Scalable Launch on El Capitan with Spindle

July 2026

Matthew LeGendre
Livermore Computing

Prepared by LLNL under Contract DE-AC52-07NA27344.

In This Talk

Spindle Overview on Scalable Library Loading

El Capitan Challenges for Spindle

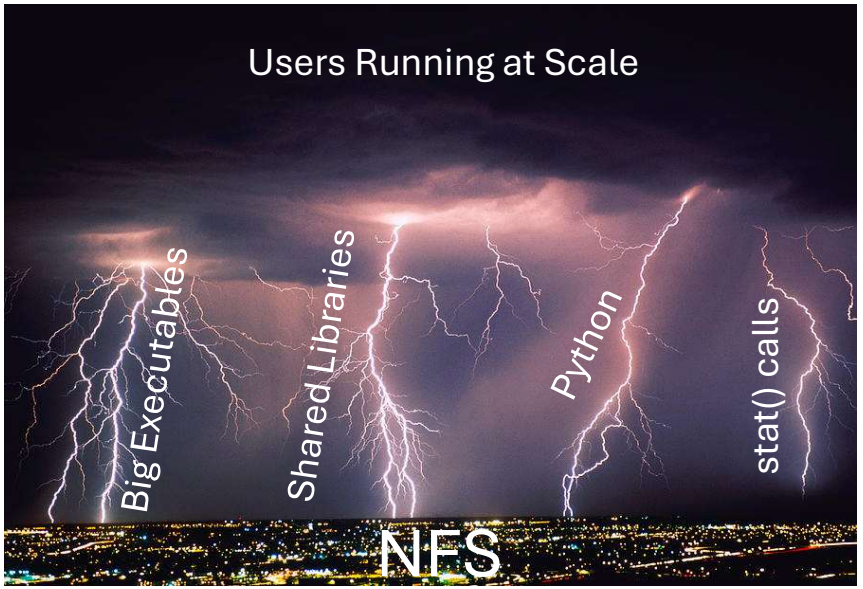
LD_AUDIT for Tool Builders

```
% export LD_AUDIT=/libtool.so
```

El Capitan

- Debuted Number #1 on Top500 (2.79 Exaflops peak, 1.74 Exaflops HPL). Currently #2.
- 11,424 nodes w/ 4 AMD MI300A APUs per node
- HPE Rabbits provide near node local storage.
  - 16 nodes share one rabbit
  - 32 TB of storage by 18 SSD drives

Application Startup can Cause File Access Storms



- Application startup can put tremendous load on our shared file systems.
- The load affects the whole center.
- One app on Dawn took ~10 hours to reach main().

Library Requests Overwhelm the File System

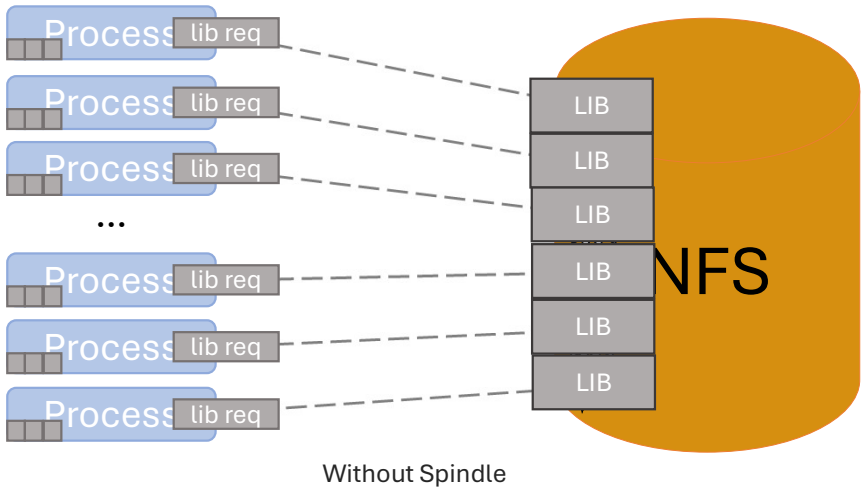
Applications are **searching** for libraries in directory locations, and **reading** library contents.

