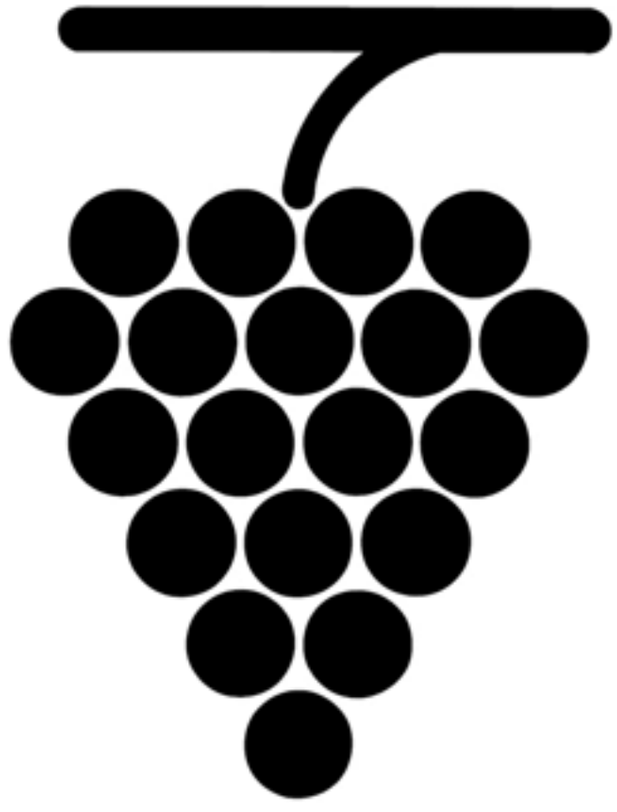


Grappa:

Faster data-intensive applications through latency tolerance

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

University of Washington

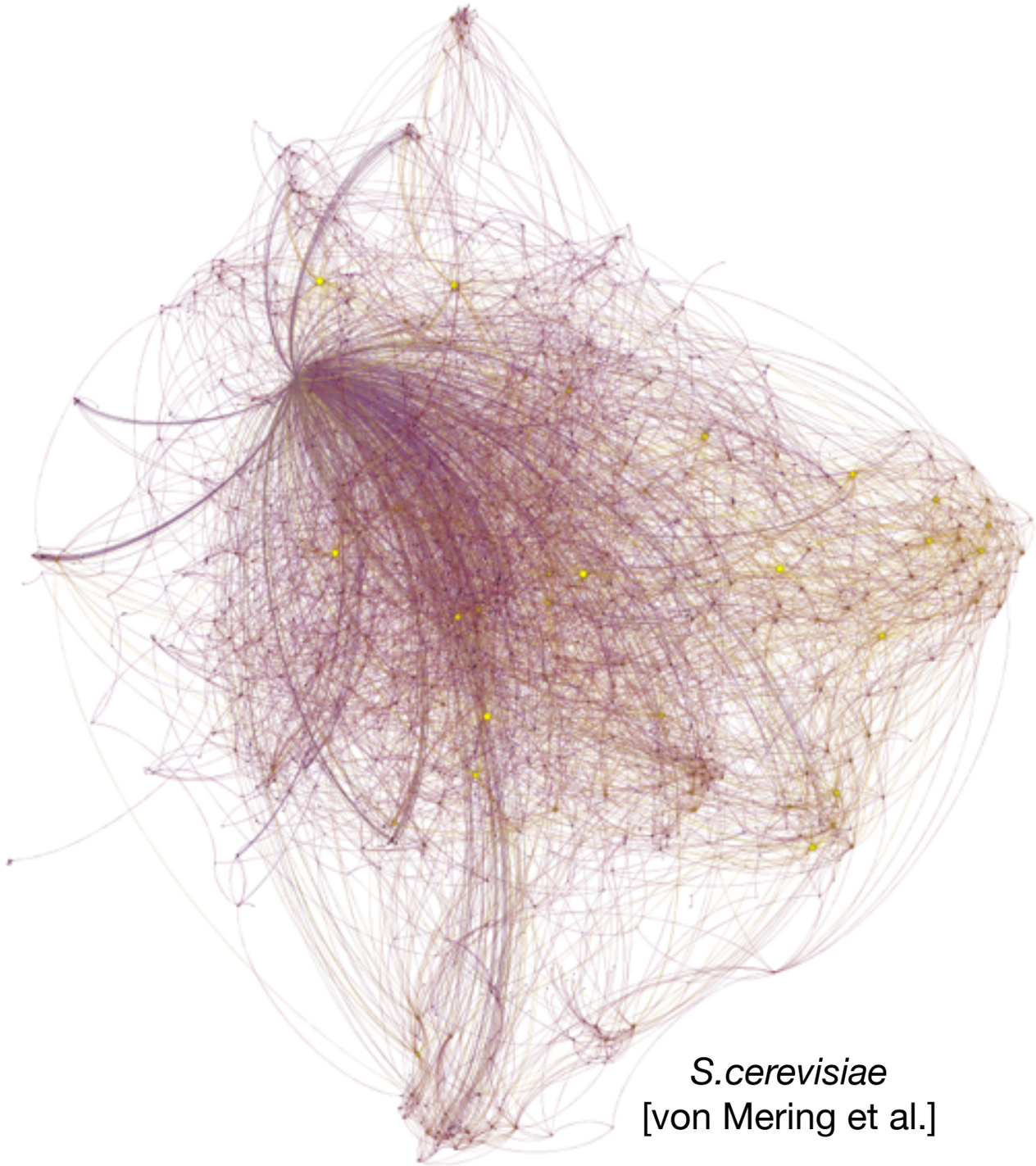
May 7, 2014

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



S.cerevisiae
[von Mering et al.]

Frameworks for data intensive applications

"Pleasantly" parallel problems

Dryad

Graph analytics

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

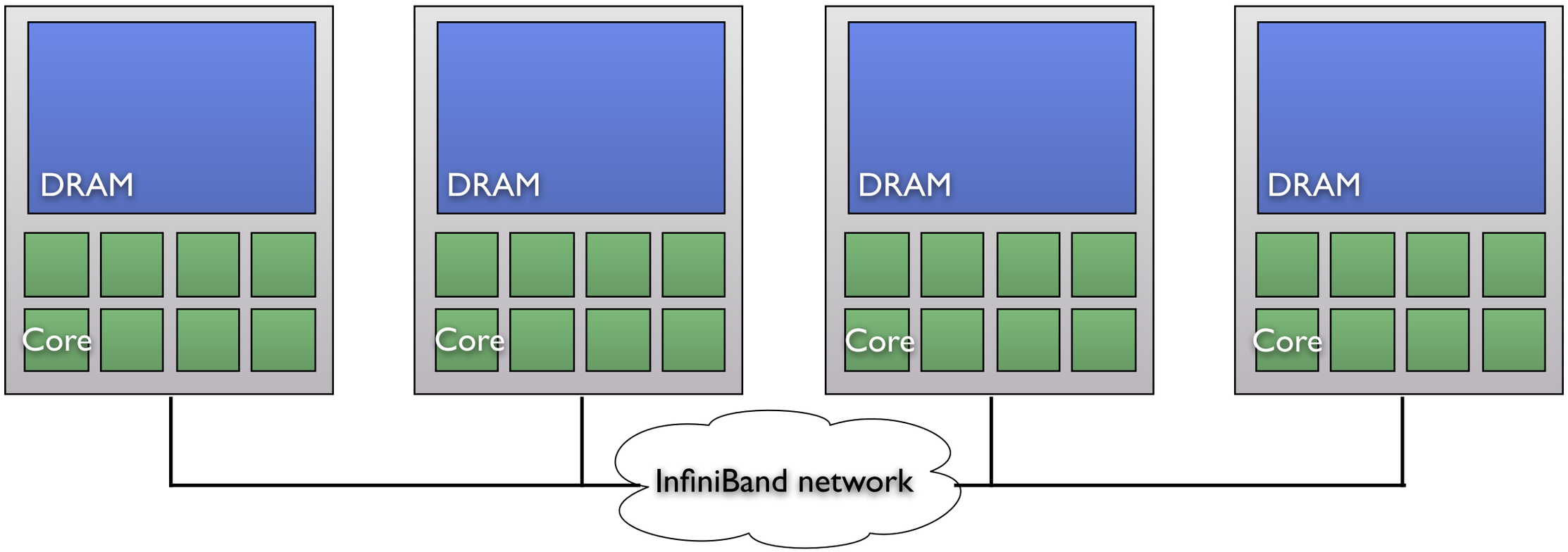
Why is everybody rolling their own?

- Specialized, restricted programming models for each application domain
- Often built from the ground up by application domain experts
- Not a lot of common infrastructure

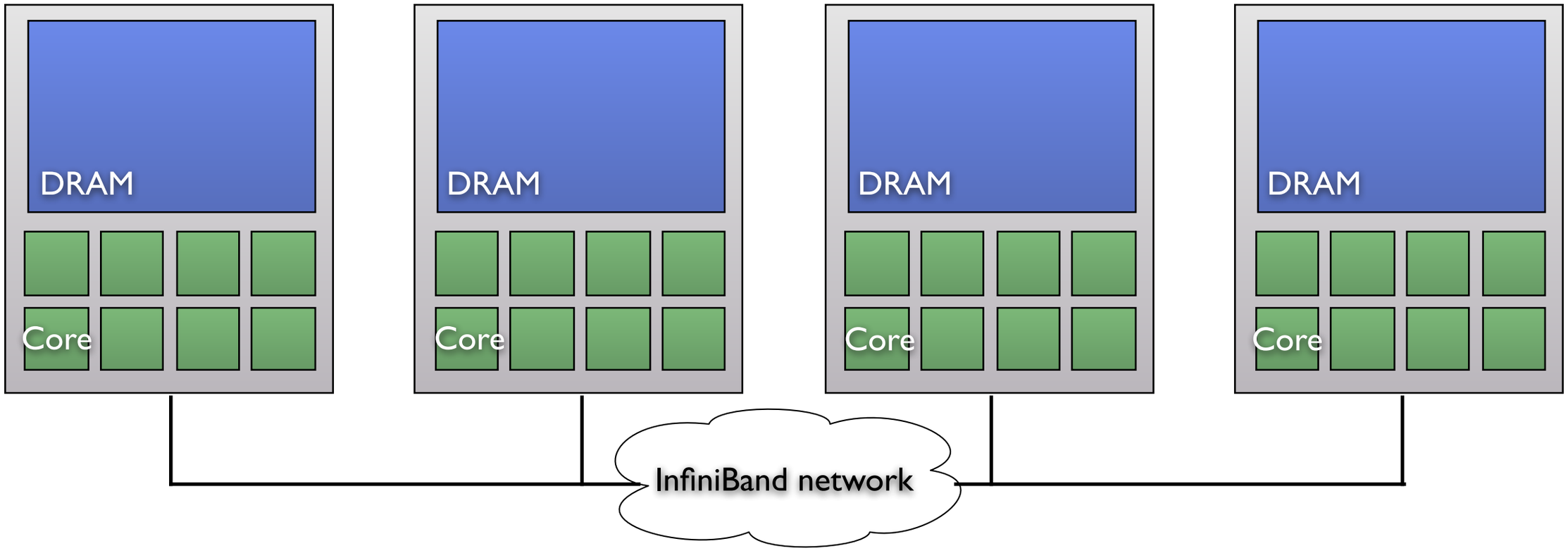
Could these different models
share a common, general platform?

