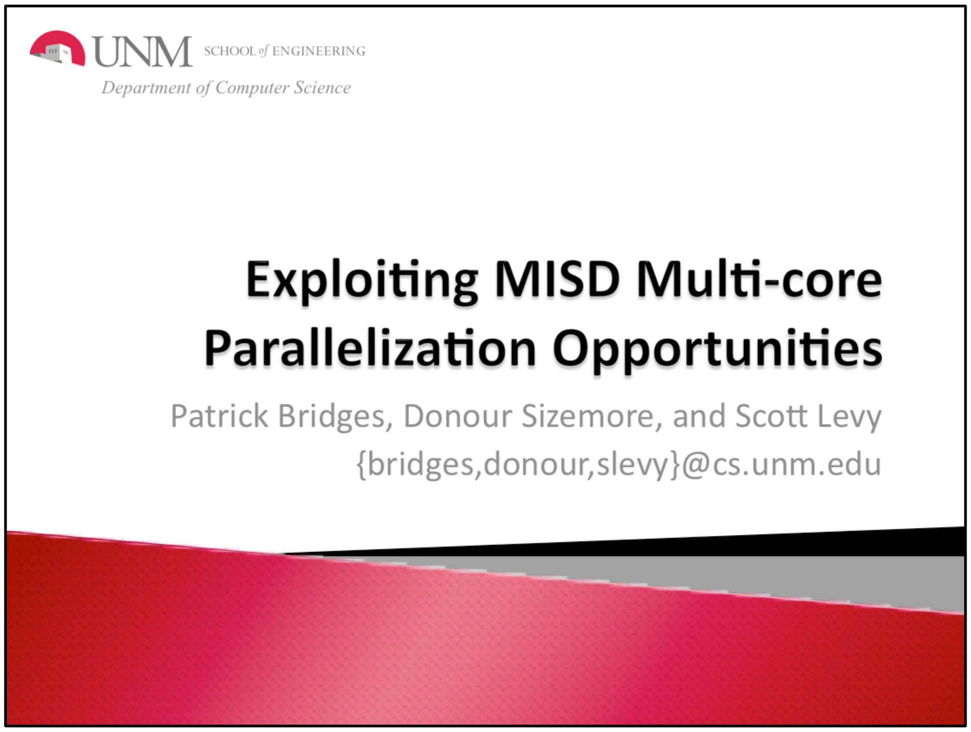
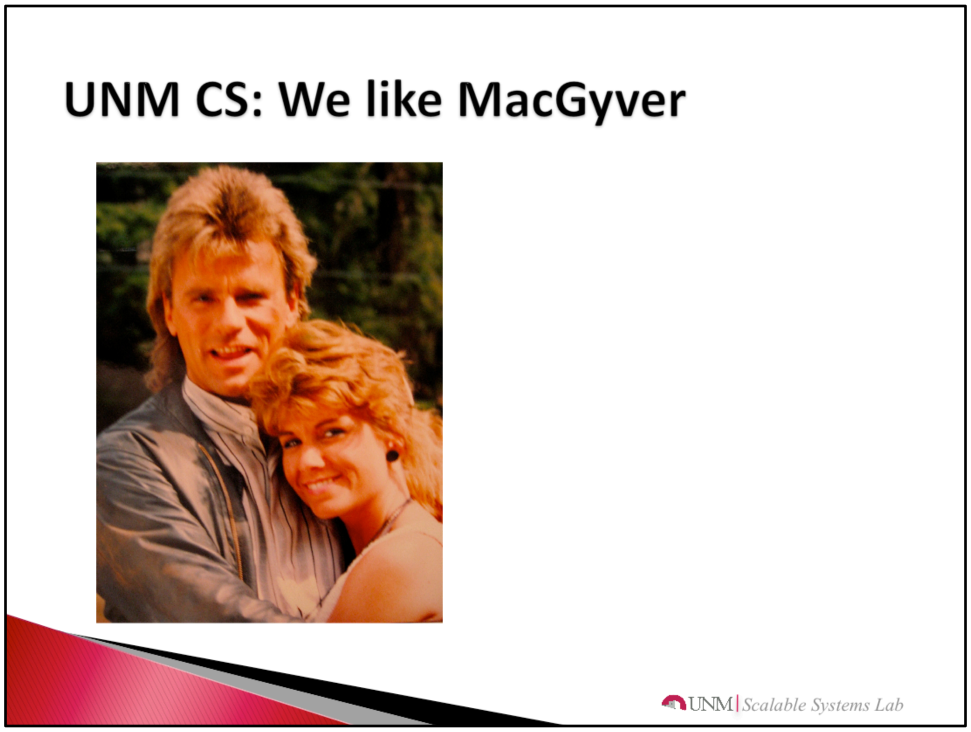


