Trading Latency for Performance in Data-Intensive Applications

JACOB NELSON, BRANDON HOLT, BRANDON MYERS, PRESTON BRIGGS, SIMON KAHAN, LUIS CEZE, MARK OSKIN

Jacob Nelson received a PhD in computer science from the University of Washington in 2014. His research interests include computer architecture and runtime systems for big data and high-performance computing. nelson@cs.washington.edu

Brandon Holt is a PhD student in Computer Science and Engineering at the University of Washington, advised by Luis Ceze and Mark Oskin. He is interested in programming models, compilers, and systems for clusters, especially abstractions to mitigate real-world challenges like high contention. bholt@cs.washington.edu

Brandon Myers is a PhD candidate and Lecturer in Computer Science and Engineering at the University of Washington, advised by Bill Howe and Mark Oskin. He is interested in building systems to enable fast and flexible parallel programming, at the intersection of high performance computing, data management, and architecture. bdmyers@cs.washington.edu

Preston Briggs is a Senior Engineer at Reservoir Labs and an Affiliate Professor in Computer Science and Engineering at the University of Washington. He received a PhD in computer science from Rice University in 1992. preston@cs.washington.edu

The rising importance of data-intensive applications has fueled the growth of a plethora of distributed computing frameworks, including Hadoop, Spark, and GraphLab. We have developed a system called Grappa [1, 2] to aid programmers in developing new frameworks. Grappa provides a distributed shared memory abstraction to hide complexity from the programmer, and takes advantage of parallelism in the data to hide remote access latency and to trade latency for more performance. These techniques allow it to outperform existing frameworks by up to an order of magnitude.

Data-Intensive Applications on Distributed Shared Memory

Software distributed shared memory (DSM) systems provide shared memory abstractions for clusters. Historically, these systems 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, 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-grained synchronization—a poor fit for many modern data-intensive applications.

DSM offers the promise of simpler implementations of data-intensive application frameworks. Figure 1 shows a minimal example of 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 runs on all nodes to process shards of the input array, hashing each key to its cell and incrementing the corresponding count atomically. The code looks similar to plain shared-memory code, yet it spans multiple nodes and scales efficiently.

Applying the DSM concept to common data-intensive computing frameworks is similarly straightforward:

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. 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 comes from 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

Luis Ceze is an Associate Professor of Computer Science and Engineering at the University of Washington. His research focuses on improving programmability, reliability, and energy efficiency of multiprocessor and multicore systems. luisceze@cs.washington.edu

Simon Kahan is an Affiliate Professor of Computer Science and Engineering at the University of Washington. His current research focuses on accelerating large-scale biological simulation and numerical linear algebra. skahan@cs.washington.edu

Mark Oskin is an Associate Professor of Computer Science and Engineering at the University of Washington. oskin@cs.washington.edu

```
// 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;
  });
});
```

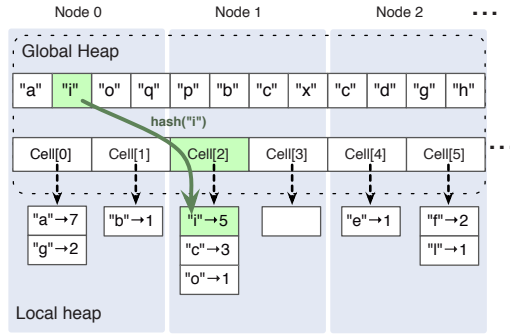

Figure 1: "Character count" with a simple hash table implemented using Grappa's distributed shared memory

table, producing the results. These steps rely heavily on consistent, fine-grained updates to hash tables.

While these frameworks are easy to express conceptually in a DSM system, obtaining good performance can be challenging for a number of reasons:

Small messages. Programs written to a shared memory model tend to access small pieces of data. On a DSM system this requires communication. What were simple load or store operations become implicit, complex transactions involving the network. When these messages are small (~32 bytes), the network (optimized for multi-kilobyte packets) struggles to achieve a fraction of its peak throughput.

Poor locality. Data-intensive applications often exhibit poor locality. For example, the volume of communication in GraphLab's gather and scatter operations is a function of the graph partition. Complex graphs frustrate even the most advanced partitioning schemes. 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-grained 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-grained 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-grained 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, as it would in, for example, a Web server, but rather on the aggregate execution time (i.e., *throughput*) of *all* tasks.

Grappa Design

Figure 2 shows an overview of Grappa's DSM system. We will first describe the multithreading and communication layers and then explore the distributed shared memory layer, which is built on top of these lower-level components. Our recent USENIX ATC paper [2] describes these in more detail.

