# Grappa:

## A latency tolerant runtime for large-scale irregular applications

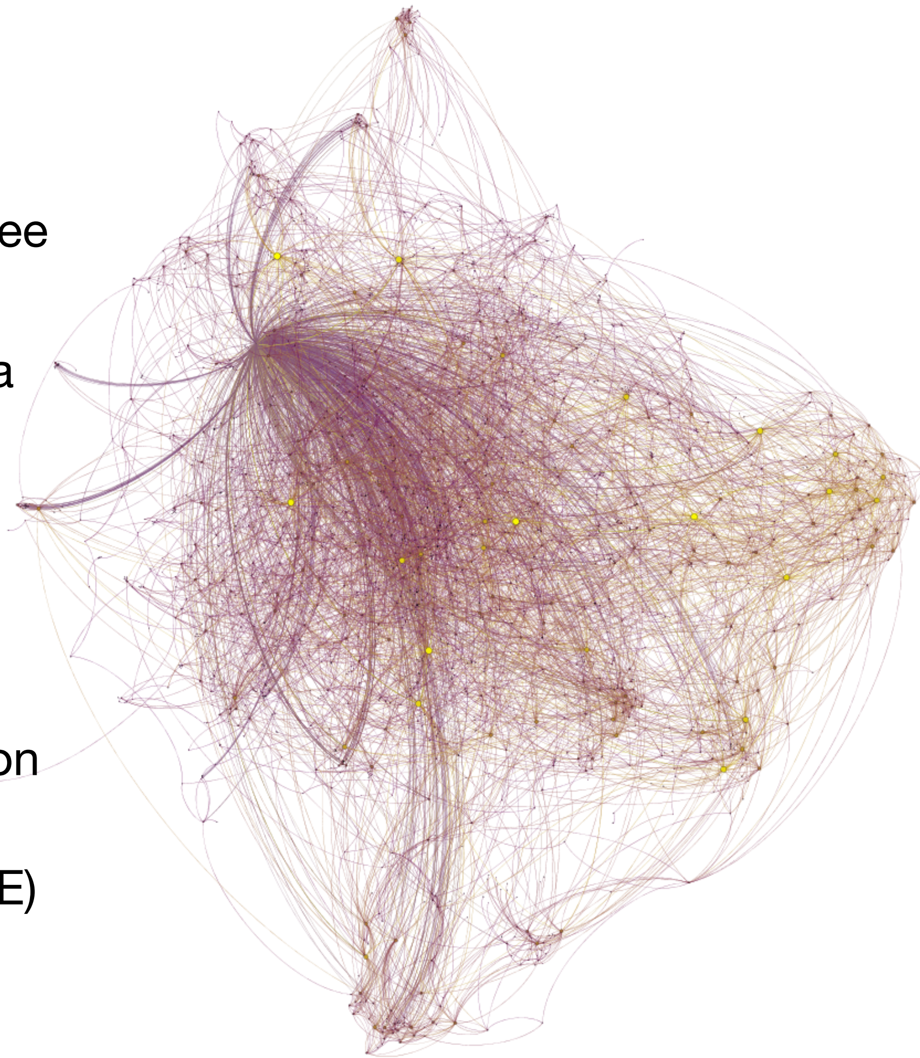
Jacob Nelson, Brandon Holt, Brandon Myers,
Preston Briggs, Luis Ceze, **Simon Kahan**, Mark Oskin

Computer Science & Engineering, University of Washington

April 13, 2014

# We want to solve big ugly problems easily and efficiently on rack scale systems (and beyond)

- Abstract example:
    - TB+ sized directed imbalanced tree
    - all memory-resident
    - traverse vertices reachable from a given start vertex
- Other more useful examples:
    - finding ephemeral patterns in streaming graph data (fraud detection)
    - branch-and-bound for optimization (routing delivery vehicles)
    - direct sparse linear solvers (SPICE)



# 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;
    parallel_for( 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;
