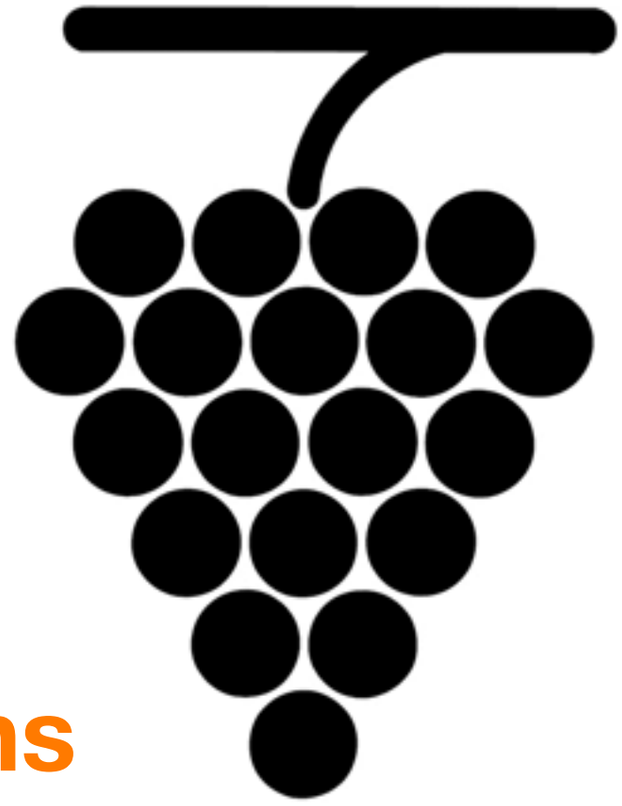
Grappa:

A latency tolerant runtime for large-scale irregular applications



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

University of Washington

January 21, 2014

A tale of two programmers

blupics@flickr

Pat's problem: traverse an unbalanced tree

- Tree is embedded in a graph

- ~1T edges in graph
  ~1B edges in tree

- Power law, low diameter

How about a big shared-memory machine?

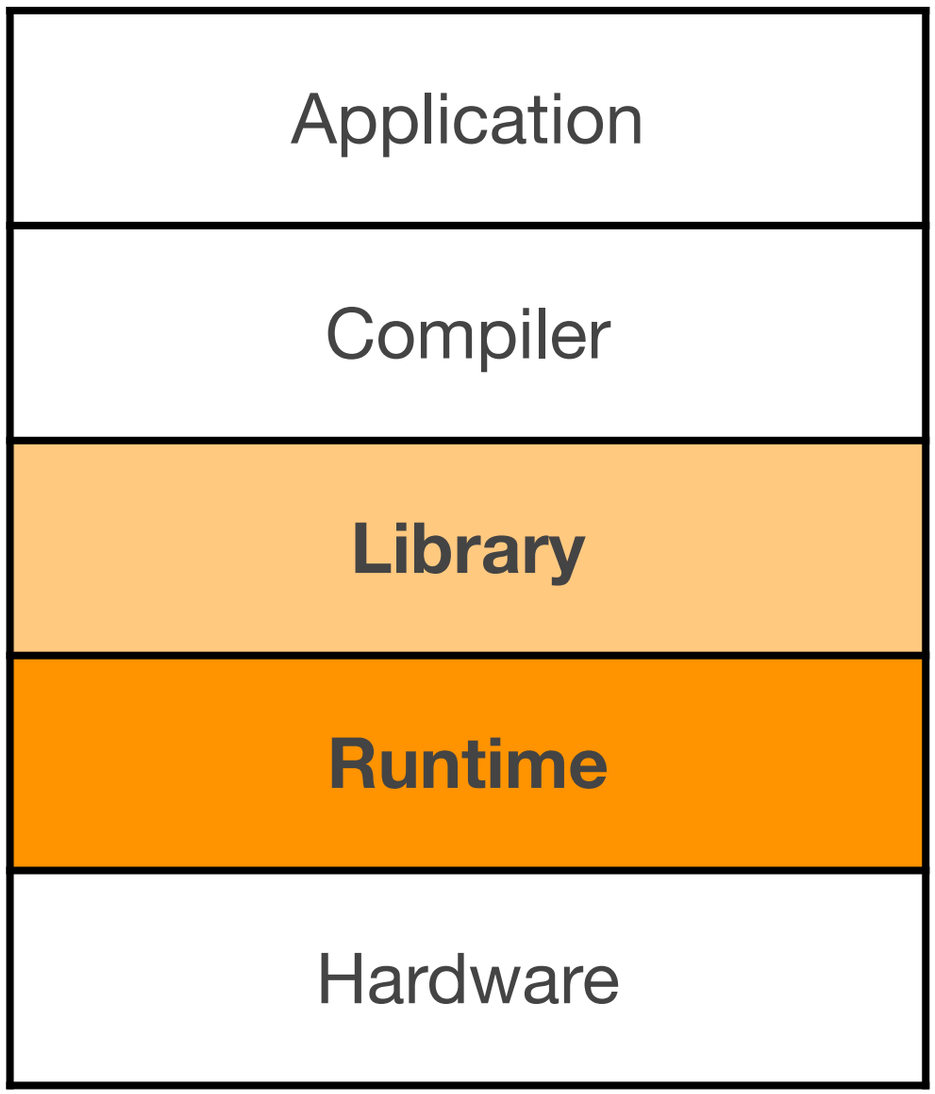
How about special purpose hardware?

How about a commodity cluster?

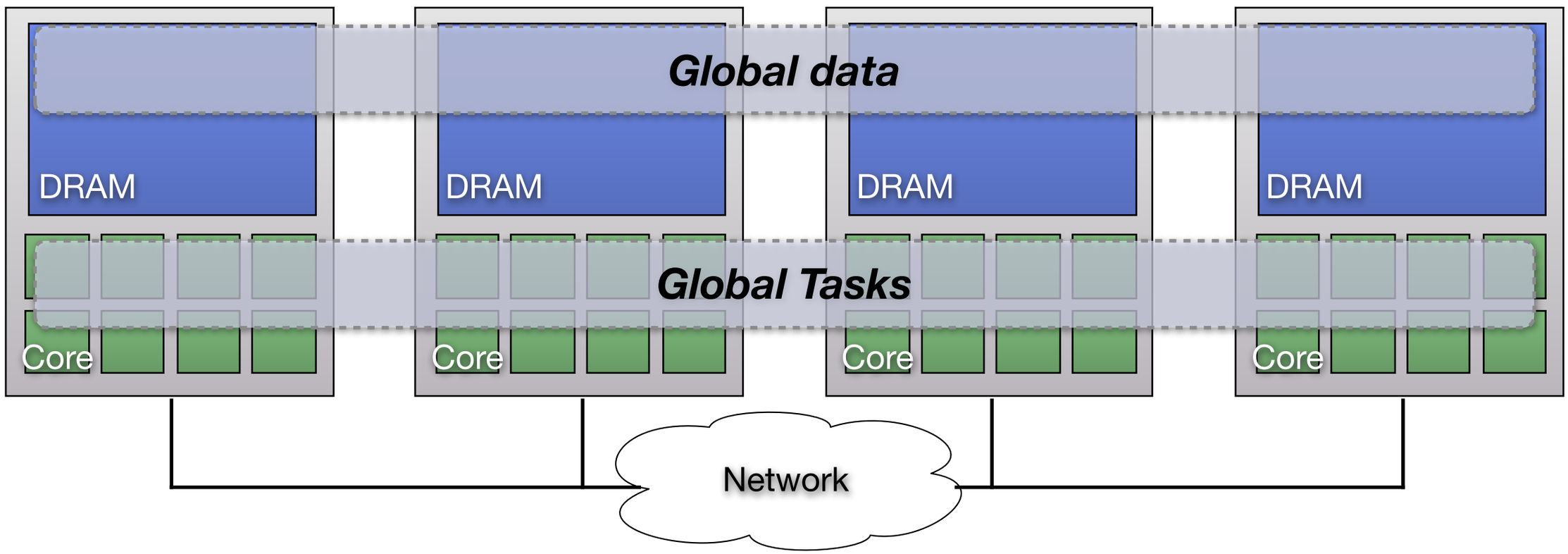
Grappa

- Goal: Provide irregular application programmers what they want
  - Global view programming model
  - Good small-message performance
  - Tasks, threads, latency tolerance
  - Fine-grained synchronization
  - Load balancing

Where is Grappa in the stack?

