Grappa: PGAS for irregular applications

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

Open source release coming soon!
Join our mailing list for more info:
http://sampa.cs.washington.edu/grappa

Irregular applications are important in a world full of big data

Irregular applications arise in:

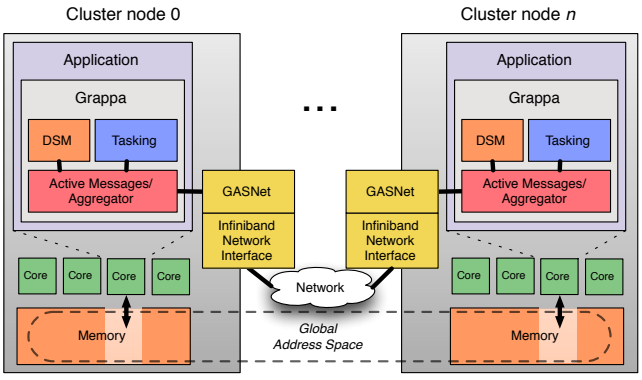
- Social network analysis
- Semantic databases
- Combinatorial optimization
- Circuit simulation
- Machine learning

Low locality and frequent random communication make these applications difficult to scale on commodity clusters.

Programming models like MapReduce and GraphLab can be restrictive; for example, parallel recursive algorithms are hard to express.

Grappa provides a familiar multithreaded programming model, making it easy to scale up irregular applications on commodity clusters.

Grappa uses *latency tolerance* to provide scalable performance on commodity clusters



Grappa's runtime system consists of three key components:

Tasking layer
Supports millions of threads across the cluster
Automatically balances load with *work stealing*

Distributed shared memory (DSM)
Provides a single system, shared memory abstraction

Communication
Supports high-throughput small message communication

See the other side of this page for details.

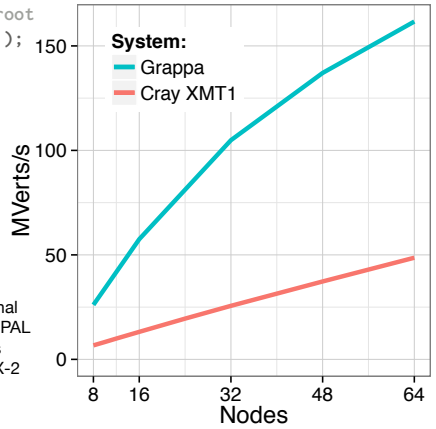
Programming Grappa

This is a simple example of doing random increments in a an array spread across the cluster. Grappa uses C++11 to make it easy to express parallelism. Grappa currently achieves 1.0 GUPS on 64 nodes. See the other side of this page for details.

```
using namespace Grappa;

void GUPS() {
  // A,B have type: GlobalAddress<long>
  auto A = global_alloc<int64_t>( N );
  auto B = global_alloc<int64_t>( N );
  random_init( B, N );

  forall( B, N, [=]( int64_t & b ){
    delegate::increment_async( A + b, 1 );
  });                          // A[B[i]]++

}
```

Below is a variant of the UTS tree search benchmark, traversing an imbalanced geometric tree in global memory. The implementation spawns tasks recursively; work stealing balances load across the cluster. Grappa performs especially well on such irregular recursive computations.

```
void search(GlobalAddress<Vertex> tree, index_t vertex_index) {
  // get current vertex
  Vertex v = delegate::read( tree + vertex_index );
  // explore children
  forall( v.first_child, v.num_children, [=]( index_t i ){
    search( tree, i );
  });
}

int main( int argc, char * argv[] ) {
  init( &argc, &argv );
  // T1XL: nodes: 1635119272, depth: 15, leaves: 1308100063
  GlobalAddress<Vertex> tree = uts_tree( "T1XL" );
  run( [=]{
    finish( [=]{
      // search from root
      search( tree, 0 );
    });
  });
  finalize();
  return 0;
```

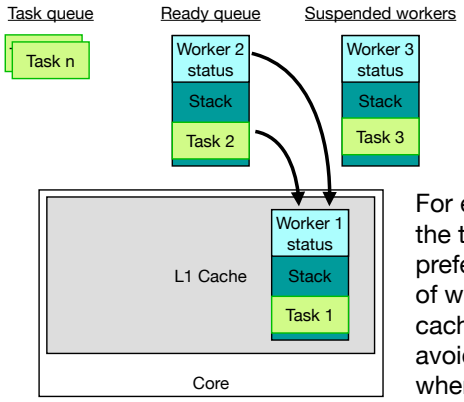


Run at Pacific Northwest National Laboratory on Cray XMT-1 and PAL cluster (1.9GHz AMD Interlagos processors, Mellanox ConnectX-2 InfiniBand network)