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*

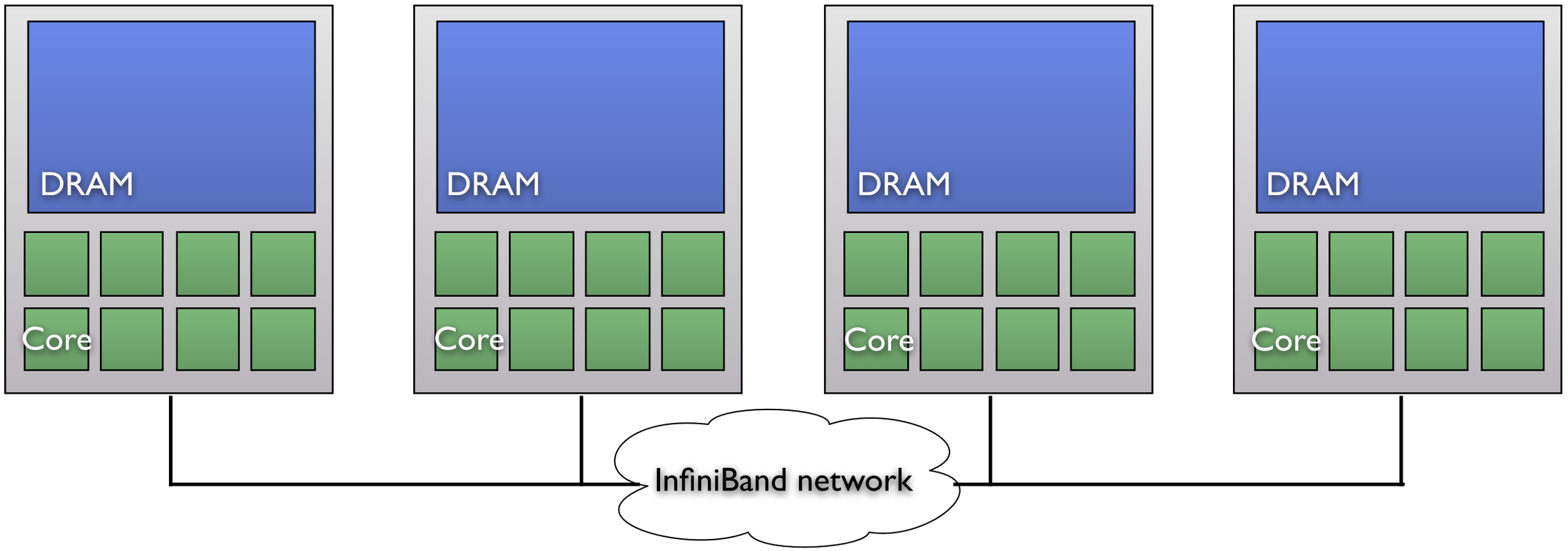

DRAM
DRAM
DRAM
DRAM
Core
Core
Core
Core
InfiniBand network

Linux / PThreads / MPI



Grappa

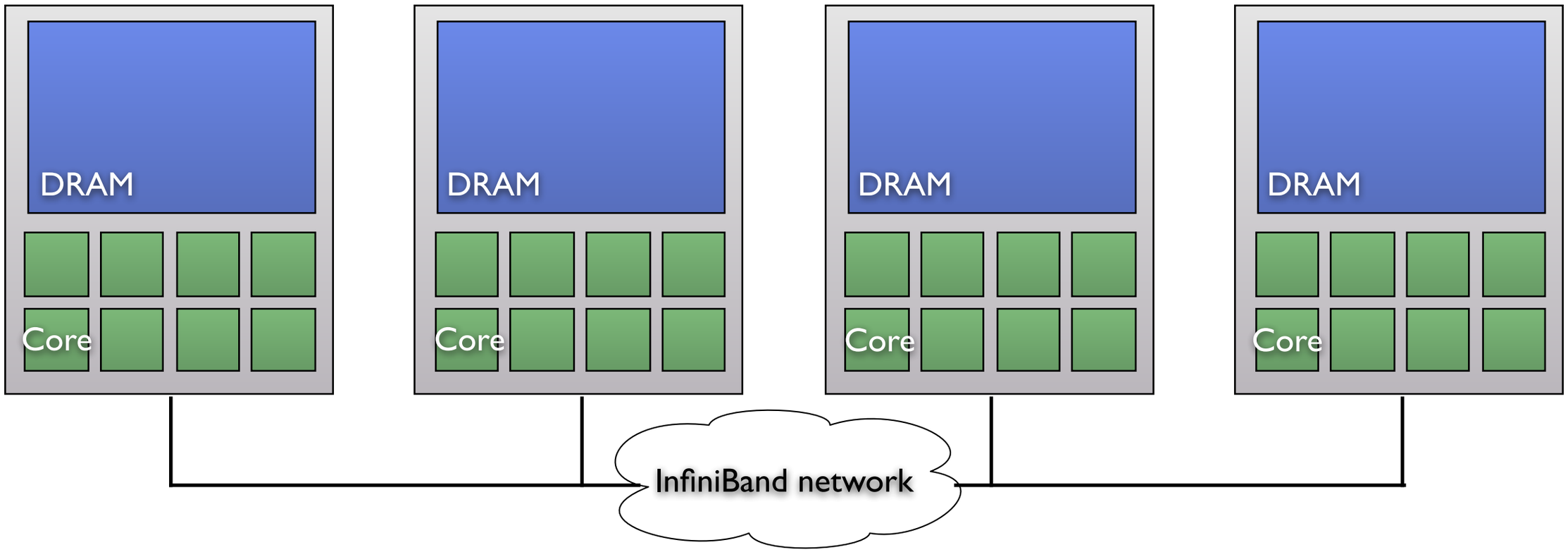
Linux / PThreads / MPI

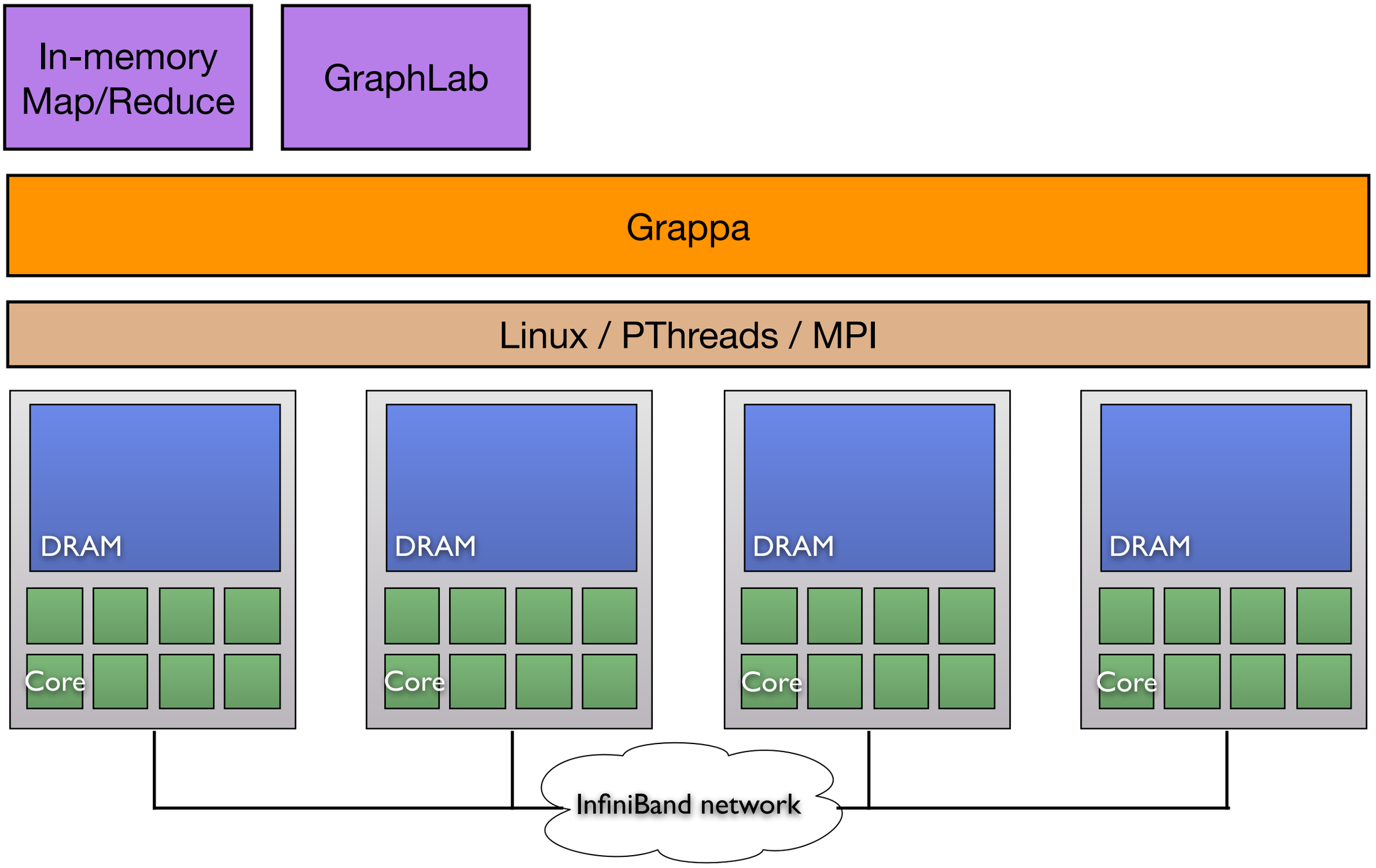


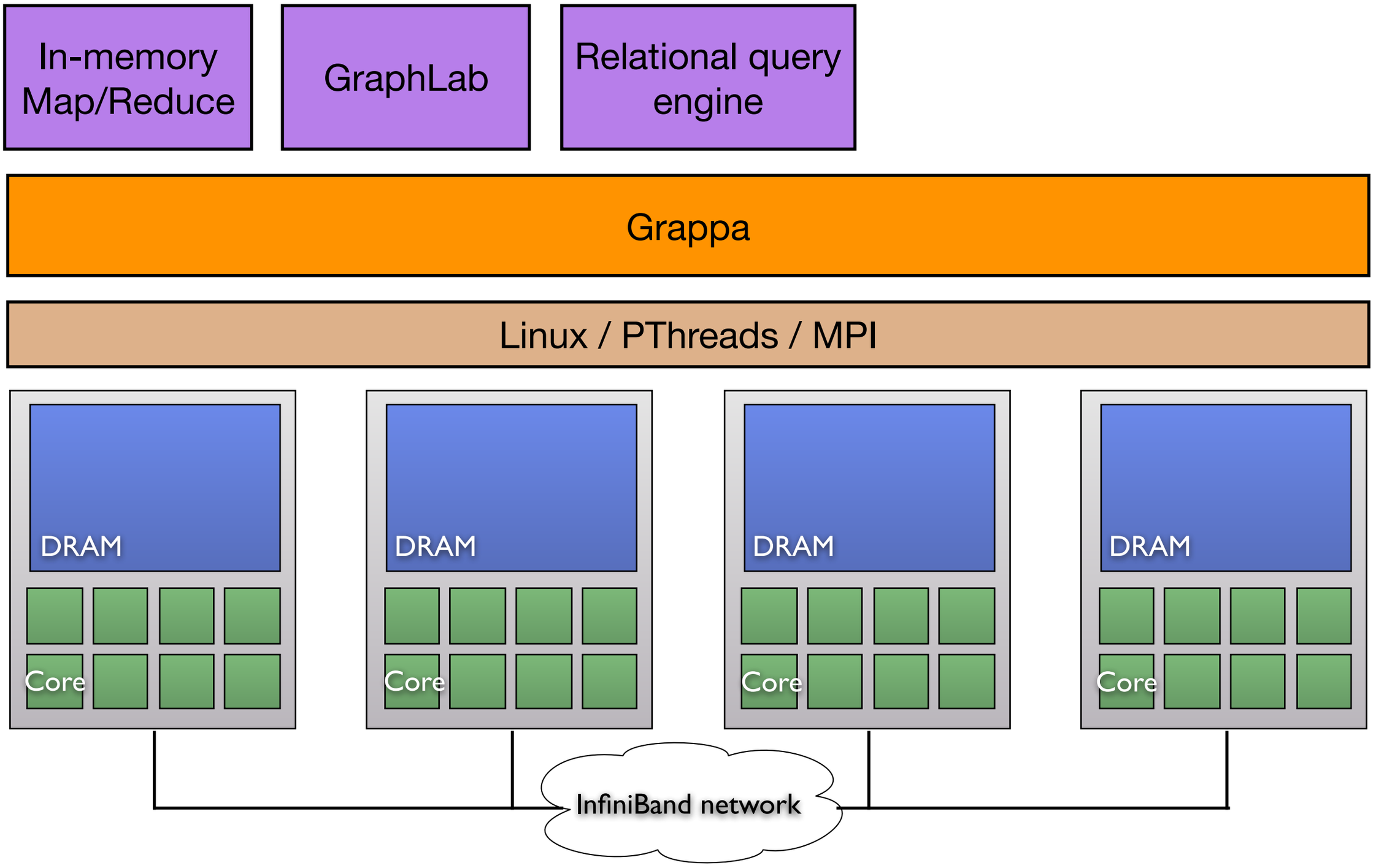
In-memory Map/Reduce

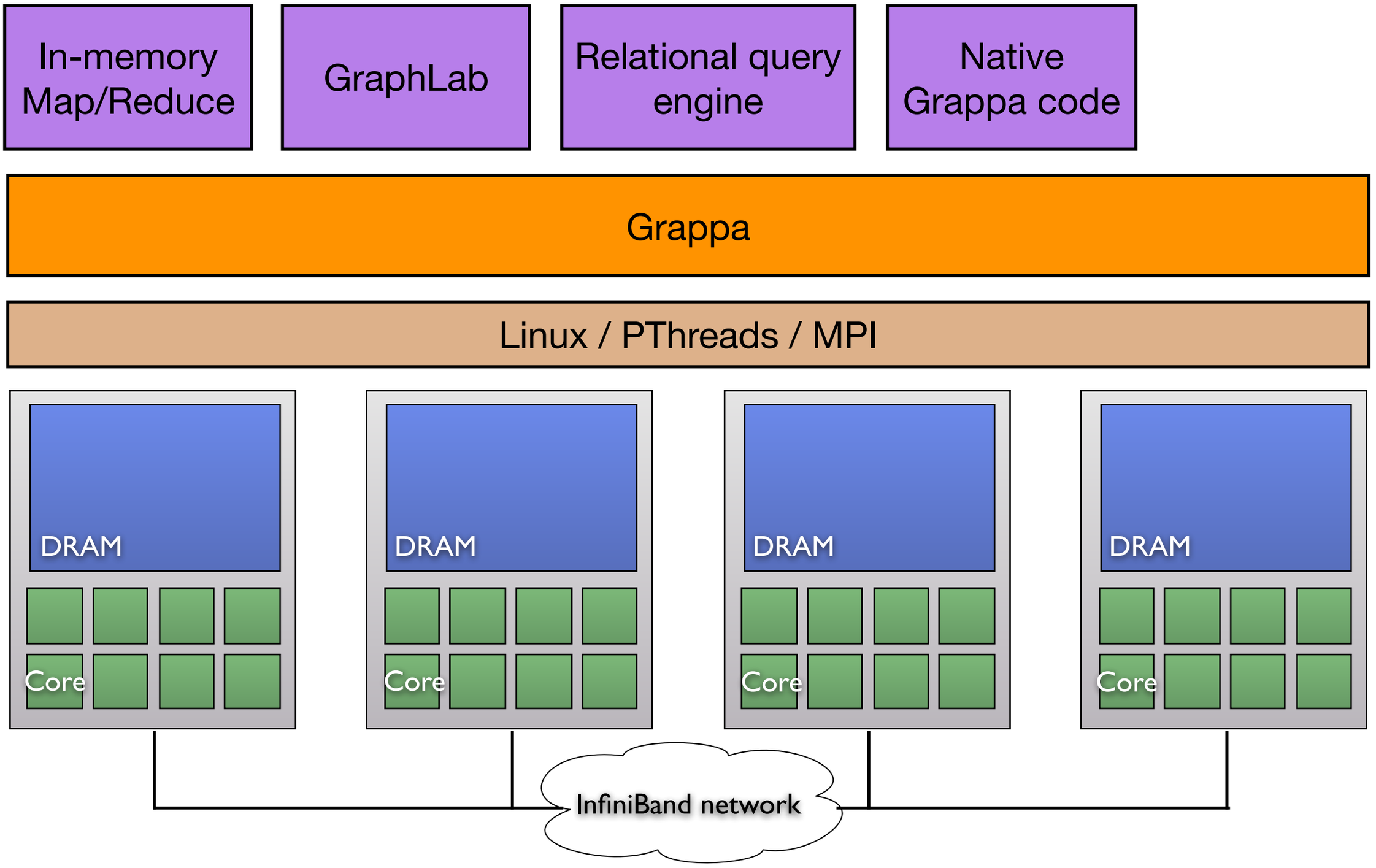
Grappa

Linux / PThreads / MPI




In-memory Map/Reduce
GraphLab
Grappa
Linux / PThreads / MPI
DRAM
DRAM
DRAM
DRAM
Core
Core
Core
Core
InfiniBand network

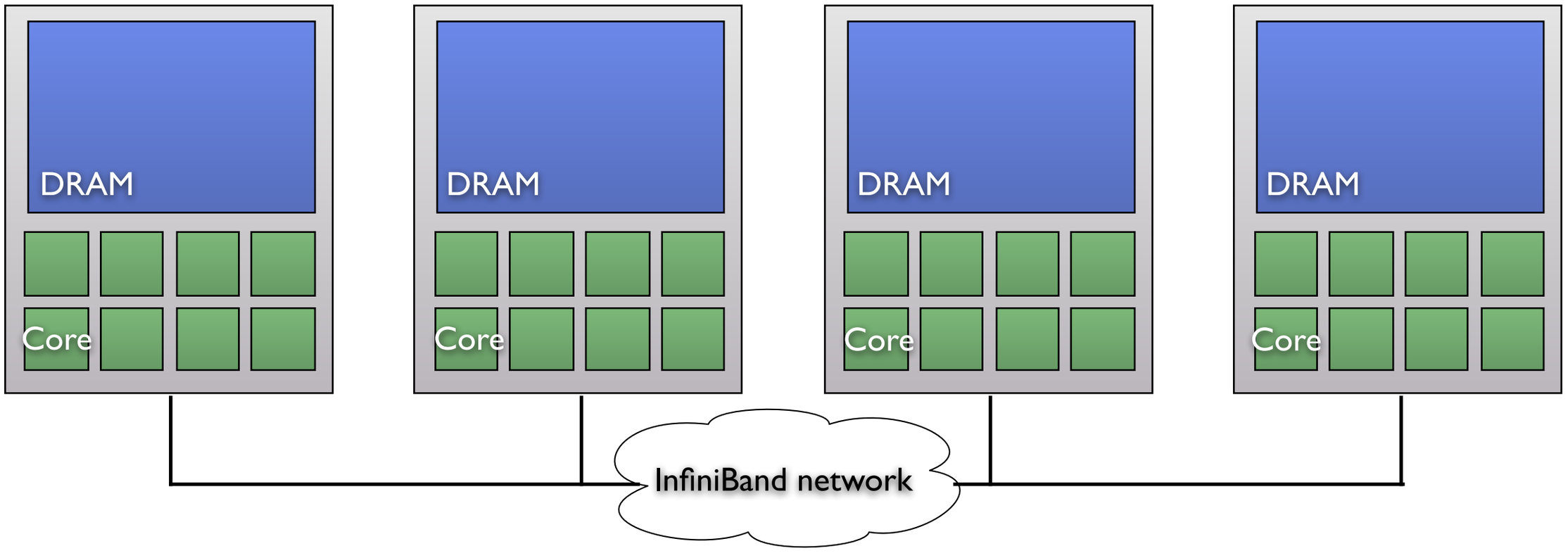

In-memory Map/Reduce
GraphLab
Relational query engine
Grappa
Linux / PThreads / MPI
DRAM
DRAM
DRAM
DRAM
Core
Core
Core
Core
InfiniBand network


In-memory Map/Reduce
GraphLab
Relational query engine
Native Grappa code
Grappa
Linux / PThreads / MPI
DRAM
DRAM
DRAM
DRAM
Core
Core
Core
Core
InfiniBand network

In-memory Map/Reduce | GraphLab | Relational query engine | Native Grappa code | *Your application here!*

Grappa

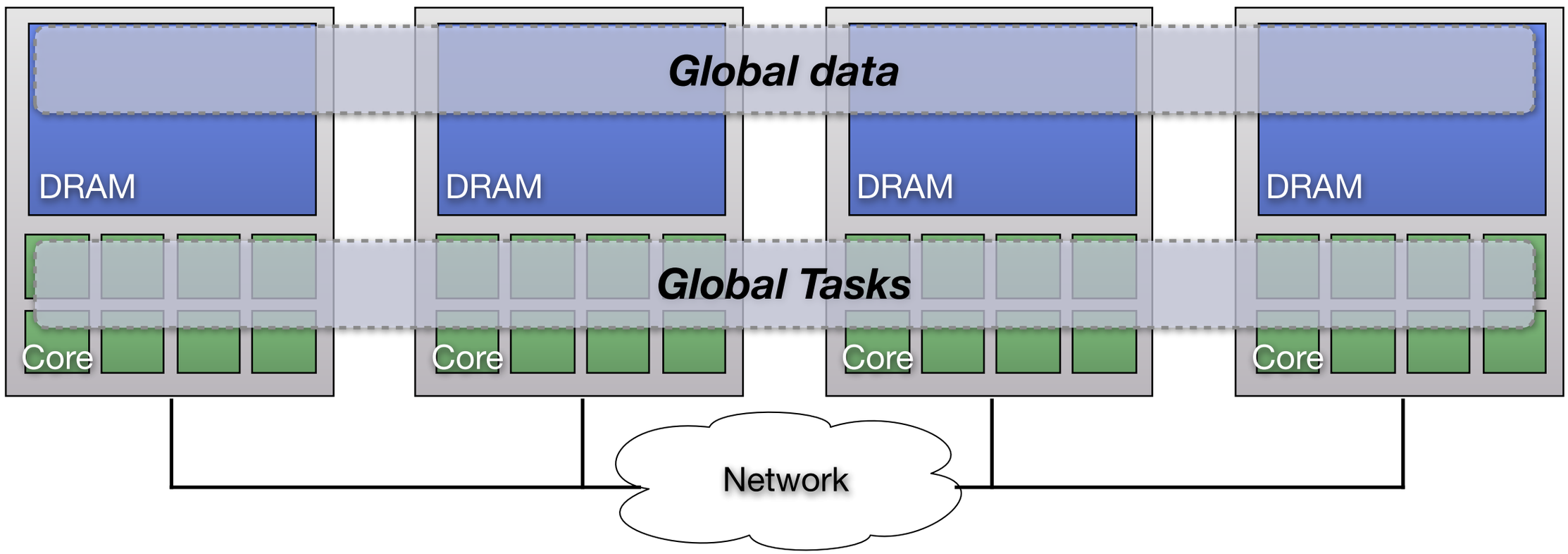
Linux / PThreads / MPI



Outline

- Motivation
- Programming Grappa
- Key components
- Building frameworks on Grappa
- Other projects