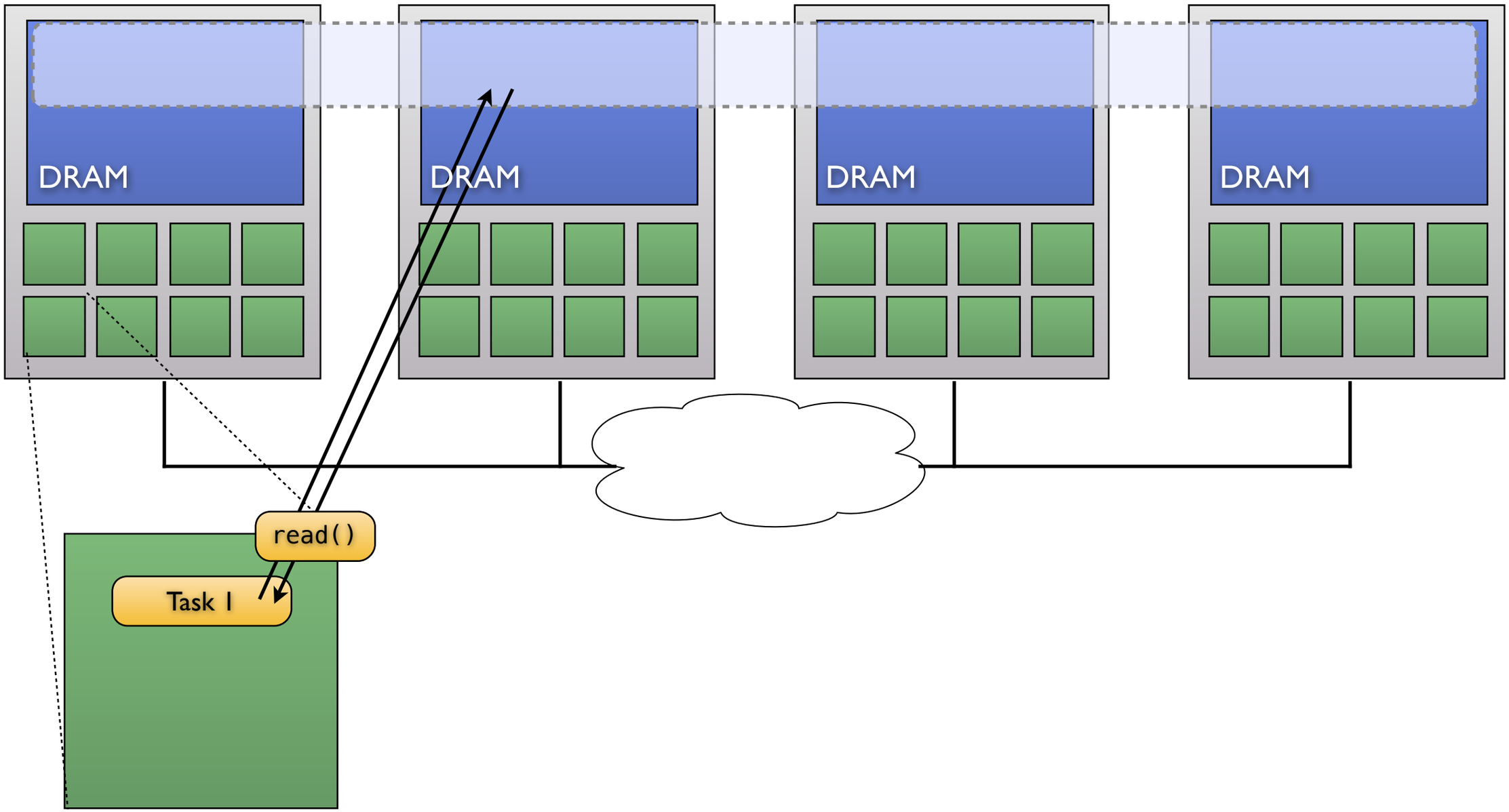


Grappa's system view

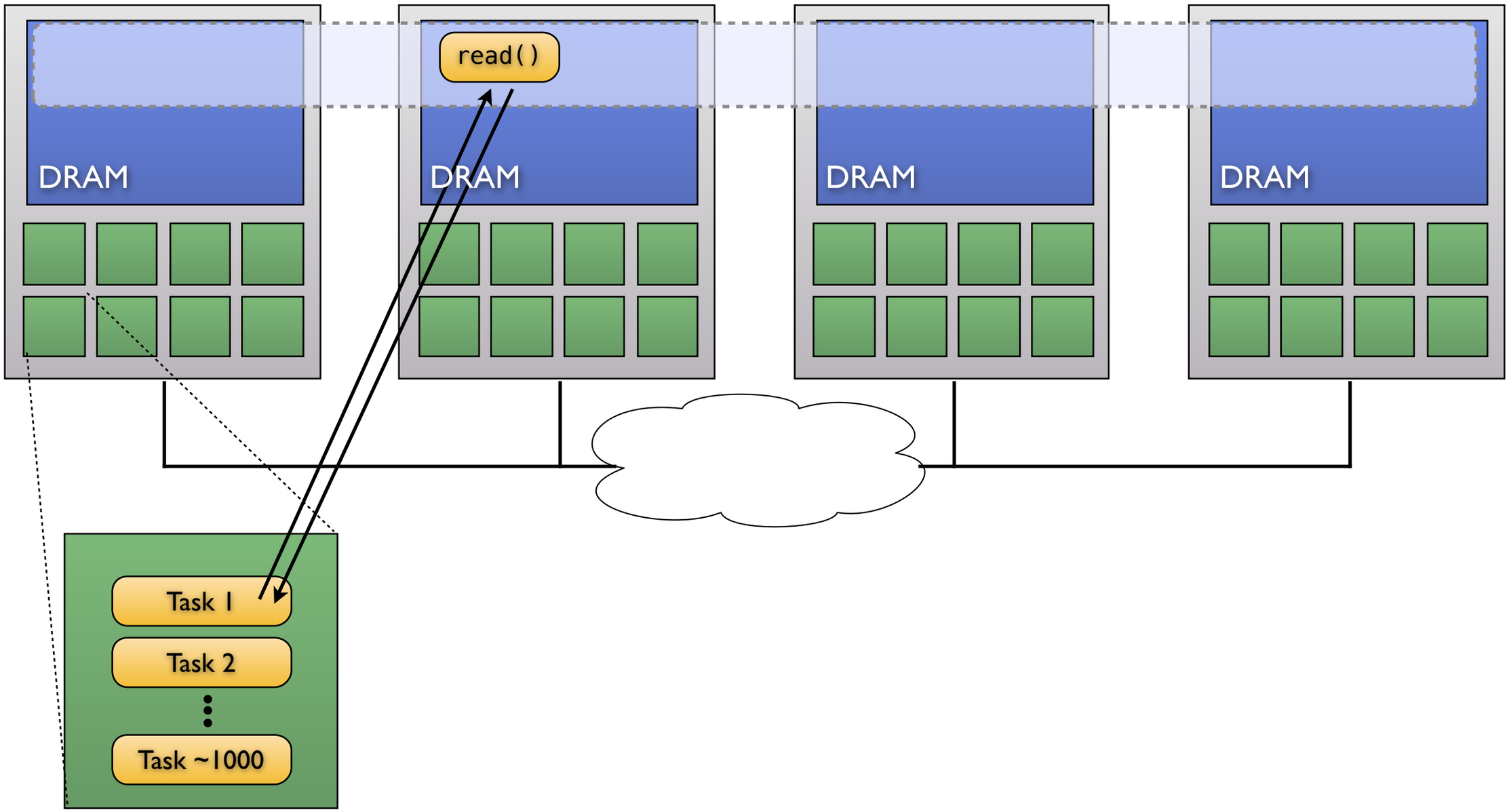


Each word of memory has a designated *home core*
All accesses to that word run on that core

