# Parallel I/O (pio):

# Fulfilling the promise

# Parallel I/O (pio)

- LU-8964: Added parallel tasks framework & pio
- Reaction to slower many core processors (KNL)
- Split I/O at cl_io layer (cl_io_loop), do in parallel
- Parallelizes page submission, allocation, & data copying (all the CPU intensive parts)
- Off by default...

# Performance

- Doesn't perform very well in practice
- Hurts performance for many real workloads
- Hence, off by default
- See Dmitry's LUG presentation

# Goals

- 1. On by default
  - “Do no harm” - don't make any workloads worse
  - Help enough (CPU cycles aren't free)
- 2. Performance = normal multi-process shared file
  - Clear yardstick for progress
- 3. (Bonus!) Improve multi-process shared file
  - Helps pio and single shared file...

# So, what makes it slow?

- 1. Too many processes
  - Does a process per “stripe chunk” right now (cl_io_loop iterates once per stripe chunk)
  - Larger I/Os lead to contention (64 MiB == 64 processes!)
- 2. Too little data per process
  - “stripe chunk” is only stripe size bytes of data
- 3. Scheduling policy
  - Padata framework binds threads to specific CPUs
  - Problematic for any workload, but especially HPC

# So, what makes it slow?

- 4. Plays havoc with readahead, hurts reads
  - Re-write readahead (LU-8964 – Seems to work well)
- 5. Overhead
  - Unnecessary serialization (completion order for different parts of I/O doesn't matter)
  - Task startup time – padata is high overhead

# What can we do?

- Don't use padata (can't avoid per-CPU binding, high overhead to start each worker)
- Remove unnecessary serialization (task completion order for I/O doesn't matter)
- Limit process count per file
- Limit total pio process count
- Split data equally between # of processes, but only above a minimum size

# Digression: Padata

- Existing in kernel parallelization framework
- Chooses a random CPU, puts item on per-cpu queue & starts worker (if not already started)
- Workers always explicitly bound to CPUs ( : ( )
- Powerful parallel execution & serial completion primitives
- Used by networking (packet encryption)
- Designed for many, many small work units (TCP packets!) and a dedicated system

# Padata Conclusion

- Padata is great... But probably not for us.
- Lots of overhead
- Switch to kthread_run and we're ~20% faster
- Switch to pre-created daemons is ~30% faster
- Less Complexity – Drop a lot of code going to kthread_run or daemons

# What have I done? (LU – Not yet.)

- Switch to pre-created ptask daemons
- Limited total number of daemons
- Removed serializaton at end of work
- Chunk data evenly above minimum size
- Limit processes per file