}
```

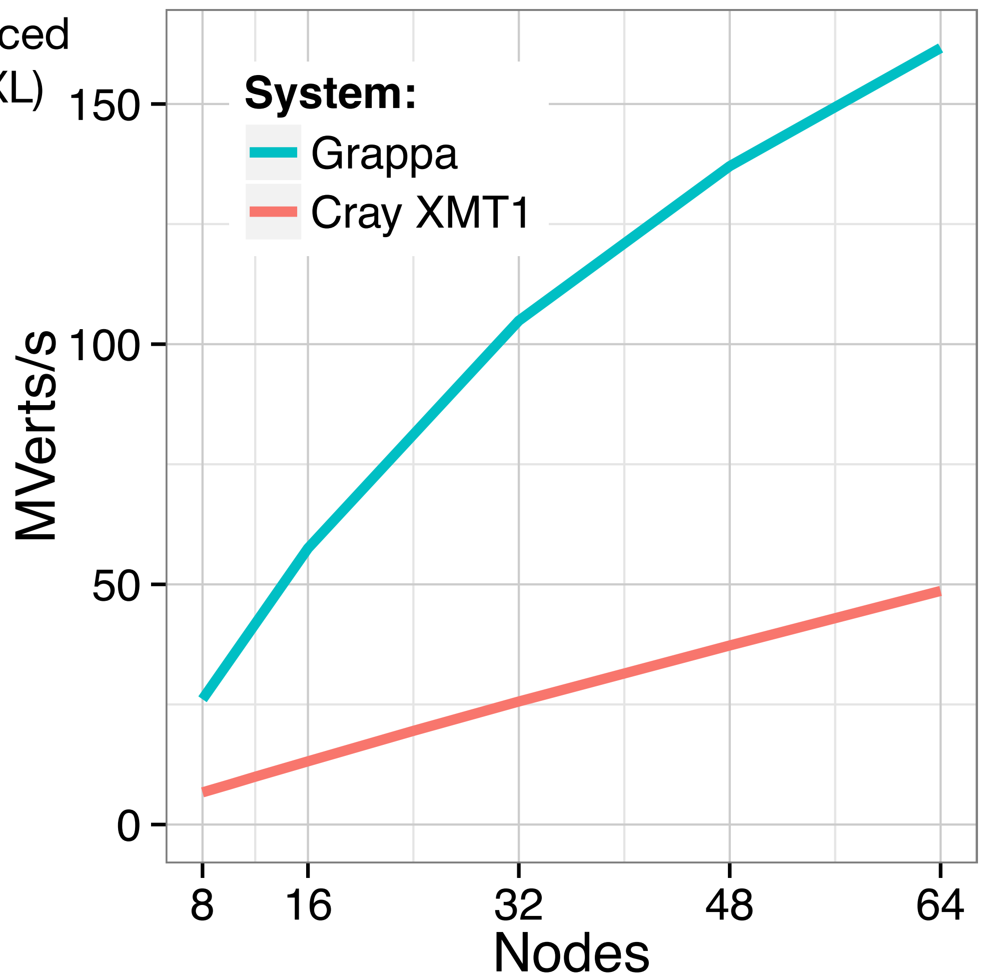
# That’s it! Grappa code for a rack scale system!

```
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;
    parallel_for( 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;
}
```

Straightforward to write, but does it work?

# Comparison against special purpose hardware

Traversing a 1.6B vertex imbalanced tree (UTS T1XL)




- **64-node AMD Interlagos cluster; 32 2.1GHz cores & 64GB RAM per node; 40Gb Infiniband**
- **128-node Cray XMT1; 1 500MHz core & 4GB per node; CrayXT Seastar interconnect**

Grappa works: we can scale up a big ugly problem easily and efficiently.

But what about Grappa is relevant to rack scale computing when solving big ugly problems?

# At rack+ system scale, chaos is required

- Even easy problems that seem to divide up evenly, become irregular at scale.