Grappa's system view

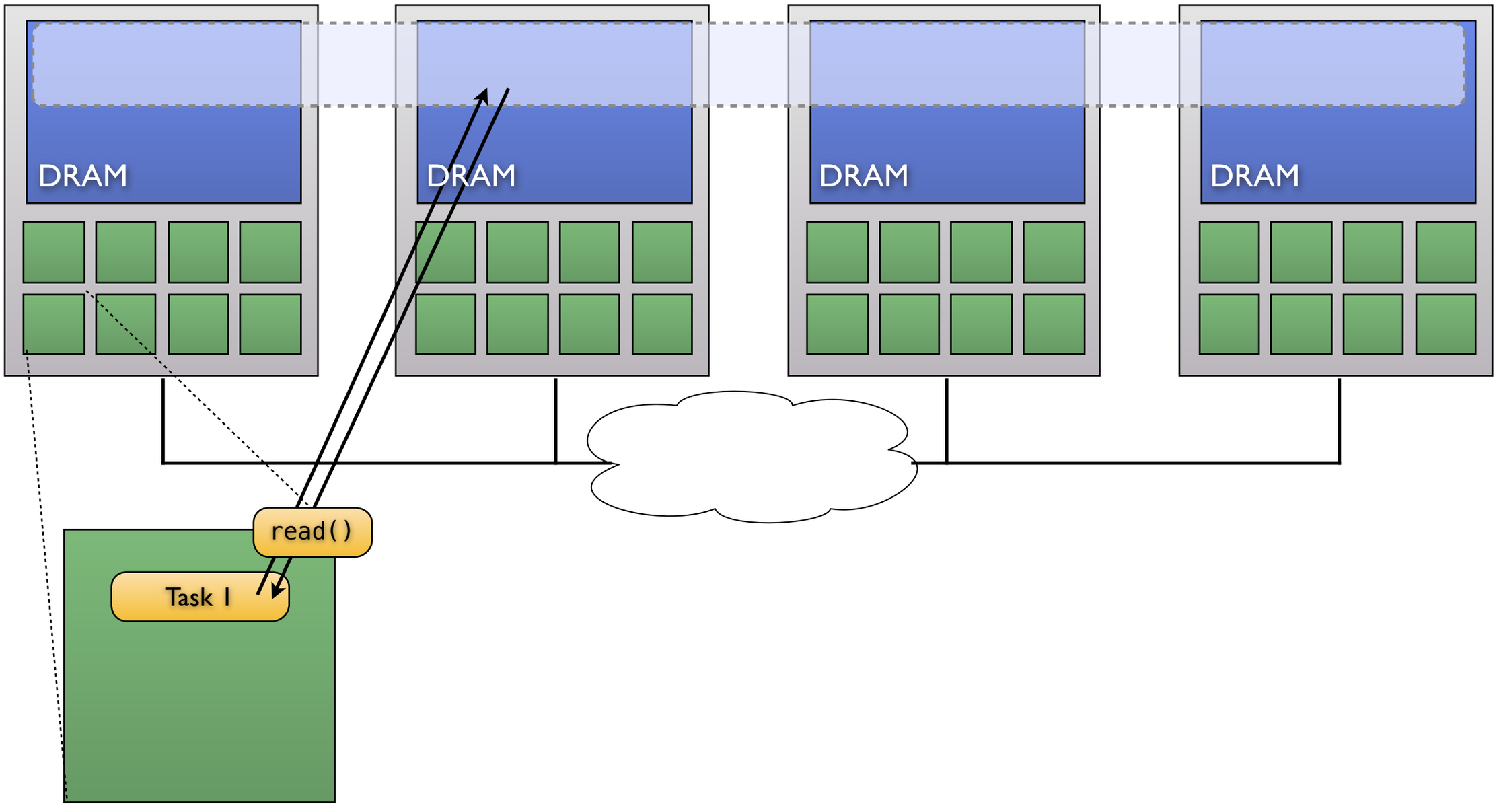


The key observations

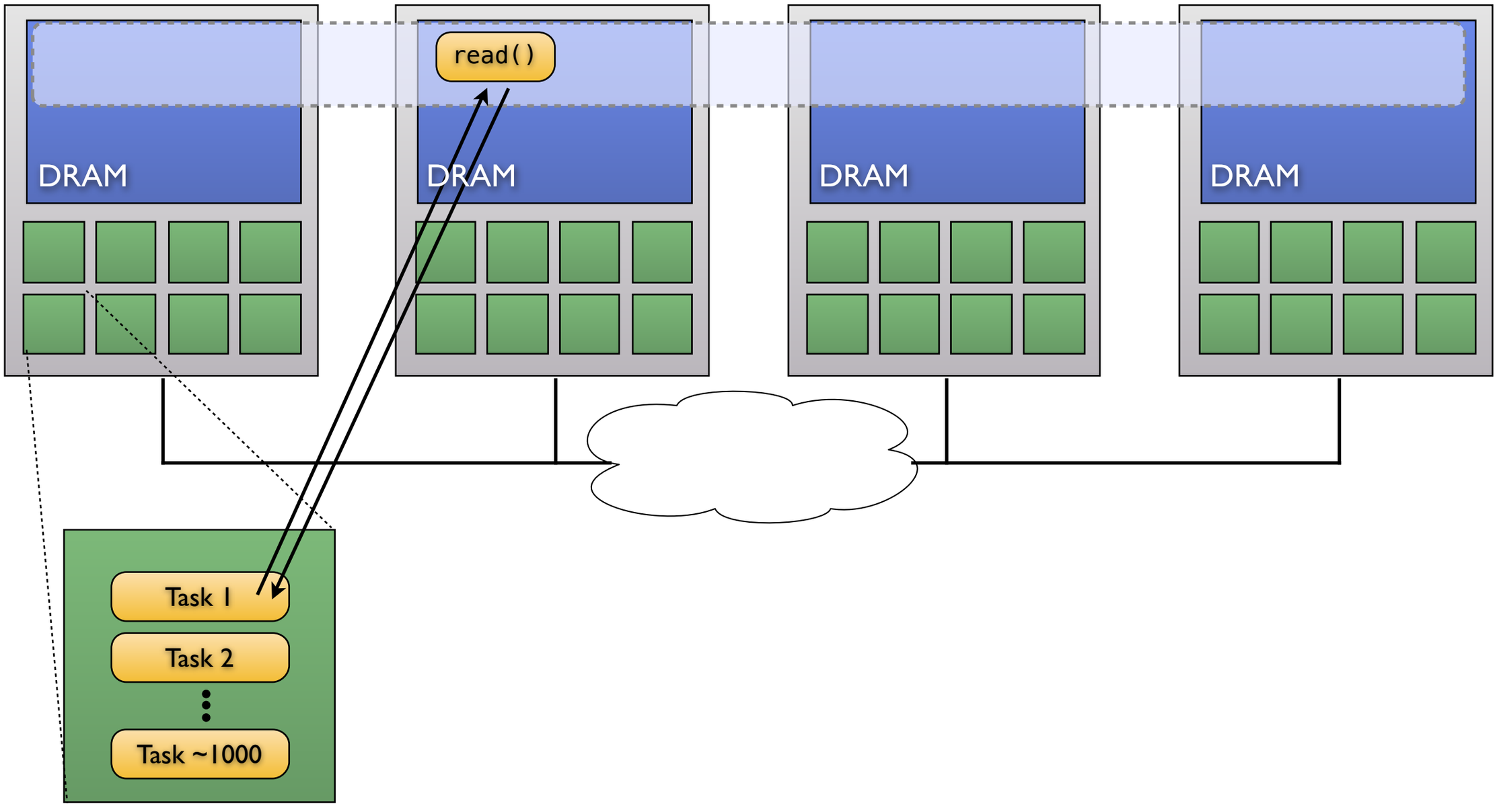
Individual operation latency doesn't matter:
We care only about overall time to solution

These applications have lots of parallelism

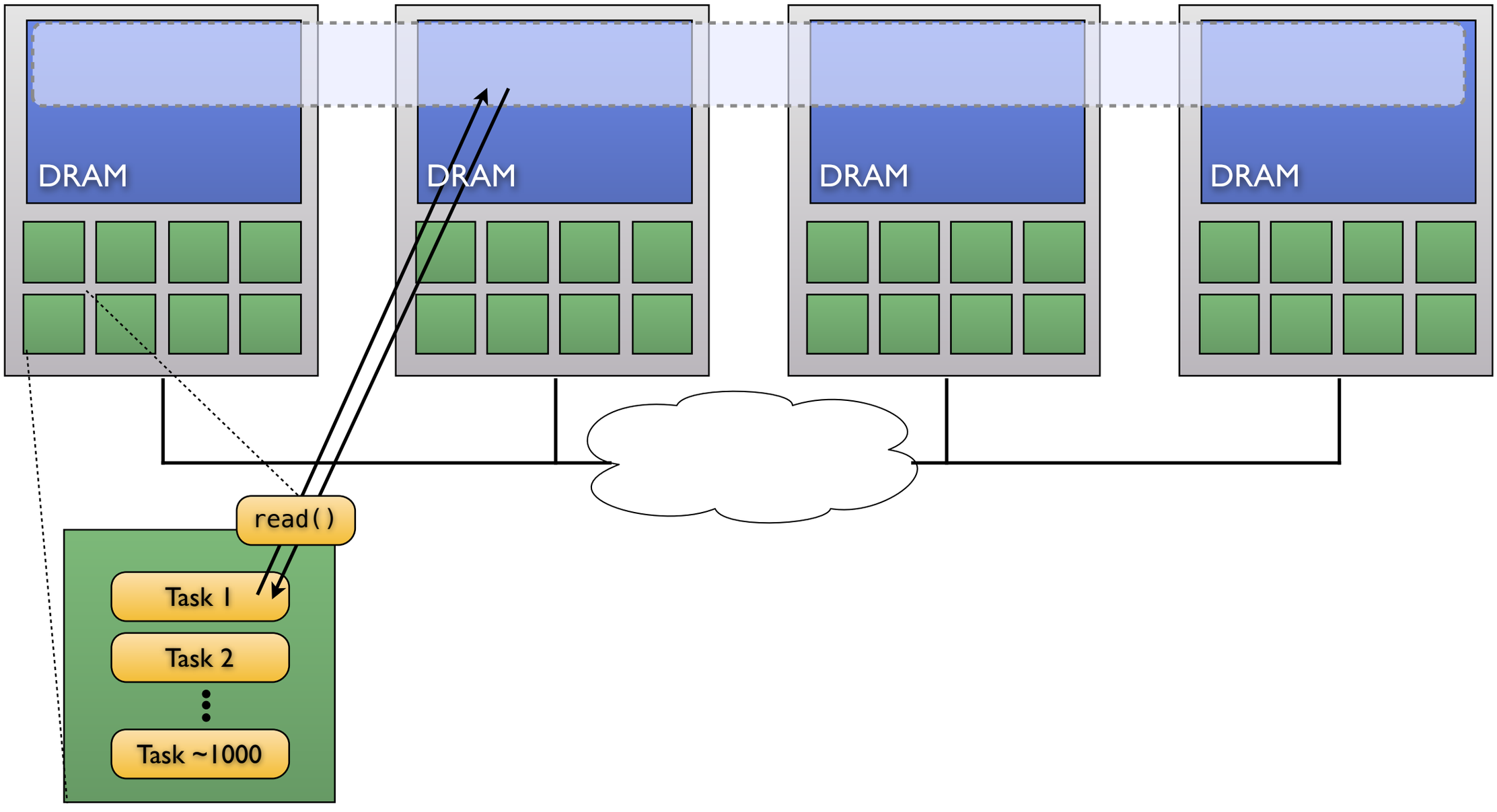
Main idea: tolerate latency with other work



Main idea: tolerate latency with other work



Main idea: tolerate latency with other work

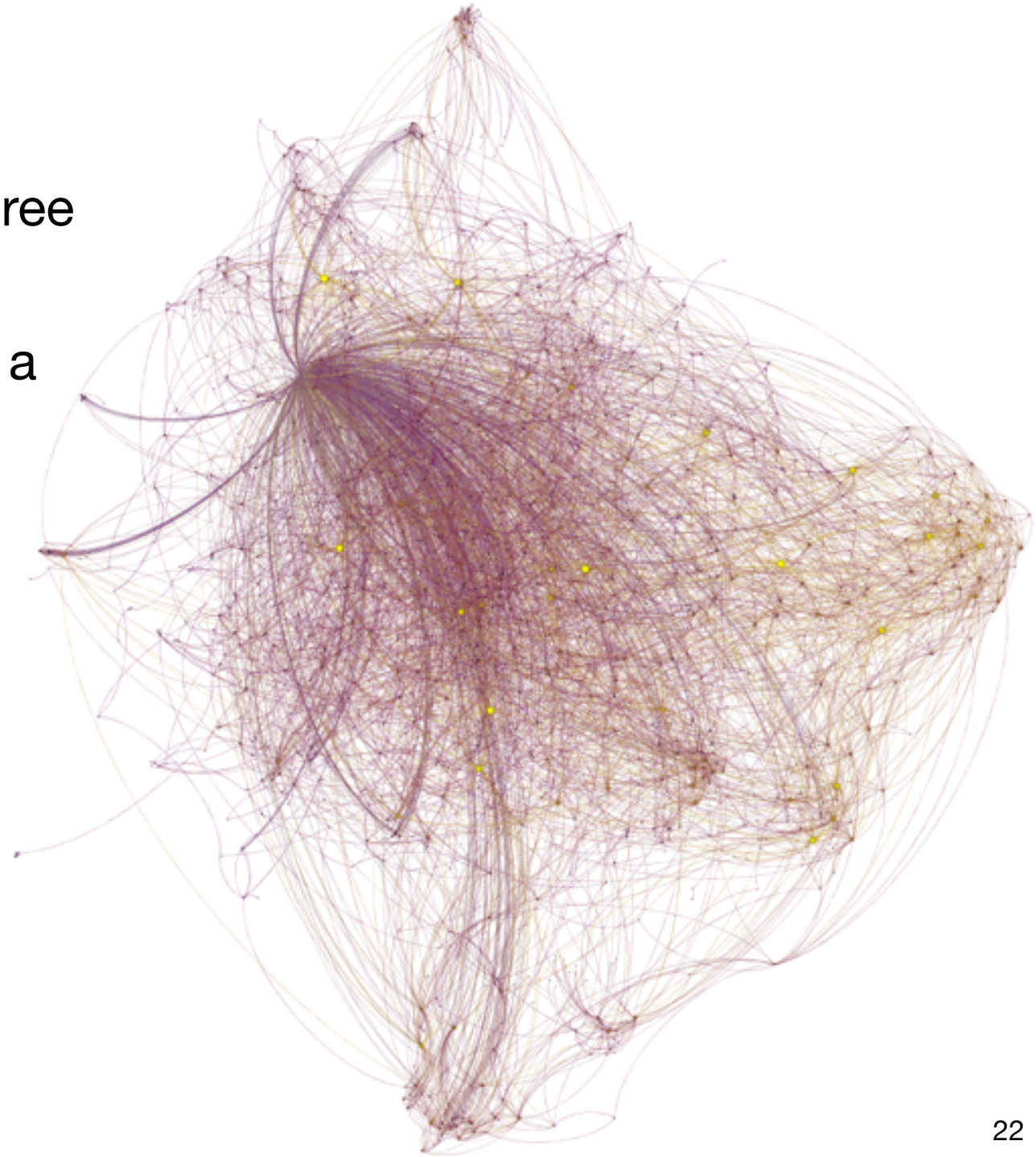


Outline

- 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

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

A single node, serial starting point

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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;
}
```

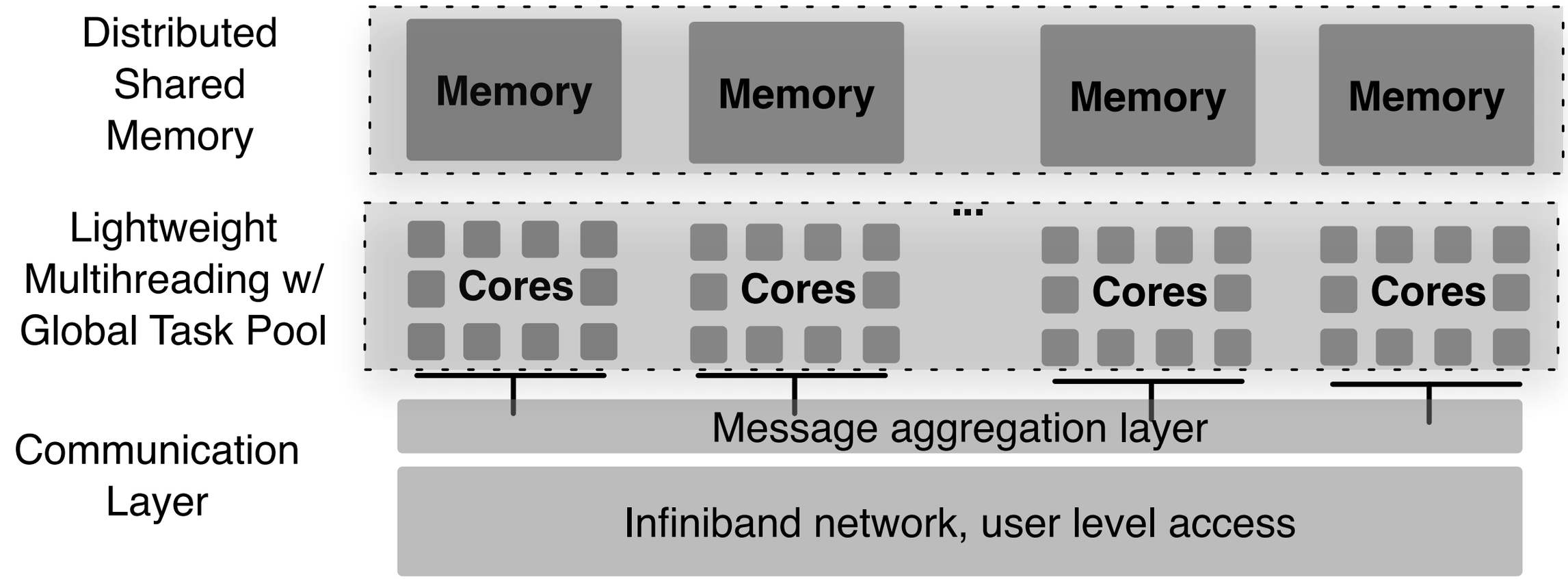
That's it! Grappa code for a cluster!

