, remote put, atomic fetch-and-add, atomic compare-and-swap, and a round-trip active message with a payload of 64-bits. Each process picks a random process for each operation, and, for the case of the RDMA operations, a random memory location. The active message experiment measures the cost of a round trip with a 64-bit payload, with the inner operation being an insertion into a remote hash table.

Experimental Setup We measured the component costs on Cori Phase I, which is a Cray XC40 supercomputer with a Cray Aries interconnect and 32 cores per node. All benchmarks were run with one process per core. Experiments were run with 100,000 local elements per process, unless noted otherwise. In each experiment, the target operation is executed a million times inside a loop, then the total time spent inside the loop is divided by the number of issued operations to calculate the operation's latency. In order to avoid systematic errors due to variations in network performance, which may be caused by varying job placement within a cluster as well as contention with other jobs, each dot on each graph was submitted as a separate job submission, averaged over at least four jobs. More details about the system used in these experiments is given in Chapter 4.

Component Benchmarks Our component benchmark results are shown in Figure 8.3. This figure includes the benchmarks discussed above, along with two extra versions for the two measured atomic operations.

Compare-And-Swap We show two benchmarks for compare-and-swap, "Single CAS," which measures the cost of a single compare-and-swap operation and "Persistent CAS,"

which measures the cost of a compare-and-swap which continually polls until it succeeds in changing the value from the previous value to a new value. The Single CAS experiment measures the cost of the CAS AMO operation, while the Persistent CAS experiment provides some measure of the cost of a persistent CAS that repeatedly polls until it succeeds in modifying the target value.

Fetch-And-Add We include two versions of the fetch-and-add operation, "FAD," which measures the cost of a fetch-and-add operation issued to a random value on another process, and "FAD, Single Variable," which measures the cost of a fetch-and-add operation issued when there is only a single target variable per process.

In general, there is a large jump in latency for RDMA operations when moving from a single node to two nodes, which is to be expected when switching from shared memory to distributed memory. From there, each operation increases in cost gradually, which can be explained by (1) the decreasing percentage of operations that will be operating on local, fast memory and (2) an increase in the distance that messages must travel across the network as the allocation size increases.

We find that the Put operation has the lowest cost, followed by a cluster of operations of similar cost, including Get, FAD, and Single CAS. As one might expect, the "Persistent CAS" operations are much more expensive, since they may require multiple round trips to succeed in swapping the target value. More surprisingly, we also find that the "Single FAD" benchmark, which operates on a single target value per process, has a much higher cost than the "FAD" benchmark, which operates on a range of values per process. This indicates that, for this operation, the *target memory locations*, not just the number of incoming operations on the target NIC, can impact the amount of time that a fetch-and-add operation will take (at least on the Cray Aries NIC). While this might be expected for a shared memory environment, where a directory or snooping protocol must be used to ensure cache coherence, NIC-accelerated atomic memory operations are not atomic with respect to CPU atomics, and the authors expected the speed of NIC-accelerated fetch-and-add atomics to be unaffected by the target address. Indeed, experiments on the Summit supercomputer, which has an Infiniband FDR interconnect, did not reveal any difference between the two benchmarks.

Queue Benchmarks Next, we measured the cost of a set of *queue data structure* microbenchmarks, and compared our empirical data structure benchmark results with our performance model's prediction using the component microbenchmark results. These results are shown in Figure 8.4. Pushing and popping are the primary queue operations, and since they are symmetric, we only show queue push results here. We compare four different queue results: (1) an "AM Push" benchmark, which uses an active message to insert into a local queue on each process, (2) a "RDMA Push C_{W}" benchmark, which is a phasal queue that allows either concurrent pushing or popping, but not simultaneously, (3) an "RDMA Push C_{RW}" benchmark, which uses an additional atomic operation to signal the completion of the write of data onto the queue, and (4) a "RDMA Checksum Push C_{RW}" benchmark, which