  - processor interruptions
  - system asymmetries

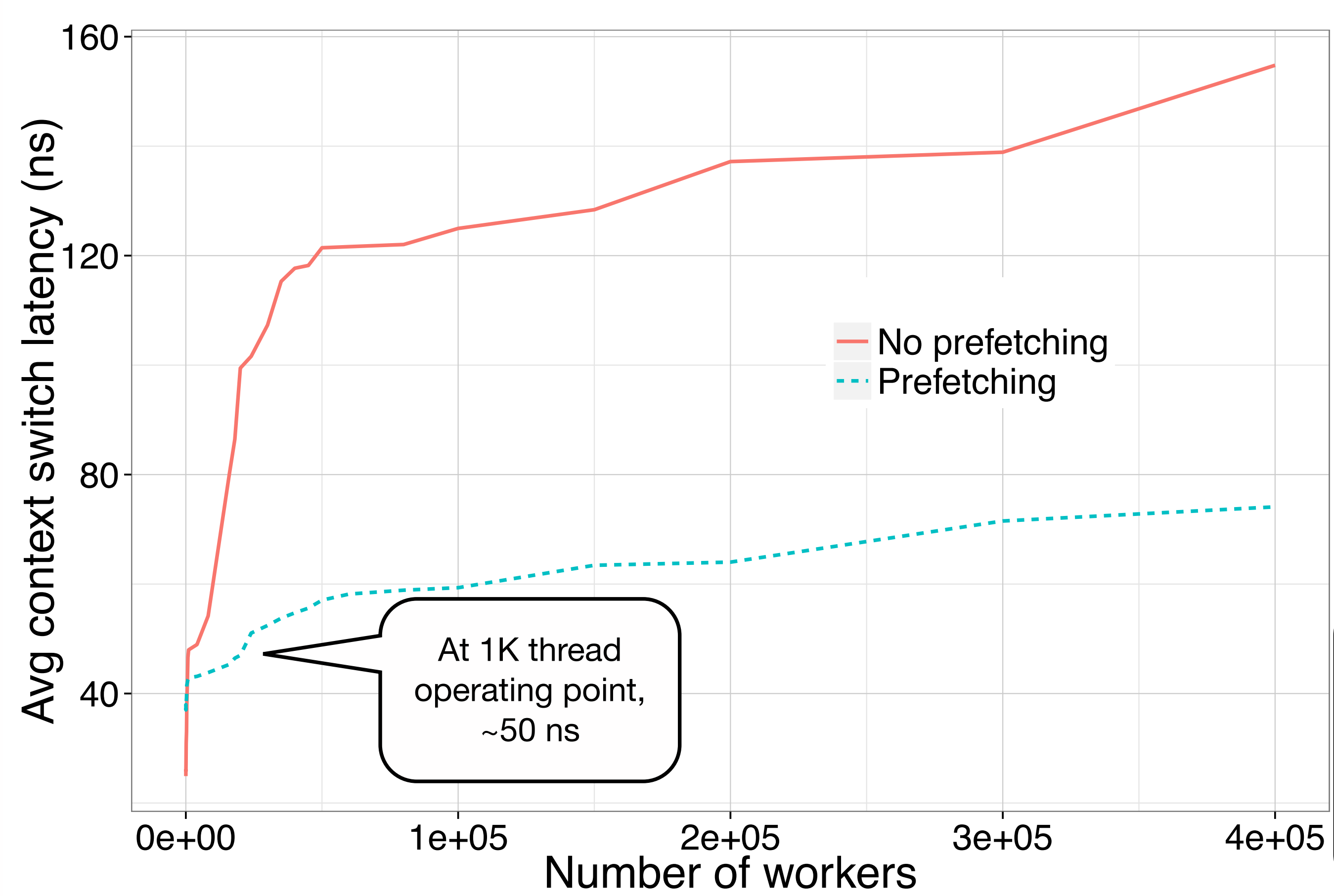
- Scaling our irregular problems demand over-decomposition and dynamic work redistribution => asynchrony
- **Chaotic, asynchronous parallelism is required to get efficient use from rack scale (or larger) systems** when applying many processors on a single large problem.

