Holes in paper

(1) System overhead

(1.A) Topaz Overhead.

Really the best approach is to measure this and show it isn't bad. I need to do a couple things

```
- Measure, ensure it isn't bad. (Task serialization/deserialization, topaz
bookkeeping, outlier detection
and comparison)
- Measure the outlier detector overhead - argue it is negligable.
- Task serialization/deserialization is specific to the MPI implementation.
- Compare to already-distributed implementations.
```

(1.B) Communication Overhead.

Communication != network delay.

```
- Just a computational model.
- Not necessarily network delay. Implemented using MPI for future experiments
on existing unreliable systems - we can easily make an entire machine unreliable.
- Consider shared memory models, where the memory is separate and transferring the
task
is a memcpy into unreliable memory.
- We can reduce the amount copied over by only copying over certain input data
structures once in
the taskset if we can promise it is not changed. (reduce transfer size)
- We can also batch jobs. (reduce amount of communication)
```

(2) Hardware model is unrealistic

(2.A) Why aren't there unreliable FP, Integer operations?

Not a lot of savings when compared to memory/srams.

(2.B) Why aren't there unreliable DRAMS?

Maybe I should. EXPERIMENTAL: investigate the amount of work.

(2.C) Why aren't there unreliable integer SRAMS?

Integer data is often signifcant (flags, control etc) - corrupting integers disproportionately leads to very frequent crashes.

(4) Computational Model

(4.A) Is it really fair to make programmers carve out state independent computations?

Yes! Many map-reduce frameworks require tasks be stateless and self-contained. Not uncommon for programmers to do this. Futhermore, the isolated nature of the tasks allows us to only consider inputs, outputs and time as factors in determining if an output is 'good enough'. Also removes need for expensive checkpointing and synchronization of state

(4.B) Is it really fair to assume we don't know the fault characteristics of the hardware?

Yes! The nature of which unreliable hardware will emerge is unknown - other people (see Vlad - Cake paper) , including myself, have come to the conclusion that we need to divorce ourselves from the underlying hardware model when reasoning about these problems. We also need to institute coarser grain blocks of computation to amortize overhead from switching to unreliable units.

(4.C) How is this acceptable when byzentine fault tolerance is the norm?

Byzentine fault tolerance assumes we may have an antagonistic network. Here, the network is not malicious - mostly random. So we can get away with a system that assumes a non-antagonistic error model.

(4.D) The communication step seems slow?

This is just a computational model - yes, nodes may be computers on a network - they may also be individual computational units within the same machine or components on a shared memory system. I used MPI to develop this system since it was easier to develop an emulator that operates on a process level.

(4.E) Why didn't you run this on physical hardware?

Yes, it is possible to run this system on an undervolted wimpy-node network, for example. There needs to be a lot more work on re-engineering the software stack and rethinking the kernel before that is possible. Consider this though: with my abstraction, the unreliable nodes can have a much simpler. Furthermore, the unreliable nodes may crash and be brought back up.

(4.F) Why not just replicate tasks? Why use an outlier detector?

We significantly reduce the overhead from re-execution without compromising the output. Think about it - re-executing 100% of the tasks is was worse than re-executing 0.1% of tasks. EXPERIMENTAL: With outlier detection vs executing every task twice and checking agreement.

(4.G) Why not just run the entire thing unreliabilely?

Crashes on critical sections.

(4.H) Why not just run the tasks unreliabily without re-execution?

Output is unacceptably bad.

(3) Outlier detector issues

(3.A) How is this better than a yes/no checker that is completely user defined?

A numerical checker allows for the system to accept 'close enough' answers, since numerical distance between transformed vectors is analagous to distance in the quality of the answers. Obviously, with this system, you can write a binary checker. EXPERIMENTAL: compare binary vs numerical checker for black scholes.

(3.B) How do we adapt to output distributions that shift over time? (ex: video feed)

We should implement something to handle this scenario. EXPERIMENTAL: implement time-decay characteristics on outlier detector. Illustrate with body track OR water over an extended period of time.

(3.C) How good is our learned outlier detector - what is our false positive rate? How

many incorrect outputs do we miss? EXPERIMENTAL: Measure.

(3.D) Can you show me that the incorrect outputs will not significantly sway the answer? How do we know

these outputs are okay? EXPERIMENTAL: Comparison with correct execution - shows that experimentally, the results are very similar. If possible, provide some sort abstraction guarentee.

(4) Is is worth all this trouble for the energy savings?

Yes! Coarse grain computation = reduces switching cost from reliable->unreliable hardware and back. Also - not only for energy savings - these machines may be cheaper (defects in a product line) or more efficient.

SRAMS+DRAMS, ~50% savings

If we consider undervolted systems - SRAMS are the gating factor in lowering the voltage. With these systems, we can reduce the voltage on half the SRAM banks.

(5) Benchmarks?

(5.A) Why not the entire parsec suite?

Ran out of time :(. Some benchmarks do not have logical checkers (canneal) - it is difficult to reason if the output is good. To make this stronger, I have to justify why we didn't use each benchmark we didn't use.

```
blackscholes: IMPLEMENTED
bodytrack: IMPLEMENTED
canneal: not a good way to judge kernel output quality.
dedup: compression - not a good candidate - heavy use of integers.
ferret: good candidate - no time.
facesim: good candidate - no time.
fluidanimate: similar to water.
freqmine: good candidate - no time.
raytrace:  good candidate - no time.
streamcluster: IMPLEMENTED
swaptions: good candidate - no time.
vips: good candidate - no time.
x264: too large.
```

(5.A2) Why not use the entire SPLASH2 suite

Older - not as comprehensive as parsec.

```
water: IMPLEMENTED
barnes: IMPLEMENTED
```

What about encryption/other bad candidate here

This model of computation specifically targets human-characterized classes of applications that have some sort of inbuilt redundency.

Why is it appropriate to use these benchmarks as distributed system benchmarks?

Once again - wrong question. We are using an isolated memory model to avoid reasoning about rectifying memory corruptions in a true shared memory computation. This can be a thread on a different unit etc etc.