```
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;
}
```

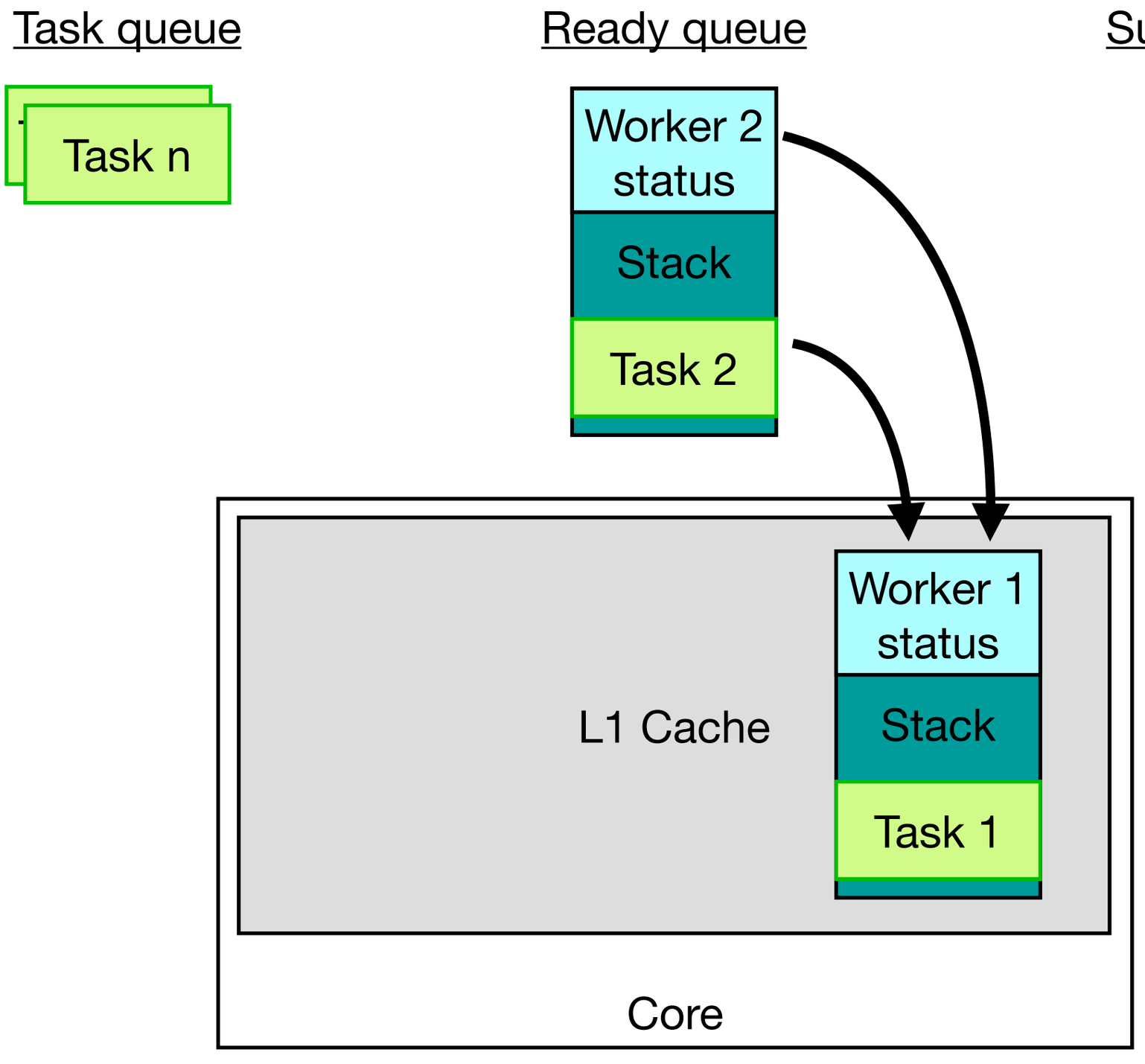
Outline

- Motivation
- Programming Grappa
- Key components
- Performance
- Conclusion

Grappa design



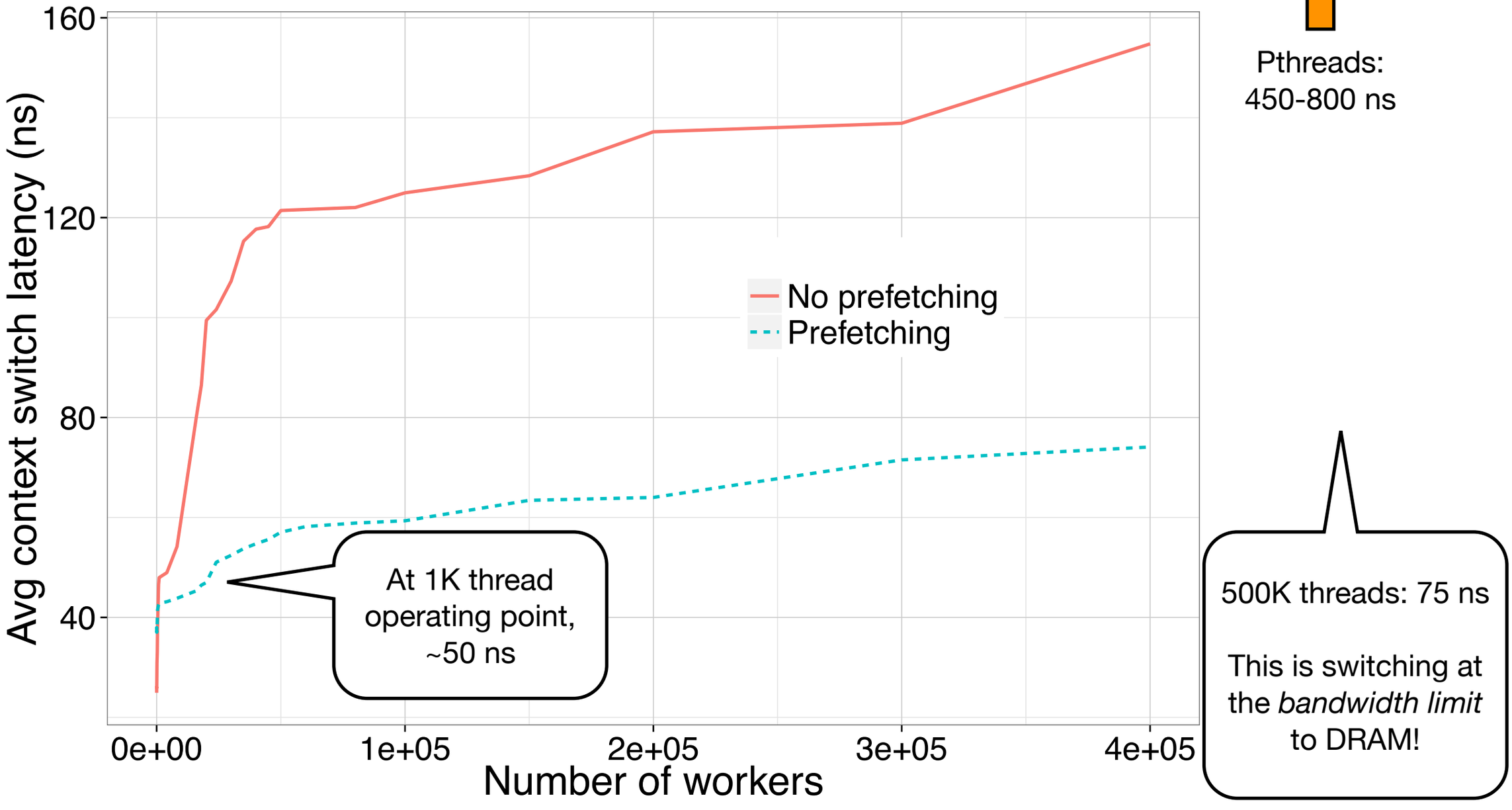
User level context switching



1 cacheline of status,
3 cachelines of stack

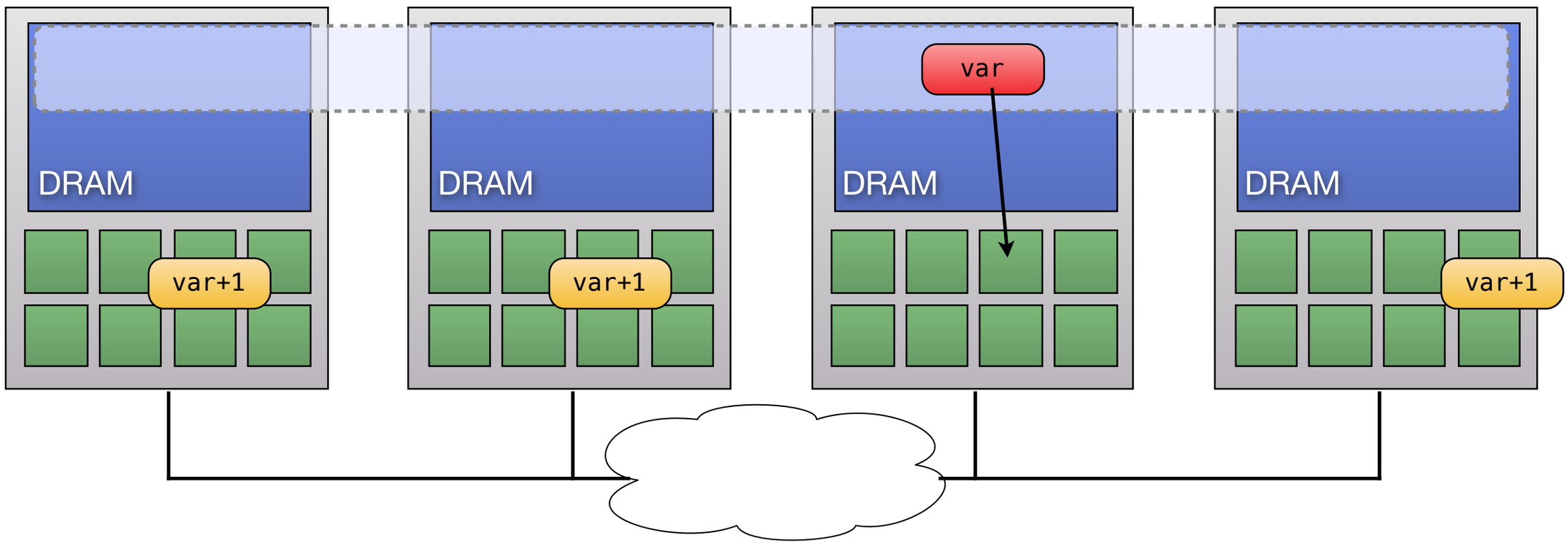
Main innovation:
We *keep state small*
and *prefetch*
to cover DRAM latency

Context switching is fast



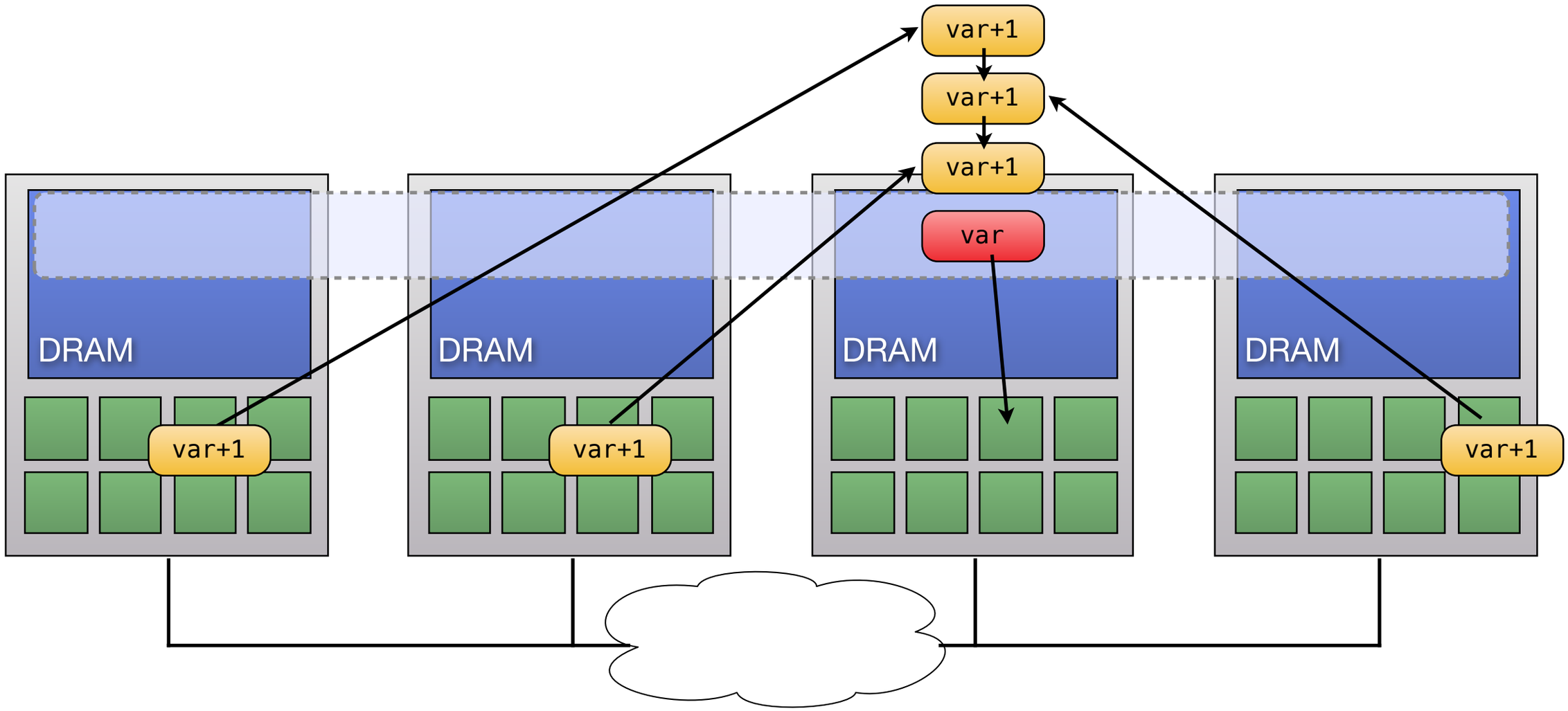
Pthreads:
450-800 ns

Accessing memory through delegates



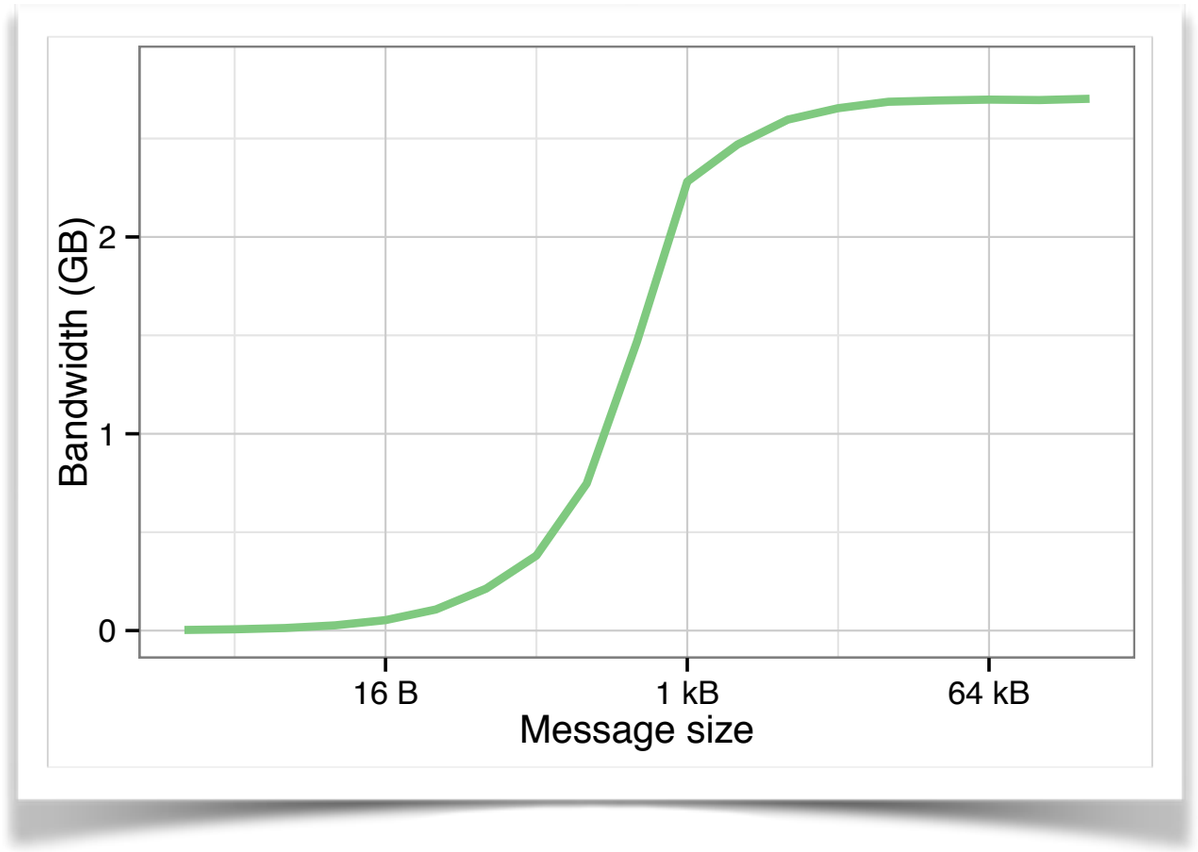
Each word of memory has a designated *home core*
