



Lustre SMP scaling

Liang Zhen 2009-11-12



Where do we start from?

- Initial goal of the project
 - > Soft lockup of Lnet on client side
 - Portal can have very long match-list (5K+)
- Survey on 4-cores machine
 - > lock-meter shows high contention on LNET_LOCK
 - 40+% UTIL (fraction of time that the lock was held during the report interval)
 - 60% CON (fraction of lock requests that found the lock was busy when it was requested)

What do we do with holding of LNET_LOCK?

- List operations (`list_add`, `list_del`, `list_for_each...`)
 - > All Lnet descriptors are put on list & hash
 - > Lnet peers are on hash table
- Data assignments & memory freeing
 - > Initialize & change state of Lnet descriptors
 - > Free Lnet descriptors
- EQ callbacks (`ptlrpc::events.c`)
 - > A bit heavy, i.e: `request_in_callback`
 - `memset`, `allocate (GFP_ATOMIC)`, `waitq wakeup`

The first step improvement

- Changes:
 - > Grained lock: One global lock(LNET_LOCK) -> three global locks
 - > Portal match list is replaced by hash table
 - > Logic independent members of global structure are put on different cacheline (protected by different locks)
 - > Some assignments are moved out from locks
 - > Lock table for EQ callbacks, so they can happen concurrently
- Results:
 - > 4 cores: better performance; Lockmeter: 4% UTIL, 15% CON
 - > 8+ cores: very limited better, still a disaster while running insanity network test like LST

What's going on? (1/4)

- Memory speeds can't catch up with CPU speeds
- Synchronization requires consistent view of data across CPUs, so synchronization is much much slower than normal instructions because of memory latency
- We made a lot of efforts to make critical section be faster, but critical section efficiency is bad
 - > T_a (lock acquisition), T_c (Critical section), T_r (lock release)
 - > Efficiency = $T_c / (T_a + T_c + T_r)$

What's going on? 2/4

- Overhead of synchronization

	1 thread	2 threads	4 threads	8 threads	16 threads
Global spinlock	18,444,000	2,082,000	440,750	101,000	25,937
Global readlock	12,189,000	2,363,000	885,250	419,750	215,937
Global atomic	30,001,000	4,070,000	1,912,500	1,117,875	745,500
Private spinlock	18,469,000	18,187,500	17,982,000	18,451,500	18,485,750

What's going on? (3/4)

- global things always hurt us
 - > Writing to global variables can result a lot of cache & bus traffic
 - Global counters
 - Global atomic / refcount
 - > Writing data to different portions of the same cacheline can result a lot of cache & bus traffic

```
Struct foobar {  
    spinlock_t locka;  
    int a;  
    spinlock lockb;  
    int b;  
};
```

What's going on? (4/4)

- Non-CPU-affinity threads-pool
 - > Thread is scheduled by a different CPUs, all data need to be fetched to it's cache
- Global waitq
 - > It's a combination of global spin_lock, global list.
 - > It will be a disaster if all CPUs race on the global waitq

smp scaling code – synchronization (1/2)

- Always try to avoid global spinlock on hot-path
 - > Hot-path: let's just say, if the code is called ≥ 1 time for each RPC, that it's hot-path
- Create our own synchronization while necessary
 - > Rcu is not portable
 - > Lustre_hash is protected by two-levels of locks
 - Each bucket has it's own lock, less contention
 - Global lock is rarely called, so doesn't really matter

smp scaling code – synchronization (2/2)

- Lock-array can't work as good as expected
 - > Example: lustre_hash
 - > Numbers (almost no change between 8->16)

	1 thread	2 threads	4 threads	8 threads	16 threads
Lock array	18,256,000	3,629,000	1,761,000	839,375	809,250

- Always be cautious while using rwlock
 - > Even read_lock has more overhead than spin_lock
 - > If write_lock is not really rare, read/write contention is much worse than spin_lock contention

smp scaling code – globals (1/2)

- How bad it can be ?
 - > 16-cores: **Oprofile** shows `lprocfs_stats_lock` taking 5% of total samples because of this (after all other smp scaling changes)
(NB: the number of samples collected in the routine is proportional to the the time spent by the processor in executing functions. The more samples collected, the more time the processor has spent executing those functions)

```
stats->ls_flags |= LPROCFS_STATS_GET_SMP_ID;
```

