High-performance parallel systems for data-intensive computing

Brandon Myers

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2016

Reading Committee:

Mark Oskin, Chair

Bill Howe

Luis Ceze

Program Authorized to Offer Degree:
Computer Science & Engineering

University of Washington

Abstract

High-performance parallel systems for data-intensive computing

Brandon Myers

Chair of the Supervisory Committee:
Associate Professor Mark Oskin
Computer Science & Engineering

Applications in data science rely on two computing paradigms: tuned high performance parallel programs and data analytics. While historically their differences were good reason to separate the paradigms into different systems, recent changes in hardware and, as a result, fast data processing techniques, call this separation into question. The goal of this dissertation is to present systems and experiments that combine high performance parallel programs and data analytics for performance while preserving programmability.

First, I present Grappa, a distributed parallel programming language implementation designed for building high performance data-intensive systems with less effort. Grappa provides a simple fine-grained programming model, while using the parallelism inherent in data-intensive applications to execute the program efficiently. Using Grappa we built native applications, and domain-specific frameworks for dataflow processing, graph processing, and relational query processing that are faster than their domain-specific counterparts.

Then, I present a survey on techniques for fast in-memory query evaluation. Using existing literature, I classified overheads in the conventional techniques for query evaluation and techniques that address them. In particular, I focus on *query compilation*, which specializes the query processor to the particular query. These ideas inspire the design of the second system.

Finally, I present Radish, a query processing engine built upon Grappa. Radish uses

efficient distributed data structures, avoids extra messages, and uses GRAPPA's runtime to execute fine-grained, tuple-by-tuple evaluation efficiently. I also developed a new query compilation technique that generates parallel code for entire processing pipelines. This compilation technique increased performance by 2.4× compared to generating fragments of code. RADISH is also competitive with other distributed parallel data processing systems.

In this dissertation, I provide supporting evidence for my thesis statement: *When applied to data-intensive applications, high-performance parallel systems and new database query evaluation techniques support improved performance, programmer productivity, and closer interaction between handwritten parallel programs and declarative queries.*

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

Mark Oskin for all the life and career advice, off-the-wall discussions mostly at Big Time, introducing me to fresh oysters, and usually telling me what I need to hear...

Bill Howe for an eScience perspective, pushing relational algebra, consistently being late to his next meeting to hammer out details with me, and taking me on as an advisee when I knew nothing about databases...

Luis Ceze for unceasing encouragement, keeping the ideas flowing, and letting his taste in food and drink rub off on me...

Brandon Holt and Jacob Nelson for countless hours and all-nighters together to work on Grappa and solving parallel computing...

Dan Grossman for so much good feedback on talks and helping me find a job...Hal Perkins for teaching advice and helping me with my first course...

Simon Kahan for wisdom on parallel computers, algorithms, and math...Preston Briggs for wisdom on parallel computers, algorithms, and compilers...

Senior members of Sampa that mentored me, including Ben Wood, Hadi Esmaeilzadeh, Tom Bergan, Brandon Lucia, Adrian Sampson, and Joe Devietti...

Dan Halperin and Andrew Whitaker for mentorship and for lots of contributions to Raco I depended on...Jeremy Hyrkas for starting the Radish project with me...

Byung-Gon Chun for mentorship, encouragement, and collaboration...

Chris Fensch for career advice, cat trips, making sure I get in enough hillwalking, and arguments about crampons, Svet Kolev for always making sure we'll have a story to tell later, Ricardo Martin for ensuring our endurance sports trips are 50% longer than initially planned, Aditya Sankar for undertaking the sport of trad climbing with me, and Daniel Perelman for

game nights and countless discussions about programming and Seattle bike routes...

Grad school companions through the years, including Seungyeop Han, Vikash Kumar, Paris Koutris, Mark Yatsgar, Thierry Moreau, Nate Vicar, Nick Hunt, Dylan Hutchison, Shrainik Jain, Seungki Kwak, Stanley Wang, and Josh Kao...

John Wawrzynek for initially taking a chance on me as an undergraduate teaching assistant and putting me on the path to a teaching career...

Everyone in the Sampa and UWDB labs...

The HCI lab for unofficially adopting me...

Others I've had the good fortune to work with, including Kivanc Muslu, Werner Dietl, Brian Cho, Seung-Hee Bae, Soumya Vasisht, Harley Montgomery, York Wei, Iris Wang, Zach Simon, Tae-Geon Um, Youngbin Bok, and Gyewon Lee...

Mentors that I've learned so much from, including Brad Chamberlain, Rusty Sears, Susan Eggers, Carl Ebeling, Zach Tatlock, Mike Ernst, Scott Hauck, Helene Martin, Magda Balazinska, Dan Suciu, and Alvin Cheung

My hosts at Pacific Northwest National Laboratory, including John Feo, David Callahan, and David Haglin...I used supercomputing resources at PNNL for much of this research...

...the many others that I have absentmindedly forgotten to list here...

DEDICATION

to my soulmate, Kyle

and my family David & Sharon, Bryan, Peter, and Rachael

Chapter 1
INTRODUCTION

Data analysis is increasingly central to many industries and sciences. Extracting insights from this data demands large-scale systems that can process data in parallel. This demand has been met by a proliferation of systems with capabilities different than those of conventional databases: a greater focus on complex analyses and data processing as opposed to data management. Two trends motivated me to design a new kind of system—one that combines high performance and data-intensive computing. These trends are 1) the need of both tuned parallel programs and data analysis in applications and 2) the adaptation of data-intensive systems to modern computer architectures.

1.1 Motivation

In enterprise data management for business intelligence, data enters the system in the form of database transactions, but scientific computing presents many examples where the data comes from elsewhere. Astrophysicists seek to understand simulations of galaxy formation with clustering algorithms [87]. Snapshots of galaxy state in the form of particles are loaded into databases for analysis. The data touched by the simulation and the data analysis is the same, yet the computation is divided into two different systems. There are three potential drawbacks of this divided architecture: 1) the need to throw away raw data before it is analyzed for lack of storage space or bandwidth, 2) additional latency and complication of loading a database before a human can start looking at the data, 3) and lower resolution or frequency of analysis from needing to move the data between systems. Bringing the two systems into one has great potential to lower these costs [147].

Trends in hardware have changed the design parameters of data management systems.

Growth of cheap DRAM capacity means that more of the working set fits in faster storage [121, 81]. Although the bottleneck of disk-based data management systems was disk I/O, once much of the data fit in DRAM, the bottleneck of many workloads shifted to the CPU and cache hierarchy [8, 102]. This trend and other hardware developments like flash, vector units, superscalar out-of-order cores, multi-core, and deeper cache hierarchies, has caused vendors to significantly redesign their data management systems to increase performance [18, 94]. Meanwhile, systems with the capability for fault-tolerant processing grew from the need to process big data [74, 48]. By using the disk for spilling from memory and for fault tolerance, the performance of these systems was largely determined by I/O. However, since the time of those systems, there has been an arms race to meet the high performance of modern databases. With a focus on in-memory processing and a more flexible fault-tolerance strategy, Spark [160] allows for data to bypass the disk more often than its predecessors. Unlike in their I/O-bound predecessors, high performance remains a first-class concern in this new generation of big data processing systems, evidenced by adoption of the same CPU-efficient techniques from modern databases [157, 5].

The new design of data management systems is becoming closer to that of high-performance processing systems where the CPU and memory system are critical. And, applications demand both data analysis and specialized parallel programs. In this thesis, I challenge the separation between high-performance computing platforms and data management platforms when they are applied to data-intensive applications.

1.2 Thesis statement

When applied to data-intensive applications, high-performance parallel systems and new database query evaluation techniques support improved performance, programmer productivity, and closer interaction between handwritten parallel programs and declarative queries.

1.3 Research questions

To support my statement, I pursued the following research questions:

- **RQ1** How can we build a high-performance system for data-intensive computing that accommodates different data processing models alongside flexible parallel programming?

- **RQ2** What techniques can improve the performance of database-like query processing in a high-performance parallel system?

- **RQ3** How close can the performance of declarative programs implemented in high-performance systems come to handwritten programs?

1.4 Thesis contributions

In this thesis, I explore the research questions by building and evaluating a sequence of systems for data-intensive applications that combine ideas from high performance computing and efficient data processing.

I investigated **RQ1** with a software distributed shared memory system called Grappa. Chapter 2 explores how to make a shared memory programming model perform well on data-intensive applications for distributed machines. The programming model was designed to be general purpose enough to build applications from any domain while still providing abstractions that make distributed programming productive. Grappa made this programming model fast on real machines with careful data movement and lightweight multitasking. Using Grappa we built domain-specific frameworks for dataflow, graph processing, and relational query processing, as well as native applications, that were faster than application-specific systems. Much of Chapter 2 is based on work presented in the following publications: we demonstrated the feasibility of multithreading for latency tolerance [114] and built and evaluated the full Grappa system [113][1].

To answer **RQ2**, I explored relational query processing with a system called Radish, which generates code for parallel shared memory systems like Grappa. In Chapter 3, I

[1]This work is joint with students Jacob Nelson and Brandon Holt. Some of the work appears in Jacob Nelson's dissertation with a greater emphasis on analysis of modern commodity hardware [115].

present a survey of prior work on fast in-memory query processing, and in particular focus on a new and increasingly employed class of techniques called query compilation. In Chapter 4, I present the design of Radish and how to make it perform well. In particular, Radish featured a new technique called *Compiled parallel pipelines*, which generated holistic code for the query using parallel loops. This code was on average 2.4× faster than the same system using conventional techniques (iterators and compiled expressions) on TPC-H [144]—a popular database analytics benchmark—queries. I explored **RQ3** by conducting a case study using Radish (Section 4.7.6), as well as surveying prior studies (Section 3.1.5). Much of Chapter 4 is based on work presented in the following publications: we demonstrated the promise of Radish by manually writing programs for queries [109] and built and evaluated the full Radish system [108].

Query compilation, the particular approach to query evaluation around which much of this dissertation is focused, continues to gain adoption in data management systems. Because of this interest, I include a discussion of software engineering of query compilers in Chapter 5. I highlight further opportunities in bringing high performance parallel programs and data analysis together in Chapter 6, present related work in Chapter 7 that is not covered by the survey, and conclude in Chapter 8.

Chapter 2

A SYSTEM FOR HIGH PERFORMANCE AND DATA-INTENSIVE APPLICATIONS

In this chapter, I present work on Grappa, a distributed shared memory system designed for data-intensive applications. I address **RQ1**: *"How can we build a high performance system for data-intensive computing that accommodates different data processing models alongside flexible parallel programming?"* Data-intensive applications have properties, such including poor data locality, that make it challenging to achieve high performance, especially in a distributed shared memory model. Grappa exploits the abundant concurrency available in data-intensive applications for performance. I evaluated Grappa by implementing three frameworks for data-intensive applications; Grappa was 10× faster than Spark on iterative MapReduce and 1.33× faster than GraphLab on graph processing. The third framework is related to the later chapters. Grappa is high performance and integrates general-purpose parallel programming with a variety of data processing models.

2.1 Introduction

It is challenging to build data-intensive applications on distributed machines. Realizing good performance requires careful partitioning of both data and computation to avoid data movement. Further, when the partitions are imperfect, we have to optimize data movement, and there are a number of parallel programming trade-offs like replication and re-computation versus sharing and the granularity of tasks. Many frameworks for data analysis, such as Dryad [74], MapReduce [48], Spark [160], Stratosphere [9], and GraphLab [58], were designed to handle these concerns for the programmer, but they often have restricted data models allowing only coarse-grained transformations.

We built a system with a flexible distributed programming model for building data-intensive applications. Having productivity in mind, we picked a shared memory programming model. Historically, distributed shared memory (DSM) systems only performed well when the application had significant locality, limited sharing, and coarse-grained synchronization—unlike modern data-intensive applications. We were inspired by partitioned global address space (PGAS) languages, a variant on software DSMs that provides explicit partitioning of memory. This explicit partitioning allows the programmer or compiler to organize the distribution and movement of data. Unfortunately, data-intensive applications have properties that require great effort from the PGAS programmer in order to optimize performance. We illustrate these challenges and then present a new DSM system, Grappa, that is designed for data-intensive applications.

2.2 Challenges of using PGAS for data-intensive computing

There are a number of challenges in writing data-intensive applications in a shared memory programming model. We use a simple kernel to illustrate: *given a collection of particles, build a histogram of their brightness.* The Chapel code in Figure 2.1a represents the histogram as an array of buckets, each to store the count of particles in a range of width B. The `dmapped` code simply indicates that this array is distributed across the partitions (i.e., the nodes of the distributed system). Each bucket is an `atomic int` to ensure that updating its count is atomic.

Unfortunately, the loop has poor data locality. The code indexes into `histogram` with "random" data (the brightness), so every iteration makes a **random access** to a bucket, which might reside in any partition. This observation about communication leads to the remainder of the challenges.

Next, in order for the Chapel runtime to implement atomic add, it might use an array of locks, as shown in Figure 2.1b. Unfortunately, this approach causes multiple round trips: one to acquire the lock, one or two to update the count, and one to release the lock.

To reduce the number of round trips, we exploit the fact that we distributed `locks`

```
1  const B1 = {1..nparticles} dmapped (Block(boundingBox={1..nparticles}))
2  const B2 = {1..nbuckets} dmapped (Block(boundingBox={1..nbuckets}))
3  var: ps [B1] Particle;
4  /* ...fill particle array... */
5  var: histogram [B2] atomic int
6  forall p in ps {
7    var i = (p.brightness/B):int
8    histogram(i).add(1)
9  }
```

(a) Original code

```
1  var: histogram [B2] int;
2  var: locks [B2] Lock;
3  forall p in ps {
4    var i = (p.brightness/B):int;
5    locks(i).acquire();
6    histogram(i) += 1;
7    locks(i).release();
8  }
```

(b) Lock per bucket

```
1  var: histogram [B2] int;
2  var: locks [B2] Lock;
3  forall p in ps {
4    var i = (p.brightness/B):int;
5    on histogram(i) {
6    // equivalent to 'on locks(i)'
7      locks(i).acquire();
8      histogram(i) += 1;
9      locks(i).release();
10   }
11 }
```

(c) Update in one round trip

Figure 2.1: Histogram program written in Chapel, a PGAS language

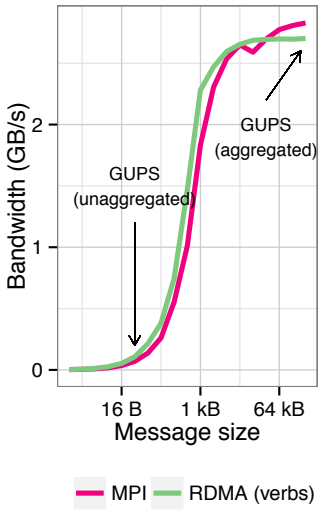


Figure 2.2: Ping pong bandwidth measured between two nodes

and `histogram` identically (i.e., they both use the index map `B2` so that the i^{th} elements are collocated). We use `on` to move execution to the partition where index `i` resides (Figure 2.1c). By moving the entire update code to the data, we reduce the number of round trips to one. Unfortunately, there is still a performance problem in most runtime systems. Each iteration sends a **small message** for the `on` (one function pointer and one integer, `i`). Current network interfaces utilize a small fraction of wire bandwidth when messages are small, even with the low overhead of kernel bypass and RDMA. Commodity network interfaces rely on large packets to achieve most of the available bandwidth of the network [21]. To illustrate this property, we measured the ping pong bandwidth between two nodes, varying the message size (Figure 2.2). The bandwidth was significantly higher with 8-kB messages than for small messages, like those used for individual histogram updates.

The observation that large messages are necessary to utilize bandwidth is well-known. The general approach to solving this problem is to coalesce individual data into large messages [34]. Writing communication code that coalesces messages effectively is difficult, especially for applications where random accesses are difficult to avoid. The language implementation should take some of this burden when possible.

To summarize, the challenges of implementing data-intensive applications on a DSM are small messages and poor locality. Despite these challenges, data-intensive applications have properties that can be exploited to make DSMs efficient. First, their abundant data parallelism provides opportunities for utilizing compute resources. Second, their measure of performance depends not on the latency of execution of any specific parallel task, as it would in transactional workload like a web server, but rather on the aggregate execution time of all tasks, that is, on throughput. We exploited these to implement an efficient DSM. Figure 2.6 shows a similar histogram program written in Grappa, embedded in C++. The input array of `Particle`s and the array of `Cell`s comprising the hash table are both distributed across nodes. The features of the runtime system make this program run efficiently.

Grappa is inspired by the Tera MTA [13, 12], a supercomputer with custom hardware. Instead of relying on locality to reduce the cost of memory accesses, the MTA depended on parallelism to keep the resources of the processor busy and hide the high cost of inter-node communication. To support fine-grained messaging like the MTA on commodity hardware, Grappa batches small messages together into larger physical network packets, thereby maximizing the available bisection bandwidth of commodity networks. To support the large amount of concurrency required for tolerating the latency of batched memory accesses over the network, Grappa employs lightweight multithreading, also in software.

2.3 Tolerating memory latency with multithreading

Before describing the distributed Grappa system, we demonstrate the feasibility of tolerating memory access latency using software multithreading. Specifically, we measured the maximum random reference rate of a multi-core processor, and then tested whether Grappa's software multithreading could achieve the same throughput.

We ran pointer-chasing benchmarks on a single multi-core node. These benchmarks are intended to model the worst-case behavior of data-intensive applications, where each memory reference causes a cache miss. We ran the experiments on a Dell PowerEdge R410 with two Xeon X5650 (Nehalem microarchitecture) chips and 24GB of DRAM, with hyper-threading

disabled. We only use one of the chips, which has its own integrated memory controller responsible for DIMMs holding half the DRAM.

First, we measured the maximum random reference rate that the processor's cores can issue by running pointer chasing code following the model of Figure 2.3a. Each core issues n list traversals in a loop; we call n * *number of cores* the number of concurrent references *offered*, since the memory system may not be able to satisfy them all in parallel. We rely on the cores' exploitation of instruction level parallelism (ILP) for memory concurrency.

Figure 2.4a shows the result. Each point represents the maximum rate pointers are traversed for a given number of concurrent offered references. We see a maximum rate of 277 million references per second, which agrees with measurements from another study [100]. This rate is achieved when the number of offered references is 36.

Next, we investigated whether Grappa's software multithreading could reach the maximum random reference rate of the processor, despite the overhead of switching between threads. We modified our pointer chasing benchmark as shown in Figure 2.3b. For this experiment, the only source of memory concurrency was the use of threads.

Figure 2.4b shows that we were to obtain a rate of 275 million references per second with 48 concurrent misses, or 8 threads per core. More concurrent requests were required to saturate memory bandwidth than in the ILP-only experiment. A drop in performance occurred beyond a certain number of threads per core. This drop occurs because each thread issues a prefetch and the core can only track a maximum of 10 prefetches (size of the L1 cache line fill buffer) before it starts squashing earlier, incomplete prefetches to make room. This effect was an example of the general principle that having too much concurrency sometimes wastes resources.

The experiments in this section demonstrated that software multithreading is a feasible approach to tolerating local memory access latency. Multithreading is an important component in the full Grappa system, which has to tolerate latencies of a distributed machine.

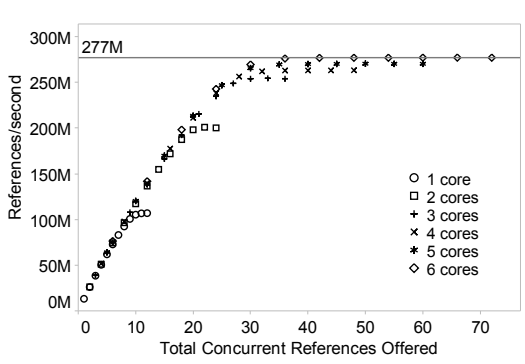
```
1  while (count-- > 0) {
2    list1 = list1->next;
3    list2 = list2->next;
4    ...
5    listn = listn->next;
6  }
```

(a)

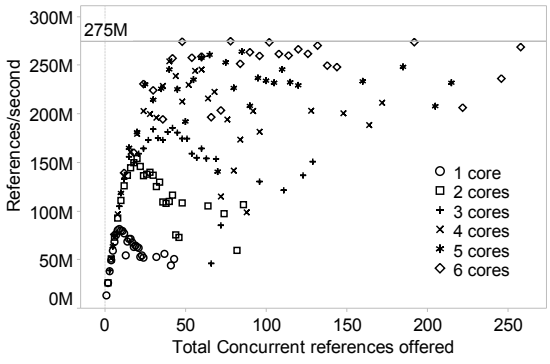
```
1  for (i=0; i < numthreads; i++) {
2    spawn thread {
3      while (count-- > 0) {
4        prefetch(&(lists[i]->next));
5        yield();
6        lists[i] = lists[i]->next;
7      }
8    }
9  }
```

(b)

Figure 2.3: Pseudocode for pointer chasing. a) using the ILP of multiple memory references for concurrency b) using multiple threads for concurrency



(a)



(b)

Figure 2.4: Pointer chasing throughput as number of outstanding memory references varies. a) Single thread per core and multiple outstanding references allowed. *Total concurrent references offered = number of lists per core * number of cores.* b) Multiple threads per core and one outstanding reference allowed per thread. *Number of threads per core = total concurrent references/number of cores.* We see that pointer chasing with threads can nearly saturate the bandwidth of the processor, although the processor is sensitive to the number of outstanding prefetches.

2.4 Latency-tolerant DSM

Figure 2.5 shows an overview of Grappa's DSM system. Before describing the Grappa system in detail, we describe its three main components:

Distributed shared memory The DSM provides fine-grained access to data anywhere in the aggregate memory of the nodes. Every piece of global memory is owned by a single core in the system. Access to data on remote nodes is made through *delegate* operations that run on the owning core. Delegate operations include normal memory operations such as *read* and *write* as well as operations useful for synchronization such as *fetch-and-add* [59]. Delegate operations help us offer a memory model that is similar to that of C/C++ [25, 75], so it is familiar to programmers.

Task system The task system supports lightweight multithreading and global dynamic load balancing — tasks can be stolen from any node in the system. Tasks are executed by cooperatively-scheduled user-level threads. Critical to the latency-tolerance of Grappa, threads that perform long-latency operations (i.e., remote memory accesses through delegates) automatically suspend while the operation is executing and wake up when the operation completes. Grappa supports 100's of concurrent tasks *per core*.

Communication layer The main goal of our communication layer is to batch[1] small messages into large ones. This process is not directly visible to the application programmer. Its interface is similar to active messages [51], messages that perform processing on the destination. Because batching and de-batching of messages needs to be efficient, we control data movement through the memory hierarchy and perform the process in parallel, carefully using lock-free synchronization operations. For portability, we use MPI [104] (the standard API for high performance message passing) for messaging and process setup and tear down.

[1] In this thesis, *batching* replaces the term *aggregation* from our original publication because I frequently refer to *aggregation* as reducing a column in relational query processing.

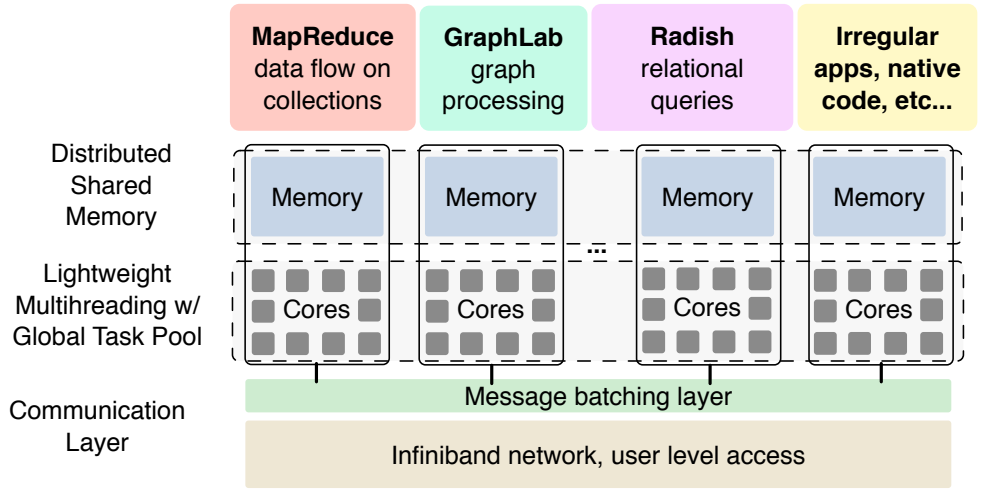


Figure 2.5: GRAPPA design overview. The distributed system is programmed using a shared memory and shared task pool. GRAPPA comprises an implementation of distributed shared memory, lightweight multithreading, and a communication layer that batches messages. We built various data intensive frameworks and applications in the GRAPPA language.

2.4.1 Distributed shared memory

We describe the implementation of GRAPPA's shared global address space and memory consistency model.

Addressing Applications written for GRAPPA may address memory in two ways: locally and globally. Local memory is local to a single core within a node in the system. Accesses occur through normal pointers. Applications use local accesses for a number of things in GRAPPA: the stack associated with a task, shortcutting accesses to global memory from the memory's home core, and accesses to debugging infrastructure local to each system node. In general, local pointers cannot access memory on other cores, and are valid only on their home core.

Global addresses point to data on any node and are created in three different ways. First, GRAPPA allows local data on a core's stacks or heap to be exported to the global address

```
1  struct Particle {
2    int id;
3    float brightness;
4  }
5
6  // distributed input array
7  GlobalAddress<Particle> prts
8      = load_input();
9
10 // distributed hash table
11 using Cell = std::map<float, int>;
12 GlobalAddress<Cell> cells
13     = global_alloc<Cell>(ncells);
14
15 forall(prts, nprts, [=](Particle& p) {
16   auto b = p.brightness;
17   // hash the brightness to determine
18   // the hash table cell
19   size_t idx = hash(b) % ncells;
20   delegate(&cells[idx], [=](Cell& cell)
21   {  // this block runs atomically
22     if (cell.count(b) == 0) cell[b] = 1;
23     else cell[b] += 1;
24   });
25 });
```

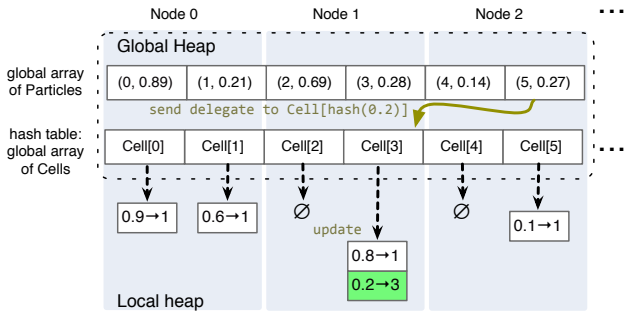


Figure 2.6: Histogram program using a simple hash table in Grappa's distributed shared memory

space so it is accessible to other cores across the system. The second and third ways to create a global address are from two types of global memory allocations: global array and symmetric. A symmetric allocation reserves space for non-coherent replica of an object on every core in the system. The behavior is identical to performing a local allocation on all cores, but the local addresses of all the allocations are guaranteed to be identical. Symmetric objects are often treated as a *proxy* to a global object, holding local copies of read-only data (e.g., number of vertices in a structure-immutable graph), or allowing operations on the global object to be transparently buffered [67]. Figure 2.7 shows how local, global array, and symmetric allocations can all be used together for the histogram data structure. Using symmetric allocation of a `Histogram` encapsulates the global array address and size of the hash table, and the local replicas are read without communication.