UNM SCHOOL of ENGINEERING
Department of Computer Science

Exploiting MISD Multi-core Parallelization Opportunities

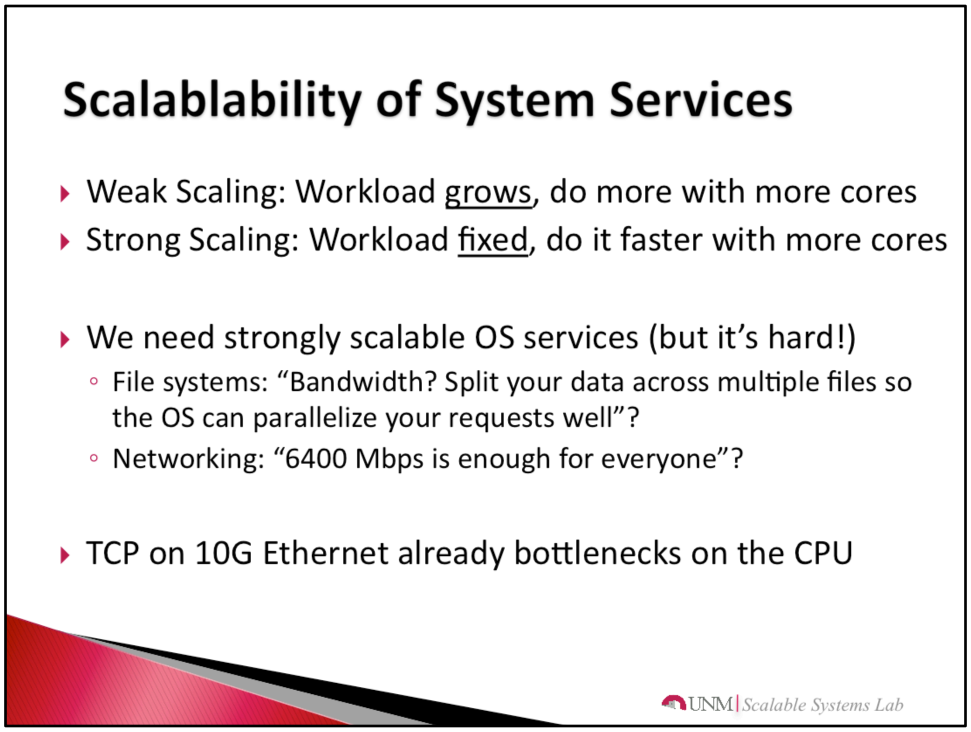
Patrick Bridges, Donour Sizemore, and Scott Levy
{bridges,donour,slevy}@cs.unm.edu



UNM CS: We like MacGyver

UNM CS has a history of pulling ideas from a diverse set of area and putting them together in unusual ways to solve problems.

UNM CS: MacGyver *is* our mascot!



Scalablability of System Services

- Weak Scaling: Workload grows, do more with more cores
- Strong Scaling: Workload fixed, do it faster with more cores
- We need strongly scalable OS services (but it's hard!)
  - File systems: "Bandwidth? Split your data across multiple files so the OS can parallelize your requests well"?
  - Networking: "6400 Mbps is enough for everyone"?
- TCP on 10G Ethernet already bottlenecks on the CPU