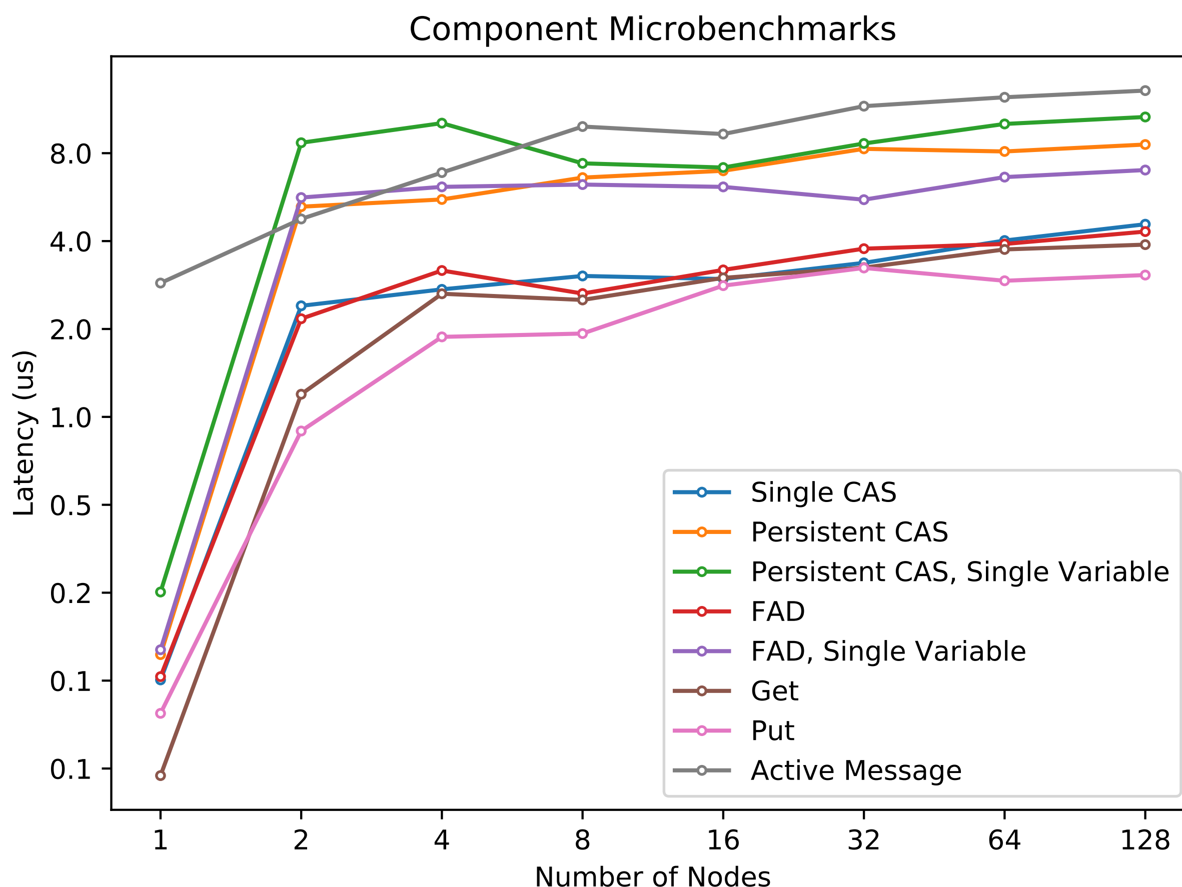


Figure 8.3: The component latencies for RDMA operations and AMs on Cori.

is a separate design of queue that in addition to writing the data values into the queue also writes a checksum value that can be used to verify whether the write has finished.

After accounting for the increased cost of the "Single FAD" experimental result and plugging that in as the parameters for our analytical performance model in Table 8.1, we found that the performance model generally predicted the behavior of the queue data structure benchmarks. The one exception is the RDMA Push (C_{RW}) benchmark, which is consistently more expensive than the performance model would predict. This appears to be due to the fact that the second atomic memory operation in the C_{RW} push operation requires more round trip attempts than the Persistent CAS microbenchmark would suggest. While our Persistent CAS microbenchmark attempts to change the target value from the previously seen value to the desired value, the persistent CAS involved in the queue benchmark attempts to increase a "tail ready" pointer that marks the frontier of values written into the queue. It may only proceed after any other insertions have finished writing, which leads to

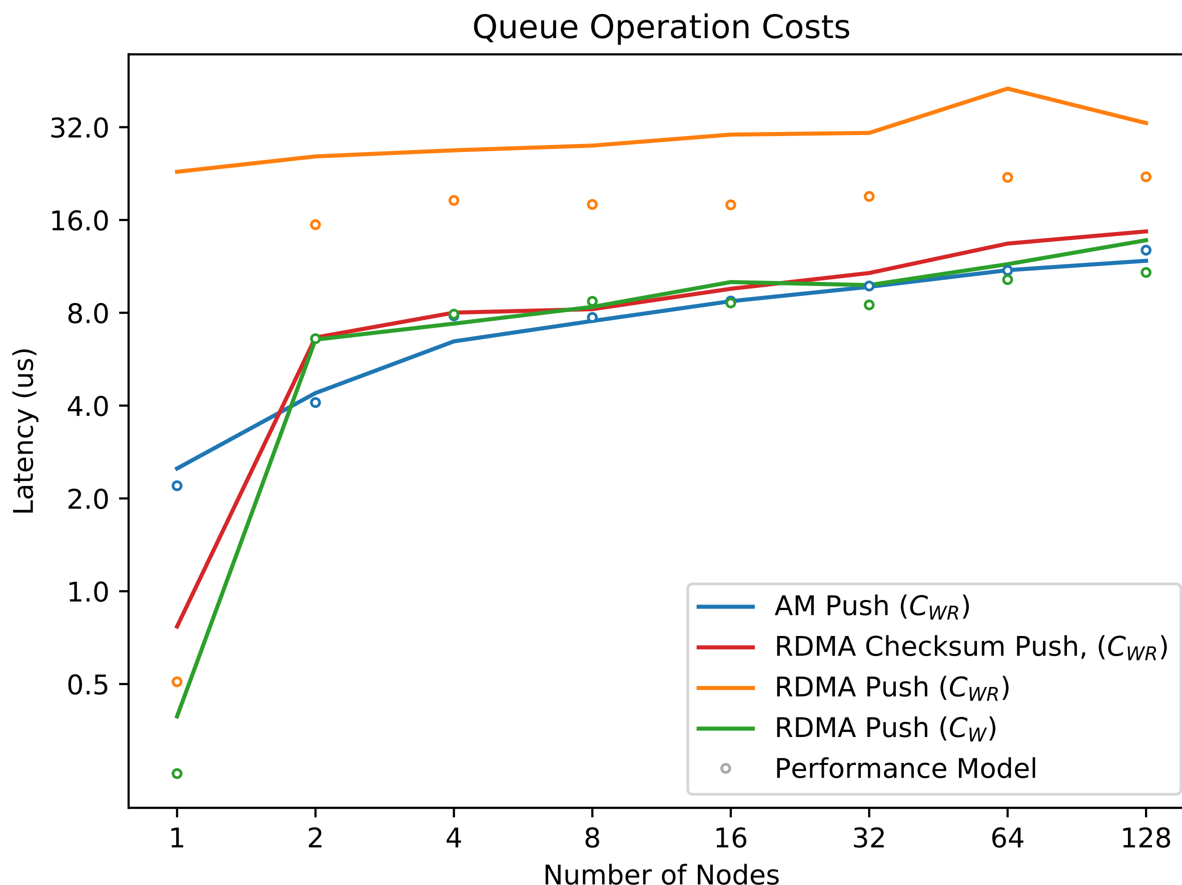


Figure 8.4: Latencies for RDMA- and RPC-based queue push operations.

some inherent serialization that is not represented in the performance model.