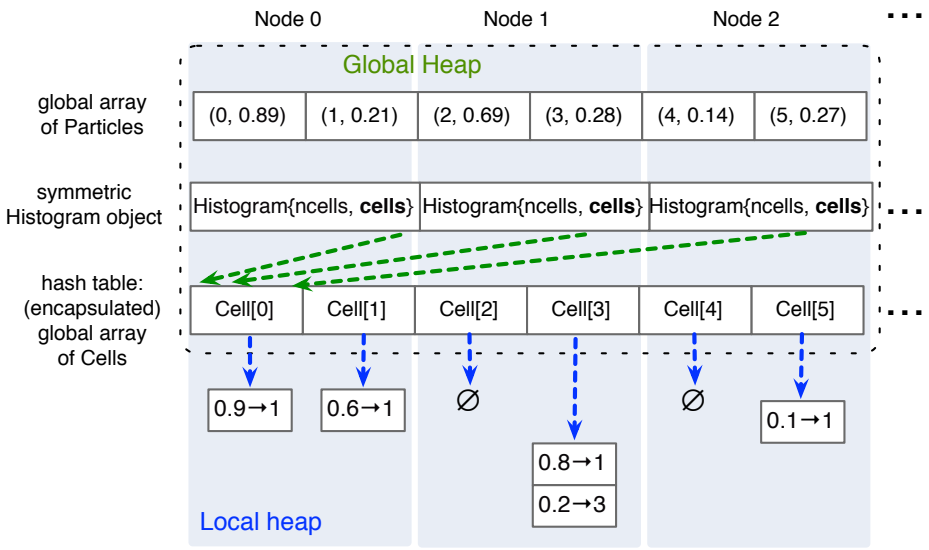


Figure 2.7: Using global addresses for the histogram program. The input `Particles` sit in a global array. The `Histogram`'s fields are locally accessible from any node, and its cells are stored in a global array. Operating on one cell of the histogram involves reading the local copy of `cells`, adding to that address the index of the target cell, and sending a single delegate operation to access the target cell.

Delegate operations When the data access pattern has low locality, it is sometimes more efficient to modify the data on its home core instead of bringing a copy to the requesting core and returning a modified version. We used this idea in the motivating code example to reduce the communication for one histogram update to a single round trip (Figure 2.1c). Access to Grappa's distributed shared memory is provided through *delegate* operations, short operations performed by the memory location's owning core. Delegate operations [114, 98] provide this capability. Besides *read/write* operations on global memory, delegates can also implement more complex *read-modify-write* and synchronizing operations (e.g., *fetch-and-add*, mutex acquire, queue insert).

Code in a delegate operation is guaranteed to execute atomically, provided that it does not

cause a context switch. Because Grappa uses cooperative multithreading, context switches only occur during specific library calls. To avoid context switches, a delegate must only touch memory owned by a single core and not call procedures that block the thread.

We use these restrictions to ensure that delegate operations at the same address from multiple requestors are serialized through a single core in the system, providing atomicity with strong isolation. A trade off of this serialization is a lack of parallel reads to a single location—parallel reads require that the programmer explicitly replicate their data. On the upside, writing operations on locations that are highly contended are faster because the cache lines do not bounce back and forth.

Memory consistency model Accessing global memory though delegate operations allows us to provide a familiar memory model. Delegate operations are solely responsible for synchronization. Since delegate operations execute on the home core of their operand in some serial order and only touch data owned by that single core, they are guaranteed to be globally linearizable [65], with their updates visible to all cores across the system in the same order. In addition, only one delegate can be in flight at a time from a particular task, that is, global memory operations from a particular task are not subject to reordering. Moreover, once an update from a delegate is visible to one core, it is also visible to all other cores. Consequently, all synchronization operations execute in program order and are made visible in the same order to all cores in the system. These properties are sufficient to provide sequential consistency for data-race-free programs [2], which underpins the memory models of C/C++ [25, 75]. The blocking property of delegates provides a clean model but is conservative: we discuss asynchronous operations, which the runtime can reorder, within the next section.

2.4.2 Task System

Grappa pins a single operating system thread to each hardware core; all code runs in these threads. The basic unit of execution in Grappa is a *task*. When a task is ready to execute,

it is mapped to a *worker* thread that is scheduled within an OS thread; hereafter, we refer to worker threads as "workers" to avoid confusion. Scheduling of tasks onto workers and workers onto the core's OS thread is carried out entirely in user-mode without intervention by the operating system. This section describes the components of the task system, the mechanisms for lightweight context switching that make software multithreading useful for latency tolerance, and how to express concurrency in Grappa.

Tasks Tasks are specified by a closure that holds both code to execute and initial state. The closure can be specified with a function pointer and explicit arguments, a C++ struct that overloads `operator()`, or a C++11 lambda. These objects, typically small ($\sim$ 32 bytes), hold read-only values such as an iteration index and pointers to common data or synchronization objects. A closures can be serialized and transported around the system and is eventually executed by a worker.

Workers Workers execute application and system (e.g., communication) tasks. A worker is simply a collection of status bits and a stack, allocated at a particular core. When a task is ready to execute, it is assigned to a worker that executes the task closure on its own stack. Once a task is assigned to a worker, it stays with that worker until it finishes.

Scheduling A worker yields control of its core whenever its task initiates a long-latency operation, allowing the processor to remain busy while waiting for the operation to complete. The programmer can also direct scheduling explicitly to implement custom synchronization mechanisms. To minimize context-switch overhead, the Grappa scheduler operates entirely in user-space and does little more than store the registers of one worker and load that of another.

Each core in a running Grappa program runs its own independent scheduler. The scheduler has a collection of active workers ready to execute called the *ready worker queue*. Each scheduler also has three queues of tasks waiting to be assigned a worker. The first two

run user tasks: a public queue of tasks that are not bound to a core yet, and a private queue of tasks already bound to the core where the data they touch is located. The third is a priority queue scheduled according to task-specific deadline constraints; this queue manages high priority system tasks, such as periodically servicing communication requests.

The separation of the system task and management tasks from the user tasks isolates compute resources similarly to the staged event-driven architecture [151] (our *tasks* are their *events*). Grappa's user tasks are bound to one pool of workers, while the system task has a dedicated worker. And since workers share one OS thread, Grappa schedules the system task's worker periodically to prevent starvation of the messaging system.

Context switching Grappa context switches between workers non-preemptively. As with other cooperative multithreading systems, we treat context switches as function calls by saving and restoring only the callee-saved state as specified in the x86-64 ABI [14] rather than the full register set required for a preemptive context switch. This context requires 62 bytes of storage.

Grappa's scheduler is designed to support a large number of workers—so large, in fact, that their combined context data does not fit in cache. In order to minimize unnecessary cache misses on context data, the scheduler explicitly manages the movement of context data into the cache. To accomplish this, the scheduler establishes a pipeline of ready worker references. This pipeline consists of *ready-unscheduled*, *ready-scheduled*, and *ready-resident* stages (Figure 2.8). When context prefetching is on, the scheduler is only ever allowed to run workers that are *ready-resident*; all other workers are assumed to be out-of-cache. The examined part of the *ready-unscheduled* queue itself must also be in cache. For a FIFO queue, the head of the queue will always be in cache due to its spatial locality. Other schedules are possible as long as the amount of data they need to examine to make a decision is independent of the total number of workers.

When a worker is signaled, its reference is marked *ready-unscheduled*. Every time the scheduler runs, one of its responsibilities is to pick a *ready-unscheduled* worker to transition

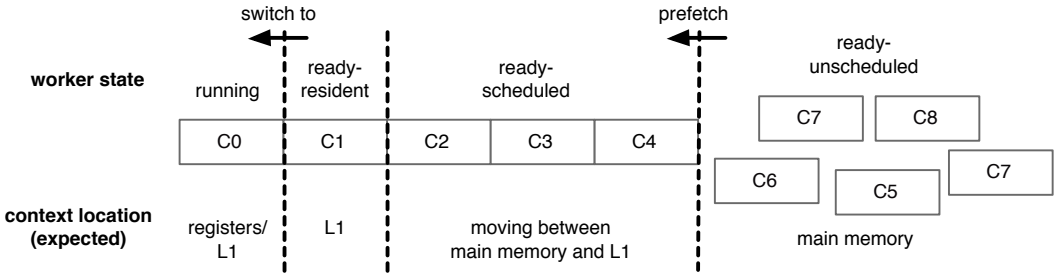


Figure 2.8: Movement of contexts during scheduling. Ready-unscheduled contexts wait in main memory. When the scheduler decides to run a context in the near future, it prefetches it towards the L1 cache.

to *ready-scheduled*; it issues a software prefetch to start moving the task toward L1. A worker needs its metadata (one cache line) and its private working set to begin executing without cache misses. Determining the exact working set may be difficult, but approximating the working set with the top 2-3 cache lines of the stack is an acceptable naïve heuristic. The worker data is *ready-resident* when it arrives in cache.

Prefetching contexts before scheduling them is an open-loop system, since notification of the arrival of a prefetched cache line is not available in x86-64. Therefore, we must determine how to configure the prefetch-to-schedule interval. If we make the assumption that tasks spend on average the same amount of time between starting and suspending, we can use a simple analytical model. Let U be the average user time spent by a task, let C be the average context-switch time when there are no CPU-stalling cache misses, and let M be the latency to read a context from main memory. The unknown W is the number of workers we need to run *between* prefetching and scheduling a target context to avoid missing. Then, to ensure there is always one worker in the *ready-unscheduled* queue to prefetch, the

following constraint must hold.

$$(U + C) * W \geq M$$
$$W \geq \frac{M}{U + C}$$

These W workers are required *in addition to* the workers required for GRAPPA's latency tolerance of user-level global memory accesses. Empirically, we found that given the U, C, and M determined by our processors and workloads, prefetching four workers ahead in the scheduling order is sufficient to prevent cache misses on context switches. As for an upper bound on W: it must not be set too high that it pollutes the L1 cache.

Fast context switching is critical to GRAPPA's ability to tolerate latency. We assess context switch overheads using a simple microbenchmark that runs a configurable number of workers on a single core, where each worker increments values in a large array to simulate work. Figure 2.9 shows the average context switch time as the number of workers grow. At the standard operating point for our cluster (≈1,000 workers), context switch time is on the order of 50 ns. As we add workers, the time increases slowly, but levels off; with 500,000 workers context switch time is around 75 ns. Without prefetching, context switching is limited by memory access latency—approximately 120 ns for 1,000 workers. Conversely, with prefetching on, context switching rate is limited by memory bandwidth—we determined this by calculating total data movement based on switch rate and cache lines per context switch in a microbenchmark. As a reference point, for the same context switching microbenchmark using kernel-level Pthreads on a single core, the switch time is 450ns for a few threads and 800ns for 1000–32000 threads.

Expressing parallelism The GRAPPA API supports the spawning of individual tasks with the option of binding them to a core or not. These tasks may be full-fledged workers with a stack and the ability to block, or they may be *asynchronous delegates*, which like delegate operations execute non-blocking regions of code atomically on a single core's memory.

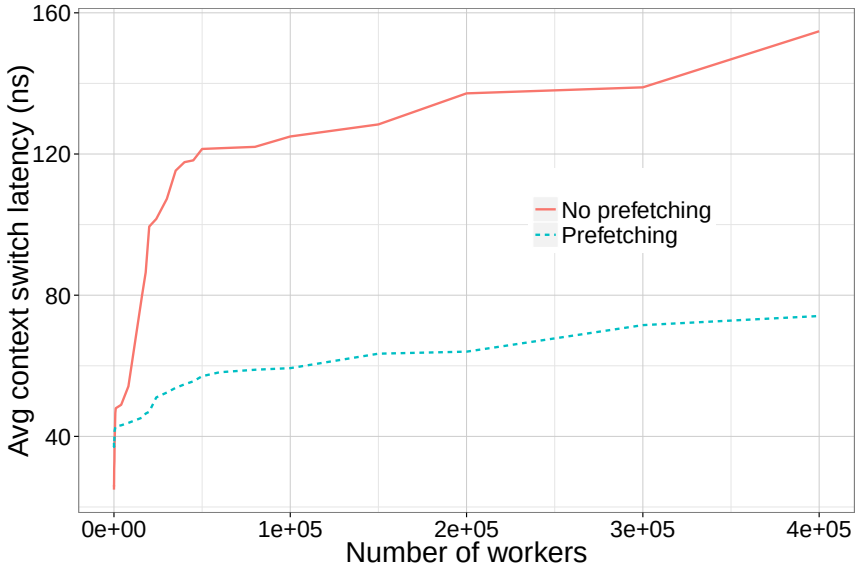


Figure 2.9: Average context switch time with and without prefetching

Asynchronous delegates are treated as task spawns in the memory model.

For better programmability, tasks are automatically generated from parallel loop constructs, as in `forall` in Figure 2.6. When executing a parallel loop, Grappa needs to provide a sufficient number of concurrent tasks to keep the machine busy, while maintaining a practical upper bound (usually linear in the number of cores). One technique for bounding concurrency for nested parallelism is *recursive decomposition*, illustrated in Figure 2.10. Recursive decomposition executes a loop by splitting the iterations into two disjoint subsets: it spawns the right subset as a new task and proceeds recursively with the left [24, 19]. To reduce the scheduling overhead of a task-per-iteration, a user-configurable threshold determines the minimum number of iterations assigned to the leaf tasks.

Grappa loops can iterate over an index space or over a segment of global memory. These loops give the option to spawn the tasks either so that they are bound to a specific core or

with no locality constraint. In the former case, each tasks is bound to a specific core—where the memory it needs to access is located. In the former case, tasks may be spawned anywhere and be stolen by any core in the system.

Loop termination GRAPPA's loops execute iterations across all partitions, so they rely upon distributed termination detection. If for all iterations i, i is guaranteed to execute on one partition, then coarse-grained completion detection is sufficient. Each core joins its own tasks and then enters a global barrier.

However, if there exists an iteration j, such that j *may* execute on multiple partitions (e.g., j is spawned with no locality constraints or j spawns tasks on other partitions), then per-task distributed termination detection is required to prevent races causing early termination. Per-task distributed termination detection is very costly when there are many tasks.

Fortunately, many kernels can be written with property called *async-finish* [88] that makes distributed termination detection less costly. *Async-finish* parallelism means that all tasks spawned by nested parallelism (the `async`) need only be joined at the root task (the `finish`). Having an all-to-one dependence relationship means tasks can be joined in bulk, which greatly reduces the number of synchronization messages. GRAPPA has an efficient bulk synchronization object, called `GlobalCompletionEvent`, that tracks all tasks spawned by `forall`s and asynchronous delegates associated with it.

2.4.3 Communication support

GRAPPA's communication layer has two components: a user-level messaging interface based on active messages, and a network-level transport layer that supports request batching for better communication bandwidth.

Active message interface At the upper (user-level) layer, GRAPPA implements asynchronous active messages [51]. Like tasks, GRAPPA's active messages are simply a C++11 lambda or other closure. We take advantage of the fact that our homogeneous cluster hard-

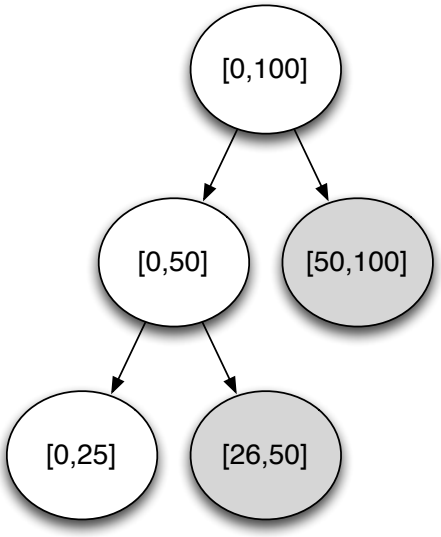


Figure 2.10: The *recursive decomposition* execution strategy for parallel loops. Labels indicate the iteration range covered by the node, white nodes are currently active tasks, gray nodes are enqueued, and arrows indicate the "spawns" relationship. By waiting to expand a gray node until white nodes to the left are completed, the space used for concurrency (i.e., the number of tasks) is proportional to the parallelism (i.e., the number of workers).

ware runs the same binary in every process: each message consists of a template-generated deserializer pointer, a byte-for-byte copy of the closure, and an optional data payload.

Message batching Batching and sending messages efficiently in Grappa is important for supporting frequent small messages. To achieve that efficiency, Grappa makes careful use of caches, prefetching, and lock-free synchronization operations.

Figure 2.11a shows the batching process. Cores keep their own outgoing message lists, with as many entries as the total number of system cores running in the Grappa program. These lists are accessible to all cores in a Grappa node so they can peek at each other's message lists. When a task sends a message, it allocates a buffer from a pool, determines the destination system node, writes the message contents into the buffer, and links the buffer into the corresponding outgoing list. These buffers are touched only twice for each message

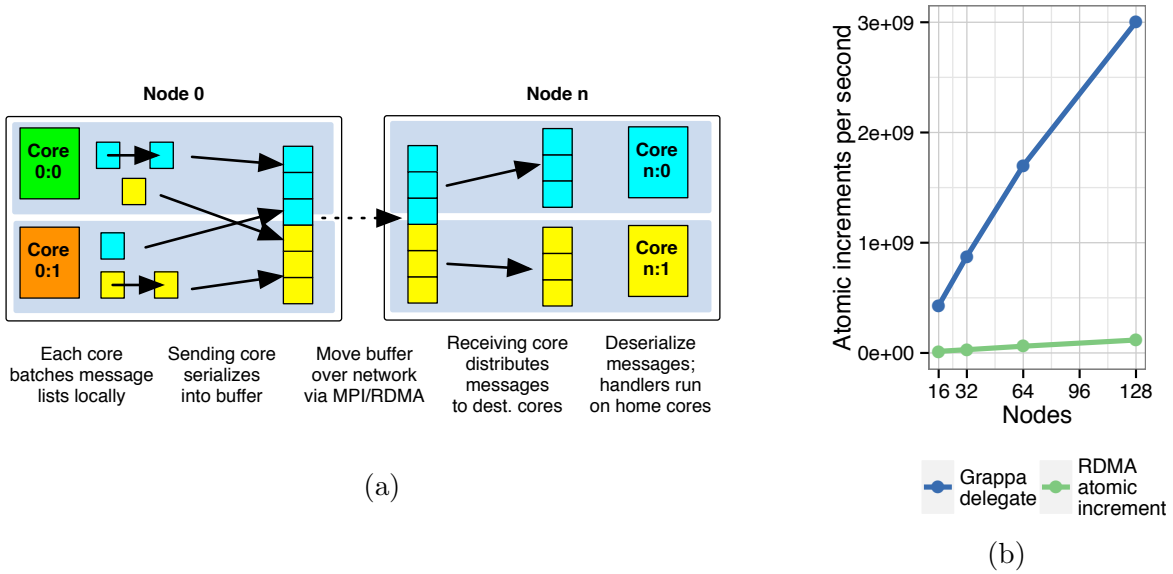


Figure 2.11: Message batching in GRAPPA. a) high-level mechanism where each core of a node is responsible for batching messages from local cores to a disjoint subset of remote cores, b) Throughput on the GUPS benchmark: random updates to a billion-integer distributed array. Batched GRAPPA delegates, implemented in software, attain a random reference rate much higher than the native increment operation of our network cards.

sent: once when the message is created, and once (much later) when the message is serialized for transmission. Therefore, the pool allocator prefetches the buffers with the non-temporal flag [41] to minimize cache pollution.

Within a node, each processing core is responsible for batching and sending the resulting messages from all cores on that node to its share of destination nodes; for example, in the diagram, core *0:0* is responsible for sending to core *n:0*. Cores periodically execute a system task that examines the outgoing message lists for each destination node for which the core is responsible. The system task flushes a message list once it is long enough or a message has waited past a timeout period. To flush, all messages to a given destination node from that source node are sent by copying them to a buffer visible to the network card. Actual message transmission is done purely in user-mode using MPI, which in turn uses RDMA.

The final message assembly process involves manipulating several shared message lists, so it uses compare-and-swap (`CAS`) operations to avoid high synchronization costs. Traversing the message lists requires careful prefetching because most of the outbound messages are no longer in the processor cache. Within a node, we use a per-core array of message lists that is only periodically shared with other cores. We experimentally determined that this approach is faster than a single shared array of message lists.

Once the remote system node has received the message buffer, a management task is spawned to manage the unpacking process. The management task spawns a task on each core at the receiving system to unpack messages destined for that core. Once all cores have processed the message buffer, the management task sends a reply to the sending system node indicating the successful delivery of the messages.

Batching versus native RDMA Given the increasing availability and decreasing cost of low-latency RDMA-enabled network hardware, it would seem logical to use this hardware to implement Grappa's DSM. Figure 2.11b shows the performance difference between native RDMA atomic increments and Grappa atomic increments using the cluster-wide random access benchmark, GUPS, on the cluster described in Section 2.5. The RDMA setup of the experiment used the network card's native atomic fetch-and-increment operation, and issued increments to the card in batches of 512. The Grappa setup issued delegate increments in a `forall`. Both setups performed increments to random locations in a 32 GB array of 64-bit integers distributed across the cluster. Figure 2.11b shows how batching allows Grappa to exceed the performance of the card by 25× at 128 nodes. We measured the effective bisection bandwidth of the cluster [66]. For GUPS, performance is limited by memory bandwidth during batching and uses $\sim 40\%$ of available bisection bandwidth.

Figure 2.2 illustrates why using RDMA directly is not sufficient: the maximum bandwidth using small messages is small fraction of the bandwidth attained by large messages. Our cluster's cards are unable to push small messages at line rate into the network; we measured the peak RDMA performance of our cluster's cards to be 3.2 million 8-byte writes per second,

while the wire-rate limit is over 76 million [73]. I refer the reader to Nelson's thesis [115] for an in-depth survey of message overheads in RDMA network interfaces. Even if native small message performance improves in future hardware, our batching support will still be useful to minimize cache line movement, PCI Express round trips, and other memory hierarchy limitations.

2.5 Evaluation

We implemented Grappa in C++ for the Linux operating system. The core runtime system system is 17K lines of code. We ran experiments on a cluster of AMD Interlagos processors with 128 nodes. Nodes have 32 cores operating at 2.1GHz, spread across two sockets, 64GB of memory, and 40Gb Mellanox ConnectX-2 InfiniBand network cards. Nodes are connected via a QLogic InfiniBand switch with no oversubscription. We used a stock OS kernel and device drivers. The experiments were run in a machine without administrator access or special privileges. GraphLab and Spark communicated using IP-over-InfiniBand (IPoIB) in Connected mode. First, we explore the role of concurrency in Grappa, and then we compare Grappa's performance on data-intensive applications to that of other domain-specific data processing frameworks.

2.5.1 Concurrency and performance

We study how concurrency affects throughput and latency in Grappa. We ran experiments on a modified version of the Unbalanced Tree Search (UTS) microbenchmark [120], which we call UTS-Mem. UTS was designed to test the dynamic load balancing capability of a system by spawning tasks in the shape of unbalanced trees according to a variety of probability distributions. UTS-Mem recycles the generation processes but tests irregular memory access performance. UTS-Mem materializes the entire tree in global memory in a generation step and then times the search.

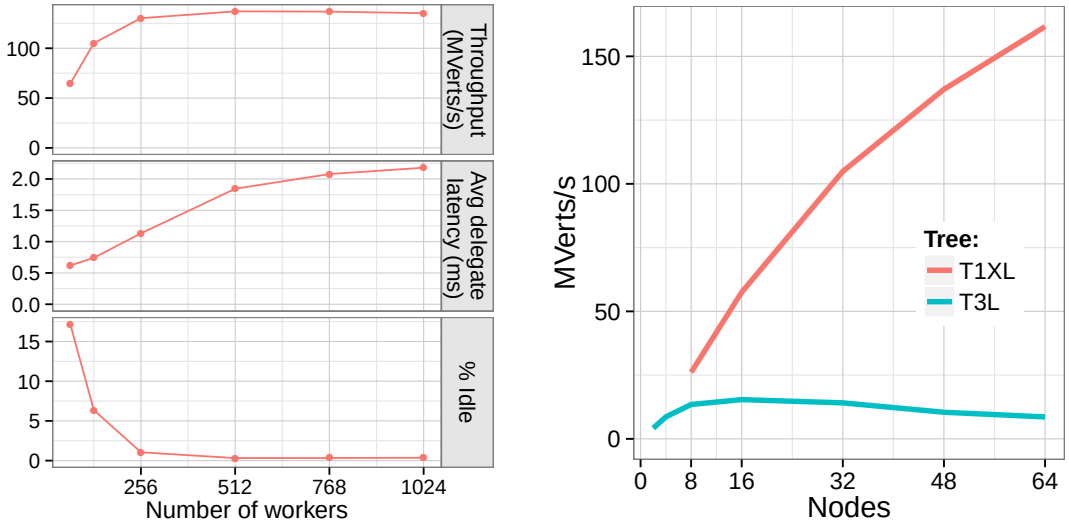
Effect of concurrency GRAPPA depends on concurrency to cover the latency of batching and remote communication. How much is required for good performance? How does round-trip latency change? We ran UTS-Mem on 48 nodes and 16 cores per node, varying the number of workers per core. In Figure 2.12a, the top pane shows the overall throughput of the tree search. The middle pane shows average blocking delegate operation latency in microseconds. The bottom pane shows idle time; that is, the fraction of the time the scheduler could not find a ready user task to run and the system task had no messages to process.

We observed three things from this figure. First, above 512 workers per core, idle time was practically zero; workers tasks generated requests fast enough to cover the latency of batching and communication. Second, the idleness was related to throughput: throughput peaked at 512 workers and gradually decreased after that due to the overhead of unnecessary context switches and larger working set. Finally, we saw that with 512 workers, the average per-request latency was 1.8ms. Beyond the point of peak throughput, request latency continued to grow; on an application where messages have *some* sensitivity to latency, this growing delay could hurt performance more than in this example.

Parallel slack Not all data-intensive applications have solely latency-tolerant memory accesses. GRAPPA's delay of individual memory accesses for high throughput relies on *parallel slack* in the application. A program has parallel slack when it presents many more concurrent tasks than there are available physical processors [146]. When the critical path of the program is long relative to the total work, delaying each message on the critical path by GRAPPA's batching time will have a detrimental effect on performance.

We looked at two classes of unbalanced trees with different shapes: T1 trees are shallow and wide, while T3 trees are thin and deep. Our hypothesis was that T1 would provide more concurrency and thus higher throughput.

Figure 2.12b shows strong scaling results. We observed that throughput on the T1 tree improved through 64 nodes because there was plenty of concurrency to keep the machine busy.



(a) (b)

Figure 2.12: Study of concurrency in Grappa using the UTS-Mem microbenchmark. a) runtime metrics for varying size of worker pool b) strong scaling of throughput for wide and narrow trees. Grappa depends on concurrency for throughput; narrow trees present insufficient concurrency to saturate the throughput of the machine.

In addition, the throughput on the T3 tree got worse after 16 nodes; this decline happens because after the machine consumes the scarce concurrency, scaling up only distributes the data more. Even though the trees are similar sizes, T1's width provides abundant concurrency. In fact, even at the 16-node data points, the average number of active tasks per core is 775 and 13 for T1 and T3, respectively; this idleness for T3 accounts for the 2× or higher difference in throughput.

