

Latency-Tolerant Software Distributed Shared Memory

Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, Mark Oskin

University of Washington
Department of Computer Science and Engineering

{nelson, bholt, bdmyers, preston, luisceze, skahan, oskin}@cs.washington.edu

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 map/reduce framework ($10\times$ faster than Spark [74]); a vertex-centric framework inspired by GraphLab ($1.33\times$ faster than native GraphLab [47]); and a relational query execution engine ($12.5\times$ faster than Shark [30]). 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 [25], Dryad [42], and Spark [74] for data-parallel applications, GraphLab [47] for certain graph-based applications, Shark [30] 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 [14, 18, 44, 46] 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 [26, 50], 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 [9, 10], 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 internode 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 (§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 map/reduce 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 $2.5\times$ better than GraphLab’s random partitioning and $1.33\times$ 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 $12.5\times$ 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.

In the next section we provide an overview of how data-intensive application frameworks can easily and efficiently

map to a shared-memory programming model. Then in §3 we describe the GRAPPA system. In §4 we present a quantitative evaluation of the GRAPPA runtime. §5 describes related work, and finally §6 concludes.

2. Frameworks for Data-Intensive Applications

Analytics frameworks – such as map/reduce, 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.

Below 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:

Map/Reduce 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 is an example of a vertex-centric execution model, designed for implementing machine-learning and graph-based applications [34, 47]. 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:

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 [34]. 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:

Concurrency Data-intensive applications have abundant data parallelism.

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.

In the next section we explore how these application properties can be exploited to implement an efficient DSM by describing how GRAPPA’s design addresses the challenges outlined earlier.

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:

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* [35]. Due to delegation, the memory model offered is similar to what underpins C/C++ [16, 43], so it is familiar to programmers.

```

// 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;
    });
});

```

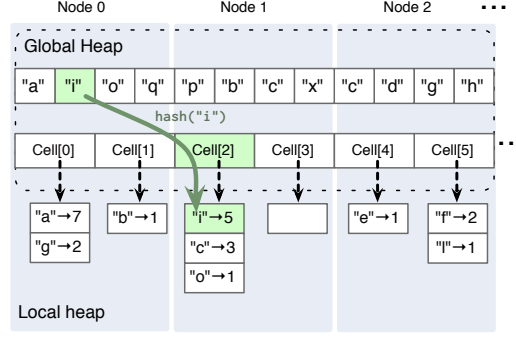


Figure 1: “Character count” with a simple hash table implemented using GRAPPA’s distributed shared memory.

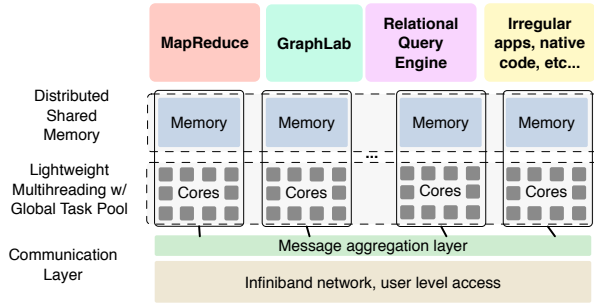


Figure 2: GRAPPA design overview

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 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.

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 [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 [49] 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 [29]) 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 [40] explored how this was used to implement GRAPPA’s synchronized global data structures, including vector and hash map.

Putting it all together Figure 3 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, 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 [48, 52] provide this capability.

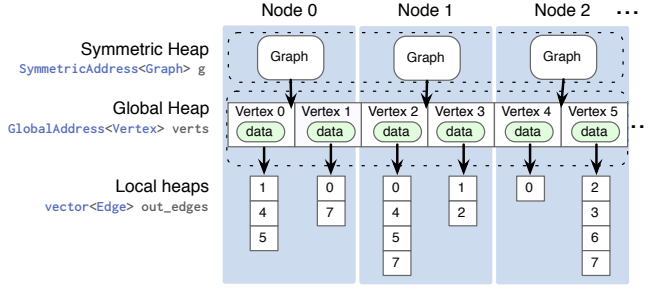


Figure 3: Using global addressing for graph layout.