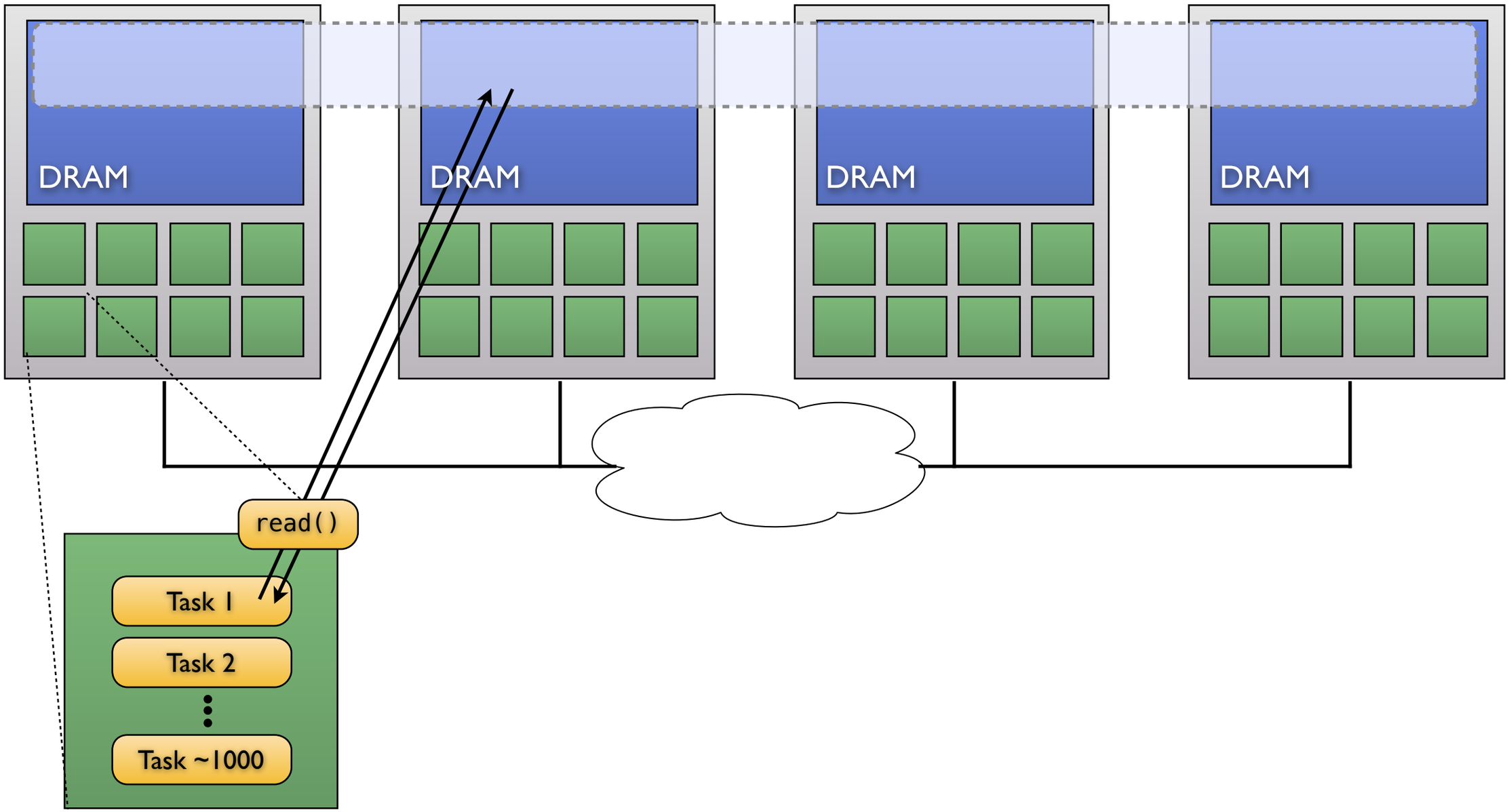
Main idea: tolerate latency with other work



Main idea: tolerate latency with other work



Main idea: tolerate latency with other work



Outline

- Motivation
- Programming Grappa
- Key components
- Performance
- Other projects

Pat's problem: traverse an unbalanced tree

- Tree is embedded in a graph

- ~1T edges in graph
  ~1B edges in tree

- Power law, low diameter

A single node, serial starting point

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

A single node, serial starting point

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

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

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

A single node, serial starting point

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

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

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

int main( int argc, char * argv[] ) {
    Vertex * root = create_tree();
    search(root);
    return 0;
}
```

The standard boilerplate (not quite right)

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

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

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

int main( int argc, char * argv[] ) {
    Grappa::init( &argc, &argv );

    Vertex * root = create_tree();
    search(root);

    Grappa::finalize();
    return 0;
}
```

Back to serial in Grappa's global view

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

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

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

int main( int argc, char * argv[] ) {
    Grappa::init( &argc, &argv );
    Grappa::run( []{
        Vertex * root = create_tree();
        search(root);
    });
    Grappa::finalize();
    return 0;
}
```

Back to serial in Grappa's global view

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

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

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

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

Addressing global memory

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

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

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

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

Accessing global memory

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

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

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

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

Global memory with delegates

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

void search(GlobalAddress<Vertex> vertex_addr) {
    Vertex v = delegate::call(vertex_addr, [=]{return *vertex_addr});

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

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

Global memory with delegates

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

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

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

Exposing some parallelism

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

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

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

Exposing more parallelism

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

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

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

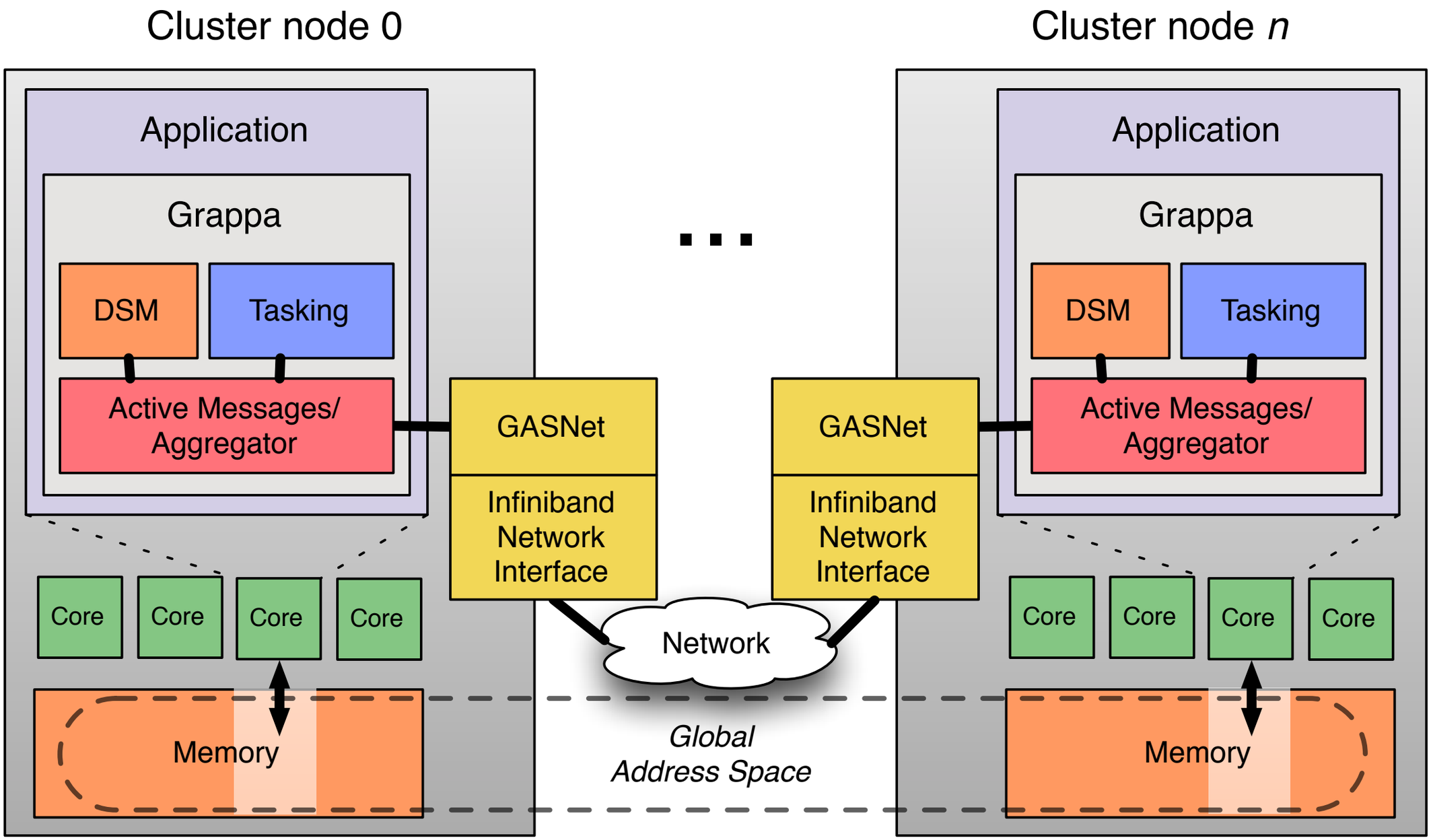
Delegation is more than just RDMA

```
struct Vertex {
    index_t id;
    GlobalAddress<Vertex> first_child;
    size_t num_children;
    color_t color;
};

GlobalAddress<int> color_counts = global_alloc<int>(NUM_COLORS);

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

    color_t c = v.color;
    bool done = delegate::call( color_counts + c, [c](int & count) {
        if( count == MAX ) { return true; }
        else { count++; return false; }
    });

    if( done ) return;