Hash Table Benchmarks We also measured the cost of several hash table data structure operations, which are displayed in Figure 8.5. The less expensive RDMA Find (C_{R}) operation is the cheapest operation, followed by the active message implementations of AM Find (C_{RW}) and AM Insert (C_{RW}), with find possibly having a slightly higher cost, due to the fact that the return trip message is slightly larger, containing a return value. The more expensive RDMA Find (C_{RW}) is initially slightly more expensive than the active message implementations, but appears to scale better, ending up at a similar cost at 128 nodes. The insert implementations, both C_{RW} and C_{W}, are more expensive. Both RDMA operations seem to roughly match their associated performance models, with some increase in the real benchmark runtime perhaps due to hash table collisions, which are not included in the performance model. Surprisingly, both hash table insertion methods vary significantly

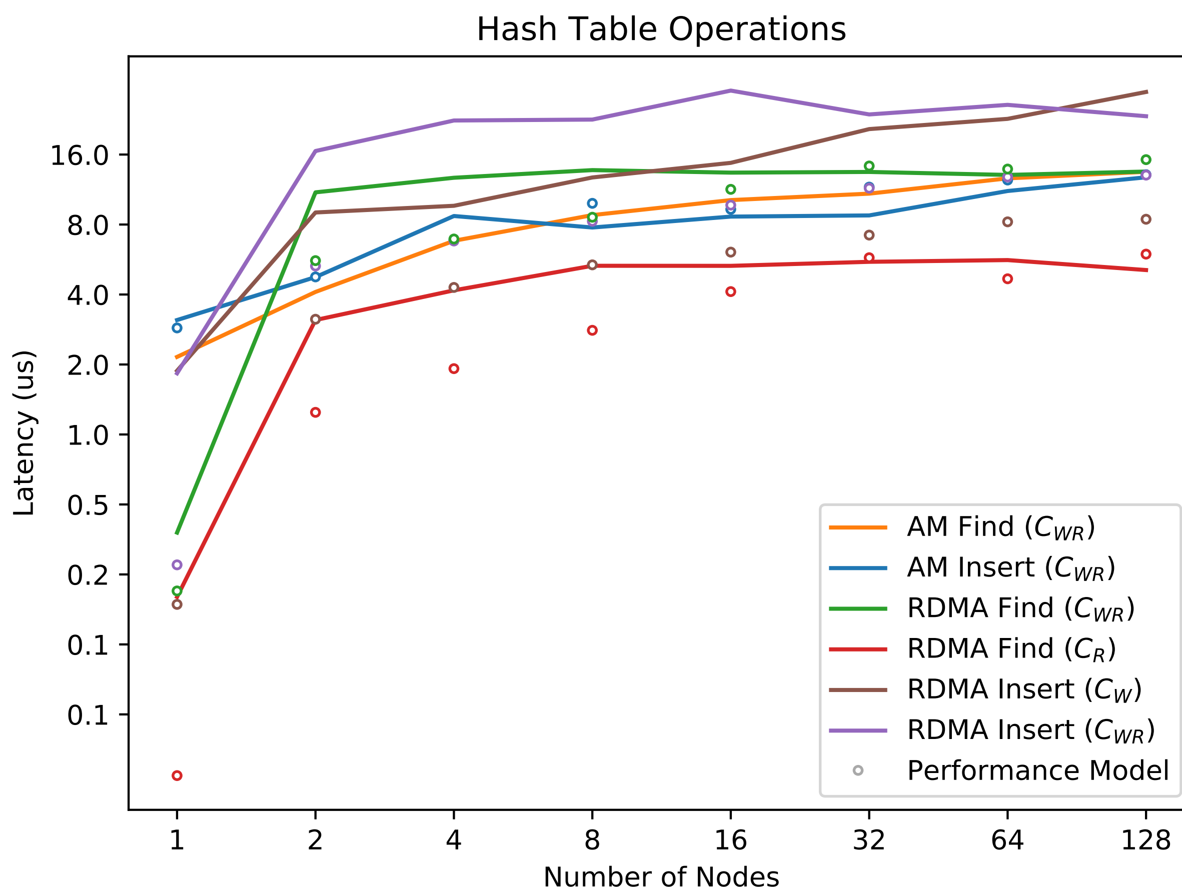


Figure 8.5: Latencies for RDMA- and RPC-based hash table operations.

from their associated cost models' prediction. However, except for RDMA Insert C_W, the predicted order of implementations in terms of performance is correct.

Attentiveness Benchmarks Each of the above active message benchmarks could be considered a close to optimal case in terms of *attentiveness*, by which we mean the availability of remote processes to service active message requests. This is because, without an independent progress thread to ensure attentiveness, which is the model assumed in the above benchmarks, a process must enter a progress function in order to ensure that inbound requests are serviced. In each of the above benchmarks, each process issues a single active message request, then polls on a progress function, servicing incoming active messages, until a reply is received for the active message request. In a more realistic scenario, remote data structure operations will be interspersed with computation, which may impact the attentiveness of remote processes, resulting in longer latencies for active messages. Figure 8.6

