Grappa: Faster data-intensive applications through latency tolerance

Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, Mark Oskin
University of Washington
May 7, 2014
In-memory data-intensive applications

- Lots of application areas:
  - Social network analysis
  - Machine learning
  - Bioinformatics
  ...

- Data size is in terabytes, not petabytes
  *fits in memory on a cluster!*

- Common element is focus on access to data, potentially with challenging access patterns
Frameworks for data intensive applications

“Pleasantly” parallel problems

Dryad

Graph analytics

GraphLab

Relational queries

And many more….
Why is everybody rolling their own?

• Specialized, restricted programming models for each application domain

• Often built from the ground up by application domain experts

• Not a lot of common infrastructure
Why is everybody rolling their own?

• Specialized, restricted programming models for each application domain

• Often built from the ground up by application domain experts

• Not a lot of common infrastructure

Could these different models share a common, general platform?
Grappa

- General infrastructure for in-memory data-intensive applications
- C++11 library that runs on your cluster
- A simple, flexible model: shared memory and threads
  - Grappa lets you program your cluster as if it was a single big machine
- Optimized for great performance in the worst case using latency tolerance
Linux / PThreads / MPI

InfiniBand network

DRAM
Core

DRAM
Core

DRAM
Core

DRAM
Core

InfiniBand network
Grappa

Linux / PThreads / MPI

InfiniBand network
In-memory Map/Reduce

Grappa

Linux / PThreads / MPI

InfiniBand network
In-memory Map/Reduce

GraphLab

Relational query engine

Grappa

Linux / PThreads / MPI

InfiniBand network

InfiniBand network
In-memory Map/Reduce
GraphLab
Relational query engine
Native Grappa code

Grappa

Linux / PThreads / MPI

InfiniBand network

Core
DRAM

Core
DRAM

Core
DRAM

Core
DRAM
In-memory Map/Reduce
GraphLab
Relational query engine
Native Grappa code

Your application here!

Grappa

Linux / PThreads / MPI

InfiniBand network

DRAM
Core
DRAM
Core
DRAM
Core
DRAM
Core
Outline

• Motivation

• Programming Grappa

• Key components

• Building frameworks on Grappa

• Other projects
Grappa’s system view

Global data

Global Tasks

Network

DRAM

Core

DRAM

Core

DRAM

Core

DRAM

Core
The key observations

Individual operation latency doesn’t matter: We care only about overall time to solution

These applications have lots of parallelism
Main idea: tolerate latency with other work
Main idea: tolerate latency with other work
Main idea: tolerate latency with other work
Outline