UNM | Scalable Systems Lab

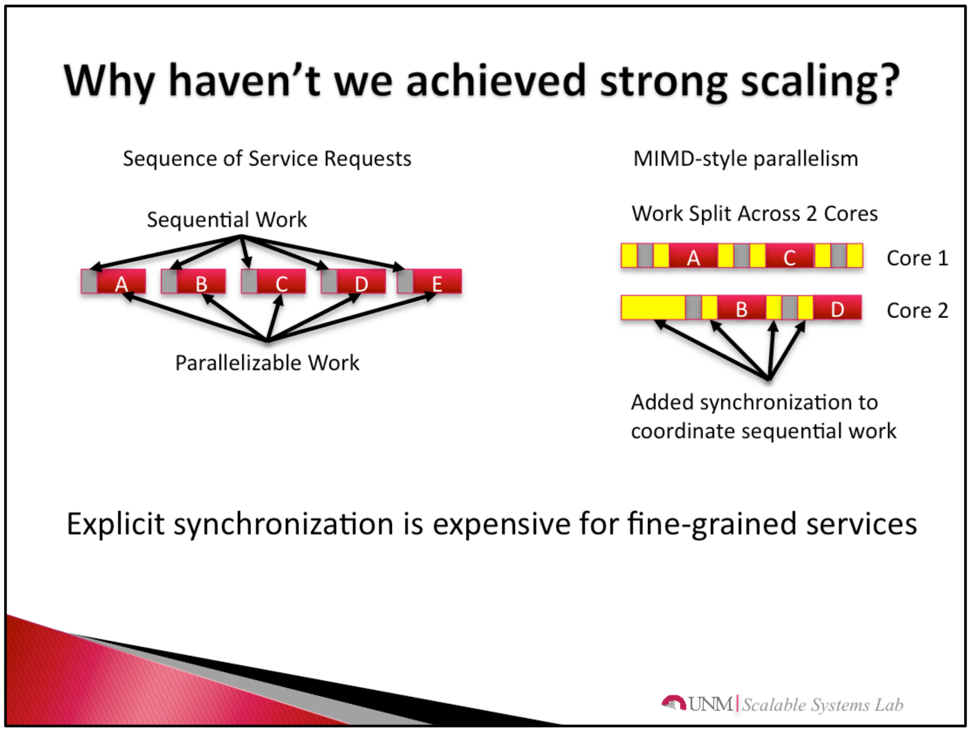
Current system software focuses on *weak scaling where* workload grows with processor count - More processes, open files, network connections

Minimal work on *strong scaling* in system software - Fixed workload executed *faster* with more processors. That means faster individual network connections, file system updates, etc.

Strongly scalable OS services increasingly important - without it, we either complicate the work of the application programmer, or worse, limit the services available them completely.

Example: Strongly scalable single TCP connections.
A single TCP connectios already bottlenecks on CPU speeds, especially at traditional MTUs
Real parallelization opportunities: data delivery, ack generation, timer expiry, etc.
Small but important inter-request dependencies: window state maintenance
Very fine-grained units of parallelization: 1500 byte packets

Synchronization kills single connection performance at this granularity!
Linux TCP connections faster with 1 core than 2 or more
Solaris doesn't even try to parallelize individual connections
Well-studied: Bjorkman 1993, Nahum 1994, Willman 2006



Why haven't we achieved strong scaling?

Sequence of Service Requests
Sequential Work
A
B
C
D
E
Parallelizable Work
MIMD-style parallelism
Work Split Across 2 Cores
A
C
Core 1
B
D
Core 2
Added synchronization to coordinate sequential work

Explicit synchronization is expensive for fine-grained services

UNM | Scalable Systems Lab

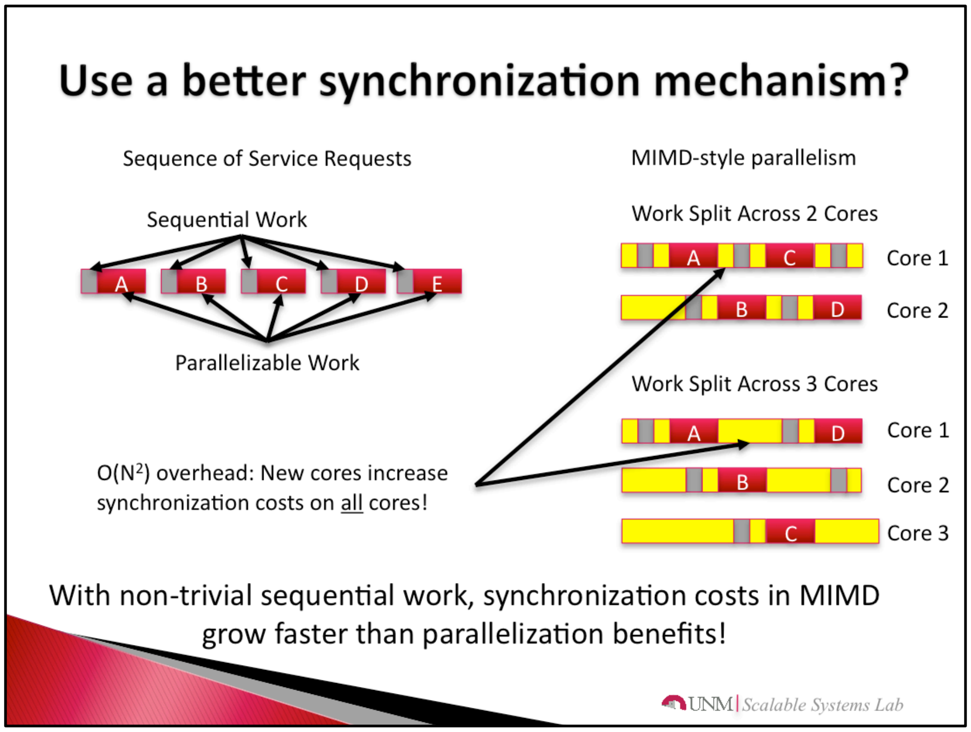
Why haven't we been able to solve this?

