Latency-Tolerant Distributed Shared Memory For Data-Intensive Applications

Jacob Eric Nelson

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

Doctor of Philosophy

University of Washington

2014

Reading Committee:

Luis Ceze, Chair

Mark Oskin, Chair

Simon Kahan

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

Abstract

Latency-Tolerant Distributed Shared Memory For Data-Intensive Applications

Jacob Eric Nelson

Co-Chairs of the Supervisory Committee:
Professor Luis Ceze
Computer Science and Engineering

Professor Mark Oskin
Computer Science and Engineering

Grappa is a modern take on software distributed shared memory (DSM) for in-memory data-intensive applications. Grappa enables users to program a cluster as if it were a single, large, non-uniform memory access (NUMA) machine. Performance scales up even for applications that have poor locality and input-dependent load distribution. Grappa addresses deficiencies of previous DSM systems by exploiting application parallelism, trading off latency for throughput.

We evaluate Grappa with an in-memory map/reduce framework (10× faster than Spark [117]); a vertex-centric framework inspired by GraphLab (1.33× faster than native GraphLab [68]); and a relational query execution engine (12.5× faster than Shark [39]). All these frameworks required only 60-690 lines of Grappa code.

Table of Contents

List of Figures

Acknowledgments

Brandon J. Simmons, for giving me the motivation to finish all of this;

Dipankar Ray and Tosh Kakar, for showing me what was possible;

Brandon Holt and Brandon Myers, for contributing so much to this project;

Andrew Hunter, for helping to get this project started;

Simon Kahan, for never giving up;

Luis Ceze, for patience and for crazy ideas;

Mark Oskin, for an organic, free-range approach;

Joseph Devietti, Brandon Lucia, Hadi Emaseilzadeh, Adrian Sampson, Tom Bergan, Ben Wood, and Emily Fortuna, for being great labmates, coauthors, and friends;

Dan Ports and Irene Zheng, for good food and absurdity;

Ed Lazowska, Hank Levy, Susan Eggers, Jean-Loup Baer, David Notkin, Carl Ebeling, Larry Snyder, and Preston Briggs, for mentorship and for inspiration;

Brad Chamberlain, for a great experience at Cray;

Mike Gunter, for a great experience at Google;

Ken Perrine, for being crazy enough to do grad school too;

Henry Cook and Sarah Bird, for conference fun;

My brother Jamie, for doing something equally interesting but completely different;

My parents Rick and Mary Ann, for starting me down this path;

Matt Kehrt, Aaron Kimball, Alex Colburn, Alex Jaffe, Laura Effinger-Dean, Sam Guarnieri, Ben Birnbaum, Jonathan Hsieh, Kate Smith, Natalie Linnell, Gilbert Bernstein, Kristi Morton, Ricardo Martin, Tony Fader, Abe Friesen, Karl Koscher, Franzi Roesner, Kayur Patel, Yaw Anokwa, Marius Nita, Michael Toomim, Ben Lerner, Travis Kriplean, Neva Durand, Andrew Putnam, Andrew Peterson, Martha Kim, Alan Liu, Benson Limketkai, Paul Pham, Nodira Khoussainova, and Lucas Kreger-Stickles, for making grad school fun;

And many others I've forgotten to include here. Thank you all!

Dedication

To Aquila Sootsprite Simmons-Nelson, for eating my first draft.

Chapter 1

Introduction

Since the first digital computers were built, advances in high performance computer architecture have been driven largely by the needs of two relatively small communities: weapons designers and codebreakers. These communities wanted flexible machines that could provide high performance on their diverse set of problems. For more than 50 years the peculiar demands of these application domains have driven architects to explore many different paradigms for high-performance computation. We've seen fast sequential machines with out-of-order execution [105, 106], vector machines [30, 85, 92], array processors [16, 48], very long instruction word machines [29], multithreaded machines [12,98], and parallel shared-memory machines [44,64,97]. Designers explored many options to obtain maximum performance as well as to improve programmer productivity.

These users had problems whose characteristics varied along multiple axes; we highlight two here that are relevant to this thesis. Some problems were more *compute-intensive*, focused on performing mathematical operations, while others were more *data-intensive*, focused on searching and manipulating large amounts of data. Some problems were more *regular*, with static work distribution and predictable memory access patterns, while others were more *irregular*, with dynamic work

creation and unpredictable access patterns. Good machines provided good performance anywhere on these axes.

The pattern of computer design changed in the 1990's. The economics of the old defense-driven supercomputer industry began to break down as the performance of commodity microprocessors caught up. It no longer made sense to pay for the development of a fast, special-purpose custom processor when a collection of many commodity processors could provide similar performance, no matter what the cost in programmer effort. This led to the development of distributed memory supercomputers [5,23,95] as well as the Beowulf concept of clusters of commodity machines [101]. While this move reduced the development cost of supercomputers, it also severely restricted the design space of supercomputer designers. The community found it too hard to build machines as flexible as they had previously. They focused on one particular class of problems: compute-intensive, regular applications.

In the 2000's, this movement gained momentum with the development of *warehouse-scale computing* [15] in companies such as Google, Facebook, and Microsoft. In order to support applications like web search (including PageRank [81]), advertisement placement, and social network analysis, these companies developed runtime systems running on clusters of commodity components. Today these clusters look the same as modern distributed-memory supercomputers: a collection of multicore nodes connected via a high-bandwidth network. While the hardware is similar to that built in the HPC community, the software stack is different. Scaling up these data-intensive applications requires careful partitioning of data and computation. Programmers must reason about data placement and parallelism explicitly, and for some applications, such as graph analytics, partitioning data is difficult. This has led to a diverse ecosystem of frameworks—MapReduce [34], Dryad [57], and Spark [117] for data-parallel applications, GraphLab [68] for certain graph-based applications, Shark [39] for relational queries, etc. They ease development by specializing to algo-

rithmic structure and dynamic behavior. This means that applications that do not fit well into one particular model may suffer in performance.

This thesis was born out of the idea that we could obtain better performance on these applications by revisiting two old ideas in supercomputing design: distributed shared memory and latency tolerance. The fully custom Tera MTA [11, 12] computer is a classic example of providing a large shared memory to applications by using concurrency to hide latencies. It had a large distributed shared memory with no caches. On every clock cycle, each processor would execute a ready instruction chosen from one of its 128 hardware thread contexts, a sufficient number to fully tolerate memory access latency. The network was designed with a single-word injection rate that matched the processor clock frequency and sufficient bandwidth to sustain a reference from every processor on every clock cycle. Unfortunately, the MTA's relatively low single-threaded performance meant that was it not general enough nor cost-effective.

Software distributed shared memory (DSM) systems provide shared memory abstractions for clusters. Historically, these systems [19, 25, 61, 66] performed poorly, largely due to limited inter-node bandwidth, high inter-node latency, and the design decision of piggybacking on the virtual memory system for seamless global memory accesses. Past software DSM systems were largely inspired by symmetric multiprocessors (SMPs), attempting to scale that programming mindset to a cluster. However, applications were only suitable for them if they exhibited significant locality, limited sharing and coarse-grain synchronization—a poor fit for many modern data-analytics applications. Recently there has been a renewed interest in DSM research [35, 72], sparked by the widespread availability of high-bandwidth low-latency networks with remote memory access (RDMA) capability.

In this paper we describe Grappa, a software DSM system for commodity clusters designed for data-intensive applications. Grappa is inspired by the MTA. Like the MTA, instead of relying on

locality to reduce the cost of memory accesses, Grappa depends on *parallelism* to keep processor resources busy and hide the high cost of inter-node communication. Grappa also adopts the shared-memory, fine-grained parallel programming mindset from the MTA. To support fine-grained messaging like the MTA, Grappa includes an overlay network that combines small messages together into larger physical network packets, thereby maximizing the available bisection bandwidth of commodity networks. This communication layer is built in user-space, utilizing modern programming language features to provide the global address space abstraction. Efficiencies come from supporting sharing at a finer granularity than a page, avoiding the page-fault trap overhead, and enabling compiler optimizations on global memory accesses.

The runtime system is implemented in C++ for a cluster of x86 machines with an InfiniBand interconnect, and consists of three main components which will be covered in greater detail later: a global address space (section 3.1), lightweight user-level tasking (section 3.2), and an aggregating communication layer (section 3.3). We demonstrate the generality and performance of Grappa as a common runtime by implementing three domain-specific platforms on top of it: a simple in-memory MapReduce framework; a vertex-centric API (i.e. like GraphLab); and a relational query processing engine. Comparing against GraphLab itself, we find that a simple, randomly partitioned graph representation on Grappa performs 2.5× better than GraphLab's random partitioning and 1.33× better than their best partitioning strategy, and scales comparably out to 128 cluster nodes. The query engine built on Grappa, on the other hand, performs 12.5× faster than Shark on a standard benchmark suite. The flexibility and efficiency of the Grappa shared-memory programming model allows these frameworks to co-exist in the same application and to exploit application-specific optimizations that do not fit within any existing model.

```
// distributed input array
GlobalAddress<char> chars = load_input();

// distributed hash table:
using Cell = std::map<char,int>;
GlobalAddress<Cell> cells = global_alloc<Cell>(ncells);

forall(chars, nchars, [=](char& c) {
  // hash the char to determine destination
  size_t idx = hash(c) % ncells;
  delegate(&cells[idx], [=](Cell& cell)
  { // runs atomically
    if (cell.count(c) == 0) cell[c] = 1;
    else cell[c] += 1;
  });
});
```

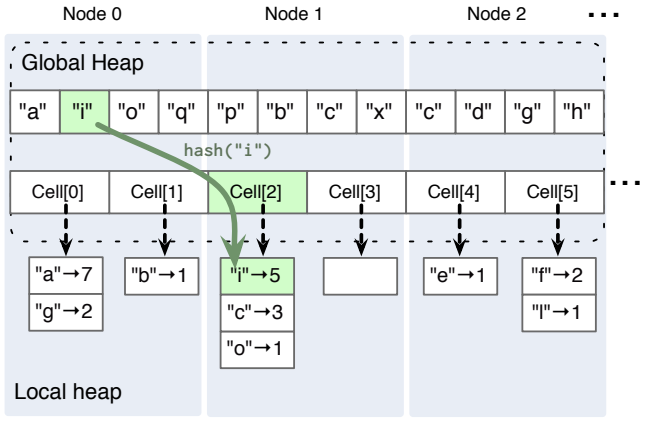

Figure 1.1: "Character count" with a simple hash table implemented using Grappa's distributed shared memory.

1.1 Frameworks for Data-Intensive Applications

Analytics frameworks—such as MapReduce, graph processing and relational query execution—are typically implemented for distributed private memory systems (clusters) to achieve scale-out performance. While implementing these frameworks in a shared-memory system would be straightforward, this has generally been avoided because of scalability concerns. We argue that modern data-intensive applications have properties that can be exploited to make these frameworks run efficiently and scale well on *distributed* shared memory systems.

Figure 1.1 shows a minimal example of implementing a "word count"-like application in actual Grappa DSM code. The input array, `chars`, and output hash table, `cells`, are distributed over multiple nodes. A parallel loop over the input array runs on all nodes, hashing each key to its cell and incrementing the corresponding count atomically. The syntax and details will be discussed in

later sections, but the important thing to note is that it looks similar to plain shared-memory code, yet spans multiple nodes and, as we will demonstrate in later sections, scales efficiently.

Here we describe how three data-intensive computing frameworks map to a DSM, followed by a discussion of the challenges and opportunities they provide for an efficient implementation.

MapReduce Data parallel operations like map and reduce are simple to think of in terms of shared memory. Map is simply a parallel loop over the input (an array or other distributed data structure). It produces intermediate results into a hash table similar to that in Figure 1.1. Reduce is a parallel loop over all the keys in the hash table.

Vertex-centric GraphLab is an example of a vertex-centric execution model, designed for implementing machine-learning and graph-based applications [43,68]. The latest version uses a 3-phase gather-apply-scatter (GAS) API for vertex programs to enable several optimizations pertinent to natural graphs. Such graphs are difficult to partition well, so algorithms traversing them exhibit poor locality. Each phase can be implemented as a parallel loop over vertices, but fetching each vertex's neighbors results in many fine-grained data requests.

Relational query execution Decision support, often in the form of relational queries, is an important domain of data-intensive workloads. All data is kept in hash tables stored in a DSM. Communication is a function of inserting into and looking up in hash tables. One parallel loop builds a hash table, followed by a second parallel loop that filters and probes the hash table, producing the results. These steps rely heavily on consistent, fine-grained updates to hash tables.

The key challenges in implementing these frameworks on a DSM are the following.

Small messages. Programs written to a shared memory model tend to access small pieces of data, which when executing on a DSM system lead to small inter-node messages. What were load or store operations become complex transactions involving small messages over the network. Conversely, programs written using a message passing library, such as MPI, expose this complexity to programmers, and hence encourage them to optimize it.

Poor locality. As previously mentioned, data-intensive applications often exhibit poor locality. For example, how much communication GraphLab's gather and scatter operations conduct is a function of the graph partition. Complex graphs frustrate even the most advanced partitioning schemes [43]. This leads to poor spatial locality. Moreover, which vertices are accessed varies from iteration to iteration. This leads to poor temporal locality.

Need for fine-grain synchronization. Typical data-parallel applications offer coarse-grained concurrency with infrequent synchronization; for instance, between phases of processing a large chunk of data. Conversely, graph-parallel applications exhibit fine-grain concurrency with frequent synchronization; for instance, when done processing work associated with a single vertex. Therefore, for a DSM solution to be general, it needs to support fine-grain synchronization efficiently.

Fortunately, data-intensive applications have properties that can be exploited to make DSMs efficient.

Concurrency. Data-intensive applications have abundant data parallelism. However, different applications have different shapes of parallelism and synchronization. Some are more coarse-grained and are amenable to bulk-synchronous approaches; this style of parallelism is relatively easy to exploit. Other applications have fine-grained tasks and dependences which place much

heavier demands on syncronization and scheduling. Without careful design scheduling overheads can dominate the application runtime.

Latency tolerance. Performance isn't dependent on the *latency* of execution of any specific parallel task/thread, as it would be in for example a web server, but rather the aggregate execution (i.e., *throughput*) of *all* tasks/threads.

This thesis describes a system that exploits these application properties to implement an efficient distributed shared memory that overcomes the challenges previously described.

Chapter 2

Commodity hardware: performance challenges and opportunities

Modern commodity hardware has excellent performance, as long as the computation it executes fits the assumptions of the hardware designers. In this chapter we explore the properties of an example commodity cluster and investigate some of the bottlenecks in order to understand what these assumptions are, and what challenges we must overcome to obtain good performance on computations that don't fit these assumptions.

2.1 Structure of a commodity cluster

Distributed memory clusters consist largely of two components: the individual shared-memory nodes and a network. Commodity clusters use mass-market parts for both of these components. The nodes use standard server processors, and the network uses widely-available, standardized hardware. This is in contrast to purpose-built supercomputers, which may use either standard or purpose-built processors but always with a specialized network.

Traditionally, the level of integration was also different between the two types of clusters: commodity machines used standard components in standard racks, and high-end supercomputers used specialized form factors that allowed for increased density. However, with the growth of warehouse-scale computing, this differentiation has become less clear: racks of nodes built by Google or Facebook for their datacenters (for example, the Open Compute Project [84] racks) follow a non-standard form factor, but still use commodity processors and network components.

2.1.1 Description of our target hardware

In order to explore the challenges and opportunities afforded by commodity hardware in detail, we will focus on a single platform. Figure 2.1 shows the cluster that we used for these experiments. It uses a commonly-available processor, the 6-core Intel Westmere Xeon X5650, and a commonly-available cluster network, Mellanox ConnectX-2 InfiniBand. There are 12 nodes, each of which has two processors and 24GB of DRAM.

Figure 2.2 shows the structure of our nodes. The Westmere Xeon is the second generation of Intel processor to include an on-chip memory controller. Each processor has three channels of registered DDR3 DRAM running at 1333 MHz in a shared address space. The processors are linked by Intel's QPI interconnect, making this a NUMA shared memory system. Cache lines are 64 bits. There are three levels of data cache. Each core has a private 32 KB L1 and 256 KB L2, and all the cores in a socket share a 12 MB L3. The L3 is inclusive, whereas the L2 is not.

2.2 Memory system performance

To perform well on irregular analytics problems. processors must support high throughput random access to memory. We evaluate this using a pointer-chasing benchmark shown in Figure 1. This

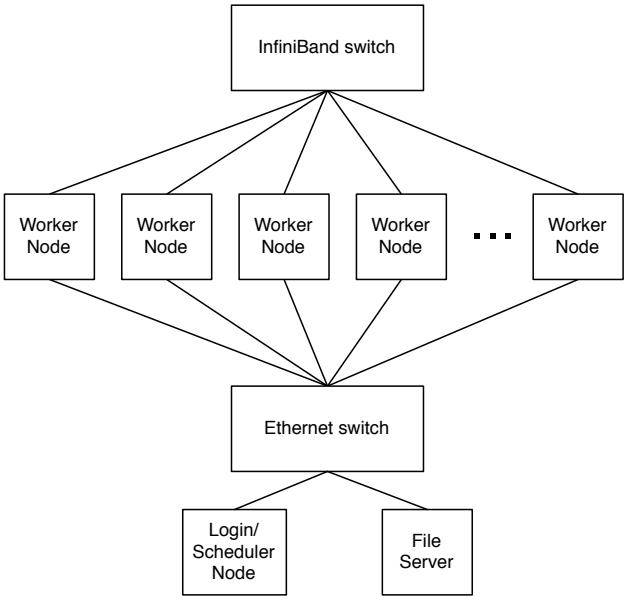

Figure 2.1: Schematic of cluster used for limit experiments.

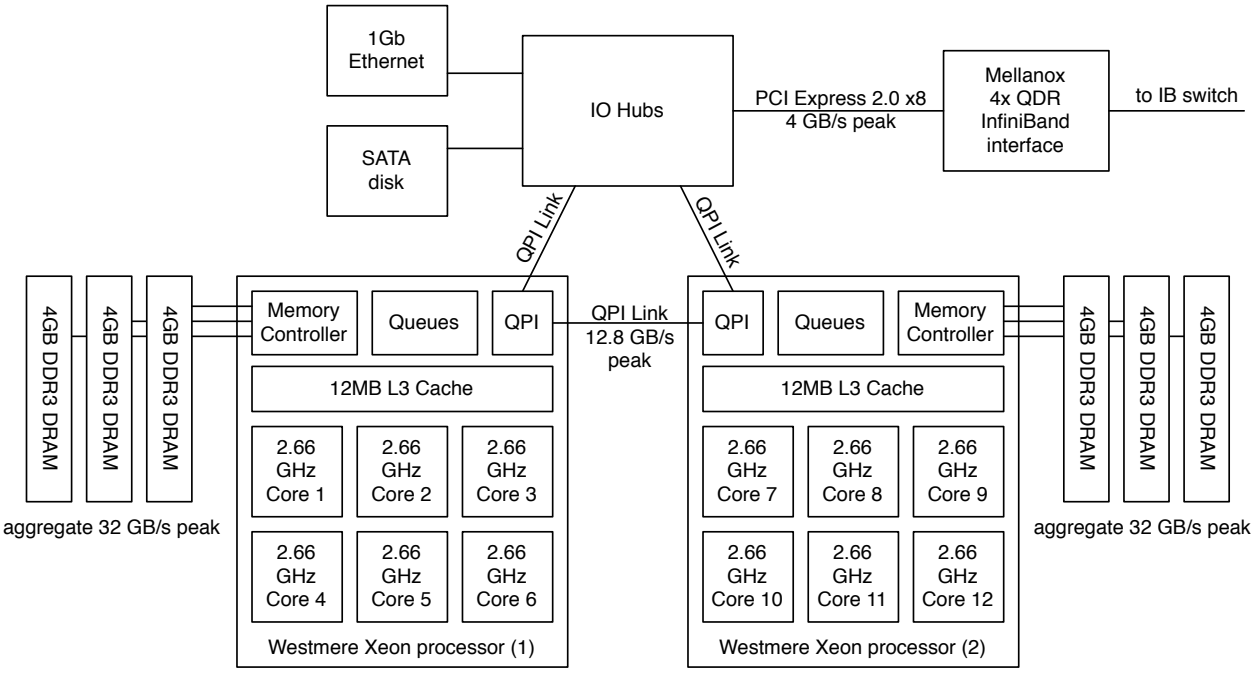

Figure 2.2: Schematic of cluster node used for limit experiments.