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

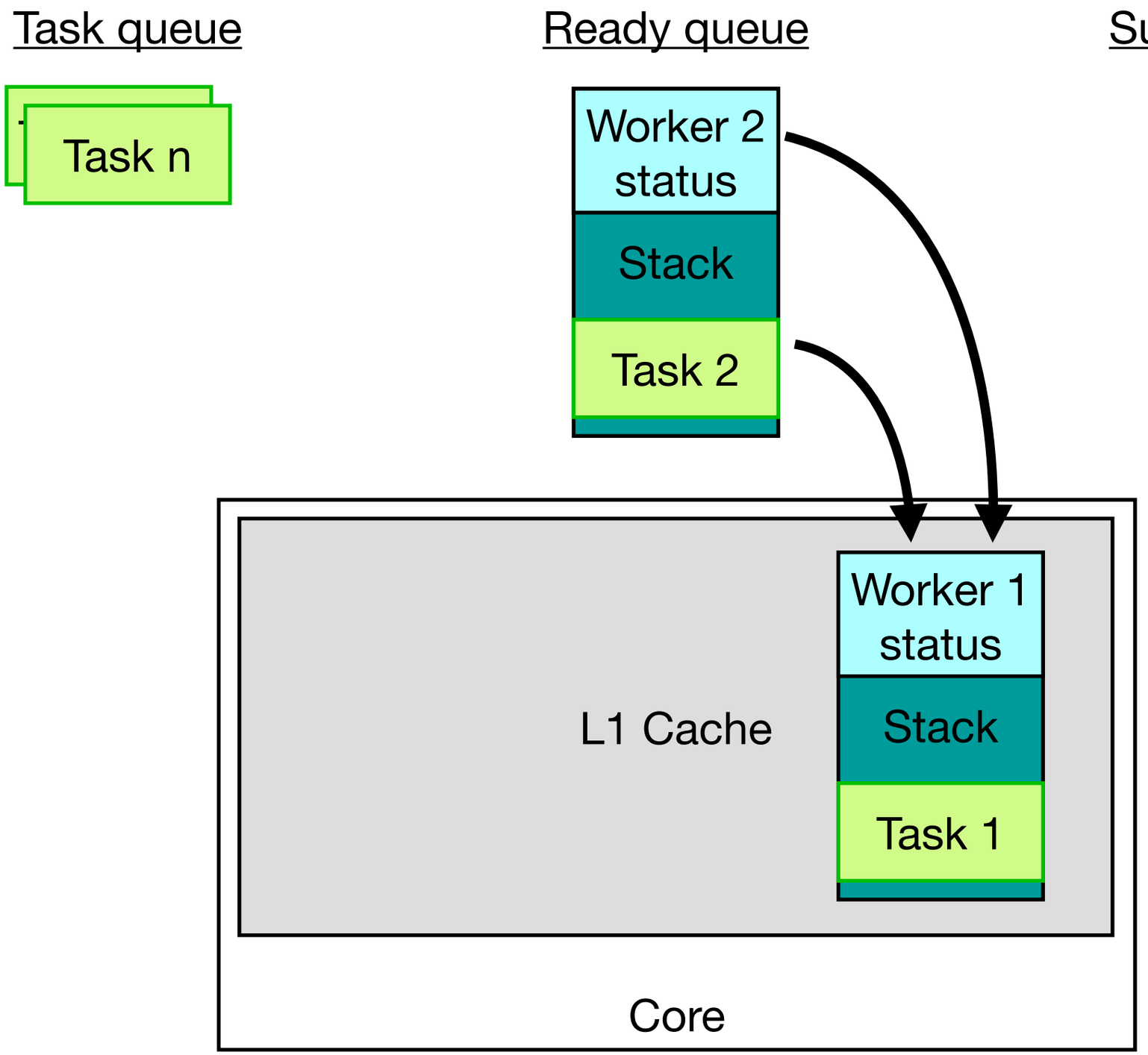
Outline

- Motivation
- Programming Grappa
- Key components
- Performance
- Other projects

Grappa design



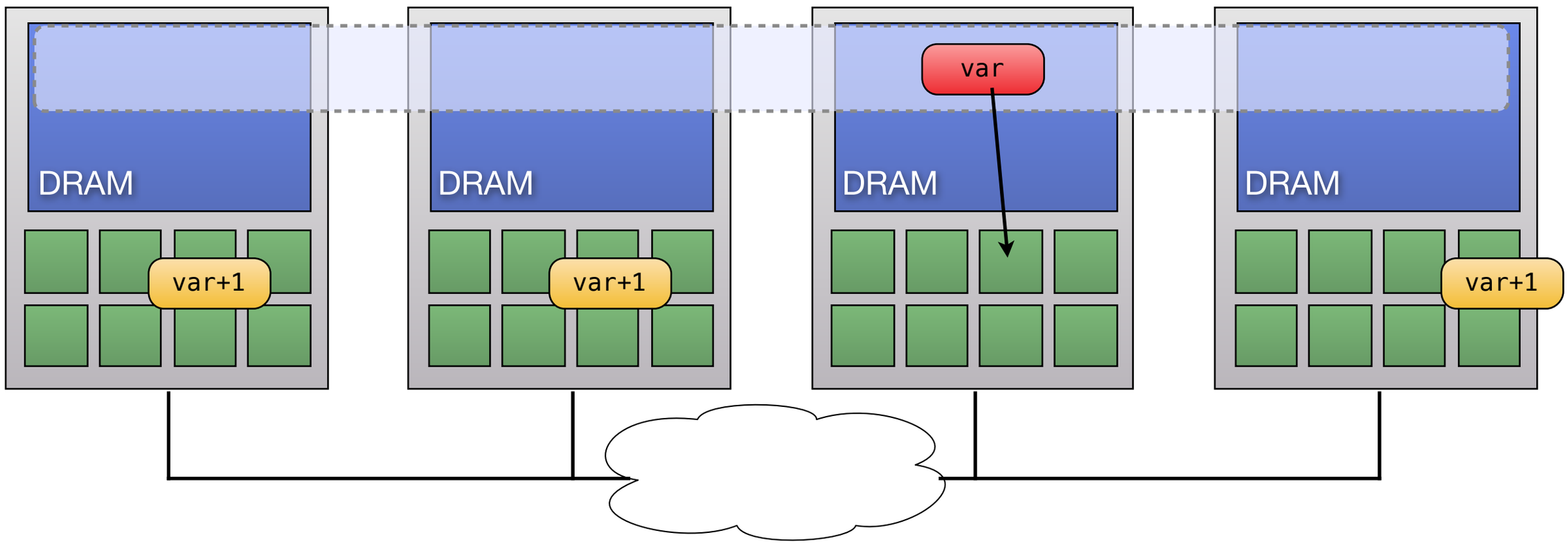
User level context switching



1 cacheline of status,
3 cachelines of stack

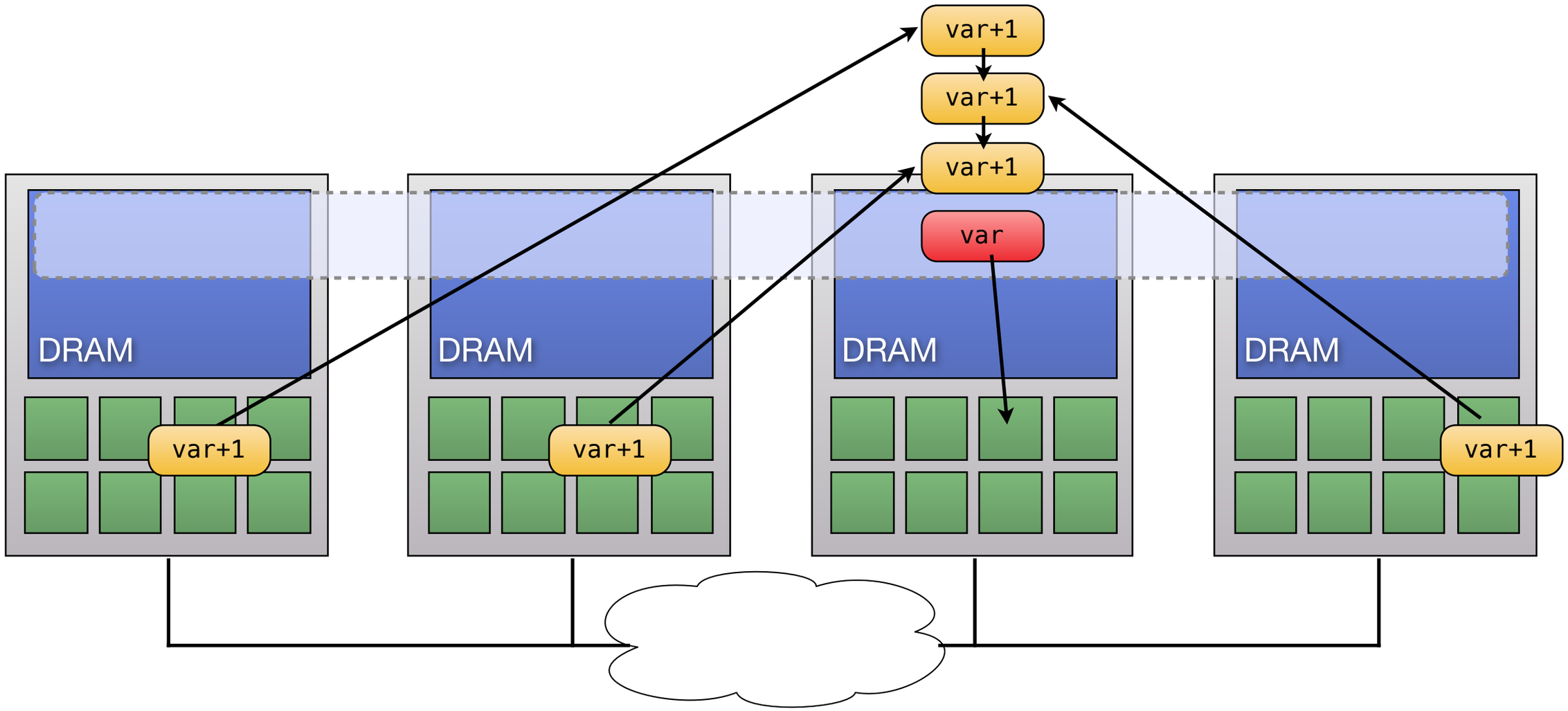
Main innovation:
We *keep state small*
and *prefetch*
to cover DRAM latency