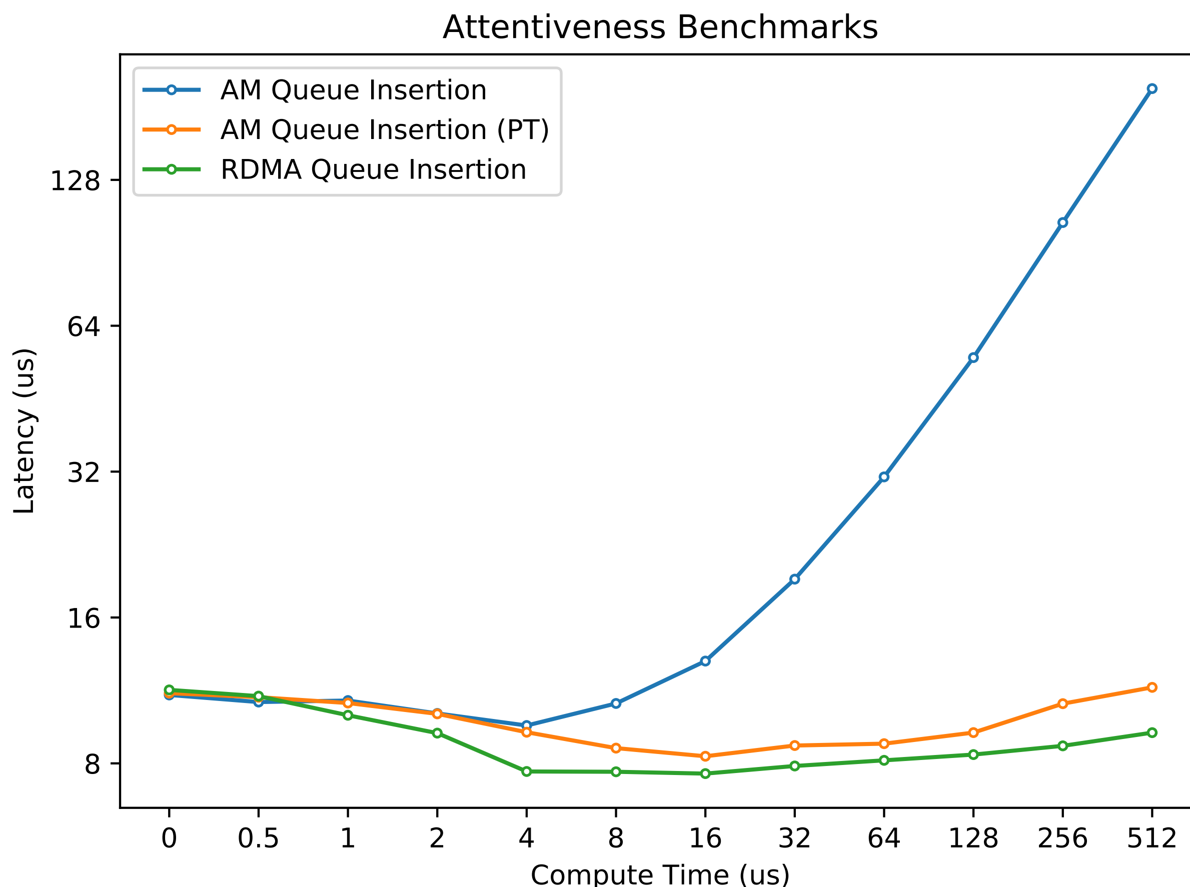


Figure 8.6: Measuring the cost of a queue insertion as remote processes become less attentive due to intermixed computation.

shows the impact of adding interspersed computation on the latency of a queue insertion. We arrived at this plot by inserting a small function to perform a given number of microseconds of computation inside a loop of queue insertions. To calculate only the time spent performing queue insertions, we subtract the compute time from the total time taken in the calculation. As shown in the plot, the active message version, while initially faster than the RDMA-based implementation, quickly becomes more expensive as the interspersed computation time exceeds 2 microseconds. From there, the increase in queue insert time grows roughly linearly with the compute time, as the average time an active message must wait to be serviced is half the compute time. We observe that the latency for the RDMA queue insertion actually goes down, which we attribute to lower latency across a quieter network as queue insertions are spaced out among a greater quantity of computation. It is important to note that RPC-based implementations can attain better attentiveness by explicitly using

a progress thread, which will continually poll for new RPC operations to execute. In the figure, “AM Queue Insertion (PT)” demonstrates this. When using a progress thread, active messages are not subject to the same pathological behavior due to lack of attentiveness, but do receive a performance hit, likely due to contention between the progress and main compute thread.

8.5 Conclusions

In this chapter, we compared implementing distributed data structures using RDMA and RPC. We developed an analytical performance model which predicts the performance of the distributed data structures based on their components, then compared this to real-world performance. In most of the cases, our model's predictions matched the real-world results. We observed the impact of system-specific hardware behavior, namely the increased cost of a fetch-and-add performed on a single memory location on Cray Aries; and also observed the impact of increased contention due to multiple round trips, as in the case of a concurrent read/write queue insertion. We also examined the impact of attentiveness on RPC performance, observing that RDMA may have advantages when it comes to communication interspersed with computation.

Chapter 9

Conclusion

In this dissertation we presented a model for building high-level, cross-platform distributed data structures using one-sided RDMA primitives. We showed how, using remote pointers, we can create globally visible data structures where each process has its own view of the entire data structure. This allows for low latency access, since processors can access or manipulate parts of the data structure with only a handful of RDMA operations. Using the remote pointer and ObjectContainer abstractions, we can develop high-level, generic abstractions that support a wide array of types, including complex user-defined types. By carefully designing our types and APIs, we can ensure that we pay little cost for this abstraction at runtime. Modern C++ compilers' aggressive inlining results in efficient code generation, with each of our data structure methods compiling down to a function that makes a few invocations to the backend communication library. By using a modular design to support multiple backends, distributed data structures can be used natively inside parallel programs written using multiple communication libraries. Indeed, we demonstrated that we can configure supplementary backends to support additional memory spaces, including memory that lives on GPUs. This enables us to support GPU-centric communication, which enables GPU applications to achieve higher performance with both CPU and GPU-initiated communication.

We explored a number of different distributed data structures, including several different queue designs. Our generic, RDMA-based queues enabled a high degree of overlap between computation and communication by exposing an asynchronous push operation, allowing for greater performance on sorting benchmarks. We explored different remote queue implementations, exposing multiple levels of consistency guarantees for each using user input about the context in which the queue methods are invoked.

We also developed an RDMA-based hash table. This data structure is also designed for minimum latency, with a locking scheme designed to allow processes to perform insert and find operations in only a few remote memory operations. Similar to our queue design, the hash table allows users to provide information about the context in which hash table operation is being issued, which allows the hash table operations to sometimes elide unnecessary synchronization operations. We introduce a high-level buffering data structure that allows

users to transparently aggregate insertions into the hash table, converting latency-bound, fine-grained insert operations into bulk bandwidth-bound operations. We also develop a novel distributed Bloom filter data structure, using a blocked Bloom filter design to ensure all operations can be completed in a single remote atomic operation.

We designed RDMA-based distributed dense and sparse matrix data structures, which allow processes to asynchronously retrieve and modify any part of the matrix data structure. This allows for the implementation of new RDMA-based asynchronous matrix multiplication algorithms, which are particularly useful for sparse matrix multiplication.

