Latency-Tolerant Software Distributed Shared Memory
Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, Mark Oskin

Abstract
We present Grappa, 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 MapReduce framework (10× faster than Spark [74]); a vertex-centric framework inspired by GraphLab (1.33× faster than native GraphLab [48]); and a relational query execution engine (12.5× faster than Shark [31]). All these frameworks required only 60-690 lines of Grappa code.

1 Introduction
Data-intensive applications (e.g., ad placement, social network analysis, PageRank, etc.) make up an important class of large-scale computations. Typical hardware computing infrastructures for these applications are a collection of multicore nodes connected via a high-bandwidth commodity network (a.k.a. a cluster). Scaling up performance requires careful partitioning of data and computation; i.e., programmers have to reason about data placement and parallelism explicitly, and for some applications, such as graph analytics, partitioning is difficult. This has led to a diverse ecosystem of frameworks—MapReduce [26], Dryad [43], and Spark [74] for data-parallel applications, GraphLab [48] for certain graph-based applications, Shark [31] for relational queries, etc. They ease development by specializing to algorithmic structure and dynamic behavior; however, applications that do not fit well into one particular model suffer in performance.

Software distributed shared memory (DSM) systems provide shared memory abstractions for clusters. Historically, these systems [15, 19, 45, 47] 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 access. 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 [27, 51], 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 Tera MTA [10,11], a custom hardware-based system. 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: a global address space (§3.1), lightweight user-level tasking (§3.2), and an aggregating communication layer (§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.

The next section provides an overview of how data-intensive application frameworks can easily and effi-
ciently map to a shared-memory programming model. §3 describes the Grappa system. §4 presents a quantitative evaluation of the Grappa runtime. §5 describes related work, and §6 concludes.

2 Data-Intensive Application Frameworks

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 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. Reduce is a parallel loop over all the keys in the hash table.

**Vertex-centric.** GraphLab/PowerGraph is an example of a vertex-centric execution model, designed for implementing machine-learning and graph-based applications [35, 48]. Its three-phase gather-apply-scatter (GAS) API for vertex programs enables 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:

**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 [35]. 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—e.g., between phases of processing a large chunk of data. Conversely, graph-parallel applications exhibit fine-grain concurrency with frequent synchronization—e.g., 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: their abundant data parallelism enables high degrees of concurrency; and their performance depends not on the latency of execution of any specific parallel task/thread, as it would in for example a web server, but rather on the aggregate execution time (i.e., throughput) of all tasks/threads. In the next section we explore how these application properties can be exploited to implement an efficient DSM.

3 Grappa Design

Figure 2 shows an overview of Grappa’s DSM system. Before describing the Grappa system in detail, we describe its three main components:
which provides automated load balancing. Concurrency (i.e., remote memory access) automatically suspend while provides fine-grain access to data anywhere in the system. processes to global memory from the memory’s home core, of things in Grappa: the stack associated with a task, accesses to global memory from the memory’s home core.

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 cooperatively-scheduled user-level threads. 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.

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 [69]. 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 [50] 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,
When programmers want to operate on data structures

While delegates can trivially implement

with their updates visible to all cores across the system

Accessing global memory though delegate operations all-

ows us to provide a familiar memory model. All synchro-

ions are done via delegate operations. Since delegate

ations 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 [38],

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 visi-
bale 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 pro-
grams [5], which is what underpins C/C++ [17, 44]. 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 user-level worker
that is scheduled within an operating system thread; we
refer to these as workers to avoid confusion. 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 speci-
fied with a function pointer and explicit arguments, a C++
struct that overloads the parentheses operator, or a C++11
lambda construct. These objects, typically small (~ 32
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 collec-
tion 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 work-
ers ready to execute called the ready worker queue. Each

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 ver-
sion. Delegate operations [49, 53] provide this capability.

While delegates can trivially implement read/write opera-
tions to global memory, they can also implement more complex read-modify-write and synchronization opera-
tions (e.g., fetch-and-add, mutex acquire, queue insert).
Figure 4 shows an 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 ob-
tain performance. In other work we have developed a
compiler [40] that automatically identifies and extracts
productive delegate operations from ordinary code.

A delegate operation can execute arbitrary code pro-
duced 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. Given these restric-
tions, 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 al-
ows us to provide a familiar memory model. All synchro-
nization 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 [38],
with their updates visible to all cores across the system

Figure 4: Grappa delegate example.
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.

**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 [12] 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 context data 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 450 ns for a few threads and 800 ns for 1000–32000 threads.

**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. Grappa’s parallel loops spawn tasks using a recursive decomposition of iterations, similar to Cilk’s cilk_for construct [16], and TBB’s parallel_for [59]. 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 [69]. 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 5 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
which the core is responsible; if the list is long enough to minimize cache pollution.

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 6 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 §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 6(left) shows how aggregation allows Grappa to exceed the performance of the card by \(25 \times 128\) nodes. We measured the effective bisection bandwidth of the cluster as described in [39]: for GUPS, performance is limited by memory bandwidth during aggregation, and uses \(~ 40\%\) of available bisection bandwidth.

Figure 6(right) illustrates why using RDMA directly is not sufficient. The data also shows that MPI over InfiniBand has negligible overhead. 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 [42]. 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 [34]. Furthermore, RDMA network cards have severely limited support for synchronization with the CPU [27, 51]. 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 [1, 4, 28, 34, 55, 57, 61, 68]: 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 [22, 60, 62] 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.
[29]. 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 [73]; 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 mechanism in [48]. 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.