The depth of T3 makes it a proxy for graphs of high diameter (e.g., road networks) and the shallowness of T1 makes it a proxy for graphs of low diameter (e.g., social networks). However, many of the edges in T3 or a high-diameter graph are *not* on the critical path. Grappa treats all messages equally, but by telling Grappa to use a low-latency "fast path" for a few messages related to the traversal of critical path edges, we might reduce the total running time. These high-priority messages may not always be easy to predict, but some

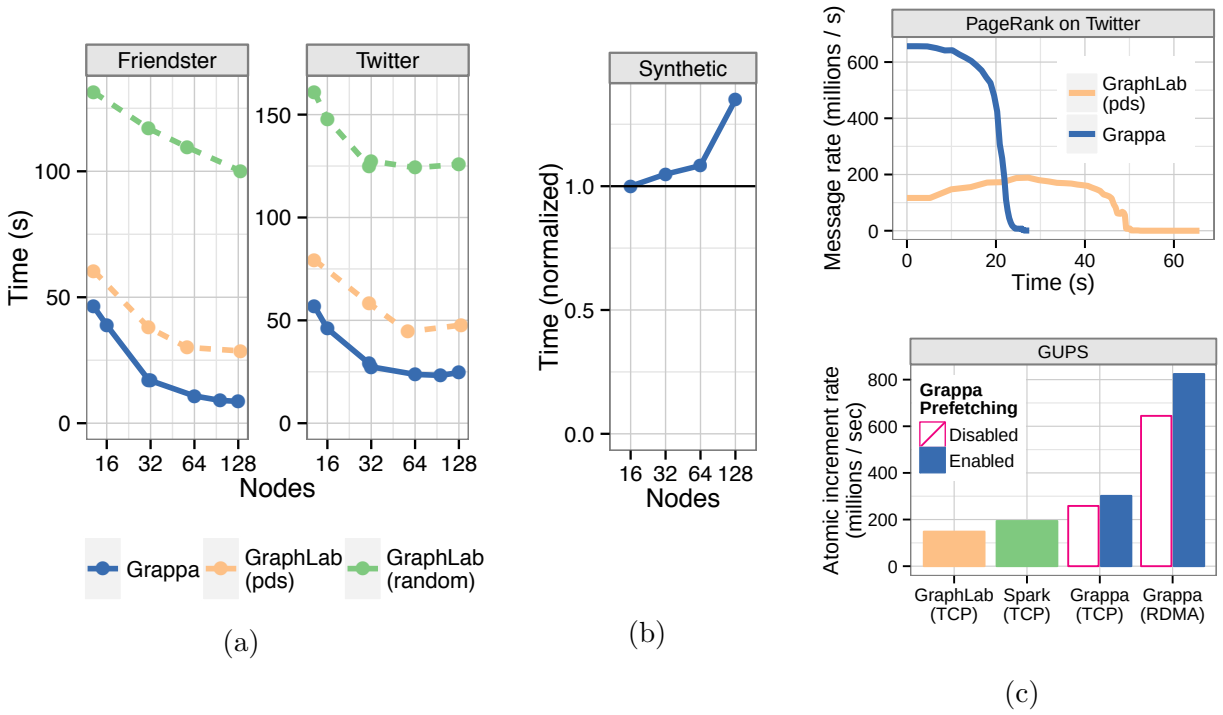


Figure 2.13: Performance results on PageRank for GRAPPA's Vertex-centric framework and GraphLab. a) Strong scaling (limited by dataset size) of time to convergence, same number of iterations. GRAPPA is faster even when using the inferior *random* graph partitioning that incurs more communication. b) Weak scaling on synthetic power-law graphs, showing scaling deteriorated less than 30% at 128 nodes. c) Message rates on 31 nodes. GRAPPA's higher message rate is due to its superior (despite being the most generic) message batching, as well as using RDMA natively.

algorithms, like single-source shortest path, admit heuristics for assigning priority to tasks. The graph processing system Galois allows priority scheduling of tasks so those that may be on the critical path can proceed [117]. Finally, we recognize that T3 is truly a worst case application; in some applications with a long critical path, the references along this path might have locality.

2.5.2 Vertex-centric Programs on GRAPPA

We implemented a vertex-centric programming framework in GRAPPA with most of the same core functionality as GraphLab [95, 58] using the graph data structure we built for the GRAPPA library. The data structure is similar to that in Figure 2.7 except each cell contains the adjacency list for a vertex. Unlike GraphLab we do not focus on intelligent partitioning; instead, we chose a simple random placement of vertices to cores. Edges are stored co-located on the same core with vertex data. Using this graph representation, we implement a subset of GraphLab's synchronous engine, including the delta caching optimization, in ∼60 lines of GRAPPA code. The graph library defines parallel iterators over the vertex array and over each vertex's outgoing edge list. Given our graph structure, we can efficiently support gather on incoming edges and scatter on outgoing edges. Users of our Vertex-centric GRAPPA framework specify the gather, apply, and scatter operations in a "vertex program" structure. Vertex program state is represented as additional data attached to each vertex. The synchronous engine consists of several parallel `forall` loops executing the gather, apply, and scatter phases within an outer "superstep" loop until all vertices are inactive.

We implemented the PageRank graph analytics application using vertex program definitions equivalent to GraphLab's. For our implementations and evaluation of additional applications—Single Source Shortest Path (SSSP), Connected Components (CC), and Breadth-first search (BFS)— I refer the reader to [113].

To evaluate GRAPPA's Vertex-centric framework implementation, we ran PageRank on the Twitter follower graph [85] (41 M vertices, 1 B directed edges) and the Friendster social network [158] (65 M vertices, 1.8 B undirected edges). We the algorithm to convergence using GraphLab's default threshold criteria, resulting in the same number of iterations for each system. We ran with GraphLab's delta caching enabled, as it performed better for this application.

For GRAPPA we use the no-replication graph structure with random vertex placement; for GraphLab, we show results for random partitioning and the current best partitioning

strategy: "PDS" which computes the "perfect difference set" but can only be run with $p^2 + p + 1$ (where p is prime) nodes.

Figure 2.13a depicts performance results at varying numbers of nodes. We saw that Grappa is faster than the best partitioning despite using the inferior random partitioning. When running PageRank, both systems issue application-level requests on the order of 32 bytes (mostly communicating updated rank values). However, since this message size would perform terribly on the network, both systems batch updates into larger wire-level messages. Grappa's performance exceeds that of GraphLab primarily because it batches faster.

Figure 2.13c(bottom) explores this difference using the GUPS benchmark from Section 2.4.3. All systems send 32-byte updates to random nodes which then update a 64-bit word in memory: this experiment models only the communication of PageRank and not the activation of vertices, etc. For GraphLab and Spark, the messaging uses TCP-over-IPoIB and the batchers make batches of 64 KB (GraphLab also uses MPI, but for job start-up only). At 31 nodes, GraphLab's batcher achieves 0.14 GUPS, while Grappa achieves 0.82 GUPS. Grappa's use of RDMA accounts for about half of that difference; when Grappa uses MPI-over-TCP-over-IPoIB it achieves 0.30 GUPS. The other half comes from Grappa's prefetching, more efficient serialization, and other messaging design decisions. The Spark result for GUPS is a best case (no batching overhead), which we obtained by writing directly to Spark's `java.nio`-based messaging API rather than Spark's user-level API.

During the PageRank computation, Grappa's unsophisticated graph representation sends 2× as many messages as GraphLab's replicated representation. However, as Figure 2.13c(top) shows, Grappa sends these messages at up to 4× the rate of GraphLab over the bulk of its execution. At the end of the execution when the number of active vertices is low, both systems' message rates drop, but Grappa's simpler graph representation allows it to execute these iterations faster as well. These factors lead to an overall speedup of 2×.

Figure 2.13a shows the result of strong scaling experiments on both datasets. Scaling was poor beyond 32 nodes for both platforms due to the relatively small size of the graphs—there was not enough concurrency for either system to scale on this hardware. To explore how

Grappa fares on larger graphs, we show results of a weak scaling experiment in Figure 2.13b. This experiment ran PageRank on synthetic graphs generated using Graph500's Kronecker generator, scaling the graph size with the number of nodes, from 200M vertices, 4B edges, up to 2.1B vertices, 34B edges. We normalized runtime to show distance from ideal scaling (horizontal line), showing that scaling deteriorated less than 30% at 128 nodes.

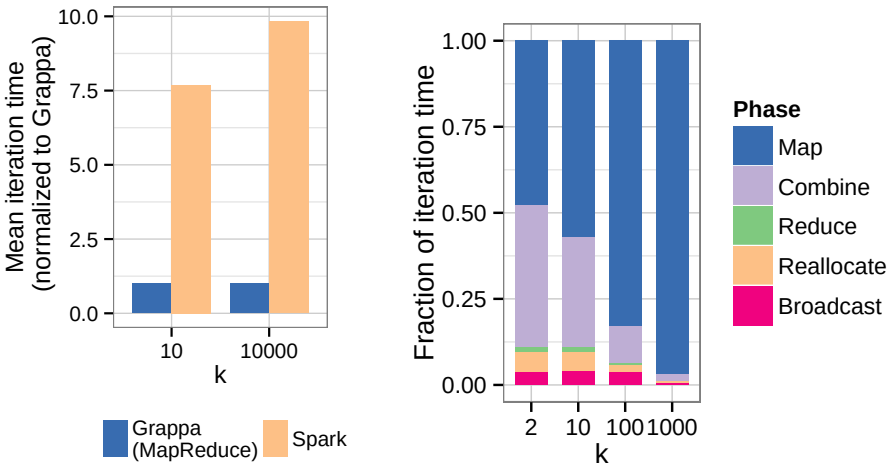
2.5.3 Iterative MapReduce on Grappa

We experiment with data parallel workloads by implementing an in-memory iterative MapReduce API in 152 lines of Grappa code. The implementation involves a `forall` over inputs followed by a `forall` over key groups, all within a C++ while loop. In the all-to-all communication, mappers push to reducers. As with other MapReduce implementations, a combiner function can be specified to reduce communication. In this case, the mappers materialize results into a local hash table, allocated with a symmetric address.

We pick k-means clustering as a test workload; it exercises all-to-all communication and iteration. To provide a reference point, we compare the performance to the `SparkKMeans` implementation for Spark. Both versions use the same algorithm: assign each point to a cluster (map), calculate the new cluster means (reduce), and broadcast local means. The Spark code caches the input points in memory and does not persist partitions. Currently, our implementation of MapReduce is not fault-tolerant. To ensure the comparison is fair, we made sure Spark did not use fault-tolerance features: we used `MEMORY_ONLY` storage level for RDDs[2], which does not replicate an RDD or persist it to disk and verified during the runs that no partitions were recomputed due to failures. We ran k-means on a dataset from SeaFlow [141], where each instance is a flow cytometry sample of seawater containing characteristics of phytoplankton cells. The dataset is 8.9GB and contains 123M instances. The clustering task is to identify species of phytoplankton so the populations may be counted.

The results are shown in Figure 2.14a for $K = 10$ (a realistic setting for this dataset) and

[2]A *resilient distributed dataset*, or RDD, is a partitioned collection of data in Spark.



(a) (b)

Figure 2.14: Data parallel k-means clustering on 64 nodes. The input is a 8.9-GB SeaFlow dataset. The plots are a) running time and b) breakdown of time spent in Grappa-MapReduce phases.

$K = 10000$. We find Grappa-MapReduce to be nearly an order of magnitude faster than the comparable Spark implementation. Absolute runtime for Grappa-MapReduce is 0.13s per iteration for $K = 10$ and 17.3s per iteration for $K = 10000$, compared to 1s and 170s respectively for Spark.

The bulk of the difference comes from the networking layer and from data serialization. Figure 2.14b shows the time Grappa spent in different phases of one iteration. As K grows, this problem *should be* compute-bound: most execution time is spent assigning points to clusters in the map step. At large K, Grappa-MapReduce spends 97% of its time in map, but Spark spends only 50% of time there; the rest is in network code in the reduce step. Grappa performed well because of its support for small messages and overlapping communication with computation.

2.5.4 Relational queries on Grappa

We implemented a relational query engine on Grappa, which executes iterative SQL queries. Grappa's ability to execute fine-grained tasks is a good match for hashing-based algorithms for aggregating and joining tables. The full system includes a parallel query planner, code generator, and library of data structures. This engine is the subject of Chapter 4, which includes another performance comparison with Spark (Section 4.8).

2.6 Discussion

My research question **RQ1** asked *"How can we build a high performance system for data-intensive computing that accommodates different data processing models alongside flexible parallel programming?"* The work in this chapter attempts to raise the ceiling of performance for data-intensive applications without greatly raising the floor of programming effort. To that end, Grappa is inspired not by symmetric multiprocessor systems but by high-performance PGAS languages and novel supercomputer hardware—the Cray MTA and XMT line of machines. This work borrows the locality-awareness of PGAS and the core insight of the MTA/XMT—that many data-intensive applications have enough concurrency to tolerate latency—and uses them to build a software runtime tuned to extract performance from commodity processors, memory systems, and networks. Our data demonstrates that frameworks such as Vertex-centric and MapReduce are easy to build and efficient on such a DSM system. The primary factor to improved performance over existing implementations of those frameworks is Grappa's efficient message batching; Grappa's lightweight multithreading and shared memory make the simple programming model possible. In my opinion, every global-view PGAS language implementation should include—as a toggled option—message batching of the sort Grappa has.

Our implementation and evaluation of a relational query engine on Grappa is the subject of much of the rest of this thesis. But first, the next chapter will give relevant background on in-memory query evaluation.

Chapter 3

STATE OF THE UNION: RELATIONAL QUERY EVALUATION

Over the last decade and a half, researchers redesigned database management systems (DBMS) and algorithms for modern hardware, comprised of lots of DRAM, multiple levels of cache, and multiple cores with the ability to exploit data, thread-level, and instruction-level parallelism. The result of this effort was an orders-of-magnitude improvement in performance for in-memory query processing.

This chapter begins to address **RQ2**: *"What techniques can improve the performance of database-like query processing in a high performance parallel system?"* I examined the prior research on understanding and improving the performance of declarative query processing in memory. First, I present the surveyed causes of inefficiency for conventional techniques applied to in-memory query processing. Then, I present approaches to mitigate these inefficiencies. I categorized the approaches into: specialization to remove overheads, changing the execution model, and using the memory hierarchy more efficiently. Finally, I discuss specialization—in the form of query compilation—in more detail, as it informs my further investigation of **RQ2**.

3.1 Techniques for in-memory processing

The conventional approach to analytical query processing is demand-driven dataflow, commonly known as pull-based iterators. Each relational operator in the dataflow graph is implemented as an iterator that processes one tuple at a time [60]. This design is popular because the iterator interface is intuitive and allows the implementation of each operator to be independent.

Table 3.1: Three types inefficiency in query processing and how they affect areas of the system. The areas of the system and emphspecific causes (the cell values) are taken from [163]. I identified the three types of inefficiency and placed the specific causes into them.

Area of system	Generality (fully interpreted)	Execution model (tuple-at-a-time iterators)	Storage model (N-ary, i.e., rows)
CPU I-cache	# instructions	# instructions and not operator-centric	
CPU D-cache	touching structures	not data-centric at CPU level	internal fragmentation
Function calls	overhead		
Tuple manipulation	overhead		extracting attribute from tuple
CPU utilization	instruction mix	instruction mix	
Compiler optimizations	not compiled or limited scope of optimization	code not amenable to optimization	
Data volume			moving all attributes

Unfortunately, demand-driven dataflow is inefficient for in-memory query processing on modern CPUs in a number of ways. In Table 3.1, I list the areas of the system where inefficiency occurs and their various causes [163]. I categorized each cause into one or more of three categories: *generality*, *execution model*, and *storage model*.

Recent techniques for improved in-memory performance mitigate fall one of these categories. Table 3.2 lists the techniques by category. The techniques that target *generality* do so by specializing the query engine according to aspects of the query, going as far as to hand-write a program for a single query. The techniques that target *execution model* relate to processing tuples in chunks and data-centric execution that pushes a datum through the processing pipeline. The techniques that target *storage model* do so by minimizing the data moved through the memory hierarchy. In the rest of this section, I summarize several of these techniques.

Table 3.2: Surveyed techniques for fast query processing, addressing three sources of inefficiency

Generality	Execution model	Storage model
type-specialization generation and compilation of for-loop code compiler optimization across operators generative programming techniques domain-specific processor per-query accelerator	block-oriented processing operator-at-a-time vector-at-a-time vectorization+pipelining query optimization that considers v+p	column storage estimation of efficient vertical partitions compression

3.1.1 Block-oriented processing

There is a spectrum of granularities by which a system can process the input of a subquery, ranging from data-centric to operator-centric. On the data-centric end is tuple-at-time processing where operators are pipelined: a single tuple travels as far down the operator graph as possible before the next tuple is processed. On the operator-centric end is operator-at-a-time processing, where an operator processes all tuples before the next operator starts. The two ends of the spectrum have different locality trade-offs.

Tuple-at-a-time processing has poor code locality because the operator that is active continuously changes. In demand-driven dataflow, the problems are even worse. Each tuple that is processed incurs stalls from branching and instruction overhead. Further, accessing the chain of operators in the pipeline results in poor data and instruction cache usage, as each operators data structures and code are accessed. These pipelining overheads can be reduced by extending demand-driven dataflow to operate on blocks of tuples—somewhere in the middle of the spectrum. Retrofitting systems to process tuples in blocks provided moderate performance gains [162, 122].

Designing systems from the ground-up to use block-oriented query processing on modern

processors have achieved far higher performance. MonetDB [26] reduced per-tuple overheads with operator-at-a-time processing. It also improved cache bandwidth and storage utilization by using a decomposition storage model [40], commonly known as column storage. MonetDB is an order of magnitude faster on analytical queries than conventional DBMSs. However, MonetDB had significant inefficiencies in complex queries for two different reasons. First, its operators comprised primitive operations, like integer add, providing a narrow instruction mix for superscalar processors when applied repeatedly to a whole column. Second, excessive materialization of intermediate results (i.e., saving them temporarily in memory) wastes memory bandwidth by successively bringing data in and out of the cache and registers.

These problems were addressed in MonetDB/X100 [27] (which became the commercial DBMS VectorWise [164]) by introducing *some* pipelining back into the processing. To introduce pipelining into MonetDB, X100 used *compound* primitives, which are more complicated operations but still vectorizable by the C compiler. Executing more of the query each time a vector is brought into the cache improved memory bandwidth usage. The greater complexity of the primitives also improved instruction mix. An example of these compound primitives is adding the square of two fields. Even when a compound primitive did not map directly to a single vector instruction on the processor, it could still be designed to provide better data locality to improve performance. Code for primitives can be structured so that the compiler can perform loop tiling [155]. An important effect of column-wise storage is avoidance of tuple access operations (pointer manipulation), which makes the code for operators easier to vectorize. In fact, MonetDB/X100's pre-compilation of every compound primitive for every possible combination of array input types would probably have been infeasible in a row store because the arity could be greater than two. MonetDB/X100 was about 4-10× faster on TPC-H queries than MonetDB.

3.1.2 Query plan compilation

Another approach for eliminating overheads of tuple-at-a-time query processing is to specialize by compiling queries. Most systems systems compile queries by generating a program

in an intermediate language and then compiling that program with an industry-strength compiler. Choosing the appropriate C implementation of a selection condition improved performance [132]. AT&T's Daytona had different modes of generating C code from its query language, but I could not find performance comparisons [61]. Compiling specialized demand-driven dataflow code eliminates virtual function calls for a DBMS written with generic iterators [128]. Reducing the number of virtual calls resulted in higher performance due to efficient memory access and more predictable branching.

A more holistic approach to compilation is to discard the demand-driven dataflow model entirely and generate source code for an entire query [84]. This approach, HIQUE, produces code that looks more like a handwritten implementation with for-loops, exhibits better data locality, and is claimed to be amenable to compiler loop optimizations [83]. Pipelining in the demand-driven dataflow model causes cache pressure as all operators may be touched to get the next tuple; while HIQUE also adopts pipelining, compiling push-based code results in far simpler code that mitigates cache pressure. A similar approach to code generation has been applied to improve the performance of .NET language integrated queries (LINQ) by an order of magnitude [106, 111] over interpretation.

Two drawbacks to holistic query evaluation were increased compile times and a lack of modularity in the code generator. If a new select algorithm was devised, then implementations of other operators might have to change to accommodate. HyPer's compiler [116] improves both problems. They decrease compile time by generating LLVM bytecode instead of source programs, and they improve modularity with an iterator-like interface for code generating operators (this second improvement is addressed in more detail in Chapter 5). Further, by targeting bytecode, the code generator has control over registers and can keep the current tuple in registers as long as possible. Using these techniques, HyPer queries were 2-4× faster than the previous approach.

3.1.3 Compiled pipelines versus vectorization

There is a trade off between block oriented execution (as in MonetDB/X100) and pipelined execution (as in HyPer); block oriented execution reduces interpretation overhead and pipelined execution increases data locality. The right choice for every segment of a query plan will depend on the particular situation. Sompolski et al. [140] examined the trade-off in case studies for projection/apply, selection, and hash probe and found both pipelining and vectorization to be important. In projection/apply, vectorization is critical, but it is beneficial to retain some pipelining. In selection, predication (turning a control dependence into a data dependence [47, 45]) is better than tight loops with data-dependent branching when selectivity is 20-80% (i.e., precisely when branch prediction fails). They found that the component operations of hash probing should each be considered for vectorization or pipelining. They concluded that vectorization of bucket lookups was necessary for one core to fully utilize memory bandwidth. However, the memory interface is oversubscribed in multi-core processors [100, 114], so I hypothesize that a join executed in parallel across many cores does not require vectorization to utilize memory bandwidth.

Cost-based query optimization has received limited attention regarding this trade-off. Choice of vectorized or pipelined map operations has been incorporated into the query planner of Tupleware [42]. Tupleware collects the statistics about CPU and memory behavior that are necessary to make this choice by using LLVM to analyze each UDF in the query. Introducing a choice of predication on a per-`select` basis creates an interesting dimension to query optimization, but we are only aware of work that explores this trade off manually [140] or in the manual design of operator algorithms [127].

3.1.4 Reducing memory bandwidth usage

Section 3.1.1 described how pipelining reduces the number of times a datum is transmitted through the memory hierarchy relative to column-oriented execution; here I discuss a number of other considerations for the memory hierarchy.

Column storage For analytical processing, column stores are significantly faster than row stores [1]. Relations tend to have many columns, while a query accesses few, so reading tuples in row format from memory often wastes bandwidth and pollutes the data caches. Column storage is inefficient for transactional workloads because they have more predictable access patterns than ad hoc analytics and must support row-wise updates.

Flexible vertical partitioning Vertical partitioning [112] algorithms slice tuples so that attributes are stored in a way that reflects expected access patterns. Predicting good vertical partitions has become important in recent hybrid transactional/analytical databases [62] where transaction performance depends on horizontal locality and analytical performance depends on vertical locality. Supporting flexible vertical partitioning in in-memory data management systems introduces another source of overhead from generality in the implementation, but query specialization can remove the overheads [126]. Drawing upon work on estimation of distinct record selection [29] for random accesses, cost models for memory access patterns [101] have been incorporated into query optimization and vertical partitioning [126].

Compression Compression in databases has been studied extensively for trading bandwidth and storage usage for CPU overhead. Compression in column stores is particularly effective since single attributes tend to have less entropy than the data as whole. In main-memory databases decompression is even more effective when it is performed between CPU and cache, although the relative CPU to I/O costs are higher. I refer the reader to [163] for a good survey of compression in databases.

3.1.5 Limit studies

So far, I have surveyed work on improving the performance of query processing systems, but it is natural to ask what the limits on performance are. I briefly discuss research projects that provide perspectives on the limits of query processing in modern hardware.

A limit study [49] hand-coded the queries from TPC-H and achieved 1-2 orders of magnitude speedup over the record holders, indicating there is room for improving even state-of-the-art query processing engines. The most important factors to performance were compilation, single-pass algorithms to reduce bandwidth usage, column-oriented in-memory processing, and non-uniform memory access (NUMA)-aware operator algorithms tuned for the particular architecture.

Research on specialized hardware provides another perspective on modern technological limits to query processing. The Q100 database processor [156] provides a different perspective of the architectural bottlenecks of analytical query processing. The authors explored the design space for a domain-specific processor for relational databases. The architecture is influenced by existing enhancements for taking advantage of modern hardware like column storage, pipelining, and vectorization. They evaluated the architecture on TPC-H. They found that more than half of the queries were insensitive to bandwidth between functional units and bandwidth to memory, but the other queries were extremely sensitive. In addition, they found that the spatial architecture of their processor, which is closer to the design of a spatial dataflow than a typical out-of-order processor, provides large performance and energy benefits compared to the out-of-order processor. The authors did not quantify the source of these benefits over conventional processors in detail, but the results suggest that there is sufficient *dataflow locality* [142] in analytical queries to merit alternatives to today's commodity server architectures. LINQits [37] takes a more extreme approach of turning individual queries into accelerators implemented on FPGAs. Other than accelerating the logic of user-defined functions, they speculate that performance improvements came from a customized memory system: the custom hardware exploited dataflow locality and coalesced memory accesses. Finally, Oracle's proprietary Sparc M7 processor [94] is highly customized for database processing. Although I am not aware of any public performance results, the designers focused on attempting to match memory rates with database-specific vector units (bit-level flexibility unlike most SIMD instructions), near-memory processing, and decompression accelerators.

3.2 Compilation in distributed query engines

A number of distributed query systems have incorporated query compilation to remove overheads or compete with hand-coded queries. Distributed Socialite [137] generates Java code to integrate Java functions with Datalog and compare performance to hand-coded algorithms [136]. It has a master-slave runtime for communication. Tupleware [42] generates C++ code for vectorized or pipelined query plans. For communication between nodes, the system is limited to using library calls for block-based fetching of data over the network. Commercial analytic parallel databases Cloudera Impala [149] and ParAccel [123] compile fragments of queries to achieve up to 2× and 20× higher performance, respectively; the memory bus is the bottleneck in many ParAccel queries. Steno, mentioned earlier, also runs its compiled LINQ queries on DryadLINQ for distributed execution.

In all of these systems, code generation provides a performance boost but the architecture is limited to stitching together sequential fragments of the query plan with a communication library. At best, communication library calls can be inlined for local optimization [42], but such transformations do not extend the compiler's scope to the code the that consumes the messages. The intermediate compiler can only translate and optimize the fragments in isolation.

The role of compilers in query processing To aid understanding of the role of general-purpose compilers in different query processing systems, I plotted the work from this section in Figure 3.1. The code that is compiled by the underlying machine compiler is generic database code, type-specialized code, or code generated for a specific query. The axis could be extended further to include systems that do dynamic query optimization *and* re-compile the code, but I am only aware of proposals [82, 147]. The *optimization scope* is how much of that code the underlying compiler is able to consider as one unit. "Expression" means a function applied to scalars (i.e., tuples and attributes) and "operator" means a function on relations (i.e., all the tuples). For some regimes, code inlining by the compiler might move the scope

to the right (e.g., expression moves toward operator when tuple access functions are inlined within the `next` function) but not significantly. The design of MonetDB and MonetDB/X100 generated functions for every combination of operation, property, and type when the database is compiled [163]. "Maximal pipeline, up to communication" means that fragments of a pipeline that do not contain communication can each be compiled as a unit. Most query compiler systems are limited in this way because they generate sequential code for pipelines and communication is implemented with a send and receive, whose connection is opaque to the compiler. The entire pipeline can be compiled as a unit if the compiler understands the communication. This novel category is "maximal pipeline", and I explore it in RADISH, my work on query processing in PGAS languages. RADISH is alone in "maximal pipeline" because it is the only system where the underlying compiler is locality-aware, allowing it to look across communication points in the code generated for a pipeline. I quantitatively compare RADISH implementations from the "operator" and "maximal pipeline" categories.

All of the systems I plotted in the last row of Figure 3.1 use two distinct tools for translating a query to machine code: the query planner and an existing source code or bytecode compiler, but other architectures are possible. In Chapter 5 I will discuss optimization flows that have more than these two distinct steps [82, 138].

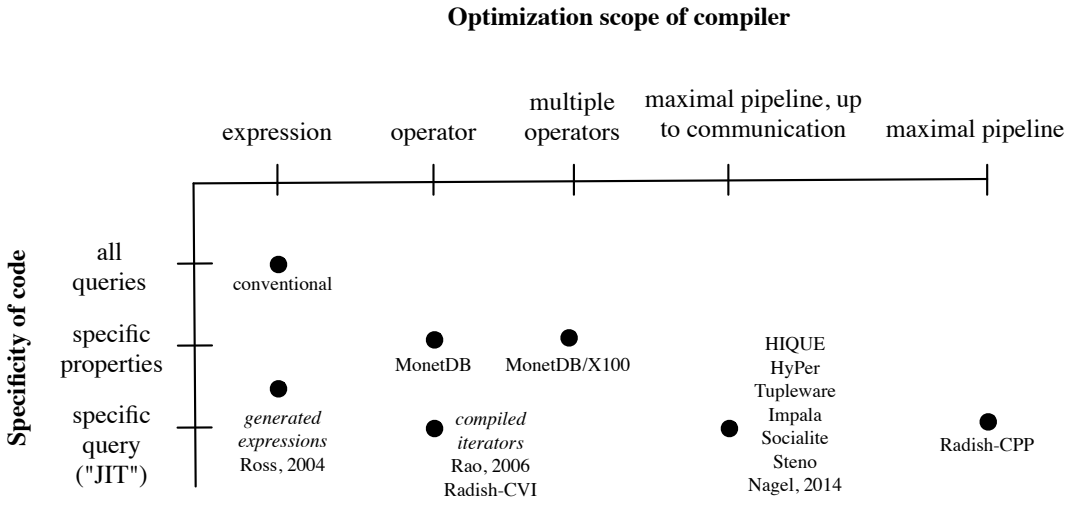


Figure 3.1: Compilation in query engines projected onto two dimensions. The optimization scope is how much code the underlying compiler (e.g., LLVM, GCC) can consider as one unit. The code that is compiled is generic database code, type-specialized code, or code generated for a specific query.