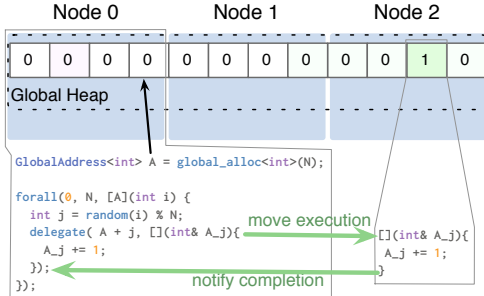


Figure 4: GRAPPA delegate example.

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 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 obtain performance. In other work we have developed a compiler [39] 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 [37], 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 [4] (all accesses to shared data are separated by synchronization), which is what underpins C/C++ [16, 43]. 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 ready to execute, it is mapped to a *worker* thread. Scheduling between tasks is carried out entirely in user-mode without operating system intervention.

Tasks Tasks are specified by a closure (or “function object” in C++ parlance) that holds both code to execute and initial state. The functor 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 functors can be serialized and transported around the system, and are eventually executed by a worker thread.

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 functor 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 [11] 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 thread 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 threads that are *ready-resident*; all other threads 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 threads.

When a thread is signaled, its reference is marked *ready-unscheduled*. Every time the scheduler runs, one of its responsibilities is to pick a *ready-unscheduled* thread to transition to *ready-scheduled*: it issues a software prefetch to start moving the task toward L1. A thread 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 thread 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 ($\approx 1,000$ 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 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 worker threads on each core. Even if there are many active threads, if they are all suspended on long-latency operations, then the core is under-utilized. The stealing policy must predict whether local tasks will likely generate enough new work soon; a similar problem is addressed in [67]. 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 threads, with optional data locality constraints. These threads may be full-fledged threads 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 [15], and TBB’s `parallel_for` [58]. 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.

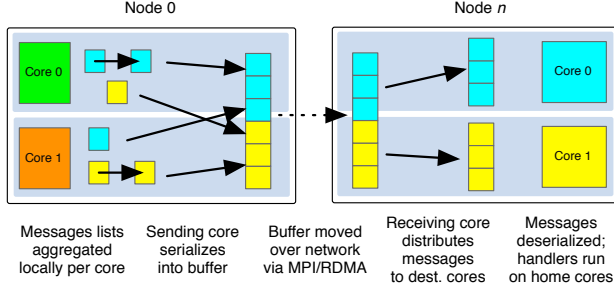


Figure 5: Message aggregation process.

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 functor. 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 functor, 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 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 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 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× at 128 nodes. We measured the effective bisection bandwidth of the cluster as described in [38]: 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 [41]. 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 [33]. Furthermore, RDMA network cards have severely limited support for synchronization with the CPU [26, 50]. Finally, framing overheads can be large: InfiniBand 8-byte RDMA writes moves 50 bytes on the wire; Ethernet-based RDMA using

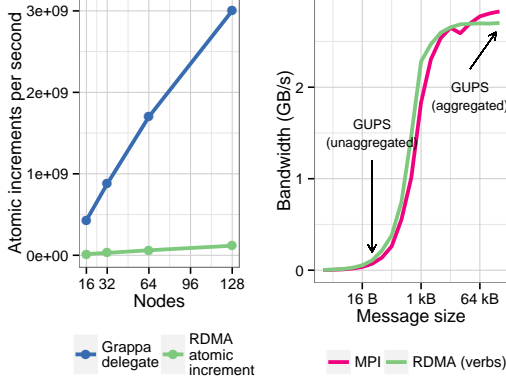


Figure 6: On the left, random updates to a billion-integer distributed array with the GUPS benchmark. On the right, ping-pong bandwidth measured between two nodes.

RoCE moves 98 bytes. Work is ongoing to improve network card small message performance [1, 3, 27, 33, 54, 56, 60, 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 [21, 59, 61] 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 [28]. 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 [47].