First: We use MIMD parallelism: Related requests split across available cores and shared state accessed via locks, IPC, transactions, etc.

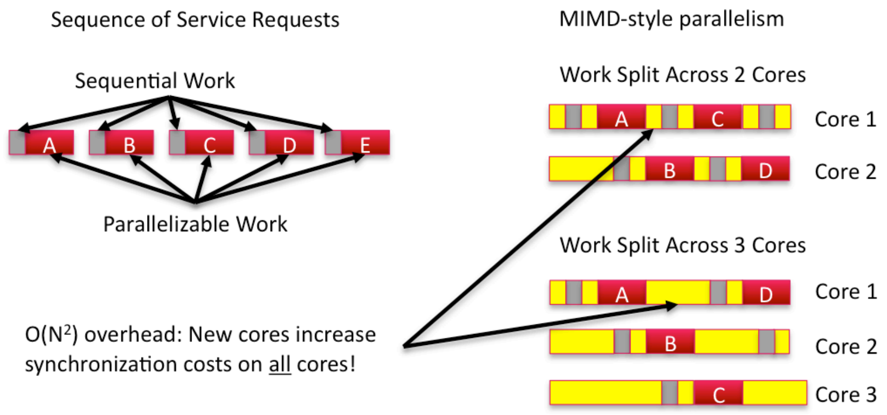
We generally use classic MIMD parallelism, which relies on explicit synchronization/ communication between processors. That's expensive compared to the unit of work at which we want to parallelize some of these services.

So why haven't the latest and greatest "special snowflake" synchronization mechanisms solved this problem?

The issue is deeper than that! What happens as you add cores in MIMD parallelism?



Use a better synchronization mechanism?



O(N^2) overhead: New cores increase synchronization costs on <u>all</u> cores!

With non-trivial sequential work, synchronization costs in MIMD grow faster than parallelization benefits!