4 Evaluation

We implemented Grappa in C++ for the Linux operating system. The core runtime system is 17K lines of code. We ran experiments on a cluster of AMD Interlagos processors with 128 nodes. Nodes have 32 cores operating at 2.1GHz, spread across two sockets, 64GB of memory, and 40Gb Mellanox ConnectX-2 InfiniBand network cards. Nodes are connected via a QLogic InfiniBand switch with no oversubscription. We used a stock OS kernel and device drivers. The experiments were run in a machine without administrator access or special privileges. GraphLab and Spark communicated using IP-over-InfiniBand 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 [35, 48] using the graph data structure provided by the Grappa library (Figure 3). 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 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 from GraphBench [3] 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 [37], 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 [46] (41 M vertices, 1 B directed edges) and the Friendster social network [72] (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 7a 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 $2.57 \times$), and $1.33 \times$ 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.
API rather than Spark’s user-level API.

We used Grappa to build a distributed backend to Raco, a relational algebra compiler and optimization framework [58]. 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.

4.2 Relational queries on Grappa

We used Grappa to build a distributed backend to Raco, a relational algebra compiler and optimization framework [58]. 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 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 [2]. 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 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 [71].

Our particular approach for the Grappa backend to Raco is source-to-source translation. We generate forall loops for each pipeline in the physical query plan. We extend the code generation approach for serial code in [54] 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, 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.

We extend the code generation approach for (including delegate operations), over the 85 iterations, making 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 9a. Grappa has a geometric mean speedup of $12.5 \times$ 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 9b) 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 7c) sustains (200MB/s/node), but its sustained bandwidth is just over half this amount (116 MB/s/node).

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 loop over inputs followed by a forall loop 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, us-
We see similar results as with Shark: the bulk of the computation is spent assigning points to clusters in the map step. At large K, 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 Spark k-means 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 [66], 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 10 for $K = 10$ and $K = 10000$. We find Grappa-MapReduce to be nearly an order of magnitude faster than the comparable Spark implementation. Absolute runtime for Grappa-MapReduce is 0.13s per iteration for $K = 10$ and 17.3s per iteration for $K = 10000$, compared to 1s and 170s respectively for Spark.

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. At large $K$, Grappa-MapReduce is clearly compute-bound, but Spark spends only 50% of its time on compute; the rest is in network failures. We run k-means on a dataset from Seaflow [66], respectively for Spark.

Figure 9: Relational query performance.

Figure 10: Data parallel experiments using k-means on a 8.9GB Seaflow dataset with 64 nodes.

Figure 11: Scaling BFS out to 128 nodes. In addition to Grappa’s GraphLab engine, we also show a custom algorithm for BFS implemented natively which employs Beamer’s bottom-up optimization to achieve even better performance.

code in the reduce step. 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 [13]. 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 11 show that this algorithm’s
performance is nearly a factor of 2 better than the pure Vertex-centric abstraction can achieve.

5 Related Work

Multi threading. Hardware-based implementations of multithreading to tolerate latency include the Denelcor HEP [65], Tera MTA [11], Cray XMT [33], Simultaneous multithreading [67], MIT Alewife [6], Cyclops [9], and GPUs [32]. 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) [24]. 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 [25] was that threading for latency tolerance was fundamentally limited because the latency of the top-level store (e.g. L1 cache) is in direct competition with the number of contexts that can fit in it. However, we 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 [70], TBB [59], Cilk [16] and Capriccio [14] is Grappa’s context prefetching. Grappa’s prefetching could likely improve from compiler analyses inspired by those of Capriccio for reducing memory usage.

Software distributed shared memory. Much of the innovation in DSM over the past 30 years has focused on reducing the synchronization costs of updates. The first DSM systems, including IVY [47], used frequent invalidations to provide sequential consistency, inducing high communication costs for write-heavy workloads. Later systems relaxed the consistency model to reduce communication demands; some systems further mitigated performance degradation due to false sharing by adopting multiple writer protocols that delay integration of concurrent writes made to the same page. The Munin [15, 19] and TreadMarks [45] systems exploited both of these ideas, but still incurred some coherence overhead. Munin and Blizzard [64] allowed the tracking of ownership with variable granularity to reduce the cost due to false sharing. Grappa follows the lead of TreadMarks and provides DSM entirely at user-level through a library and runtime. FaRM [27] 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.

Partitioned Global Address Space languages. The high-performance computing community has largely discarded the coherent distributed shared memory approach in favor of the Partitioned Global Address Space (PGAS) model. Examples include Split-C [23], Chapel [20], X10 [21], Co-array Fortran [56] and UPC [30]. What is most different between general DSM systems and PGAS ones is that remote data accesses are explicit, thereby encouraging developers to use them judiciously. Grappa follows this approach, implementing a PGAS system at the language level, thereby facilitating compiler and programmer optimizations.

Distributed data-intensive processing frameworks. There are many other data-parallel frameworks like Hadoop, Haloo [18], and Dryad [43]. 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 synchronous communication and distributed key-value storage built from RPCs [7, 8]. The incremental data-parallel system Naiad [52] 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.

6 Conclusions

Our work builds on the premise that writing data-intensive applications and frameworks in a shared memory environment is simpler than developing custom infrastructure from scratch. To that end, Grappa is inspired not by SMP systems, but by novel supercomputer hardware – the Cray MTA and 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. Based on this premise, we show that a DSM system can be efficient for this application space by judiciously exploiting the key application characteristics of concurrency and latency tolerance. Our data demonstrates that frameworks such as MapReduce, vertex-centric computation, and query execution are easy to build and efficient. 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.
Acknowledgements

This work was supported by NSF Grant CCF-1335466, Pacific Northwest National Laboratory, and gifts from NetApp and Oracle.

References