- > it's disappeared from oprofile if we change it to:
if `((stats->ls_flags & LPROCFS_STATS_GET_SMP_ID) == 0)`
 `stats->ls_flags |= LPROCFS_STATS_GET_SMP_ID;`

smp scaling code – globals (2/2)

- Lazy update
 - > Don't need worry about rare updated globals
- Per-CPU data
 - > Each CPU has it's own data, no race most time
 - Buffers, stats
 - > Linux is using per-cpu data everywhere, but we don't
- Big-refcount
 - > Global refcount can be a problem on smp system
 - > It's inheritance of per-cpu data
 - > The big-refcount is updated rarely

smp scaling code – cpu-affinity thread

- global waitq is extremely bad
 - > Most LNDs has one global waitq, each ptlrpc service has a global waitq
- per-cpu waitq and CPU affinity thread can greatly reduce data traffic
 - > LST shows that, if all threads are bound on just exact 1 CPU, we can handle max to 100K PRCs/sec, but we never saw number like while running lots of non-cpu-affinity threads on all-cores, so single CPU plays better than mutiple-cores, ☹️

Smp scaling code – soft lockup

- soft lockup
 - > CPU is hugged by one or several threads, kernel rises warning for that
- We suffered a lot already
 - > Rehash of lustre_hash
 - > CDEBUG in big loop
 - > LST on fat-cores machines
- How to avoid
 - > Avoid big-loop with holding spinlock, CPU could be starved while there are a lot cores
 - > Cond_resched() if necessary

SMP scaling code – Misc (1/2)

- Reduce cacheline conflict
- large inline function is not good
- Always be cautious of hash function
- Put assignments in locks only if it's really necessary

```
foo->a = v1; // cache miss
```

```
spin_lock(&lock);
```

```
foo->b = v2;
```

```
spin_unlock(&lock);
```

```
spin_lock(&lock);
```

```
foo->a = v1; // cache miss
```

```
foo->b = v2;
```

```
spin_unlock(&lock);
```

SMP scaling code – Misc (2/2)

- Memset is burning CPU
 - LIBCFS_ALLOC in Lnet
 - Memset in ptlrpc
- ASSERTION is stealing CPU
 - `LASSERT(module_refcount(key->lct_owner) > 0);`
- directly logic statements is better than memcmp
 - `If (memcmp(&a, &b) == 0)`
 - `If (a->v1 == b->v1 && a->v2 == b->v2)`

Where are we now?

- LIBCFS
- Lnet & LND
- lu_site & lu_context_key
- Ptlrpc & Idlm

LIBCFS

- CPU abstract layer
 - > Map one or more cores to one “libcfs cpu node”
 - > Bind thread to one “libcfs cpu node”
 - > Configurable
- Per-CPU data memory allocator
 - > Allocate/free cacheline alignment buffers for each “libcfs cpu node”
- `cfs_waitq_add_exclusive_head`
 - > Always add the thread to head of exclusive waitq
 - > Greatly reduced active threads of ptlrpc service
- Watchdog cleanup

Lnet & LND (1/2)

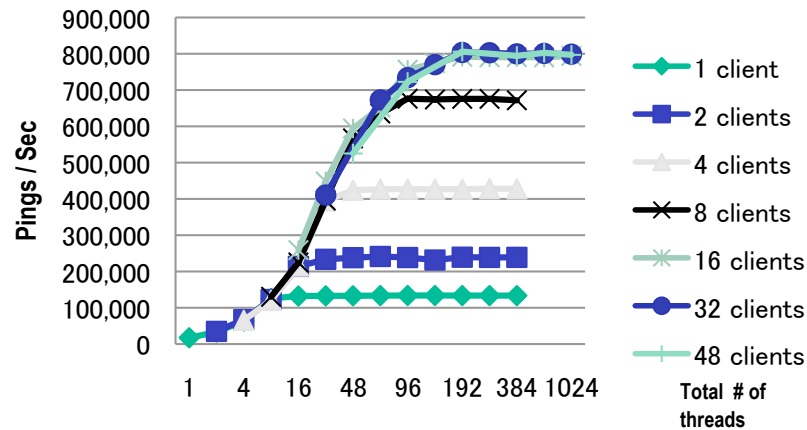
- per-CPU peer table and pools for LND, peers are hashed to LND schedulers by NID
- CPU affinity LND threads, per-CPU waitq for LND threads
- per-CPU lock & data for Lnet
 - > each CPU maintains it's own ME & MD on portal
 - > LNetMDBind always bind MD on current CPU if there is
 - > lnet_match_md can steal buffer from other CPUs (request portal only)
 - > MD created by LNetMDAttach is hashed to different CPUs by match_bits
 - > each CPU has it's own peer table
 - > CPU id is saved in lnet_handle_*_t(transparent to upper layer)