• Motivation

• Programming Grappa

• Key components

• Performance

• Other projects
A simple example

• Abstract example:
  • TB+ sized directed imbalanced tree
  • all memory-resident
  • traverse vertices reachable from a given start vertex
A single node, serial starting point

```c
struct Vertex {
    index_t id;
    Vertex * children;
    size_t num_children;
};
```
A single node, serial starting point

```c
struct Vertex {
    index_t id;
    Vertex * children;
    size_t num_children;
};

int main( int argc, char * argv[] ) {
    Vertex * root = create_big_tree();
    search(root);
    return 0;
}
```
A single node, serial starting point

```c
struct Vertex {
    index_t id;
    Vertex * children;
    size_t num_children;
};

void search(Vertex * vertex_addr) {
    Vertex v = *vertex_addr;

    Vertex * child0 = v.children;
    for (int i = 0; i < v.num_children; ++i) {
        search(child0+i);
    }
}

int main( int argc, char * argv[] ) {
    Vertex * root = create_big_tree();
    search(root);
    return 0;
}
```
A single node, serial starting point

```c
struct Vertex {
    index_t id;
    Vertex * children;
    size_t num_children;
};

void search(Vertex * vertex_addr) {
    Vertex v = *vertex_addr;
    Vertex * child0 = v.children;
    for(int i = 0; i < v.num_children; ++i) {
        search(child0+i);
    }
}

int main(int argc, char * argv[]) {
    Vertex * root = create_big_tree();
    search(root);
    return 0;
}
```
A single node, serial starting point

```c
struct Vertex {
    index_t id;
    Vertex * children;
    size_t num_children;
};

void search(Vertex * vertex_addr) {
    Vertex v = *vertex_addr;
    Vertex * child0 = v.children;
    for (int i = 0; i < v.num_children; ++i) {
        search(child0+i);
    }
}

int main( int argc, char * argv[] ) {
    Vertex * root = create_big_tree();
    search(root);
    return 0;
}
```
Add boiler-plate Grappa code

```c
struct Vertex {
    index_t id;
    Vertex * children;
    size_t num_children;
};

void search(Vertex * vertex_addr) {
    Vertex v = *vertex_addr;

    Vertex * child0 = v.children;
    for (int i = 0; i < v.num_children; ++i) {
        search(child0+i);
    }
}

int main( int argc, char * argv[] ) {
    init( &argc, &argv );
    run( []{
        Vertex * root = create_big_tree();
        search(root);
    });
    finalize();
    return 0;
}
```
Making graph & vertices into global structures

```c
struct Vertex {
    index_t id;
    GlobalAddress<Vertex> children;
    size_t num_children;
};

void search(GlobalAddress<Vertex> vertex_addr) {
    Vertex v = *vertex_addr;

    GlobalAddress<Vertex> child0 = v.children;
    for (int i = 0; i < v.num_children; ++i) {
        search(child0+i);
    }
}

int main( int argc, char * argv[] ) {
    init( &argc, &argv );
    run( []{
        GlobalAddress<Vertex> root = create_big_global_tree();
        search(root);
    });
    finalize();
    return 0;
}
```
Making graph & vertices into global structures

```cpp
struct Vertex {
    index_t id;
    GlobalAddress<Vertex> children;
    size_t num_children;
};

void search(GlobalAddress<Vertex> vertex_addr) {
    Vertex v = delegate::read(vertex_addr);

    GlobalAddress<Vertex> child0 = v.children;
    for (int i = 0; i < v.num_children; ++i) {
        search(child0+i);
    }
}

int main(int argc, char * argv[]) {
    init(&argc, &argv);
    run([]{
        GlobalAddress<Vertex> root = create_big_global_tree();
        search(root);
    });
    finalize();
    return 0;
}
```
Make the loop over neighbors parallel

```c
struct Vertex {
    index_t id;
    GlobalAddress<Vertex> children;
    size_t num_children;
};

void search(GlobalAddress<Vertex> vertex_addr) {
    Vertex v = delegate::read(vertex_addr);
    GlobalAddress<Vertex> child0 = v.children;
    forall( 0, v.num_children, [child0](int64_t i) {
        search(child0+i);
    })
}

int main( int argc, char * argv[] ) {
    init( &argc, &argv );
    run( []{
        GlobalAddress<Vertex> root = create_big_global_tree();
        search(root);
    });
    finalize();
    return 0;
}
```
That’s it! Grappa code for a cluster!

```cpp
struct Vertex {
    index_t id;
    GlobalAddress<Vertex> children;
    size_t num_children;
};

void search(GlobalAddress<Vertex> vertex_addr) {
    Vertex v = delegate::read(vertex_addr);

    GlobalAddress<Vertex> child0 = v.children;
    forall( 0, v.num_children, [child0](int64_t i) {
        search(child0+i);
    }
}

int main( int argc, char * argv[] ) {
    init( &argc, &argv );
    run( []{
        GlobalAddress<Vertex> root = create_big_global_tree();
        search(root);
    });
    finalize();
    return 0;
}
```
Outline

• Motivation

• Programming Grappa

• Key components

• Performance

• Conclusion
Grappa design

Distributed Shared Memory

Lightweight Multithreading w/ Global Task Pool

Communication Layer

Memory

Cores

Message aggregation layer

Infiniband network, user level access
User level context switching

Main innovation: We keep state small and prefetch to cover DRAM latency

1 cacheline of status, 3 cachelines of stack

Task queue

Ready queue

Suspended workers

Task n

Task queue

Core

Task n+1

Stack

Task 2

Worker 2

status

Ready queue

Suspended workers

Worker 1

status

Stack

Task 1

L1 Cache

Worker 2

status

Stack

Task 2

Worker 1

status

Stack

Task 1

Task 1

1 cacheline of status, 3 cachelines of stack

Main innovation: We keep state small and prefetch to cover DRAM latency
Accessing memory through delegates