Finally, we built an analytical performance model to breakdown and evaluate the cost of some of our RDMA-based data structure operations in terms of their component remote memory operations. We also evaluated our RDMA-based data structure operations against RPC-based implementations, demonstrating that RDMA-based operations offer competitive performance with RPC-based implementations, matching or exceeding their performance, while not being susceptible to the attentiveness issues that can arise when processes are not constantly entering the RPC progress function.

Overall, we have demonstrated that RDMA-based distributed data structures offer an excellent path toward easing the burden on parallel programmers. They allow for programmers to implement parallel programs at a higher level of abstraction by providing a global data model. The costs paid for this higher level of abstraction are generally low compared to hand-tuned, application specific data structures, and allow for code reuse.

Bibliography

[1] Shrirang Abhyankar et al. "PETSc/TS: A Modern Scalable ODE/DAE Solver Library". In: *arXiv preprint arXiv:1806.01437* (2018).

[2] Seher Acer, Oguz Selvitopi, and Cevdet Aykanat. "Improving performance of sparse matrix dense matrix multiplication on large-scale parallel systems". In: *Parallel Computing* 59 (2016), pp. 71–96.

[3] Marcos K. Aguilera et al. "Designing Far Memory Data Structures: Think Outside the Box". In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. HotOS '19. Bertinoro, Italy: ACM, 2019, pp. 120–126. ISBN: 978-1-4503-6727-1. DOI: 10.1145/3317550.3321433. URL: http://doi.acm.org/10.1145/3317550.3321433.

[4] Marco Aldinucci et al. "An efficient unbounded lock-free queue for multi-core systems". In: *European Conference on Parallel Processing*. Springer. 2012, pp. 662–673.

[5] George Almasi. "PGAS (Partitioned Global Address Space Languages)". In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Boston, MA: Springer US, 2011, pp. 1539–1545. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_490. URL: https://doi.org/10.1007/978-0-387-09766-4_490.

[6] Bob Alverson et al. "Cray XC series network". In: *Cray Inc., White Paper WP-Aries01-1112* (2012).

[7] *Amazon AWS EC2 Instance Types*. https://aws.amazon.com/ec2/instance-types. Accessed March 29, 2022.

[8] Krste Asanovic et al. "The landscape of parallel computing research: A view from berkeley". In: (2006).

[9] *Ascending to Summit, Announcing Summitdev*. https://www.olcf.ornl.gov/2017/02/28/ascending-to-summit-announcing-summitdev. Access March 29, 2022.

[10] Ariful Azad, Aydin Buluç, and John Gilbert. "Parallel triangle counting and enumeration using matrix algebra". In: *IPDPSW*. IEEE. 2015, pp. 804–811.

[11] Ariful Azad et al. "A distributed-memory algorithm for computing a heavy-weight perfect matching on bipartite graphs". In: *SIAM Journal on Scientific Computing* 42.4 (2020), pp. C143–C168.

[12] Ariful Azad et al. "Combinatorial BLAS 2.0: Scaling combinatorial algorithms on distributed-memory systems". In: *IEEE Transactions on Parallel and Distributed Systems* (2021).

[13] Ariful Azad et al. "Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication". In: *SIAM Journal on Scientific Computing* 38.6 (2016), pp. C624–C651.

[14] J Bachan et al. "UPC++ Programmer's Guide (v1. 0-2019.3. 0)". In: (2019).

[15] John Bachan. Personal Communication. 2019.

[16] John Bachan et al. "The UPC++ PGAS library for Exascale Computing". In: *Proceedings of the Second Annual PGAS Applications Workshop*. ACM. 2017, p. 7.

[17] John Bachan et al. "UPC++: A High-Performance Communication Framework for Asynchronous Computation". In: *Proceedings of the 33rd IEEE International Parallel & Distributed Processing Symposium*. 2019.

[18] Satish Balay et al. *PETSc Web page*. `https://www.mcs.anl.gov/petsc`. 2021. URL: `https://www.mcs.anl.gov/petsc`.

[19] Grey Ballard et al. "Minimizing communication in numerical linear algebra". In: *SIAM Journal on Matrix Analysis and Applications* 32.3 (2011), pp. 866–901.

[20] Ray Barriuso and Allan Knies. *SHMEM user's guide for C*. Tech. rep. Technical report, Cray Research Inc, 1994.

[21] Michael Bauer et al. "Legion: Expressing locality and independence with logical regions". In: *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE. 2012, pp. 1–11.

[22] Burton H Bloom. "Space/time trade-offs in hash coding with allowable errors". In: *Communications of the ACM* 13.7 (1970), pp. 422–426.

[23] Dan Bonachea and P Hargrove. "GASNet Specification, v1. 8.1". In: (2017).

[24] Dan Bonachea and Paul H. Hargrove. *GASNet-EX: A High-Performance, Portable Communication Library for Exascale*. Tech. rep. LBNL-2001174. Languages and Compilers for Parallel Computing (LCPC'18). Lawrence Berkeley National Laboratory, Oct. 2018. DOI: `10.25344/S4QP4W`. URL: `https://escholarship.org/uc/item/1g3t9n5n`.

[25] Urban Borštnik et al. "Sparse matrix multiplication: The distributed block-compressed sparse row library". In: *Parallel Computing* 40.5-6 (2014), pp. 47–58.

[26] Benjamin Brock, Aydın Buluç, and Katherine Yelick. "BCL: A Cross-Platform Distributed Data Structures Library". In: *Proceedings of the 48th International Conference on Parallel Processing*. ICPP 2019. Kyoto, Japan: ACM, 2019, 102:1–102:10. ISBN: 978-1-4503-6295-5. DOI: `10.1145/3337821.3337912`. URL: `http://doi.acm.org/10.1145/3337821.3337912`.

[27] Aydin Buluç and John Gilbert. “Parallel Sparse Matrix-Matrix Multiplication and Indexing: Implementation and Experiments”. In: *SIAM Journal on Scientific Computing* 34 (Sept. 2011). DOI: `10.1137/110848244`.