Lnet & LND (2/2)

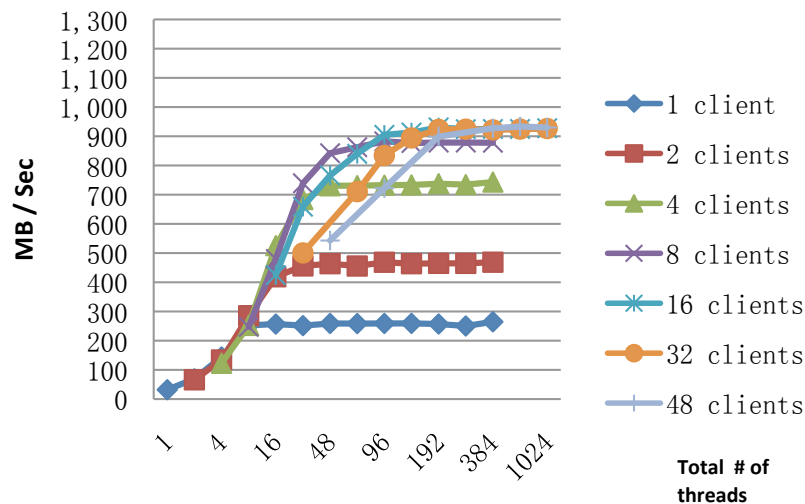
- per-CPU refcount on EQ
- EQ callbacks can happen concurrently on all CPUs
- per-CPU refcount & credit on NI & router-buffer
 - > Open issue: Does it matter to have per-cpu router-buffer and rb-credits?
- smp scalable lnet_selftest

Performance charts

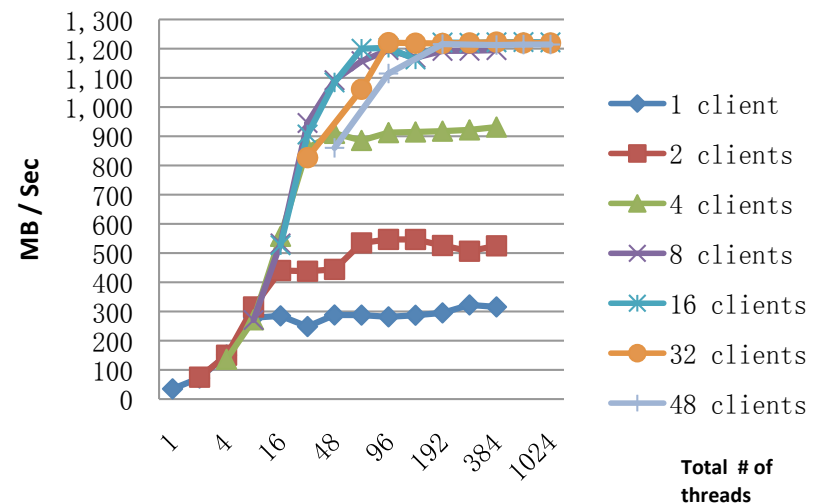
1st ping



1st brw 4K read



1st brw 4K write



Lustre: lu_site & lu_context_key

- better hash for lu_site
 - > We create a very large hash table for lu_site
 - > Most time we only can only hash object (by fid) to 16K-64K buckets
 - We may search for hundreds of times on list if there are Millions of objects on lu_site
- per-site rwlock is replaced by per-bucket spinlock
 - > Per-site rwlock is not good enough, a lot of read/write contention while creating files
- Lu_context_init / lu_context_fini
 - > Per-CPU entry for these APIs

