Grappa: enabling next-generation analytics tools via latency-tolerant distributed shared memory

Jacob Nelson, Brandon Myers, Brandon Holt, Preston Briggs, Luis Ceze, Simon Kahan, Mark Oskin
{nelson, bholt, bdmyers, preston, luisceze, skahan, oskin}@cs.washington.edu
University of Washington

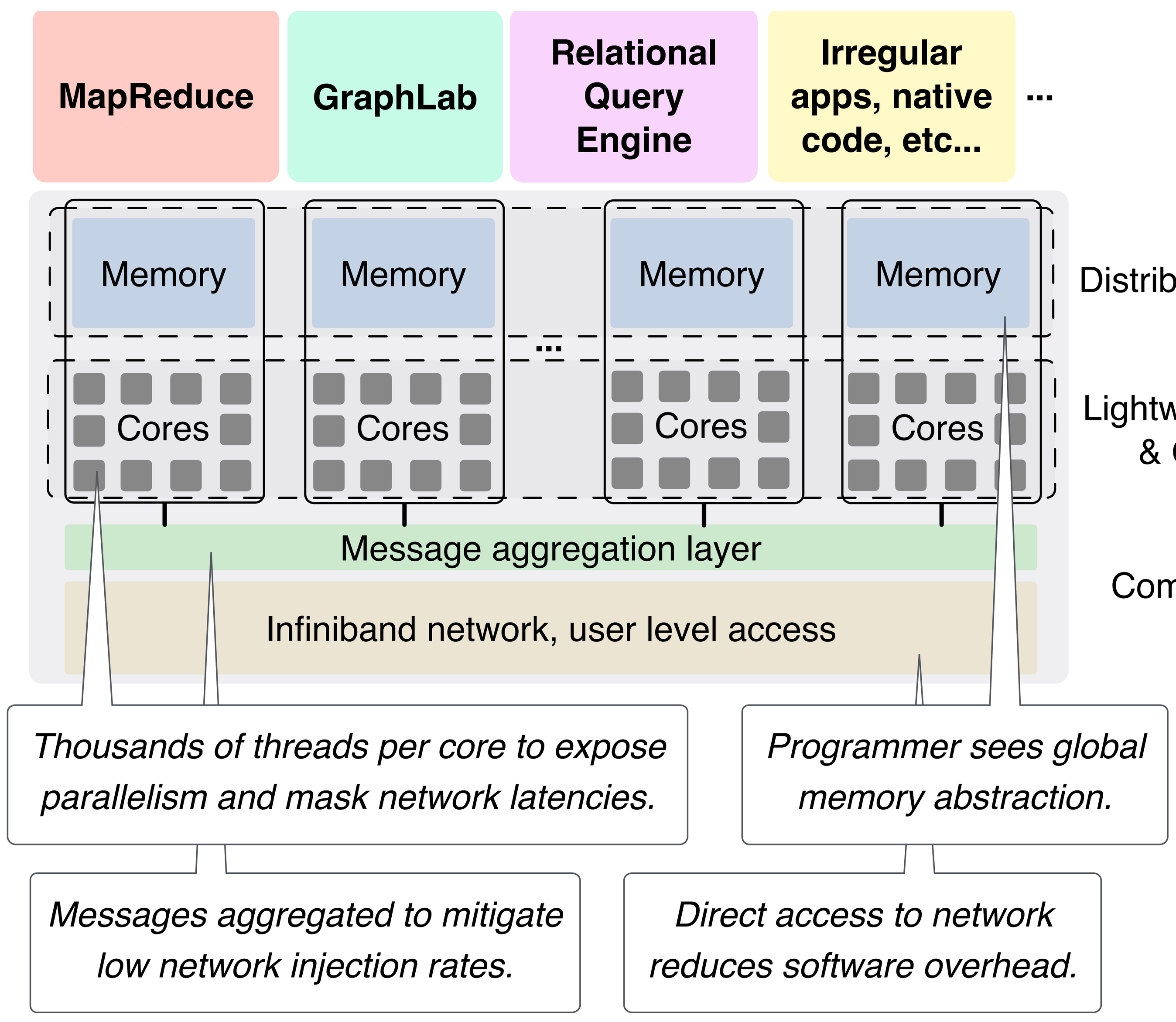
More info, tech report, downloads:
http://grappa.io

Grappa is a modern take on software distributed shared memory, tailored to *exploit parallelism* inherent in data-intensive applications to overcome their poor locality and input-dependent load distribution.