All accesses to that word run on that core
Requestor blocks until complete

Accessing memory through delegates

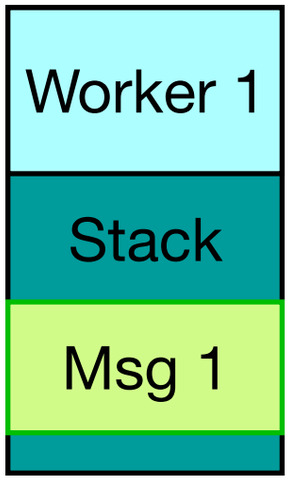


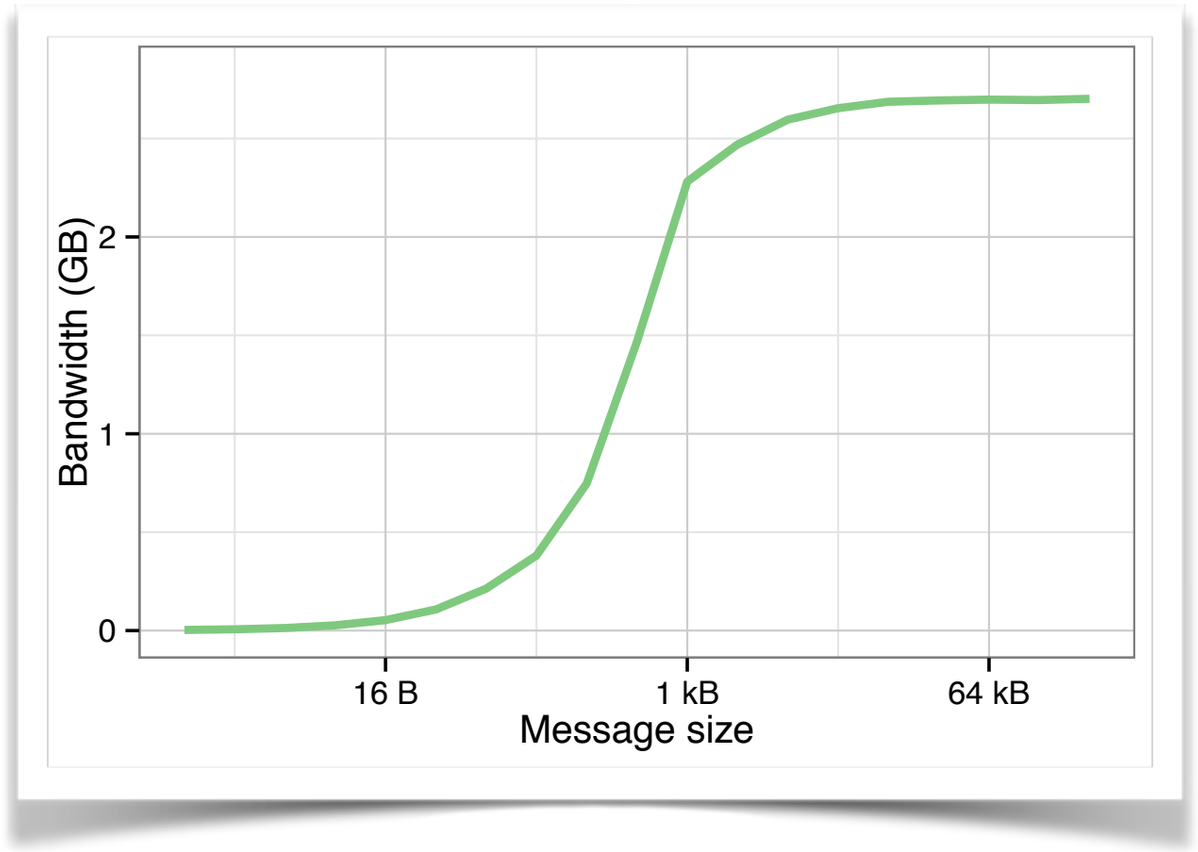
Since var is private to home core,
updates can be applied without expensive synchronization

Mitigating low injection rate with aggregation



Mitigating low injection rate with aggregation

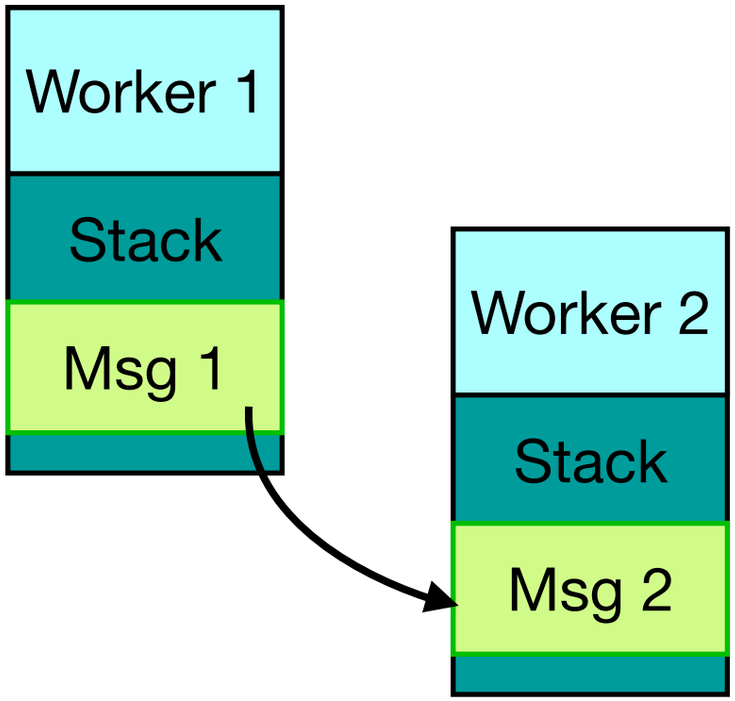


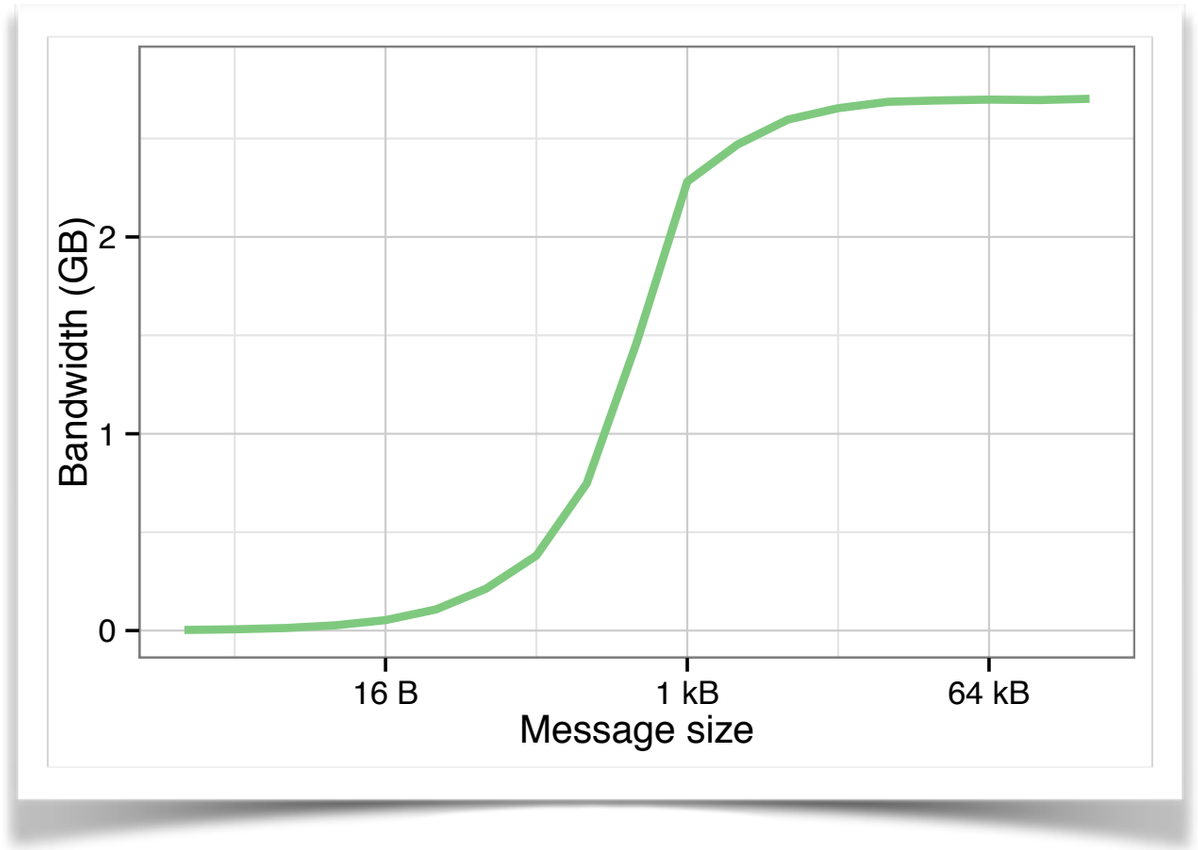


Node 0

Node *n*

Mitigating low injection rate with aggregation

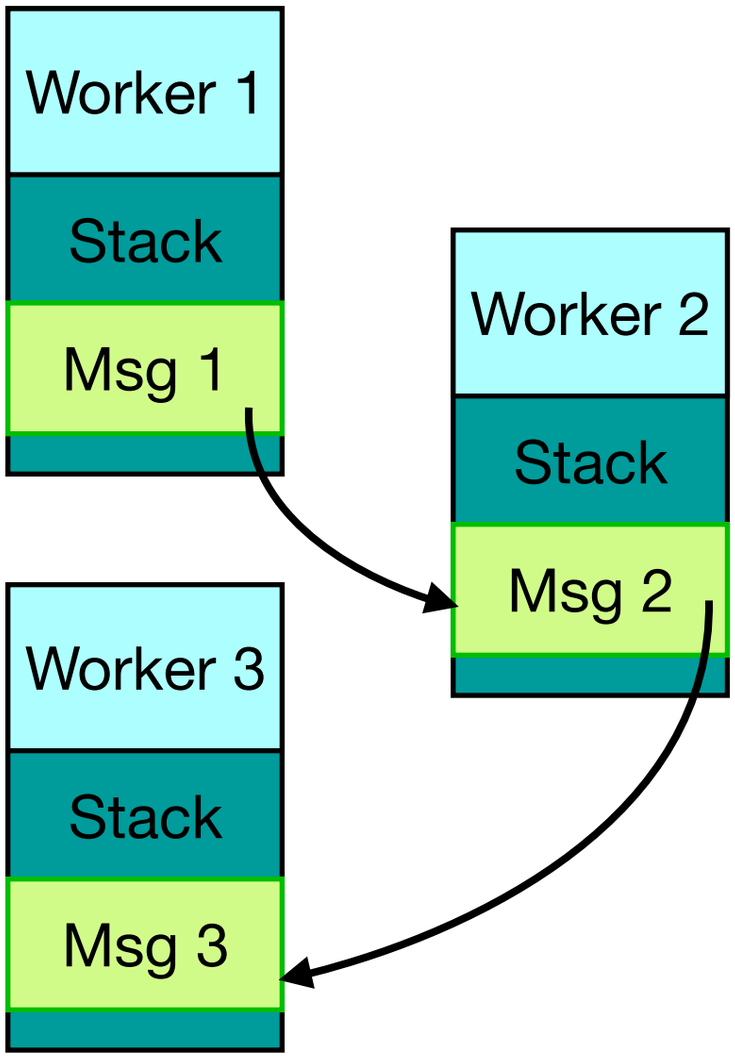


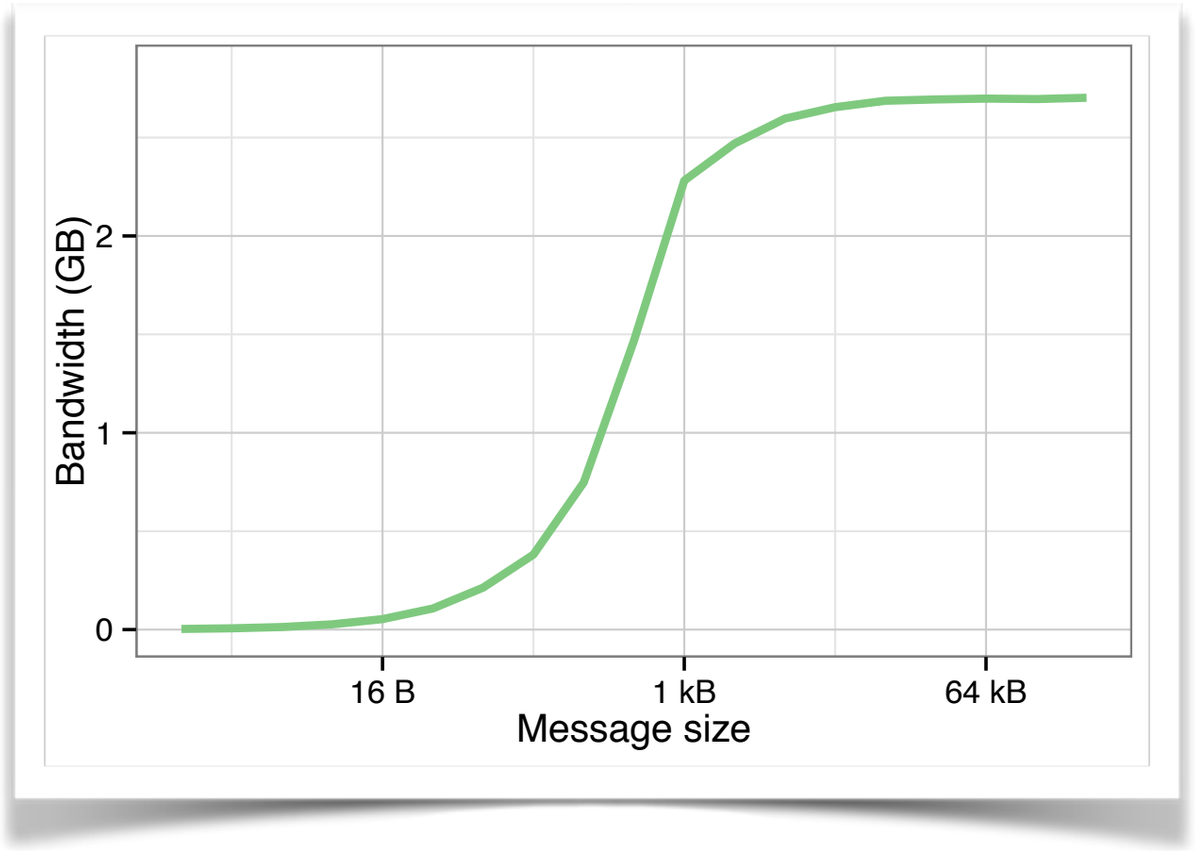


Node 0

Node *n*

Mitigating low injection rate with aggregation