Each word of memory has a designated *home core*
All accesses to that word run on that core
Requestor blocks until complete
Accessing memory through delegates

Since var is private to home core, updates can be applied without expensive synchronization
Mitigating low injection rate with aggregation

![Graph showing bandwidth (GB) against message size (16 B, 1 kB, 64 kB)]
Mitigating low injection rate with aggregation

Node 0

Node $n$
Mitigating low injection rate with aggregation

Node 0

Worker 1
Stack
Msg 1
Stack
Msg 2

Node n

Graph showing the relationship between message size and bandwidth (GB). The x-axis represents message size in bytes and the y-axis represents bandwidth in GB. The graph shows a curve that starts low and rises sharply, indicating a significant increase in bandwidth with larger message sizes.

Legend:
- Node 0
- Node n
- Worker 1
- Worker 2
- Stack
- Msg 1
- Msg 2

Message size values: 16 B, 1 kB, 64 kB
Mitigating low injection rate with aggregation

![Diagram showing message injection rates and bandwidths](image)

Node 0

Node n
Mitigating low injection rate with aggregation

Worker 1
   Stack
   Msg 1

Worker 2
   Stack
   Msg 2

Worker 3
   Stack
   Msg 3

Wire
Msg

Node 0

Node n

Graph showing Bandwidth (GB) vs. Message size (in bytes). The x-axis represents message size in 16 B, 1 kB, and 64 kB sections, while the y-axis represents Bandwidth (in GB) ranging from 0 to 2.
Mitigating low injection rate with aggregation

The diagram shows a network setup with Worker 1, Worker 2, and Worker 3 connected to Node 0, and Node n. The Stack layers are indicated for each Worker. There is a message flow diagram with messages labeled Msg 1, Msg 2, and Msg 3. A graph illustrates the Bandwidth (GB) against Message size, showing a curve that peaks at around 64 kB.
Mitigating low injection rate with aggregation

Node 0

Worker 1
Stack

Worker 3
Stack

Worker 2
Stack

Node n

Wire
Msg

Msg 1

Msg 2

Msg 3

Graph:
- X-axis: Message size (16 B, 1 kB, 64 kB)
- Y-axis: Bandwidth (GB)
- Curve showing increasing bandwidth with message size.
Mitigating low injection rate with aggregation

[Diagram showing network components and a graph illustrating bandwidth vs. message size]

- Worker 1
  - Stack
- Worker 2
  - Stack
- Worker 3
  - Stack

Node 0

Node n

- Wire Msg
- Msg 1
- Msg 2
- Msg 3

Graph:
- X-axis: Message size (16 B, 1 kB, 64 kB)
- Y-axis: Bandwidth (GB)
Mitigating low injection rate with aggregation
Delegation + aggregation makes random access fast.

GUPS benchmark increments random elements of distributed array.
Outline

• Motivation

• Programming Grappa

• Key components

• Performance

• Conclusion
Exploring Grappa’s performance

• Current implementation: 17K lines
  Runs on x86 Linux clusters with MPI and fast networks (InfiniBand)

• We built three frameworks:
  A subset of the GraphLab API
  A relational query execution engine
  A simple in-memory Map/Reduce engine

• Ran on AMD Interlagos cluster at Pacific Northwest National Laboratory
  128 nodes
  Each node: 32 2.1GHz cores, 64GB, 40Gb InfiniBand network
GraphLab on Grappa

• Subset of the GraphLab API described in PowerGraph paper:
  • Synchronous engine
  • Delta caching optimization

• GraphLab: replicated graph representation and complex partitioning strategy; Grappa: simple adjacency list and random partitioning

• 60 lines of code!

• Four benchmarks: PageRank, connected components, single-source shortest path, breadth-first search

• Graphs: Friendster (65M vertices, 1.8B edges), Twitter (41M vertices, 1B edges)
Grappa/GraphLab application performance

We used Grappa to build a distributed backend to Raco, a relational algebra compiler and optimization framework. Raco supports a variety of relational query language representations.

4.2 Scaling

deteriorates less than 30% at 128 nodes.

Figure 9:

(a) Application performance (31 nodes)

(b) Communication metrics at 31 nodes on PageRank.