Expressing and Exploiting Parallelism

Work is most commonly expressed in Grappa using parallel `for` loops. Tasks may also be spawned individually, with optional data locality constraints. Under the hood, both methods

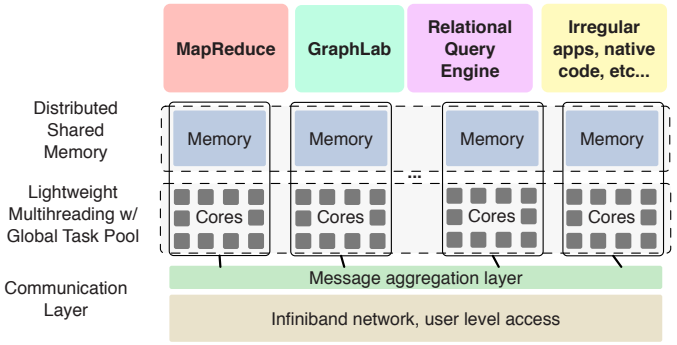

Figure 2: Grappa's distributed shared memory abstraction is designed to make it easy to implement data-intensive application frameworks. It uses lightweight threads to tolerate remote access latencies by exploiting fine-grained parallelism in the data, and it transparently aggregates small messages into larger ones to improve communication performance.

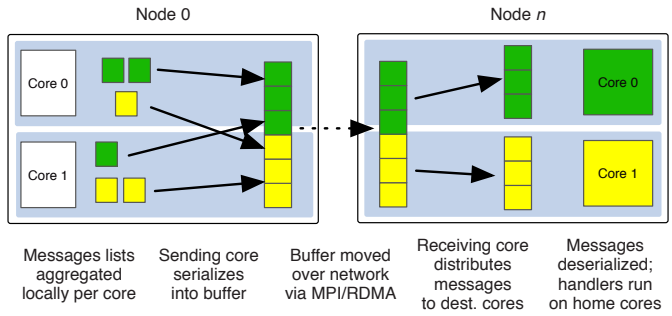

Figure 3: Grappa achieves high throughput for small messages by automatically batching messages with a common destination in order to move larger packets over the network, amortizing network invocation and delivery costs over multiple messages.

work by pushing closures into a global task pool. These closures are generally expressed using C++11 lambda constructs to provide both code to execute and initial state. Tasks are executed by idle threads on cores across the system, which pull from the global task queue subject to the tasks' locality constraints. When a task executes a long-latency operation, it is suspended until the operation is complete; the core it is running on is kept busy with other, independent, work.

Grappa is built around a user level, cooperative multithreading system. Due to the large inter-node latencies that must be tolerated in a distributed system like Grappa, the scheduler is built to support on the order of 1000 concurrent threads per core. We do this by storing and switching minimal context for threads, and by prefetching thread contexts into cache before switching to them, thereby enabling context switches to happen at a rate limited only by DRAM bandwidth, rather than cache miss latency.

Communication Support

Grappa's communication layer has two components. The upper (user-level) layer is designed to support sending very small messages—tens of bytes—at a high rate, with low memory overhead. We use an asynchronous active message approach: the sender creates a message holding a C++11 lambda or other closure, and the receiver executes the 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 dynamically sized data payload.

At the lower (network) level, Grappa moves these small messages over the network efficiently by transparently aggregating independent messages destined for common network destinations. This process, shown in Figure 3, works as follows. When a compute task sends a message, the data is not immediately placed on the network but instead is stored in a per-core buffer. A communication task runs periodically; when it finds a large group of messages headed for the same node, or messages that have been waiting for a long time, it serializes them into a single, large network packet, which it sends to the destination node. When the remote node receives the packet, it distributes the messages to their destination cores, where messages are deserialized and their handlers are executed.

Grappa uses RDMA to move messages, but only indirectly. User-level messages are created using non-temporal memory operations and prefetches to avoid cache pollution. Aggregated messages are moved between nodes using MPI for portability, tuned to use RDMA when available. By amortizing network invocation costs across many messages, we are able to obtain significantly better performance than using native RDMA operations: on a simple random-access benchmark, Grappa's DSM operations performed atomic increments 25 times faster than native RDMA increments on our 128-node AMD Interlagos cluster connected with 40 Gb Mellanox ConnectX-2 InfiniBand cards.

Addressing in Grappa's Distributed Shared Memory

In Grappa, memory is partitioned across cores; each byte is considered local to a single core within a node in the system. Accesses to local memory occur through conventional pointers. Local pointers cannot refer to memory on other cores; they are valid only on their home core. Local accesses are used to reference many things in Grappa, including the stack associated with a task, scheduling and debugging data structures, and the slice of global memory local to a core.

Accesses to non-local memory occur through global pointers. Grappa allows any local data on a core's stacks or heap to be exported to the global address space and made accessible to other cores across the system. This uses a partitioned global address space (PGAS) model, where each address is a tuple of a core ID and an address local to that core.