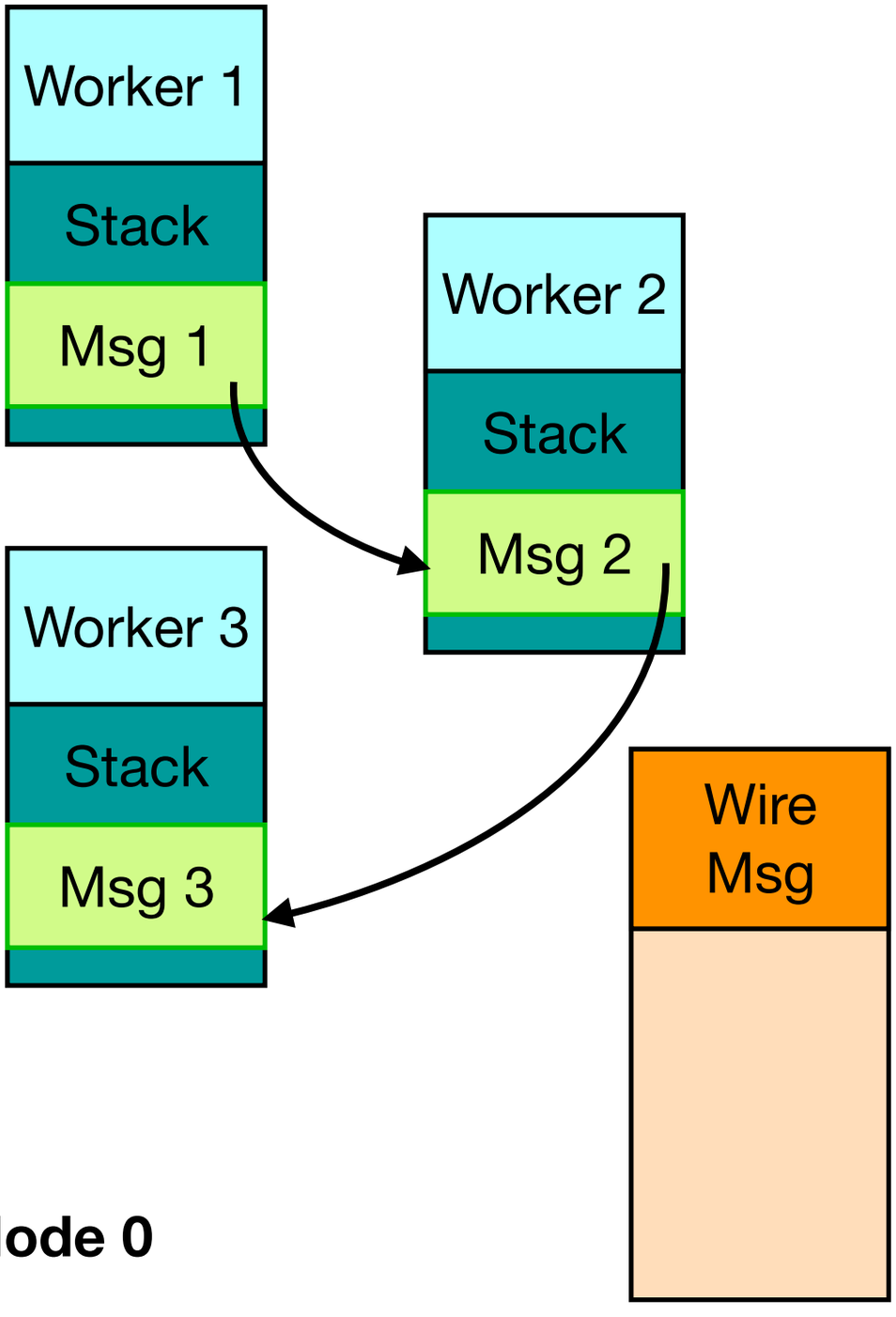


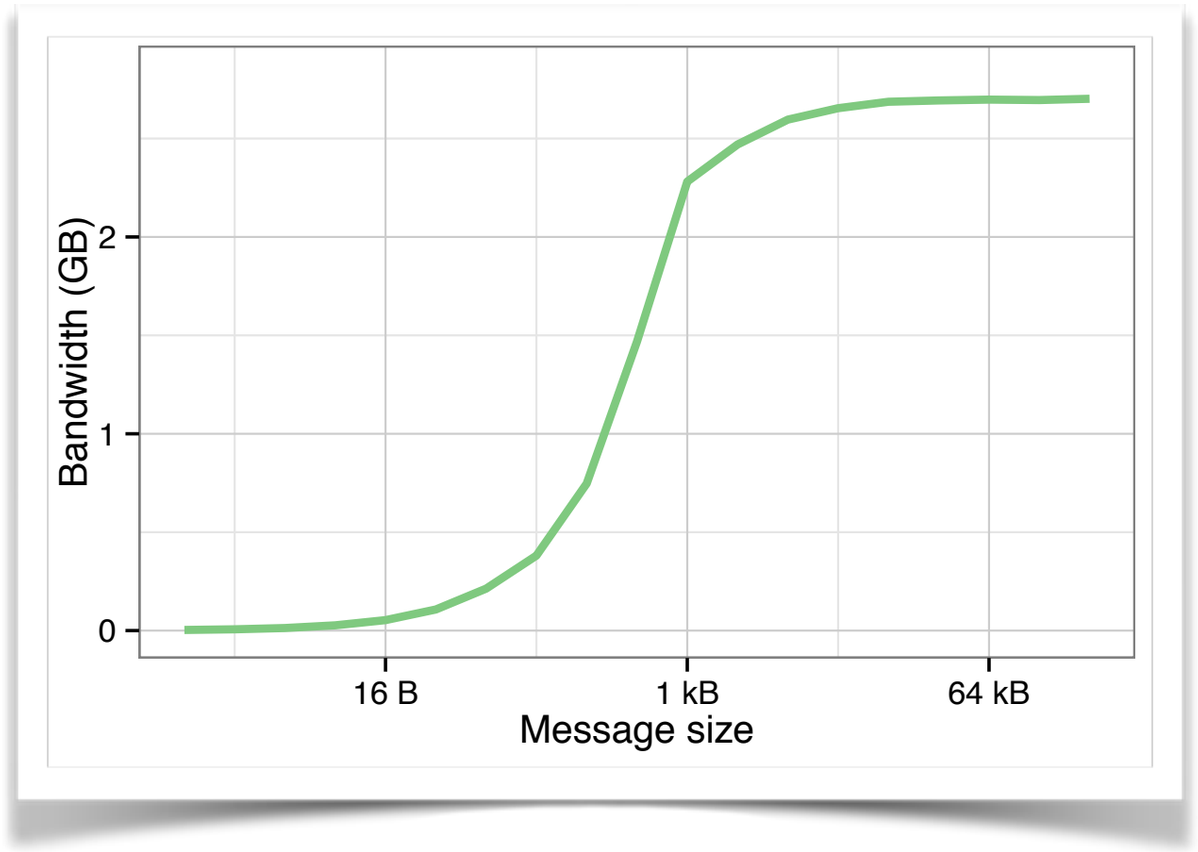


Node 0

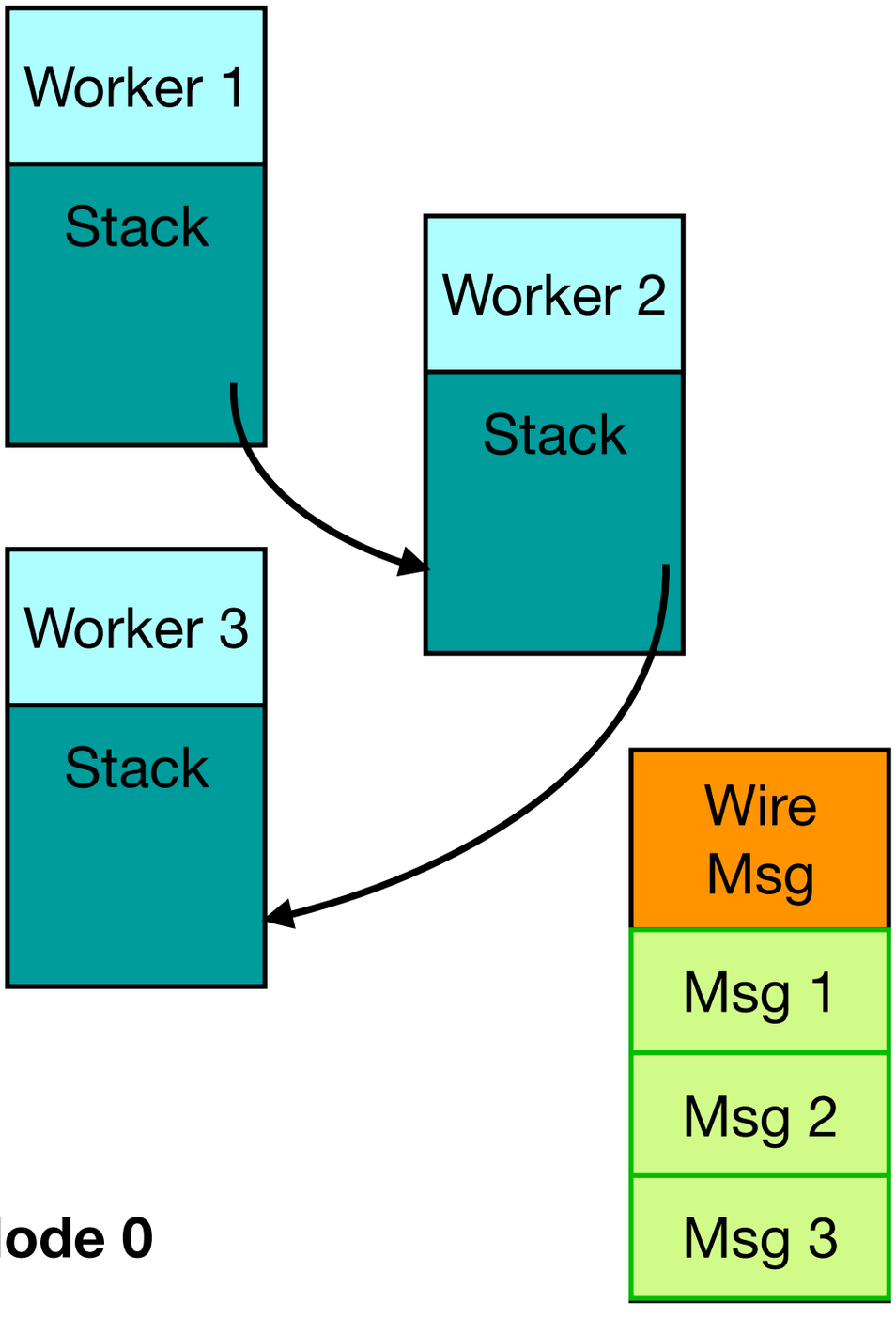
Node *n*

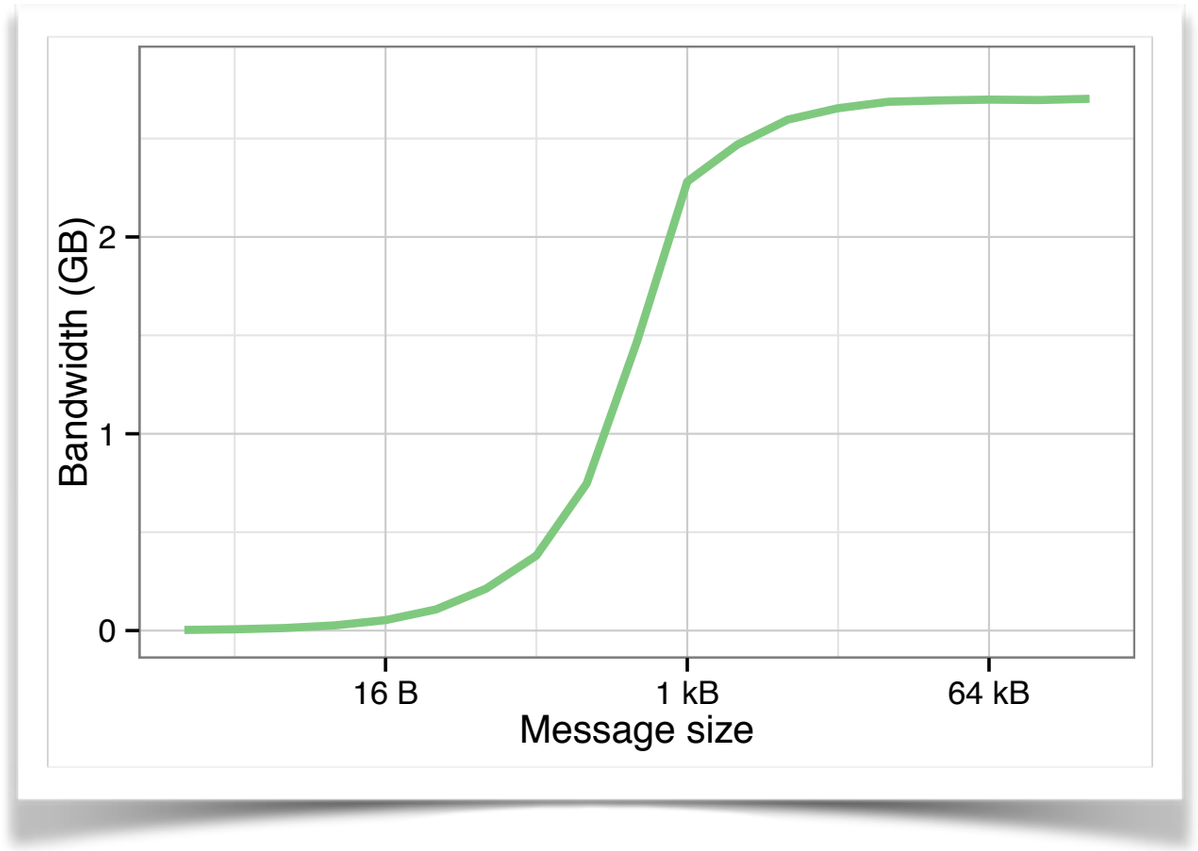
Mitigating low injection rate with aggregation



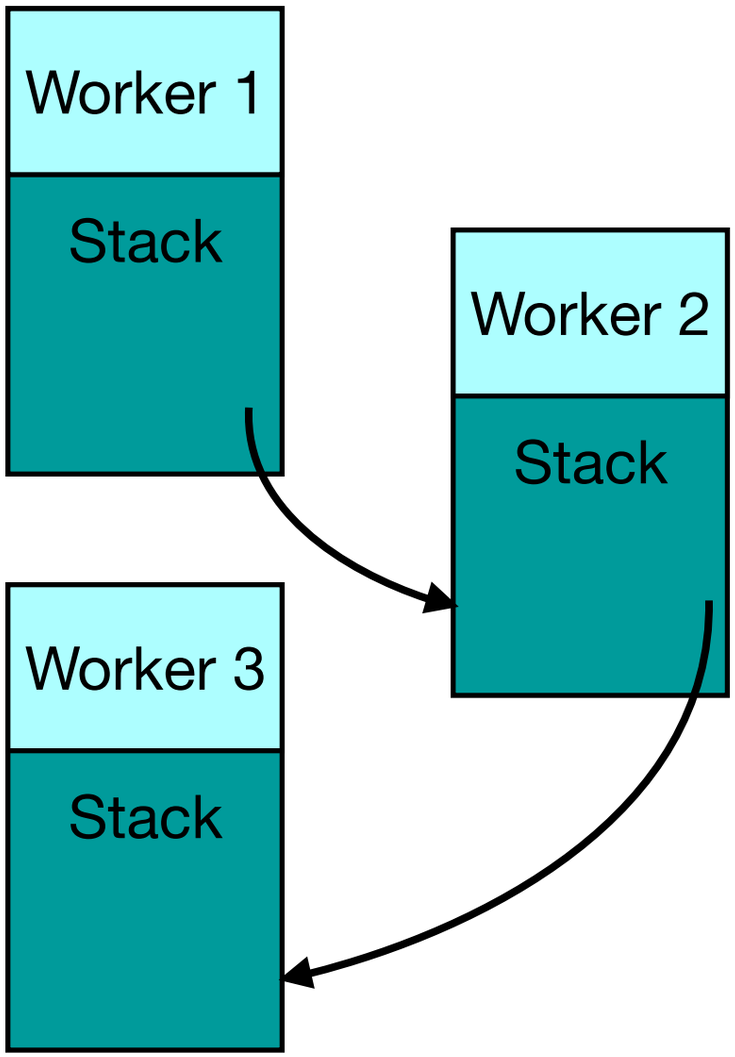


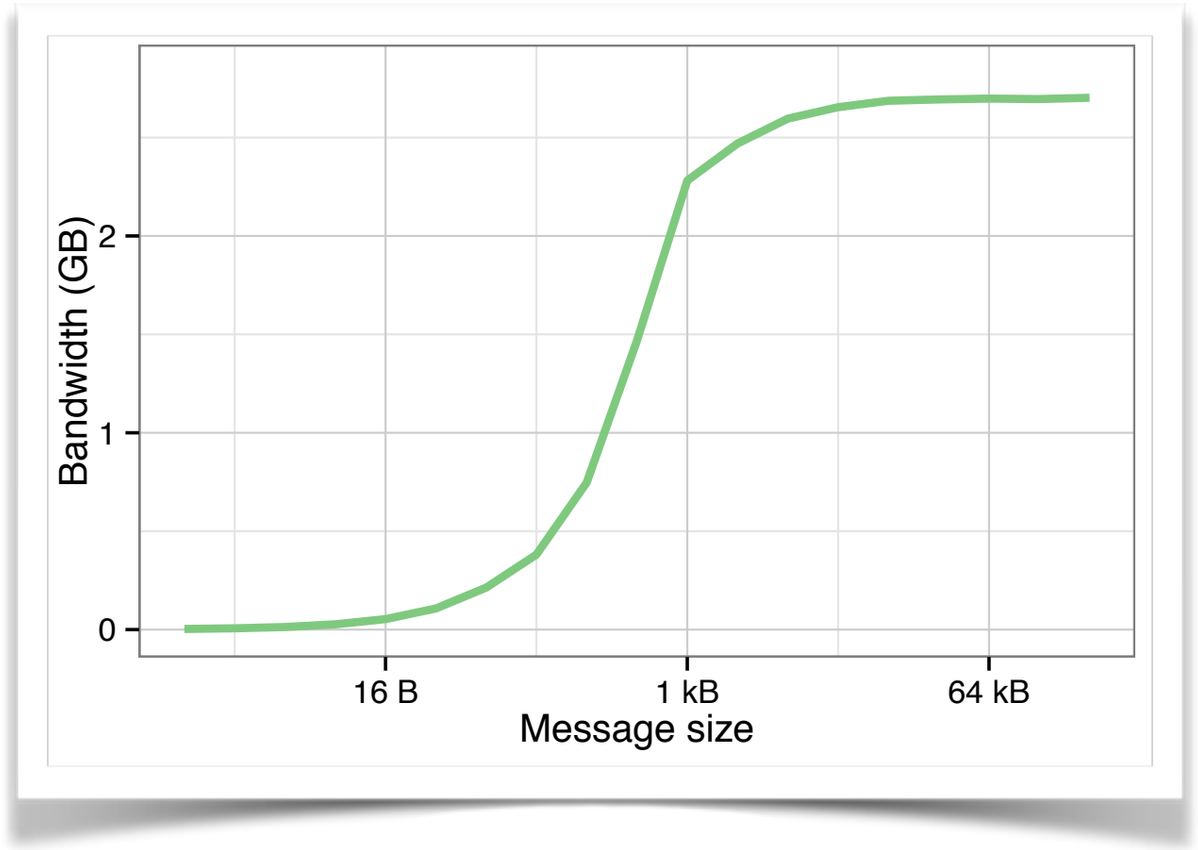
Mitigating low injection rate with aggregation

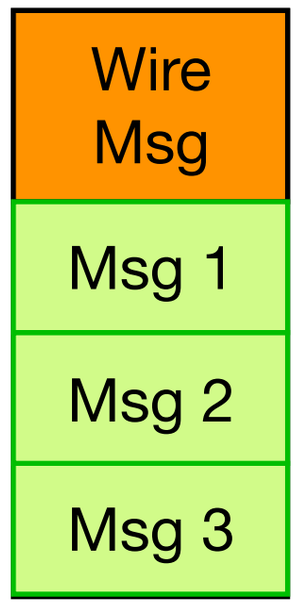




Mitigating low injection rate with aggregation



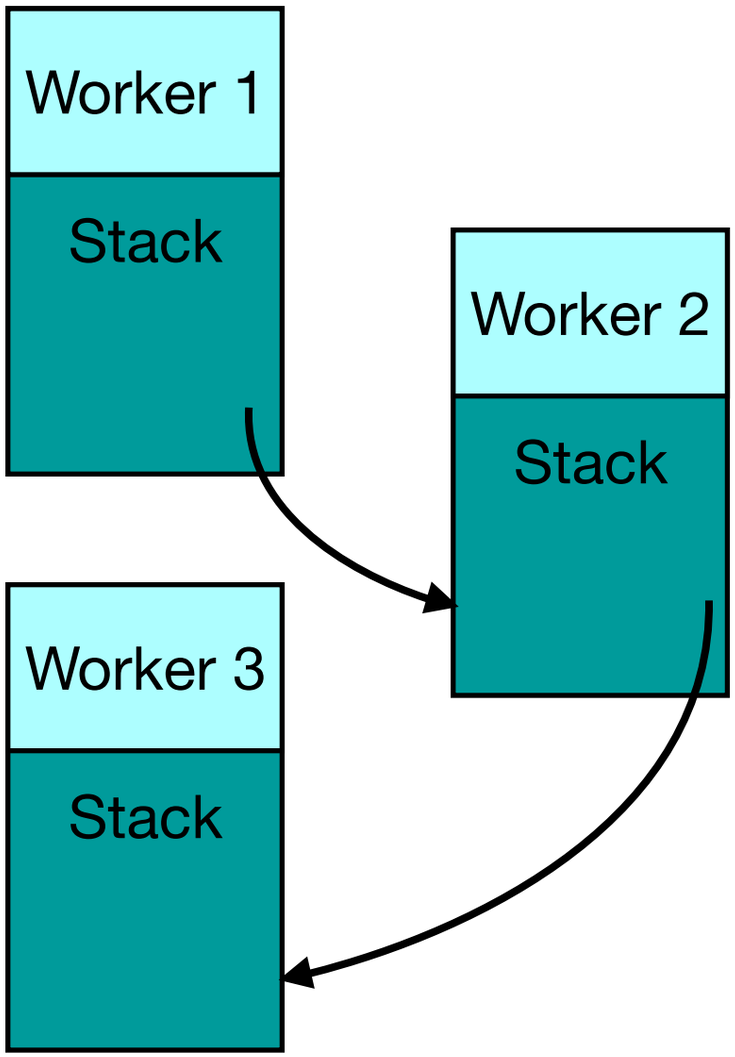


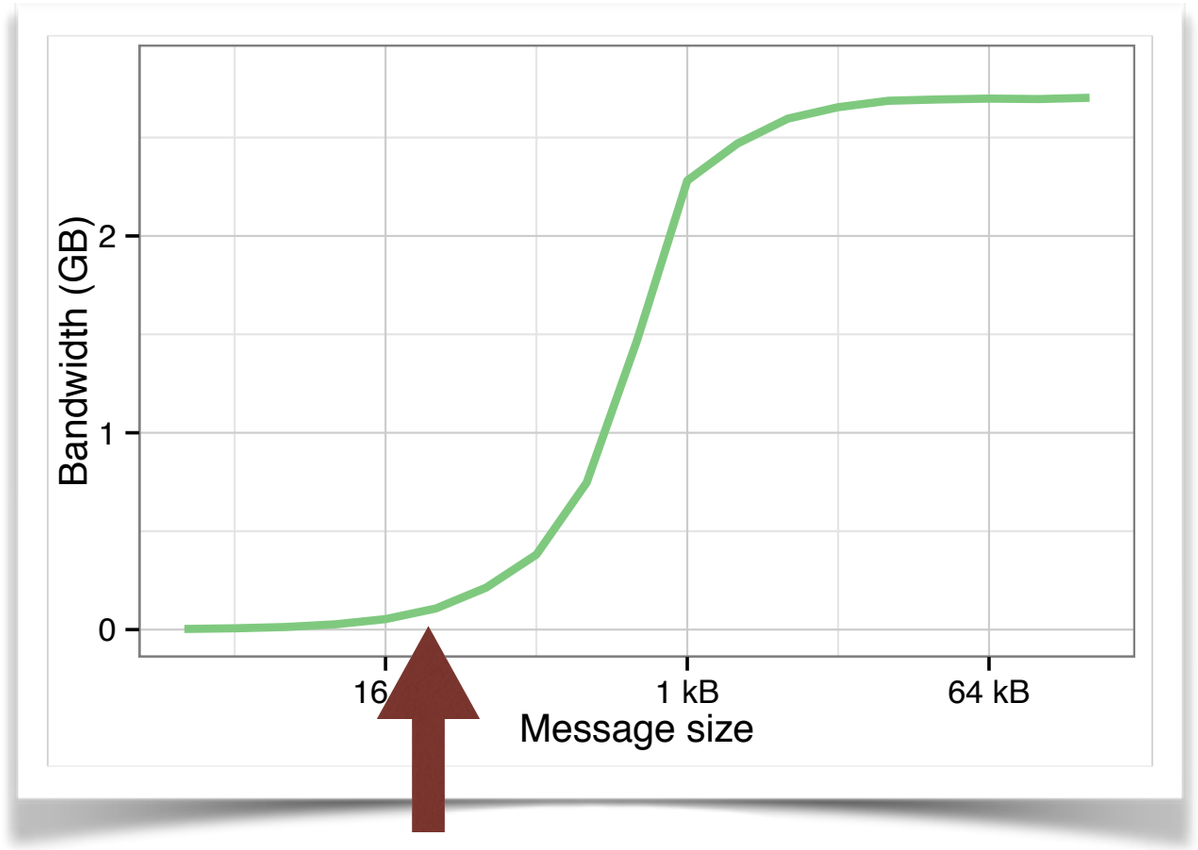


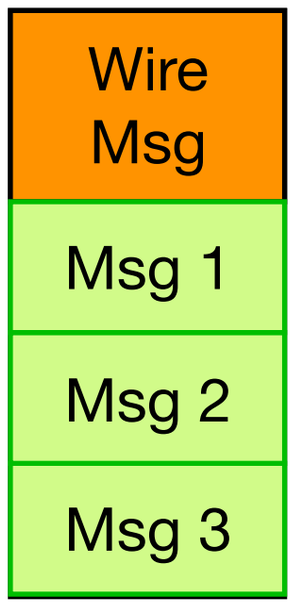
Node 0

Node *n*

Mitigating low injection rate with aggregation



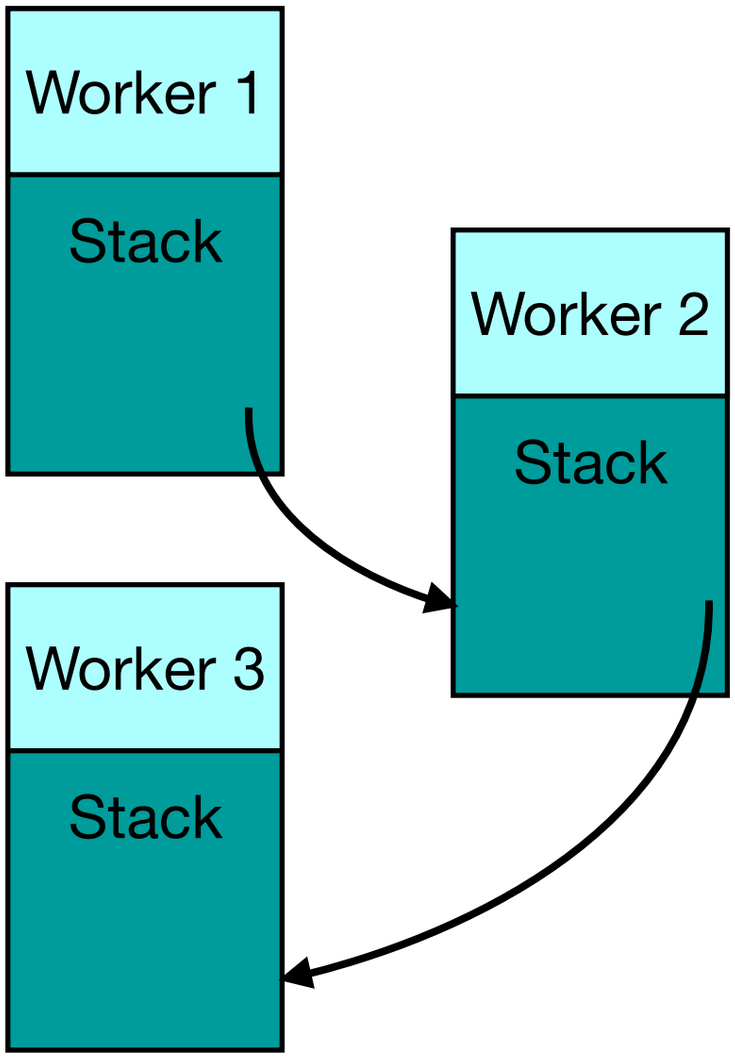