File search operations:

# of tests = # of processes
x # of locations
x # of libraries

File read operations:

# of reads = # of processes
x # of libraries



Without Spindle

Spindle Changes Library Loading to be Scalable

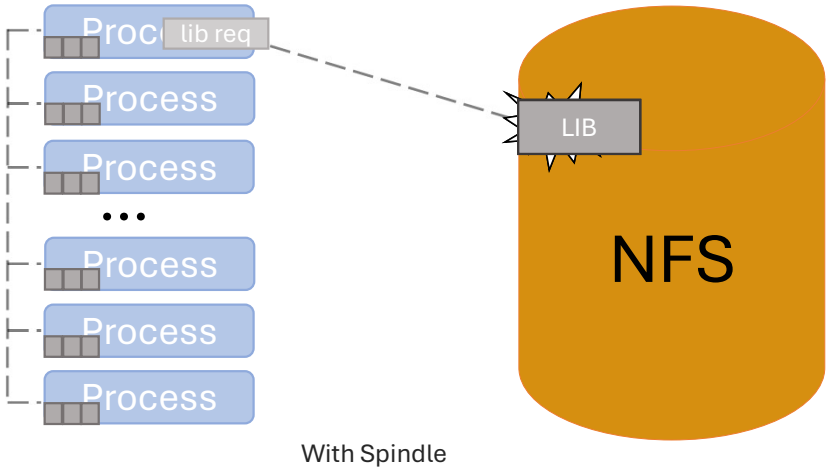
With Spindle one node reads files and scalably broadcasts results to caches on other nodes.

File search operations:

# of tests = *# of locations* x *# of libraries*

File read operations:

# of reads = *# of libraries*



With Spindle

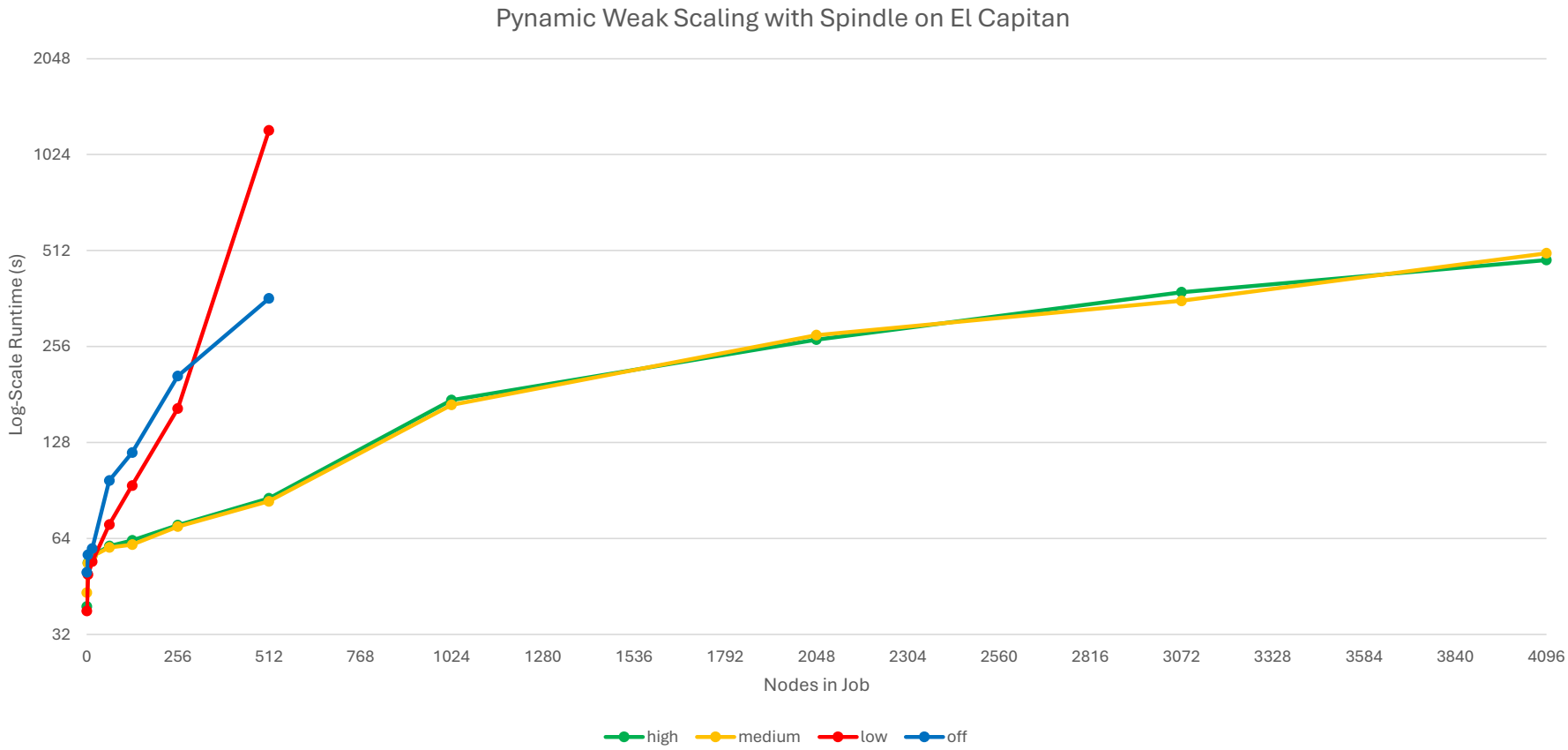
Spindle Scalably Loads Many File Operations