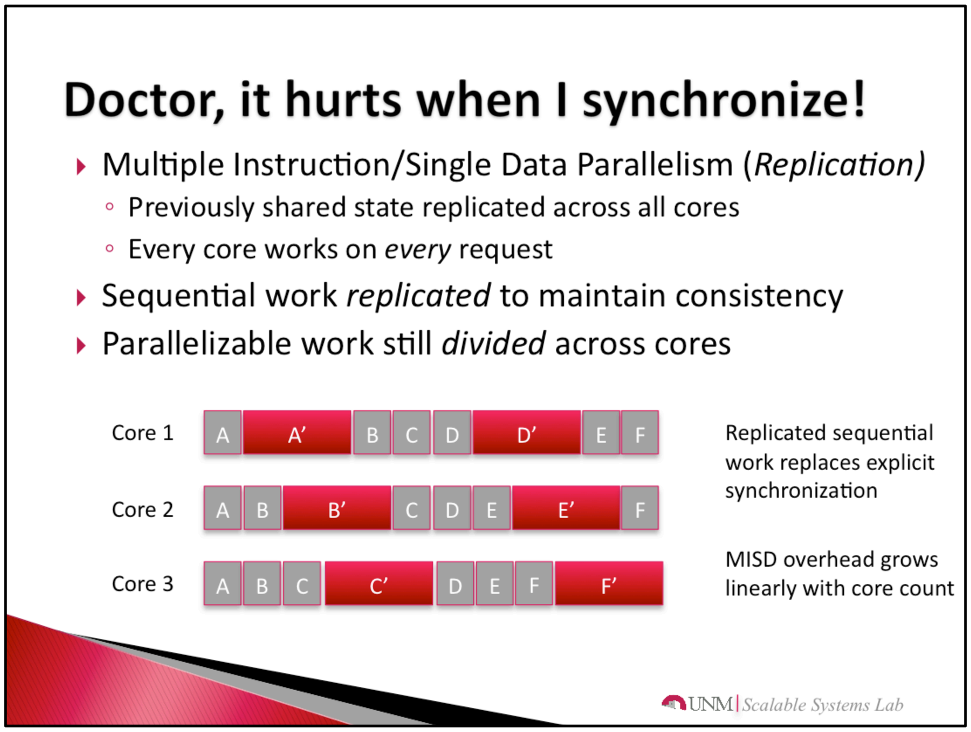
UNM | Scalable Systems Lab

Simply using a “better” synchronization method isn’t good enough – MIMD fundamentally requires explicitly coordinating N activities, and the cost of that grows quickly.

Each core you add reduces parallelizable work that has to be done on other cores but *increases* synchronization costs on *all* processors (lock contention, IPC latency, transaction rollbacks, etc.)

As you add cores, the increase in synchronization overhead is quadratic but the benefit is linear – at high core counts, any non-zero amount of explicit synchronization will kill your performance!

MIMD Parallelism kills strong scaling!



Doctor, it hurts when I synchronize!

- Multiple Instruction/Single Data Parallelism (*Replication)*
  - Previously shared state replicated across all cores
  - Every core works on *every* request
- Sequential work *replicated* to maintain consistency
- Parallelizable work still *divided* across cores

Core 1
A
A'
B
C
D
D'
E
F
Core 2
A
B
B'
C
D
E
E'
F
Core 3
A
B
C
C'
D
E
F
F'

Replicated sequential work replaces explicit synchronization

MISD overhead grows linearly with core count

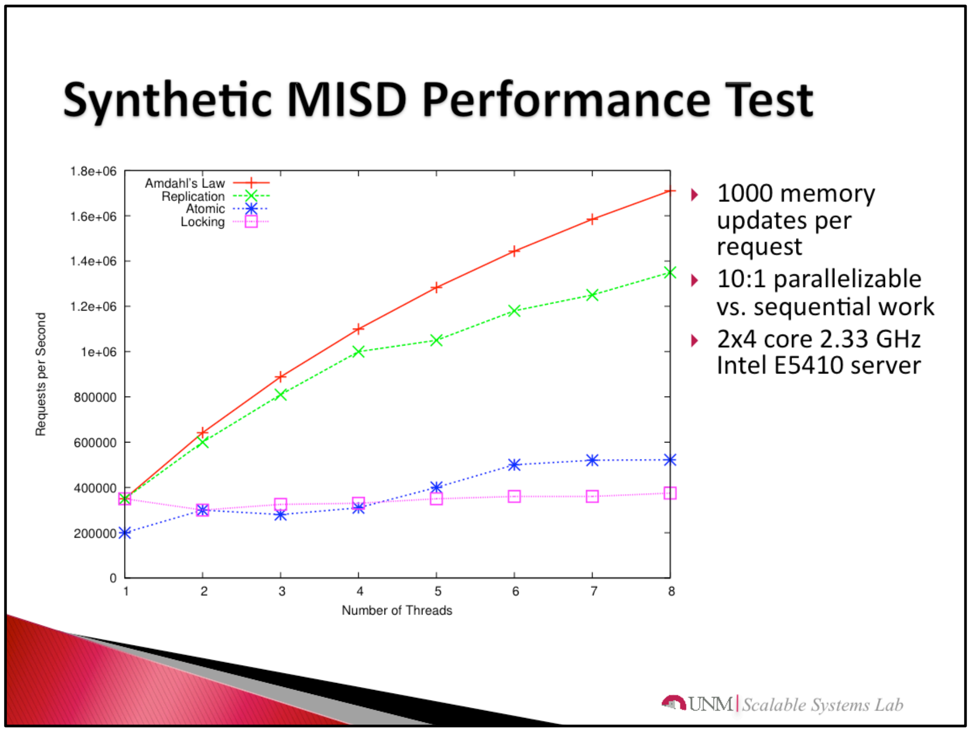
UNM | Scalable Systems Lab

So, we need to avoid explicit synchronization but still deal with non-trivial inter-request dependencies