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

Listing 1: Pseudocode for inner loop of pointer chasing benchmark.

benchmark lays out a collection of linked lists of cacheline-sized records in memory in order to traverse them concurrently. The layout may be sequential or shuffled, allowing the evaluation of sequential or random-access memory performance. Each core in this system has one traversal thread bound to it, which traverses up to N lists simultaneously. No context-switching occurs here; for $N > 1$ the only mechanisms for exploiting concurrency are the threads running on different cores and the out-of-order execution engine of each core exploiting the memory concurrency of each iteration of the while loop.

We investigate three questions here. First, what fraction of the theoretical peak DRAM bandwidth is achievable in the best case, with sequential accesses? Second, what fraction is achievable in the worst case, with random accesses? Finally, what limitations are there in the memory system that keep us from reaching these peak results?

Figure 2.3 shows the maximum sequential memory bandwidth available to varying numbers of cores accessing a single NUMA domain's memory. The core IDs in this figure match those in Figure 2.2. All memory for this experiment was allocated in socket 1's DRAM, and all voltage and frequency scaling was disabled, along with hyperthreading. Each core traverses a single list ($N = 1$ in Figure 1); since the lists are laid out sequentially for this experiment, the Westmere's hardware prefetchers are able to stream in the data into the core's cache, minimizing misses.

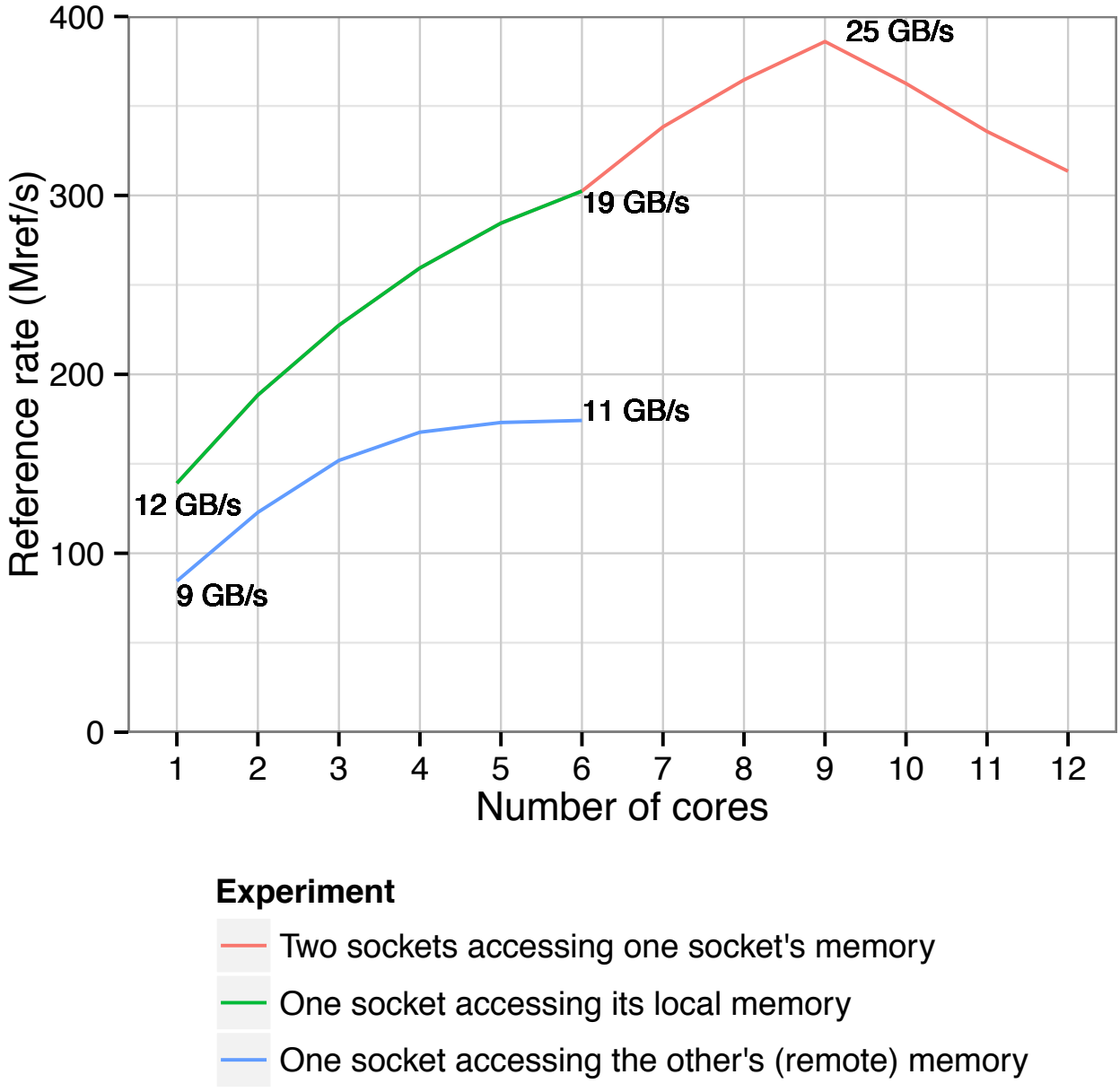

Figure 2.3: Sequential memory access performance between two sockets.

Three different experiments are shown in the figure: bandwidth from socket 1's cores to its local DRAM, bandwidth from all the cores to socket 1's DRAM, and bandwidth from socket 2's cores to socket 1's DRAM.

We observe a number of results in this plot. First, the peak bandwidth achievable from a single memory controller is approximately 80% of the theoretical maximum. This limitation is likely due to bank switching overheads in the DRAM which keep the DRAM buses from being utilized continuously: while each thread is able to exploit spatial locality with its prefetchers and caches, the memory controller must deal with concurrent references from each thread to different parts of the DRAM. The throughput falloff as more than 9 cores are used may be due to increased demand for bank switching due to the additional concurrent list traversals.

Second, the six local cores are not able to reach the maximum bandwidth of their local memory controller on their own; requests from cores on the other are required to reach this peak. We postulate that this is a design decision intended to avoid starvation when NUMA-unaware applications allocate memory on a socket whose cores are fully utilized.

Finally, we see that the cores on socket 2 are able to nearly saturate the QPI link when referencing data in socket 1's memory. It is likely that the difference between the theoretical and achieved bandwidth here is due to framing overheads on the QPI link.

2.2.1 Random access

The previous experiment was able to exploit significant locality. How does the memory system perform when there is no locality to be exploited? Figure 2.4 shows the maximum random-access memory bandwidth available to varying numbers of cores accessing memory within a single socket. Because the list entries are permuted randomly in this experiment none of the hardware prefetchers are able to activate.

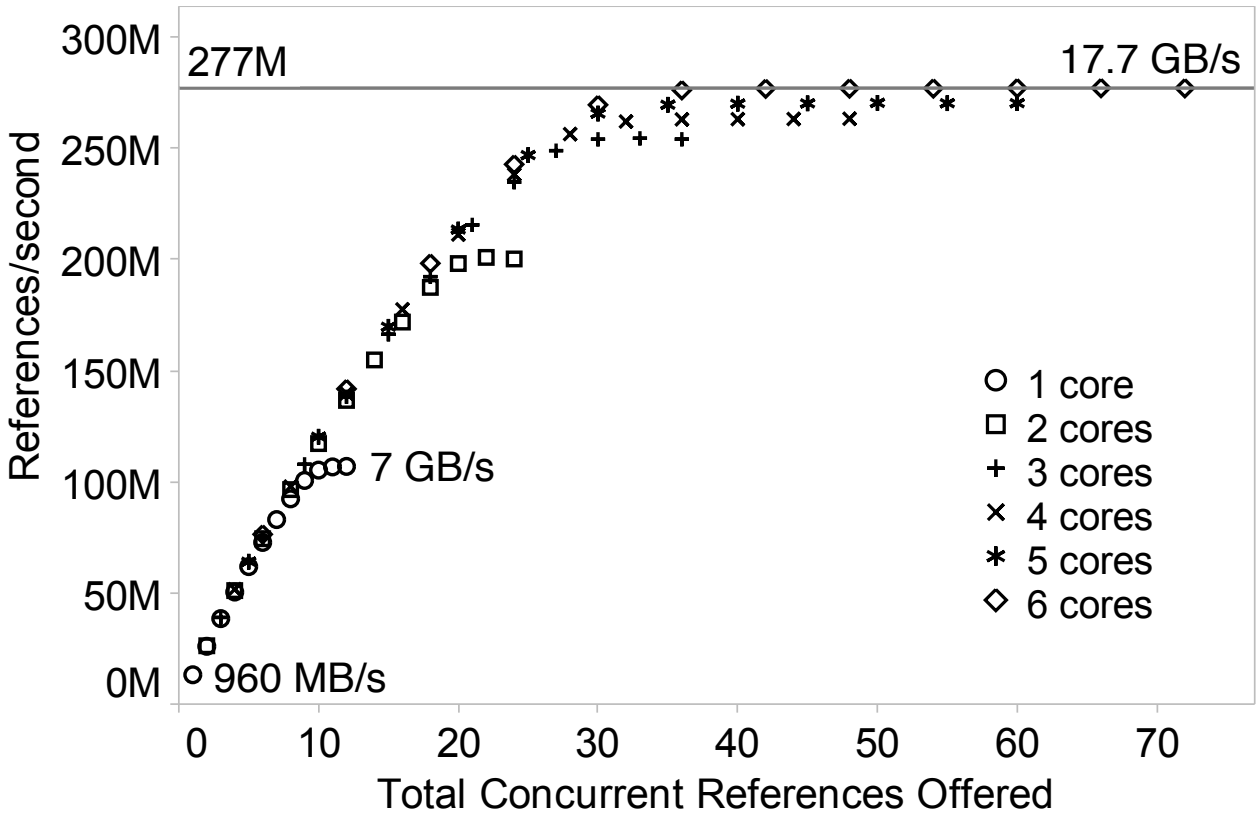

Figure 2.4: Random access to socket-local memory with varying number of concurrent references per core.

This plot is laid out differently than the previous one: the x-axis denotes the total number of concurrent references across all cores in the system. Different shapes show different core counts, and all cores for a given setting traverse the same number of lists per core ((*N* in Figure 1) concurrently.

We observe two key results in this plot. First, the total random access bandwidth is 17.7 GB/s, only slightly lower than the 19 GB/s limit in the sequential case. We reach this limit with only 6 concurrent references per core, for a total of 36 concurrent references in the socket. We saw higher aggregate performance when using cores from the second socket to issue references to this socket's memory as well, although we do not show that data here. Together, these facts suggest that the random reference rate is not limited by DRAM but in fact by a component of the Westmere's memory system. As discussed in [55, 56, 65, 104], all memory references leaving a socket's cores must be enqueued in a set of queues in the uncore, collectively referred to as the *global queue*. This structure has only 32 entries for in-socket read operations, which imposes a hard limit on total concurrent accesses, and thus total random access bandwidth.

The second important result is that the per-core random reference rate is limited to something significantly below the whole socket random reference rate. We see that once we have reached 10 concurrent references on a single core little increase in random-access bandwidth is possible. This is confirmed by the Intel documentation: each core has 10 line fill buffers which are used for all data cache misses (including prefetches), limiting the total memory concurrency per core.

2.2.2 Controlling cache contents

The traditional view of caches is that they are a largely transparent way to provide high-bandwidth, low-latency access to memory. Caches provide this abstraction by exploiting locality; for applications that exhibit little locality, the low-locality accesses may conflict with data that has locality,

causing useful data to be evicted. These situations could be avoided if low-level programmers had the ability to explicitly control the movement of data into and out of caches, as well as the placement of data in caches. This would allow the caches to be used as scratchpad memories with deterministic performance.

There are three cache operations that would benefit low-level programmers:

Explicit data movement. The ability to move data into and out of the processor without disturbing the cache would provide programmers with a mechanism to complete no-locality accesses while leaving data with locality untouched.

Explicit data placement. The ability to place data in a programmer-controlled place in the cache would provide programmers with a mechanism to support phase-based or streaming applications so that some data from one phase can be locked in the cache without being evicted by another phase or by streaming accesses.

Presence detection. The ability to test whether a particular line is in the cache would allow programmers to overlap data movement into caches with computation efficiently, making sure that prefetched data is in cache before touching it.

Unfortunately, these operations are not currently practical to implement. Querying the state of the cache in an out-of-order processor would yield speculative results which would be difficult to utilize and which would add complexity to one of the processor's critical paths. Instead, processor manufacturers provide a set of instructions with *non-temporal hints* with the intention that the processor will do something smart to minimize cache pollution. The exact mechanism is not well-specified and may change from generation to generation.

In our processors, there are three classes of non-temporal instructions. The first example is the prefetch instruction `prefetchnta`. When it is executed, its argument will be loaded into the L1 cache, but may be quickly evicted. From personal discussions with Intel engineers, we suspect that the way this is implemented in our processors is that the data is placed in the L1 cache without updating the LRU information, so that the next miss that maps to that set in the cache will evict it. This instruction is easy to use because it has no logical effect on a program's execution; if it fetches a line that is evicted before being used, or if it attempts to access a word which is mapped but swapped out to disk, it fails silently. Any later accesses to that line will act like an ordinary cache miss.

The second class is the non-temporal load instructions like `movnta` and `movntdqa`. These load a word (of 8 or 16 bytes, respectively) from memory into a register in the processor's SIMD unit without allocating space for it in the cache. Unfortunately this requires the source memory region to be mapped as *write-combining*, a special mode intended for IO devices that disables ordinary caching. This makes these instructions difficult to use in ordinary code.

The third class is the non-temporal store instructions like `movntq` and `movntdq`. These write a word (of 8 or 16 bytes, respectively) from a register in the processor's SIMD unit to DRAM without allocating space in any caches. Unlike the non-temporal loads, these instructions operate just fine on ordinary cacheable memory; if a non-temporal write hits in a cache, the line is modified and evicted from the cache. We have found these instructions very difficult to use in a high performance way: for the best performance `movntq` instructions must be issued consecutively for all the words in a cache line, with no other instructions interleaved. If any instructions do interleave of if all the words of the line are not written the processor appears to fall back to a slow multi-cycle read-modify-write mode.

2.3 Network performance

To support irregular and graph applications, Grappa must support high random access rates across the cluster. This places heavy demands on the cluster's network. This section discusses three challenges in getting good performance from cluster networks.

We focus here on the network technology known as InfiniBand. InfiniBand has historically provided a number of features which make it attractive for high-performance computing: high bandwidth, low latency, and support for remote access to nodes' memories, bypassing the CPU. These features are not unique to InfiniBand—they are provided both by Cray's proprietary networks and by modern versions of Ethernet—but these other technologies also share the same challenges in getting irregular application performance.

InfiniBand was created around the turn of the millennium, as system designers were approaching the electrical limits of parallel buses, and networks of servers were becoming widely used [52]. InfiniBand sought to unify the IO interfaces of servers. Rather than having networking (Ethernet) and storage (Fiber Channel) adapters plugged into a PCI bus that was connected to a CPU and memory, the CPU and memory would have only an InfiniBand interface. Disks would plug directly into the InfiniBand network, and long-distance networks would be connected with bridges attached to the edge of the InfiniBand network.

In practice, InfiniBand has rarely been used as this sort of universal bus. Instead, people have used it primarily for networking in high-performance computers. The spec was written to make it easy to build cheap, low-latency switches, and vendors were able to market high-performance InfiniBand networks before Ethernet vendors could provide similar levels of performance. Today, InfiniBand is the most common interconnect on the Top500 list [107], accounting for over 40 percent of systems.

```
1       int a[BIG] = {0};        // array to be incremented
2
3       int b[n];                // index array
4       initialize_randomly(b);
5
6       for (int i=0; i<n; ++i) {
7         a[b[i]]++;
8       }
```

Listing 2: Pseudocode for GUPS benchmark.

InfiniBand implements a number of features developed in other high-performance networks in the past. InfiniBand is a user-mode network [59, 110] and implements a reliable channel abstraction in firmware [100]. InfiniBand cards have the ability to execute memory operations at remote nodes [103], which is referred to as RDMA.

A common tool used to evaluate cluster performance is the Giga-Updates Per Second (GUPS) benchmark. This benchmark allocates a large array on each node of the cluster, and then all nodes perform a simple computation (usually an increment) on randomly-chosen words of the arrays on randomly-chosen nodes. Figure 2 shows pseudocode for a single-node version of the benchmark.

Implementing a cluster version of this benchmark using InfiniBand is straightforward. All that is necessary to perform a single iteration is to enqueue an InfiniBand RDMA increment message to a randomly-generated node and address on that node. The requesting node's InfiniBand network adapter will send the request to the remote node and wait for an acknowledgment; the remote nodes' adapter will execute the increment and send a completion notification.

Our overall goal for the cluster network is to support high rates of randomly-addressed small messages like those in the GUPS benchmark. Providing high performance for large messages to few destinations is easy; providing high performance for small messages to many destinations is hard. With support for both we broaden the class of applications our system can handle and make

programmers' lives easier. However, there are three key challenges that get in the way of this goal. We explore these challenges in the context of our development hardware: a cluster connected with 40 gigabit QDR InfiniBand provided by Mellanox ConnectX-2 adapters.

2.3.1 Framing overhead

Consider the RDMA-based GUPS benchmark discussed previously. Each increment message carries a 64-bit address long with a 64-bit operand. In a system purpose-built for doing increments, these 16 bytes might be all that were needed to be sent to a remote node to do an increment. With a network with a 40 gigabit signaling rate, this would give a theoretical maximum increment rate of 312.5 million per second per node, comfortably above the per-socket local random reference rate measured previously.

However, getting this payload to the remote node reliably requires the packet to include additional information: delimiters for the start and end of the packet, addressing information, RDMA protocol headers, and CRCs to detect data corruption. In the InfiniBand network these fields add another 40 bytes of overhead to a message, making the minimum size increment packet 56 bytes. This is over three times larger than the minimum required data for the operation. Now the theoretical maximum increment rate is only 89.3 million per second per node.

In addition, our InfiniBand network encodes all data at the physical level with an 8b/10b code [113] in order to obtain more robust electrical properties. This code sends 10 bits of data on the wire for every 8 bits of user data. This adds 20% overhead, limiting the theoretical maximum increment rate to 71.4 million per second per node.