# Yet, at component scale, order is required

- Hardware components designed for order and structure:
  - Caches
    - efficient when **references are grouped or repeated**
  - Prefetching
    - efficient when **access is predictable**
  - Pipelines
    - efficient when there are **few computational dependences**
  - Network interfaces
    - efficient when **messages are infrequent and large (>4KB)**
  - Atomics, fences
    - efficient when not used ( ie, when **operations do not induce races** )
- **Ordered parallelism is required for efficiency from individual components**

Grappa addresses this dilemma by using parallel slack and latency tolerance

# Software prefetching of contexts

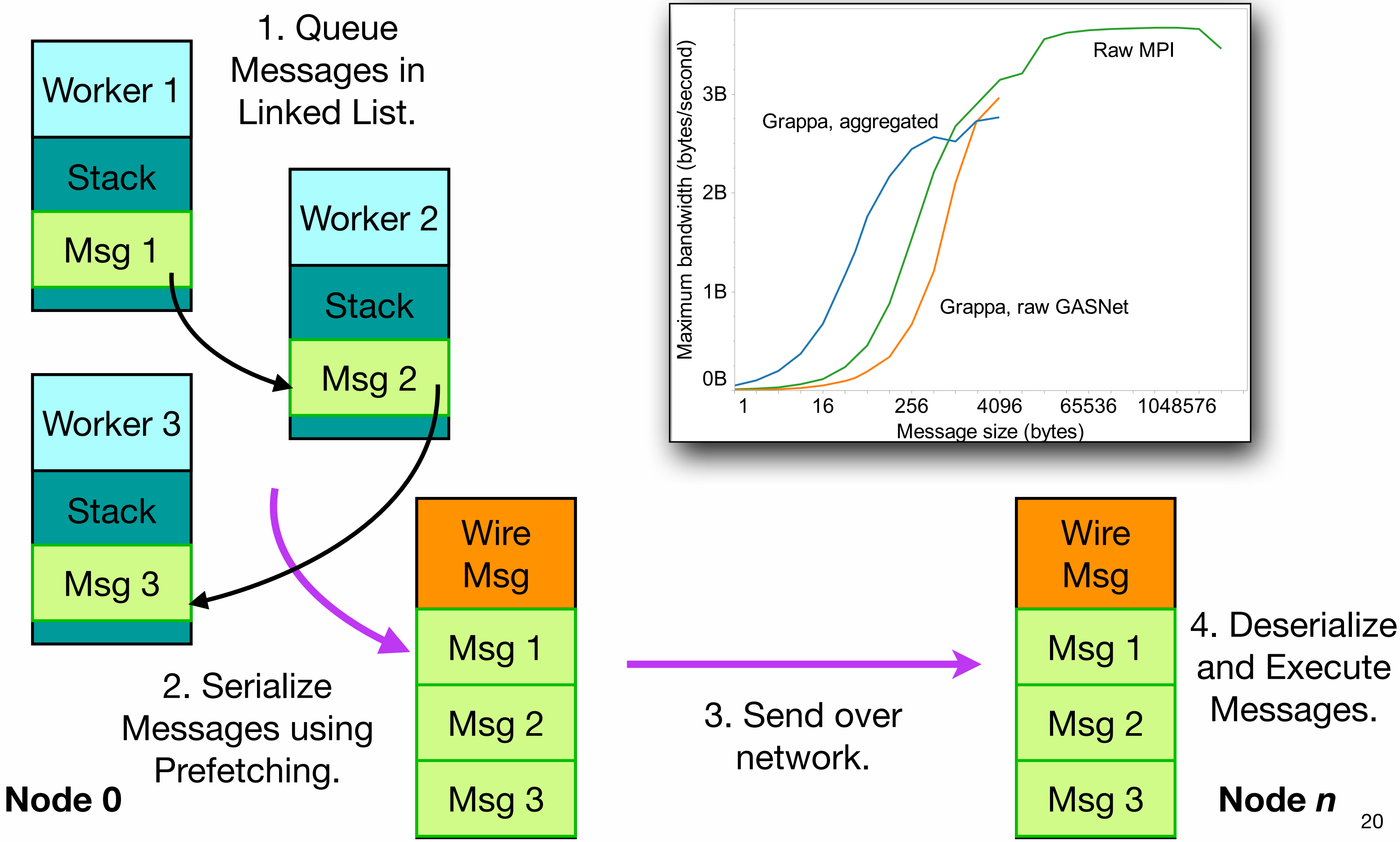



Pthreads:
450-800 ns

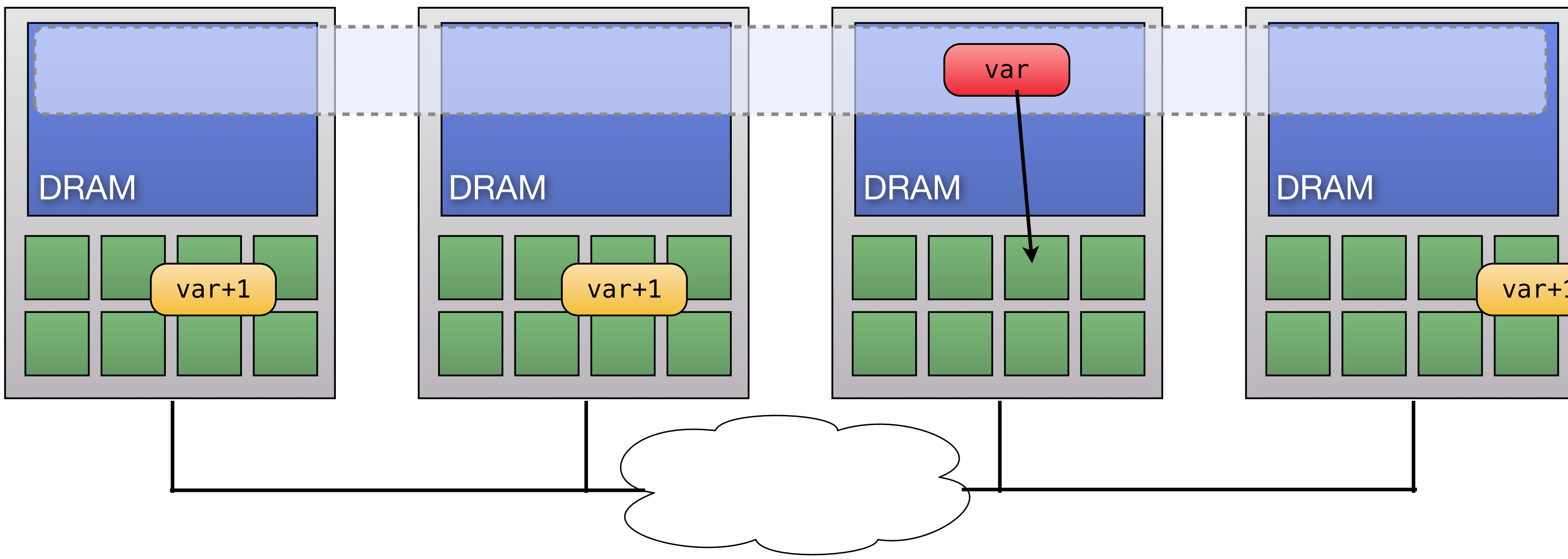
500K threads: 75 ns

This is switching at the *bandwidth limit* to DRAM!