[28] Aydın Buluç and John R Gilbert. “The Combinatorial BLAS: Design, implementation, and applications”. In: *The Intl. Journal of High Performance Comp. Applications* 25.4 (2011), pp. 496–509.

[29] Alhadi Bustamam, Kevin Burrage, and Nicholas A Hamilton. “Fast parallel Markov clustering in bioinformatics using massively parallel computing on GPU with CUDA and ELLPACK-R sparse format”. In: *IEEE/ACM TCBB* 9.3 (2012), pp. 679–692.

[30] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. “R-MAT: A recursive model for graph mining”. In: *SDM*. SIAM. 2004, pp. 442–446.

[31] Soumen Chakrabarti et al. “Multipol: A distributed data structure library”. In: *PPoPP*. 1995.

[32] Bradford L Chamberlain, David Callahan, and Hans P Zima. “Parallel programmability and the chapel language”. In: *The International Journal of High Performance Computing Applications* 21.3 (2007), pp. 291–312.

[33] *Chapel Implementation of ISx*. `https://github.com/chapel-lang/chapel/tree/master/test/release/examples/benchmarks/isx`. Accessed March 10, 2018.

[34] Barbara Chapman et al. “Introducing OpenSHMEM: SHMEM for the PGAS community”. In: *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*. ACM. 2010, p. 2.

[35] Jarrod A Chapman et al. “Meraculous: de novo genome assembly with short paired-end reads”. In: *PloS one* 6.8 (2011), e23501.

[36] Philippe Charles et al. “X10: an object-oriented approach to non-uniform cluster computing”. In: *Acm Sigplan Notices*. Vol. 40. 10. ACM. 2005, pp. 519–538.

[37] Thomas Cheatham et al. “Bulk synchronous parallel computing—a paradigm for transportable software”. In: *Tools and Environments for Parallel and Distributed Systems*. Springer, 1996, pp. 61–76.

[38] Yuxin Chen et al. *Atos: A Task-Parallel GPU Dynamic Scheduling Framework for Dynamic Irregular Computations*. 2021. DOI: `10.48550/ARXIV.2112.00132`. URL: `https://arxiv.org/abs/2112.00132`.

[39] Jaeyoung Choi et al. “Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines”. In: *Scientific Programming* 5.3 (1996), pp. 173–184.

[40] P Colella et al. “Chombo software package for AMR applications design document”. In: *Available at the Chombo website: http://seesar. lbl. gov/ANAG/chombo/(September 2008)* (2009).

[41] UPC Consortium et al. “UPC language specifications v1. 2”. In: *Lawrence Berkeley National Laboratory* (2005).

[42] Timothy A Davis. "Graph algorithms via SuiteSparse: GraphBLAS: triangle counting and k-truss". In: *HPEC*. IEEE. 2018, pp. 1–6.

[43] Timothy A. Davis. "Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra". In: *ACM Trans. Math. Softw.* 45.4 (Dec. 2019). ISSN: 0098-3500. DOI: `10.1145/3322125`. URL: `https://doi.org/10.1145/3322125`.

[44] James Dinan et al. "An implementation and evaluation of the MPI 3.0 one-sided communication interface". In: *Concurrency and Computation: Practice and Experience* 28.17 (2016), pp. 4385–4404.

[45] Stijn van Dongen. "Graph Clustering by Flow Simulation". PhD thesis. University of Utrecht, 2000.

[46] Michael Driscoll. "Domain-Specific Techniques for High-Performance Computational Image Reconstruction". PhD thesis. EECS Department, University of California, Berkeley, Dec. 2018. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-184.html`.

[47] Thorsten von Eicken et al. "Active Messages: A Mechanism for Integrated Communication and Computation". In: *Proceedings of the 19th Annual International Symposium on Computer Architecture*. ISCA '92. Queensland, Australia: ACM, 1992, pp. 256–266. ISBN: 0-89791-509-7. DOI: `10.1145/139669.140382`. URL: `http://doi.acm.org/10.1145/139669.140382`.

[48] Karl Fürlinger, Tobias Fuchs, and Roger Kowalewski. "DASH: A C++ PGAS Library for Distributed Data Structures and Parallel Algorithms". In: *Proceedings of the 18th IEEE International Conference on High Performance Computing and Communications (HPCC 2016)*. Sydney, Australia, Dec. 2016, pp. 983–990. DOI: `10.1109/HPCC-SmartCity-DSS.2016.0140`.

[49] Al Geist et al. *PVM: Parallel virtual machine: a users' guide and tutorial for networked parallel computing.* MIT press, 1994.

[50] Evangelos Georganas et al. "HipMer: an extreme-scale de novo genome assembler". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM. 2015, p. 14.

[51] Evangelos Georganas et al. "Parallel de bruijn graph construction and traversal for de novo genome assembly". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press. 2014, pp. 437–448.

[52] Robert Gerstenberger, Maciej Besta, and Torsten Hoefler. "Enabling highly-scalable remote memory access programming with MPI-3 one sided". In: *Scientific Programming* 22.2 (2014), pp. 75–91.

[53] John R Gilbert, Cleve Moler, and Robert Schreiber. “Sparse matrices in MATLAB: Design and implementation”. In: *SIAM journal on matrix analysis and applications* 13.1 (1992), pp. 333–356.

[54] William Gropp et al. “A high-performance, portable implementation of the MPI message passing interface standard”. In: *Parallel computing* 22.6 (1996), pp. 789–828.

[55] Zhixiang Gu et al. “Bandwidth Optimized Parallel Algorithms for Sparse Matrix-Matrix Multiplication using Propagation Blocking”. In: *SPAA*. 2020, pp. 293–303.

[56] Giulia Guidi et al. “Parallel String Graph Construction and Transitive Reduction for De Novo Genome Assembly”. In: *IPDPS*. IEEE. 2021.

[57] Ulf Hanebutte and Jacob Hemstad. “ISx: A scalable integer sort for co-design in the exascale era”. In: *Partitioned Global Address Space Programming Models (PGAS), 2015 9th International Conference on*. IEEE. 2015, pp. 102–104.