But InfiniBand is not the only network over which the data must travel to reach a remote memory controller. InfiniBand cards in our system are connected to the processor over a PCI Express (PCIe) [96] bus. PCIe is in fact not a bus, but another switched network, with its own signal

encoding and framing overhead. Our system uses 8 lanes of PCIe 2.0 to connect the InfiniBand interface. At the electrical level this provides 40 gigabits per second of bandwidth, but like our InfiniBand network it is encoded with an 8b/10b code. PCIe also adds is own frame delimeters, addressing, and reliability information while moving the data, adding an additional 24 bytes. PCIe packets Now the theoretical maximum increment rate is 50 million per second per node.

2.3.2 Message injection rate

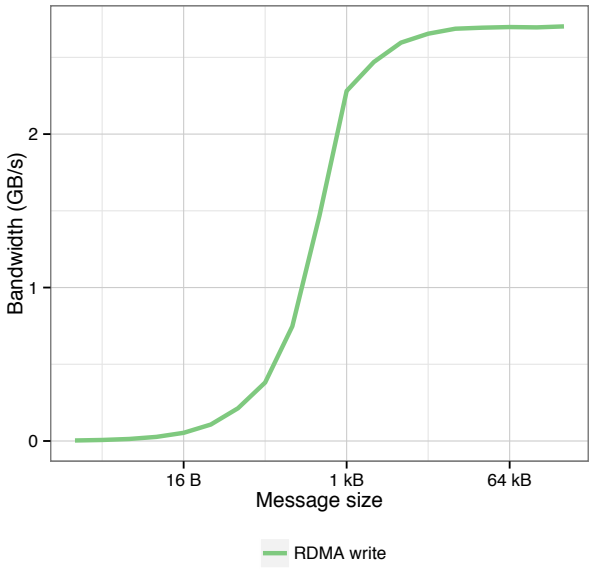

Figure 2.5: RDMA write performance between two cluster nodes. For small messages, only a fraction of the network bandwidth is able to be used.

Our InfiniBand cards are not very good at sending small messages. This is common among commodity networks: the majority are optimized for applications moving data in large, kilobyte-sized packets. For messages like our 16-byte increment, only a small fraction of the network bandwidth is able to be used. Figure 2.5 illustrates the magnitude of this problem. This experiment

has one node sending a stream of RDMA writes to another node, varying the size of the region written. We can see that for a 16-byte message like that of our hypothetical increment benchmark, only 5% of the peak network bandwidth is able to be used. This point corresponds to 3.4 million write operations per second.

This is not a fundamental limit; newer versions of the cards support higher message rates. But there are a number of challenges in making networks like InfiniBand perform well for small messages. The primary challenge is this: moving data over PCIe is expensive. A single round trip may take hundreds of nanoseconds to complete; we estimate 250 ns for our cluster's nodes. While sending a message ought to involve filling out a descriptor and writing it into the card's address space in a single PCIe bus transaction, InfiniBand cards often need to access additional data to send a message.

First, InfiniBand sends messages using a scatter-gather list approach. For very small messages the data can be inlined into the initial send request, but larger messages require the card to issue one or more reads over PCIe to get the data to send.

Second, InfiniBand's reliable delivery requires it to update a sequence number and other state whenever sending a packet. This data is stored on a per-connection basis in the system's DRAM. The card contains a cache, but for large jobs, once the number of active destinations exceeds the cache size the card must refill the cache using memory operations over the PCIe bus.

Finally, InfiniBand's RDMA capability requires it to copy memory protection information into messages it sends, and to check the message against protection data on the receiving side. Again, this data is cached in the card, and for large numbers of memory regions cache misses will require additional PCIe round trips.

Counting PCIe round trips and multiplying by PCIe latency is not sufficient to compute maximum message rate. Message sends should be able to be pipelined, and so multiple concurrent PCIe

transactions could be executing simultaneously. However, the PCIe interfaces on the CPU and in the card must contain logic and storage to track outstanding requests that require an acknowledgement or response (referred to as *non-posted* in PCIe). For the Intel 5500 IO hub chipset in our cluster, storage for CPU-to-perhipheral transactions is allocated based on the width of the PCIe link to a perhipheral: a x4 link supports two, a x8 link supports four, and a x16 link supports 8, according to the specification [53]. Our network cards use a x8 link, and so four concurrent CPU-to-perhipheral transactions are supported. Concurrent transactions from perhipheral to DRAM are counted differently; the maximum in our system is not clear but may be up to 8.

2.4 Conclusion

The performance of commodity hardware is impressive. Given sufficient concurrency, the memory system of our processors handles random access nearly as well as it does sequential access. Given large enough packets, the throughput of our network is close to peak. The challenge in exploiting this hardware for irregular applications is the mismatch between application properties and the hardware properties. To obtain good performance on irregular applications, we must design a runtime that allows us to expose sufficient concurrency to make random access tolerable, and to convert small messages into large ones to make good use of the network.

Chapter 3

Grappa Design

Figure 3.1 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 system provides fine-grain access to data anywhere in the system. Every piece of global memory is owned by a particular core in the system. Access to data on remote nodes is provided by *delegate* operations that run on the owning core. Delegate operations may include normal memory operations such as *read* and *write* as well as synchronizing operations such as *fetch-and-add* [44]. Due to delegation, the memory model offered is similar to what underpins C/C++ [22, 58], so it is familiar to programmers.

Tasking system The tasking system supports lightweight multithreading and global distributed work-stealing—tasks can be stolen from any node in the system, which provides automated load balancing. Concurrency is expressed through user-level threads which are scheduled cooperatively. Threads that perform long-latency operations (i.e., remote memory access) automatically suspend while the operation is executing and wake up when the operation completes.

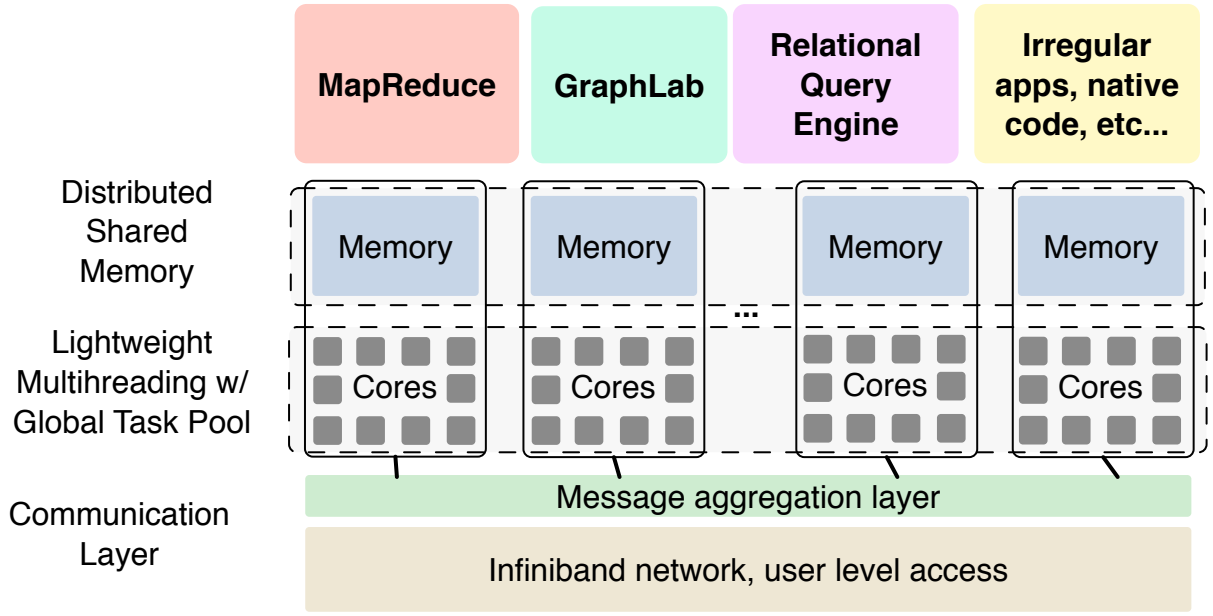

Figure 3.1: Grappa design overview

Communication layer The main goal of our communication layer is to aggregate small messages into large ones. This process is invisible to the application programmer. Its interface is based on active messages [111]. Since aggregation and deaggregation of messages needs to be very efficient, we perform the process in parallel and carefully use lock-free synchronization operations. For portability, we use MPI [71] as the underlying messaging library as well as for process setup and tear down.

3.1 Distributed Shared Memory

Below we describe how Grappa implements a shared global address space and the consistency model it offers.

3.1.1 Addressing Modes

Local memory 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 conventional pointers. Applications use local accesses for a number of things in Grappa: the stack associated with a task, accesses to global memory from the memory's home core, and accesses to debugging infrastructure local to each system node. Local pointers cannot access memory on other cores, and are valid only on their home core.

Global memory addressing Grappa allows any local data on a core's stacks or heap to be exported to the global address space to be made accessible to other cores across the system. This uses a traditional PGAS (partitioned global address space [38]) addressing model, where each address is a tuple of a rank in the job (or global process ID) and an address in that process.

Grappa also supports *symmetric* allocations, which allocates space for a copy (or proxy) 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 constant data, or allowing operations to be transparently buffered. A separate publication [51] explored how this was used to implement Grappa's synchronized global data structures, including vector and hash map.

Putting it all together Figure 3.2 shows an example of how global, local and symmetric heaps can all be used together for a simple graph data structure. In this example, vertices are allocated from the global heap, automatically distributing them across nodes. Symmetric pointers are used to access local objects which hold information about the graph, such as the base pointer to the vertices,

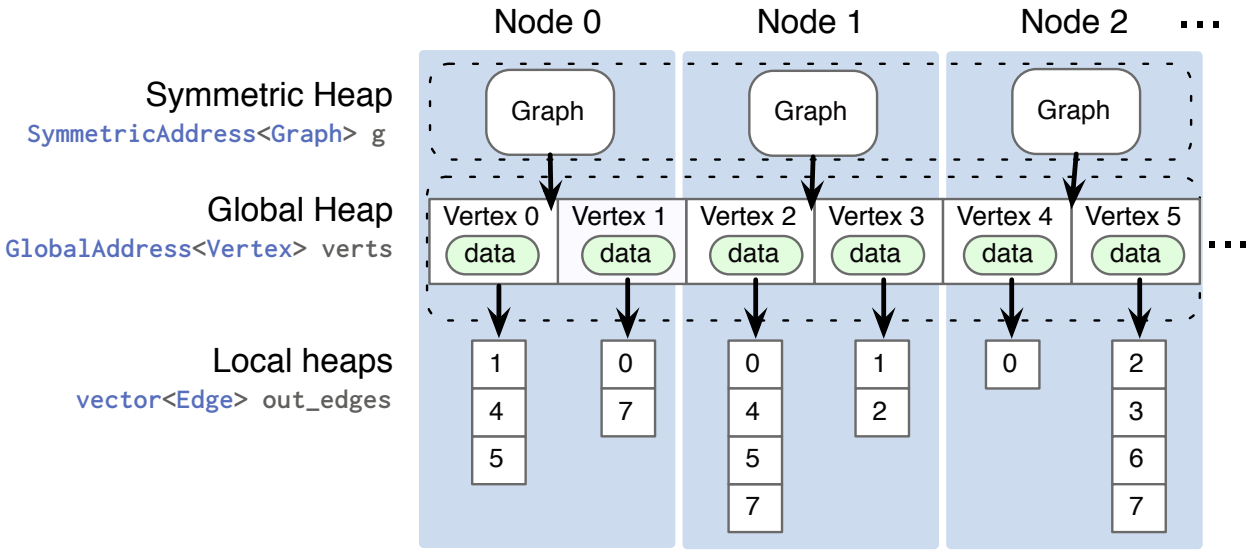

Figure 3.2: Using global addressing for graph layout.

from any core without communication. Finally, each vertex holds a vector of edges allocated from their core's local heap, which other cores can access by going through the vertex.

3.1.2 Delegate Operations

Access to Grappa's distributed shared memory is provided through *delegate* operations, which are short operations performed at the memory location's home node. When the data access pattern has low locality, it is more efficient to modify the data on its home core rather than bringing a copy to the requesting core and returning a modified version. Delegate operations [69, 76] provide this capability. While delegates can trivially implement *read*/*write* operations to global memory, they can also implement more complex *read-modify-write* and synchronization operations (e.g., *fetch-and-add*, mutex acquire, queue insert). Figure 3.3 shows an example.

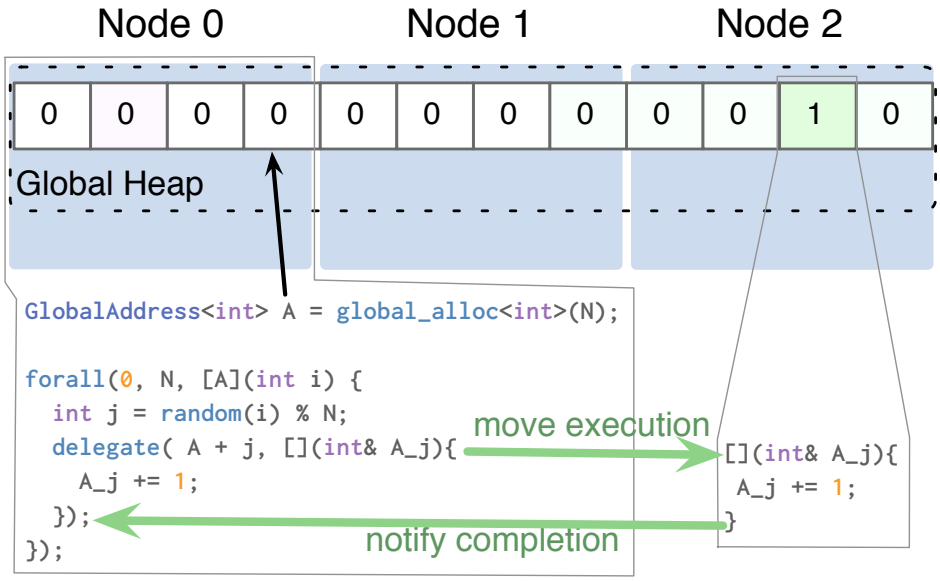

Figure 3.3: Grappa delegate example.

Delegate operations must be expressed explicitly to the Grappa runtime, a change from the traditional DSM model. In practice, even programmers using implicit DSMs had to work to express and exploit locality to obtain performance. In other work we have developed a compiler [50] that automatically identifies and extracts productive delegate operations from ordinary code.

A delegate operation can execute arbitrary code provided it does not lead to a context switch. This guarantees atomicity for all delegate operations. To avoid context switches, a delegate must only touch memory owned by a single core. A delegate is *always* executed at the home core of the data addresses it touches. We limit delegate operations to operate on objects stored on a single core so they can be satisfied with a single network request. Given these restrictions, we can ensure that delegate operations for the same address from multiple requesters are always serialized through a single core in the system, providing atomicity with strong isolation. A side benefit is that atomic operations on data that are highly contended are faster. When programmers want to operate on data

structures spread across multiple nodes, accesses must be expressed as multiple delegate operations along with appropriate synchronization operations.

3.1.3 Memory Consistency Model

Accessing global memory though delegate operations allows us to provide a familiar memory model. All synchronization is done via delegate operations. 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 [47], with their updates visible to all cores across the system in the same order. In addition, only one synchronous delegate will be in flight at a time from a particular task, i.e., synchronization operations from a particular task are not subject to reordering. Moreover, once one core is able to see an update from a synchronous delegate, all other cores are too. 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 guarantee a memory model that offers sequential consistency for data-race-free programs [6] (all accesses to shared data are separated by synchronization), which is what underpins C/C++ [22,58]. The synchronous property of delegates provides a clean model but is restrictive: we discuss asynchronous operations within the next section.

3.2 Tasking System

Each hardware core has a single operating system thread pinned to it; all Grappa 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 Grappa*worker* thread that is scheduled within an operating system thread; we refer to

these as workers to avoid confusion with operating system threads. Scheduling between tasks is carried out entirely in user-mode without operating system intervention.

Tasks Tasks are specified by a closure (also referred to as a functor or "function object" in C++) 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 the parentheses operator, or a C++11 lambda construct. These objects, typically very small (on the order of 64 bytes), hold read-only values such as an iteration index and pointers to common data or synchronization objects. Task closures can be serialized and transported around the system, and are 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 mapped to a worker it stays with that worker until it finishes.

Scheduling During execution, a worker yields control of its core whenever performing a long-latency operation, allowing the processor to remain busy while waiting for the operation to complete. In addition, a programmer can direct scheduling explicitly. To minimize context-switch overhead, the Grappa scheduler operates entirely in user-space and does little more than store state of one worker and load that of another. When a task encounters a long-latency operation, its worker is suspended and subsequently woken when the operation completes.

Each core in a Grappa system has 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.

Whenever a task yields, the scheduler makes a decision about what to do next. First, any task in the system task queue whose deadline is imminent is chosen for execution. Second, the scheduler determines if any workers with running tasks are ready to execute; if so, one is scheduled. Finally, if no workers are ready to run, but tasks are waiting to be matched with workers, an idle worker is woken (or a new worker is spawned), matched with a task, and scheduled.

Context switching Grappa context switches between workers non-preemptively. As with other cooperative multithreading systems, we treat context switches as function calls, saving and restoring only the callee-saved state as specified in the x86-64 ABI [13] rather than the full register set required for a preemptive context switch. This requires 62 bytes of storage.

Grappa's scheduler is designed to support a very large number of concurrently-active workers—so large, in fact, that their combined context data will 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, we establish a pipeline of ready worker references in the scheduler. This pipeline consists of *ready-unscheduled*, *ready-scheduled*, and *ready-resident* stages. 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 queue itself must also be in cache. In a FIFO schedule, 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 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. Determining the exact working set might be difficult, but we find that approximating the working set with the top 2-3 cache lines of the stack is the best naive heuristic. The worker data is *ready-resident* when it arrives in cache. Since the arrival of a prefetched cache line is generally not part of the architecture, we must determine the latency from profiling.

At our standard operating point on 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 determine this by calculating total data movement based on switch rate and cache lines per switch in a microbenchmark. As a reference point, for the same yield test using kernel-level Pthreads on a single core, the switch time is 450ns for a few threads and 800ns for 1000–32000 threads.

Work stealing When the scheduler finds no work to assign to its workers, it commences to steal tasks from public queues of other cores. It chooses a node at random until it finds one with a non-zero amount of work in its public task queue. The scheduler steals half of the tasks it finds at that node. Work stealing is particularly interesting in Grappa since performance depends on having many active workers on each core. Even if there are many active workers, if they are all suspended on long-latency operations, then the core is underutilized. The stealing policy must predict whether local tasks will likely generate enough new work soon; a similar problem is addressed in [109].

Because global memory accesses involve the core where the memory is physically located, in some cases it is a profitable programming decision to bind tasks to a particular core ahead of time. Such tasks are *not* stealable, however, potentially leading to load imbalance.

Expressing parallelism The Grappa API supports spawning individual tasks, with optional data locality constraints. These tasks may run as 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. Asynchronous delegates are treated as task spawns in the memory model.

For better programmability, tasks are automatically generated from parallel loop constructs, as in Figure 1.1. Grappa's parallel loops spawn tasks using a *recursive decomposition* of iterations, similar to Cilk's cilk_for construct [21], and TBB's `parallel_for` [87]. This generates a logarithmically-deep tree of tasks, stopping to execute the loop body when the number of iterations is below a user-definable threshold.

Grappa loops can iterate over an index space or over a region of shared memory. In the former case, tasks are spawned with no locality constraints, and may be stolen by any core in the system. In the latter case, tasks are bound to the home core of the piece of memory on which they are operating so that the loop body may optimize for this locality, if available. The local region of memory is still recursively decomposed so that if a particular loop iteration's task blocks, other iterations may run concurrently on the core.