# Mitigating low injection rate with aggregation

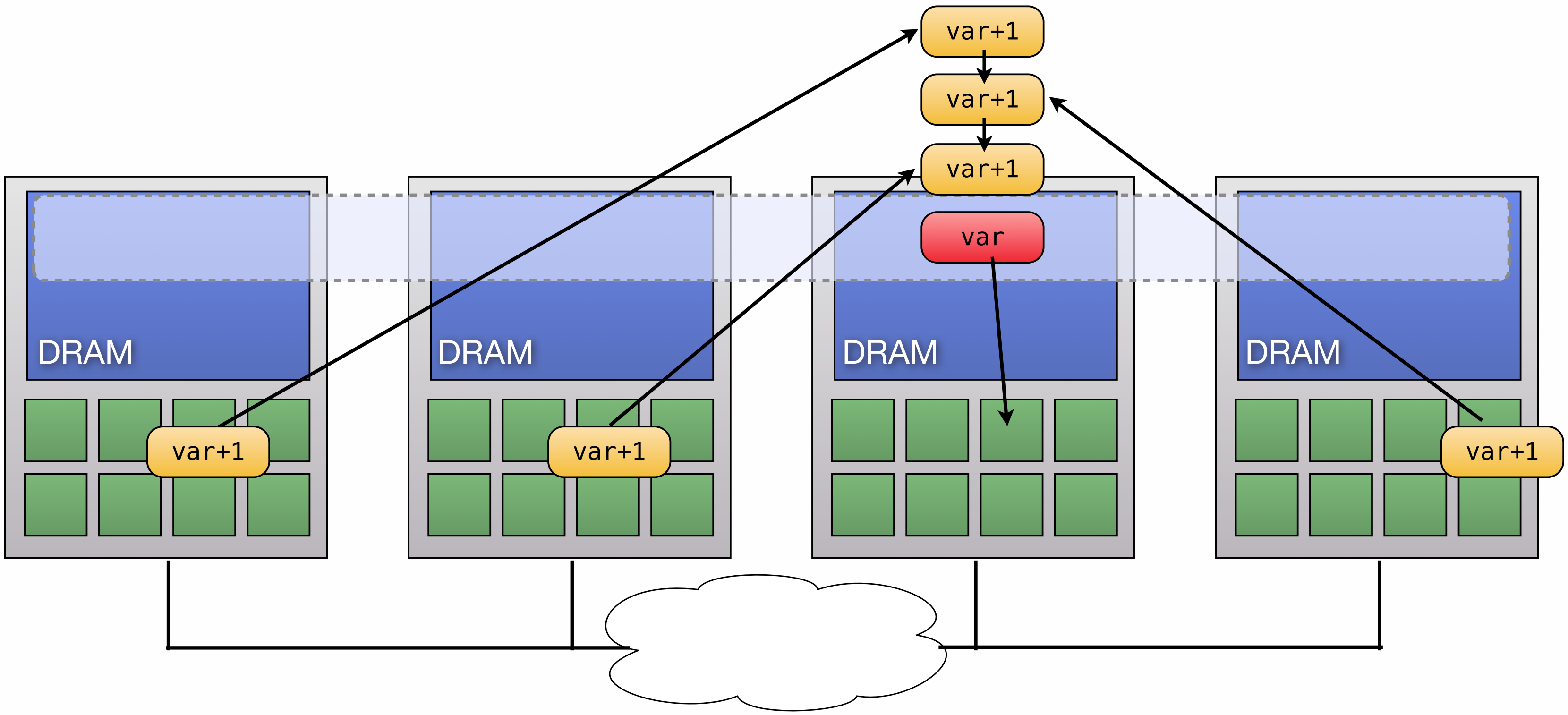


# Accessing memory through delegates




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

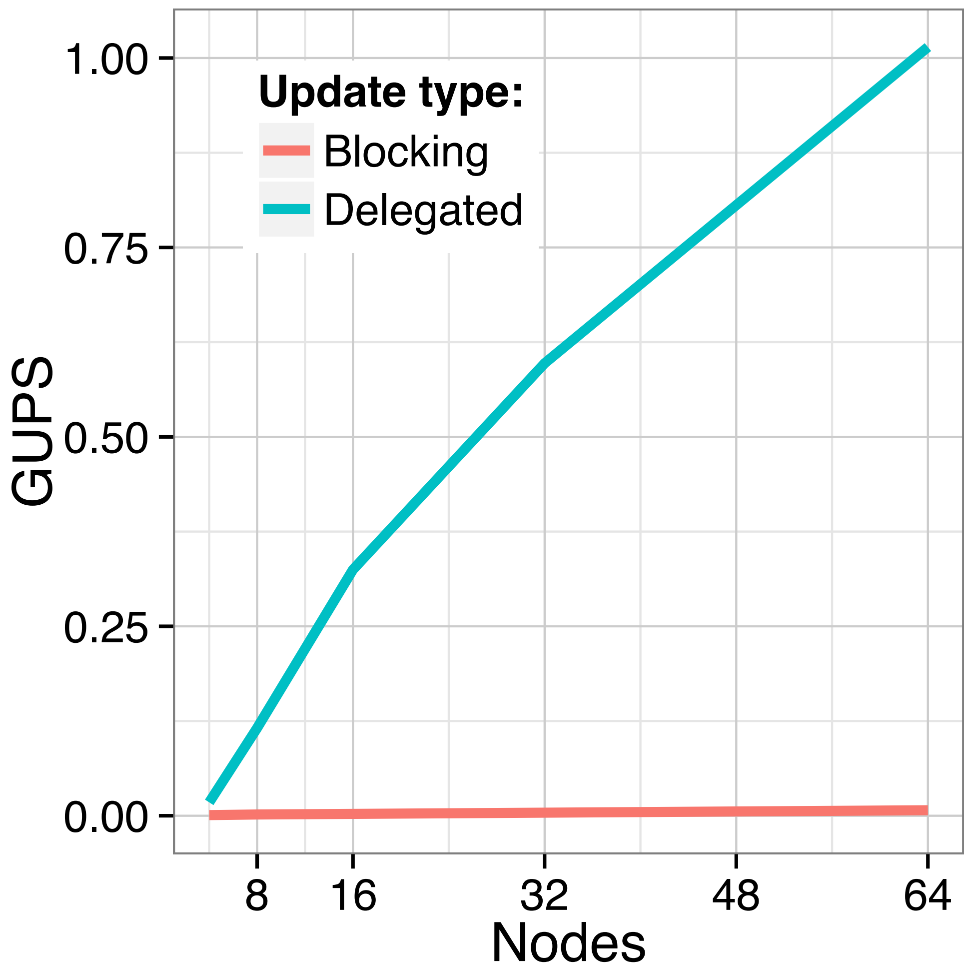
# Accessing memory through delegates




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

# Random update BW is good

Minimal context switching




Theoretical peak at 64 nodes is 6.4 GUPs, so this is work in progress.