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 [34, 47] 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 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 [36], 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 [45] (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 dif-

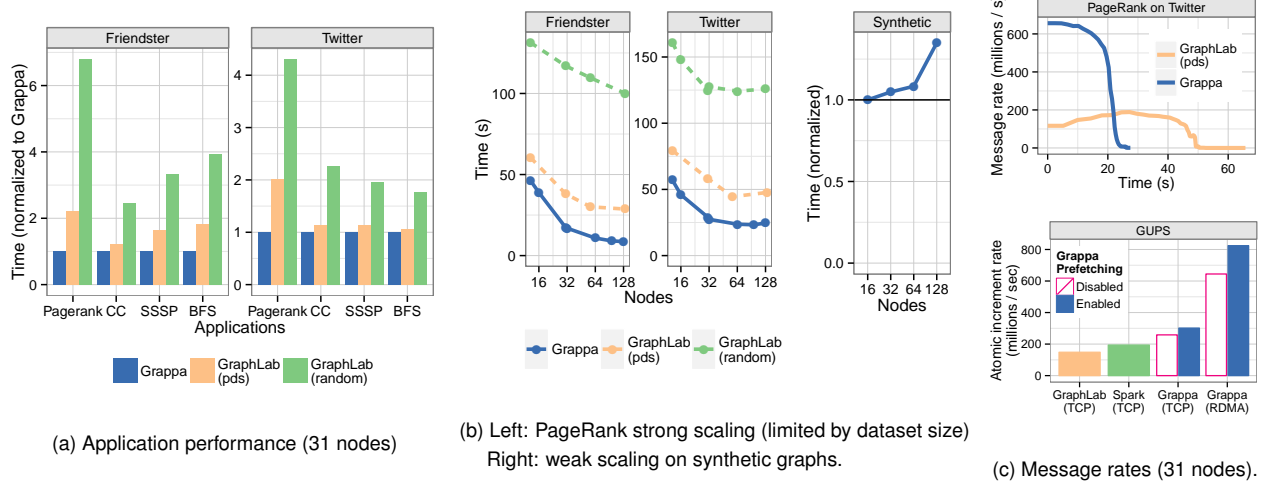


Figure 7: Performance characterization of GRAPPA’s Vertex-centric framework (a) shows time to converge (same number of iterations) normalized to GRAPPA, on the Twitter and Friendster datasets. (b) shows scaling results for PageRank out to 128 nodes – Friendster and Twitter measure *strong* scaling, and *weak* scaling is measured on synthetic power-law graphs scaled proportionally with nodes. (c) On top, cluster-wide message rates (average per iteration) while computing PageRank. On the bottom, 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.

ference 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.

Figure 7c(bottom) explores this difference using the GUPS benchmark from §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 $2\times$ as many messages as

GraphLab’s replicated representation. However, as can be seen in Figure 7c(top), GRAPPA sends these messages at up to $4\times$ 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 $2\times$ speedup.

Figure 8 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 §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 7b(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 7b(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.

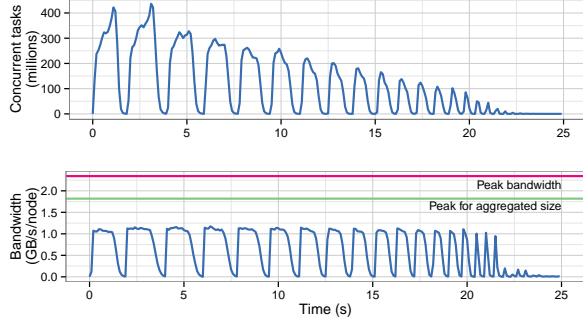


Figure 8: GRAPPA PageRank execution over time on 32 nodes. The top shows the total number of concurrent tasks (including delegate operations), over the 85 iterations, peaks diminish as fewer vertices are being updated. The bottom shows message bandwidth per node, which correlates directly with the concurrency at each time step, compared against the peak bandwidth, and the bandwidth for the given message size.

4.2. Relational queries on GRAPPA

We used GRAPPA to build a distributed backend to RelComp, a relational algebra compiler and optimization framework [57]. RelComp 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 [71].

Our particular approach for the GRAPPA backend to RelComp is source-to-source translation. We generate forall for each pipeline in the physical query plan. We extend the code generation approach for *serial* code in [53] 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 RelComp. 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 [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 ran conjunctive queries from SP²Bench [62]. 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 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.