3.3 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 aggregation for better communication bandwidth.

Active message interface At the upper (user-level) layer, Grappa implements asynchronous active messages [111]. Our active messages are simply a C++11 lambda or other closure. We take advantage of the fact that our homogeneous cluster hardware 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 aggregation Since communication is very frequent in Grappa, aggregating and sending messages efficiently is very important. To achieve that, Grappa makes careful use of caches, prefetching, and lock-free synchronization operations.

Figure 3.4 shows the aggregation process. Cores keep their own outgoing message lists, with as many entries as the number of system cores in a Grappa system. These lists are accessible to all cores in a Grappa node to allow cores to 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 referenced only twice for each message sent: once when the message is created, and (much later) when the message is serialized for transmission. The pool allocator prefetches the buffers with the non-temporal flag to minimize cache pollution.

Each processing core in a given system node is responsible for aggregating and sending the resulting messages from all cores on that node to a set of destination nodes. Cores periodically

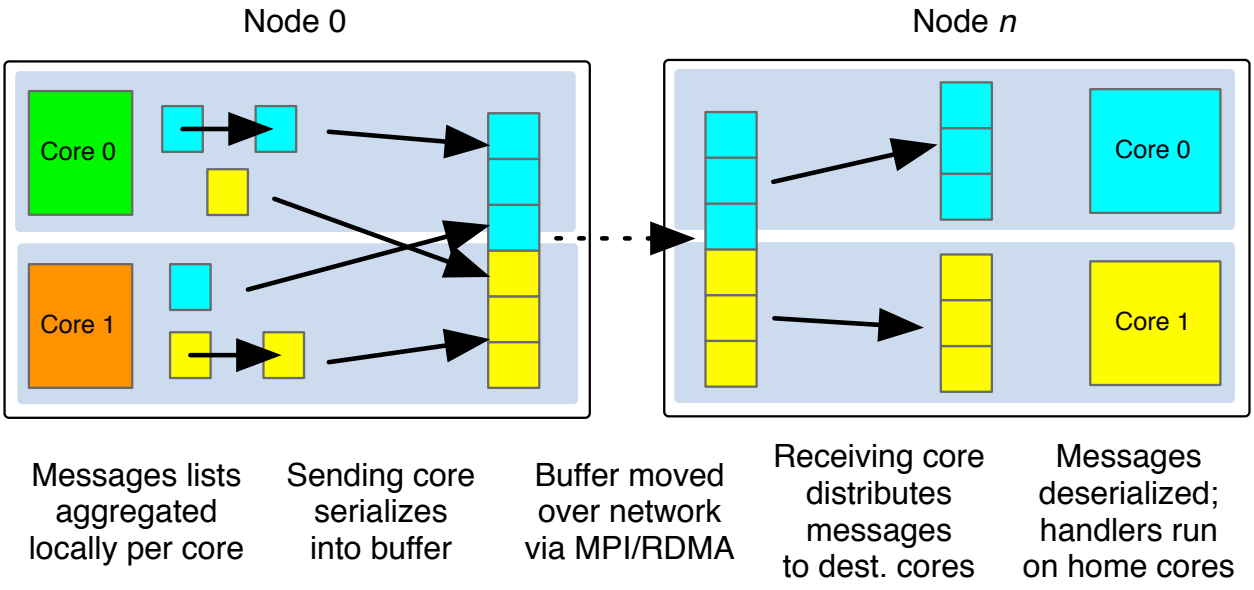

Figure 3.4: Message aggregation process.

execute a system task that examines the outgoing message lists for each destination node for which the core is responsible; if the list is long enough or a message has waited past a time-out period, all messages to a given destination system node from that source system node are sent by copying them to a buffer visible to the network card. Actual message transmission can be done purely in user-mode using MPI, which in turn uses RDMA.

The final message assembly process involves manipulating several shared data-structures (the message lists), so it uses CAS (compare-and-swap) operations to avoid high synchronization costs. This traversal requires careful prefetching because most of the outbound messages are *not* in the processor cache at this time (recall that a core can be aggregating messages originating from other cores in the same node). Note that we use a per-core array of message lists that is only periodically modified across processor cores, having experimentally determined that this approach is faster (sometimes significantly) than a global per-system node 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 simultaneously unpack messages destined for that core. Upon completion, these unpacking tasks synchronize with the management task. 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.

3.3.1 Why not just use native RDMA support?

Given the increasing availability and decreasing cost of RDMA-enabled network hardware, it would seem logical to use this hardware to implement Grappa's DSM. Figure 3.5 shows the performance difference between native RDMA atomic increments and Grappa atomic increments using the GUPS cluster-wide random access benchmark using the cluster described in chapter 4. The cluster has Mellanox ConnectX-2 40Gb InfiniBand cards connected through a QLogic switch with no oversubscription. The RDMA setting 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 setting issued delegate increments in a parallel for loop. Both settings perform increments to random locations in a 32 GB array of 64-bit integers distributed across the cluster. Figure 3.5(left) shows how aggregation allows Grappa to exceed the performance of the card by 25× at 128 nodes. We measured the effective bisection bandwidth of the cluster as described in [49]: for GUPS, performance is limited by memory bandwidth during aggregation, and uses $\sim 40\%$ of available bisection bandwidth.

Figure 3.5(right) illustrates why using RDMA directly is not sufficient. 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, when the wire-rate

limit is over 76 million [52]. We believe this limitation is primarily due to the latency of the multiple PCI Express round trips necessary to issue one operation; a similar problem was studied in [42]. Furthermore, RDMA network cards have severely limited support for synchronization with the CPU [35, 72]. Finally, framing overheads can be large: InfiniBand 8-byte RDMA writes moves 50 bytes on the wire; Ethernet-based RDMA using RoCE moves 98 bytes. Work is ongoing to improve network card small message performance [2, 4, 36, 42, 78, 83, 89, 110]: even if native small message performance improves in future hardware, our aggregation support will still be useful to minimize cache line movement, PCI Express round trips, and other memory hierarchy limitations.

3.4 Fault tolerance discussion

A number of recent "big data" workload studies [28, 88, 91] suggest that over 90 percent of current analytics jobs require less than one terabyte of input data and run for less than one hour. We designed Grappa to support this size of workload on medium-scale clusters, with tens to hundreds of nodes and a few terabytes of main memory. At this scale, the extreme fault tolerance found in systems like Hadoop is largely wasted — e.g., assuming a per-machine MTBF of 1 year, we would estimate the MTBF of our 128-node cluster to be 2.85 days.

We could add checkpoint/restart functionality to Grappa, either natively or using a standard HPC library [37]. Writing a checkpoint would take on the order of minutes; for instance, our cluster can write all 8 TB of main memory to its parallel filesystem in approximately 10 minutes. It is important to balance the cost of taking a checkpoint with the work lost since the last checkpoint in the event of a failure. We can approximate the optimum checkpoint interval using [116]; assuming a checkpoint time of 10 minutes and a per-machine MTBF of 1 year, we should take checkpoints every 4.7 hours. These estimates are similar to what Low et al. measured for Graphlab's checkpoint

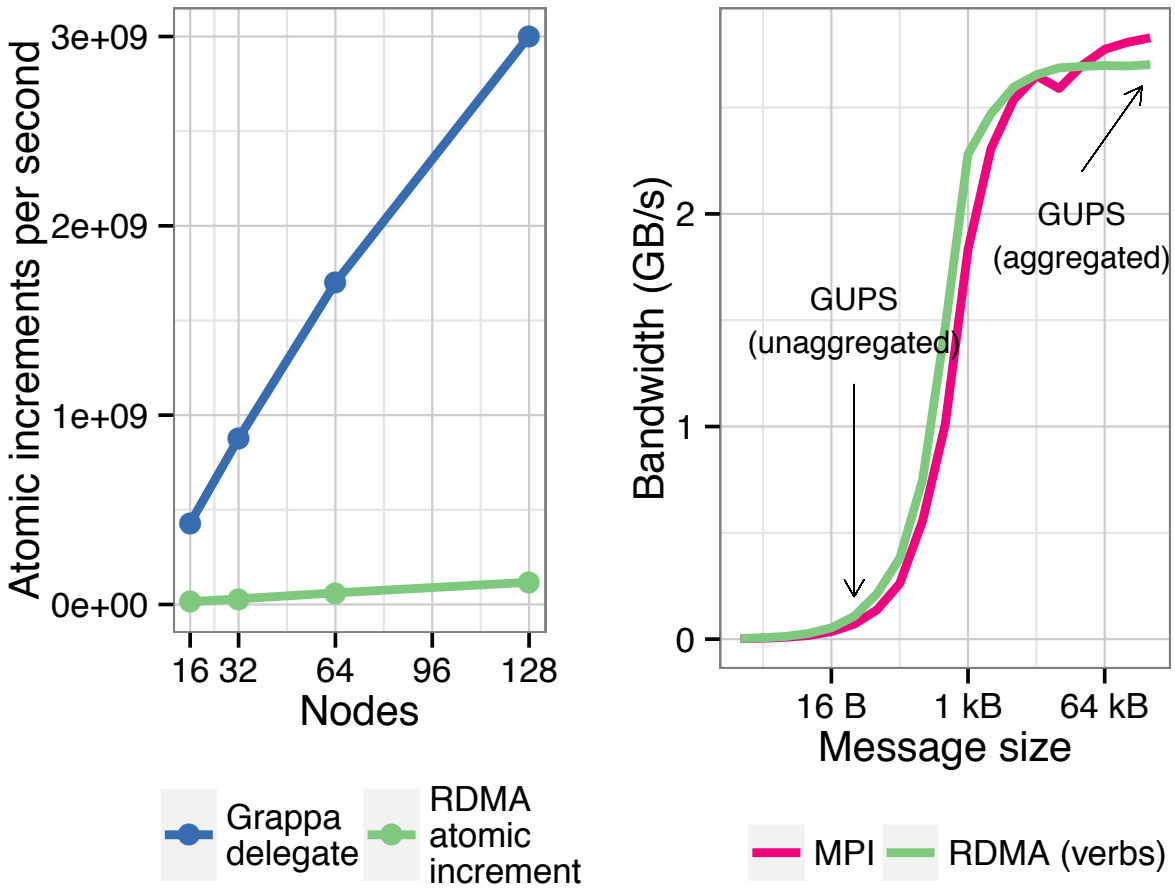

Figure 3.5: On the left, random updates to a billion-integer distributed array with the GUPS benchmark. On the right, ping-pong bandwidth measured between two nodes.

mechanism in [68]. In this regime, it is likely cheaper to restart a failed job than it is to pay the overhead of taking checkpoints and recovering from a failure.

Given these estimates, we chose not to implement fault tolerance in this work. Adding more sophisticated fault tolerance to Grappa for clusters with thousands of nodes is an interesting area of future work.

Chapter 4

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 40 gigabit 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 in Connected mode.

4.1 Vertex-centric Programs on Grappa

We implemented a vertex-centric programming framework in Grappa with most of the same core functionality as GraphLab [43, 68] using the graph data structure provided by the Grappa library (Figure 3.2). Unlike GraphLab we do not focus on intelligent partitioning, instead choosing 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

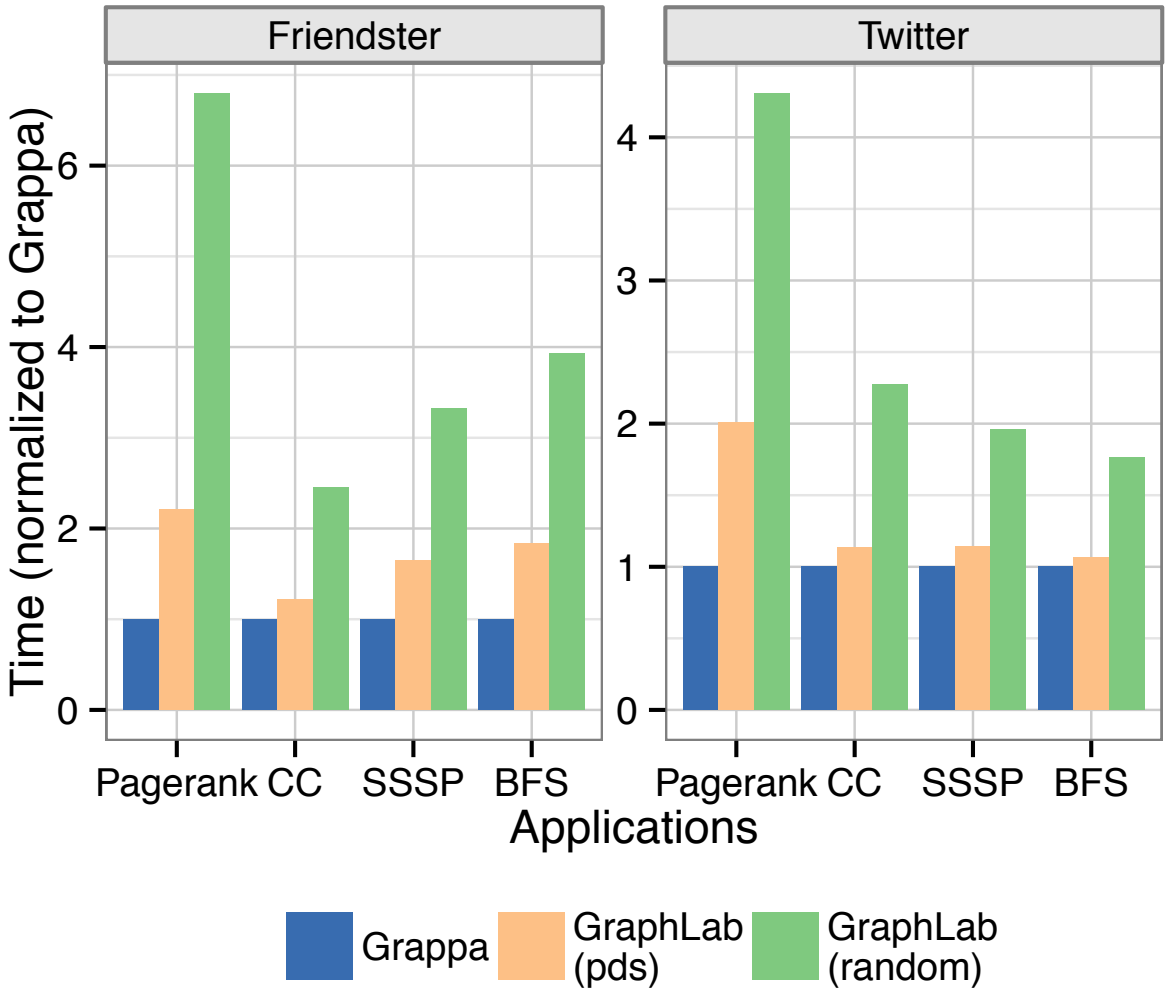

Figure 4.1: Vertex-centric performance of Grappa compared to GraphLab with random and with optimized PDS partitioning strategies, showing time to converge (same number of iterations) normalized to Grappa, on the Twitter and Friendster datasets, running on 31 nodes.

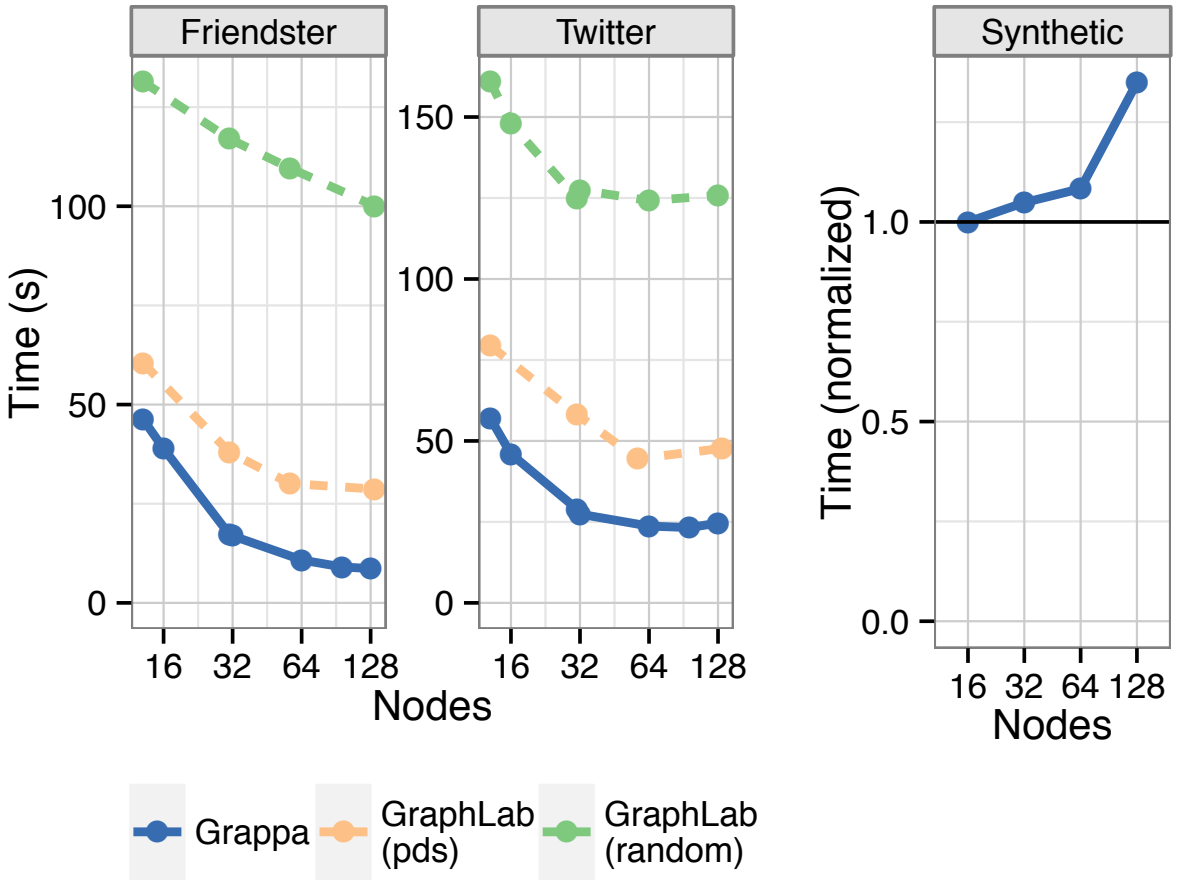

Figure 4.2: Vertex-centric scaling results for PageRank out to 128 nodes. On the left we use the Friendster and Twitter datasets to measure *strong* scaling, which is limited in this case by the size of the datasets. We compare Grappa with GraphLab using both their random and optimized PDS partitioning strategies. On the right we use synthetic power-law graphs scaled proportionally with cluster node count to measure *weak* scaling.

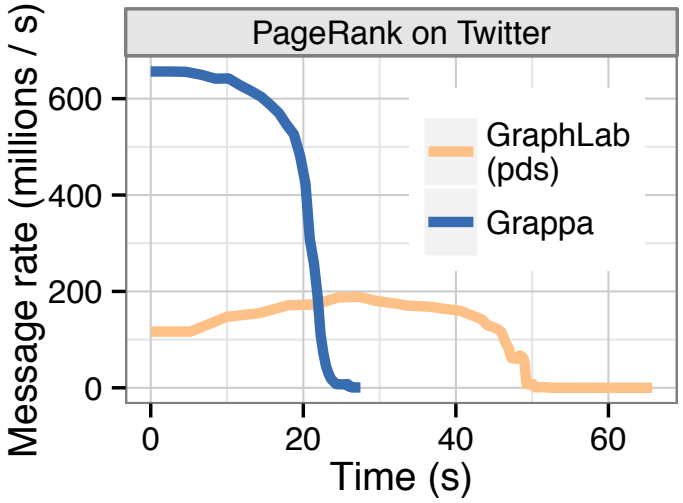

Figure 4.3: Cluster-wide message rates (average per iteration) while computing PageRank on 31 nodes, comparing Grappa with GraphLab using their PDS partitioning strategy.