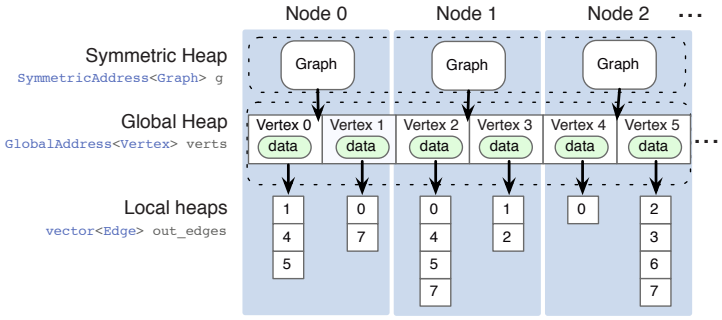

Figure 4: Using global addressing for graph layout

Grappa also supports *symmetric* allocations, which reserves space for a copy 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 for a global object, holding local copies of constant data, or allowing operations to be transparently buffered. A separate publication [3] describes how this was used to implement Grappa's synchronized global data structures, including vector and hash map.

Figure 4 shows an example of how global, local, and symmetric memory 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.

Accessing Memory with Delegate Operations

Access to Grappa's distributed shared memory is provided through *delegate* operations, which are short operations performed at a memory location's home core. 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. 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).

We have explored two approaches for expressing delegate operations. In the first, the programmer calls functions in Grappa's API—a change from the traditional DSM model. Generally, these delegates are expressed as C++11 lambdas or other closures; Figure 5 shows an example. The second approach uses a compiler pass implemented with LLVM to automatically identify and extract productive delegate operations from ordinary code; this approach is explored in another publication [4]. In practice, we usually use the library-based approach, since exploiting available locality is important for getting maximum performance in a distributed system, and writing explicit delegate operations is an easy way to express that locality.

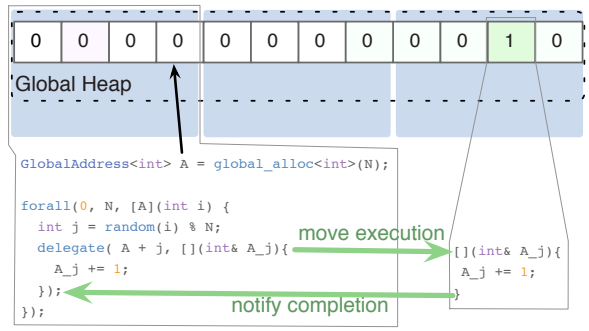

Figure 5: Grappa delegate example

Delegates and Memory Consistency

Memory consistency and efficient synchronization are a result of delegation in Grappa.

All sharing, whether between cores within a node or between two nodes, as well as synchronization, is done via delegate operations. A delegate operation can execute arbitrary code subject to two restrictions: first, the code can reference only data local to the core on which the delegate is executing; and second, the code may not execute operations that lead to a context switch.

Since delegate operations execute on a particular core in some serial order and only touch data owned by that core, they are guaranteed to be globally linearizable, 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, so 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, which is what underpins C/C++.

The synchronous property of delegates provides a clean model but can be overly restrictive for operations that are protected by collective synchronization like a global barrier. For such cases, we also support *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 and are generally linked with a collective synchronization operation to detect completion.

Measuring Performance with Prototype Application Frameworks

We implemented three prototype application frameworks in Grappa. The first is an in-memory MapReduce implementation, which we compared with Spark [5] with fault tolerance disabled. The second is a distributed backend for the Raco relational algebra compiler and optimization framework [6], which we compared with Shark [7]. The third is a vertex-centric programming framework in the spirit of GraphLab [8], which we compare with native GraphLab.

The full performance results are reported in our USENIX ATC paper [2]; here we provide a brief summary. On the cluster mentioned previously, we found the Grappa MapReduce implementation to be 10 times faster than Spark on a k-means clustering benchmark. The Grappa query processing engine was 12.5 times faster than Shark on the SP2Bench benchmark suite [9]. The Grappa vertex-centric framework was 1.33 times faster than GraphLab on graph analytics benchmarks from the GraphBench suite [10].

Conclusion

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. 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 work demonstrates that frameworks such as MapReduce, vertex-centric computation, and query execution can be easy to build and are efficient in a DSM system.

Acknowledgments

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

References

[1] Grappa Web site and source code: http://grappa.io/.

[2] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin, "Latency-Tolerant Software Distributed Shared Memory," in *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC '15),* Santa Clara, CA.

[3] Brandon Holt, Jacob Nelson, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin, "Flat Combining Synchronized Global Data Structures," International Conference on PGAS Programming Models (PGAS), October 2013.

[4] Brandon Holt, Preston Briggs, Luis Ceze, and Mark Oskin, "Alembic: Automatic Locality Extraction via Migration," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '14)*, 2014.

[5] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica, "Spark: Cluster Computing with Working Sets," in *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud '10),* 2010.

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

[7] Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica, "Shark: SQL and Rich Analytics at Scale," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13),* 2013.

[8] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin, "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI '12),* 2012.

[9] Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel, "SP2Bench: A SPARQL Performance Benchmark," *Computing Research Repository*, abs/0806.4627, 2008.

[10] GraphBench: http://graphbench.org/, 2014.