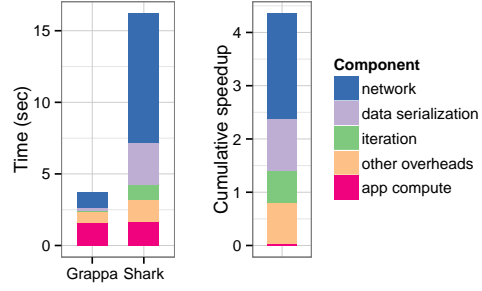
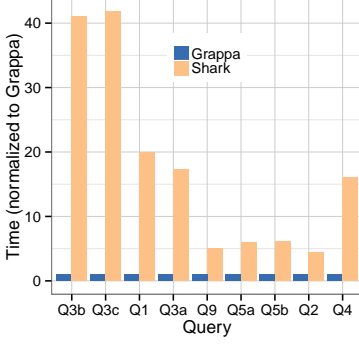
There are many differences between the two runtime systems (e.g. messaging layers, JVM and native) and the query processing approach (e.g. iterators vs compiled code), making it challenging to clearly understand the source of the performance difference between the two systems. To do so, we computed a 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).

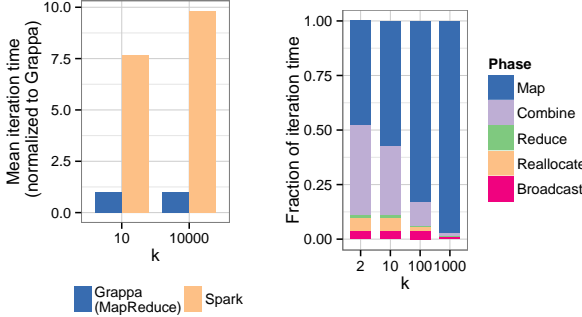
4.3. Iterative MapReduce on GRAPPA

We experiment with data parallel workloads by implementing an in-memory MapReduce API in 152 lines of GRAPPA code. The implementation involves a forall 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



(a) The SP²Bench benchmark on 16 nodes. Query Q4 is a large workload so it was run with 64 nodes. (b) Performance breakdown of speedup of GRAPPA over Shark on Q2.

Figure 9: Relational query performance.



(a) Performance for 64 nodes

(b) Breakdown of time spent in MapReduce phases for GRAPPA-MapReduce

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

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 [65], 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.

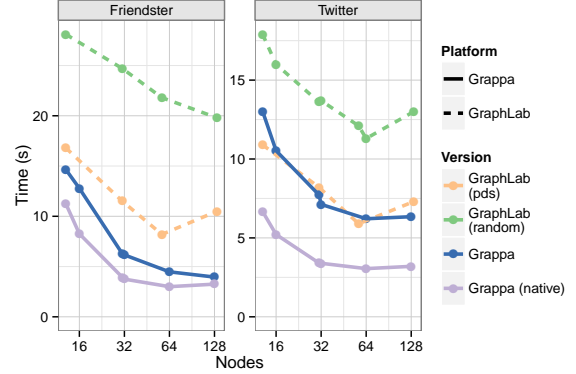


Figure 11: Scaling BFS out to 128 nodes. In addition to GRAPPA’s GraphLab engine, we also show a custom algorithm for BFS implemented natively which employs Beamer’s bottom-up optimization to achieve even better performance.

The results are shown in Figure 10a 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. Figure 10b 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 [12]. 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

Multithreading Hardware-based massive multithreading to tolerate latency include the Denelcor HEP [64], Tera MTA [10], Cray XMT [32], Simultaneous multithreading [66], MIT Alewife [5], Cyclops [8], and GPUs [31]. 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) [23]. 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 [24] 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 [58], Cilk [15] and Capriccio [13] 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 [46], 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. Release consistency, for example, allows updates to be buffered between synchronization events. 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 [14, 18] and TreadMarks [44] systems exploited both of these ideas, but still incurred some coherence overhead. Paging strategies have presented an additional opportunity for innovation in reducing update cost: ownership and transmission of large pages make better use of processor page management mechanisms and network wire bandwidth when locality is abundant, but otherwise result in increased false sharing and wasted bandwidth moving large pages. Munin and Blizzard [63] allowed the tracking of ownership with variable granularity to address these problems. GRAPPA follows the lead of TreadMarks and provides DSM entirely at user-level through a library and runtime. FaRM [26] 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 [22], Chapel [19], X10 [20], Co-array Fortran [55] and UPC [29]. 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, Haloop [17], and Dryad [42]. 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 [6, 7]. The incremental data-parallel system Naiad [51] 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.