3.3 Discussion

In preparation to answer **RQ2** ("What techniques can improve the performance of database-like query processing in a high performance parallel system?"), this chapter surveyed the existing techniques for fast in-memory query processing. I categorized inefficiencies in the conventional approaches as due to generality, execution model, or storage model. In particular, I focused on reducing generality with query compilation. The next chapter will attack **RQ2** directly, using query compilation as a primary tool.

Chapter 4

COMPILING QUERIES FOR THE HIGH-PERFORMANCE STACK

I designed Grappa with runtime features that reduce programmer burden for achieving high performance on data-intensive applications. However, Grappa's HPC programming model presents a barrier to productivity if it is to be applied to ad hoc analysis. Analysis codes are often used briefly, so the economics of developer time versus utility do not favor HPC languages. Declarative programming is the modus operandi of data management systems. The high productivity enabled by declarative programming is critical for *ad hoc* analyses.

In this chapter, I address **RQ2**, *"What techniques can improve the performance of database-like query processing in a high performance parallel system?"* I describe Radish, our implementation of an in-memory query processing system for PGAS languages like Grappa. The design is inspired by techniques surveyed in Chapter 3, and in particular, I quantitatively explored the use of compiled pipelines (introduced in Section 3.1.2). In this research, I learned that compiling pipelined plans is important for performance, and I developed, to the best of my knowledge, the first query-to-parallel-program compiler. I also learned that Grappa was a suitable target for such a compiler because it efficiently executes the fine-grained tasks that the approach admits. Near the end of this chapter, I present a case study that I performed to explore **RQ3**, *"How close can the performance of declarative programs implemented in high performance systems come to handwritten programs?"* Although I found that Radish's generated programs were nearly an order of magnitude slower than handwritten parallel programs, I provide evidence supporting why better performance is within reach.

4.1 Introduction

Data-intensive applications are motivating new interactions between the models of databases and the algorithms and platforms of high-performance computing. The use-case that I suggested in Chapter 1 was in-situ analysis of output from a large cosmology simulation. Considering the cost of development, ad hoc analytics tasks are a severe mismatch for the distributed programming models of HPC. When programmers express their analytics tasks in terms of high-level dataflow, there are costs: 1) existing dataflow systems do not make efficient use of HPC environments characterized by fast interconnects, fast messaging libraries, and high CPU to I/O capacity [96]; and 2) crossing system boundaries incurs the cost of impedance mismatch in data representation. Hand-tuned parallel algorithms continue to play a role in real applications; for example, GenBase calls out to the distributed dense linear algebra package ScaLAPACK for parts of its complex analyses [143]. Intermingling high-level dataflow programming with point-to-point message-passing algorithms is an emerging requirement [79].

I present new techniques for compiling queries for distributed HPC environments, aiming to exploit both database-style algebraic optimization and the performance benefits of parallel compilers. We targeted partitioned global address space (PGAS) languages like Grappa, which provide shared-memory programming as well as partitioned memory to enable the programmer (and compiler) to perform locality-aware communication optimizations. By targeting PGAS we exploit 1) *language constructs* for flexible parallel execution, data layout, and task migration, 2) *runtimes* built upon MPI, the standard of high-performance messaging, and 3) *compilers* that are parallel aware and offer optimizations that complement those of databases.

A distributed query compiler for PGAS environments requires the design of a new architecture for distributed query evaluation, raising challenges. Expressing query execution in distributed memory using the shared memory model results in frequent expensive fine-grained global memory operations. Although use of shared data structures enables flexible

execution and optimization, fine-grained sharing has a cost—one that is exacerbated greatly by distributed memory. We must also adapt existing algorithms for distributed query execution such that they capitalize on the capabilities of the PGAS languages and compilers. As discussed in Section 3.2, existing distributed query processing systems generate sequential program fragments and stitch them into a distributed program using calls to a networking library. Applying this existing approach to the generation of PGAS code would obfuscate optimization opportunities around parallelism and locality that the PGAS compilers were designed to exploit.

In this chapter, we introduce *Compiled parallel pipelines* (CPP), a technique that generates efficient parallel programs. We implement CPP in a new parallel query processor called Radish. To evaluate our technique, we compare the performance of code generated by CPP to that of *Compiled Volcano iterators* (CVI), a query processor built in Radish using the iterator model. CVI uses the same query plan and same data structures. It also has compiled expressions and tuple materialization so that we eliminate interpretation overheads and focus our evaluation on the novel aspects of CPP relative to other query compilers.

To evaluate whether our implementation of Radish achieves acceptable performance, we also compare it to Impala, a commercial parallel database[1] that uses code generation, using TPC-H.

We make the following contributions:

1. CPP, an approach for translating queries into efficient PGAS code leveraging existing parallel compiler. To the best of our knowledge, this is the first system that does "maximal pipeline" query compilation, as defined in Section 3.2.

2. A library of distributed data structures for PGAS query processing supporting efficient fine-grained operations

3. An end-to-end, open source implementation integrating a distributed query optimizer

[1] Also known as a *massively parallel processing* (MPP) database

(Raco) and our PGAS system Grappa

4. An experimental evaluation of CPP compared to a conventional iterator strategy. On TPC-H queries, executing queries by CPP is on average 2.4× faster than CVI. We also compare Radish with a state-of-the-art database platform on TPC-H queries and find that its performance is competitive.

5. A case study on the performance of Radish-generated code versus a handwritten Grappa program. Specifically, we use PageRank, show the causes of slowdown in the generated program, and give support for why parity is within reach.

6. A performance comparison to the dataflow system Spark. Continuing the analysis from Section 2.5, we examine where the time goes in both systems.

The rest of this chapter is organized as follows. Section 4.2 defines an abstract model of a PGAS language that we target with code generation. Section 4.3 explains how to evaluate queries with PGAS programs. Section 4.4 describes the code generation abstractions and algorithm. Section 4.5 discusses how the code maps to concrete PGAS runtime systems. In Section 4.7, we present our evaluation of CPP and the overall Radish system, and in Section 4.8, we present our comparison to Spark. Section 4.9 discusses the applicability of Radish. Finally, I conclude and review the research questions in Section 4.10.

4.2 Targeting PGAS with a compiler

We target PGAS[2] languages as our model—over other parallel programming interfaces—for four reasons. First, the model provides a means for data layout, a flexible execution model based on task-level parallelism, and support for migrating tasks, which we use for communication. Second, PGAS runtimes are built on MPI [55] or other HPC messaging APIs, for which most high-performance networks have an optimized implementation. Third,

[2]more precisely, we refer to *global-view PGAS* [30] as PGAS in this chapter. Grappa is also a global-view PGAS language.

`spawn t body`	Asynchronously run `body` as task `t` on the local partition
`t.sync`	Block until task `t` is finished
`forall i in I body`	Run an iteration of `body` for each element in `I`. Iterations *may* run in parallel, so they are allowed to synchronize with each other but may not have inter-dependences, such as a barrier. The iteration for element I_j will start running on the partition where element I_j resides.
`on partition(L) body`	Move to the partition `L` before executing `body`. `L` may be a constant expression or dynamically evaluated expression of address type. If there are further statements after an `on partition`, then the task moves back to the original partition.
`global_new [T](N)`	Allocate global array where elements have type `T` and where the `N` elements are distributed evenly between partitions in a block pattern
`global_delete globalptr`	Free the global data pointed to by `globalptr`
`atomic { body }`	Execute `body` atomically

Table 4.1: PGAS model to target with code generation

compilers for these languages are locality and parallelism aware, meaning that they may offer optimizations that complement the algebraic optimization techniques associated with databases. Fourth, targeting shared memory code and relying on the compiler and runtime for certain low-level optimizations simplifies the design of the system, making a comprehensive distributed query compiler feasible.

The techniques in this chapter refer to our abstract model of a PGAS language in Table 4.1. In addition to these language constructs, our model includes the features of a typical shared memory language: local heaps, stacks, and control structures.

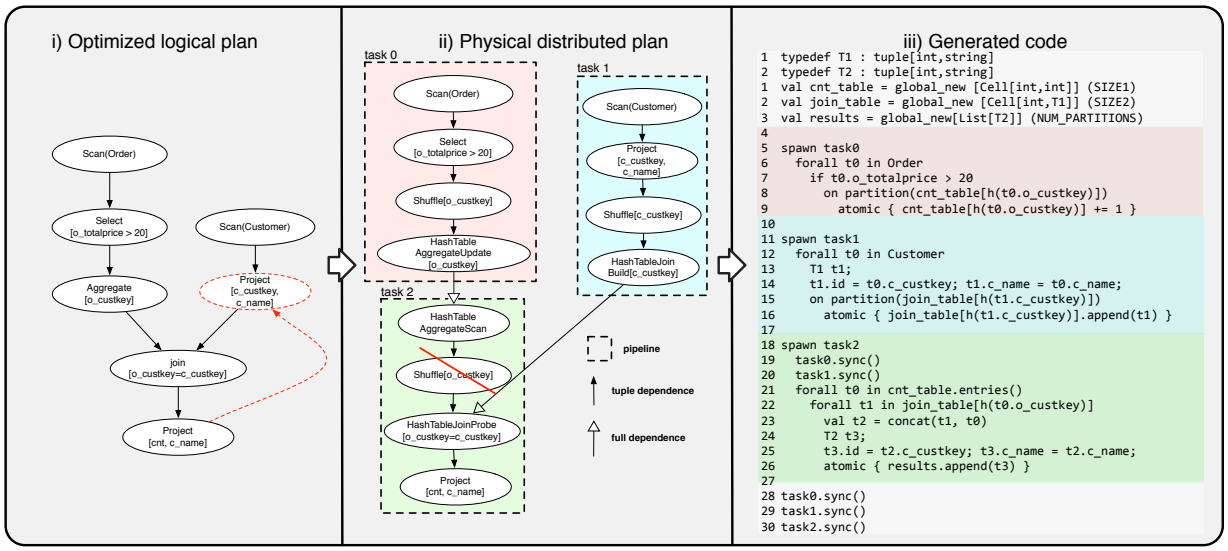


Figure 4.1: System overview: translation from query plan to PGAS code. (i) query plan is optimized to reduce communication and processing. (ii) logical operators are translated to physical operators; operators that depend on all inputs before executing break the plan into pipelines. (iii) for each pipeline do a bottom-up traversal to generate push-based code.

4.3 Query evaluation in PGAS

To illustrate how a PGAS program might evaluate a query, we consider the query "*find the number of orders with a price over 20 for each customer with any such orders*". In SQL:

```
1 select cnt, c_name
2 from Customer,
3      (select count(*) as cnt, o_custkey
4        from Order
5        where o_totalprice > 20
6        group by o_custkey) CountByCust
7 where c_custkey = o_custkey;
```

Figure 4.1 illustrates the steps of RADISH's code generation for this query. In (i), a logical plan is formed and optimized; shown is pushing a projection operator to eliminate unused attributes. In (ii), the physical plan is chosen; for example, to implement the `Aggregate`,

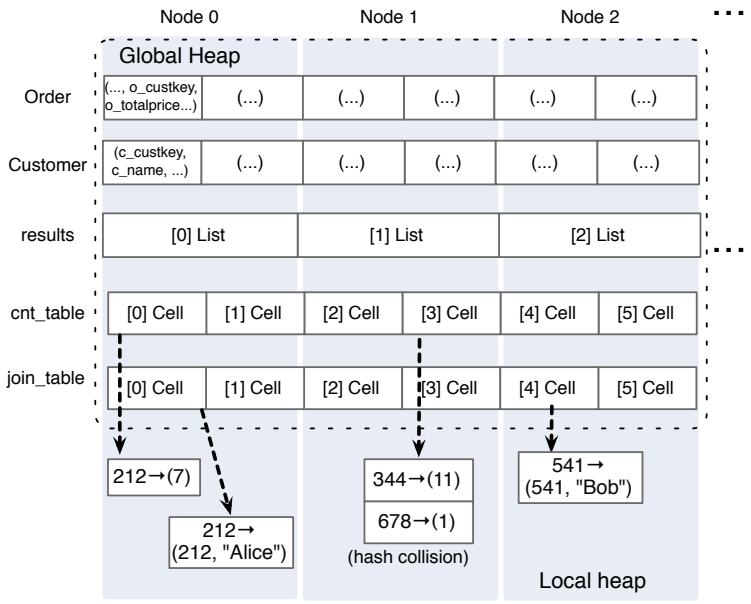


Figure 4.2: Physical layout of data across compute nodes in a distributed memory cluster for the example query. Each node is a PGAS *partition*, although other mappings are possible. The diagram represents a snapshot in time while `task0` and `task1` are still executing.

the optimizer chooses a hash-based aggregation algorithm, implemented by the operators `Shuffle` (to partition tuples by `o_custkey`), `HashTableAggregateUpdate` (to count tuples using a hash table) and `HashTableAggregateScan` (to read the counts). The query optimizer also applies locality-aware optimizations common to MPP databases, such as removing the redundant `Shuffle` on `o_custkey` (indicated by a cross-out). Panel (iii) shows the query as a program written in the PGAS model. Pipelines, outlined by dotted lines, are segments of the physical plan that stream data tuple by tuple. For each pipeline, RADISH emits code PGAS code that materializes an intermediate (as in `task 0` and `task 1`) or final result (as in `task 2`).

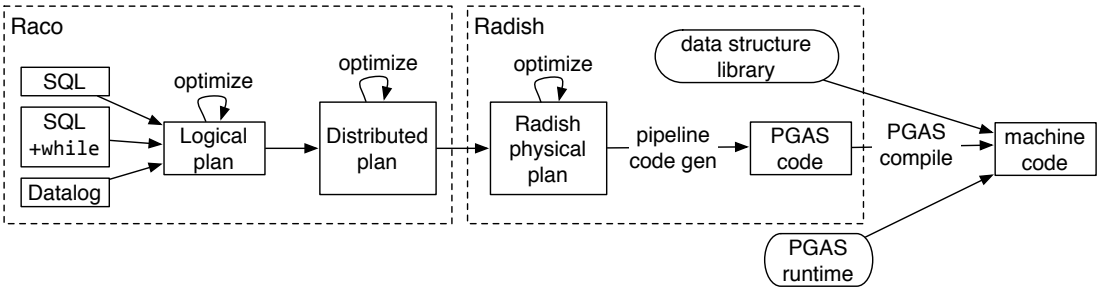


Figure 4.3: Query evaluation in RADISH. Boxes are representations of the query and arrows are transformations. The distinguishing feature compared to other query compilers is the generation of PGAS code, which is compiled by a distribution-aware compiler.

Highlights of the PGAS program The rest of this section explains the PGAS programs in detail; here, we give the salient features of the program in our example. Pipelines run concurrently via calls to `spawn` (lines 5, 11, and 18) and their interdependencies are encoded by calls to `sync` (lines 19 and 20). All inputs and intermediate results are stored in global arrays distributed across partitions using the data structure in Figure 4.2. These data structures are updated using `atomic` operations and iterated over in parallel using `forall`. The communication operators, `Shuffle`, are implemented by `on partition`. Running an `on partition` is like creating a closure and running it on a remote node, so accessing the same data before and inside of the body implies communication. For example, `t0` is read on line 9 and because `t0` originally resides on a different partition determined by the distribution of `Order`, `t0` is copied between partitions.

Radish: end-to-end Figure 4.3 shows the end-to-end system. RADISH sends the emitted PGAS code and RADISH data structures through the PGAS compiler, which links with the PGAS runtime to generate machine code. Low level optimizations performed by the PGAS compiler are complementary to those of RACO and RADISH. Depending on the quality of code emitted by RADISH, the PGAS compiler can also apply higher level optimizations, like communication avoidance, although we do not explore them in depth.

RADISH receives an optimized query plan from the RACO optimizer.[3] Its plan rewriting library includes logical heuristics (moving filters, applies, and projects, join ordering, symmetric or build/probe hash join), as well as distributed heuristics (avoiding redundant `Shuffle` and using broadcast for cross product with a small relation).

4.3.1 Physical operators

A RADISH physical plan composes operators from the RADISH physical algebra, where each operator represents a transformation of an input relation that is realizable in PGAS. The variety of operators in this algebra is determined by 1) algorithms for parallel data transformations, 2) interfaces to global data structures, and 3) granularities of synchronization.

Algorithms RADISH operators build upon well-known algorithms for distributed, parallel query evaluation. Joins and aggregates match tuples using either sort-merge or hashing. The major departure from other implementations of these algorithms is that we implemented them in a shared memory paradigm, where data structures are partitioned yet accessible to all workers.

Interfaces to global data structures Global data structures provide implementations of one or both of the following interfaces: *global-view* and *partition-view*. Global-view operations encapsulate communication and partition-view operations touch only the local partition. In Figure 4.1(ii), we have shown only partition-view for clarity: by using operators that do partition-view updates, the `Shuffle` operators are explicit in the physical query plan. Having explicit `Shuffle`s in the plan allows for allows the query planner to perform certain locality-aware optimizations (e.g., removing redundant a `Shuffle`, as shown).

Data structures with a partition-view update method do not typically require a global-view update method. Since `Shuffle` is pipelined, a partition-view update composed with a `Shuffle` generates the code for a global-view update. Thus, a global-view update method

[3]https://github.com/uwescience/raco

is typically used when the target language already provides an implementation of the global data structure (e.g., Chapel has distributed associative arrays [30]).

Granularities of synchronization The operators include those with *tuple-grain synchronization* and *relation-grain synchronization*. Tuple-grain synchronization is where individual tuples are added to a shared data structure. Relation-grain synchronization is the situation where a full intermediate result can be written before it is read. `HashTableAggregateUpdate` / `HashTableAggregateScan` (Figure 4.1(ii)) involves both types of synchronization and fully pipelined operators like symmetric hash join (see Section 4.3.3) use only tuple-grain.

Relation-grain synchronization breaks the physical plan into pipelines. Figure 4.1(ii) shows the plan for the customer order count query. Black arrows indicate tuple dependences, where pipelining may occur, and white arrows indicate full materialization dependences. Full materialization dependences are created when a logical operator is broken into two physical operators, one of which depends on the full results of the other. These operators, such as the build materialization for hash table join (`HashTableBuild`), are called *pipeline breakers*. Pipeline-breaking is accomplished by the code generation algorithm.

4.3.2 Parallel code for one pipeline with CPP

The overall design space for PGAS algorithms is large; we made design decisions that would have high pay off for many queries. The structure of our parallel programs is inspired by compiled push-based pipelines, as described in Section 3.1.2. RADISH produces one code fragment for each pipeline by using PGAS constructs for expressing locality, communication, and concurrency. The result is holistic code, where each pipeline is compiled and optimized as a whole by the PGAS compiler. Further, the code emitted by RADISH is data-centric for locality and uses async-finish parallelism, which makes fine-grained parallelism more efficient.

Push-based and data-centric RADISH generates push-based, data-centric code for each entire pipeline, pushing one tuple at a time through the operators. RADISH uses `forall` to

iterate over inputs and intermediate results. Section 4.5 elaborates on how unlike explicit task-based query execution, using `forall` allows scheduling to be decided by the compiler and runtime system. `Forall`s may be nested, which is required to implement pipelined operators that produce multiple outputs, such as flatmap and join.

By using `forall` to drive the pipeline, each tuple is allowed to proceed through the pipeline independently. The `forall` only demands that any schedule of iterations must be serializable to guarantee execution is correct and deadlock-free. Physical operator implementations emit calls to the global data structure interface.

The data-centric code is efficient because it brings the tuple into the processor cache once. However, it is not trivial to maintain locality when the pipeline contains an `on partition` that pushes the tuple to another partition for further processing. Section 4.5 describes how Radish extends the data-centric property to the distributed setting by keeping the working set as small as possible.

Async-finish parallelism A property of the physical operators we have implemented in Radish that affects performance is *async-finish* parallelism, as described in Section 2.4.2. The analogy to basic tuple processing is intuitive: within one pipeline, an input tuple produces an output independently of other input tuples and the only consumer of that output is other pipelines. The async-finish property allows for very efficient concurrent execution.

It is of course possible to write PGAS algorithms in Radish that are not async-finish. In fact, there is an interesting trade-off: while async-finish algorithms require fewer synchronization messages, *non*-async-finish algorithms are often more memory-efficient when finishing a task frees memory. For example, consider the query

```
1 select x,y,sum(z) from R group by x,y;
```

An async-finish algorithm is

```
1 allocate large hash table ht
2 forall x,y,z in R: on partition(ht[(x,y)]) { ht[(x,y)] += z }
3 forall x,y,sz in ht: emit (x,y,sz)
```

This code uses memory for the hash table proportional to the number of unique pairs (x, y). There is a non async-finish algorithm that uses memory proportional to the degree of actual parallelism because it deletes space as it is no longer needed:

```
1 // if R is already sorted or clustered by x
2 forall x, lst in Rclustered:
3   allocate small hash table ht
4   forall (y, z) in lst: ht[y] += z
5   forall y, sz in ht: emit (x,y,sz)
6   deallocate ht
```

This algorithm is not async-finish because the first inner `forall` (line 4) spawns new tasks that must finish before the remainder of the outer loop's body (lines 5 and 6) can execute.

4.3.3 Shared data structures

We designed new data structures—closely considering synchronization and data movement—to implement Radish's physical operators. Tasks sharing a physical operator coordinate through shared data structures. The data structures are distributed across partitions, and their internal structures are organized to enable efficient data movement. Because pipelines are driven by `forall`s, any two iterations could be concurrent; therefore, the data structures must support tuple-grain synchronization between producers, as opposed to only relation-grain. When a communication operator (e.g., `Shuffle`) is pipelined, it must support tuple-grain synchronization between producers *and* consumers. Pipeline-breaking operators (e.g., `HashTableAggregate`) can rely on data structures that have relation-grain synchronization between producers and consumers. When there is only relation-grain synchronization between producers and consumers, async-finish parallelism is possible. The pipelines that share a data structure manage its memory.

Examples of data structures We implement asymmetric hash join using a global hash table implemented using a global array. We implement symmetric hash join using a global double hash table implemented using a pair of global arrays. We implement aggregation using a global hash table implemented using partition-local hash tables. We store unsorted relations using a global collection implemented with either a global array of tuples or partition-local vectors.

Data movement Naïve shared memory code will incur a network message for every memory load and store involved in accessing a global data structure, as in the histogram example of Section 2.2. Consider the `cnt_table` in Figure 4.2 used for aggregation: adding a new tuple involves a read of the cell, a read of each element in the collision list, a read of the current value, and a write of the new value. To reduce the number of round trip messages between partitions, RADISH uses distributed continuation passing [77]. RADISH data structures are laid out to exploit locality (all internal data related to a single entry lies on a single partition), and the entire update is moved to the data using `on partition`. This transformation increases performance by up to an order of magnitude in communication-intensive applications [21, 68, 161].

Using `on partition` also simplifies data movement in operator implementations because the PGAS compiler takes responsibility for detecting the data accessed in the continuation. One example is passing shared references to dynamically allocated data structures (e.g. `join_table`) through `forall` and `on partition`: the PGAS compiler detects that the data structure reference never changes. It therefore broadcasts the global reference to all partitions before any loops so that they reference the local pointer. We did a case study on the join hash table of *Q17* from TPC-H, where a reference to a global data structure can be removed from the messages sent in the critical pipeline's loops. We measured that this optimization gave a 13% reduction in average message size.

Relation-grain synchronization If the physical operator calls for relation-grain synchronization, RADISH generates a scheduling dependence from the producer to the consumer. Such a dependence occurs in `HashTableJoin`, where the build must complete fully before the probe starts. RADISH generates producer-consumer synchronization between the two pipeline tasks to indicate such a dependence (as in line 19 and 20 of Figure 4.1(iii)).

For deep plans, `HashTableJoin` limits the available concurrency by relying on relation-grain synchronization; the probe operation is solely dependent on the build pipeline, but we delay the *pipeline* containing the probe until the build side is complete. In contrast, the fully streaming `SymmetricHashTableJoin` provides more tasks to the runtime, but it incurs the higher cost of tuple-grain synchronization.

Tuple-grain synchronization Data structure operations with tuple-grain synchronization have an interface like conventional concurrent data structures. These are indicated by atomic updates (lines 9, 16, 26 in Figure 4.1(iii)). Mutable data accessed in an atomic operation must reside in a single partition whenever possible to avoid distributed transactions. This locality is exploited by RADISH's use of `on partition` for `Shuffle`.

We illustrate tuple-grain synchronization using the example of `SymmetricHashTableJoin`, which uses one hash table per input. The input pipelines are concurrent, so to avoid missed updates, the insert-and-lookup operation on the two hash tables must be atomic. Figure 4.4 shows the layout of two hash tables used in symmetric hash join. By identically distributing the arrays `left hash table` and `right hash table` and their adjacency lists, the atomic region can be implemented as a single migration of the probing task to the owner of `Cell[3]` followed by a *local* atomic operation.

Memory management RADISH uses coarse-grained reference counting to delete intermediate data structures that are no longer needed. When a pipeline starts, for each sink data structure S that does not yet exist, RADISH allocates S with a reference count equal to the number of pipelines that will share S. When a pipeline finishes, RADISH decrements the

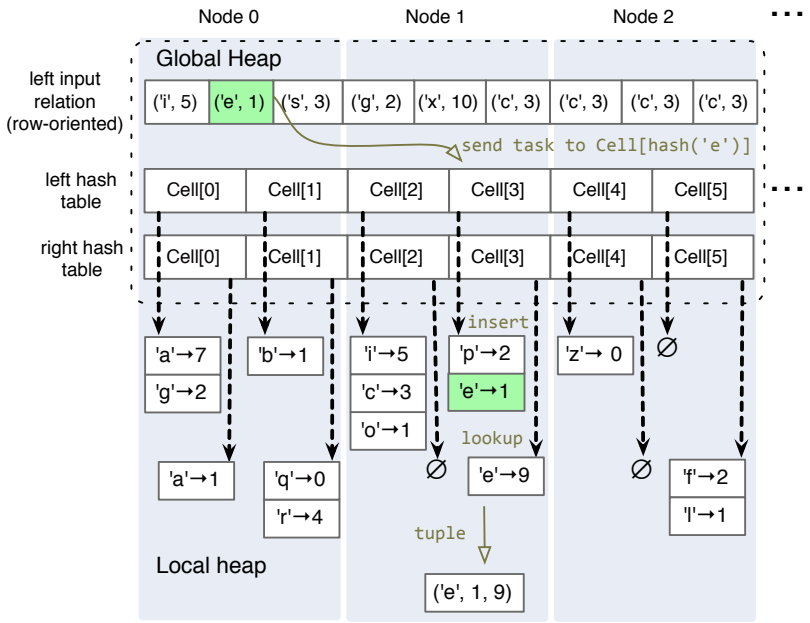


Figure 4.4: Storage layout for two hash tables in a fully-pipelined symmetric hash join. The relations (only one shown) and cells of the hash tables are partitioned across the global heap. The operation of `left_lookup_insert` for tuple $('e', 1)$ is illustrated on the layout. The lookup and insertion together must be atomic to avoid missing or duplicating matches. The two tables are partitioned the same way so that the atomic `left_lookup_insert` can run in a single partition.

count on all data structures it touches.