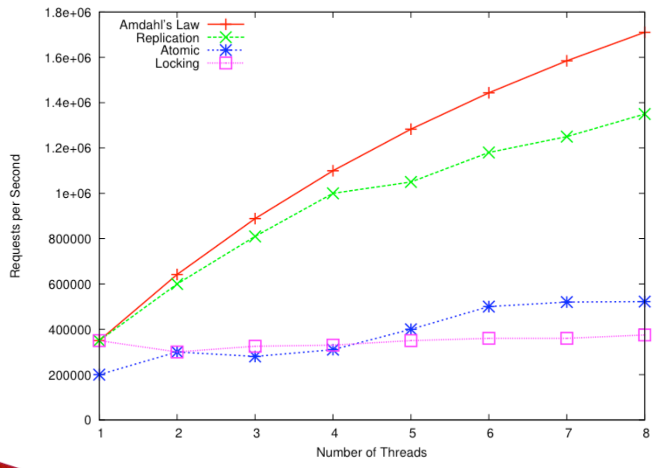
We repurpose a well-known technique from parallel and distributed computing to address this problem: MISD parallelization, that is, replicating not just state across cores as in K42, Tornado, and Barrellfish, but also replicating work on all requests on all cores.

So how do we get speedup? Like some parallel algorithms and unlike in most classic distributed systems, we replicate only the sequential work on all cores. Parallelizable work is not replicated and is still split across cores!
Essentially, we're replacing locking around sequential work with doing all the sequential work everywhere. This works well whenever it's cheaper to "just do it again" than to do explicit synchronization.

Better for fine-grained workloads where any synchronization is prohibitive, or with large core counts. Unlike MIMD, a new core adds new replicated work (overhead) only on that new core, not the previously-existing cores. So, MISD overheads grow linearly with increased core counts instead of quadratically.



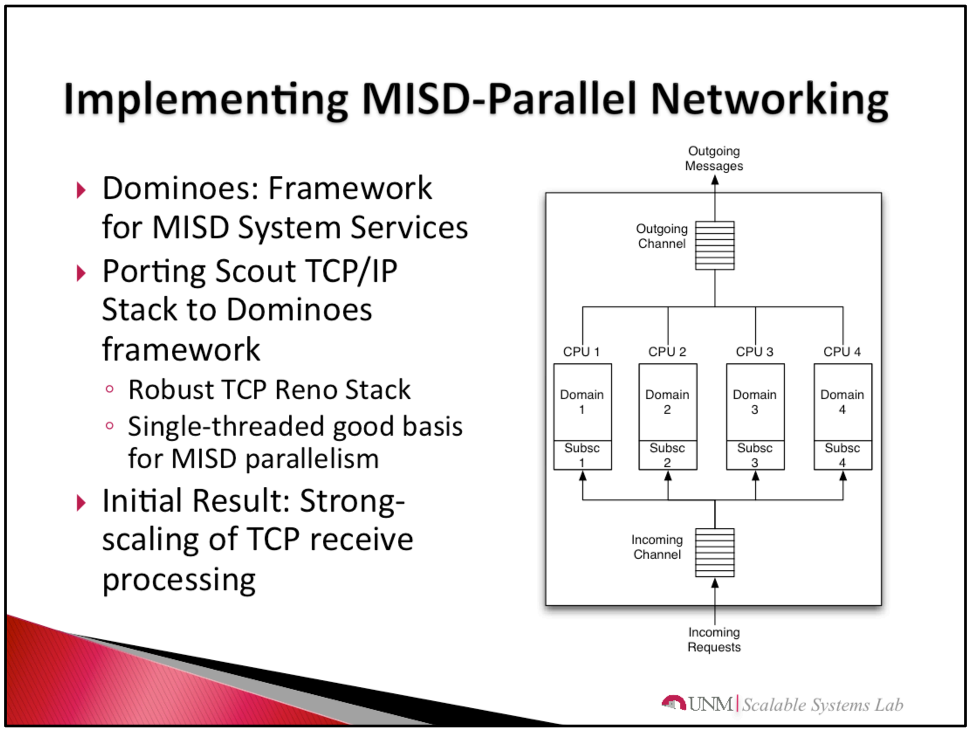
Synthetic MISD Performance Test



- 1000 memory updates per request
- 10:1 parallelizable vs. sequential work
- 2x4 core 2.33 GHz Intel E5410 server

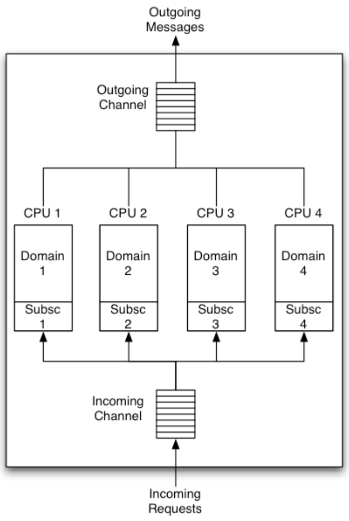
UNM | Scalable Systems Lab

The result of this is that for fine-grained workloads, which we expect to be common in system software, a MISD-based approach tracks the best possible performance as given by Amdahl's law much better than approaches based on explicit synchronization.