(c) Scaling PageRank: strong scaling on Twitter and weak scaling on synthetic graphs.

Average: 1.33x

(31 nodes)
A closer look at PageRank

![Graph showing PageRank execution over time on 32 nodes](image)

- **Concurrent tasks (millions)**: The graph illustrates the variation in the number of concurrent tasks over time. Peaks are observed, indicating bursts of activity, which then reduce as the execution progresses. The y-axis measures the number of concurrent tasks in millions.

- **Bandwidth (GB/s/node)**: This graph shows the bandwidth utilization over time. Peaks are evident, dropping as activity diminishes. The y-axis measures bandwidth in GB/s per node.

- **Time (s)**: Both graphs' x-axis represents time in seconds, ranging from 0 to 25 seconds.

- **Nodes**: The experiment was conducted on 31 nodes.

**Figure 10:** Grappa PageRank execution over time on 32 nodes.

**Notes:**
- A closer look at PageRank execution over time, showing peaks and drops in concurrent tasks and bandwidth.
- The graphs demonstrate the efficiency of the system in managing parallel tasks and data transfer.

---

**Table 1:** Communication and Performance Metrics

<table>
<thead>
<tr>
<th>Application</th>
<th>Total Data Moved (GB)</th>
<th>Mean Aggregated Packet Size (kB)</th>
<th>Peak for Aggregated Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pagerank</td>
<td>100</td>
<td>20</td>
<td>Peak bandwidth</td>
</tr>
<tr>
<td>Twitter</td>
<td>200</td>
<td>40</td>
<td>Peak aggregated size</td>
</tr>
<tr>
<td>Friendster</td>
<td>300</td>
<td>80</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 9:**

- **Figure 9a:** Application performance (31 nodes) for relational queries on Grappa.
- **Figure 9b:** Communication metrics at 31 nodes on PageRank.
- **Figure 9c:** Strong scaling results on both Twitter and Friendster, weak scaling on synthetic graphs.

---

**Performance Characterization of Grappa's GraphLab API:**

- Grappa achieves competitive performance with parallel databases. Raco is source-to-source translation, generating competitive query plans. Shark is optimized for in-memory execution, outperforming Grappa in terms of speed and execution times for the given message size.

**Conclusion:**

- Grappa is a scalable and efficient backend for distributed graph processing, suitable for applications requiring high concurrency and data transfer rates.
Relational query execution

• Built a backend for the Raco relational algebra compiler/optimizer
  • Queries are compiled into Grappa for loops
  • ~700 lines

• Compare with Shark, a Hive/SQL-like query system built on Spark using SP2Bench benchmark
We focus on workloads that can be processed in memory. While in terms of programming model this a departure from the shared-nothing architecture of nearly all parallel query processing engines, Grappa can leverage the locality-oriented execution model of shared memory code. The generated code is sent through a normal C++11 compiler.

A portion of the performance difference is attributable to Grappa's pipelined query plans. Currently, our implementation of MapReduce is not fault-tolerant. To ensure the comparison is fair, we verify that no bytes were written to the filesystem by the worker. The mappers materialize results into a local hash table, using a mechanism for materializing output (appending to a C++ STL vector) is slower. This accounts for the 2x relative decrease in performance on Q3a and Q9. Although we see a single join on Shark comes much closer to the performance of Grappa, the large gap in the performance of select with 99% selectivity falls to 4.23x, so Grappa's performance of Grappa allows iterations to be implemented by the application programmer with a relatively simple extension of the generator for serial code in [49] to generating parallel code in [56].

4.3.1 Iterative MapReduce on Grappa

In the all-grouped join queries (Q1, Q3b, and Q3c) we can attribute up to 1 order of magnitude of Grappa's speedup to difference in selectivity, Grappa is 12.8x faster (Figure 11b). Thus, a single relational operator. For a select query of 1% selectivity, Grappa is 12.8x faster (Figure 11b). Thus, a single relational operator. For a select query of 1% selectivity, Grappa is 12.8x faster (Figure 11b).