[58] Paul Hargrove and Dan Bonachea. Personal Communication. 2019.

[59] Charles R Harris et al. “Array programming with NumPy”. In: *Nature* 585.7825 (2020), pp. 357–362.

[60] Jacob Hemstad et al. “A study of the bucket-exchange pattern in the PGAS model using the ISx integer sort mini-application”. In: *PGAS Applications Workshop (PAW) at SC16*. 2016.

[61] Maurice Herlihy. “Wait-free synchronization”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13.1 (1991), pp. 124–149.

[62] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. ISBN: 0123705916.

[63] Desmond J Higham and Nicholas J Higham. *MATLAB guide*. SIAM, 2016.

[64] Changwan Hong et al. “Adaptive sparse tiling for sparse matrix multiplication”. In: *PPOPP*. 2019, pp. 300–314.

[65] Yuwei Hu et al. “FeatGraph: A Flexible and Efficient Backend for Graph Neural Network Systems”. In: *SC’20*. 2020.

[66] Guyue Huang et al. “GE-SpMM: General-purpose Sparse Matrix-Matrix Multiplication on GPUs for Graph Neural Networks”. In: *SC’20*. 2020.

[67] Dror Irony, Sivan Toledo, and Alexander Tiskin. “Communication lower bounds for distributed-memory matrix multiplication”. In: *Journal of Parallel and Distributed Computing* 64.9 (2004), pp. 1017–1026.

[68] Yossi Itigin. *Scalable Hardware Atomics over DC Transport*. Invited talk at Conference on Partitioned Global Address Space Programming Models (PGAS). 2014. URL: %7Bhttps://www.csm.ornl.gov/OpenSHMEM2014/documents/Mellanox_Invite_OUG14.pdf%7D.

[69] Sylvain Jeaugey. “Nccl 2.0”. In: *GPU Technology Conference (GTC)*. Vol. 2. 2017.

[70] Hartmut Kaiser et al. “HPX: A task based programming model in a global address space”. In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. ACM. 2014, p. 6.

[71] Engin Kayraklioglu et al. “Locality-Based Optimizations in the Chapel Compiler”. In: *International Workshop on Languages and Compilers for Parallel Computing*. Springer. 2022, pp. 3–17.

[72] Ken Kennedy, Charles Koelbel, and Hans Zima. “The Rise and Fall of High Performance Fortran”. In: *Commun. ACM* 54.11 (Nov. 2011), pp. 74–82. ISSN: 0001-0782. DOI: 10.1145/2018396.2018415. URL: https://doi.org/10.1145/2018396.2018415.

[73] Ken Kennedy, Charles Koelbel, and Hans Zima. “The rise and fall of High Performance Fortran: an historical object lesson”. In: *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. 2007, pp. 7–1.

[74] Jungwon Kim, Seyong Lee, and Jeffrey S Vetter. “PapyrusKV: a high-performance parallel key-value store for distributed NVM architectures”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM. 2017, p. 57.

[75] Christos Kozyrakis and Kunle Olukotun. “The stanford pervasive parallelism lab”. In: *2009 IEEE Hot Chips 21 Symposium (HCS)*. 2009, pp. 1–29. DOI: 10.1109/HOTCHIPS.2009.7478358.

[76] Hyuk-Jae Lee, James P Robertson, and José AB Fortes. “Generalized Cannon’s algorithm for parallel matrix multiplication”. In: *Proceedings of the 11th international conference on Supercomputing*. 1997, pp. 44–51.

[77] Mingzhe Li. “Designing High-Performance Remote Memory Access for MPI and PGAS Models with Modern Networking Technologies on Heterogeneous Clusters”. PhD thesis. The Ohio State University, 2017.

[78] Mingzhe Li et al. “Designing MPI Library with On-Demand Paging (ODP) of InfiniBand: Challenges and Benefits”. In: *SC ’16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2016, pp. 433–443. DOI: 10.1109/SC.2016.36.

[79] F. Miller Maley and Jason G. DeVinney. *Conveyors for Streaming Many-To-Many Communication*. June 2019. URL: https://github.com/jdevinney/bale/blob/master/uconvey.pdf.

[80] Pall Melsted and Jonathan K Pritchard. “Efficient counting of k-mers in DNA sequences using a Bloom filter”. In: *BMC bioinformatics* 12.1 (2011), p. 333.

[81] Philipp Moritz et al. “Ray: A distributed framework for emerging {AI} applications”. In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 561–577.

[82] Felix Mössbauer et al. "A Portable Multidimensional Coarray for C++". In: *PDP*. Cambridge, UK, Mar. 2018.

[83] Shubhendu S. Mukherjee et al. "Efficient Support for Irregular Applications on Distributed-Memory Machines". In: *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP '95. Santa Barbara, California, USA: Association for Computing Machinery, 1995, pp. 68–79. ISBN: 0897917006. DOI: 10.1145/209936.209945. URL: https://doi.org/10.1145/209936.209945.

[84] *National Energy Research Scientific Computing Center (NERSC) Cori Supercomputer User Guide*. https://docs.nersc.gov/systems/cori. Accessed March 29, 2022.

[85] *NERSC Meraculous Benchmark*. http://www.nersc.gov/research-and-development/apex/apex-benchmarks/meraculous/. Accessed March 2, 2018.

[86] 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.

[87] *Oak Ridge Leadership Computing Facility (OLCF) Summit Supercomputer User Guide*. https://docs.olcf.ornl.gov/systems/summit_user_guide.html. Accessed March 29, 2022.

[88] Md Mostofa Ali Patwary et al. "Parallel efficient sparse matrix-matrix multiplication on multicore platforms". In: *ISC*. Springer. 2015, pp. 48–57.

[89] Domagoj Margan Per Fuchs and Jana Giceva. *Sortledton: a universal, transactional graph data structure*. https://ldbcouncil.org/event/fourteenth-tuc-meeting/attachments/per-fuchs-sortledton.pdf. 2022.