4.4 Code generation

RADISH extends the algorithm used by the HyPer compiler [116] for translating query plans into pipelined code. In this section, we also describe other execution strategies for RADISH.

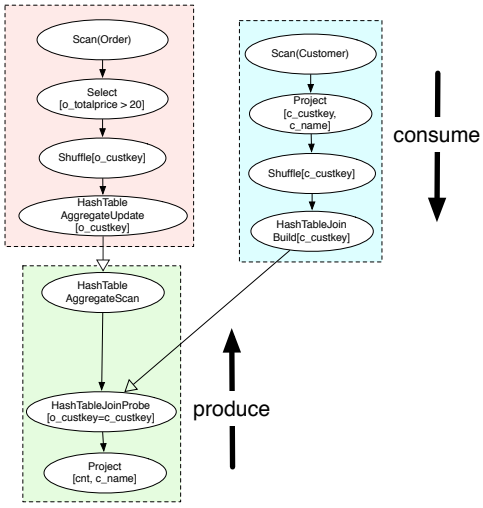
4.4.1 Pipeline generation algorithm

In the code generator design, every physical operator has a two-sided iterator interface (`produce` and `consume`), and the operators generate code that *pushes* tuples through transformations: code for a consuming operator is inlined into the code for the producing operator. Code generation begins at the single sink operator of the plan. Top-down traversal proceeds by calling `produce` on predecessors, and at sources, bottom-up traversal proceeds by calling `consume` on successors (see Figure 4.5). Compiler `state` includes pipeline-global properties and unresolved symbols, and `input_tuple` is an abstract tuple provided by the upstream operator.

The physical plan is broken into pipelines by the behavior of the `produce` and `consume` implementations. For source operators (e.g., `HashTableAggregateScan`), `produce` generates code to iterate over full relations. For pipelined operators (e.g., `Select` and `Project`) `consume` generates code to process and send the input tuple to the next operator. For sinks, or "pipeline breakers" (e.g., `HashTableAggregateUpdate` and `HashTableJoinBuild`), `consume` generates code to materialize the input tuple. Crucially, this code generator design is modular with respect to the operators: each operator implementation solely generates code to produce and/or consume tuples; it is not necessary to implement fused operators (e.g., join followed by project fused into a projecting join) unless the desired algorithm for a fused operator significantly differs from that of the composition of the operators.

4.4.2 Tuple accesses

During code generation, abstract tuples are passed between operators through `consume`. Abstract tuples that appear in the expressions of operators (e.g., the predicate of `Select`)




```
1  interface Operator {
2    // emit code to produce a tuple
3    // or relation
4    produce(state) : unit
5
6    // return code to consume a tuple
7    // or relation
8    consume(input_tuple, state) : string
9  }
```

Figure 4.5: Iterator-like interface for code-generating operators. Traversal of the query plan occurs through calls to each operator's `produce` and `consume` methods. These methods generate code for producing and consuming tuples or relations.

```
1  // returns code for reading the attribute at the position
2  getattr(position) : string
3
4  // returns code for writing the attribute at the position
5  setattr(position, compiled_val): string
6
7  // returns code for concatenating two tuples
8  create(tuple1, tuple2) : string
```

Figure 4.6: Interface for abstract tuples

are translated to code using the interface shown in Figure 4.6.This interface encapsulates how tuples are accessed, making operator implementations decoupled from how tuples are stored (e.g., in array of structs or struct of arrays). The abstraction only exists in the compiler, so it does not create runtime overhead.

4.4.3 Other implementation strategies

We implemented a baseline execution strategy, *Compiled Volcano iterators* (CVI), in Radish using compiled iterators. We also discuss how to implement column-oriented processing in

RADISH with few changes.

Compiled Volcano Iterators (CVI) We implemented a version of RADISH that uses demand-driven dataflow (see Section 3.1). In CVI, each operator is implemented as a sequential iterator with a `nextTuple` method. Intra-pipeline parallelism is achieved by having multiple instances of each iterator. Communication within a pipeline is achieved by `Shuffle` sinks and sources, which push tuples to partitions and materialize them in a local data structure. One task on each instance polls the `nextTuple` method of the sink operator until it returns `false`.

To isolate the benefits of the CPP technique from the known benefits of compilation over interpretation, CVI shares much of the same library code and is compiled. CVI uses the same messaging and data structure code as CPP, but its operators are written generically using the iterator interface. The `nextTuple` method of individual operators are compiled, including arithmetic expressions, predicates, concatenation of tuples, and access of attributes. This design point is listed as RADISH-CVIin the diagram of compilation scope, Figure 3.1.

Column-oriented execution As described in Section 3.1.4, column-oriented execution is often faster than row-oriented on analytics. The benefits are attributable to I/O savings for unused columns, compression, late-materialization (lazy copying of attributes), and block iteration [1]. I briefly discuss these factors with respect to row-oriented RADISH. For unused columns: row-oriented RADISH only wastes memory bandwidth for internal fragmentation of cache lines because the query planner pushes `Project`s to be early in the plan. Late-materialization must be weighed against the possibility of additional network messages. Benefits from block iteration (avoiding a function call per tuple to extract an attribute) are less important because tuple extraction overheads are compiled away even in the row-based engine.

RADISH is compatible with a column-oriented format for tables. Given that RADISH is solely in-memory, implementing column-oriented code generation requires relatively few

changes, although extensive support for column oriented algorithms would require greater effort. The compiler's interface for an abstract tuple includes methods `getattr`, `create`, and `setattr`. For column-oriented execution, these methods are implemented to emit code for array access rather than struct field access. Data structures can also be reused unmodified: instead of storing tuples directly, the column-oriented hash table stores only row indexes, which are used to access the attribute arrays. Adding column-based distributed algorithms will involve reasoning about locality and data movement (e.g., when the match should be materialized and where) in either RADISH or the query planner.

4.5 Runtime issues

So far, we have discussed the data structures and generated code for queries implemented in PGAS languages, but there are aspects of the language runtimes that are important for performance. In this section, we discuss issues in concrete PGAS languages and runtimes Grappa (Chapter 2), Chapel, and X10.

4.5.1 Parallel loops and tasks

RADISH processes tuples with tasks, but rather than using an explicit execution strategy (such as worker threads taking chunks of a table at a time [89] for sequential processing), RADISH uses `forall`. This construct corresponds to the data parallel loops `Grappa::forall`, Chapel `forall`, and X10 `ateach`. These data parallel loops are primarily implemented with tasks that are consumed by a fixed number of threads. We discuss two issues regarding task-oriented execution: concurrency management and loop termination.

Concurrency management An important factor for performance is that the workload offers just enough concurrency that it utilizes all machine resources without wasting resources. This concern inspires a careful approach to spawning new tasks and scheduling existing tasks. RADISH relies on recursive decomposition, as described in Section 2.4.2, to limit the number of active tasks be proportional to the number of available processors.

Task scheduling has an effect on efficient utilization of the memory hierarchy. Section 4.3.2 discussed how RADISH generates data-centric code for each pipeline by pushing individual tuples. Maintaining data-centricity for pipelines that *include communication* additionally involves task scheduling. Specifically, `on partition` sends a running task to a remote partition, but that remote partition will most likely be running other tasks. The best that the task scheduler can do is try to keep the working set as small as possible.

To keep a small working set, we rely on two aspects of smart task scheduling. First, recursive decomposition limits the spawning of *new* tasks. Second, for *existing* tasks, the task scheduler uses following heuristic priority order: 1) started tasks with local origin, 2) started tasks with remote origin, and 3) local tasks that are not yet started. This order ensures that tasks using resources relinquish them sooner and that data related to started tasks is touched again before new data is touched.

Skew Because tasks are tied to the data in most distributed analytics systems, skew in the data will directly cause skew in the execution—some partitions will have more work to do than others. The typical solutions to skewed data focus on better partitioning of the data [3, 86, 58]. Unlike these systems, RADISH can additionally employ *task*-oriented skew mitigation. Using `forall`, in addition to the valid-anywhere nature of addresses in PGAS, allows RADISH to take advantage of dynamic task balancing mechanisms provided by the runtime system, at a measured cost to locality.

Loop termination The algorithms we use for RADISH have the async-finish property, which is defined in Section 2.4.2. This property ensures that the nested `forall`s common RADISH programs are synchronized efficiently despite creating many tasks. Synchronization for nested `forall`s that are async-finish is available using GRAPPA's `GlobalCompletionEvent` and X10's `async` and `finish`.

4.5.2 Grappa and fine-grained tasks

Translating a query to parallel tuple-at-a-time code produces tasks and remote messages (`on partition`) at the granularity of a tuple. This approach makes low-level details, such as block-based communication, orthogonal to the computation specified by the intermediate parallel program. As a result, the PGAS compiler and runtime have more flexibility in how they implement loops and communication. On the other hand, fine granularity tasks and remote calls induce overheads on the network and CPU. In this section we explain how we mitigate these overheads by building upon Grappa.

Network overhead Section 2.2 discussed how data-intensive programs often generate small messages that do not utilize network bandwidth. To avoid this problem, modern parallel databases operators often pull data in chunks over the network [42, 160] after sorting them locally. Radish must also cope with the small message problem; we measured that the average message size in TPC-H queries was 40-200 bytes, 2-3 orders of magnitude too small to utilize network bandwidth. Fortunately, Radish is able to use Grappa's automatic message batching (Section 2.4.3) to achieve high bandwidth.

CPU overhead Assigning a task for every tuple in a pipeline incurs overhead in the CPU for spawning, scheduling, and context switching. Grappa's lightweight task system (Section 2.4.2) makes it possible to express the computation as task-per-tuple, while keeping overhead low. The 50-ns context switch time of Grappa tasks is critical to performance, and assigning chunks of loop iterations to each task (performed by `forall`'s recursive decomposition) further reduces scheduling overhead. In the SP^{2}Bench queries, setting this chunk size appropriately improved performance by up to 23%.

4.6 Evaluation methodology

We built Radish as an open-source extension to Raco, a relational algebra$^+$ compiler and optimization framework. We built a back end to Radish that emits Grappa code. In

the evaluation we refer to Grappa programs generated by Radish as RadishX. RadishX includes a variety of hash-based algorithms for joins and aggregations.

Our primary goal was to implement both CPP and the more conventional design, CVI, in RadishX and compare performance. Our secondary goal was to determine whether RadishX is a practical system for analytics by comparing its performance with a competitive system: the parallel database Impala. Impala is a state-of-the-art commercial database system that features parallel execution of individual queries over a cluster and code generation. These features make it one of the most similar systems to compare to RadishX. In all performance comparison plots, the error bars represent 95% confidence intervals.

4.6.1 Setup

Query systems We linked Grappa code emitted by both Radish-CPP and Radish-CVI against the MPI implementation MVAPICH2 v1.9b [107].

For each trial using Impala, we ran the query twice and recorded the second result. Full table and column statistics were collected for every input table to ensure good query plans. All input tables use random partitioning, as does RadishX.

Hardware We ran TPC-H experiments on a cluster of 16 AMD Opteron processors connected by QDR Infiniband. Each node has 16 3.1-GHz cores across two sockets and 128GB of memory. We ran SP^{2}Bench experiments on a cluster of AMD Interlagos processors connected by QDR Infiniband. Each node has 32 2.1-GHz cores across two sockets and 64GB of memory.

4.6.2 Benchmark queries

TPC-H We assume the conventional ad hoc TPC-H scenario with no tuning of the physical storage; RadishX uses full scans to implement selections, and all joins—including primary-foreign key joins—and aggregates require partitioning at query time. We used scale factor 10 for comparing CPP with CVI and scale factor 100 for comparing to Impala. We fix the

TPC-H validation parameters [144] as the substitution parameters of the queries. The code for the RadishX experiments is available online,[4] and the code for the Impala experiments is available online as a fork of an open source implementation of TPC-H.[5]

Graph data benchmark We evaluated scaling of RadishX using the benchmark SP^{2}Bench [133] that uses synthetic dblp data stored as RDF. This benchmark is useful because it has challenging joins resembling graph traversals. We used the native triples of the data, with one relation of three attributes. We did not perform the indexing and co-partitioning that is common in the loading step in RDF databases. To limit the number of operators we had to implement to evaluate RadishX we removed `DISTINCT` from the queries.[6] We used SP^{2}Bench to generate a dataset of 100 million triples (10 GB).

4.7 Results

4.7.1 CPP *vs.* CVI *performance*

Table 4.2 shows the runtime of CPP (with two different plans) and CVI. CPP-AsymJoin uses the best query plan available, involving asymmetric hash joins (buid/probe as in Figure 4.1 ii). However, CVI only supports symmetric hash join. The difference in performance between the two join algorithms is notable when the join includes large `lineitem` table because it must be materialized in the symmetric hash join, whereas it is not materialized in the asymmetric hash join. To support asymmetric hash join in CVI would have required significant changes to the communication layer of RadishX to support queuing, which would make it less comparable to CPP. Instead, to make the comparison apples-to-apples, we also measured the results for CPP using the symmetric hash join (CPP-SymJoin). Figure 4.7 shows the speedup of CPP-SymJoin relative to CVI. In many cases, this plan is nearly

[4]https://github.com/uwescience/tpch-myrial

[5]https://github.com/bmyerz/tpc-h-impala

[6]When we ran the graph data benchmark experiments, Radish did not support `DISTINCT`, but it has since been implemented.

as fast, and when there is a difference, CPP-SymJoin is still faster than CVI. This result demonstrates that there is a performance advantage of CPP due to the code generation strategy. On average, CPP with the identical plan is 2.4×faster than CVI.

For most queries, an important factor in the speedup over CVI is due to the code from CPP being more amenable to existing compiler optimizations. To measure this effect, we formulated another experiment. We ran two versions of each CPP-SymJoin and CVI: one with no compiler optimizations turned on (`-O0` in gcc) and one with all optimizations turned on (`-O3` in gcc). It is important to note that Grappa does not use a true parallel compiler, so the set of optimizations is more limited than that of other PGAS languages. The results of the experiment are shown in Figure 4.8.[7] The compiler was able to improve CPP programs more than CVI programs: the geomean speedups are 3.2×and 2.0×, respectively. We examined the absolute running times of both systems to validate that in most cases CPP was not simply slower than CVI with `-O0`. Our experiment is similar to one in the original work on holistic (sequential) code generation for queries [83]. However, their results (Table 3.3 in that document) supported only that the generated code was *able* to be further optimized by the C compiler and not that the generated code was optimized *better*.

The remaining speedup was mostly the result of fewer instructions executed, due to more efficient code. Other sources of overhead did not explain all of the speedup. Various differences in the code CVI uses to access data structures accounted for a fraction of the slowdown in only a few queries. The systems exhibit insignificant differences in the number of data cache misses, instruction cache misses, and CPU stalls, probably because they have the same runtime system. An outlying result is that *Q19* exhibits significant speedup not attributable to compiler optimizations. The slowdown of CVI on that query is due to the fact that CVI's symmetric hash join is not streaming, while CPP's is. We did not build a streaming symmetric hash join for CVI because it would require a queueing system to convert from push to pull or pull-based communication, but RADISH uses push-based communication

[7]Queries 5,8,9,10,11 were not immediately able to run in `-O0` for CPP-SymJoin, and the existing results support the conclusion that CPP-SymJoin was optimized better than CVI.

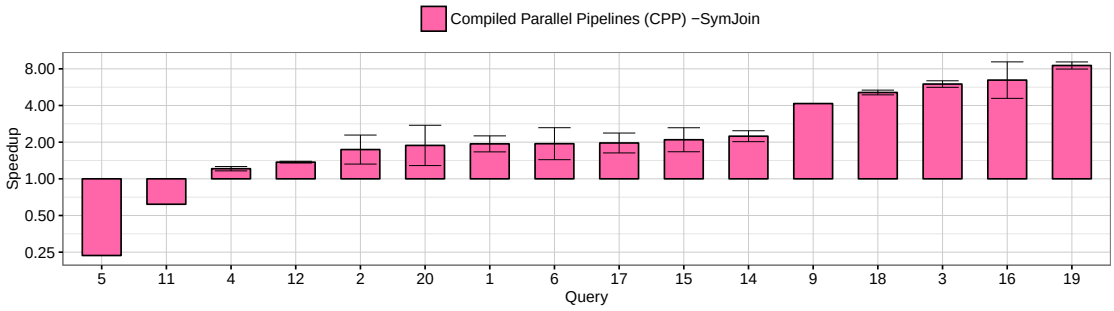


Figure 4.7: Speedup of RadishX using CPP over CVI on TPC-H queries

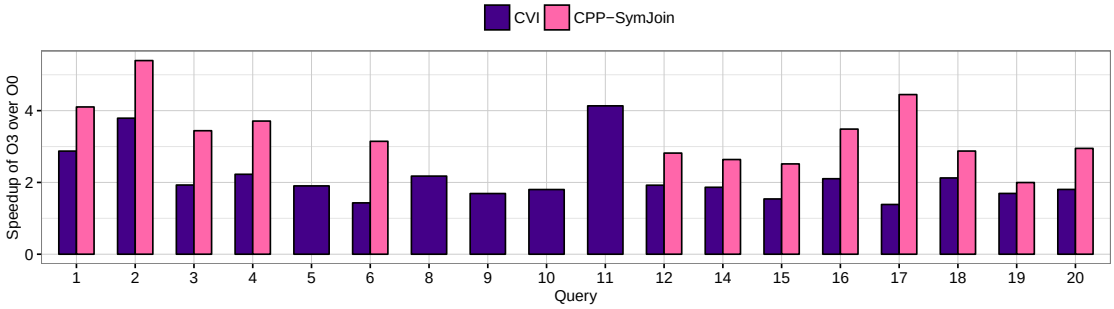


Figure 4.8: Speedup of optimization level `-O3` over `-O0` on CPPSymJoin and CVI on the TPC-H queries. The result supports that code generated by CPP is more amenable to compiler optimizations than code generated by CVI.

that relies on Grappa's batching message system. Such a large difference in the message system would make the two execution strategies hard to compare.

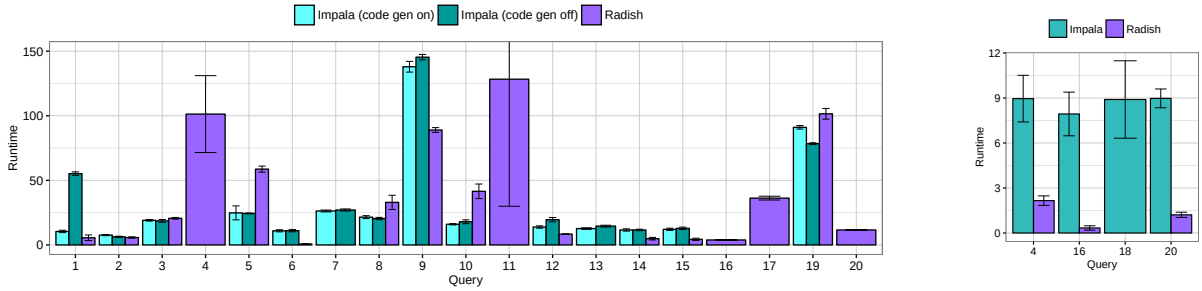
4.7.2 Contextual performance

We compared running times for TPC-H queries between RadishX and Impala to validate that our implementation was competitive with a commercial query processor. The running times for RadishX and Impala on TPC-H are shown in Figure 4.9a. Running times for sf10 for queries we could not get to run on Impala at sf100 are shown in Figure 4.9b. In the 15

Table 4.2: Running times on TPC-H queries for different versions of RADISH. Running times are in seconds. CPP-SymJoin and CVI use the same plan, composed of symmetric hash joins. CPP-AsymJoin uses an asymmetric hash join that only materializes one side.

query	CPP_AsymJoin	CPP_SymJoin	CVI
1	0.55	0.44	0.91
2	0.56	0.69	1.21
3	1.08	1.80	8.61
4	2.85	4.24	5.00
5	3.40	282.92	70.43
6	0.06	0.08	0.17
9	5.78	11.74	38.50
11	3.85	2.77	1.74
12	0.58	1.22	1.76
14	0.40	0.39	1.02
15	0.35	0.36	0.72
16	0.24	0.42	2.39
17	2.69	2.99	6.28
18	7.60	15.07	67.60
19	3.47	7.92	69.22
20	1.05	1.41	2.39

queries that we could run on both systems, RADISHX is on-par-or-faster on all but 3 queries. Even though RACO's aggregate decomposition optimization (creates partial aggregates to reduce communication when there are few enough unique keys) improves RADISHX on *Q12* and *Q14* and significantly on *Q15* and *Q16*, without it RADISHX is still faster than Impala. In all queries, Impala's plan was at least as efficient as the one used by RADISHX. In some cases RADISHX may be using a less efficient plan, but this is acceptable because the purpose of this experiment is simply to benchmark RADISHX's implementation of query execution. We ran Impala with its LLVM-based runtime code generation [149] toggled. We found that *Q1* showed significant speedup but there was no discernible benefit for any other queries. The single case of speedup due to codegen, *Q1*, is unsurprising because this query admits embarrassingly parallel, arithmetic-intensive code.



(a) (b)

Figure 4.9: Performance of RadishX and Impala on TPC-H queries, 16 nodes. (a) sf100, (b) sf10 for queries that would not run on Impala at sf100. The running time of queries on the two systems is comparable, usually within 2×, demonstrating that the RadishX implementation is realistic.

There are some data points missing from the plots. RadishX ran indefinitely on *Q7* and crashed on *Q18* due to bugs in the runtime, and Impala crashed on *Q17*. We did not implement *Q13* or *Q22* in RadishX as Raco did not yet support nulls (although there are workarounds). Impala did not yet support the cross products necessary in *Q11* and *Q22*.

4.7.3 Weak scaling

Because RadishX is a distributed system, we evaluated weak scaling. We used SP^{2}Bench with 1.4M rows per node. Results in Figure 4.14 show that *Q2* and *Q5a* both scale well (flat), even though for *Q2* the intermediate result size to input size ratio increases with input size. RadishX does not scale as well on *Q9*, which also has an increasing ratio of result to input size. This poor scaling is not due to small input size: running the full input size on 64 nodes gave a speedup of 4.9× over 8 nodes. The quadratic *Q4* scales well; although intermediate result size to input size ration increases with small input size, its author-author join degree is ~ 33 for the 50M-row dataset. For *Q4* we tried other plans: symmetric hash join (SYM) and bushy join shape (bushy). Bushy is better for this query because it reduces the intermediate result size by 3×. SYM scales better than HJ in the bushy join shape. We

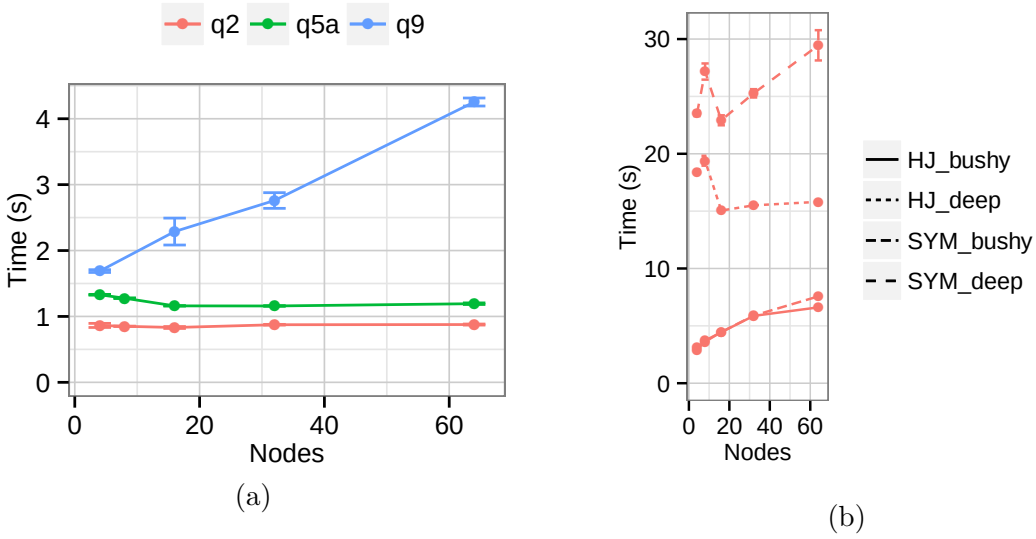


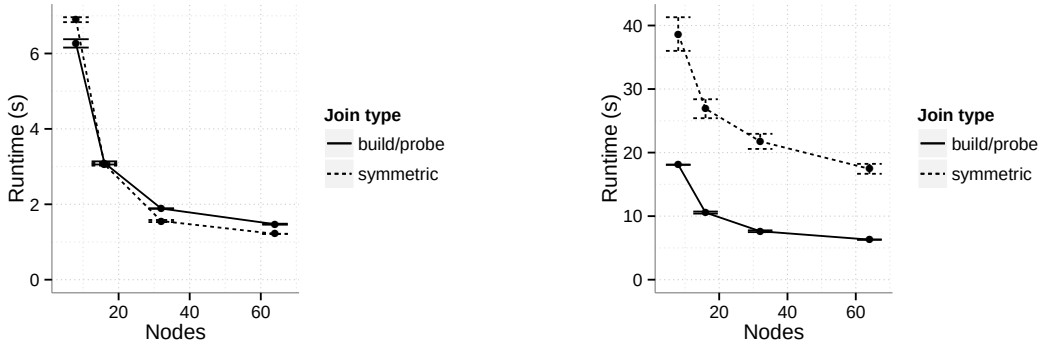
Figure 4.10: Weak scaling on SP^{2}Bench queries. a) Weak scaling on queries with medium join result size, b) weak scaling on *Q4*, the join query with an average degree of 33.

found that on the other SP^{2}Bench queries, an all-`HJ` plan was consistently faster (up to 2× faster) than the equivalent all-`SYM` plan. Although the fully pipelined `SYM` plan provides more concurrency by using only tuple-grain synchronization, it must materialize both sides of the join into hash tables, as discussed in Section 4.7.1.

4.7.4 Scaling of different join algorithms

The fully pipelined symmetric hash join allows for greater concurrency than the asymmetric hash join. However, a secondary outcome of the experiment in Section 4.7.1 was that symmetric hash join was usually slower; greater concurrency did not compensate for the overhead of materializing both inputs to the join. We ran strong scaling experiments to see if there are cases when more concurrency is profitable. Specifically, we ran the SP^{2}Bench queries on increasing numbers of machines, with a fixed size of data.

Figure 4.11 shows the results for two representative queries. In *Q5*, the extra parallelism of a fully-pipelined symmetric hash join plan provides a small benefit when more processors are



(a) Query Q5a

(b) Query *Q9*

Figure 4.11: Strong scaling of join algorithms for left-deep plans of SP^{2}Bench queries. Despite presenting more concurrency to the system, plans with symmetric hash join are at best marginally faster than plans using asymmetric hash join.

available. The worst result is *Q9*, where the fully-pipelined plan causes extra materialization.

4.7.5 Compilation time

Although we did not spend effort on minimizing compilation time, we include a discussion of it for completeness (Figure 4.12). For the TPC-H queries, Raco and Radish take, on average, 1.26s to parse the query, optimize the plan, and generate the RadishX program. Compilation of the RadishX program with optimization level `-O3` takes 10s of seconds. This significant time is partly attributable to extensive inlining of C++ template header files for the Grappa runtime and could be reduced with engineering effort.