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 map/reduce, Vertex-centric execution, and query execution are *easy to build* and *efficient*. Our map/reduce and query execution implementations are an order of magnitude faster than the custom frameworks for each. Our vertex-centric GraphLab-inspired API is $1.33\times$ faster than GraphLab itself, without the need for complex graph partitioning schemes.

References

- [1] Intel Data Plane Development Kit. <http://goo.gl/AOvjss>, 2013.
- [2] Big data benchmark. <https://amplab.cs.berkeley.edu/benchmark>, Feb 2014.
- [3] Snabb Switch project. <https://github.com/SnabbCo/snabbswitch>, May 2014.
- [4] S. V. Adve and M. D. Hill. Weak ordering – A new definition. In *ISCA-17*, 1990.
- [5] 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.
- [6] 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.
- [7] 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.
- [8] 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.
- [9] 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.
- [10] 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.
- [11] AMD64 ABI. <http://www.x86-64.org/documentation/abi-0.99.pdf>, July 2012.
- [12] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. In *Conference on Supercomputing (SC-2012)*, November 2012.
- [13] R. V. Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: Scalable threads for internet services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 268–281. ACM Press, 2003.
- [14] 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.
- [15] 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.
- [16] 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.
- [17] 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.
- [18] 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.
- [19] 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.
- [20] 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.
- [21] Y. Chen, S. Alsbaugh, 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.
- [22] 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.
- [23] D. E. Culler, S. C. Goldstein, K. E. Schauer, and T. von Eicken. TAM – A compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, 18:347–370, July 1993.
- [24] D. E. Culler, K. E. Schauer, 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.
- [25] 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.
- [26] 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.
- [27] 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.
- [28] J. Duell, P. Hargrove, and E. Roman. The design and implementation of Berkeley Lab's Linux checkpoint/restart. Technical Report LBNL-54941, December 2002.
- [29] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. *UPC: Distributed Shared Memory Programming*. John Wiley and Sons, Inc., Hoboken, NJ, USA, 2005.
- [30] 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.
- [31] K. Fatahalian and M. Houston. A closer look at GPUs. *Communications of the ACM*, 51:50–57, October 2008.
- [32] 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.
- [33] 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.

- [34] 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.
- [35] 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.
- [36] Graph 500. <http://www.graph500.org/>, July 2012.
- [37] 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.
- [38] 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.
- [39] B. Holt, P. Briggs, L. Ceze, and M. Oskin. Alembic. In *International Conference on PGAS Programming Models (PGAS)*, Oct 2013.
- [40] 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.
- [41] InfiniBand Trade Association. InfiniBand Architecture Specification, Version 1.2.1. 2007.
- [42] 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.
- [43] 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.
- [44] 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.
- [45] 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.
- [46] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, Nov. 1989.
- [47] 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.
- [48] R. Lublinerman, J. Zhao, Z. Budimlic, S. Chaudhuri, and V. Sarkar. Delegated isolation. In *OOPSLA'11*, pages 885–902, 2011.
- [49] Message Passing Interface Forum. Mpi: A message-passing interface standard, version 2.2. Specification, September 2009.
- [50] 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.
- [51] 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.
- [52] 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.
- [53] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, June 2011.
- [54] 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.
- [55] R. W. Numrich and J. Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, Aug. 1998.
- [56] 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.
- [57] Raco: The relational algebra compiler. <https://github.com/uwescience/datalogcompiler>, April 2014.
- [58] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly Media, 2007.
- [59] 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.
- [60] 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.
- [61] 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.
- [62] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. Sp2bench: A sparql performance benchmark. *CoRR*, abs/0806.4627, 2008.
- [63] 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.
- [64] 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.
- [65] J. Swallow, 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.
- [66] 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.
- [67] 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.
- [68] 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.
- [69] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauer. 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.
- [70] 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.
- [71] 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.
- [72] 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.
- [73] J. W. Young. A first order approximation to the optimum checkpoint interval. *Commun. ACM*, 17(9):530–531, Sept. 1974.
- [74] 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.