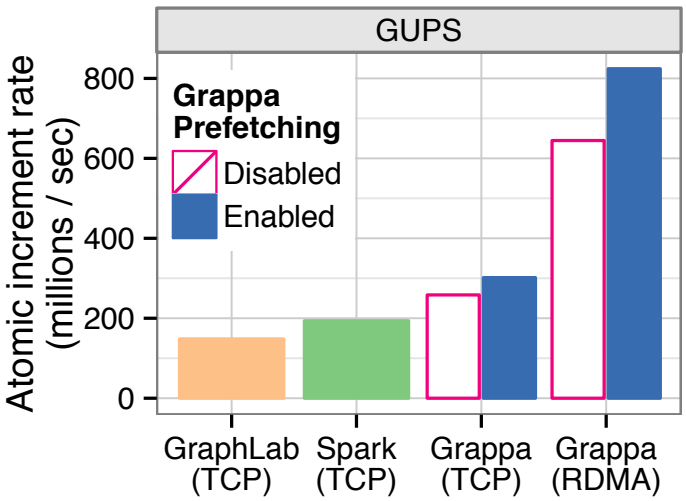

Figure 4.4: GUPS message rates for GraphLab, Spark, and Grappa on 31 nodes. Grappa is shown using both TCP-based and RDMA-based configuration, with message prefetching on and off.

engine, including the delta caching optimization, in ~60 lines of Grappa code. Parallel iterators are defined 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 three graph analytics applications using vertex program definitions equivalent to GraphLab's: PageRank, Single Source Shortest Path (SSSP), and Connected Components (CC). In addition, we implemented a simple Breadth-first search (BFS) application in the spirit of the Graph500 benchmark [45], which finds a "parent" for each vertex with a given source. The implementation in the GraphLab API is similar to the SSSP vertex program.

4.1.1 Performance

To evaluate Grappa's Vertex-centric framework implementation, we ran each application on the Twitter follower graph [62] (41 M vertices, 1 B directed edges) and the Friendster social network [115] (65 M vertices, 1.8 B undirected edges). For each we run to convergence – for PageRank we use GraphLab's default threshold criteria – resulting in the same number of iterations for each. Additionally, for PageRank we ran with delta caching enabled, as it proved to perform better. 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. Most of the comparisons are done at 31 nodes for this reason.

Figure 4.1 depicts performance results at 31 nodes, normalized to Grappa's execution time. We can see that Grappa is faster than random partitioning on all the benchmarks (on average 2.57×), and 1.33× faster than the best partitioning, despite not replicating the graph at all. Both implementations of PageRank issue application-level requests on the order of 32 bytes (mostly communicating updated rank values). However, since these would perform terribly on the network, both systems aggregate updates into larger wire-level messages. Grappa's performance exceeds that of GraphLab primarily because it does this faster.

Figure 4.4 explores this difference using the GUPS benchmark from subsection 3.3.1. 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 IPoIB and the aggregators make 64KB batches. At 31 nodes, GraphLab's aggregator 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 is an upper bound 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 can be seen in Figure 4.3, 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. Overall, this leads to a 2× speedup.

Figure 4.5 demonstrates the connection between concurrency and aggregation over time while executing PageRank. We see that at each iteration, the number of concurrent tasks spikes as *scatter*

delegates are performed on outgoing edges, which leads to a corresponding spike in bandwidth due to aggregating the many concurrent messages. At these points, Grappa achieves roughly 1.1 GB/s per node, which is 47% of peak bisection bandwidth for large packets discussed in subsection 3.3.1, or 61% of the bandwidth for 80 kB messages, the average aggregated size. This discrepancy is due to not being able to aggregate packets as fast as the network can send them, but is still significantly better than unaggregated bandwidth.

Figure 4.2(left) shows strong scaling results on both datasets. As we can see, scaling is poor beyond 32 nodes for both platforms, due to the relatively small size of the graphs—there is not enough parallelism 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 4.2(right). This experiment runs 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. Runtime is normalized to show distance from ideal scaling (horizontal line), showing that scaling deteriorates less than 30% at 128 nodes.

4.2 Relational queries on Grappa

We used Grappa to build a distributed backend to Raco, a relational algebra compiler and optimization framework [86]. Raco supports a variety of relational query language frontends, including SQL, Datalog, and an imperative language, MyriaL. It includes an extensible relational algebra optimizer and various intermediate query plan representations.

We compare performance of our system to that of Shark, a fast implementation of Hive (SQL-like), built upon Spark. We chose this comparison point because Shark is optimized for in-memory execution and performs competitively with parallel databases [114].

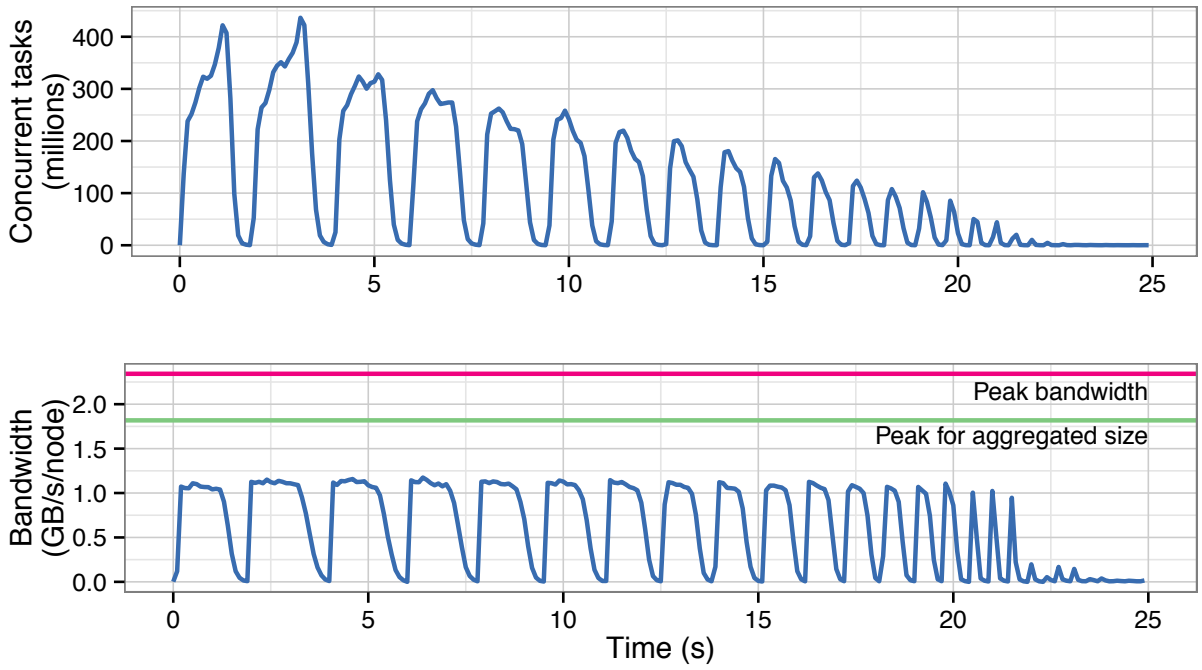

Figure 4.5: Grappa PageRank execution over time on 32 nodes. The top shows the total number of concurrent tasks (including delegate operations), over the 85 iterations, peaks diminishing as fewer vertices are being updated. The bottom shows message bandwidth per node, which correlates directly with the concurrency at each time step, compared against the peak bandwidth, and the bandwidth for the given message size.

Our particular approach for the Grappa backend to Raco is source-to-source translation. We generate `forall`s for each pipeline in the physical query plan. We extend the code generation approach for *serial* code in [77] to generating parallel shared memory code. The generated code is sent through a normal C++11 compiler.

All data structures used in query execution (e.g. hash tables for joins) are globally distributed and shared. While this a departure from the shared-nothing architecture of nearly all parallel databases, the locality-oriented execution model of Grappa makes the execution of the query virtually identical to that of traditional designs. We expect (and later demonstrate) that Grappa will excel at hash joins, given that it achieves high throughput on random access.

Implementing the parallel Grappa code generation was a relatively simple extension of the generator for serial C++ code that we use for testing Raco. It required less than 90 lines of template C++/Grappa code and 600 lines of support and data structure C++/Grappa code to implement conjunctive queries, including two join implementations.

4.2.1 Performance

We focus on workloads that can be processed in memory, since storage is out of scope for this work. For Grappa, we scan all tables into distributed arrays of rows in memory, then time the query processing. To ensure all timed processing in Shark is done in memory, we use the methodology that Shark's developers use for benchmarking [3]. In particular, all input tables are cached in memory and the output is materialized to an in-memory table. The number of reducer tasks for shuffles was set to 3 per Spark worker, which balances overhead and load balance. Each worker JVM was assigned 52GB of memory.

We ran conjunctive queries from SP^{2}Bench [93]. The queries in this benchmark involve several joins, which makes it interesting for evaluating parallel in-memory systems. We show results on

16 nodes (we found Shark failed to scale beyond 16 nodes on this data set) in Figure 4.6. Grappa has a geometric mean speedup of 12.5× over Shark. The benchmarks vary in performance due to differences in magnitude of communication and output size.

There are many differences between the two runtime systems (e.g. messaging layers, JVM and native) and the query processing approach (e.g. iterators vs compiled code), making it challenging to clearly understand the source of the performance difference between the two systems. To do so, we computed a detailed breakdown (Figure 4.7) of the execution of Q2. We took sample-based profiles of both systems and categorized CPU time into five components: *network* (low-level networking overheads, such as MPI and TCP/IP messaging), *serialization* (aggregation in Grappa, Java object serialization in Shark), *iteration* (loop decomposition and iterator overheads), *application* (actual user-level query directives), and *other* (remaining runtime overheads for each system).

Overall, we find that the systems spend nearly the same amount of CPU time in application computation, and that more than half of Grappa's performance advantage comes from efficient message aggregation and a more efficient network stack. An additional benefit comes from iterating via Grappa's compiled parallel for-loops compared to Shark's dynamic iterators. Finally, both systems have other, unique overheads: Grappa's scheduling time is higher than Shark due to frequent context switches, whereas Shark spends time dynamically checking the types of data values.

Shark's execution of these queries appears to place bursty demands on the network, and is sensitive to network bandwidth. On query Q2, Shark achieves the same peak bandwidth as GUPS (Figure 4.3) sustains (200MB/s/node), but its sustained bandwidth is just over half this amount (116 MB/s/node).

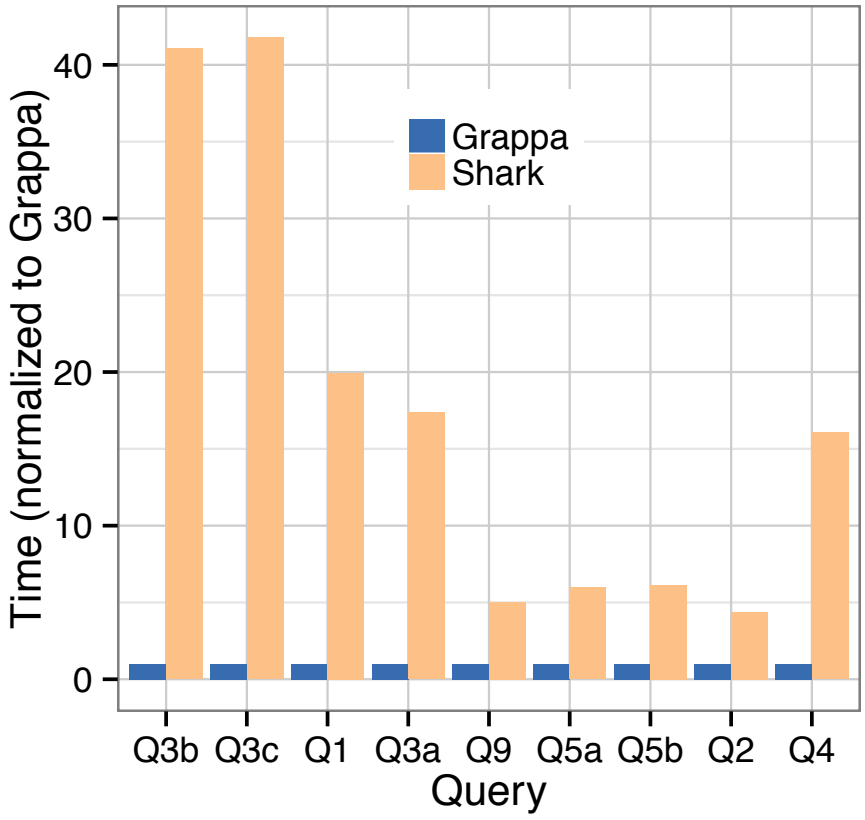

Figure 4.6: The SP^{2}Bench benchmark on 16 nodes. Query Q4 is a large workload so it was run with 64 nodes.

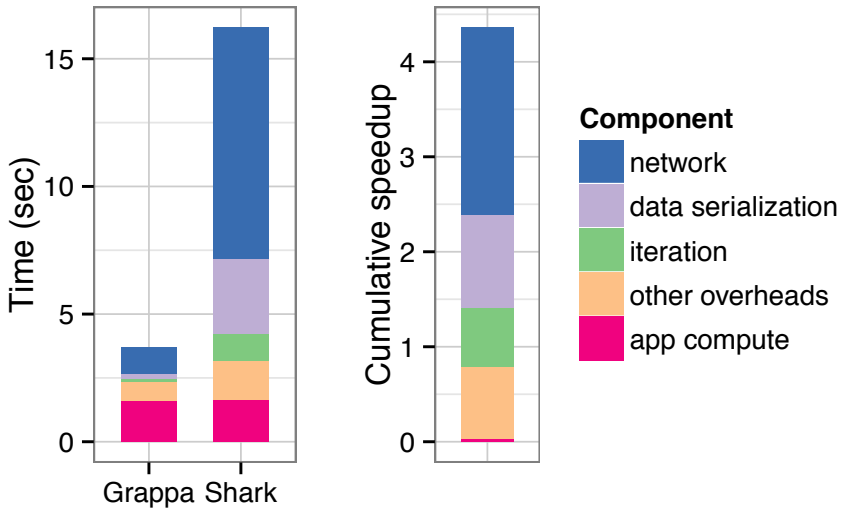

Figure 4.7: Performance breakdown of speedup of Grappa over Shark on Q2. Both systems spend the same time doing computation; Grappa's increased performance comes primarily from its more efficient use of the network, with an additional benefit from its use of compiled queries, which show up as iteration and serialization time for Shark.

4.3 Iterative MapReduce on Grappa

We experiment with data parallel workloads by implementing an in-memory MapReduce API in 152 lines of Grappa code. The implementation involves a `forall` over inputs followed by a `forall` over key groups. 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, using Grappa's partition-awareness. The global-view model of Grappa allows iterations to be implemented by the application programmer with a `while` loop.

4.3.1 Performance

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: map the points, reduce the cluster means, 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, 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 run k-means on a dataset from Seaflow [102], 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 4.8 for $K = 10$ and $K = 10000$. We find Grappa-MapReduce to be nearly an order of magnitude faster than the comparable Spark implementation. Absolute

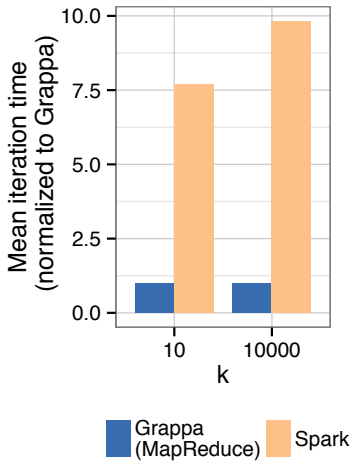

Figure 4.8: Data parallel experiments using k-means on a 8.9GB Seaflow dataset running on 64 nodes. Grappa's performance advantage comes primarily from its network layer as well as reduced iteration overhead as in the Shark comparison (Figure 4.7).

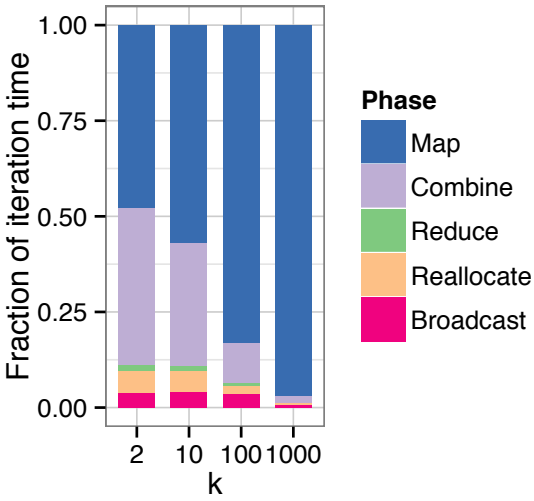

Figure 4.9: Breakdown of time spent in MapReduce phases for Grappa-MapReduce for k-means on the Seaflow dataset.

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.

We examined profiles to understand this difference. We see similar results as with Shark: the bulk of the difference comes from the networking layer and from data serialization. As K grows, this problem *should be* compute-bound: most execution time is spent assigning points to clusters in the map step. Figure 4.9 shows the time breakdown for Grappa-MapReduce. At large K, Grappa-MapReduce is clearly compute-bound. However, the equivalent profile for Spark shows only 50% of the execution time in map; the rest of the time is in the reduce step in network code. Grappa's efficient small message support and support for overlapping communication and computation help it perform well here.

4.4 Writing directly to Grappa

Not all problems fit perfectly into current restricted programming models – for many, a better solution can be found by breaking these restrictions. An advantage of building specialized systems on top of a flexible, high-performance platform is that it makes it easier to implement new optimizations into domain-specific models, or implement a new algorithm from scratch natively. For example, for BFS, Beamer's direction-optimizing algorithm has been shown to greatly improve performance on the Graph500 benchmark by traversing the graph "bottom-up" in order to visit a subset of the edges [17]. This cannot be written in a pure Vertex-centric framework like GraphLab. We implemented the Beamer's BFS algorithm directly on the existing graph data structure in 70 lines of code. Performance results in Figure 4.10 show that this algorithm's performance is nearly a factor of 2 better than the pure Vertex-centric abstraction can achieve.