Accessing memory through delegates



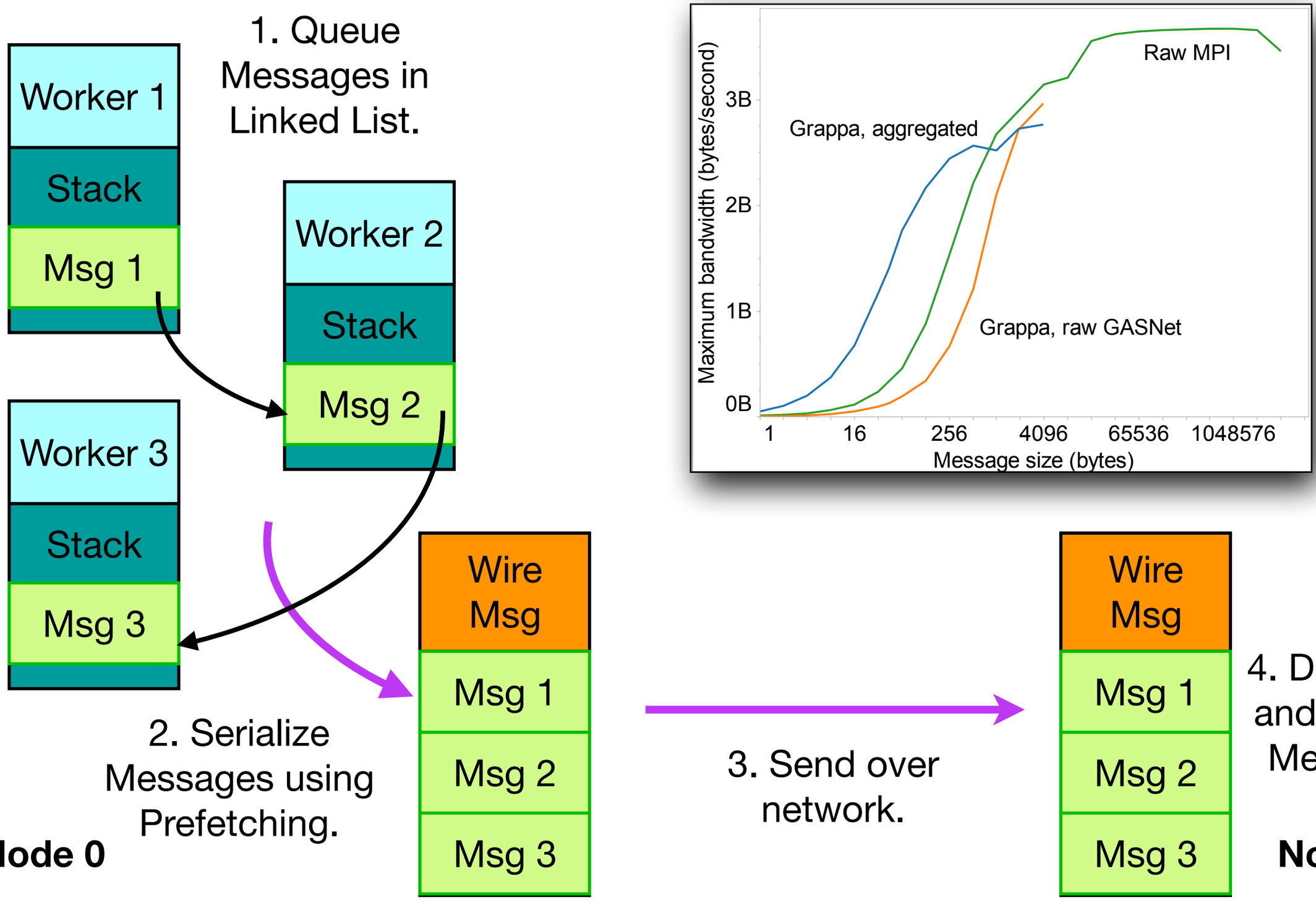
Each word of memory has a designated *home core*
All accesses to that word run on that core
Requestor blocks until complete

Accessing memory through delegates



Since var is private to home core,
updates can be applied

Mitigating low injection rate with aggregation



Outline

- Motivation
- Programming Grappa
- Key components
- Performance
- Other projects

A snapshot of current performance

- Current implementation: 15K lines
  Usable, but many optimizations still outstanding

- Three questions:
  Do Grappa's components work individually?
  Do Grappa's components work together?
  How do we compare with other systems?

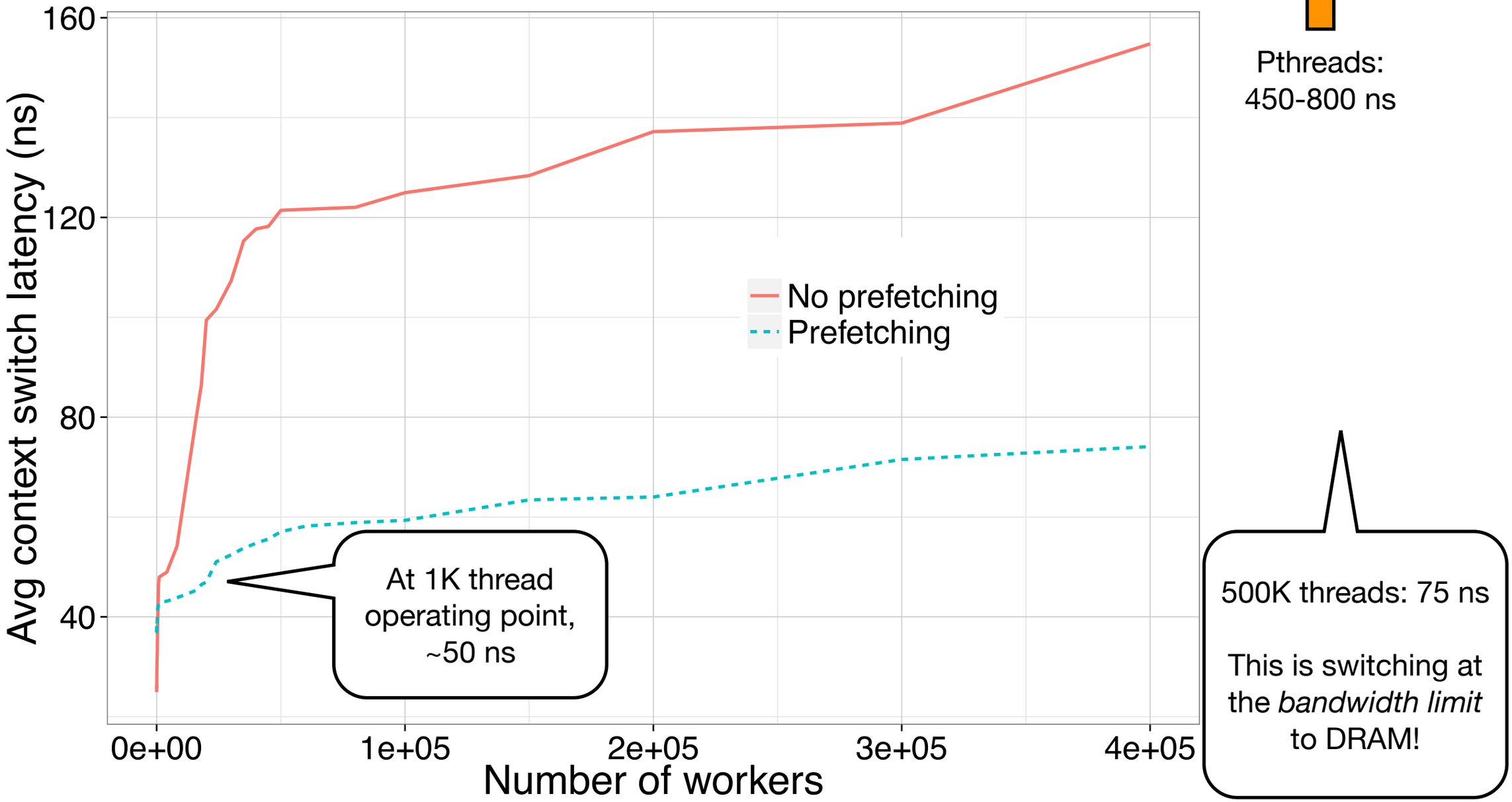
- Ran on AMD Interlagos cluster; 32 2.1GHz cores, 64GB, 40Gb Infiniband
  Compared with 128-processor Cray XMT1 (500MHz, 128 streams each)