4.7.6 Comparison to handwritten code

Because Radish generates a PGAS program that is similar to what a programmer would write, we used it to explore **RQ3**, "how close can the performance of declarative programs implemented in high performance systems come to handwritten programs?" We explored this question with a case study on the PageRank algorithm. For the handwritten version,

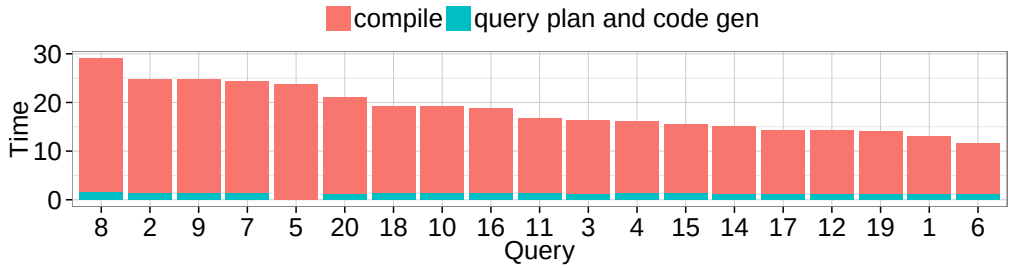


Figure 4.12: Total compilation time for TPC-H queries. The current implementation incurs a high cost for compiling the generated code (roughly proportional to the size of the query plan). With compilation in the 10's of seconds, RadishX is useful in analytical querying where that cost of compilation can be amortized.

we used our PageRank implementation from Section 2.5.2. We used Radish to generate a program from MyriaL[8] code (listings in Section A.1). We ran both versions on 16 nodes, using the 7.6M-edge Berkeley-Stanford web graph [91].

Table 4.3 shows the results. First, picking the right join order between the source vertex relation, edge relation, and destination vertex relation changed the running time of iterations by $\sim 29\times$ due to the joined relation being properly partitioned after the join as needed by the aggregate. Second, from the mean iteration times we see that incremental PageRank updates according to active vertices is important for the performance of later iterations. We attempted to emulate this behavior by modifying the query to compute the new PageRank based on the old PageRank, using an outer join. The result, in column "Radish active vertices", was that the runtime was not improved. This lack of improvement was caused by the overheads of the more complicated query. Finally, the Radish code was $7.5\times$ slower than the handwritten code. This difference is primarily due to three related factors, in order of expected importance: combining, graph mutation, and loop invariance.

[8]MyriaL is a query language that I describe as "SQL with statements and while loops". Documentation can be found at http://myria.cs.washington.edu/docs/myrial.html.

Table 4.3: Running time for PageRank on handwritten versus Radish-generated code.

	Handwritten	Radish bad join order	Radish good join order	Radish w/o group by opt	Radish active vertices
Max iter (s)	0.08	19.21	0.67	0.92	0.72
Mean iter (s)	0.01	17.88	0.60	0.92	0.72

Combining The RADISH code does not incorporate local combining that is important for reducing communication. We would expect RACO's aggregate decomposition to have a large effect on performance (difference between "good join order" and "w/o group by opt") by communicating at most $V * Partitions$ PageRank contributions instead of $V * Degree$. However, RACO did not fully realize the partial PageRanks optimization because it does not have the required rewrite rules to find the plan that does a local join, then partial aggregate, then the global join and aggregate.[9] Instead it computes the entire distributed join before doing any aggregation.

Graph mutation The RADISH code creates a new graph, while the handwritten code mutates the graph in-place. To have in-place updates, we could borrow techniques from view maintenance. By expressing the graph in our query as a view, we could update it incrementally [57]. In general, this may not eliminate copying by the execution engine, but we can reduce the number of them.

Loop invariant code The RADISH code recomputes the join between the source vertex relation and edge relation during each iteration. Better loop-invariant code motion optimizations in RACO might eliminate some re-computation.

[9]Raco feature issue https://github.com/uwescience/raco/issues/526

4.8 Performance comparison to a dataflow engine

RADISH is similar in capabilities to dataflow engines like Spark that provide query processing within a general purpose language. We present further comparison of the performance of GRAPPA and Spark Section 2.5. For Spark, we ran queries using two native query systems built on it called Shark and SparkSQL.

4.8.1 Simple query

To understand the foundational differences in the systems, we took sample-based profiles of both systems for a microbenchmark query. The query, *SEL1*, is just a filter with a selectivity of 1%. We categorized CPU time from the profiles into: *network*, *serialization* (message de-aggregation in RADISHX, object serialization in Shark), *iteration* (loop decomposition and iterator calls), *application* (actual user-level directives), and *other*. Figure 4.13a shows that Shark spends two-thirds of the time in network code: given that *SEL1* requires no communication, this may indicate idle time (appears in profile as polling the network) that is due to poor load balance and task spawning. Even so, significant slowdown in Shark is due to other factors: the inner loop suffers deserialization and iteration overheads. The serialization overheads come from Java's non-native object storage in memory. RADISHX had negligible impact from application code, while Shark spent 5% of its time there (filter operator).

We ran the same analysis for SparkSQL, the recent replacement for Shark.[10] SparkSQL stores tuples natively in Spark's in-memory RDD format, unlike Shark. SparkSQL still executed *SEL1* with a similar profile to that of Shark, although the time goes to different serialization procedures. While Shark spent a lot of time in object serializations and transforming between column format, most serialization time in SparkSQL consisted of `productToRowRDD`, which transforms the format of tuples.

[10]At the time of experimentation, SparkSQL was in alpha and did not yet support enough of SQL to implement all of SP^{2}Bench

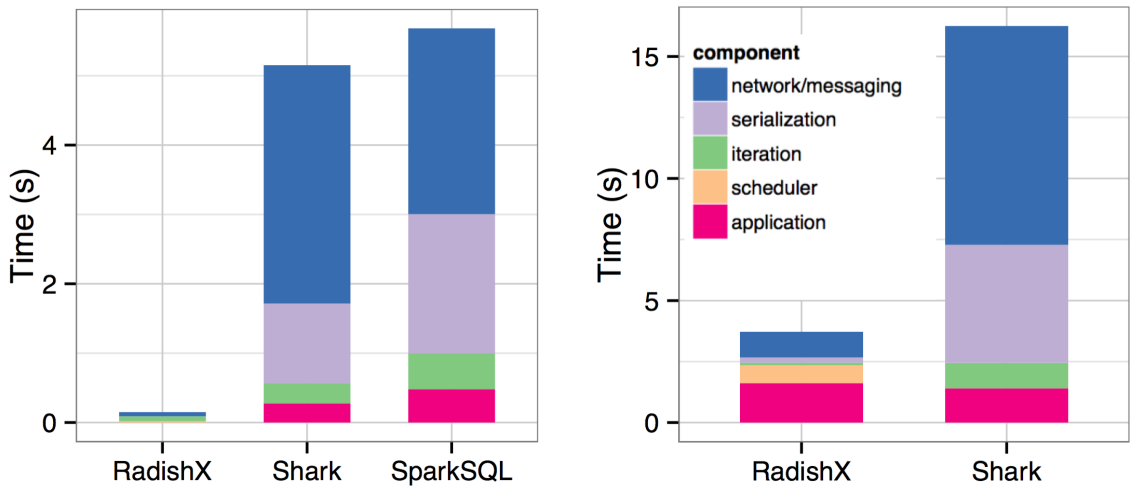


Figure 4.13: Time breakdown for Spark and RadishX on two queries. a) *SEL1*, 20GB: Both Shark and SparkSQL have significant serialization overhead. Compared to RadishX, Shark's application code (the filtering operation) is non-negligible and iteration overhead is 4.3× greater. Percentages for RadishX are 40.2% network, 47.0% iteration, 12.8% scheduler. b) SP^{2}Bench *Q2*: On communication-heavy workloads, application time is similar, but Shark's has high serialization and networking cost.

4.8.2 Graph data benchmark

Figure 4.14 compares the performance of RadishX and Shark on SP^{2}Bench. The results are grouped roughly by category: *Q3b*,*Q3c*, *Q1*, *Q3a* are select-join and are selective so join input is small; *Q9*, *Q5a*, *Q5b*, and *Q2* are select-join and join input is large; *Q9* includes a union; and *Q4* allows for indexes to be used more than once and has a large average join degree of 33.

On the highly selective queries *Q3b* and *Q3c*, the comparative runtime followed the observation for *SEL1*. As the number of inputs to the join increasem, communication dominates, and the performance gap decreases to a different level, as illustrated by *Q1* and *Q3a* and the four queries with large join input.

We present a time breakdown for the communication-heavy *Q2*, as well (Figure 4.13b).

The systems spend nearly the same amount of CPU time in application computation. About half of RadishX's performance advantage comes from efficient message batching and HPC network stack.

Additional benefit comes from iterating via RadishX's compiled parallel `forall` loops compared to Shark's Radish-CVI-like iterators. RadishX's scheduling time is higher than Shark; its general purpose task system does frequent context switches. However, RadishX makes up for scheduling overhead in every other category. Shark spent negligible time in compression (included in serialization) in *SEL1* and 9% in *Q2*. This is from shuffle hash joins of *Q2* storing intermediate results in memory rather than disk. RadishX spends 22× less time in serialization and 10× less time in iteration; if only these two components could be accelerated in Shark, the total speedup would be 1.5×.

Shark's execution of these queries appears to place bursty demands on the network, and is sensitive to network bandwidth, while we find RadishX to be more consistent. On query *Q2*, Shark achieves the same *peak* bandwidth as it can sustain in the GUPS random access benchmark (200MB/s/node from Figure 2.13c), but its sustained bandwidth in the query workload is just over half this amount (116 MB/s/node).

The Shark query planner chooses to do a sequence of joins, which are performed using shuffle. Although the map outputs are materialized in memory, these steps are very expensive.

4.8.3 The performance of Spark

The same inefficiencies in Spark that I have pointed out in this section are by now well-known to Databricks developers, and there is an ongoing effort to improve the performance. The JVM object model causes significant overhead in data-intensive applications. They have eliminated some of this overhead using the non-portable `sun.misc.Unsafe` library for explicit memory management and native data types [157]. More recently, developers expanded Spark's code generation: from individual expressions to sequential fragments of query plans [5]. These efforts demonstrate that high performance is desired even in big data

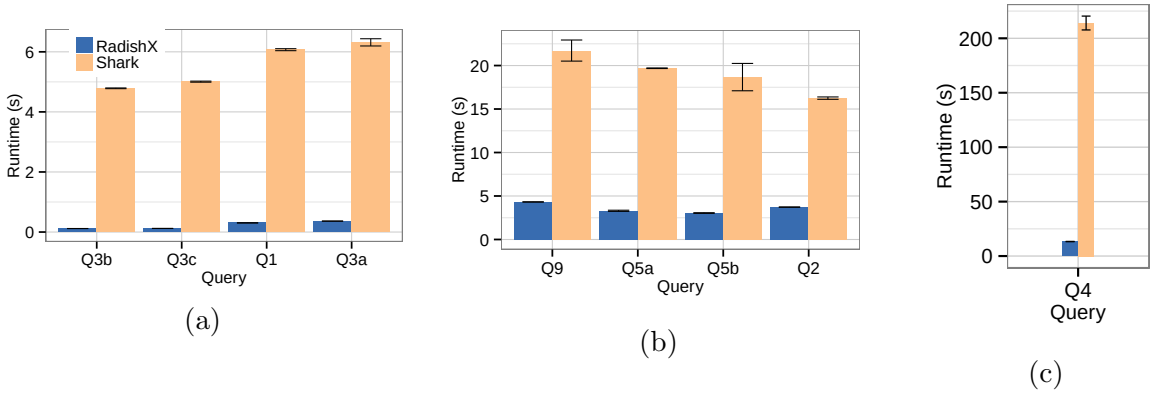


Figure 4.14: Runtime of RadishX and Shark on SP^{2}Bench queries. a) queries with small input to the join, on 16 nodes, b) queries with large input to the join, on 16 nodes, (c) *Q4* with worst-case quadratic intermediate result size, on 64 nodes

systems and that the techniques developed for fast in-memory query processing in RDMSs are being applied more broadly.

4.9 Using Radish

We discuss how Radish should be used and how PGAS compilers might optimize the generated code.

4.9.1 Using RADISH

As with other MPI-based HPC systems and most MPP databases, Radish does not support recovery of failed tasks. Therefore, it is not a replacement for systems like SparkSQL [15] that can run data-parallel jobs on large clusters with less frequent full checkpointing. However, recent efforts have argued that most clusters in practice are 20-50 machines, a scale at which fault tolerance is not a dominant concern [42, 130].

Compilation time for RadishX is currently quite high (Section 4.7.5). Sub-second query compilers [116] gain efficiency by generating only "glue code" and pre-compiling most of the

operator implementations. RadishX is applicable for repetitive queries (like classifiers, Section A.2) or those where the compilation time is amortized by the performance improvement.

We expect Radish to be effective as the back end to language-integrated queries in analytics pipelines that include parallel *table-valued functions* (TVFs) or whole stages written in PGAS code that cannot be run efficiently in dataflow systems; for example, array computation, irregular applications [16, 125], and algorithms with frequently updated shared state like parameter servers [6]. All the user may need is a statistic or summarization over a large output after a simulation. The developer could divide the HPC and dataflow processing into two different systems. However, this division would incur the costs of transforming and moving raw data across system boundaries [111] and prevent co-optimization of the two programs.

4.9.2 PGAS compiler optimizations

When there are UDFs, PGAS optimizations could provide optimizations inaccessible to a query planner. By inlining a UDF into the generated code, the PGAS compiler can send only live fields instead of the whole tuple during `on partition(...)` by using *scalar replacement* [20, 63]. GraphX achieves a similar projection by bytecode introspection [57], rather than inlining.

Radish always benefits from accessing a global hash table using `on partition`, but it is not obvious where to run the rest of the pipeline after the match. A similar decision must be made for hash joins in columns stores and MPPs. Alembic [68] does such an optimization on general PGAS code by inspecting data layout to infer good task migrations and would be complementary because it works at the granularity of memory accesses and works in the presence of UDFs.

4.10 Discussion

In **RQ2**, I asked "what techniques can improve the throughput of database-like query processing in a high performance parallel system?" In this chapter I described the design

and evaluation of an in-memory query processor built with PGAS languages. Our particular approach was a compiler that generates code that takes advantage of the locality and parallel-awareness of PGAS languages. Our code generation technique executed queries 2.4× faster than conventional compiled execution. Using PGAS to efficiently execute queries also required efficient data structures, generating code that avoids extra messages, and mitigating the overhead of an execution model based on fine-grained tasks. For the last item, Grappa was uniquely useful. I also found that while Radish-generated code is still a factor of 7.5 off of handwritten PGAS programs, improvements in Radish's parallel query optimizer would bring parity within reach.

Radish is the first step toward parallel language-integrated queries, which will be useful for writing high-performance analytics codes. I also posit that targeting parallel languages as an intermediate representation for queries is a valuable approach for performance and simpler design of query engines for distributed systems.

Such a system, where the query is ultimately implemented in a high performance language, has the potential to enable direct querying of data produced by a high performance program.

Chapter 5
SOFTWARE ENGINEERING OF QUERY COMPILERS

This chapter does not address a primary research question of this thesis, but it does discuss important issues in query compilers. Both existing and new commercial systems [164, 149, 5] are adopting query compilation. Common wisdom used to be that code generators for database operators were hard to maintain [128]. While I make no scientific claim that recent techniques have actually made query compilers easier to maintain, I posit that they have. I discuss approaches to building query compilers in the context of software engineering, as well as my effort to build a bug-free query compiler.

5.1 Software engineering of code generation

Recent techniques for query compilation depend on implementing database primitives using generative programming. We describe how query compilers should be built to have maintainability and extensibility.

5.1.1 Template-based

Template-based code generation is the most common generative technique used in current query compilers. In this technique, database operators are implemented in two parts: 1) templates with holes and 2) logic to fill the holes and compose the templates. HIQUE [84] has a monolithic code generation algorithm specific to relational database operators and lacks a common interface for implementing operators; thus, the authors remarked that HIQUE was hard to extend and maintain. HyPer [116] improves the software engineering of template-based code generation for queries by defining an intuitive interface for operators: `produce` and `consume`. Unlike holistic code generation, this interface provides modularity so that

different operators can be composed into a single pipeline. RADISH extends HyPer's approach (Section 4.3). Other query compilers generate code with similar algorithms. Nagel et al. creates a tree of operators [111]. Steno [106] translates queries to an intermediate language of six extensible operators, and it compiles the IL with a finite-state machine.

5.1.2 Multi-stage programming

LegoBase [82] departs from template-based code generation with a more powerful generative programming paradigm (or extensible compiler) called lightweight modular staging (LMS), built upon the Scala programming language. In existing template-based query compilers, there is no IR between the query plan and the final emitted code and so analysis and transformations can occur only on the query plan. LMS provides a framework for multi-stage compilation using the type system: expressions that have normal Scala types are computed in the current stage and expressions of a `Rep` type are saved for later. These `Rep` expressions can be pattern matched and transformed.

With this approach, LegoBase operators are written once in the iterator interface and transformed orthogonally into a variety of execution strategies. Two examples of transformations are 1) transforming a pull-based (conventional iterators) engine to a push-based one (thus achieving the effect of the `produce`/`consume` interface using only the conventional `next`) and 2) turning a row-store DBMS into a column store using an array-of-structs to struct-of-arrays transformation. The number of demonstrated examples of transformations is still limited. For example, the developers have not yet demonstrated transforming tuple-at-a-time operators to block-at-a-time. Recent follow-on work to LegoBase by Shaikhha et al. [138] provides a principled design space exploration of multi-level query compilers. They provide evidence that more levels allows for better separation of concerns and enables new kinds of optimizations. Certain abstractions in RADISH embody concepts in LegoBase; for example, the abstract tuple (Section 4.4) would be a *future-stage* expression during query plan optimization and a *present-stage* expression during code generation when `getattr` gets called.

I believe that a complete system for query compilation should use some kind of multi-stage translation to allow for the highest level of productivity and the most opportunity for optimizations. It is attractive to consider re-architecting RADISH using a multi-level approach to generating code. The final step in LegoBase compilation is to take the optimized sequential Scala code that implements the query and lower it to C code. Given that distributed shared memory code resembles sequential code, we might lower the Scala code to PGAS code using the ideas of RADISH. We could either introduce a concept of partitions into LegoBase or just hide partition-awareness inside of RADISH's global-view data structures (Section 4.3.1). For example, a Scala `HashMap` would be lowered to RADISH's global hash table. Shaikhha et al. speculate about how to add parallelism to their multi-level compiler but restrict the discussion to partitioned parallelism rather than RADISH-like parallel programs with shared data structures.

5.1.3 Intermediate bytecode generation

HyPer operators are written in a mix of intermediate compiler bytecode (specifically LLVM) and C++ code [116]. Naturally, bytecode is harder to maintain, so they only write critical processing like tuple access and transformations in bytecode and leave the more complex manipulation of operator data structures to a pre-compiled C++ library. The higher performance of HyPer's generated bytecode relative to generated source code reminds us that ultimately we must redesign the interfaces of our compilation tools to allow for domain-specific optimizations. I think it is still an open question where and how the optimization flow should be split into steps and abstractions. Other projects have also adopted LLVM bytecode as the target language [42, 149].

5.2 Towards a formally verified query compiler

In some domains, there may be a high expectation of correctness for the database, especially when data is protected or changes to the database are persisted [94]. Testing-based techniques can provide high-assurance with the right collection of test cases but cannot

guarantee that the program adheres to its specification. Database systems need lots of tests because they take both programs and data as inputs. An alternative to testing is formal verification. In this approach, the developer writes a precise specification of correctness and an implementation and with the help of a checker, proves that the implementation adheres to the specification. This approach currently involves much more development effort than testing but proves the absence of bugs; more humbly, it removes the implementation from the trusted computing base, leaving only the specification and the checker implementation.

As surveyed in Chapter 3, high-performance query engines rely on code generation. Whether the entire database system is verified or tested, formally verifying the code generator would provide value. There is some evidence that the code generator contributes its share of bugs to the query engine. From an informal examination of the JIRA for the entire Cloudera Impala project [38], I found 2379 issues marked *bug* to date, and of those, 243—about 10%—contained the keyword "codegen". From a random sample, I also noted that many of these "codegen" bugs talked about correct behavior when codegen was disabled and incorrect behavior when codegen was enabled, demonstrating that bugs were directly related to code generation.

We built Crimp [110], a query transformer written in the Coq proof assistant that generates imperative programs from declarative queries. Crimp is verified to generate imperative programs with the same output as the query plan interpreter. By using Coq, we prove theorems about our implementation, so bugs cannot be introduced by programmer translation from an abstract model to implementation.

Crimp comprises a definition of relational algebra (CrimpRA), an interpreter for CrimpRA, an imperative language (Imp), an interpreter for Imp, a translator, and a machine-checked proof of the correctness of the translator. Imp is closer to C or Java than CrimpRA, but it is still quite far and not expected to be nearly as high performance. Although Imp is executable, the ultimate intended design for Crimp is to eventually translate the query into a C program. Our approach is to lower the input program through a sequence of imperative languages each meant to get the program closer to a C or Java version. The CompCert C to

assembly compiler has 8 intermediate languages [90]. Thus, a number of translators must be verified, but the small difference between each language makes the proofs easier.

5.2.1 Modeling and correctness specification

We model a tuple as a list of natural numbers and a relation as a list of tuples. Queries may be a project, select, or join. The interpreter for queries has the signature:

```
Definition runQuery (q : Query) (inputRelation1 : relation) (inputRelation2 : relation)
                : option relation
```

The `option` in the return type allows for runtime errors, such as projecting with an index that is out of bounds.

Our Imp language is a fairly typical imperative language with statements, expressions, and a small-step operational semantics (`runImp`). To make the equivalence proof simpler, it has a number of features specific to programs generated by queries: a tuple heap containing only tuples bound in for loops, a relation heap containing only the result relation, and language-level subroutines for selecting and projecting a tuple and matching two tuples. These domain-specific features would be relaxed incrementally in the compiler's subsequent translation steps. The compiler from query to Imp program generates a loop over the input relation. The structure of this loop is close to that of the recursive evaluation functions in the query interpreter.

Our correctness specification for the compiler is: *If the compiler accepts the query, and the query produces successful output, then the Imp program will succeed and produce the same output.* In Coq,

```
Theorem queryEquivalence:
  forall (q : Query) (p : ImpProgram),
    queryToImp q = Some p →
      forall (r1 r2 r' : relation), runQuery q r1 r2 = Some r' →
        runImp p r1 r2 = Some r'.
```

The general proof shape of `Project` is induction on q, inversion on the inductive hypothesis, induction on r1, and then iterative unfolding of terms and case analysis until the

inductive hypothesis for r1 can be applied. The automatic search tactic `crush` tends to succeed on the base case and after the inductive hypothesis is applied.

5.2.2 Next steps

There are currently two main limitations in our formalism and proofs. First, we have not yet proven the main theorem for `Join` queries. We have, however, made significant progress using a lemma equating the inner loop of the join query and with the inner loop of the Imp program. Including ordering in the representation of relations (lists of lists) makes equivalence overly strict, but it simplified early proofs. A less constrained definition of relations may be necessary when we attempt to build more complex physical operators.

Second, our query language does not yet allow composition of operators. The query language allows for only select, project, or join queries, without composition of these operators. The equivalence proof for a compositional query language could be obtained by using `queryEquivalence` for each operator as a lemma. However, composing operators only as sequence of transformations on relations would limit our imperative implementations to operator-at-a-time. We know that composing operators with pipelining is important to performance because it preserves data locality. Allowing for pipelining requires an additional step. There are two possible paths to prove equivalence between the original query semantics and a pipelined Imp program transitively: 1) *set-wise RA semantics* ≡ *pipelined RA semantics* ≡ *pipelined Imp semantics* or 2) *set-wise RA semantics* ≡ *set-wise Imp semantics* ≡ *pipelined Imp semantics*.

5.3 Discussion

In this chapter, I briefly reviewed software engineering issues in query compilers. While most query compilers use a template-based approach, recent work based on domain-specific language compiler frameworks provides evidence for better optimizations and developer productivity by using generative programming that is entirely embedded with the host language.

I gave anecdotal evidence of the need for sound testing methods in query compilers and presented early work on Crimp, a verified query compiler.

Chapter 6
DISCUSSION

In my introduction I stated that my research goal was to improve performance and programmer productivity for data-intensive systems and bring high performance systems closer to data analytical systems. In chapters 2-4, I presented systems, a survey, and experiments that supported this goal. In this chapter, I discuss the limitations of the work in this dissertation and then opportunities for future work.

6.1 Limitations

6.1.1 Grappa

In the design and evaluation of Grappa, we were able to build a productive general purpose programming model for data-intensive applications that gets good performance. I discuss some of the limitations in the runtime system and in the presented analysis of the system.

Dataflow with multithreading Our programming model of many latency-tolerant, fine-grained tasks attempts to abuse a cluster of general purpose processors as a dataflow machine using multithreading, but we did not evaluate whether we implemented this model to be robustly efficient for many applications. In both Grappa's task and message batching systems, we used prefetching to maintain various software-controlled data pipelines to avoid stalls on cache misses. We also tried to conserve precious memory bandwidth; for example, the batcher only touches messages once and we always run a context that we have prefetched. Despite these efforts, one main limitation was clear to us when we designed applications: the random access throughput of blocking delegates was far less than half that of asynchronous delegates. This suggested that our task wake-up mechanism was not efficient, and an in-

depth investigation in a system like Grappa would be interesting. A related limitation is that our mechanism for avoiding cache misses during context switches was not robust; we heuristically prefetch the top of the stack and then hope that ILP in the restarted task will overlap misses on other data. I believe that static and/or dynamic code and data analysis—informed by a deeper study of application behaviors—are necessary to make sure the right data is always prefetched.

Critical path and discretionary batching Grappa batches all delegates into large messages by delaying their transmission; however, if some delegates are on the critical path of the application then delaying them might hurt overall completion time. In Section 2.5.1, I presented a study of how concurrency affects Grappa's throughput. Using tree traversal benchmark, we found that narrow trees did not provide sufficient throughput to saturate the machine. However, we did not study how much faster we could complete the traversal if we prioritized messages on the critical path. Unlike traversing a linked list, traversing an unbalanced tree presents edges with a *range* of latency-tolerance levels, so Grappa's batcher would still be useful.

One DRAM-packed node? In this thesis, I constrained my focus mainly to *distributed* execution of data-intensive workloads. However, there is also research interest in using single shared memory machines packed with DRAM (on the order of a TB) for complex analytics [103, 135, 124]. In an early study with handwritten queries in Grappa, I found that Grappa required 8 16-core nodes to exceed the performance of OpenMP on 1 16-core node of the cluster; on the other had, Grappa's performance continued to improve through 64 nodes [109]. Studying the use of PGAS languages for query processing in big shared memory machines is an interesting future direction.

6.1.2 Radish

Query processing techniques and cost-based optimization I surveyed many techniques in Chapter 3 for improving in-memory query processing, and in-memory hash algorithms and compiled pipelines inspired the design of Radish. However, Radish does not employ some important strategies like column-oriented storage, compression, and decomposed join algorithms. I picked rows mainly for simplicity (Raco is row-oriented). For hash joins—other than symmetric versus asymmetric (Section 4.3.3)—Radish currently has just one algorithm: send the full tuples from both sides to the target partition and start processing the matches there. There is a broader design space of additional choices; for example, for each side should we send the whole tuple or just the key? The answer depends on the data, but we did not use a cost-based optimizer with statistics. The lack of a cost-based optimizer also meant other parameters were not configurable like how much memory to allocate for data structures. For this reason, I broadly interpret the results of our performance comparison to DBX in Section 4.7.2 to mean that Radish is competitive with other distributed query engines—and thus interesting for further study.

Applicability As discussed in Section 4.9.1, Radish's high compile times (10s of seconds) and lack of efficient fault tolerance make it applicable to a certain range of tasks: those that run at least a minute or multiple times, on clusters of less than 1000 nodes. To reduce compile times we could generate bytecode with calls to the PGAS runtime library but would give up the benefits of a PGAS compiler.

6.2 Opportunities for future work

The work in this thesis motivates future work around integrating and co-compiling hand-written parallel algorithms for complex analytics tasks.

6.2.1 Optimization amidst user-defined functions

One way that analytics are written with a combination of general purpose and query languages is user-defined functions (UDFs). UDFs are very common for when programmers need to operate on special data types and run special algorithms on tuples. Query optimizers that reason over only a known set of physical operators and treat UDFs as black boxes may pick poor query plans. I briefly present prior work that analyzed UDFs to improve query planning and then discuss how a system like RADISH might enable new UDF optimizations.