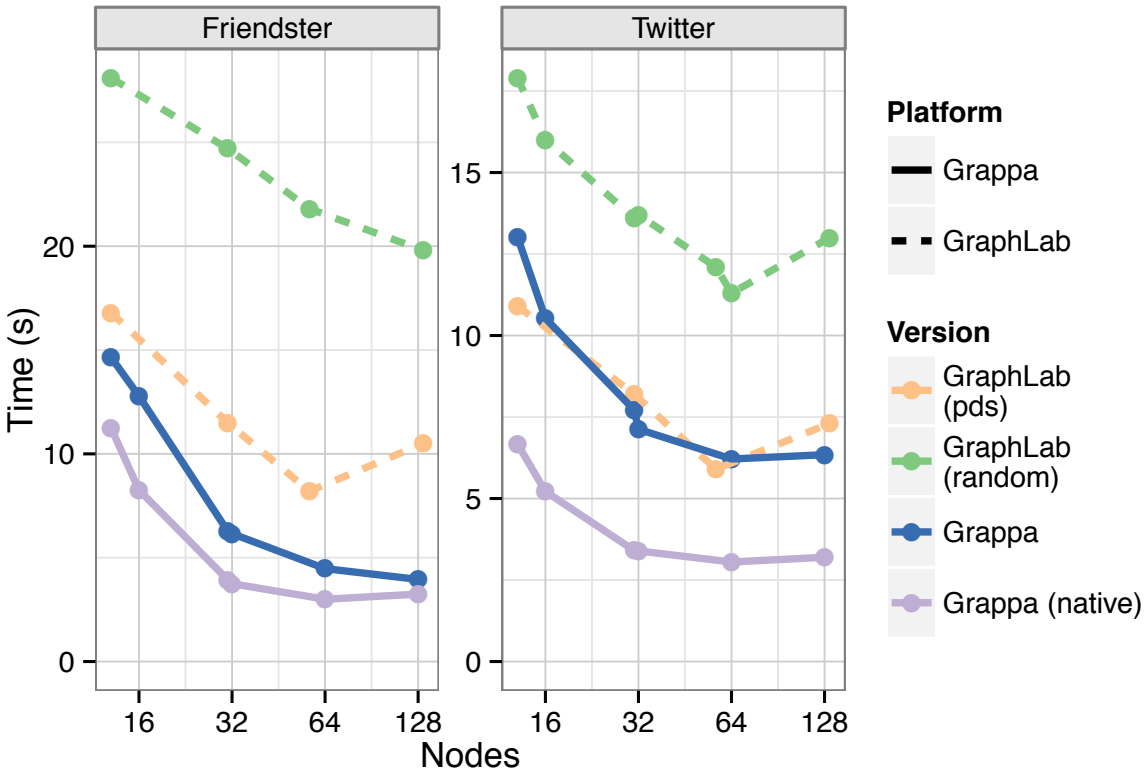

Figure 4.10: Scaling BFS out to 128 nodes. In addition to Grappa's vertex-centric engine, we also show a custom algorithm for BFS implemented natively which employs Beamer's bottom-up optimization to achieve even better performance.

4.5 Measuring Grappa's overhead

Grappa's programming abstraction comes at a cost. This section explores the overhead of providing the Grappa abstraction along a number of axes.

4.5.1 Comparing Grappa's memory operation cost

Remote memory operations in Grappa are significantly more expensive than local memory operations. We can compare the costs using the GUPS benchmark. The GUPS performance of a single cluster node on an array larger than the L3 cache is limited by the random access bandwidth of the node. While we did not build a node-local version of GUPS optimized to measure this number, we can use our previous lined-list measurements from Section 3.1 to estimate that the rate would be on the order of 300 million updates per second. In comparison, Grappa achieves 3 billion updates per second on 128 nodes. Depending on the number of nodes in the job, Grappa is able to perform remote random accesses at a rate between 20 and 30 million updates per second per node. Grappa's delegate-based random access throughput per node is at least an order of magnitude lower than local random access.

The difference in memory latency is even higher. Node-local random access to a large dataset in DRAM takes on the order of 100 nanoseconds on our cluster nodes. Because Grappa uses aggregation to increase throughput, its per-request latency in GUPS is on the order of 1 millisecond: four orders of magnitude greater.

4.5.2 Overhead of iteration

To measure the cost of using Grappa's parallel loops, we compare a Grappa version of BFS with three implementations from the Graph500 benchmark: an MPI version (`bfs_simple`), an OpenMP

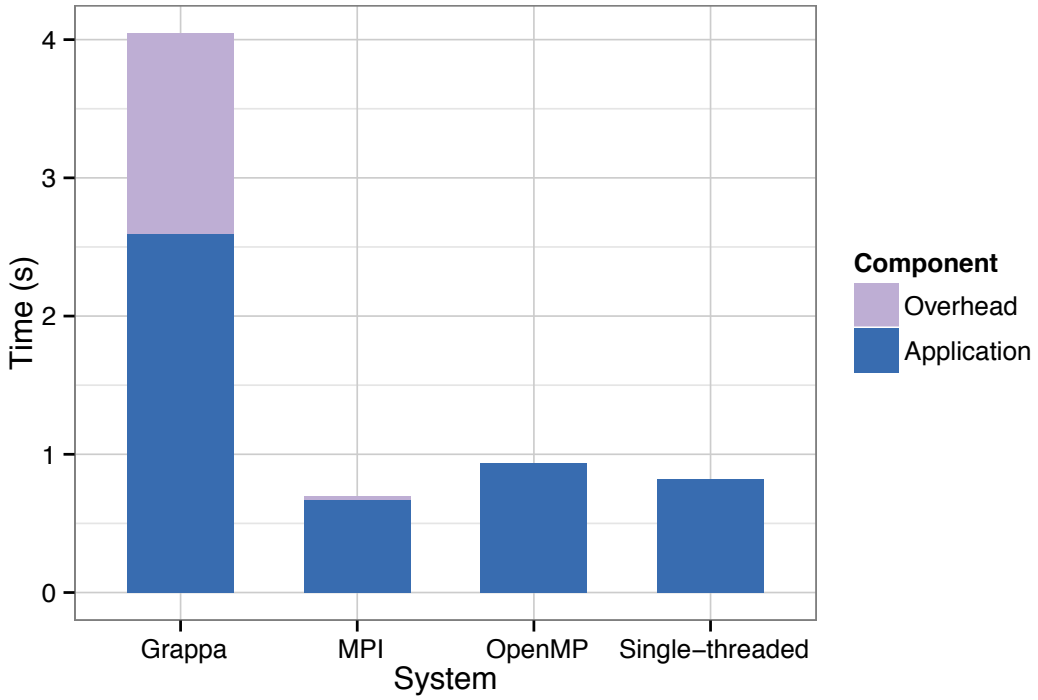

Figure 4.11: Single-core performance of BFS on Grappa compared with three implementations from the Graph500 benchmark. *Overhead* represents time spent converting global addresses to local addresses, expanding the recursive loop decomposition, and calling functions in the MPI library, even though no communication is occuring. *Application* time is spent inspecting and updating vertices.

version, and a single-threaded version, all run on a single core. We use a simple Grappa version that matches the computational pattern of the Graph500 MPI version, rather than the vertex-centric version described previously. This allows us to factor out the cost of communication and focus only on the cost of iteration.

Figure 4.11 shows the result of this comparison run on a scale 21 Kronecker graph. We used a sampling profiler to divide the execution time into two categories: *overhead* represents time spent converting global addresses to local addresses, expanding the recursive loop decomposition, and calling functions in the MPI library, even though no communication was occuring. *Application* time is spent inspecting and updating vertices.

The Graph500 MPI version performs better than the OpenMP and single-threaded versions due to use of a bit vector to track visited vertices. The Grappa version uses the same strategy as the OpenMP and single-threaded versions and stores this information in each vertex.

The bodies of the loops in all the Graph500 versions are very simple: just for loops with a handful of memory operations. Grappa's loop bodies (which make up the application time on the plot) must execute much more code per iteration: both the closures that make up the loop bodies and the delegate operations that modify the graph must be unwrapped and executed. Some of this overhead could be reduced by reorganizing the code to better support inlining.

4.5.3 Overhead of communication

To explore the overhead of Grappa's communication layer, we compare with two benchmarks: first, the Graph500 MPI BFS implementation `bfs_simple`, and second, a stencil code from the Intel Parallel Benchmark suite [54].

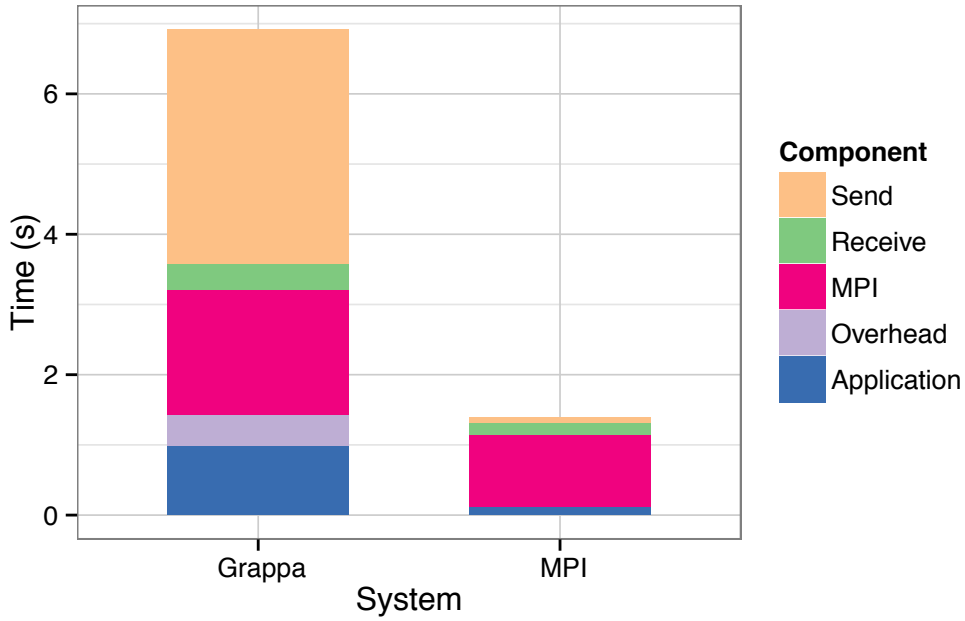

Figure 4.12: Performance of BFS compared with MPI on 8 nodes with 8 cores per node.

Figure 4.12 compares Grappa's vertex-centric BFS with Graph500's MPI `bfs_simple` on 8 nodes and 8 cores per node with a scale 25 Kronecker graph. We see that the MPI version is approximately 5 times faster.

We classified time spent during the run into five different categories using a sampling profiler as in the single-core experiments. The application and overhead categories represent the essentially same components of the system as before other than the MPI library, and we see a similar difference in performance in these categories.

We pull out time spent in the MPI library into its own category; this includes time spent inside MPI runtime functions involved in polling the network stack as well as manipulating buffers for in-flight messages.

Both the Grappa and Graph500 versions of BFS do aggregation, but while Grappa's version is general, the Graph500 version is specialized for BFS. The Send category includes time spent preparing messages to be sent using MPI in both systems. For Grappa this includes enqueuing messages in aggregation buffers and serializing messages into buffers for MPI as well as calling the MPI send functions. For the Graph500 version this includes time in MPI sends as well as writing messages to be sent into the implementation's specialized aggregation buffers. The Receive category is similar: for Grappa it includes time spent pulling apart received buffers, delivering the messages to the proper cores, and deserialzing them as well as time spent in the MPI receive function, and for Graph500 it includes MPI receives and reading from the received aggregation buffers.

We can see that for both systems the runtime is dominated by time doing communication, even given the large difference in actual iteration time. We can futher see that for Grappa the primary contributor is aggregation of messages to be sent. The Graph500 version's specialized form of aggregation is able to operate in a much more cache-friendly manner since it only needs to write 16 bytes per "message" sent, while the Grappa version must include a header and function pointer in its messages, making them 32 bytes. Furthermore, Grappa allocates a whole cache line per message while the messages are waiting to be sent and then allocates another partial cache line as it builds the buffer of aggregated messages to pass to MPI. Clearly this is the most ripe opportunity for optimization in this benchmark, and we have a number of ideas to reduce this overhead which will be explored in future work.

This result does not mean that Grappa is always slower than the equivalent MPI code, though. Figure 4.13 shows the structure of the stencil kernel `synch_p2p` from the Intel Parallel Research Kernels suite [54]. The kernel iterates over a two-dimensional array, computing the value of each

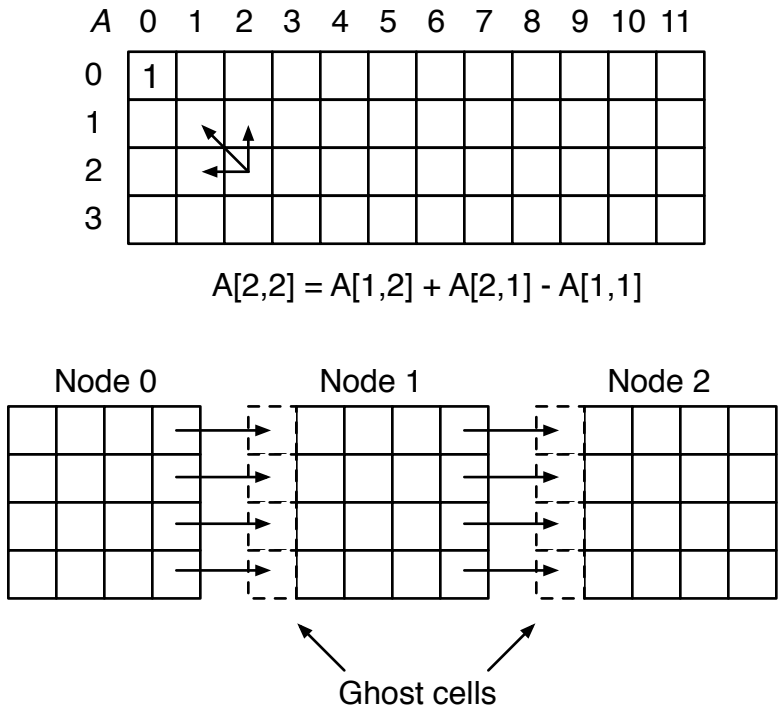

Figure 4.13: Structure of Intel stencil kernel. The top figure shows the logical view of the computation of one cell. The bottom figure shows how the benchmark is mapped to the cluster; ghost cells are denoted with dotted lines, and arrows show how values are communicated between nodes.

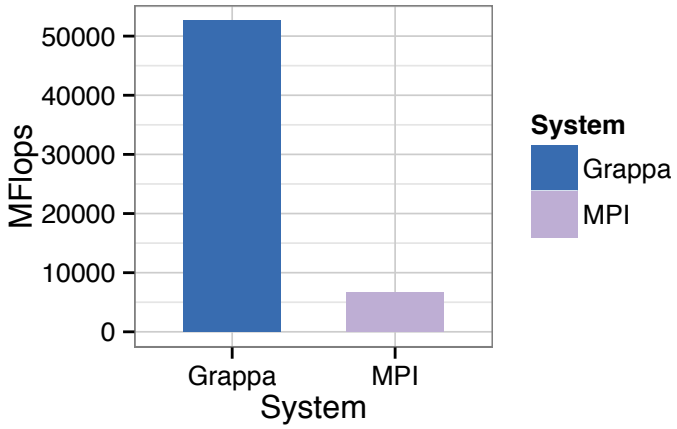

Figure 4.14: Performance of Intel stencil kernel on Grappa compared with reference MPI version on 8 nodes with 8 cores per node. Grappa is able to aggregate updates to ghost cells which increases the amount of pipeline parallelism and thus performance.

cell by using three nearby cells with the relation

$$A_{i,j} = A_{i-1,j} + A_{i,j-1} - A_{i-1,j-1}.$$

There are numerous ways to do this computation, but the MPI reference code implements a style which is amenable to a pipeline-parallel optimization. The rows of the array are partitioned between cores in a blocked fashion shown in the bottom part of the figure. Ghost cells are used to hold copies of the cells to the left of the first column on each core. Each core iterates over its local chunk of the array in the following way: for each row, it waits to receive a message from the core to the left and store the value in the ghost cell. Then it iterates over its local segment of the row using only local values. Finally, once it has computed the last cell of the segment, it sends this value in a message to the next core in the row. This allows the MPI code to pipeline the computation of a row: once a core has sent is message for row i, it can start computing row $i+1$ while the next core works

on row i. Because the MPI code uses blocking sends and receives, this supports P concurrent row segment computations, where P is the number of cores in the system.

We ported this code to Grappa using the same structure, with the following changes: each of the ghost cells was wrapped in a full bit. Each core has a single task that iterates over its local chunk of the array; for each row segment, it first blocks until the full bit is full, then iterates locally just as in the MPI version, and finally sends an asynchronous message to fill the full bit on the next core in the row. The code is of the same complexity as the MPI code.

Figure 4.14 shows that Grappa provides a significant performance benefit over the MPI code for this benchmark. There are two ways in which the Grappa runtime enables more performance from essentially the same code. First, since the full bit message is asynchronous, a core does not have to wait for the next core to receive its message before proceeding to the next row. This allows cores to quickly move to computing the next row immmediately after sending the message for the current row. Second, Grappa's aggregation means that communication is coarsened, allowing the cores to do more work before communicating. The leftmost core is able to compute multiple row segments and queue up the full bit writes. When these writes arrive at the next core, many full bits are filled at once, allowing that core to again compute over many row segments without doing communication. All in all, aggregation increases the time it takes to fill the pipeline, but also drastically increases the amount of concurrent work being executed in the pipeline to $P \times D$, where D is the number of local row segments that can be computed before the aggregator timeout.

4.5.4 When does a single big-memory node make sense?

Computations that could be performed on Grappa could also be performed on a single node with sufficient memory. Writing an application for a single node can take advantage of the lower-overhead accesses to local memory and possibly a simpler programming model. Writing appli-

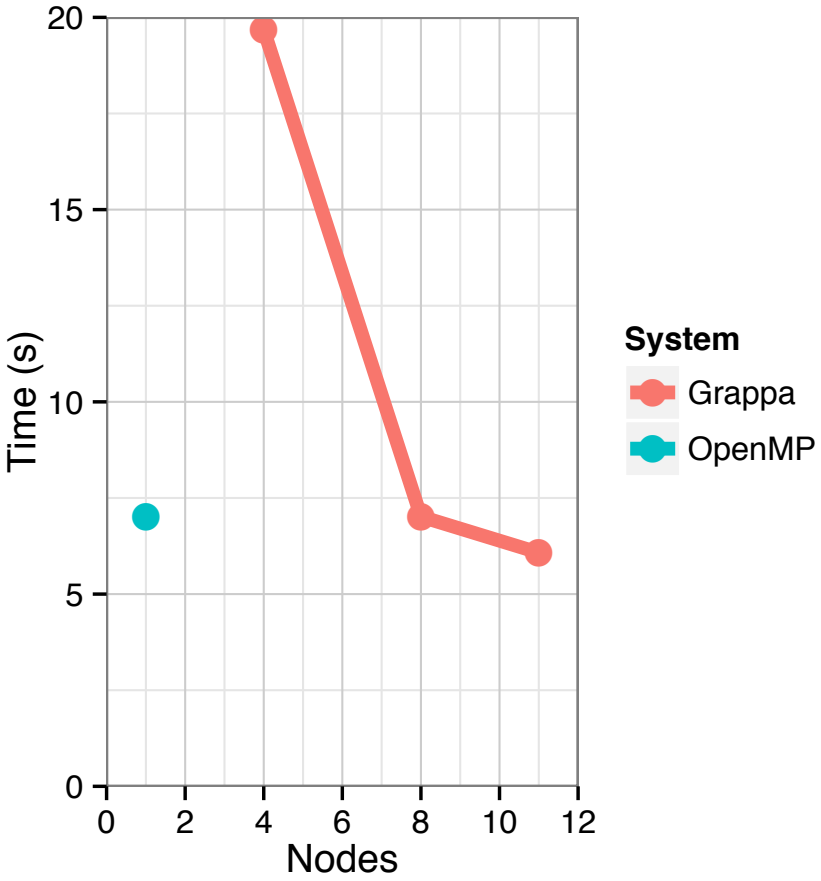

Figure 4.15: Performance of BFS on one node using OpenMP compared to multiple nodes using Grappa with 8 cores per node. On 8 nodes, Grappa matches the performance of the OpenMP version; with more nodes Grappa exceeds OpenMP's performance.

cations for Grappa allows for scalability in three directions: memory capacity, memory bandwidth, and exploitation of parallelism.