Measuring context switch performance

- Simple: N tasks yield in a loop on a single core
  We vary N to see how context switch time changes

- Context switching is isolated here: the system is doing nothing else

Context switching is fast



Pthreads:
450-800 ns

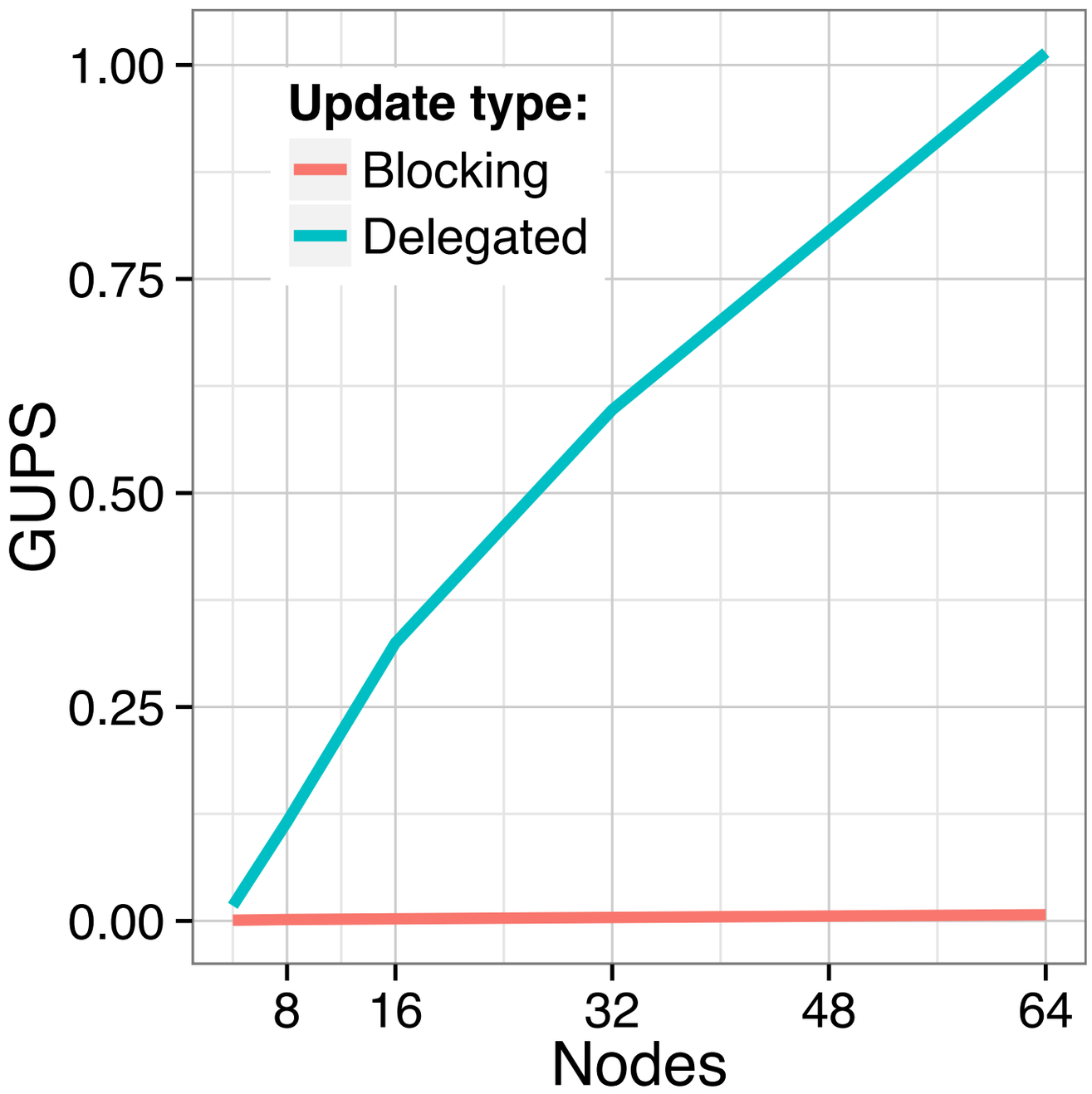
Measuring random access bandwidth

- Giga updates per second (GUPs) benchmark
  measures cluster-wide random access bandwidth

- Only one task per core sends messages, so aggregator is essentially isolated

Random access BW is good

Minimal context switching



Theoretical peak at 64 nodes is 6.4 GUPs

Unbalanced tree search in memory

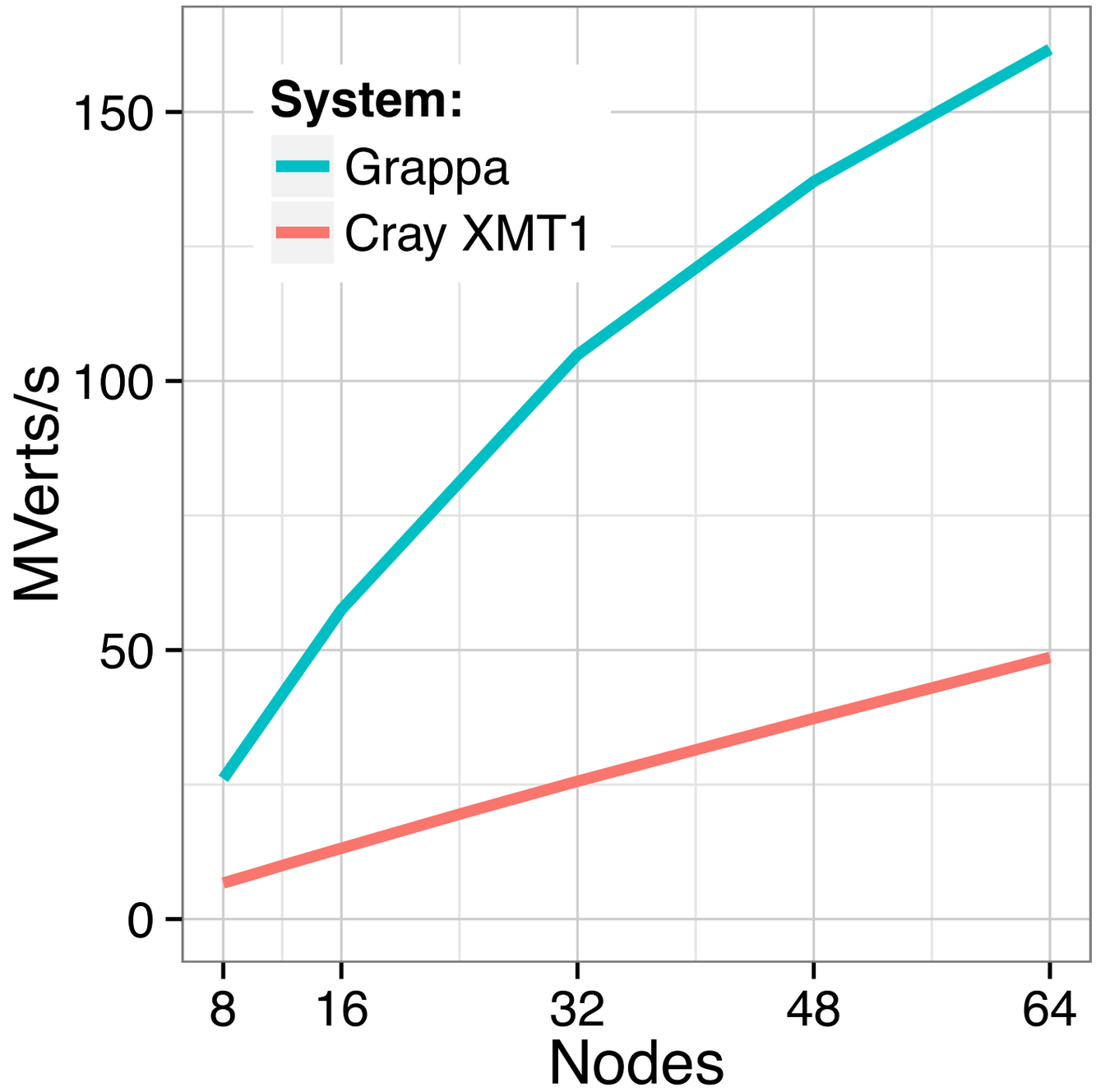
- Original UTS benchmark designed to exercise work stealing
  We modified it to include memory access as well