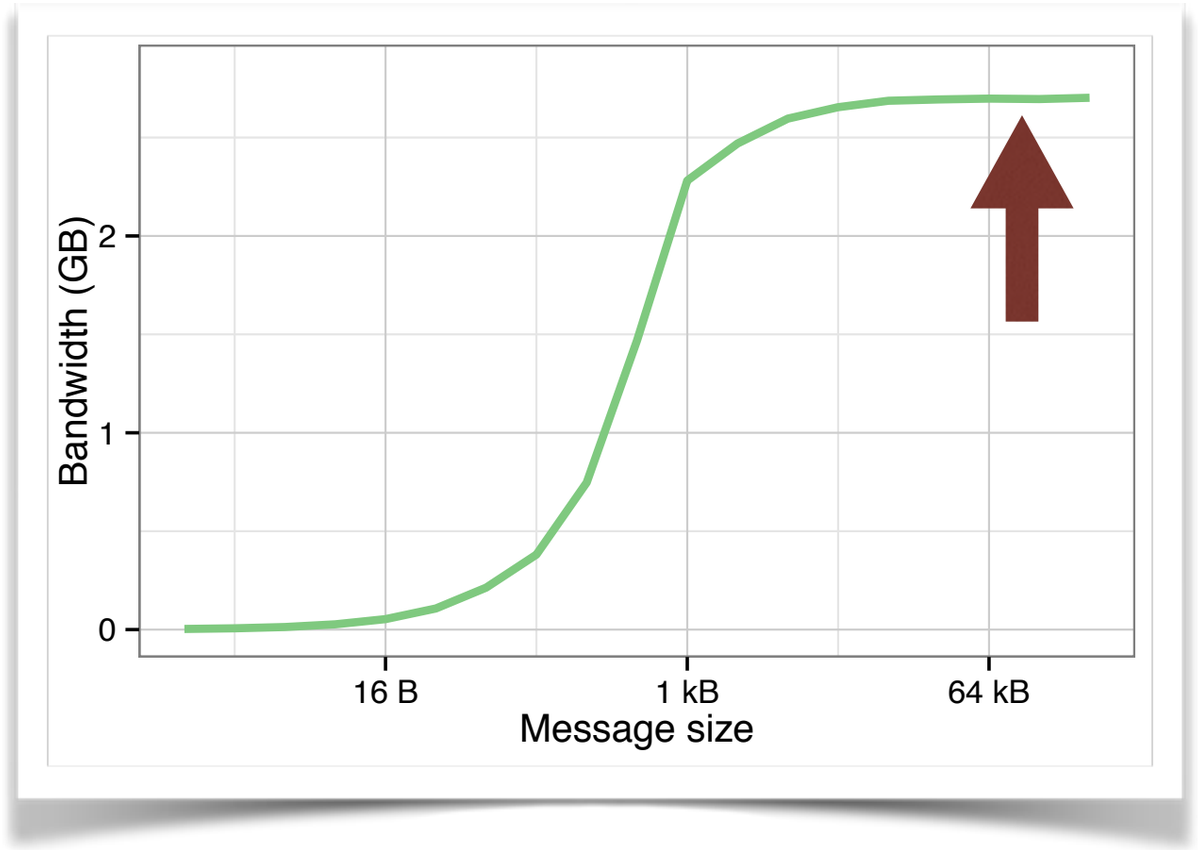


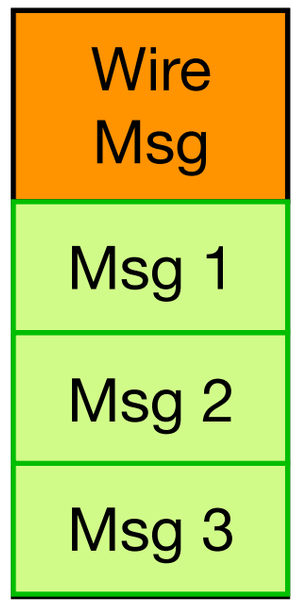
Node 0

Node *n*

Mitigating low injection rate with aggregation



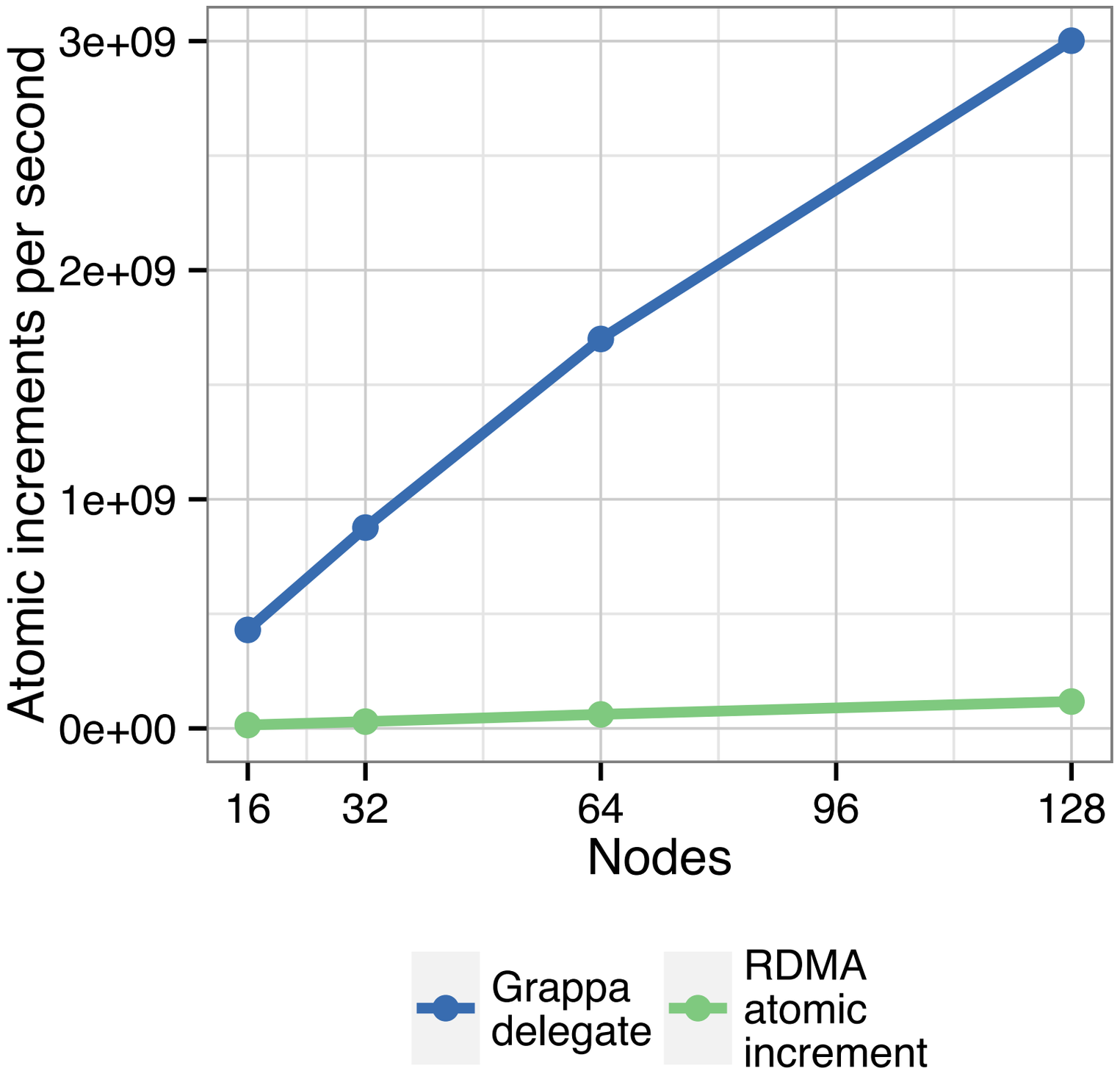




Node 0

Node *n*

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

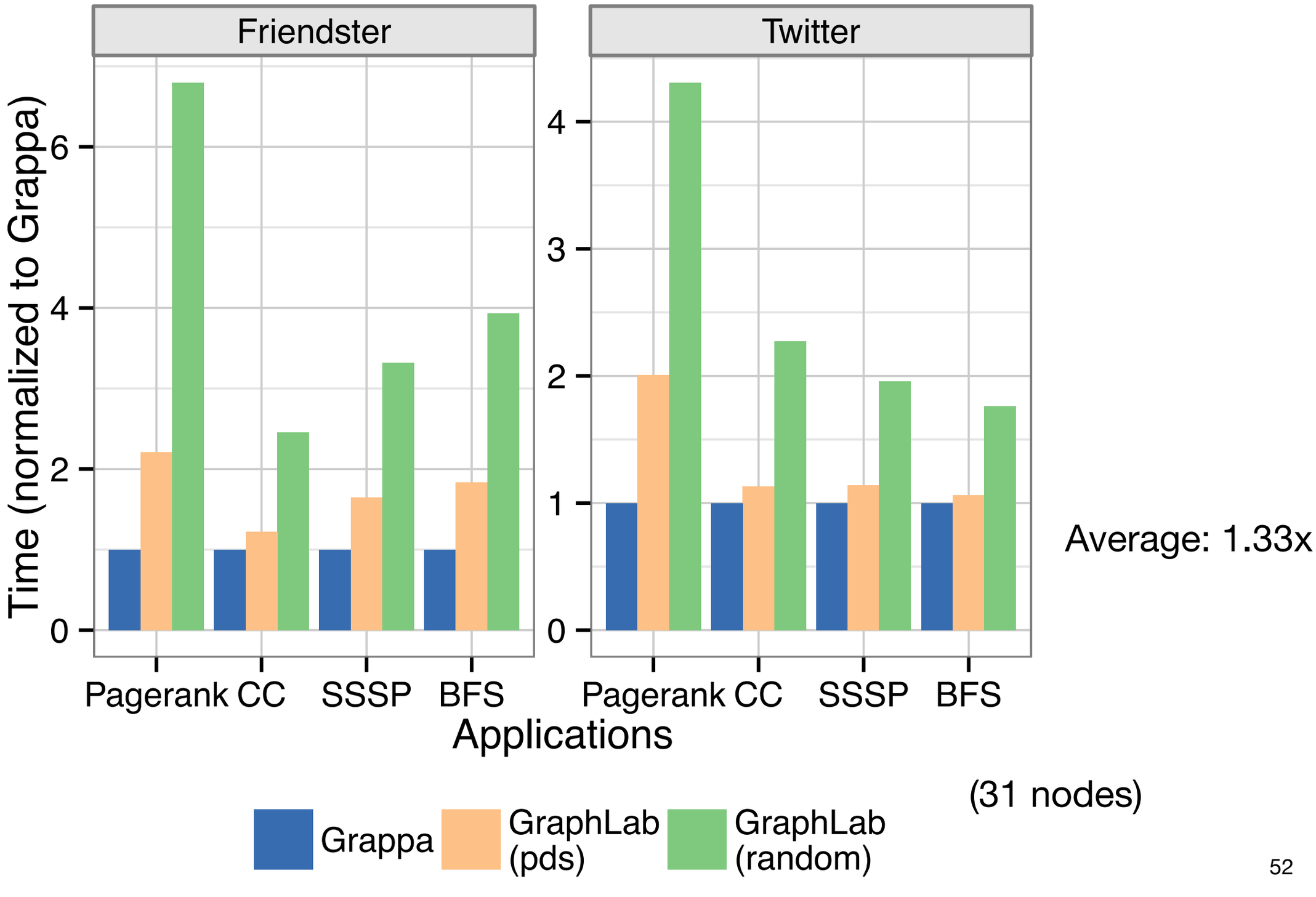
- GraphLab: replicated graph representation and complex partitioning strategy;
  Grappa: simple adjacency list and random partitioning

- 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)

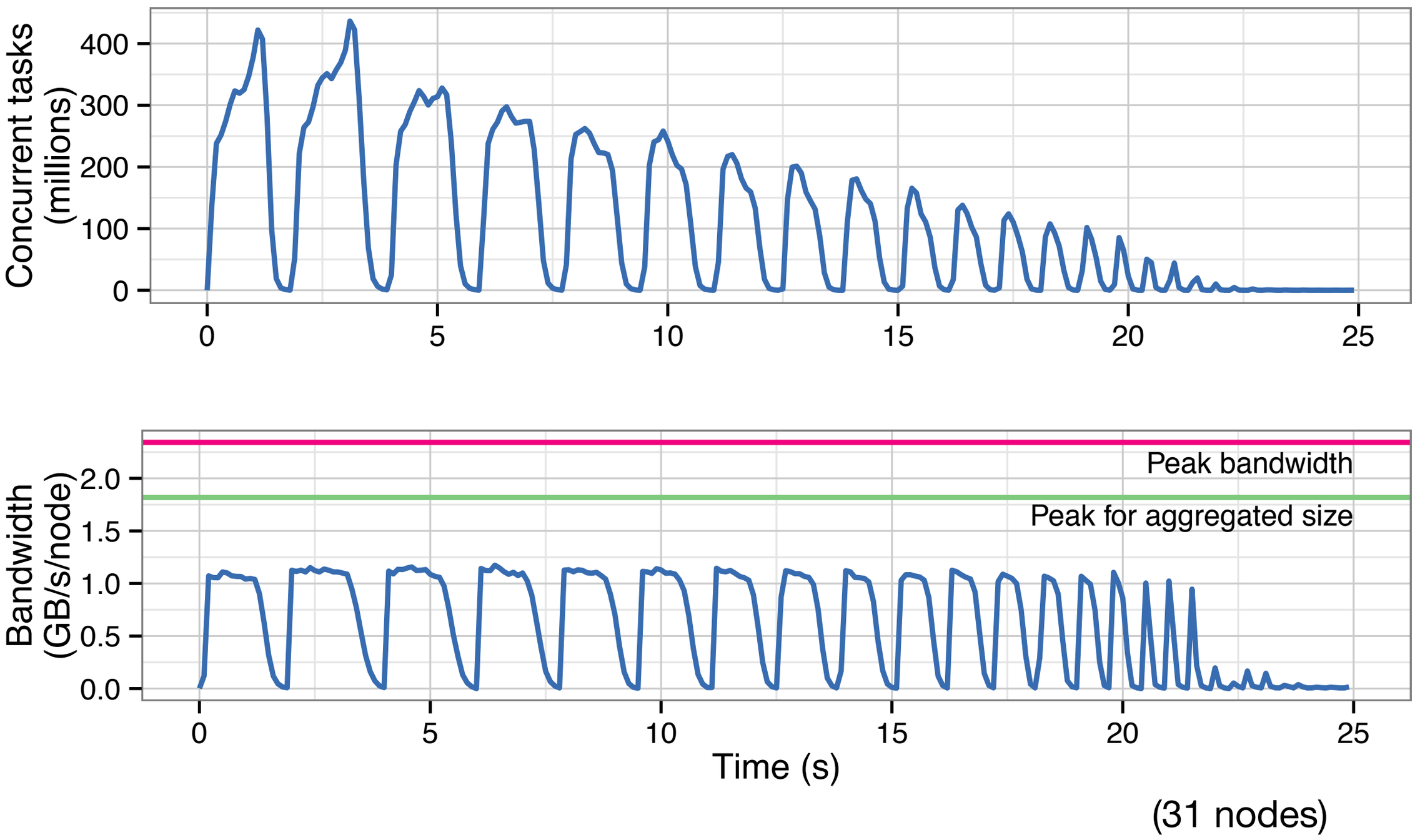
Grappa/GraphLab application performance



Average: 1.33x

(31 nodes)

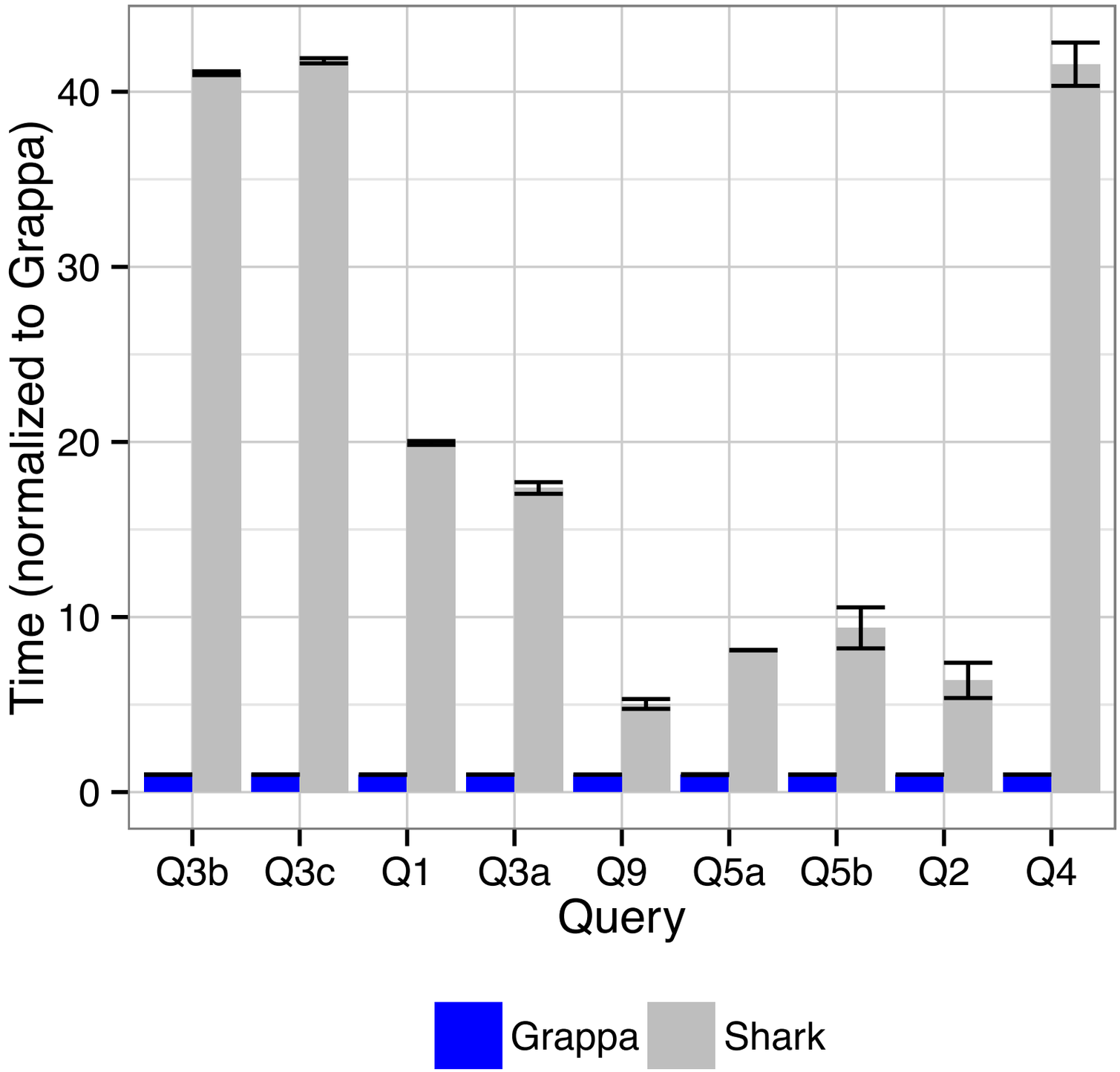