Implementing MISD-Parallel Networking

- Dominoes: Framework for MISD System Services
- Porting Scout TCP/IP Stack to Dominoes framework
  - Robust TCP Reno Stack
  - Single-threaded good basis for MISD parallelism
- Initial Result: Strong-scaling of TCP receive processing

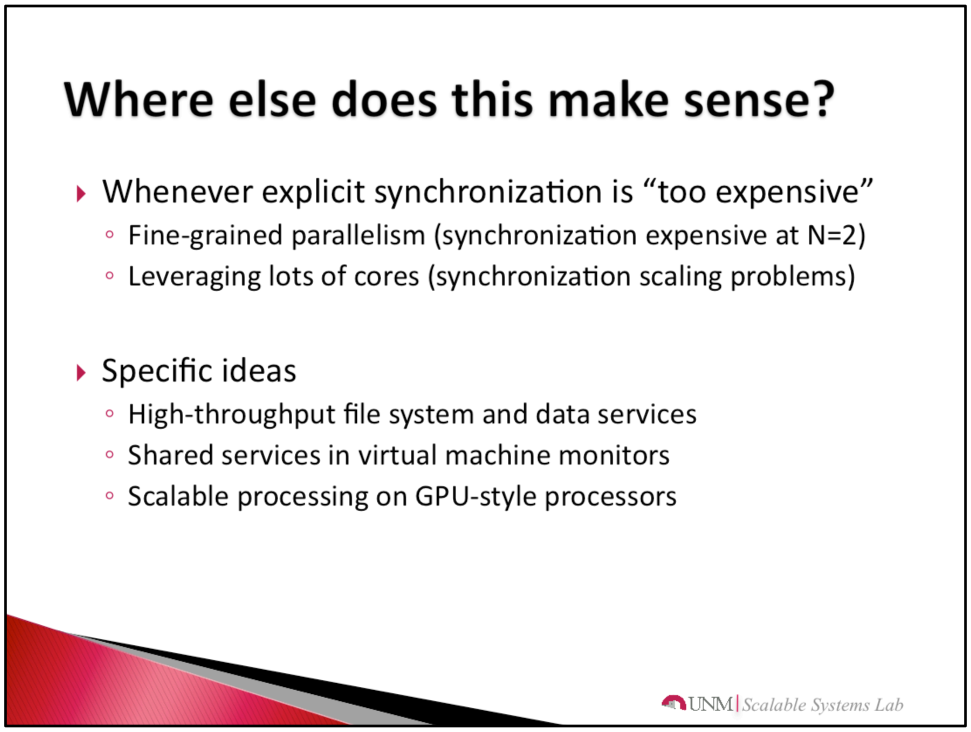


UNM | Scalable Systems Lab

Dominoes: Framework for MISD System Services
Broadcast FIFO channels to distribute requests
Publish/subscribe event-based programming model
More framework details in paper

Porting Scout TCP/IP Stack to Dominoes
Single-threaded stack easy to integrate and parallelize
Evaluate difficulty of using MISD parallelization with a single-threaded event-based service

Initial result: First-ever strong-scaling of TCP receive processing
1.8x improvement in TCP receive throughput using 4 cores
TCP Send is harder
Working on optimized zero-copy library-level implementation
Only modest changes to scout to get it to run in dominoes



Where else does this make sense?

- Whenever explicit synchronization is "too expensive"
  - Fine-grained parallelism (synchronization expensive at N=2)
  - Leveraging lots of cores (synchronization scaling problems)
- Specific ideas
  - High-throughput file system and data services
  - Shared services in virtual machine monitors
  - Scalable processing on GPU-style processors

UNM | Scalable Systems Lab

High-throughput file system and data services

Replicate in-memory metadata (FS state, buffer cache info)