- Creates unbalanced tree in memory and times traversal

- Visiting a vertex requires remote access, so we are context switching, aggregating delegate messages, and work stealing all at the same time

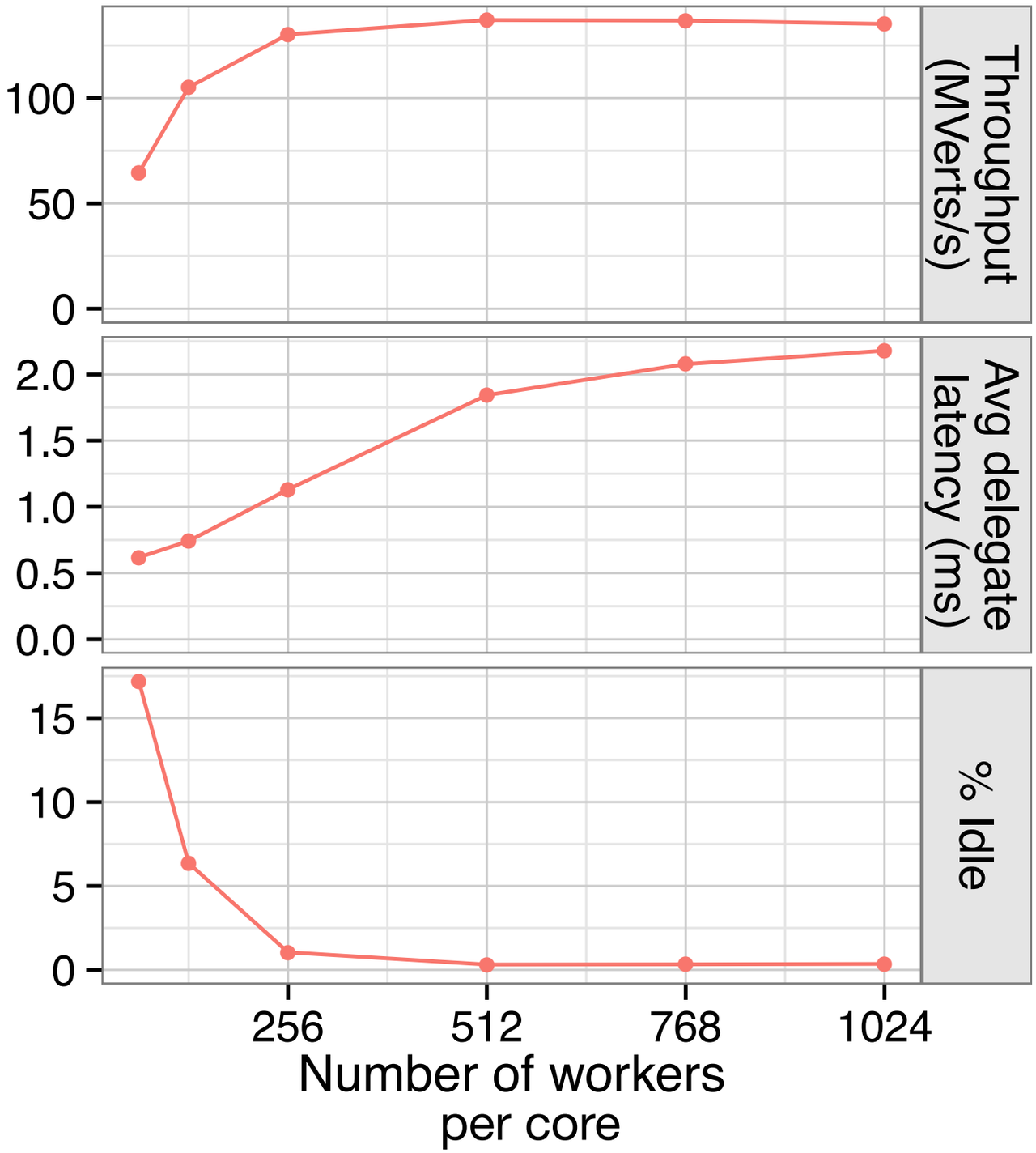
- Metric is vertex throughput

Grappa is able to exploit parallelism in UTS



T1XL tree

Grappa tolerates latency in UTS



With 512 active tasks per core, throughput peaks and idle time is practically zero.

(48 nodes, T1XL tree)

Comparing application performance

- Additional benchmarks:
  Breadth-first search: Simple version of Graph500 benchmark
  Integer sort: NAS Parallel Benchmark bucket/counting sort
  PageRank: Google's web graph centrality metric

- For these three, MPI and XMT currently beat Grappa, but
  - *Price/performance* is still better than XMT
  - Grappa code is *shorter and simpler* than MPI
- There is a cost to Grappa's generality
  (but we're working on reducing that cost!)

Comparing GUPS performance

	Grappa	XMT	MPI
GUPS	1	2.23	0.11
UTS (T1)			
BFS			
IntSort			
Pagerank			

Performance normalized to Grappa, 64 nodes

- XMT version: basic GUPS with hardware fetch-and-add
- MPI version: HPCC RandomAccess
- Contains specialized implementation of aggregation, but
  - not optimized for out-of-cache
  - limited support for concurrent communication

Comparing UTS-in-memory performance

	Grappa	XMT	MPI
GUPS	1	2.23	0.11
UTS (T1)	1	0.38	
BFS			
IntSort			
Pagerank			

Performance normalized to Grappa, 64 nodes

- XMT implementation uses fine-grained synchronization at each vertex, which is unnecessary
- Difficult to avoid; baked into compiler, OS, hardware, etc.
- Grappa supports this too, but also allows coarse-grained synchronization
- Grappa also takes advantage of locality in spawns and edge lists

Comparing BFS performance

	Grappa	XMT	MPI
GUPS	1	2.23	0.11
UTS (T1)	1	0.38	
BFS	1	1.63	3.52
IntSort			
Pagerank			

Performance normalized to Grappa, 64 nodes

- BFS_Simple from Graph500 (no Beamer-like optimizations)
- MPI version includes specialized implementation of aggregation, moving only two 8-byte vertex IDs
- Grappa runs additional code, requires more space to support general aggregation
  - BFS messages include additional 16 bytes of deserialization/synchronization information

Comparing IntSort performance

	Grappa	XMT	MPI
GUPS	1	2.23	0.11
UTS (T1)	1	0.38	
BFS	1	1.63	3.52
IntSort	1	3.59	5.36
Pagerank			

Performance normalized to Grappa, 64 nodes

- NAS Parallel Benchmarks, class D
- Grappa writes keys directly to destination with delegates
- MPI version implements specialized aggregation with local sort plus collective communication with Alltoallv
  - Can we implement aggregation with collectives in Grappa?

Comparing Pagerank performance

	Grappa	XMT	MPI
GUPS	1	2.23	0.11
UTS (T1)	1	0.38	
BFS	1	1.63	3.52
IntSort	1	3.59	5.36
Pagerank	1	4.35	4.87

Performance normalized to Grappa, 64 nodes

- Grappa version is a straightforward nested loop
- MPI version uses optimized Trilinos sparse matrix library

Where are we?

- Fundamentals are strong:
  Context switching is fast
  Aggregation is fast
  UTS composes them and gets good performance

- There is currently a cost to our generality

- MPI is mostly beating us for now,
  often replicating what we're doing in a specialized way,
  as well as using tricks we may be able to take advantage of

- We are working on our next-generation networking layer now

Outline

- Motivation
- Programming Grappa
- Key components
- Performance
- Other projects

Compiler support: automatic delegates

For where your data is, there your code will be also.