Grappa differs from traditional DSMs in three ways:

- Instead of minimizing per-operation latency for performance, Grappa *tolerates latency* with concurrency (latency-sensitive apps need not apply!)
- Grappa *moves computation to data* instead of caching data at computation
- Grappa operates at *byte granularity* rather than page granularity

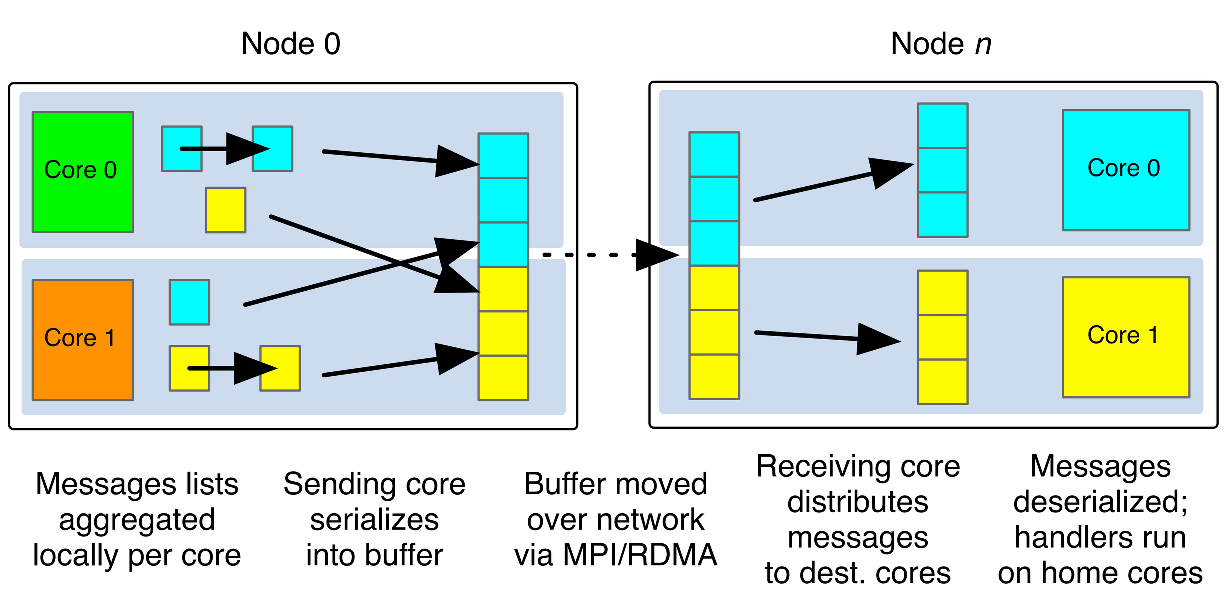
Latency tolerance has been applied successfully in hardware for nanosecond latencies (e.g., superscalar processors and GPUs). This project explores the application of this idea at distributed system scales with millisecond latencies.