Prior work on adapting query planning to expensive UDFs Query planners often reduce the number of tuples processed by multiple operators by employing the heuristic of pushing selections as early as possible in the dataflow graph. However, when the query includes predicates are costly to compute, the most efficient execution plan may be one where the predicates come later, such as after a reducing aggregate. Hellerstein [64] and Chaudhuri et al. [32] present optimization techniques for query plans that include predicates that are expensive to evaluate, such as correlated subqueries and UDFs. These efforts propose estimating two costs for the predicates: the cost to process a tuple and the conventional cost of estimating selectivity. The space of query plans explodes when predicates can be reordered, so these works present algorithms that involve ranking predicates to make the search asymptotically efficient. Hellerstein proposes three specific inputs to the cost function (time per invocation, time per byte of input, percentage of bytes accessed), but left the task of estimating them accurately as a future challenge.

Prior work on analyzing UDFs within data parallel programming models Tupleware [42] uses LLVM to analyze UDFs to estimate two factors for an operator: compute time and load time. Compute time is estimated from static instructions and load time is estimated as the number of cycles to load operands. This measure is based only on bandwidth and not latency. Tupleware uses these factors in a rough heuristic to determine whether a map operator should be vectorized or pipelined. If compute time ¿ load time, then the operator

is predicted to be compute bound and is vectorized. Otherwise, the operator is pipelined to increase data locality to reduce memory bandwidth usage.

Including operators with UDFs in query optimization requires that their properties be known. Manimal [78] and HadoopToSQL [76] use simple static analyses on map UDFs in MapReduce jobs to infer selections and projections to apply database-style optimizations. Specifically, Manimal reduces usage of disk bandwidth using relational indexing and column-oriented storage and HadoopToSQL executes the extracted query on a database. Hueske et al. [70, 69] presented techniques to automatically reorder a plan of "MapReduce-style" UDFs. These UDFs are passed to parallel second-order functions like Map, Reduce, and Match. The technique uses knowledge of what attributes UDFs read and write found with a conservative analysis along with reordering proofs for five higher- order functions to enable a number of plan rewrites. SQL/MapReduce [56] combines SQL and table-valued, MapReduce-style UDFs into one programming model. At query time, the UDF interface defines the input/output schema. The graph framework GraphX [57] analyzes UDFs at runtime using Java's reflection to find unused attributes of vertices and eliminate joins with those tables.

UDFs within Radish queries Radish is a first step towards integrating query processing with PGAS languages. However, the integration has been quite loose: Radish only adds code generation to link a conventional query optimizer with a PGAS compiler. Applications are built as complex queries comprising UDFs or general-purpose code with language-integrated queries.

Because Radish generates parallel- and locality-aware code, we could unlock optimizations on black-box UDFs that would be invisible to both a query planner and a sequential compiler. Rather than analyze the UDF with LLVM as in Tupleware, we would just inline the UDF into the generated code so that the PGAS compiler can see it. For example, two UDFs separated by communication might be able to be co-optimized if the compiler can look across the communication point.

Consider an example inspired by astronomical image processing. We have two collections

```
1  Diff(im1: Image, im2: Image)
2      : Image {
3    z = new Image()
4    for i in z.len {
5      z[i] = im1[i] - im2[i]
6    }
7    return z
8  }
9
10 Blur(im: Image) : Image {
11  w = new Image()
12  for i in w.len {
13     w[i] = (im[i-1] +
14             im[i] +
15             im[i+1]) / 3
16   }
17 }
18
19 select Obs.rasc as rasc,
20        Obs.decl as decl,
21        Diff(Blur(Obs.im),
22             Hist.im) as diffed
23   from Obs, Hist
24   where Obs.rasc = Hist.rasc
25       and Obs.decl = Hist.decl;
```

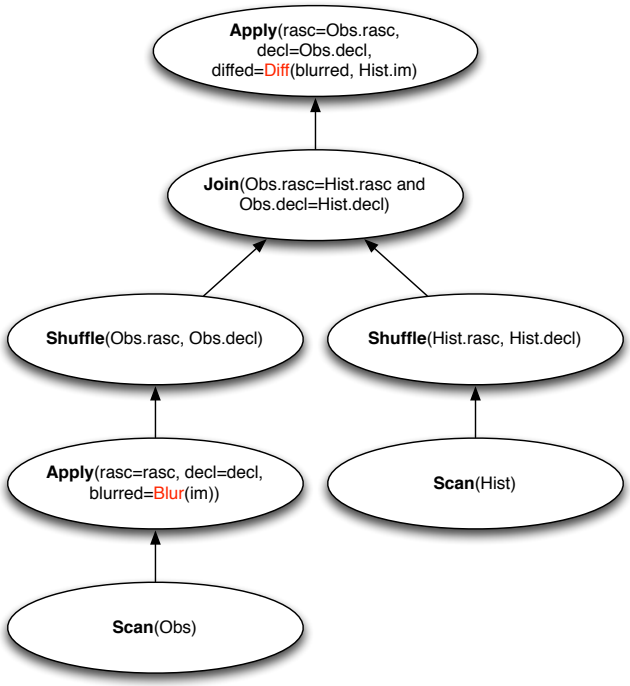


Figure 6.1: Query and query plan for blur-and-difference.

of images of the sky, identified by coordinates right ascension and declination. One collection, `Hist`, comprises previous observations and the other collection, `Obs`, comprises new observations. We wish to compare the each historical image to a blurred version of the corresponding new image. We model the collections as relations with the schema `(rasc: float, decl: float, im: Image)`. For simplicity, we assume 1-dimensional images. Since `Image` is an opaque data type to the query optimizer, we write the blur-and-difference operation as a query that uses UDFs for the image processing (Figure 6.1, left).

The resulting query plan might look like the one shown on the right side of Figure 6.1. The call to `Blur` was pushed as early in the dataflow as possible, while the call to `Diff` had to be placed after the join because it uses both images.

The generated code with the UDFs inlined might look like the left program in Figure 6.2. The parallel compiler can then perform loop fusion on the two loops over images, shown on

```
1  forall obs in Obs {
2    w = new Image()
3    for i in w.len {
4      w[i] = (obs.im[i-1] +
5              obs.im[i] +
6              obs.im[i+1]) / 3
7    }
8    Tuple t1(obs.rasc, obs.decl, w)
9    on partition(hash(t1.rasc, t1.decl)) {
10     forall t2 in
11         probe(hash(t1.rasc, t1.decl)) {
12       z = new Image()
13       for i in z.len {
14         z[i] = t1.im[i] - t2.im[i]
15       }
16       Tuple t3(t1.rasc, t1.decl, z)
17       emit(t3)
18     }
19   }
20 }
```

```
1  forall obs in Obs {
2    Tuple t1(obs.rasc, obs.decl, obs.im)
3
4    on partition(hash(t1.rasc, t1.decl)) {
5      forall t2 in
6          probe(hash(t1.rasc, t1.decl)) {
7        z = new Image()
8        for i in z.len {
9            z[i] = (t1.im[i-1] +
10                   t1.im[i] +
11                   t1.im[i+1]) / 3
12                 - t2.im[i]
13       }
14       Tuple t3(t1.rasc, t1.decl, z)
15       emit(t3)
16     }
17   }
18 }
```

Figure 6.2: PGAS code for blur-and-difference, with UDFs inlined. The left is before loop fusion and the right is after loop fusion.

the right of Figure 6.2. The communication (`on partition`) implementing the `Shuffle` does not impede this transformation. We might expect the fused version to be faster because it only traverses the `Obs` image once, rather than traversing it once to blur and then traversing the blurred image.

6.2.2 Optimizing programs with language-integrated queries

Another programming model for application-query integration is language-integrated queries (LIQs[1]), or queries that appear within a program written in a general purpose programming language. Declarative queries and imperative programming languages differ in several ways: how they are optimized, how they are expressed and tuned, and in their approaches to data types and modularity [39]. The term "impedance mismatch"[99] usually refers to the inefficiencies in object-relational mappings but also applies to these other differences. This mismatch remains an open problem but there are numerous relatively successful integrated

[1]LIQ is not to be confused with LINQ, which is Microsoft's .NET integrated query language. LINQ is an example of a LIQ programming model.

systems [39]. I briefly discuss work that has explored optimizing across a query and sequential host language. Then I discuss possibilities for optimizing across a query and *parallel and locality-aware* host language.

Prior work on optimizing LIQs and application code together Similarly to my motivation, Viglas et al. [147] argue that generating code from queries is an effective vehicle for tighter integration between queries and imperative languages. Specifically, they propose work in co-designing memory management, memory layout, and use of data semantics.

The interface between imperative code and databases in applications is often inefficient or used inefficiently by programmers [35]. Prior work uses code analysis to optimize these applications. Weidermann et al. [153, 154] used static analysis of programs with implicit database interaction to extract more efficient queries. Their approach uses an operational semantics of the host language for discovering data traversal paths and conditions in the program. These paths and conditions are combined to form a database query. The technique supports translating joins that are generated by iteration of an object field (e.g., for each employee print their managers name) but does not infer joins from conditions in the code. QBS [36] achieved a similar goal and inferred more queries. QBS takes a different approach: the technique generates Hoare-style verification conditions for a code fragment and then uses constraint-based synthesis to find a relational algebra expression that satisfies the conditions. This technique is able to translate looping code beyond just iteration: it handles joins derived from program conditions, as well as aggregates. I am not aware of a similar system that supports updates or insertions to the database.

LIQs in PGAS programs When the host language is parallel and locality-aware, consideration of application context for query optimization becomes even more important. To avoid costly data movement, the query optimizer needs to consider the distribution of the input and output data. In PGAS languages, this flavor of information is available to share with the query engine. However, locality information typically stops short of the meaning of

data structures. For example, an input table to a LIQ might already be in the form of an index (e.g., in a distributed hash table).

Interesting research questions are 1) what kinds of contextual information from a parallel application are useful for optimization of integrated queries? 2) how can we automatically infer structural information about distributed data structures that aids query optimization? 3) how do we balance the high-performance programmers intuition with portability and the query optimizers decisions? 4) what is the most effective way to profitably pick a distribution for optimizing the whole application?

6.2.3 Parity with handwritten parallel programs

In Section 4.7.6, I presented the results of a case study comparing the performance of Radish-generated and handwritten versions of PageRank. The remaining differences that I identified are within the reach of current techniques. However, more specialized algorithms and data structures exist that we did not even employ in the handwritten code (e.g., compressed sparse representations for matrices [28]), so we should study how to incorporate them into a query planner.

6.2.4 Query processing algorithms for PGAS

Radish currently has a small number of algorithms, but there are many others for non-uniform memory access architectures and distributed systems. We propose that a PGAS language could be the right interface for an engine that is portable to both large shared memory machines and clusters. Guiding research questions include: 1) how must existing locality-aware algorithms change to adapt to PGAS? 2) how can the new algorithms with generic elements derive benefits from query compilation?

Chapter 7
RELATED WORK

I discuss aspects of work related to GRAPPA and RADISH that are not covered in Chapter 3's survey of fast analytical query processing techniques.

7.1 Distributed shared memory

The techniques used in GRAPPA's language and runtime system are closely related to multithreading systems, software distributed shared memory systems, and partitioned global address space languages.

7.1.1 Multithreading

Systems with hardware-based multithreading to tolerate memory latency include the Denelcor HEP [139], Tera MTA [13], Cray XMT [54], Simultaneous multithreading [145], MIT Alewife [4], Cyclops [10], and GPUs [53]. Hardware multithreading often co-occur with lower single-threaded performance. As a software implementation of multithreading for mainstream general-purpose processors, GRAPPA provides the benefits of latency tolerance only when warranted, leaving single-threaded performance intact.

GRAPPA's closest software-based multithreading ancestor is the Threaded Abstract Machine (TAM) [44]. TAM is a software runtime system designed for prototyping dataflow execution models on distributed memory supercomputers. Like GRAPPA, TAM supports internode communication, management of the memory hierarchy, and lightweight asynchronous scheduling of tasks to processors, all in support of computational throughput despite the high latency of communications. A notable conclusion [46] was that multithreading for latency tolerance was fundamentally limited because the latency of the top-level store (e.g.,

L1 cache) is in direct competition with the number of contexts that can fit in it. However, we found prefetching is effective at hiding the DRAM latency in context switching. Indeed, a key difference between Grappa's support for lightweight threads and that of other user level threading packages, such as QThreads [152], TBB [129], Cilk [23] and Capriccio [148] is Grappa's context prefetching. Grappa's choice of what to prefetch could be improved by compiler analyses inspired by Capriccio's for reducing memory usage.

7.1.2 Software distributed shared memory

Much of the innovation in DSM over the past 30 years focused on reducing the synchronization costs of updates. The first DSM systems, including IVY [92], used frequent invalidations to provide sequential consistency, inducing high communication costs for write-heavy workloads. Later systems relaxed the consistency model to reduce communication demands; some systems further mitigated performance degradation due to false sharing by adopting multiple-writer protocols that delay integration of concurrent writes made to the same page. The Munin [22] and TreadMarks [80] systems exploited both of these ideas but still incurred some coherence overhead. Munin and Blizzard [134] allowed the tracking of ownership with variable granularity to reduce false sharing. Grappa follows the lead of TreadMarks and provides DSM entirely at user-level through a library and runtime. FaRM [50] offers lower latency and higher throughput updates to DSM than TCP/IP via lock free and transactional access protocols exploiting RDMA, but remote access throughput is still limited to the RDMA operation rate, which is typically an order of magnitude less than the per node network bandwidth.

7.1.3 Partitioned Global Address Space languages

The high-performance computing community has largely discarded the coherent distributed shared memory approach in favor of the Partitioned Global Address Space (PGAS) model. Examples include Split-C [43], Titanium [159], Chapel [30], X10 [31], Co-array Fortran [119] and UPC [52]. What is most different between conventional DSM systems and PGAS sys-

tems is that locality is explicit, thereby encouraging developers to consider remote memory. Grappa follows this approach, implementing a PGAS system at the language level, thereby facilitating compiler and programmer optimizations.

Communication optimizations Other PGAS languages use compiler and runtime methods for improving communication and overlapping communication with computation. Chen et al. [33] use compile time transformations on memory accesses in UPC programs to do *split-phase* communication, that is, separate initiation and completion. Because static use-of-variable analyses can be too conservative, HUNT [72] delayed the completion point longer at execution time. Other work studied using one-sided communication and overlapping communication with computation to improve scalability of UPC programs [21, 118]. Grappa relies solely on multithreading for creating overlap, but the overhead of scheduling would be reduced by also using compiler assisted overlap.

Other work [33, 11] used compiler hints to the runtime regions of code where multiple initiations for the same partition may occur before any blocking completions so they might be batched into one message. In other work they performed coalescing at compile time [34]. Grappa performs batching across different tasks, but batching could be made more efficient with hints from the programmer or compiler.

Husbands et al. [71] optimized a single kernel, parallel LU factorization, on UPC using lightweight multithreading and one-sided communication. Grappa combines multithreading with automatic message batching to achieve high throughput without as much programmer effort for communication optimization, but application-specific batching will reduce the overhead of scheduling or save memory bandwidth by improving the locality of message buffers.

7.1.4 Asynchrony in data-intensive processing frameworks

There have been efforts to build data-intensive processing with more fine-grained computation. There are parameter servers for distributed machine learning algorithms using asynchronous communication and distributed key-value storage built from RPCs [6, 7]. The

incremental data-parallel system Naiad [105] achieves both high-throughput for batch workloads and low-latency for incremental updates. These two systems are built directly on messaging libraries rather than a DSM.

7.2 Parallel query evaluation

I discuss aspects of other work on parallel query evaluation as it relates to Radish that has not been covered in Chapter 3 and Chapter 5.

7.2.1 Query execution using tasks

Radish's execution model is most resembles morsel-driven parallelism (MDP) for NUMA shared memory machines [89]. Both are task-based and exploit shared data structures so that data and computation parallelism are not conflated. Radish is different in that it expresses the task-parallelism in `forall` so that the PGAS runtime chooses how to run tasks, and it produces parallel programs for distributed memory clusters.

MDP is based on a hardcoded execution model of pulling chunks of input tuples from a work queue. Radish's back end, on the other hand, may group `forall` iterations into tasks and distribute the tasks over executors however it likes. Use of `forall`, in addition to the valid-anywhere nature of addresses in PGAS, allows Radish to—with no changes—take advantage of dynamic load balancing mechanisms provided by the runtime system, while MDP's reliance on hardware shared memory restricts transparent dynamic load balancing to a single node.

The task-based execution model of MDP and Radish is prone to overheads of scheduling. MDP avoids these overheads by taking *morsels*, or chunks of tuples, at a time from a shared work queue and executing them without scheduling. Radish uses a task per tuple but is helped by Grappa's lightweight runtime system and fusing `forall` iterations together, as discussed in Section 4.5.2.

MDP relies on the traditional locality-oblivious model of shared memory that Grappa and Radish eschew for its poor fit to data-intensive workloads. GDP uses first-touch page

policy for allocating data, while RADISH distributes data with intent. MDP relies on hardware cache coherence for moving cache lines between workers, while RADISH moves data more explicitly with `on partition`.

7.2.2 Removing overheads of tuple-at-a-time

Chapter 3 discussed many ways to reduce the overhead of processing tuples in memory, including column-oriented storage, compression, and vectorized code. While RADISH does not generate vectorized code for the CPU, our back end, GRAPPA, uses lightweight threads and batching of network messages to coarsen tuple-grain execution to achieve high bandwidth. As discussed in Section 3.1.3, there is a non-trivial trade off between pipelined and vectorized code. To adopt vectorized execution, RADISH could emit vectorizable sequential code by dividing the iterations of `forall` into chunks. The Chapel compiler has experimental vectorization of `forall` [131], but currently I am unaware of a compiler vectorizes code containing `on partition`.

7.2.3 Data analytics on HPC systems

A number of systems have adapted in-memory data analytics platforms to HPC environments. Wang et al. [150] improves the task scheduling of in-memory Spark jobs on an HPC system. DataMPI [93] extends MPI with features to support dataflow jobs. Lu et al. [97] enhances Spark performance by re-implementing its data shuffle operation natively on RDMA. In the work of this thesis, we provided a data analytics programming model on an HPC stack by instead exploiting existing HPC languages and runtimes, but we did not explore scaling up to large deployments.

Chapter 8
CONCLUSION

In the final chapter, I summarize the dissertation, summarize my research contributions, and make concluding remarks.

8.1 Summary of prior chapters

In Chapter 1, I discussed two important developments that motivated this thesis: 1) applications demand both data analytics and high-performance algorithms and 2) in-memory query processing techniques provide high performance for declarative programs on modern hardware. In this thesis, I challenge the separation between the high performance computing and data management platforms when they are applied to data-intensive applications.

In Chapter 2, I argued that performance and programmer productivity demand the implementation of a shared memory model for data-intensive computing. I presented the system we built, Grappa, which exploits the concurrency in data-intensive applications to utilize commodity hardware efficiently. Our evaluation of Grappa showed that the system could be used to build data-intensive frameworks that were faster than other implementations.

In Chapter 3, I surveyed techniques that improved the performance of in-memory query processing. I categorized the work into three categories of inefficiency that they address. I focused further on specialization and presented an illustration of the ways analytical databases use compilation. The survey provided inspiration for the work in the following chapter.

In Chapter 4, I presented Radish, an in-memory query processor implemented in distributed shared memory programming models like Grappa. Radish generates parallel programs for pipelines in the query plan. Compared to compiled iterators, Radish achieves 2.4× lower running time on TPC-H queries. I also explained the performance gap between

generated and handwritten code in a case study.

Because of the recent surge of interest in query compilation, in Chapter 5 I discussed issues on software engineering. I reviewed ideas constructing and testing query compilers.

In Chapter 6 I explained the limitations of the systems and experiments presented in this dissertation, as well as opportunities for future research. The limitations of GRAPPA related to dataflow computing and critical paths in data-intensive programs. The limitations of RADISH included a discussion of better query optimizations and applicability. Future work focused on co-optimization of handwritten parallel codes and queries and further exploration of using shared memory programming models to build query processors.

8.2 Summary of contributions

My thesis statement is

When applied to data-intensive applications, high performance parallel systems and new database query evaluation techniques support improved performance, programmer productivity, and closer interaction between handwritten parallel programs and declarative queries.

In my work I contributed two research artifacts, GRAPPA and RADISH, as well as experimental evidence supporting increased performance. GRAPPA supports productivity by providing a shared memory programming model for clusters, while maintaining good performance. RADISH supports productivity by implementing fast processing for declarative analytics code. By implementing query processing with a combination of a query planner and high-performance in-memory techniques, this thesis takes a step toward closer interaction between the two computing paradigms.

The specific contributions are the following:

- The GRAPPA system implementation, which provides a shared memory model for writing data-intensive applications and frameworks for clusters. GRAPPA uses massive concurrency to keep the machine busy despite lack of locality, uses memory bandwidth-efficient message batching to overcome the low injection rate of network cards and

utilize the full bandwidth of commodity networks, provides an efficient means to move computation to reduce the number of messages, and software multithreading that has low switching overhead to support massive parallelism.

- Experimental results on microbenchmarks, which show that GRAPPA's message batching outperforms native atomic operations in RDMA network cards and that GRAPPA's context switching is limited by memory bandwidth not latency. We also presented experiments that demonstrate the role of concurrency in GRAPPA's operation.

- Implementations of domain-specific data analysis frameworks upon GRAPPA, as well as example applications built in those frameworks.

- Experimental results that show that GRAPPA is faster than other data-intensive frameworks GraphLab and Spark, primarily due to its superior communication stack. GRAPPA is faster than GraphLab even when it uses a naïve graph partitioning that has less locality.

- A survey of in-memory query evaluation techniques, including a classification under three categories of inefficiency: generality, execution model, and storage model.

- A survey of the role of compilers for general purpose languages in query evaluation, including a classification of existing work according to specificity of code being compiled and the scope of optimization of the compiler.

- *Compiled parallel pipelines* (CPP), an approach for translating queries into efficient PGAS code leveraging existing parallel compilers

- An end-to-end implementation of query processing, called RADISH, which integrates a distributed query optimizer (RACO), CPP, shared distributed data structures, and GRAPPA

- An experimental evaluation of CPP compared to a compiled iterator strategy (CVI). On TPC-H queries, executing queries by CPP is on average 2.4× faster than CVI.

- An experimental comparison of RADISH with a state-of-the-art database platform on TPC-H queries, which shows that its performance is competitive

- A case study on the performance of RADISH-generated code versus a handwritten GRAPPA program. Specifically, we use PageRank, show the causes of slowdown in the generated program, and give support for why parity is within reach.

8.3 Remarks

As I conducted this research, I acquired a whole-system perspective of the design and implementation of data-intensive computations. Just as this thesis questions the separation between database systems and tuned parallel programs, we should examine critically the circles we draw around every component in the system. This call includes the role of compilation in query evaluation systems. Most work on compilation of queries, including RADISH, recycles two existing tools: query planners and general purpose compilers. This design choice is convenient and practical; there is plenty to study in the translation between the program representations used by these two tools before significantly redesigning the tools themselves. However, it is the role of research to reconsider the entire implementation of query evaluation, from query to executable code. This vertical reconsideration reflects efforts across computing; as the improvements in general-purpose processors wane, researchers are turning to specialization and examining the abstractions of the entire computing stack. We need to develop useful abstractions and smart automatic tools to make specialization sustainable.

BIBLIOGRAPHY

[1] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. row-stores: how difference are they really? In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data - SIGMOD '08*, page 967, New York, New York, USA, jun 2008. ACM Press.

[2] Sarita V. Adve, Mark D. Hill, Sarita V. Adve, and Mark D. Hill. Weak ordering—a new definition. In *Proceedings of the 17th annual international symposium on Computer Architecture - ISCA '90*, volume 18, pages 2–14, New York, New York, USA, 1990. ACM Press.

[3] Foto N. Afrati and Jeffrey D. Ullman. Optimizing Multiway Joins in a Map-Reduce Environment. *IEEE Transactions on Knowledge and Data Engineering*, 23(9):1282–1298, sep 2011.

[4] Anant Agarwal, Ricardo Bianchini, David Chaiken, Frederic T Chong, Associate Member, Kirk L. Johnson, David Kranz, John D Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The MIT Alewife Machine. *International Symposium on Computer Architecture*, 87(3):430–444, 1999.

[5] Sameer Agarwal, Davies Liu, and Reynold Xin. Apache spark as a compiler: Joining a billion rows per second on a laptop. `https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html`, 2016.

[6] Amr Ahmed, Moahmed Aly, Joseph Gonzalez, Shravan Narayanamurthy, and Alexander J. Smola. Scalable inference in latent variable models. *Proceedings of the fifth ACM international conference on Web search and data mining - WSDM '12*, page 123, 2012.

[7] Amr Ahmed, Nino Shervashidze, Shravan Narayanamurthy, Vanja Josifovski, and Alexander J Smola. Distributed Large-scale Natural Graph Factorization. *Proceedings of the 22Nd International Conference on World Wide Web*, pages 37–48, 2013.

[8] Anastassia Ailamaki, David J Dewitt, Mark D Hill, and David a Wood. DBMSs On A Modern Processor: Where Does Time Go? *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB'99)*, 1394:266–277, 1999.

[9] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, and Others. The Stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964, 2014.

[10] George Almási, Cacaval Clin, José G. Castaños, Monty Denneau, Derek Lieber, José E. Moreira, and Henry S. Warren. Dissecting Cyclops: A Detailed Analysis of a Multithreaded Architecture. *Computer Architecture News*, 31(1):26–38, mar 2002.

[11] Michail Alvanos, Montse Farreras, Ettore Tiotto, José Nelson Amaral, and Xavier Martorell. Improving Communication in PGAS Environments: Static and Dynamic Coalescing in UPC. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 129–138, New York, NY, USA, 2013. ACM.

[12] Gail Alverson, Robert Alverson, David Callahan, Brian Koblenz, Allan Porterfield, and Burton Smith. Exploiting Heterogeneous Parallelism on a Multithreaded Multiprocessor. In *Proceedings of the 6th International Conference on Supercomputing*, ICS '92, pages 188–197, New York, NY, USA, 1992. ACM.

[13] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. *Proc. of 4th iInt. Conf. on Supercomputing*, 18(3):1–6, sep 1990.

[14] AMD64 ABI. `http://www.x86-64.org/documentation/abi-0.99.pdf`, July 2012.

[15] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, Matei Zaharia, and U C Berkeley. SparkSQL: Relational Data Processing in Spark.

[16] Krste Asanovíc, Rastislav Bodik, Bryan Catanzaro, Joseph Gebis, Parry Husbands, Kurt Keutzer, David Patterson, William Plishker, John Shalf, Samuel Williams, and Katherine Yelick. The Landscape of Parallel Computing Research : A View from Berkeley. pages 1–54, 2006.

[17] K. Bache and M. Lichman. UCI machine learning repository, 2013.

[18] Ronald Barber, Guy Lohman, Vijayshankar Raman, Richard Sidle, Sam Lightstone, and Berni Schiefer. In-memory BLU acceleration in IBM's DB2 and dashDB: Optimized for modern workloads and hardware architectures. In *Proceedings - International Conference on Data Engineering*, volume 2015-May, pages 1246–1252. IEEE, apr 2015.

[19] Rajkishore Barik, Raghavan Raman, and Vivek Sarkar. Work-first and help-first scheduling policies for async-finish task parallelism. *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12, may 2009.

[20] Rajkishore Barik, Jisheng Zhao, David Grove, Igor Peshansky, Zoran Budimlic, and Vivek Sarkar. Communication Optimizations for Distributed-Memory X10 Programs. *2011 IEEE International Parallel & Distributed Processing Symposium*, (1):1101–1113, may 2011.

[21] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick. Optimizing bandwidth limited problems using one-sided communication and overlap. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, page 63. IEEE, 2006.

[22] J. K. Bennett, J. B. Carter, W. Zwaenepoel, J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: distributed shared memory based on type-specific memory coherence, 1990.

[23] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. *Journal of Parallel and Distributed Computing*, 37(1):55–69, aug 1996.