For many applications memory capacity is not a compelling reason to choose a distributed programming model like Grappa. x86-based machines with 2 terabytes of RAM are available at the time this was written, with prices similar to that of a small cluster. Our 128-node cluster's aggregate memory capacity is only 8 terabytes (64 GB per node), and given that some memory per node must be used for the OS and runtime data structures, only part of this would be available to applications. These number are very similar. To provide a compelling capacity difference, either cluster nodes must have more DRAM, which increases cost, or clusters must be larger, which increases the probability of failures.

Clusters can provide two key benefits. We have previously seen that the aggregate bandwidth of our cluster using Grappa is significantly greater than the bandwidth of a single node. This single-node bandwidth stays (essentially) constant no matter how much memory each node includes. A cluster also has more cores than a single node, and is able to exploit more parallelism than a single node. Both of these benefits, of course, require that the application actually has parallelism to be exploited.

Since Grappa has higher per-request overhead than a single node programming models, there will be some crossover point where a Grappa cluster application Figure 4.15 shows this point for BFS on a scale 25 Kronecker graph: the plot shows runtime on a single node using the Graph500 OpenMP implementation running on all 12 cores of one of our Intel cluster nodes compared to three different Grappa job configurations running on multiple cluster nodes. We can see that at 8 nodes the Grappa performance is the same as the OpenMP version. This is the point at which Grappa's aggregate memory bandwidth is comparable to the local memory bandwidth exploited by

the OpenMP version. The graph has sufficient parallelism that with 11 nodes the runtime is lower than the maximum performance of the OpenMP version.

In general, this crossover point will change depending on the number of cores and number of memory controllers in the big-memory node compared to the aggregate network bandwidth of the cluster it is compared against. However, for applications with sufficient parallelism, the cluster will always offer the possibility of scaling.

Chapter 5

Related Work

5.1 Multithreading

Grappa uses multithreading to tolerate memory latency. This is a well known technique. Hardware implementations include the Denelcor HEP [99], Tera MTA [12], Cray XMT [41], Simultaneous multithreading [108], MIT Alewife [7], Cyclops [10], and even GPUs [40]. Hardware multithreading often pays with lower single-threaded performance that may limit appeal in the mainstream market. 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) [32]. TAM is a software runtime system designed for prototyping dataflow execution models on distributed memory supercomputers. Like Grappa, TAM supports inter-node 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 [33] was that threading for latency tolerance was fundamentally limited be-

cause 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 find prefetching is effective at hiding 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 [112], TBB [87], Cilk [21] and Capriccio [18] is Grappa's context prefetching. Grappa's prefetching could likely improve from compiler analyses inspired by those of Capriccio for reducing memory usage.

5.2 Software distributed shared memory

The goal of providing a shared memory abstraction for a distributed memory system goes back nearly 30 years. Here I present a few of the ideas explored in this area.

Much of the innovation in SDSM has occurred around reducing the synchronization cost of doing updates. The first DSM systems, including IVY [66], used frequent invalidations to provide sequential consistency, which imposed significant communication cost for write-heavy workloads. Later systems relaxed the consistency model to reduce communication demands. Release consistency, for example, allows updates to be buffered between synchronization events. Furthermore, multiple writer protocols that sent only modified data were developed to help combat false sharing. The Munin [19, 25] and TreadMarks [61] systems exploited both of these ideas, but some coherence overhead was still required. In contrast, Grappa's delegate-based approach to updates avoids synchronization overhead entirely, providing sequential consistency for data-race-free programs without the cost of a coherence protocol. This cost of this communication is mitigated through latency tolerance.

Another way in which SDSM systems differ is in the granularity of access control. Many SDSM systems, including IVY and TreadMarks, tracked ownership at a granularity of a kilobyte

or more. There are two main justifications for this design choice: first, networks are more efficient with large packets, and second, multi-kilobyte granularity allowed systems to reuse the processor's paging mechanisms to accelerate access control checks and to provide shared memory transparently. Unfortunately, this meant that these systems depended on lots of locality to amortize the cost of moving these large blocks, and the systems were very susceptible to false sharing. Other systems, including Munin and Blizzard [94], allowed tracking ownership with variable granularity to address these problems. Grappa's delegate-based approach is similar; since updates are always performed at data's home node, only modified data needs to be moved. FaRM [35] 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.

SDSM systems often required extensive modifications to the system software stack, including the OS (IVY, Blizzard) and compilation infrastructure (Munin). This made these systems difficult to port to new platforms. Grappa follows the lead of TreadMarks and provides SDSM entirely at user-level through a library and runtime.

In summary, while Grappa's DSM system is conceptually similar to prior work, we accept the random access aspect of irregular applications and optimize for throughput rather than low latency. Our DSM system must support enough memory concurrency to tolerate the latency of the network and additional latency overhead imposed by the runtime system.

5.3 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. Split-C [31] is

the earliest example I know of this style of language; contemporary examples include Chapel [26], X10 [27], Co-array Fortran [79] and UPC [38]. Grappa shares many parts of its design philosophy with these languages.

There are two key ideas Grappa draws from PGAS languages. First, PGAS languages implement DSM at the language, rather than the system level. This allows for a number of optimizations, including efficient split-phase access, variable data granularity, and support for user-customizable synchronization operations. Grappa follows this approach for this reason.

Second, in PGAS languages, each piece of data has a single canonical location on a particular node. The language designers expect programmers to modify algorithms to take advantage of locality by processing node-local data as much as possible; access to data stored on other nodes is possible but is seen as something to avoid if possible. Grappa is designed to support the opposite view: Grappa is optimized for random access to data anywhere in the cluster, and locality can be exploited when it is available.

Most PGAS languages adopt a SPMD programming model: the programmer must reason about what is happening on every node in the system. Chapel is the exception: it provides a global view of control, while allowing programmers to direct individual cores when necessary. Grappa follows the same approach, providing a single-machine abstraction to the programmer along with support for controlling the locality of computation when requested.

5.4 Distributed data-intensive processing frameworks

There are many other data-parallel frameworks like Hadoop, Haloop [24], and Dryad [57]. These are designed to make parallel programming on distributed systems easier; they meet this goal by targeting data-parallel programs. There have also been recent efforts to build parameter servers for

distributed machine learning algorithms using asynchronous communication and distributed key-value storage built from RPCs [8, 9]. The incremental data-parallel system Naiad [74] achieves both high-throughput for batch workloads and low-latency for incremental updates. Most of these designs eschew DSM as an application programming model for performance reasons. However, a high-throughput DSM like Grappa is a useful building block on its own for applications or higher-level abstractions. Future work may expand Grappa to support low-latency networking for critical tasks similar to Naiad.

While Grappa is a general runtime system for any large-scale concurrent application, it has been designed to perform especially well on graph analysis. Other distributed graph processing frameworks include Pregel [70] and GraphLab [43, 68]. Pregel adopts a bulk-synchronous parallel (BSP) execution model, which makes it inefficient on workloads that could prioritize vertices. GraphLab overcomes this limitation with an execution mode that schedules vertex computations individually, allowing prioritization, which gives faster convergence in a variety of iterative algorithms. GraphLab, however, imposes a rigid computation model where programmers must express computation as transformations on a vertex and its edge list, with information only from adjacent vertexes. Pregel is only slightly less restrictive, as the input data can be any vertex in the graph. Grappa also supports dynamic parallelism with asynchronous execution, but parallelism is expressed as tasks or loop iterations, which is a far more general programming model for irregular computation tasks.

Grappa's latency-tolerant approach to irregular and graph applications is catching on. Colleagues at PNNL have developed GMT [73], a system that follows the same basic approach as Grappa but with more of a focus on node-shared resources. After discussions with the Grappa team, the open-source version of GraphLab now includes lightweight context switching for latency tolerance [67].

Chapter 6

Conclusion

Our work builds on the premise that writing data-intensive applications and frameworks in a shared memory environment is simpler than developing custom distributed computing infrastructure from scratch. To that end, Grappa is inspired not by SMP systems, but by novel supercomputer hardware—the Tera MTA and Cray XMT line of machines. This work borrows the core insight of those hardware systems and builds it into a software runtime tuned to extract performance from commodity processors, memory systems, and networks.

Our experiments show that frameworks such as MapReduce, vertex-centric computation, and query execution are *easy to build* and *efficient* in the Grappa framework. Our MapReduce and query execution implementations are an order of magnitude faster than the custom frameworks for each. Our vertex-centric GraphLab-inspired API is 1.33× faster than GraphLab itself, without the need for complex graph partitioning schemes.

It is clear that distributed shared memory can be efficient for irregular and data-intensive applications when it judiciously exploits the key application characteristics of concurrency and latency tolerance.

Appendix A

Future work

The completion of my PhD is not the end Grappa project: there are many topics yet to be explored. Here are five high-level areas full of interesting questions that are ripe for future investigation.

Tolerating node-local latency Grappa has so far focused on tolerating latency of remote access on clusters. However, our characterization of our nodes' memory systems demonstrates that there is an opportunity to tolerate latency to local memory as well. Our work on prefetching in the Grappa scheduler and messaging layer helps tolerate cache miss latencies when context-switching or sending messages, but there are many other opportunities. Accesses to low-locality user data in local DRAM could be prefetched. Synchronization operations could also prefetch worker contexts before enqueueing them to be run.

Collaborations and applications Much of our work on Grappa has been bottom-up, focused on exploring low-level runtime features. But the project will only be truly successful with application wins. Since we are not domain experts, it is important for us to work with those experts to understand what is needed. The Raco relational query backend described here is the first fruit of this effort.

Conversations with other domain experts have led us to believe Grappa may be successfully applied in a number of areas. A new algorithm for community detection [90] requires efficient manipulation of large trees; we expect Grappa to do well here. Branch-and-bound implemented on Grappa would take advantage of the same large tree manipulation capability, and may help scale up solvers for large integer linear programs and other optimization problems. Grappa's small message performance could help accelerate the solution of sparse linear systems, or of partial differential equations using stencil codes. Other users may simply be looking for a way to use the computational models provided by MapReduce or Spark without using languages or tools from the Java ecosystem.

Improving performance, programmability, and robustness The design of Grappa provides the tools necessary to implement the SC-DRF memory model but does not enforce their proper use. Can we enforce SC-DRF in the runtime without hurting performance?

Given the amount of concurrency available in our applications/datasets and our ability to tolerate latency, I believe we have considerable flexibility in reordering operations from different threads across the system to increase performance while still providing SC-DRF semantics, especially within parallel loops. Can we formalize these tradeoffs in a memory model, in the style of Cilk [20]? How can we help programmers make these tradeoffs easily?

How can we use higher-level semantic information to optimize performance? Providing consistency at the level of memory may be unnecessarily detailed when we are dealing with data structures accessed through an API: if we can control the interactions between the data structure's API and other parts of the program, the data structure may be able to reorder and merge operations in a way that is safe according to the API's spec but not allowed by a strict memory model [46, 60]

Grappa's current stacks are of fixed size, which bounds the total number of workers. This means that we can run out of execution resources if parallelism increases too quickly. Can we apply cactus stacks as in Cilk [21] but in a distributed memory setting? This would require compiler support and changes to calling conventions: does this impact compatibility with external libraries?

What about IO? Grappa has very basic parallel IO support now. The combination of user-level threading and system calls leads to complex interactions, and Linux appears to no longer support scheduler activations [14]. However, may be opportunities to apply work-stealing to IO.

Grappa currently assumes hardware will not fail over the lifetime of a job; this is of course not true. What can we learn from in-memory storage projects like RamCloud [80]? Are there special challenges or opportunities given our latency tolerance capability?

Increasing capacity with solid state storage Current systems for scaling irregular applications are quite sensitive to the size of of the data involved, due to the applications' dependence on random accesses. If the data can fit in DRAM, application performance can be good. When the data exceeds the size of DRAM, performance is limited by the random access rate of the secondary storage device, which is orders of magnitude slower than that of DRAM—even the fastest Flash-based devices claim only a million I/O operations per second. Furthermore, individual accesses have a high latency.

Can we reuse Grappa's latency tolerance techniques to store our shared heap in Flash? Recent work [63, 82] suggests that the we could aggregate requests going to common pages in Flash devices in order to mitigate the per-page access latency. Alternatively, we could redesign Flash storage controllers to support orders of magnitude more throughput—an idea we explored briefly in [75]. Finally, we could explore designs for emerging memory technologies like PCM that would get us the performance properties we need.

Leveraging more parallelism with SIMD operations Today Grappa exploits parallelism in two ways. First, threads run in parallel on cores in nodes across the system. Second, since our cores are out-of-order superscalar processors, instructions run in parallel on the cores. Our systems have a third mechanism for parallelism, which we have completely ignored: SIMD instructions. Even though our applications are irregular, we speculate there is sufficient control isomorphism among separate tasks in the applications that we could execute multiple tasks in parallel using SIMD instructions, using latency tolerance to support the necessary task reorderings. We might even be able to run on GPUs. These approaches could increase the rate we generate remote requests, and move us closer to a balanced system.

Appendix B

Example Grappa programs

This chapter describes some of the features of the Grappa API though the lens of two example programs. Grappa's API is in flux: the code shown here uses the current version of the API, but some features are likely to change in the future. The most current version of Grappa is stored in a repository linked to from `http://grappa.io`; these examples are taken from the tutorial in the Grappa repository's documentation folder.

B.1 GUPS

Listing 3 shows the bulk of a Grappa implementation of the Giga-Updates Per Second (GUPS) benchmark, with the loop that performs the updates omitted. We first discuss the setup and tear-down code for the benchmark, and then present multiple methods for implementing the update loop.

The example starts by including the Grappa header file and using the Grappa namespace. After that, we see a few declarations. Lines 5 and 6 declare command-line flags using the GFlags library [1]; flags are referenced in the code by prepending the `FLAGS_` prefix as seen on lines 17 and

24. Lines 10 and 11 declare "metric" variables that are tracked by the runtime in order to support sampling and collection of statistics. In this example we do not use these more advanced features; we only set and read the values in lines 34 through 37.

Execution begins in `main()`. Lines 14 and 42 set up and tear down, respectively, Grappa's runtime system; they play the same role as `MPI_Init` and `MPI_Finalize` in MPI. At this point, the program is still a standard MPI-like SPMD program.

The `run()` call on line 15 switches the program into Grappa's global view mode. The `run()` function is passed a C++11 lambda containing the first code to run in global view mode.

Lines 17 and 24 allocate arrays in Grappa's global address space. The arrays are allocated in a block-cyclic fashion across all the cores in the job with a 64-byte block size by default. Lines 39 and 40 later free these arrays. Line 19 is a parallel loop that initializes the elements of the `A` array to 0. This loop runs in parallel on all the cores of the system; each core iterates only over the elements of the array that are stored locally. Other styles of `forall` loops exist and will be discussed later. Line 26 similarly initializes each element of the `B` array to a random index into the `A` array.

Now we discuss a number of possible loop bodies to insert at line 32. Listing 4 shows five different approaches.

The first approach, at line 2, is only conceptual—this code would not compile, but it shows what we are trying to accomplish. For each element of the index array, we increment the element of the increment array `A` referenced by the current element of the index array `B`.

The second approach, at line 7, shows a `forall()` loop that iterates over integer indices. In this case, the lambda will be called `FLAGS_sizeB` times, with the argument ranging from 0 to `FLAGS_sizeB` − 1. For each iteration, we do two operations on global memory. First, we read the value from the ith element of the index array B using a delegate read operation. Then we increment the appropriate entry in the increment array A using a delegate fetch-and-add operation. As is usual,

the fetch-and-add operation returns the value in the array cell before being incremented. Both of these delegate operations may require remote communication and thus have long latency, so each one yields to the scheduler after initiating its operation. The `forall` is recursively decomposed so that other iterations may start while running iterations wait for responses. The iterations of this loop do not depend on locality and so could run on any core in the job. This version of the `forall` loop starts by partitioning the index space across all the cores in a blocked fashion. Without additional options, those iterations stay bound to the cores on which they are initially assigned; a later example will show how to remove that binding and allow work-stealing to balance load.

The third approach, at line 12, takes advantage of locality in the index array to optimize away one of the delegate operations in the previous loop. We do this with a special version of the `forall` loop optimized for iterating over arrays. In this case, the first argument is a global pointer to the array, and the second argument is the number of elements in the array. The `forall` loop then spawns tasks on each core of the job to iterate over all the elements of the array which are assigned to that core by the block-cyclic distribution. The lambda is passed a reference to each of these local elements. Because of this assumption of locality the tasks must stay bound to the cores on which they are first assigned. Since the lambda is passed the value from the index array directly, only one blocking operation per iteration is required, instead of two in the previous version.

The fourth approach, at line 17, uses the same localized iteration as the previous approach but removes the requirement of context switching by using an "asynchronous" increment operation that does not return a value. This operation couples the completion of each increment to the synchronization object used by the `forall` loop to detect the completion of all loop iterations. The increments are only guaranteed to have completed after the whole loop is complete, whereas in the previous example the completion of an iteration meant that the increment for that iteration was complete.

Finally, the fifth approach, at line 22, shows a generalized delegate operation. This code performs exactly the same work as the previous version, but the delegate operation takes a lambda to execute on the home core of the location being modified, rather than executing a hard-coded increment operation.

B.2 Tree search

Listing 5 and Listing 6 show an example of nested dynamic parallelism: searching a tree stored in global memory. Each vertex of the tree has an ID, a color value, and an array of pointers to its children.

The variables declared in lines 1 and 2 demonstrate the idea of *symmetric* allocation of storage. Because these variables are declared in global scope, every process has space allocated for these variables, and the virtual address of each variable is the same in every process. Since Grappa runs one process per core in the job, this means every core has its own private copy of these variables. Grappa also supports dynamic allocation of symmetric storage by using the functions `symmetric_global_alloc<>()` and `symmetric_global_free()`.

We use this property on line 6 of Listing 6 and line 22 of Listing 5 to keep a local count of matching vertices per core and then to do a collective reduction across all cores. In the reduction, we pass the address of the local copy of a symmetric variable knowing that every process in the job has data at that address.

Line 12 of Listing 5 shows how to escape Grappa's global view abstraction for the underlying SPMD model; `on_all_cores()` runs its lambda argument on every core in the job, and blocks until all the tasks have completed.

Line 10 shows how to expose parallelism recursively. The `forall_here()` call iterates over the vertex's child array and spawns an asynchronous task for each child. As with asynchronous delegates, the tasks are registered with a cluster-wide global synchronization object. This object is managed by the `finish()` call on line 17 of Listing 5; the call prepares the synchronization object before executing its lambda and starting the recursion, and it blocks until all child tasks are complete.