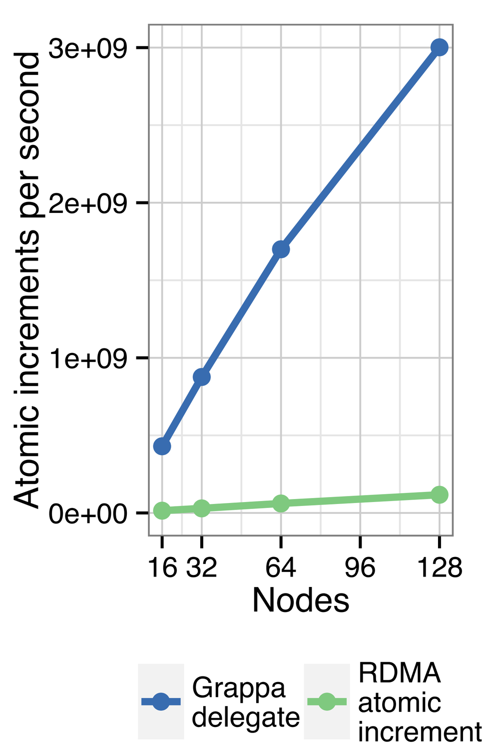


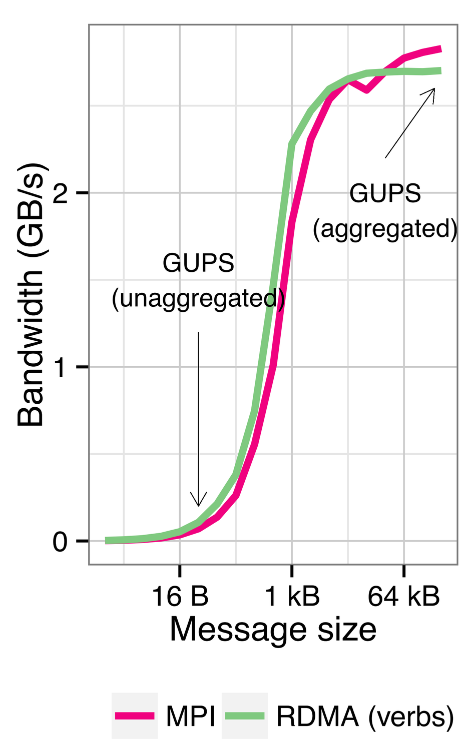
Grappa's current target is small, ~128 node clusters and does not yet implement fault tolerance. Work is ongoing.

Key feature: Message Aggregation

Commodity networks have a limited message injection rate, so building up larger packets from unrelated tasks is essential for small-message throughput (fine-grained random access to global memory).







Programming example

Grappa's familiar single-system multithreaded C++ programming model enables easier development of analysis tools for terabyte-scale data. We provide sequential consistency for race-free programs using RPC-like atomic *delegate operations*, along with standard multithreaded synchronization primitives.

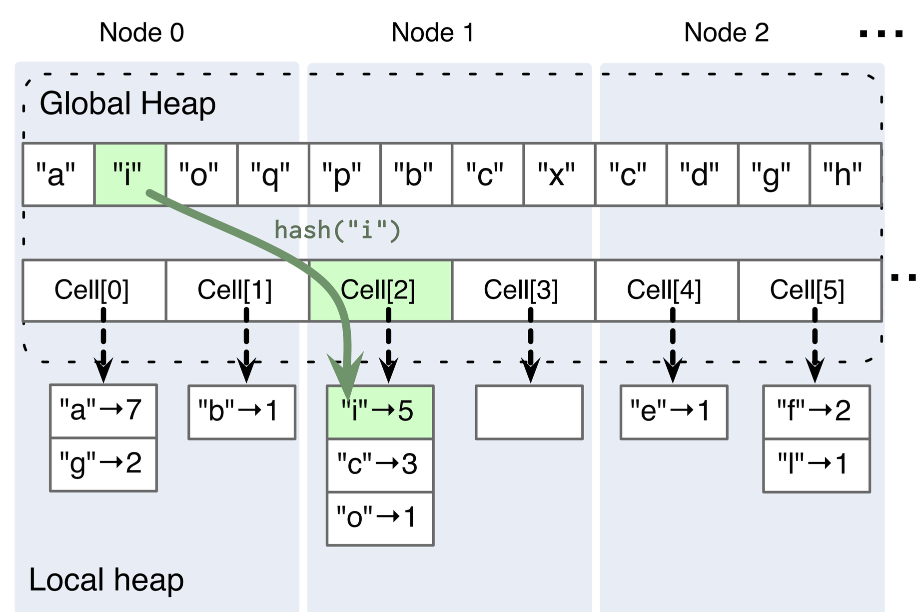
Here is an example of using Grappa's C++11 library to build a distributed parallel word-count-like application with a simple hash table:

```
// distributed input array
GlobalAddress<char> chars;

// distributed hash table:
using Cell = map<char,int>;
GlobalAddress<Cell> cells;

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;
  });
});
```



Visit http://grappa.io for more info on memory allocation, data distribution, and hotspot avoidance via *combining*.