A closer look at PageRank



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

Relational query execution



Average: 14x

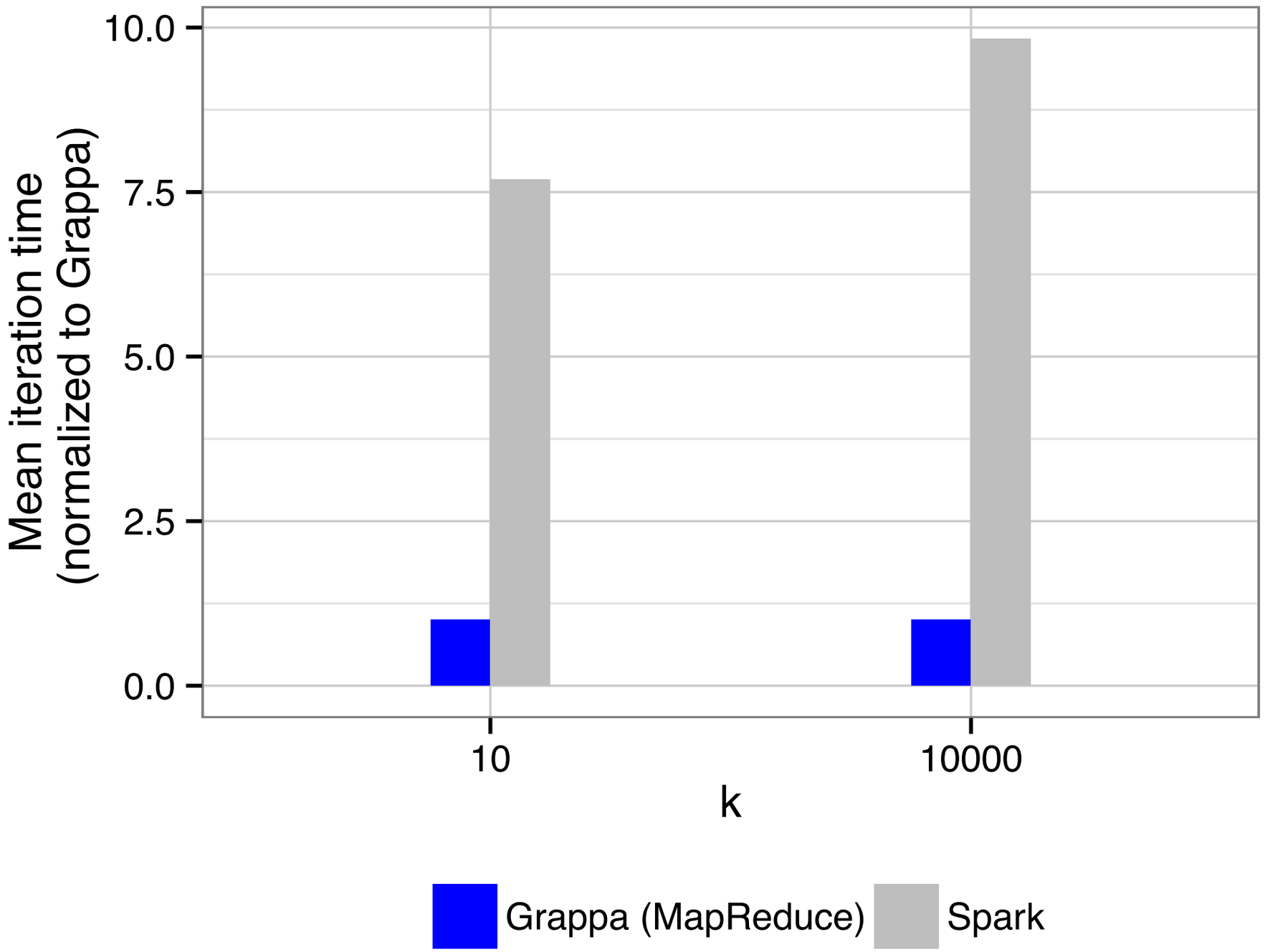
In-memory Map/Reduce

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

- 152 lines

- Compared with Spark, configured to avoid fault-tolerance

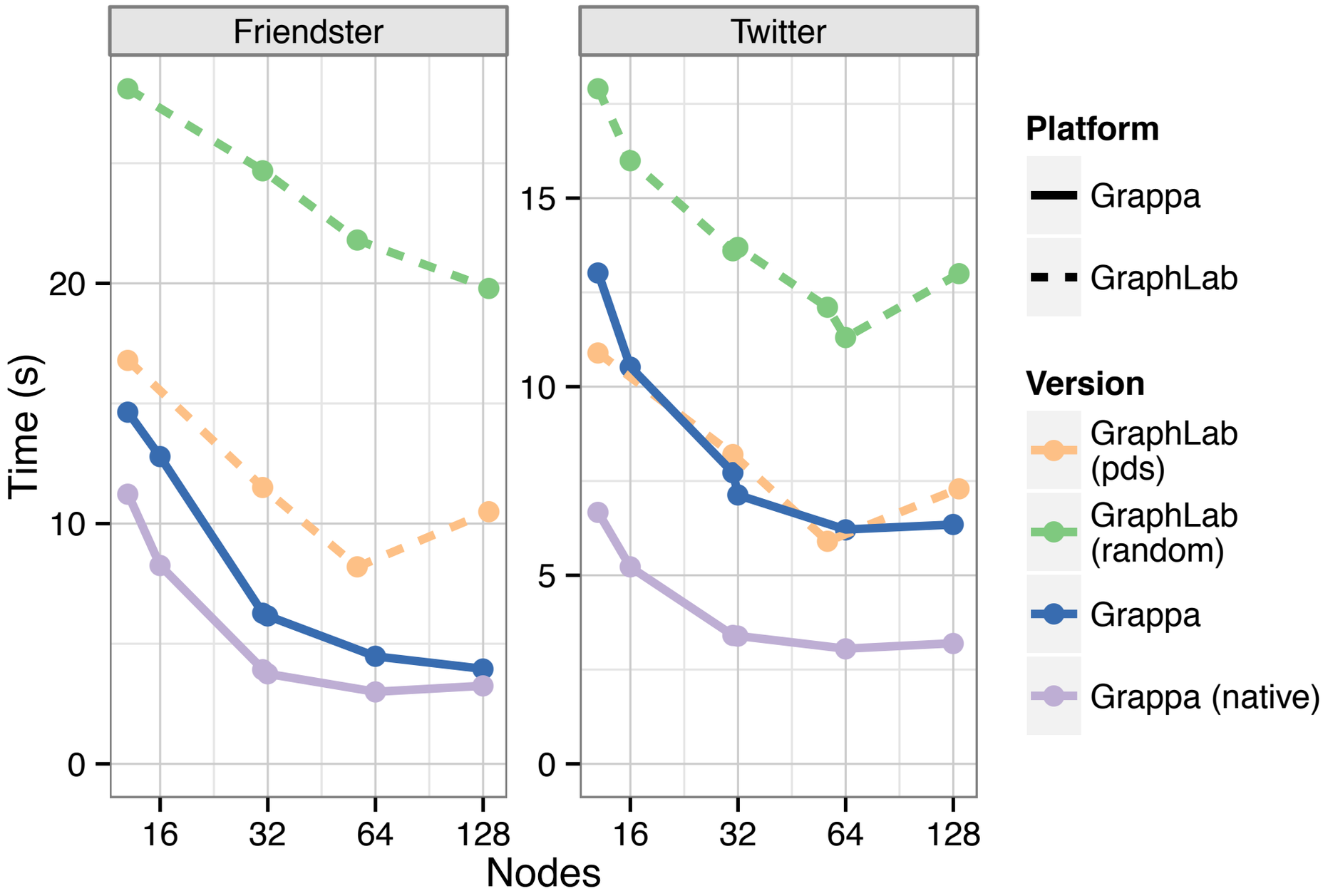
- Benchmark: K-Means on SeaFlow dataset (8.9GB)


10.0
7.5
5.0
2.5
0.0
Mean iteration time
(normalized to Grappa)
10
10000
k
Grappa (MapReduce)
Spark

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

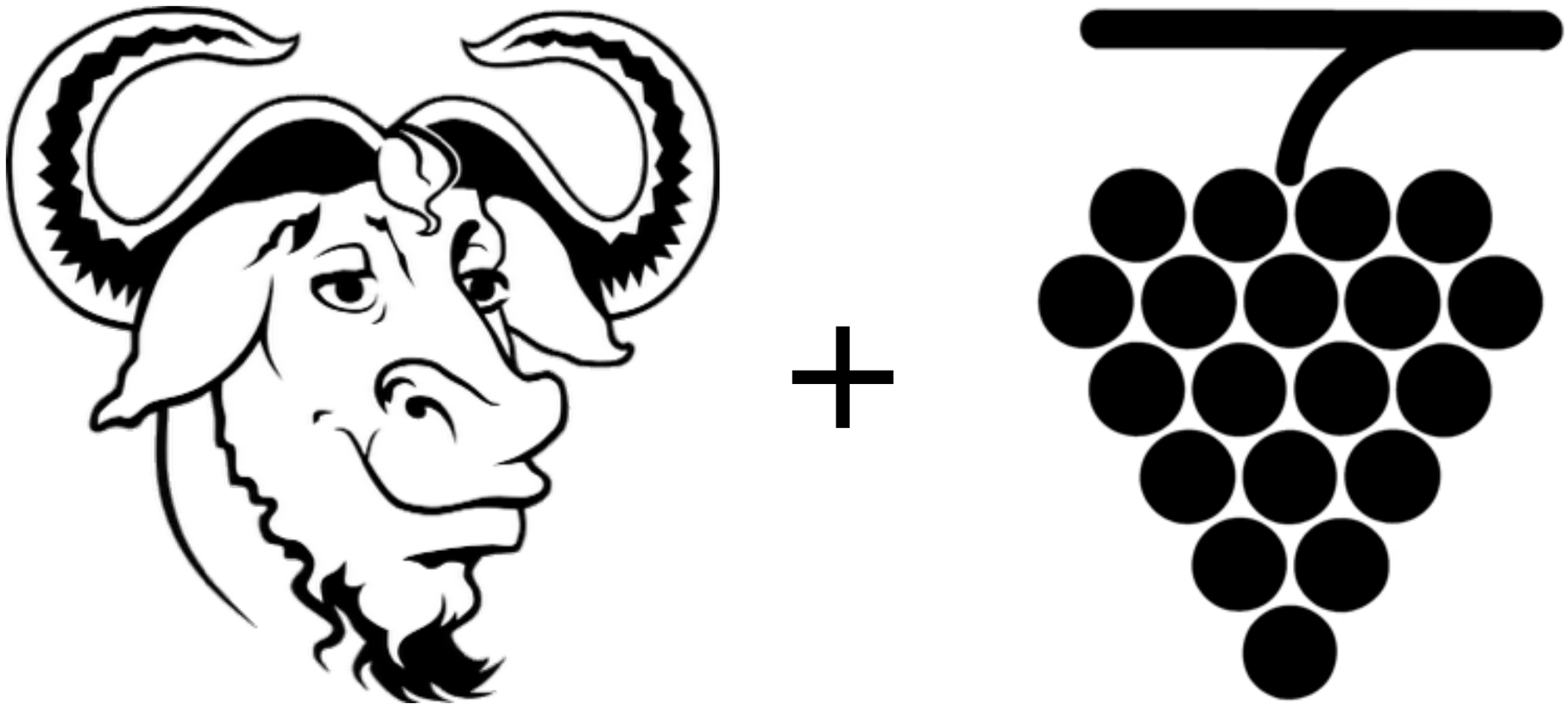
Writing against Grappa directly



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?