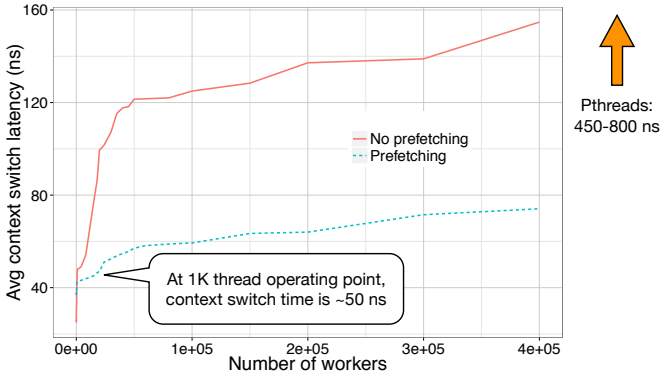
Latency tolerance in Grappa's runtime system

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

Grappa's tasking layer: Millions of threads, switching at the DRAM bandwidth limit

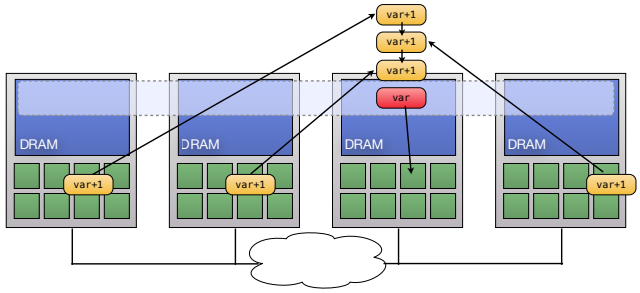


For each context switch, the task scheduler prefetches one cache-line of worker state and three cache-lines of stack, thus avoiding cache misses when loading contexts.



This micro-benchmark has many workers yielding in a loop. With 500K threads per core, context switch time is ~75 ns.
This is switching at the *bandwidth limit* to DRAM!

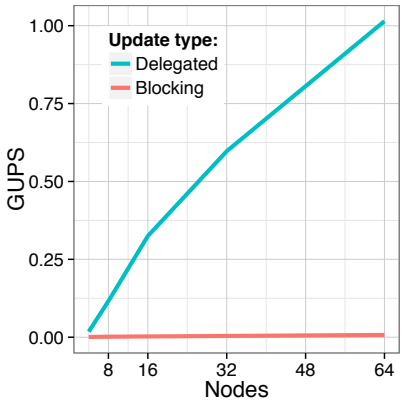
Distributed shared memory (DSM) in Grappa: Fast coherent updates through *delegation*



Every word in the system has a *home core*.

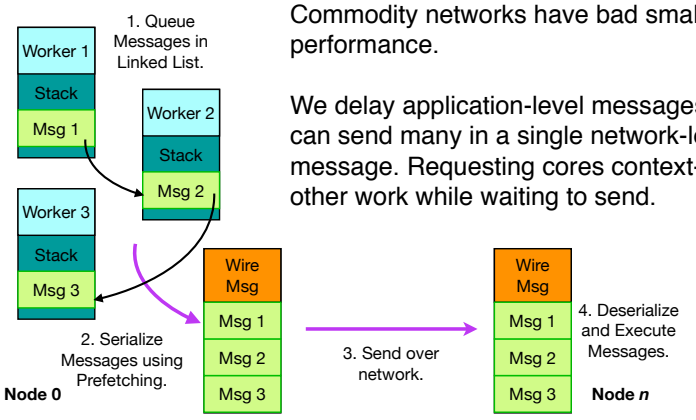
All memory accesses to a word are *delegated* to that core. Requesting cores context-switch to other work while waiting for a reply.

This gives us high-throughput atomic updates!



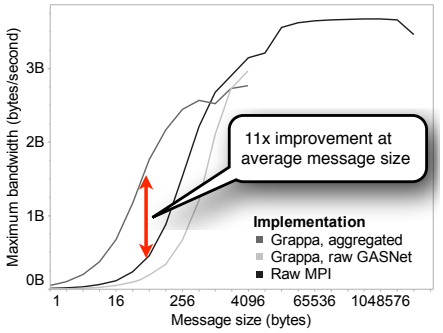
GUPS (RandomAccess) on PAL cluster at PNNL (1.9GHz AMD Interlagos processors, Mellanox ConnectX-2 Infiniband network)

Message aggregation



Commodity networks have bad small-message performance.

We delay application-level messages until we can send many in a single network-level message. Requesting cores context-switch to other work while waiting to send.



This micro-benchmark does ping-pong communication between two nodes.