We experiment with data parallel workloads by implementing an in-memory MapReduce API in 152 lines of Grappa code. The implementation involves a mechanism for materializing output (appending to a C++ vector) is slower. This accounts for the 2x relative decrease in performance on Q3a and Q9. Although we see a single join on Shark comes much closer to the performance of Grappa, the large gap in the performance of select with 99% selectivity falls to 4.23x, so Grappa's performance of Grappa allows iterations to be implemented by the application programmer with a relatively simple extension of the generator for serial code in [49] to generating parallel code in [56].

Relational query execution

Average: 14x
In-memory Map/Reduce

• Simple implementation of Map/Reduce model for iterative applications (no fault-tolerance)

• 152 lines

• Compared with Spark, configured to avoid fault-tolerance

• Benchmark: K-Means on SeaFlow dataset (8.9GB)
The high-performance computing community has largely disenchanted the coherent distributed shared memory approach in favor of the Partitioned Global Address Space (PGAS) model. Examples include Split-C, Chapel, and Blizzard allowed the tracking of ownership with mechanisms and network wire bandwidth when locality is abundant, but otherwise result in increased false sharing. There are many other data-parallel frameworks like GraphLab-inspired API is built into the runtime is ongoing. In addition work is at appropriate points. Research into generic mechanisms 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 instances is that remote data accesses are explicit, thereby encouraging developers to use them judiciously. Grappa followed the lead of TreadMarks and provides DSM entirely at user-level through a library and runtime. FaRM systems exploited both of these ideas, but still incurred some coherence overhead. Paging strategies have been 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 result in increased false sharing by adopting multiple instances. There is no built-in fault-tolerance mechanism. The fundamental challenge with fault-tolerance with Grappa is the value storage built from RPCs. The incremental data-parallel system Naiad achieves both high-throughput networking for critical tasks similar to Naiad. There have also been recent efforts to build parameterized abstractions. Future work may expand Grappa to support building block on its own for applications or higher-level applications or higher-level applications. Grappa is a useful programming model for performance reasons. How-ever, a high-throughput DSM like Grappa is a useful programming model for performance reasons. How-ever, a high-throughput DSM like Grappa is a useful programming model for performance reasons. How-ever, a high-throughput DSM like Grappa is a useful programming model for performance reasons. How-ever, a high-throughput DSM like Grappa is a useful programming model for performance reasons.

**Figure 12: Data parallel experiments using k-means on a 8.9GB Seaflow dataset.**

(a) performance for 64 nodes

(b) breakdown of time spent in MapReduce portions for Grappa-MapReduce

(c) message rate of hashtable appends

<table>
<thead>
<tr>
<th>Nodes</th>
<th>K</th>
<th>Appendix messages/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>1e+00</td>
</tr>
<tr>
<td>10</td>
<td>10000</td>
<td>2e+07</td>
</tr>
<tr>
<td>10</td>
<td>100000</td>
<td>4e+07</td>
</tr>
<tr>
<td>10</td>
<td>1000000</td>
<td>6e+07</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>k</th>
<th>Grappa (MapReduce)</th>
<th>Spark</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10.0</td>
<td>5.0</td>
</tr>
<tr>
<td>10000</td>
<td>2.5</td>
<td>7.5</td>
</tr>
<tr>
<td>100000</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

**Conclusions & Future work**

While the core Grappa runtime is complete, there is a wealth of ongoing research and development. Presently there is no built-in fault-tolerance mechanism.
Writing against Grappa directly

- Implemented Beamer’s direction-optimizing BFS for low-diameter scale-free graphs

- Not expressible in GraphLab model

- Compared with GraphLab BFS implementation
Writing against Grappa directly

![Graph showing performance comparison between Friendster and Twitter platforms using different versions of Grappa and GraphLab.](image)

**Platform**
- **Grappa**
- **GraphLab**

**Version**
- **GraphLab (pds)**
- **GraphLab (random)**
- **Grappa**
- **Grappa (native)**
Outline

• Motivation

• Programming Grappa

• Key components

• Performance

• Conclusion
Grappa is Open Source Software

http://grappa.io
licensed under AGPLv1
Future steps

• Add to library of data structures

• Expand GraphLab API support

• Support and grow open-source project

• Collaborate with you!
Conclusion

• Grappa is a platform for accelerating in-memory data intensive applications

• **Extreme latency tolerance** helps us build a general, fast platform

• Try it out!
http://grappa.io

Questions?