The `forall_here()` call on line 10 has two parameters that control task locality. First, the `_here` prefix causes tasks to be enqueued on the spawning core, rather than immediately scheduling iterations of the loop across the job as in the regular `forall` case. Second, the <unbound> argument marks the tasks as stealable: if another core in the system is idle, it will iterate over cores in the job randomly and attempt to steal tasks in order to balance load.

```
#include <Grappa.hpp>
using namespace Grappa;

// define command-line flags (third-party 'gflags' library)
DEFINE_int64(sizeA, 1<<30, "Size of array that GUPS increments");
DEFINE_int64(sizeB, 1<<20, "Number of iterations");

// define custom metrics which are tracked by the runtime
// (here we're just printing these ourselves)
GRAPPA_DEFINE_METRIC(SimpleMetric<double>, gups_runtime, 0.0);
GRAPPA_DEFINE_METRIC(SimpleMetric<double>, gups_throughput, 0.0);

int main(int argc, char * argv[]) {
  init(&argc, &argv);
  run([]{
    // allocate target array from the global heap
    auto A = global_alloc<int64_t>(FLAGS_sizeA);
    // fill the array with all 0's (in parallel on all cores)
    forall(A, FLAGS_sizeA, [](int64_t& a) {
      a = 0;
    });

    // allocate another array
    auto B = global_alloc<int64_t>(FLAGS_sizeB);
    // initialize the array with random indices
    forall(B, FLAGS_sizeB, [](int64_t& b) {
      b = random() % FLAGS_sizeA;
    });

    double start = walltime();

    // GUPS code goes here

    gups_runtime = walltime() - start;
    gups_throughput = FLAGS_sizeB / gups_runtime;
    LOG(INFO) << gups_throughput.value() << " UPS in "
              << gups_runtime.value() << " seconds";

    global_free(B);
    global_free(A);
  });
  finalize();
}
```

Listing 3: Grappa code for GUPS.

```
// GUPS algorithm, conceptually:
// for (int i = 0; i < sizeB; i++) {
//   A[B[i]] += 1;
// }

// index-based forall
forall(0, FLAGS_sizeB, [=](int64_t i){
  delegate::fetch_and_add( A + delegate::read(B+i), 1);
});

// localized iteration over index array
forall(B, FLAGS_sizeB, [=](int64_t& b){
  delegate::fetch_and_add( A + b, 1 );
});

// Asynchronous updates
forall(B, FLAGS_sizeB, [=](int64_t& b){
  delegate::increment<async>( A + b, 1);
});

// Generalized delegate operation
forall(B, FLAGS_sizeB, [=](int64_t i, int64_t& b){
  delegate::call<async>(A+b, [i](int64_t* a){
    *a += i;
  });
});
```

Listing 4: Inner loops for GUPS.

```
int64_t count;
long search_color;

int main( int argc, char * argv[] ) {
  init( &argc, &argv );
  run([]{
    size_t num_vertices = 1000;

    GlobalAddress<Vertex> root = create_tree(num_vertices);

    // initialize all cores
    on_all_cores([]{
      search_color = 7; // arbitrary search key
      count = 0;
    });

    finish([=]{
      search( root );
    });

    // compute total count
    int64_t total = reduce<int64_t,collective_add<int64_t>>(&count);

    LOG(INFO) << "total count: " << total;

  });
  finalize();
  return 0;
}
```

Listing 5: Grappa code for tree search.

```
void search(GlobalAddress<Vertex> vertex_addr) {
  // fetch the vertex info
  Vertex v = delegate::read(vertex_addr);

  // check the color
  if (v.color == search_color) count++;

  // search children
  GlobalAddress<Vertex> children = v.first_child;
  forall_here<unbound,async>(0, v.num_children, [children](int64_t i){
    search(children+i);
  });
}
```

Listing 6: Tree search main routine.

Bibliography

[1] GFlags C++ command-line flags library `https://code.google.com/p/gflags`.

[2] Intel Data Plane Development Kit. `http://goo.gl/AOvjss`, 2013.

[3] Big data benchmark. `https://amplab.cs.berkeley.edu/benchmark`, Feb 2014.

[4] Snabb Switch project. `https://github.com/SnabbCo/snabbswitch`, May 2014.

[5] N. Adiga, G. Almasi, G. Almasi, Y. Aridor, R. Barik, D. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. Blumrich, A. Bright, J. Brunheroto, C. Cascaval, J. Castanos, W. Chan, L. Ceze, P. Coteus, S. Chatterjee, D. Chen, G. Chiu, T. Cipolla, P. Crumley, K. Desai, A. Deutsch, T. Domany, M. Dombrowa, W. Donath, M. Eleftheriou, C. Erway, J. Esch, B. Fitch, J. Gagliano, A. Gara, R. Garg, R. Germain, M. Giampapa, B. Gopalsamy, J. Gunnels, M. Gupta, F. Gustavson, S. Hall, R. Haring, D. Heidel, P. Heidelberger, L. Herger, D. Hoenicke, R. Jackson, T. Jamal-Eddine, G. Kopcsay, E. Krevat, M. Kurhekar, A. Lanzetta, D. Lieber, L. Liu, M. Lu, M. Mendell, A. Misra, Y. Moatti, L. Mok, J. Moreira, B. Nathanson, M. Newton, M. Ohmacht, A. Oliner, V. Pandit, R. Pudota, R. Rand, R. Regan, B. Rubin, A. Ruehli, S. Rus, R. Sahoo, A. Sanomiya, E. Schenfeld, M. Sharma, E. Shmueli, S. Singh, P. Song, V. Srinivasan, B. Steinmacher-Burow, K. Strauss, C. Surovic, R. Swetz, T. Takken, R. Tremaine, M. Tsao, A. Umamaheshwaran, P. Verma, P. Vranas, T. Ward, M. Wazlowski, W. Barrett, C. Engel, B. Drehmel, B. Hilgart, D. Hill, F. Kasemkhani, D. Krolak, C. Li, T. Liebsch, J. Marcella, A. Muff, A. Okomo, M. Rouse, A. Schram, M. Tubbs, G. Ulsh, C. Wait, J. Wittrup, M. Bae, K. Dockser, L. Kissel, M. Seager, J. Vetter, and K. Yates. An overview of the BlueGene/L supercomputer. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 60–60, Nov 2002.

[6] S. V. Adve and M. D. Hill. Weak ordering – A new definition. In *ISCA-17*, 1990.

[7] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife machine: Architecture and performance. In *22nd Annual International Symposium on Computer Architecture*, June 1995.

[8] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. J. Smola. Scalable inference in latent variable models. In *Proceedings of the Fifth ACM International Conference on Web Search and Data Mining*, WSDM '12, pages 123–132, New York, NY, USA, 2012. ACM.

[9] A. Ahmed, N. Shervashidze, S. Narayanamurthy, V. Josifovski, and A. J. Smola. Distributed large-scale natural graph factorization. In *Proceedings of the 22Nd International Conference on World Wide*

Web, WWW '13, pages 37–48, Republic and Canton of Geneva, Switzerland, 2013. International World Wide Web Conferences Steering Committee.

[10] G. Almási, C. Caşcaval, J. G. Castaños, M. Denneau, D. Lieber, J. E. Moreira, and H. S. Warren, Jr. Dissecting Cyclops: A detailed analysis of a multithreaded architecture. *SIGARCH Computer Architecture News*, 31:26–38, March 2003.

[11] G. Alverson, R. Alverson, D. Callahan, B. Koblenz, A. Porterfield, and B. 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.

[12] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *Proceedings of the 4th International Conference on Supercomputing*, ICS '90, pages 1–6, New York, NY, USA, 1990. ACM.

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

[14] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, SOSP '91, pages 95–109, New York, NY, USA, 1991. ACM.

[15] L. A. Barroso and U. Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. 2009.

[16] K. E. Batcher. STARAN parallel processor system hardware. In *Proceedings of the May 6-10, 1974, National Computer Conference and Exposition*, AFIPS '74, pages 405–410, New York, NY, USA, 1974. ACM.

[17] S. Beamer, K. Asanovi, and D. Patterson. Direction-optimizing breadth-first search. In *Conference on Supercomputing (SC-2012)*, November 2012.

[18] R. V. Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: Scalable threads for internet services. In *In Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 268–281. ACM Press, 2003.

[19] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the Second ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPOPP '90, pages 168–176, New York, NY, USA, 1990. ACM.

[20] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. Dag-consistent distributed shared memory. In *Proceedings of the 10th International Parallel Processing Symposium*, IPPS '96, pages 132–141, Washington, DC, USA, 1996. IEEE Computer Society.

[21] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.

[22] H.-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.

[23] J. Brandenburg. Technology advances in the Intel Paragon system. In *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '93, pages 182–, New York, NY, USA, 1993. ACM.

[24] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: Efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3(1-2):285–296, Sept. 2010.

[25] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 152–164, New York, NY, USA, 1991. ACM.

[26] 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.

[27] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[28] Y. Chen, S. Alspaugh, and R. Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *Proc. VLDB Endow.*, 5(12):1802–1813, Aug. 2012.

[29] R. P. Colwell, W. E. Hall, C. S. Joshi, D. B. Papworth, P. K. Rodman, and J. E. Tornes. Architecture and implementation of a VLIW supercomputer. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, Supercomputing '90, pages 910–919, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[30] H. G. Cragon and W. J. Watson. The TI Advanced Scientific Computer. *Computer*, 22(1):55–64, Jan. 1989.

[31] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, Supercomputing '93, pages 262–273, New York, NY, USA, 1993. ACM.

[32] D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. von Eicken. TAM – A compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, 18:347–370, July 1993.

[33] D. E. Culler, K. E. Schauser, and T. v. Eicken. Two fundamental limits on dataflow multiprocessing. In *Proceedings of the IFIP WG10.3. Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, PACT '93, pages 153–164, Amsterdam, The Netherlands, The Netherlands, 1993. North-Holland Publishing Co.

[34] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th USENIX conference on Operating Systems Design and Implementation*, OSDI'04, Berkeley, CA, USA, 2004. USENIX Association.

[35] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX.

[36] P. Druschel, L. L. Peterson, and B. S. Davie. Experiences with a high-speed network adaptor: A software perspective. *SIGCOMM Comput. Commun. Rev.*, 24(4):2–13, Oct. 1994.

[37] J. Duell, P. Hargrove, and E. Roman. The design and implementation of Berkeley Labs Linux checkpoint/restart. Technical Report LBNL-54941, December 2002.

[38] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. *UPC: Distributed Shared Memory Programming*. John Wiley and Sons, Inc., Hoboken, NJ, USA, 2005.

[39] C. Engle, A. Lupher, R. Xin, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: Fast data analysis using coarse-grained distributed memory. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 689–692, New York, NY, USA, 2012. ACM.

[40] K. Fatahalian and M. Houston. A closer look at GPUs. *Communications of the ACM*, 51:50–57, October 2008.

[41] J. Feo, D. Harper, S. Kahan, and P. Konecny. Eldorado. In *Proceedings of the 2nd Conference on Computing Frontiers*, CF '05, pages 28–34, New York, NY, USA, 2005. ACM.

[42] M. Flajslik and M. Rosenblum. Network interface design for low latency request-response protocols. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 333–346, Berkeley, CA, USA, 2013. USENIX Association.

[43] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[44] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer: Designing an MIMD shared memory parallel computer. *IEEE Transactions on Computers*, C-32(2):175 –189, Feb. 1983.

[45] Graph 500. `http://www.graph500.org/`, July 2012.

[46] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, SPAA '10, pages 355–364, New York, NY, USA, 2010. ACM.

[47] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[48] W. D. Hillis. *The Connection Machine*. MIT Press, Cambridge, MA, USA, 1986.

[49] 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.

[50] B. Holt, P. Briggs, L. Ceze, and M. Oskin. Alembic. In *International Conference on PGAS Programming Models (PGAS)*, Oct 2013.

[51] B. Holt, J. Nelson, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Flat combining synchronized global data structures. In *International Conference on PGAS Programming Models (PGAS)*, Oct 2013.

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

[53] Intel. Intel 5520 chipset and intel 5500 chipset datasheet.

[54] Intel. Intel parallel research kernels `https://github.com/ParRes/Kernels`.

[55] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Number 248966-018. March 2009.

[56] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Number 253669-033US. December 2009.

[57] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2007 ACM SIGOPS European Conference on Computer Systems*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[58] 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.

[59] E. Y. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: A highly scalable user-level TCP stack for multicore systems. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 489–502, Berkeley, CA, USA, 2014. USENIX Association.

[60] S. Kahan and P. Konecny. MAMA!: a memory allocator for multithreaded architectures. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 178–186, New York, NY, USA, 2006. ACM.

[61] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the USENIX Winter 1994 Technical Conference*, WTEC'94, pages 115–131, Berkeley, CA, USA, 1994. USENIX Association.

[62] H. Kwak, C. Lee, H. Park, and S. 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.

[63] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: large-scale graph computation on just a PC. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46, Berkeley, CA, USA, 2012. USENIX Association.

[64] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA highly scalable server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, pages 241–251, New York, NY, USA, 1997. ACM.

[65] D. Levinthal. Performance analysis guide for Intel Core i7 processor and Intel Xeon 5500 processors. Technical report, 2009. `http://software.intel.com/sites`.

[66] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, Nov. 1989.

[67] Y. Low. Personal communication.

[68] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *PVLDB*, 2012.

[69] R. Lublinerman, J. Zhao, Z. Budimlic, S. Chaudhuri, and V. Sarkar. Delegated isolation. In *OOPSLA'11*, pages 885–902, 2011.

[70] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

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

[72] C. Mitchell, Y. Geng, and J. Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 103–114, Berkeley, CA, USA, 2013. USENIX Association.

[73] A. Morari, A. Tumeo, D. Chavarría-Miranda, O. Villa, and M. Valero. Scaling irregular applications through data aggregation and software multithreading. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 1126–1135, Washington, DC, USA, 2014. IEEE Computer Society.

[74] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[75] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Pomace: a grappa for non-volatile memory. NVMW2013, 2013.

[76] J. Nelson, B. Myers, A. H. Hunter, P. Briggs, L. Ceze, C. Ebeling, D. Grossman, S. Kahan, and M. Oskin. Crunching large graphs with commodity processors. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Parallelism*, HotPar'11, pages 10–10, Berkeley, CA, USA, 2011. USENIX Association.

[77] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, June 2011.

[78] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. Scale-out NUMA. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 3–18, New York, NY, USA, 2014. ACM.

[79] R. W. Numrich and J. Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, Aug. 1998.

[80] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, D. Ongaro, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for RAMCloud. *Commun. ACM*, 54(7):121–130, July 2011.

[81] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[82] R. Pearce, M. Gokhale, and N. M. Amato. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[83] S. Peter and T. Anderson. Arrakis: A case for the end of the empire. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 26–26, Berkeley, CA, USA, 2013. USENIX Association.

[84] O. C. Project. Open compute project website `http://www.opencompute.org`.

[85] C. J. Purcell. The Control Data STAR-100: Performance measurements. In *Proceedings of the May 6-10, 1974, National Computer Conference and Exposition*, AFIPS '74, pages 385–387, New York, NY, USA, 1974. ACM.

[86] Raco: The relational algebra compiler. `https://github.com/uwescience/datalogcompiler`, April 2014.

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

[88] K. Ren, Y. Kwon, M. Balazinska, and B. Howe. Hadoop's adolescence: An analysis of Hadoop usage in scientific workloads. *Proc. VLDB Endow.*, 6(10):853–864, Aug. 2013.

[89] L. Rizzo. Netmap: A novel framework for fast packet I/O. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[90] M. Rosvall and C. T. Bergstrom. Maps of information flow reveal community structure in complex networks. In *Proceedings of the National Academy of Sciences*, pages 1118–1123, 2007.

[91] A. Rowstron, D. Narayanan, A. Donnelly, G. O'Shea, and A. Douglas. Nobody ever got fired for using Hadoop on a cluster. In *1st International Workshop on Hot Topics in Cloud Data Processing (HotCDP 2012)*. ACM, April 2012.

[92] R. M. Russell. The CRAY-1 computer system. *Commun. ACM*, 21(1):63–72, Jan. 1978.

[93] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. SP2Bench: A SPARQL performance benchmark. *CoRR*, abs/0806.4627, 2008.

[94] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-grain access control for distributed shared memory. In *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, ASPLOS VI, pages 297–306, New York, NY, USA, 1994. ACM.

[95] S. L. Scott. Synchronization and communication in the T3E multiprocessor. *SIGPLAN Not.*, 31(9):26–36, Sept. 1996.

[96] P. SIG. PCI Express specification website `https://www.pcisig.com/specifications/pciexpress/base3`.

[97] J. P. Singh, T. Joe, J. L. Hennessy, and A. Gupta. An empirical comparison of the Kendall Square Research KSR-1 and Stanford DASH multiprocessors. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, Supercomputing '93, pages 214–225, New York, NY, USA, 1993. ACM.

[98] B. Smith. The architecture of HEP. In *on Parallel MIMD computation: HEP supercomputer and its applications*, pages 41–55, Cambridge, MA, USA, 1985. Massachusetts Institute of Technology.

[99] B. J. Smith. Architecture and applications of the HEP multiprocessor computer system. In *Proceedings of SPIE 0298, Real-Time Signal Processing IV, 241*, volume 298, pages 241–248, July 1982.

[100] E. Speight, H. Abdel-Shafi, and J. K. Bennett. Realizing the performance potential of the virtual interface architecture. In *Proceedings of the 13th international conference on Supercomputing*, ICS '99, pages 184–192, New York, NY, USA, 1999. ACM.

[101] T. Sterling, D. J. Becker, D. Savarese, J. E. Dorband, U. A. Ranawake, and C. V. Packer. Beowulf: A parallel workstation for scientific computation. In *In Proceedings of the 24th International Conference on Parallel Processing*, pages 11–14. CRC Press, 1995.

[102] J. Swalwell, F. Ribalet, and E. Armbrust. Seaflow: A novel underway flow-cytometer for continuous observations of phytoplankton in the ocean. *Limnology & Oceanography Methods*, 9:466–477, 2011.

[103] C. A. Thekkath, H. M. Levy, and E. D. Lazowska. Separating data and control transfer in distributed operating systems. *SIGOPS Oper. Syst. Rev.*, 28(5):2–11, Nov. 1994.

[104] M. E. Thomadakis. The architecture of the Nehalem processor and Nehalem-EP smp platforms. Technical report, December 2010. `http://sc.tamu.edu/systems/eos/nehalem.pdf`.

[105] J. E. Thornton. The CDC 6600 project. *IEEE Ann. Hist. Comput.*, 2(4):338–348, Oct. 1980.

[106] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. Res. Dev.*, 11(1):25–33, Jan. 1967.

[107] Top500. Top 500 supercomputer sites. http://www.top500.org/, 2010.

[108] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ISCA '95, pages 392–403, New York, NY, USA, 1995. ACM.

[109] R. V. van Nieuwpoort, T. Kielmann, and H. E. Bal. Efficient load balancing for wide-area divide-and-conquer applications. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, PPoPP '01, pages 34–43, New York, NY, USA, 2001. ACM.

[110] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: a user-level network interface for parallel and distributed computing (includes url). In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP '95, pages 40–53, New York, NY, USA, 1995. ACM.

[111] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, pages 256–266, New York, NY, USA, 1992. ACM.

[112] K. B. Wheeler, R. C. Murphy, and D. Thain. Qthreads: An API for programming with millions of lightweight threads. In *IPDPS*, pages 1–8. IEEE, 2008.

[113] A. X. Widmer and P. A. Franaszek. A DC-balanced, partitioned-block, 8B/10B transmission code. *IBM J. Res. Dev.*, 27(5):440–451, Sept. 1983.

[114] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 13–24, New York, NY, USA, 2013. ACM.

[115] J. Yang and J. 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.

[116] J. W. Young. A first order approximation to the optimum checkpoint interval. *Commun. ACM*, 17(9):530–531, Sept. 1974.

[117] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.