- Scalable file loads and operations for:

Shared Libraries	Executables	Python Libraries
Existence Tests	stat()	realpath()
readlink()	System Configs	User-Specified Files

- But…
  - Spindle breaks POSIX by caching I/O results.
  - Safe if done on files that aren't written to.
  - Spindle focuses on files needed for startup (libraries, configs, …)

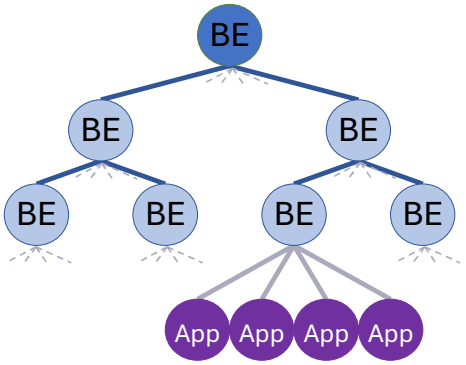
Spindle Drastically Improves Start-Up Performance



The Spindle Network

- A "BE" server on each node manages caches and networks.
- A "client" library in each application process intercept I/O calls.

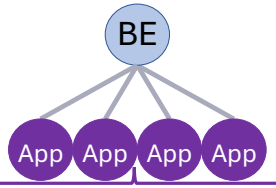
Requests for I/O are intercepted and sent up a TBON.



I/O is performed by one node. Results are broadcast down.

I/O results are cached on nodes.

Spindle intercepts Libraries Searches with LD_AUDIT



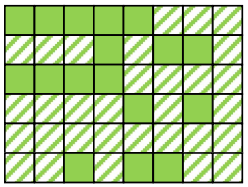

```
export LD_AUDIT=mytool.so

mytool.so:

char *la_objsearch(const char *name, ...) {
  //Callback when ld-linux.so searches for lib
}

char *la_objopen(struct link_map* map, ...) {
  //Callback when ld-linux.so opens a lib
}

uintptr_t la_symbind64(Elf64_Sym *sym, ...) {
  //Callback when ld-linux.so looks up symbol
}
```

- Spindle's injected libraries gets callbacks when libraries/python load files.
- File requests are transferred to the servers/networks.

Spindle Caches Libraries on Each Node

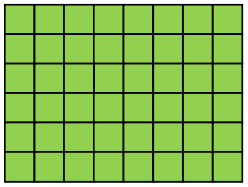
- Spindle's cache is in-memory TMPFS (like a ramdisk).
- Linux shares physical memory pages between cache and application.

Without Spindle:



- Code pages are loaded to physical memory when first touched
- Untouched pages do not take physical memory
- But, pages may thrash in/out in low-memory situations

With Spindle:



- All code pages are always resident (since they're backed by in-memory TMPFS)
- No thrashing, since pages can't eject from TMPFS

Spindle's overhead is a function of an application memory residency, but generally low
Spindle changes low-memory behavior

Spindle on El Capitan

Lessons Learned:

- Component reliability for large systems drove tool reliability.
- AI-Centric workflows very different from traditional HPC workflows.
- At scale, every I/O operation counts.
- Long tail of hardening work: CI/CD, LD_AUDIT bugs, Spack

Tool Reliability

- Part count, not node count, drives El Capitan reliability.
- Rabbits rely on newer technology in shared PCI bus.

Requires re-engineering Spindle's storage and network:

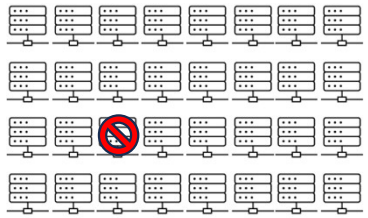
- Health checks for local I/O
- TBON fault tolerance under implementation.
- On-node fault tolerance—fail gracefully.