Parallelize data manipulation

Virtual machine services

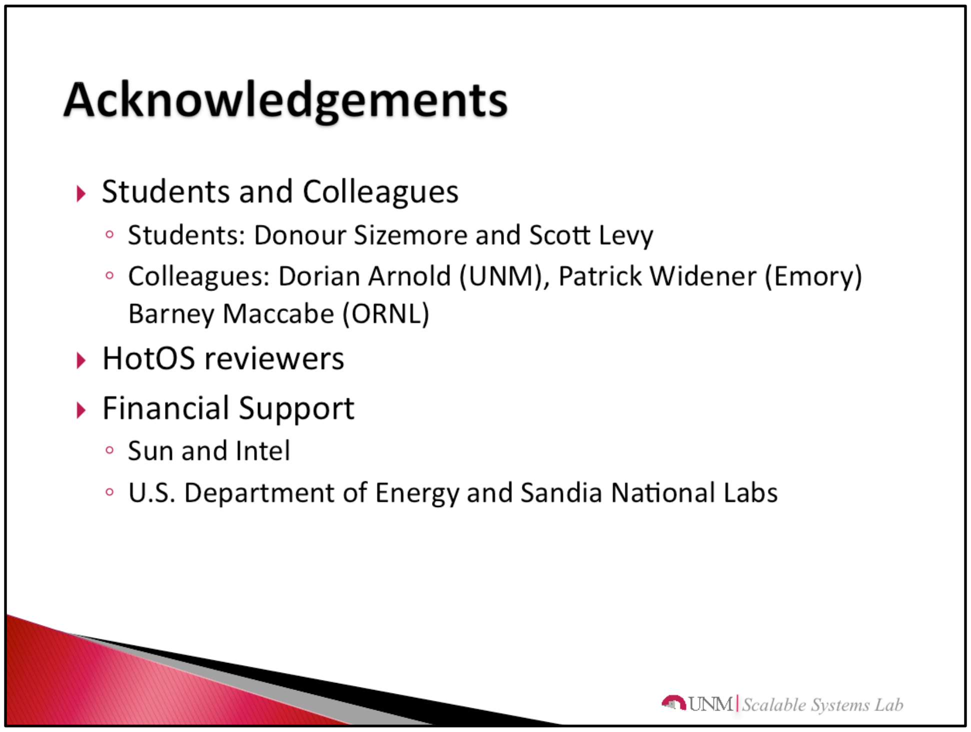
Shared virtual devices – virtual network switches

Memory page de-duplication and/or compression?

Parallel OS services on GPUs

MISD replication can leverage large amounts of parallelism

Without the locks that are prohibitive on GPUs

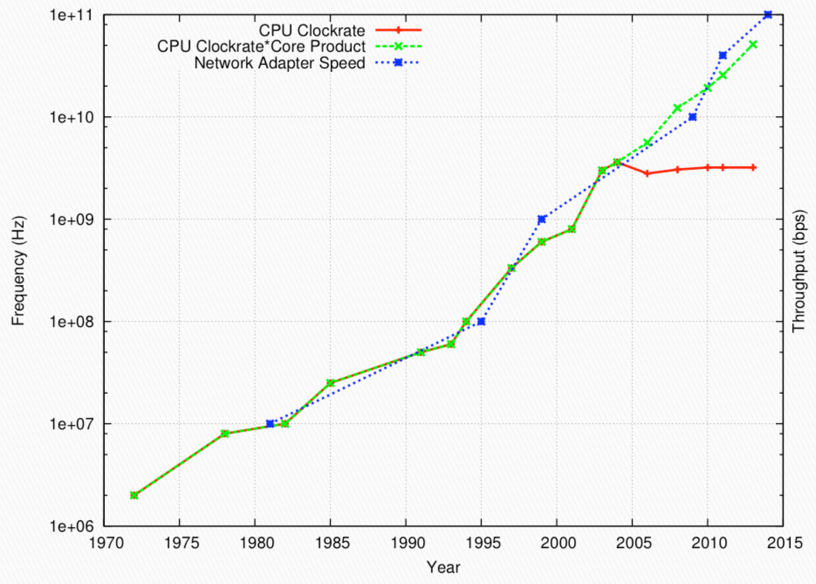


Acknowledgements

- Students and Colleagues
  - Students: Donour Sizemore and Scott Levy
  - Colleagues: Dorian Arnold (UNM), Patrick Widener (Emory) Barney Maccabe (ORNL)
- HotOS reviewers
- Financial Support
  - Sun and Intel
  - U.S. Department of Energy and Sandia National Labs

UNM | Scalable Systems Lab

Now that’s a gap…



Solving these problems without parallelism (or protocol changes!) means exponential increases somewhere else – for example in MTU,