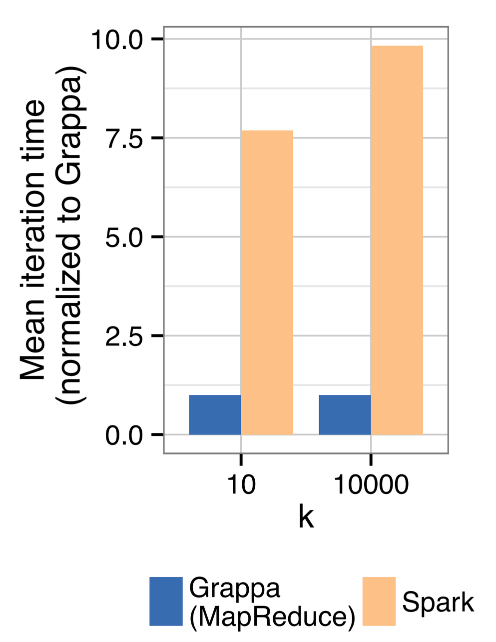
Experiments run at Pacific Northwest National Laboratory on PAL cluster (1.9GHz AMD Interlagos processors, Mellanox ConnectX-2 InfiniBand network)

Three prototype data analytics tools

In-memory MapReduce

~150 lines of code, implemented with forall loop over inputs followed by forall over keys

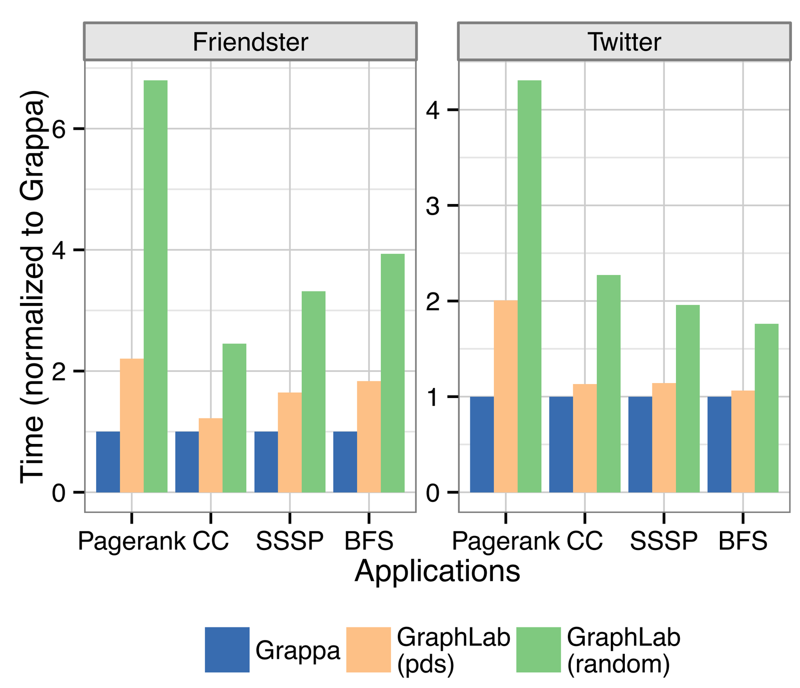
K-Means computation with 64 nodes on SeaFlow flow cytometry dataset with two different k values, compared with Spark using MEMORY_ONLY fault tolerance



GraphLab-like API

~60 lines of code, implementing:
- Synchronous engine with delta caching
- Random graph partition with no replication

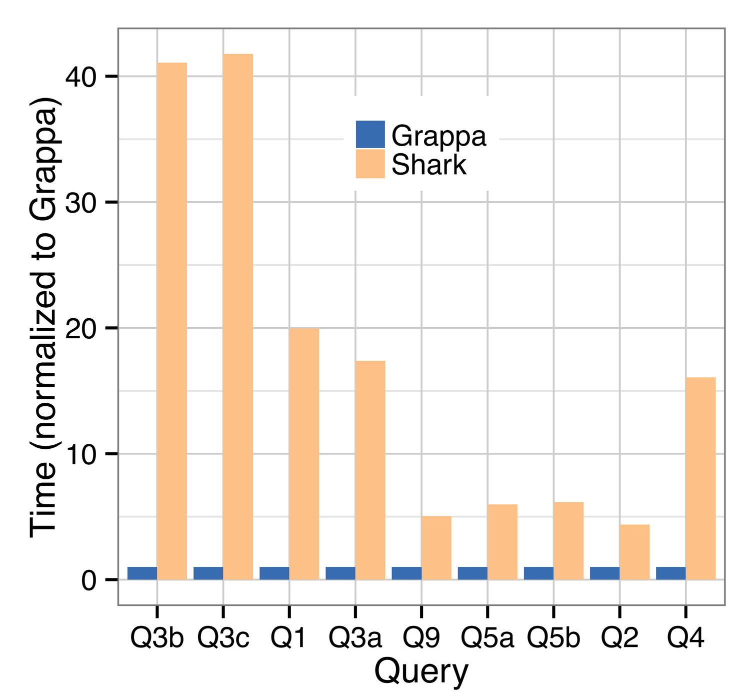
Benchmarks run on 31 nodes using 1.8B edge Friendster social network graph and 1.4B edge Twitter follower dataset, compared with GraphLab using two partitioning strategies