El Capitan Lessons – AI Workloads are not HPC Workloads

- Pytorch has a unique library load fingerprint
  - ~50k – Large HPC Application #I/O-ops/process at startup
  - ~150k – A minimal pytorch test #I/O-ops/process at startup
  - Pytorch creates JIT compiled libraries at runtime.

PyTorch

- ML training runs are inherently more fault resilient than HPC runs. Spindle must match that.



Every I/O Operation Counts

- Spindle had to capture additional I/O calls:
  - readlink, readlinkat, realpath
- Spindle now intercepts core dumps
  1. Identify a crash site key:
     1. Assert string
     2. Fault type+fault location
  2. Use spindle network to select 1-n processes to dump core per fault site
  3. Create a log file of which core files dump represent which processes.

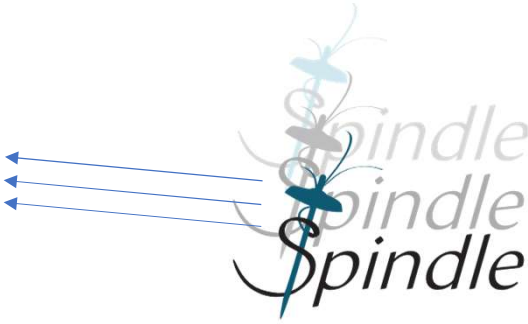
Spindle is integrated into every El Capitan job via Flux

- Any flux run on El Capitan uses Spindle

```
% flux run -N 1024 ./a.out
```

- Group runs under a single spindle session for better performance

```
% spindle --start-session
%   flux run -N 8 …
%   flux submit …
% spindle --end-session
```



Spindle Experiences on El Capitan

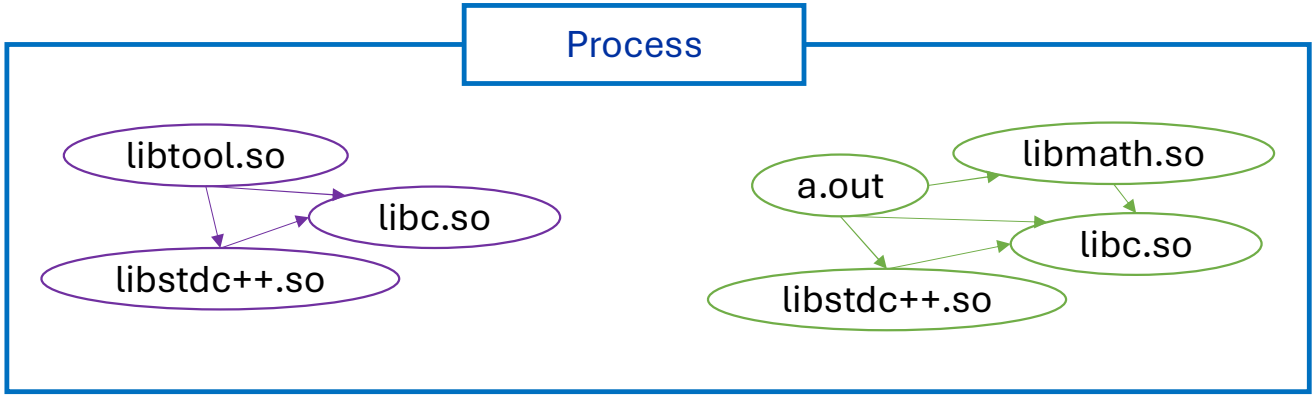
- Lots of spindle tunables. Simplified with ‘level’ argument.
  - `flux run -o spindle.`**`level=off`** – Turns spindle off
  - `flux run -o spindle.`**`level=low`** – Only caches file existence test results, not file contents.
  - `flux run -o spindle.`**`level=medium`** – Caches libraries, python files, other I/O results. Does not cache executables.
  - `flux run -o spindle.`**`level=high`** – Caches everything from medium, plus executables. Breaks debuggers.
- Made medium default.

Always-On Spindle for El Capitan has generated lots of work

- Anything that writes executables, python, or shared libraries is fundamentally incompatible with spindle:
  - Compilers/linkers, Text editors, package managers, etc…
  - Always-on spindle needed tuning to recognize and turns itself off for these.
- Lots of bug fixes for applications that behave different when libraries are loaded from caches.
  - Focus on making spindle's changes invisible to applications.