Best performance comes from executing task where the data is

- **delegate operations** execute some small region of code atomically on the core that owns the memory
- generic delegate::call() executes the enclosed region of the task (expressed in a lambda) remotely

Delegated regions can be inferred automatically

- **inspect** uses of Grappa global pointers, find regions that use global pointers on a particular core, and **extract** into a delegate
- Implementing with a custom LLVM pass

Additional ideas

- **pass continuation** to allow delegates to hop between multiple cores before returning to the original caller
- **specialize** multiple versions of delegate regions and select the best one dynamically based on runtime values

```
int main(int argc, char* argv[]) {
  init(&argc, &argv);
  run([]{

    long global* A = global_alloc<long>(Asize);
    long global* B = global_alloc<long>(Bsize);

    forall(B, Bsize, [=](double& b){
      delegate::call((A+b).core(), [A,b]{
        long * Ab = (A+b).pointer();
        (*Ab) %= b;
      });
    });

    forall(B, Bsize, [=](double& b){
      A[b] %= b;
    });

  });
  finalize();
}
```

Manual delegate

Automatic delegate

"Schrödinger" Consistency

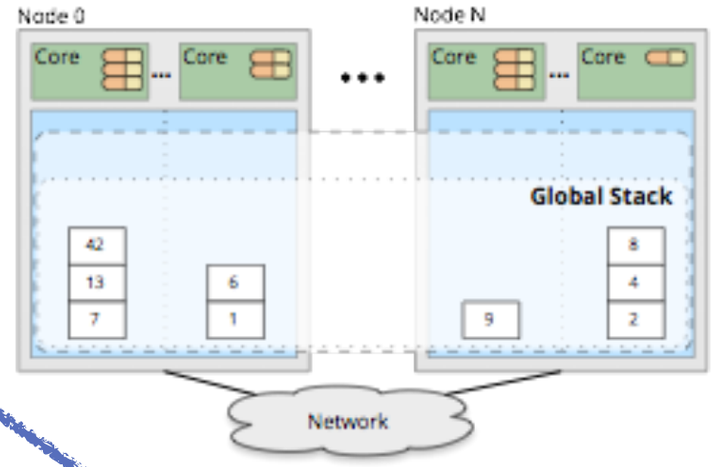
until observed, operations can be committed **and** not

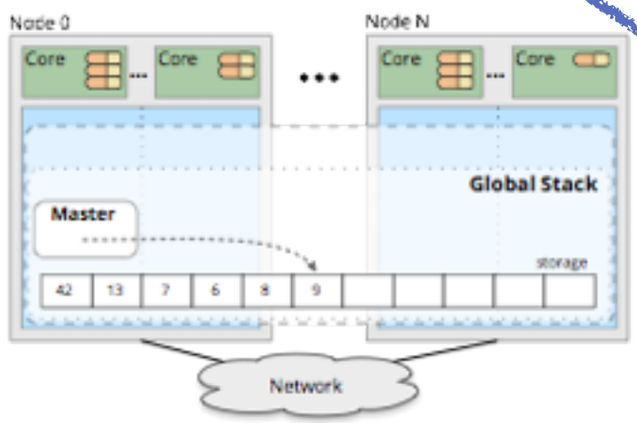
Delay synchronization as long as possible

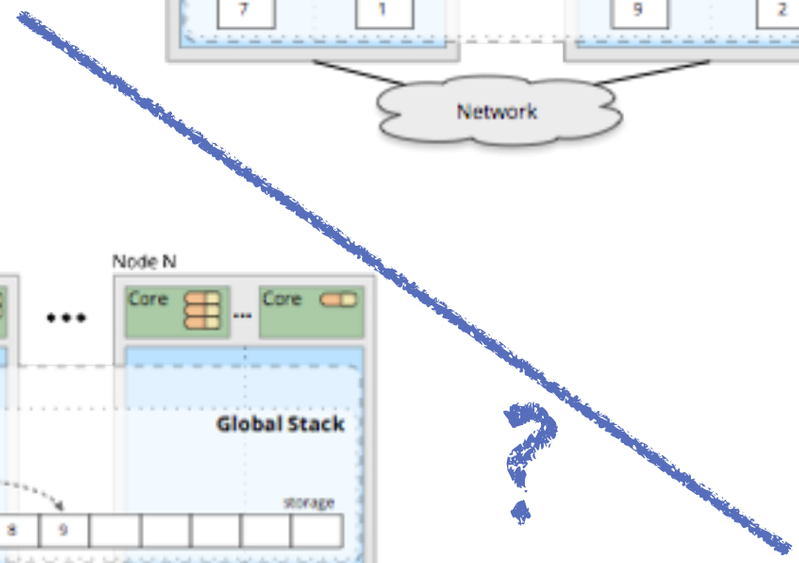
- commit when operation would be able to **observe** order
- example: *pushes* kept local, *pops* search for an available *push*

Abstract data structure semantics

- express how operations affect and observe abstract state
- **abstract locks** allow commutative ops to proceed in parallel, and block conflicting ops to run late
- **inverse** operations **annihilate** locally, needing no synchronization
- Similar in spirit to "Transactional Boosting"







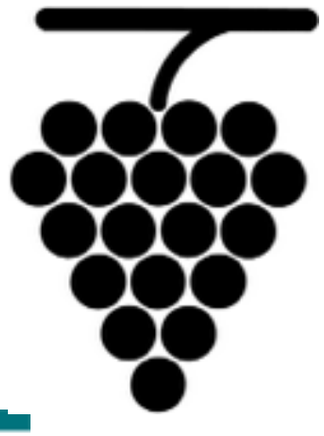
Maurice Herlihy & Eric Koskinen. PPoPP 2008.
Transactional Boosting: A Methodology for Highly-Concurrent Transactional Objects.

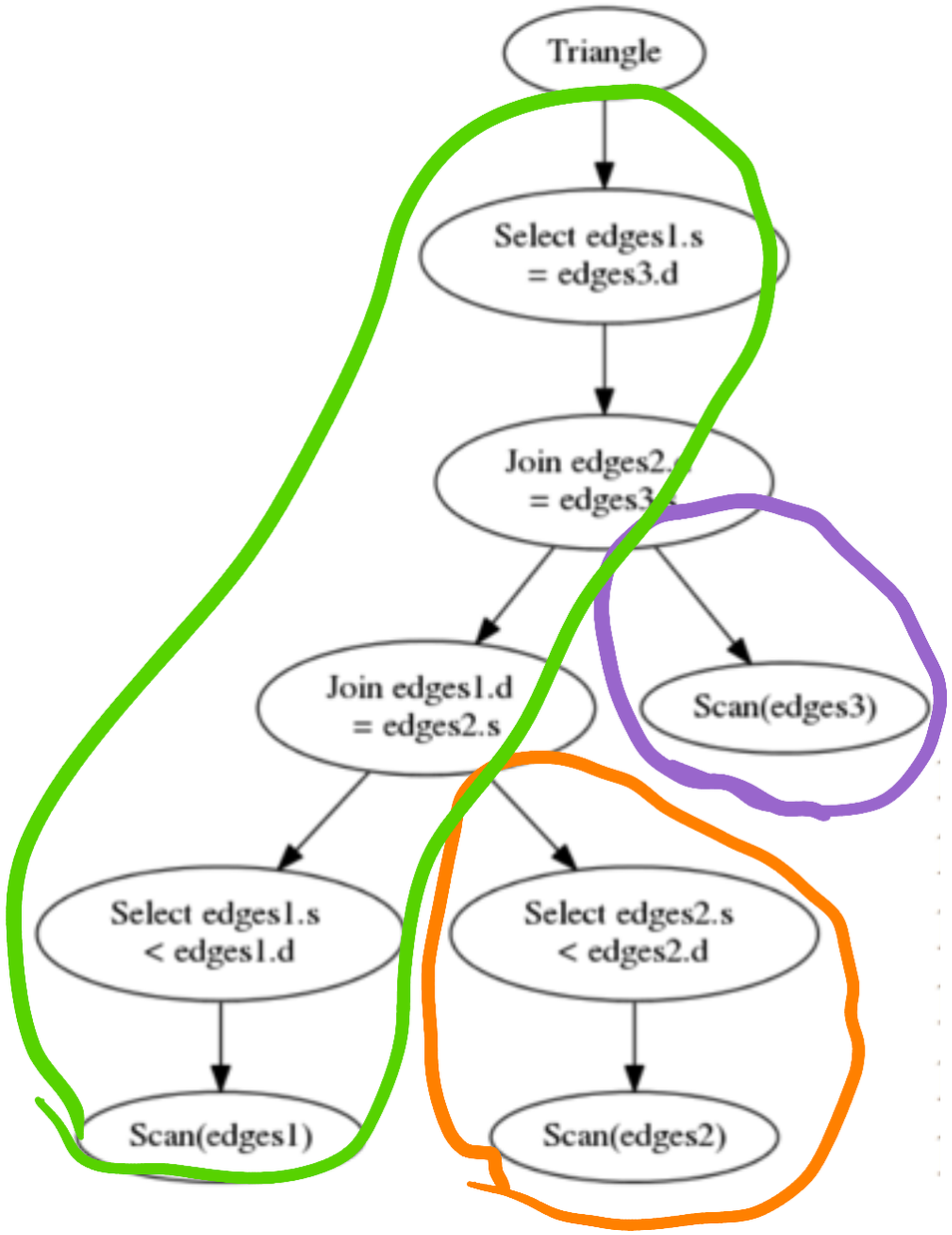
Compiling queries for parallel shared memory platforms

```
Q1(yr) :- R( journal, “type”, “Journal” ),
          R( journal, “title”, “Nature” ),
           R( journal, “issued”, yr )
```


parallel shared memory code

Grappa





Conclusion

- **Extreme latency tolerance** helps us scale irregular applications

- Grappa's runtime system provides
  - a task library
  - a distributed shared memory system
  - a network aggregator

- Context switches are fast, even with many threads
  Random access bandwidth is good
  Aggregation is effective

Questions?