[90] Sreeram Potluri, Nathan Luehr, and Nikolay Sakharnykh. "Simplifying Multi-GPU Communication with NVSHMEM". In: *GPU Technology Conference*. NVIDIA. 2016.

[91] Sreeram Potluri et al. "Efficient inter-node MPI communication using GPUDirect RDMA for InfiniBand clusters with NVIDIA GPUs". In: *2013 42nd International Conference on Parallel Processing*. IEEE. 2013, pp. 80–89.

[92] Jack Poulson et al. "Elemental: A new framework for distributed memory dense matrix computations". In: *ACM Transactions on Mathematical Software (TOMS)* 39.2 (2013), pp. 1–24.

[93] 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.

[94] Felix Putze, Peter Sanders, and Johannes Singler. "Cache-, hash-, and space-efficient Bloom filters". In: *Journal of Experimental Algorithmics (JEA)* 14 (2009), p. 4.

[95] Matthew Rocklin. "Dask: Parallel computation with blocked algorithms and task scheduling". In: *Proceedings of the 14th python in science conference*. Vol. 130. Citeseer. 2015, p. 136.

[96] Erik Saule, Kamer Kaya, and Ümit V Çatalyürek. "Performance evaluation of sparse matrix multiplication kernels on Intel Xeon Phi". In: *PPAM*. Springer. 2013, pp. 559–570.

[97] Gerald Schubert et al. "Hybrid-parallel sparse matrix-vector multiplication with explicit communication overlap on current multicore-based systems". In: *Parallel Processing Letters* 21.03 (2011), pp. 339–358.

[98] Kayla Seager et al. "Design and Implementation of OpenSHMEM using OFI on the Aries Interconnect". In: *Workshop on OpenSHMEM and Related Technologies*. Springer. 2016, pp. 97–113.

[99] George M Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. "Order or shuffle: Empirically evaluating vertex order impact on parallel graph computations". In: *IPDPSW*. IEEE. 2017, pp. 588–597.

[100] Marc Snir. "Universal parallel computing research center at Illinois". In: *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE. 2009, pp. 1–36.

[101] Edgar Solomonik and Torsten Hoefler. "Sparse tensor algebra as a parallel programming model". In: *arXiv preprint arXiv:1512.00066* (2015).

[102] Edgar Solomonik et al. "Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions". In: *IPDPS*. IEEE. 2013, pp. 813–824.

[103] "STAPL Beta Release Tutorial Guide". In: (2017).

[104] Ion Stoica. *A Berkeley View of Big Data*. 2011.

[105] Gabriel Tanase et al. "The STAPL Parallel Container Framework". In: *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*. PPoPP '11. San Antonio, TX, USA: ACM, 2011, pp. 235–246. ISBN: 978-1-4503-0119-0. DOI: 10.1145/1941553.1941586. URL: http://doi.acm.org/10.1145/1941553.1941586.

[106] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. "Optimization of collective communication operations in MPICH". In: *The International Journal of High Performance Computing Applications* 19.1 (2005), pp. 49–66.

[107] Alok Tripathy, Katherine Yelick, and Aydın Buluç. "Reducing communication in graph neural network training". In: *SC'20*. 2020, pp. 1–17.

[108] Robert A Van De Geijn and Jerrell Watts. "SUMMA: Scalable universal matrix multiplication algorithm". In: *Concurrency: Practice and Experience* 9.4 (1997), pp. 255–274.

[109] M Mitchell Waldrop. "More than moore". In: *Nature* 530.7589 (2016), pp. 144–148.

[110] David W Walker and Jack J Dongarra. "MPI: a standard message passing interface". In: *Supercomputer* 12 (1996), pp. 56–68.

[111] Michele Weiland. “Chapel, Fortress and X10: novel languages for HPC”. In: *EPCC, The University of Edinburgh, Tech. Rep. HPCxTR0706* (2007).

[112] J. Willcock et al. “Active Pebbles: Parallel Programming for Data-Driven Applications”. In: *Proceedings of the 2011 ACM International Conference on Supercomputing (ICS’11)*. Tucson, AZ: ACM, June 2011, pp. 235–245. ISBN: 978-1-4503-0102-2.

[113] J. Willcock et al. “AM++: A Generalized Active Message Framework”. In: *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. Vienna, Austria: ACM, Sept. 2010, pp. 401–410. ISBN: 978-1-4503-0178-7.

[114] Jiakun Yan. “An RPC Communication System Based on Node-Level Aggregation”. Bachelor’s Thesis. Shanghai Jiao Tong University, 2019.

[115] Carl Yang, Aydın Buluç, and John D Owens. “Design principles for sparse matrix multiplication on the GPU”. In: *EuroPar*. Springer. 2018, pp. 672–687.

[116] Katherine Yelick et al. “The parallelism motifs of genomic data analysis”. In: *Philosophical Transactions of the Royal Society A* 378.2166 (2020), p. 20190394.

[117] Katherine A Yelick. “Programming models for irregular applications”. In: *ACM SIGPLAN Notices* 28.1 (1993), pp. 28–31.

[118] Kathy Yelick et al. “Titanium: A high-performance Java dialect”. In: *Concurrency Practice and Experience* 10.11-13 (1998), pp. 825–836.

[119] Yang You et al. “ImageNet Training in Minutes”. In: *Proceedings of the 47th International Conference on Parallel Processing*. ICPP 2018. Eugene, OR, USA: Association for Computing Machinery, 2018. ISBN: 9781450365109. DOI: 10.1145/3225058.3225069. URL: https://doi.org/10.1145/3225058.3225069.

[120] Weiqun Zhang et al. “AMReX: A framework for block-structured adaptive mesh refinement”. In: *Journal of Open Source Software* 4.37 (May 2019), p. 1370. DOI: 10.21105/joss.01370. URL: https://doi.org/10.21105/joss.01370.

[121] Yili Zheng et al. “UPC++: a PGAS extension for C++”. In: *28th IEEE International Parlalel and Distributed Processing Symposium*. IEEE. 2014, pp. 1105–1114.