LD_AUDIT in Theory ☺

- Point LD_AUDIT at a library, similar to LD_PRELOAD
- Callback interface supported by dynamic linker
  - At library search attempts
  - At library loads
  - At symbol binding
  - At program startup/exit
  - At PLT-bound function calls
- Tooling gets its own library namespace (ala dlmopen)



LD_AUDIT in Practice ☹

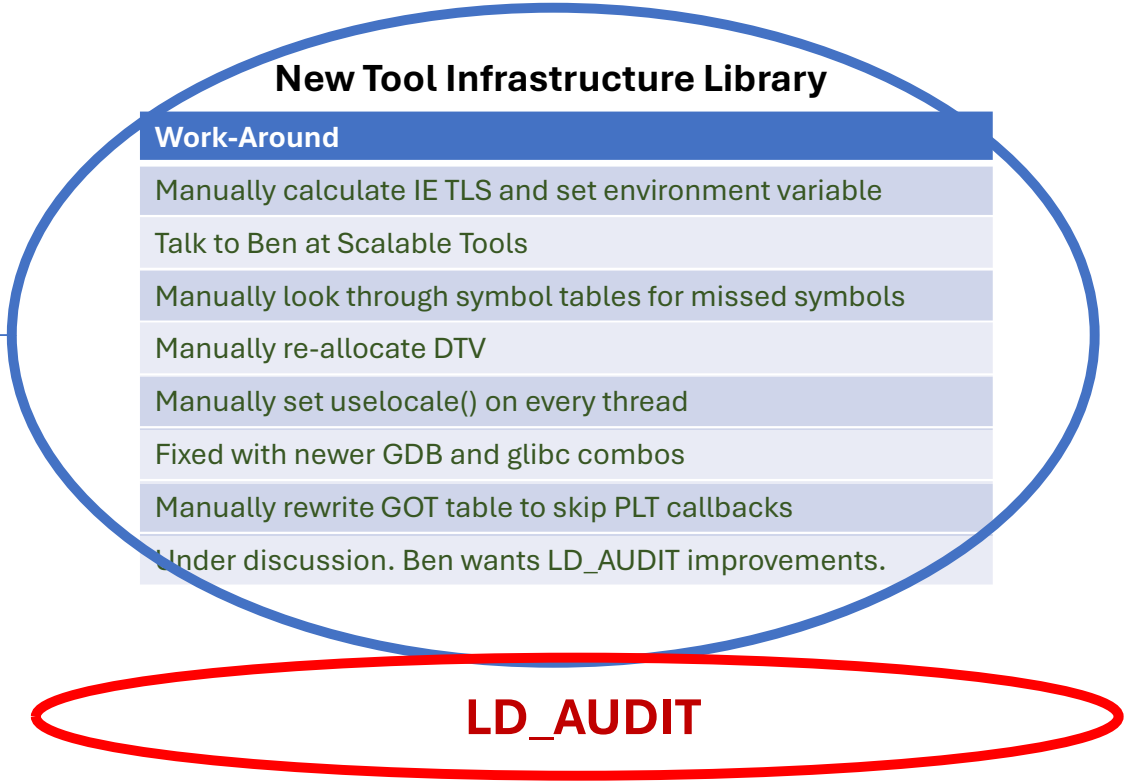
LD_AUDIT Bug in GLIBC	Work-Around
IE TLS not allocated sufficiently	Manually calculate IE TLS and set environment variable
IE TLS does not align correctly	Talk to Ben at Scalable Tools
Symbol binding callbacks not triggered for data symbols	Manually look through symbol tables for missed symbols
DTV realloc crash when too many app libraries use TLS	Manually re-allocate DTV
App-created threads crashes accessing TLS in tool.	Manually set uselocale() on every thread
AUDIT-loaded code is invisible to debuggers	Fixed with newer GDB and glibc combos
High overheads when using PLT callbacks	Manually rewrite GOT table to skip PLT callbacks
Can't AUDIT other AUDIT libraries	Under discussion. Ben wants LD_AUDIT improvements.

Most of These are Fixed in Newer GLIBCs. But we still carry workarounds for old GLIBCs.

Proposal: Let's Build an LD_AUDIT Tool Wrapper Library

Major Capabilities:

- Wrap app functions
- Replace app libraries
- Init/Fini Callbacks
- Multi-tool management
- …



Questions?