[24] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, sep 1999.

[25] Hans-J. Boehm. A Less Formal Explanation of the Proposed C++ Concurrency Memory Model. C++ standards committee paper WG21/N2480 = J16/07-350, `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2007/n2480.html`, December 2007.

[26] PA Boncz. *Monet; a next-Generation DBMS Kernel For Query-Intensive Applications*. 2002.

[27] Peter Boncz, Marcin Zukowski, and Niels Nes. MonetDB / X100 : Hyper-Pipelining Query Execution. 2005.

[28] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. Parallel Sparse Matrix-vector and Matrix-transpose-vector Multiplication Using Compressed Sparse Blocks. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 233–244, New York, NY, USA, 2009. ACM.

[29] AF Cárdenas. Analysis and performance of inverted data base structures. *Communications of the ACM*, 1975.

[30] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 21(3):291–312, aug 2007.

[31] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, 2005.

[32] Surajit Chaudhuri and Kyuseok Shim. Optimization of queries with user-defined predicates. *TODS*, 1999.

[33] Wei-yu Chen, Dan Bonachea, Costin Iancu, and Katherine Yelick. Automatic nonblocking communication for partitioned global address space programs. *Proceedings of the 21st annual international conference on Supercomputing - ICS '07*, pages 158–167, 2007.

[34] W.Y. Chen, C. Iancu, and K. Yelick. Communication optimizations for fine-grained UPC applications. *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, pages 267–278, 2005.

[35] Alvin Cheung. *Rethinking the Application-Database Interface.* PhD thesis, MIT, September 2015.

[36] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Optimizing database-backed applications with query synthesis. *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation - PLDI '13*, page 3, 2013.

[37] Eric S Chung, John D Davis, and Jaewon Lee. LINQits: Big Data on Little Clients. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 261–272, New York, NY, USA, 2013. ACM.

[38] Cloudera. Cloudera issues, impala. `https://issues.cloudera.org/projects/IMPALA/issue`, April 2016.

[39] William R Cook and Ali H Ibrahim. Integrating Programming Languages & Databases: What's the Problem? Technical Report 0448128, 2006.

[40] George P. Copeland and Setrag N. Khoshafian. A decomposition storage model. *ACM SIGMOD Record*, 14(4):268–279, may 1985.

[41] Intel Corporation. Intel 64 and ia-32 architectures optimization reference manual, January 2016.

[42] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Cetintemel, and Stan Zdonik. An Architecture for Compiling UDF-centric Workflows. pages 1466–1477.

[43] David E Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten Von Eicken, and Katherine Yelick. Parallel programming in Split-C. In *Supercomputing*, pages 262–273. IEEE, 1993.

[44] David E Culler, Seth Copen Goldstein, Klaus Erik Schauser, and T. Voneicken. TAM - A Compiler Controlled Threaded Abstract Machine. *Journal of Parallel and Distributed Computing*, 18(3):347–370, jul 1993.

[45] David E. Culler, Anurag Sah, Klaus E. Schauser, Thorsten von Eicken, John Wawrzynek, David E. Culler, Anurag Sah, Klaus E. Schauser, Thorsten von Eicken, John Wawrzynek, David E. Culler, Anurag Sah, Klaus E. Schauser, Thorsten von Eicken, John Wawrzynek, David E. Culler, Anurag Sah, Klaus E. Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain parallelism with minimal hardware support: a compiler-controlled threaded abstract machine. *ACM SIGARCH Computer Architecture News*, 19(2):164–175, apr 1991.

[46] D.E. Culler, K.E. Schauser, and Thorsten Von Eicken. Two fundamental limits on dataflow multiprocessing. *Architectures and Compilation*, 3:1–14, 1993.

[47] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, oct 1991.

[48] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):1–13, 2008.

[49] J Dees and P Sanders. Efficient many-core query execution in main memory column-stores. *Data Engineering (ICDE)*, 2013.

[50] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, 2014.

[51] T Von Eicken, DE Culler, and SC Goldstein. Active messages: a mechanism for integrated communication and computation. *ACM SIGARCH*, 1992.

[52] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. UPC: Distributed Shared-Memory Programming. jul 2005.

[53] Kayvon Fatahalian and Mike Houston. A closer look at GPUs. *Communications of the ACM*, 51(10):50, 2008.

[54] John Feo, David Harper, Simon Kahan, and Petr Konecny. ELDORADO. *Computing Frontiers*, pages 28–34, 2005.

[55] Message Passing Interface Forum. MPI: A message-passing interface standard, version 3.0. Technical report, 2012.

[56] Eric Friedman, Peter Pawlowski, and John Cieslewicz. SQL/MapReduce: a practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *Proceedings of the VLDB Endowment*, 2(2):1402–1413, aug 2009.

[57] JE Gonzalez, RS Xin, and A Dave. GraphX: graph processing in a distributed dataflow framework. *Proceedings of the 11th Operating Systems Design and Implementation*, 2014.

[58] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. *OSDI*, 2012.

[59] A Gottlieb, R Grishman, and CP Kruskal. The NYU Ultracomputer; Designing an MIMD Shared Memory Parallel Computer. *IEEE Transactions*, 1983.

[60] Goetz Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, SIGMOD '90, pages 102–111, New York, NY, USA, 1990. ACM.

[61] Rick Greer. All About Daytona, 2013.

[62] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel Madden. HYRISE. *Proceedings of the VLDB Endowment*, 4(2):105–116, nov 2010.

[63] Akihiro Hayashi, Jisheng Zhao, Michael Ferguson, and Vivek Sarkar. LLVM-based communication optimizations for PGAS programs. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC - LLVM '15*, pages 1–11, New York, New York, USA, 2015. ACM Press.

[64] Joseph M. Hellerstein. Optimization techniques for queries with expensive methods. *ACM Transactions on Database Systems*, 23(2):113–157, jun 1998.

[65] MP Herlihy and JM Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages*, 1990.

[66] T. Hoefler, T. Schneider, and A. Lumsdaine. Multistage switches are not crossbars: Effects of static routing in high-performance networks. In *Cluster Computing, 2008 IEEE International Conference on*, pages 116–125, Sept 2008.

[67] B Holt, J Nelson, B Myers, P Briggs, and L Ceze. Flat combining synchronized global data structures. *Conference on PGAS Programming Models*, 2013.

[68] Brandon Holt, Preston Briggs, Luis Ceze, and Mark Oskin. Alembic: Automatic Locality Extraction via Migration. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '14. ACM, 2014.

[69] Fabian Hueske, Mathias Peters, Aljoscha Krettek, Matthias Ringwald, Kostas Tzoumas, Volker Markl, and Johann-Christoph Freytag. Peeking into the optimization of data flow programs with MapReduce-style UDFs. *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 1292–1295, apr 2013.

[70] Fabian Hueske, Mathias Peters, and MJ Sax. Opening the black boxes in data flow optimization. *Proceedings of the VLDB Endowment*, pages 1256–1267, 2012.

[71] Parry Husbands and Katherine Yelick. Multi-threading and one-sided communication in parallel LU factorization. *Supercomputing*, (c):1, 2007.

[72] C. Iancu, P. Husbands, and P. Hargrove. HUNTing the overlap. *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, pages 279–290, 2005.

[73] InfiniBand Trade Association. InfiniBand Architecture Specification, Version 1.2.1. 2007.

[74] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *Eurosys*, pages 59–72, 2007.

[75] ISO/IEC JTC1/SC22/WG21. ISO/IEC 14882, Programming Language, C++ (Committee Draft). `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2800.pdf`, 2008.

[76] Ming-Yee Iu and Willy Zwaenepoel. HadoopToSQL: a mapReduce query optimizer. In *Proceedings of the 5th European conference on Computer systems - EuroSys '10*, page 251, New York, New York, USA, apr 2010. ACM Press.

[77] Suresh Jagannathan. Communication-passing style for coordination languages. In *Coordination Languages and Models*, pages 131–149. Springer, 1997.

[78] Eaman Jahani, Michael J. Cafarella, and Christopher Ré. Automatic optimization for MapReduce programs. *Proceedings of the VLDB Endowment*, 4(6):385–396, mar 2011.

[79] Somesh Jha, Jian Qiu, Andre Luckow, Pradeep Mantha, and Geoffrey C Fox. A tale of two data-intensive paradigms: Applications, abstractions, and architectures. In *Big Data (BigData Congress), 2014 IEEE International Congress on*, pages 645–652. IEEE, 2014.

[80] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. TreadMarks: distributed shared memory on standard workstations and operating systems, 1994.

[81] Alfons Kemper and Thomas Neumann. HyPer: A Hybrid OLTP & OLAP Main Memory Database System Based on Virtual Memory Snapshots. pages 195–206, 2011.

[82] I Klonatos and C Koch. Building Efficient Query Engines in a High-Level Language. *Proceedings of the VLDB Endowment*, pages 853–864, 2014.

[83] Konstantinos Krikellas. *The case for holistic query evaluation.* PhD thesis, 2010.

[84] Konstantinos Krikellas, Stratis D. Viglas, and Marcelo Cintra. Generating code for holistic query evaluation. *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 613–624, 2010.

[85] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *Proceedings of the 19th International Conference on the World Wide Web*, WWW '10, pages 591–600, New York, NY, USA, 2010. ACM.

[86] YC Kwon, K Ren, M Balazinska, B Howe, and J Rolia. Managing Skew in Hadoop. *IEEE Data Eng. Bull.*, 2013.

[87] Yongchul Kwon, Dylan Nunley, Jeffrey P. Gardner, Magdalena Balazinska, Bill Howe, and Sarah Loebman. Scalable clustering algorithm for N-body simulations in a shared-nothing cluster. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6187 LNCS:132–150, 2010.

[88] Jonathan K Lee and Jens Palsberg. Featherweight X10: A Core Calculus for Async-finish Parallelism. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 25–36, New York, NY, USA, 2010. ACM.

[89] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD '14: Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 743–754, 2014.

[90] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[91] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[92] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, nov 1989.

[93] F Liang, C Feng, X Lu, and Z Xu. Performance benefits of DataMPI: a case study with BigDataBench. *Big Data Benchmarks, Performance ...*, 2014.

[94] Juan Loaiza. Engineering database hardware and software together (oracle). In *Keynote at VLDB 2015*, Sep 2015.

[95] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[96] X Lu and NS Islam. High-performance design of Hadoop RPC with RDMA over InfiniBand. *Parallel Processing*, 2013.

[97] Xiaoyi Lu, Md Wasi Ur Rahman, Nahina Islam, Dipti Shankar, and Dhabaleswar K Panda. Accelerating spark with RDMA for big data processing: Early experiences. In *High-Performance Interconnects (HOTI), 2014 IEEE 22nd Annual Symposium on*, pages 9–16. IEEE, 2014.

[98] Roberto Lublinerman, Jisheng Zhao, Zoran Budimli'c, Swarat Chaudhuri, and Vivek Sarkar. Delegated Isolation. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, volume 46, pages 885–902, New York, New York, USA, 2011. ACM Press.

[99] David Maier. Advances in Database Programming Languages. In François Bancilhon and Peter Buneman, editors, *Advances in Database Programming Languages*, chapter Representi, pages 377–386. ACM, New York, NY, USA, 1990.

[100] Anirban Mandal, Rob Fowler, and Allan Porterfield. Modeling memory concurrency for multi-socket multi-core systems. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 66–75. IEEE, mar 2010.

[101] Stefan Manegold, Peter Boncz, and Martin L. Kersten. Generic database cost models for hierarchical memory systems. pages 191–202, aug 2002.

[102] Stefan Manegold, Peter A Boncz, and Martin L Kersten. Optimizing database architecture for the new bottleneck: memory access. *The VLDB Journal*, 9:231–246, 2000.

[103] Frank McSherry. Scalability! but at what cost? `http://www.frankmcsherry.org/graph/scalability/cost/2015/01/15/COST.html`, Jan 2015.

[104] Message Passing Interface Forum. Mpi: A message-passing interface standard, version 2.2. Specification, September 2009.

[105] Derek G Murray, Frank Mcsherry, and Rebecca Isaacs. Naiad : A Timely Dataflow System. *Symposium on Operating Systems Principles*, 2013.

[106] DG Murray, Michael Isard, and Y Yu. Steno: automatic optimization of declarative queries. *Conference on Programming Language Design and Implementation*, 2011.

[107] MVAPICH. MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE, sep 2014.

[108] Brandon Myers, Bill Howe, and Mark Oskin. Compiling queries for high-performance computing. Technical report, 2016.

[109] Brandon Myers, Jeremy Hyrkas, Daniel Halperin, and Bill Howe. Compiled Plans for In-Memory Path-Counting Queries. In *In-memory Data Management and Analysis*, pages 28–43. Springer, 2015.

[110] Brandon Myers, Kivanc Muslu, and Zachary Tatlock. Crimp: a certified relational algebra to imperative compiler. `https://github.com/uwplse/crimp`, 2014.

[111] Fabian Nagel, Gavin Bierman, and SD Viglas. Code generation for efficient query processing in managed runtimes. *Proceedings of the VLDB Endowment*, 7(12):1095–1106, 2014.

[112] S Navathe, S Ceri, G Wiederhold, and J Dou. Vertical partitioning algorithms for database design. *Transactions on Database Systems*, 1984.

[113] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-Tolerant Software Distributed Shared Memory. In *USENIX ATC*, jul 2015.

[114] Jacob Nelson, Brandon Myers, and AH Hunter. Crunching large graphs with commodity processors. *Proceedings of the 3rd USENIX conference on Hot topic in parallelism.*, pages 10–10, 2011.

[115] Jacob Eric Nelson. *Latency-Tolerant Distributed Shared Memory for Data-Intensive Applications*. PhD thesis, University of Washington, 2014.

[116] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment*, 4(9):539–550, 2011.

[117] D Nguyen, A Lenharth, and K Pingali. A lightweight infrastructure for graph analytics. *of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013.

[118] Rajesh Nishtala, Paul H. Hargrove, Dan O. Bonachea, and Katherine a. Yelick. Scaling communication-intensive applications on BlueGene/P using one-sided communication and overlap. *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12, may 2009.

[119] Robert W Numrich and John Reid. Co-Array Fortran for parallel programming. In *ACM Sigplan Fortran Forum*, volume 17, pages 1–31. ACM, 1998.

[120] Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P Sadayappan, and C.W. Tseng. UTS: An unbalanced tree search benchmark. *Languages and Compilers for Parallel Computing*, pages 235–250, 2007.

[121] John Ousterhout, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, Ryan Stutsman, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, and Diego Ongaro. The case for RAMCloud. *Communications of the ACM*, 54(7):121, jul 2011.

[122] S. Padmanabhan, T. Malkemus, A. Jhingran, and R. Agarwal. Block oriented processing of relational database operations in modern computer architectures. In *Proceedings 17th International Conference on Data Engineering*, pages 567–574. IEEE Comput. Soc, 2001.

[123] ParAccel. The paraccel analytic database: A technical overview, 2010.

[124] Yonathan Perez, Rok Sosič, Arijit Banerjee, Rohan Puttagunta, Martin Raison, Pararth Shah, and Jure Leskovec. Ringo: Interactive graph analytics on big-memory machines. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1105–1110. ACM, 2015.

[125] Keshav Pingali, Milind Kulkarni, Donald Nguyen, Martin Burtscher, Mario Mendez-Lojo, Dimitrios Prountzos, Xin Sui, and Zifei Zhong. Amorphous data-parallelism in irregular algorithms. *Computer*, 2009.

[126] Holger Pirk, Florian Funke, Martin Grund, Thomas Neumann, Ulf Leser, Stefan Manegold, Alfons Kemper, and Martin Kersten. CPU and cache efficient management of memory-resident databases. *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 14–25, apr 2013.

[127] Orestis Polychroniou, Arun Raghavan, and Kenneth a. Ross. Rethinking SIMD Vectorization for In-Memory Databases. *Sigmod*, pages 1493–1508, 2015.

[128] J Rao and H. Pirahesh. Compiled query execution engine using JVM. *Data Engineering, 2006.*, pages 23–23, 2006.

[129] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly Media, 2007.

[130] Kai Ren, YongChul Kwon, Magdalena Balazinska, and Bill Howe. Hadoop's Adolescence: An Analysis of {H}adoop Usage in Scientific Workloads. *Proc. VLDB Endow.*, 6(10):853–864, aug 2013.

[131] Elliot Ronaghan. Vectorization of chapel code. In *Chapel Implementers and Users Workshop (CHIUW)*, June 2015.

[132] Kenneth A. Ross. Selection conditions in main memory. *ACM Transactions on Database Systems*, 29(1):132–161, mar 2004.

[133] Michael Schmidt and Thomas Hornung. SPˆ2Bench: a SPARQL performance benchmark. *Data Engineering, 2009.*, 2009.

[134] Ioannis Schoinas, Babak Falsafi, and Ar Lebeck. Fine-grain access control for distributed shared memory. *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems (ASPLOS)*, 29(11):297–306, 1994.

[135] David Patterson Scott Beamer, Krste Asanović. Locality Exists in Graph Processing : Workload Characterization on an Ivy Bridge Server. *2015 IEEE International Symposium on Workload Characterization*, 2015.

[136] Jiwon Seo, Stephen Guo, and MS Lam. SociaLite: Datalog extensions for efficient social network analysis. *Data Engineering (ICDE), 2013*, 2013.

[137] Jiwon Seo, Jongsoo Park, J Shin, and MS Lam. Distributed socialite: a datalog-based language for large-scale graph analysis. *Proceedings of the VLDB Endowment*, 2013.

[138] Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. How to Architect a Query Compiler. In *SIGMOD*, 2016.

[139] BJ Smith. Architecture and applications of the HEP multiprocessor computer system. *25th Annual Technical Symposium*, 1982.

[140] Juliusz Sompolski, Marcin Zukowski, and Peter Boncz. Vectorization vs. compilation in query execution. *Proceedings of the Seventh International Workshop on Data Management on New Hardware - DaMoN '11*, (DaMoN):33–40, 2011.

[141] Jarred E Swalwell, Francois Ribalet, and E. Virginia Armbrust. SeaFlow: A novel underway flow-cytometer for continuous observations of phytoplankton in the ocean. *Limnology and Oceanography: Methods*, 9:466–477, 2011.

[142] Steven Swanson, Andrew Schwerin, and Mark Oskin. WaveScalar. *MICRO-36*, 2003.

[143] Rebecca Taft, Manasi Vartak, Nadathur Rajagopalan Satish, Narayanan Sundaram, Samuel Madden, and Michael Stonebraker. GenBase: A Complex Analytics Genomics Benchmark. In *SIGMOD*, SIGMOD '14, pages 177–188, New York, NY, USA, 2014. ACM.

[144] Transaction Processing Performance Council. TPC Benchmark H v2.14.4, 2012.

[145] D.M. Tullsen, S.J. Eggers, and H.M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. *Proceedings 22nd Annual International Symposium on Computer Architecture*, 23(2):392–403, 1995.

[146] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8), 1990.

[147] S Viglas, GM Bierman, and Fabian Nagel. Processing Declarative Queries Through Generating Imperative Code in Managed Runtimes. *IEEE Data Engineering Bulletin*, pages 12–21, 2014.

[148] Rob von Behren, Jeremy Condit, Feng Zhou, George C Necula, and Eric Brewer. Capriccio: Scalable Threads for Internet Services. *Symposium on Operating Systems Principles*, 37(5):268, 2003.

[149] S Wanderman-Milne and N Li. Runtime Code Generation in Cloudera Impala. *IEEE Data Eng. Bull.*, pages 31–37, 2014.

[150] Y Wang and R Goldstone. Characterization and optimization of memory-resident mapreduce on HPC systems. *Object-oriented programming systems languages and applications*, 2014.

[151] M Welsh, D Culler, and E Brewer. SEDA: an architecture for well-conditioned, scalable internet services. *ACM SIGOPS Operating Systems Review*, 2001.

[152] Kyle B. Wheeler, Richard C. Murphy, and Douglas Thain. Qthreads: An API for programming with millions of lightweight threads. In *International Parallel and Distributed Processing Symposium*, pages 1–8. IEEE, apr 2008.

[153] Ben Wiedermann and William R. Cook. Extracting queries by static analysis of transparent persistence. In *Principles of programming languages*, volume 42, page 199, New York, New York, USA, jan 2007. ACM Press.

[154] Ben Wiedermann, Ali Ibrahim, and William R. Cook. Interprocedural query extraction for transparent persistence. *ACM SIGPLAN Notices*, 43(10):19, oct 2008.

[155] Carter Wolfe, Michael Joseph/Shanklin and Leda Ortega. High Performance Compilers for Parallel Computing. jun 1995.

[156] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. Q100: the architecture and design of a database processing unit. *ASPLOS*, 42(1):255–255–255–268–268–268, apr 2014.

[157] Ronald Xin and Josh Rosen. Project tungsten: Bringing spark closer to bare metal. `https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html`, 2015.

[158] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. In *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*, page 3. ACM, 2012.

[159] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance Java dialect. *Concurrency: Practice and Experience*, 10(11):825–836, sep 1998.

[160] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, page 2, Berkeley, CA, USA, 2012. USENIX Association.

[161] Junchao Zhang, Babak Behzad, and Marc Snir. Optimizing the Barnes-Hut algorithm in UPC. *Supercomputing*, page 1, 2011.

[162] Jingren Zhou and Kenneth A. Ross. Buffering database operations for enhanced instruction cache performance. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data - SIGMOD '04*, page 191, New York, New York, USA, jun 2004. ACM Press.

[163] Marcin Zukowski. *Balancing vectorized query execution with bandwidth-optimized storage*. 2009.

[164] Marcin Zukowski, Mark van de Wiel, and Peter Boncz. Vectorwise: A Vectorized Analytical DBMS. In *2012 IEEE 28th International Conference on Data Engineering*, pages 1349–1350. IEEE, apr 2012.

Appendix A

RADISH APPLICATIONSS

This appendix contains examples of applications I have run on RADISH.

A.1 PageRank

The following queries were used as input to RADISH for the case study on handwritten versus generated code.

```
1  alpha = [.85];
2  epsilon = [.0001];
3
4  Edge = scan(public:adhoc:edges);
5
6  -- the distinct and union query to calculate vertex
7  srcs = select src as id, src as ignore from Edge;
8  dsts = select dst as id, dst as ignore from Edge;
9  dups = unionall(srcs, dsts);
10 vertexdedup = select id, max(ignore) from dups; --dedup
11 Vertex = select id from vertexdedup;
12
13 N = [from Vertex emit count(id) as val];
14 min_rank = [(1 - *alpha) / *N];
15
16 OutDegree = [from Edge emit Edge.src as id, count(Edge.dst) as cnt];
17 PageRank = [from Vertex emit Vertex.id as id, 1.0 / *N as rank];
18
19 do
20     -- Calculate each node's outbound page rank contribution
21     PrOut = [from PageRank, OutDegree where PageRank.id == OutDegree.id
22              emit PageRank.id as id, PageRank.rank / OutDegree.cnt as out_rank];
23
24     -- Compute the inbound summands for each node
25     Summand = [from PrOut, Edge, Vertex
26                 where Edge.dst == Vertex.id and Edge.src == PrOut.id
```

```
27                  emit Vertex.id as id, PrOut.out_rank as summand];
28
29      -- Sum up the summands; adjust by alpha
30      NewPageRank = [from Summand emit id as id,
31                     *min_rank + *alpha * sum(Summand.summand) as rank];
32      Delta = [from NewPageRank, PageRank where NewPageRank.id == PageRank.id
33               emit abs(NewPageRank.rank - PageRank.rank) as val];
34      Continue = [from Delta emit max(Delta.val) > *epsilon];
35      PageRank = NewPageRank;
36  while Continue;
37
38  store(PageRank, pageranks);
```

```
1   alpha = [.85];
2   epsilon = [.0001];
3
4   Edge = scan(public:adhoc:edges);
5
6   -- the distinct and union query to calculate vertex
7   srcs = select src as id, src as ignore from Edge;
8   dsts = select dst as id, dst as ignore from Edge;
9   dups = unionall(srcs, dsts);
10  vertexdedup = select id, max(ignore) from dups; --dedup
11  Vertex = select id from vertexdedup;
12
13  N = [from Vertex emit count(id) as val];
14  min_rank = [(1 - *alpha) / *N];
15
16  OutDegree = [from Edge emit Edge.src as id, count(Edge.dst) as cnt];
17  PageRank = [from Vertex emit Vertex.id as id, *min_rank as rank];
18  ActiveVertices = [from PageRank emit id, 1.0 / *N as rankdiff];
19
20  do
21      -- Calculate each node's outbound page rank contribution
22      PrOut = [from ActiveVertices, OutDegree where ActiveVertices.id == OutDegree.id
23               emit ActiveVertices.id as id,
24                    ActiveVertices.rankdiff / OutDegree.cnt as out_rankdiff];
25
26      -- Compute the inbound summands for each node
27      Summand = [from PrOut, Edge, Vertex
28                 where Edge.dst == Vertex.id and Edge.src == PrOut.id
29                 emit Vertex.id as id, PrOut.out_rankdiff as summand];
```

```
30
31      -- Sum up the summands; adjust by alpha
32      NewPageRankDiff = [from Summand emit Summand.id as id,
33                         *alpha * sum(Summand.summand) as rank];
34
35      -- left outer join(PageRank, NewPageRankDiff)
36      NewPageRank = [from PageRank, NewPageRankDiff where PageRank.id @= NewPageRankDiff.id
37                       emit PageRank.id as id, PageRank.rank + NewPageRankDiff.rank as rank];
38
39      Delta = [from NewPageRank, PageRank where NewPageRank.id == PageRank.id
40               emit PageRank.id as id, NewPageRank.rank - PageRank.rank as val];
41      ActiveVertices = [from Delta where abs(Delta.val) > *epsilon emit id, val as rankdiff];
42      PageRank = NewPageRank;
43      Continue = [from Delta emit max(abs(Delta.val)) > *epsilon];
44  while Continue;
45
46  store(PageRank, pageranks);
```

A.2 Naive Bayes

To evaluate runtime on an analytics task, we implemented a naïve Bayes classifier in RADISH in two steps: a training task and a classification task. Both queries first pivot the input into a sparse format (`input-id, feature-index, feature-value`) so that the remainder of the query does not depend on the number of features. The training task is comprised of three SQL queries to compute the conditional probabilities. The classifier task joins the conditional probabilities with the input, and then uses a user-defined aggregate to compute argmax over outcomes and likelihoods.

We use a subset of the Million Song Dataset prepared by the UCI Machine Learning Repository [17]. The task is to predict song year in a $515,345$-song dataset from eight timbre features. Feature values were discretized into intervals of size 10.

Figure A.1 shows strong scaling for the classification query on the Million Song Dataset. The order of the join between sparse inputs and the conditional probabilities table was critical to performance. By building the hash table from the conditional probabilities rather than from the sparsified inputs, each probe finds a constant number of matches (one per

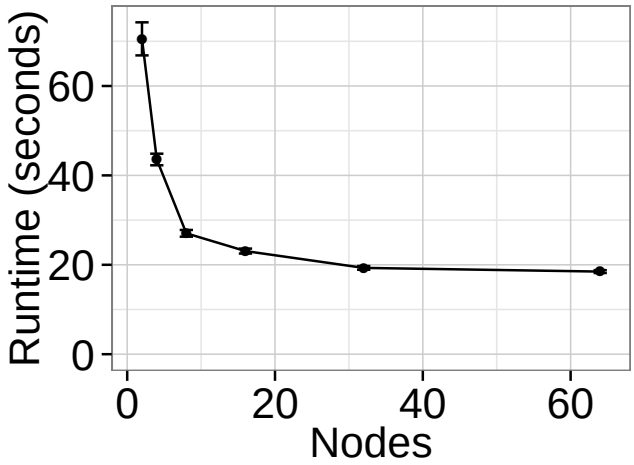


Figure A.1: Strong scaling of RadishX query-generated naive Bayes classification

outcome).

A.3 Additional examples

See `https://github.com/uwescience/sparseMatProjects` for additional examples.