GUPS pseudocode:

```
global int a[BIG];
int b[n];
for (i=0;i<n;i++)
  a[b[i]]++;
```

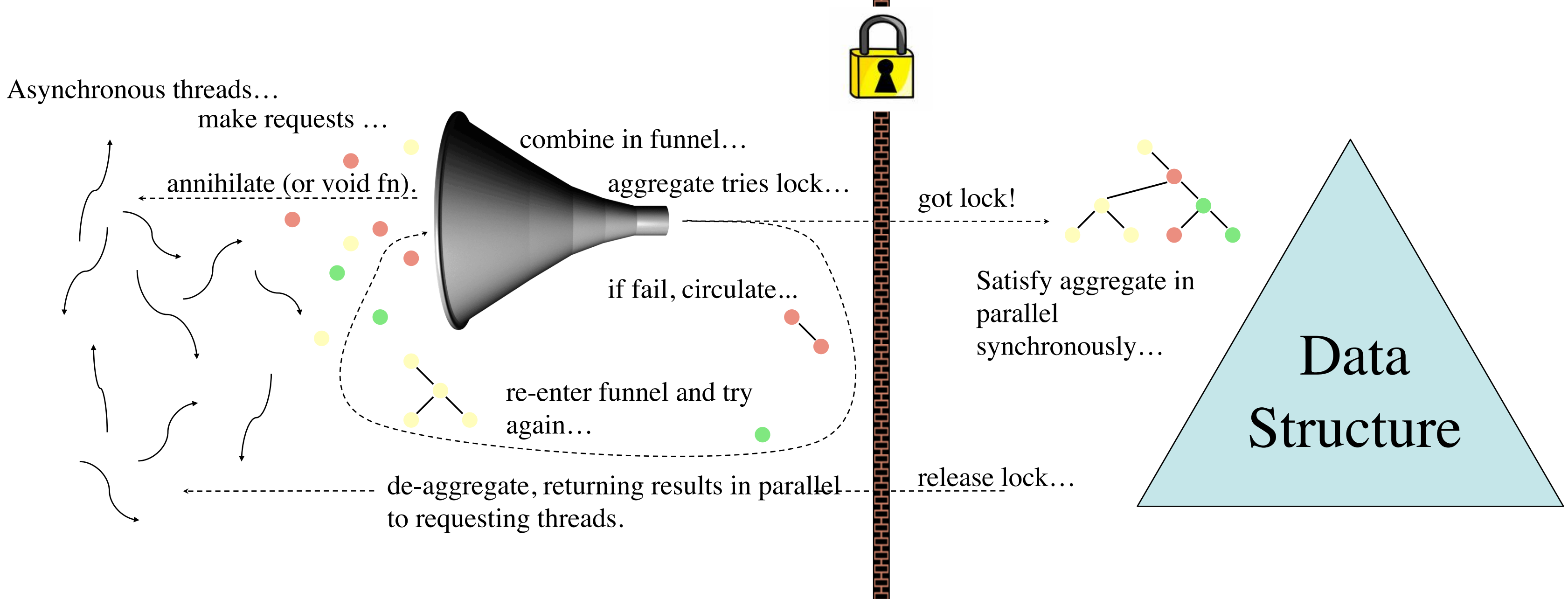
# General Combining Scheme

## Asynchronous (Competitive)

- Arbitrary # computations
- Any number of threads
- Timing of interaction arbitrary
- Chaos!

## Synchronous (Collaborative)

- Single computation
- Number of threads is explicit
- Synchronized, exclusive access to data
- Order!




Similar to “Combining Funnels: a Dynamic Approach to Software Combining”, Nir Shavit, Asaph Zemach, 1999

# Conclusion

- **Grappa allows easy expression of asynchronous parallelism**
  - providing the concurrency needed to get high system utilization on big ugly problems on rack scale computers
- **and efficient transformation into ordered parallelism**
  - by transforming the chaos to the order for which individual components are designed.
- **through use of parallel slack to tolerate latency.**

- **What this means is that we can more easily write programs to attack large ugly problems at scale.**

- **Try it!**

http://grappa.io/

# Questions?