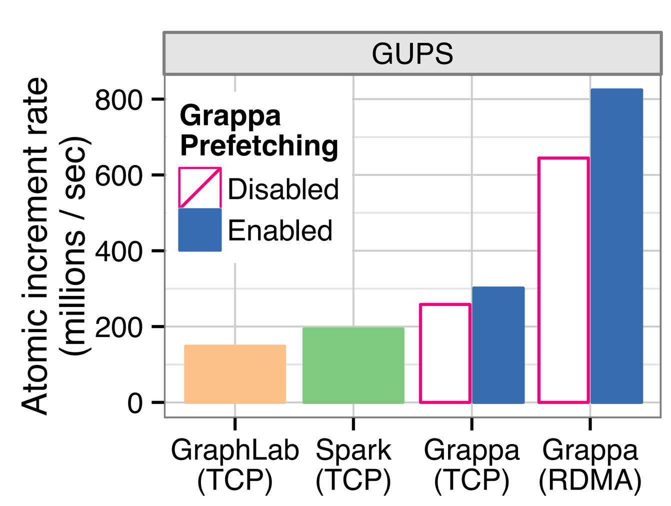
Backend for Raco relational query compiler

~700 lines of code, translating physical query plan into C++11 code using Grappa forall loops

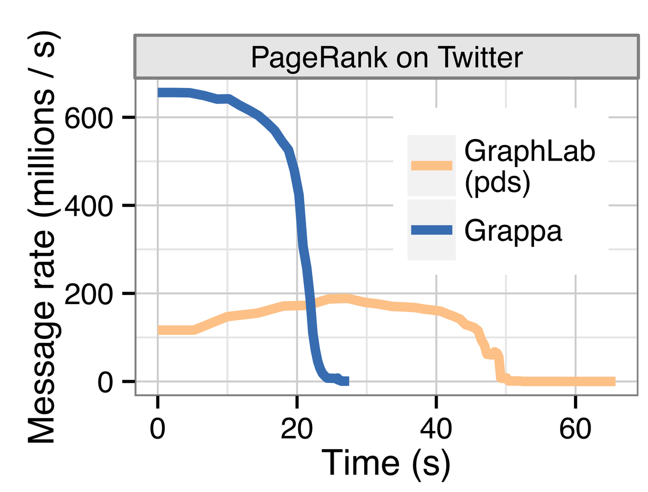
SP²Bench benchmark run on16 nodes, compared with Shark distributed query engine



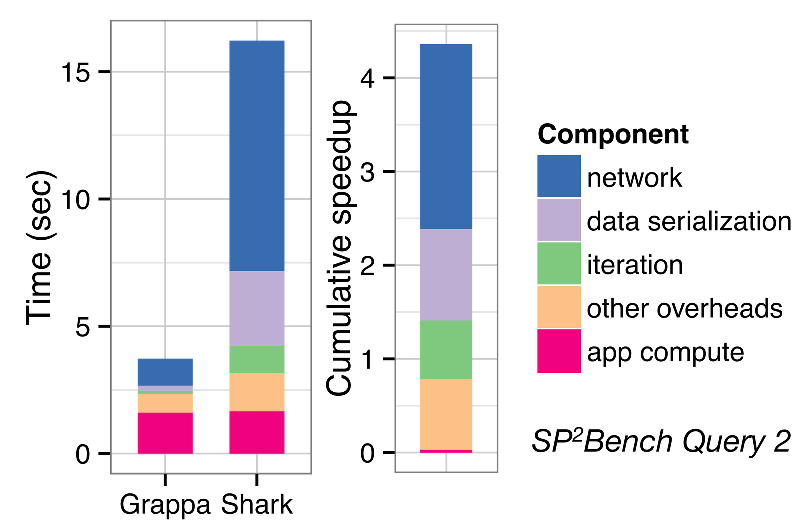
Performance breakdown (or *"why is Grappa faster?"*)



Kernel-bypass communication with cache-aware data placement.



High-concurrency program phases enable aggregation and thus high message rates.



More efficient network layer, lower serialization cost.