Lustre::ptlrpc

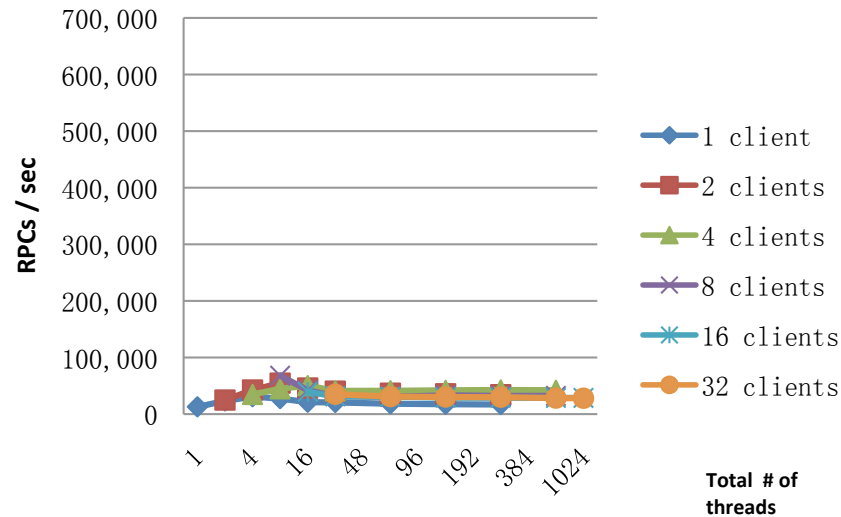
- Per-cpu data for ptlrpc service
 - > Locks, Rqbd, request queue, reply, AT...
- More grained lock for ptlrpc service
 - > Ptlrpc_service::srv_lock is protecting too many different things
 - Although we will have per-cpu service lock, but if we have cross-cpu access, we still have a lot of contention
 - We can have independent locks to protect: a) req_in_queue, b) active_reqs, c) active_replies, d) rqbds, e) adaptive_timeout (already have)
- CPU affinity threads-pool
 - > cfs_waitq_add_exclusive_head is used at here, much less active threads while testing with 128x8 threads

Lustre::ptlrpc (open issue)

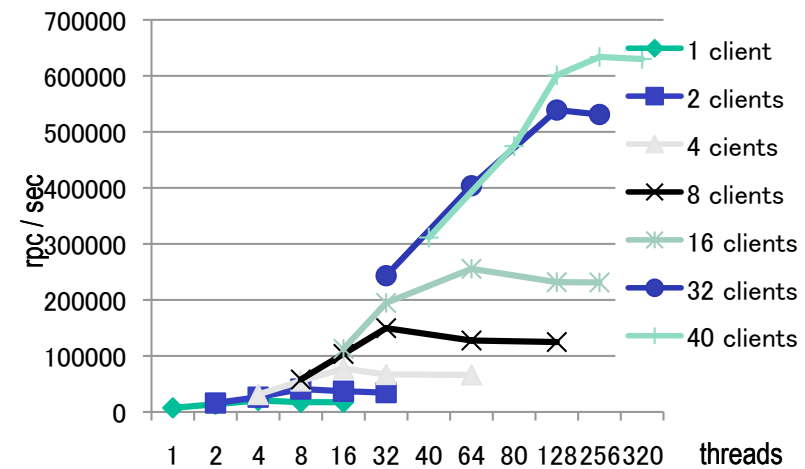
- Preprocessing ptlrpc request in request_in_callback?
 - > It's not a problem if policy is SPTLRPC_POLICY_NULL or SPTLRPC_POLICY_PLAIN, unpacking of them are light-weight and never sleep
 - > sptlrpc_svc_unwrap_request()->gss_svc_accept could have some high-weight operations, even worse, it could sleep at some places
 - Load ptlrpc_gss module
 - create/destroy gss_svc_ctx (for each peer)
 - maintain gss_svc_ctx in rsc_cache (lookup, create, update)
 - gss_verify_msg or gss_unseal_msg need memory allocation

Lustre::ptlrpc (echo performance charts 1/2)

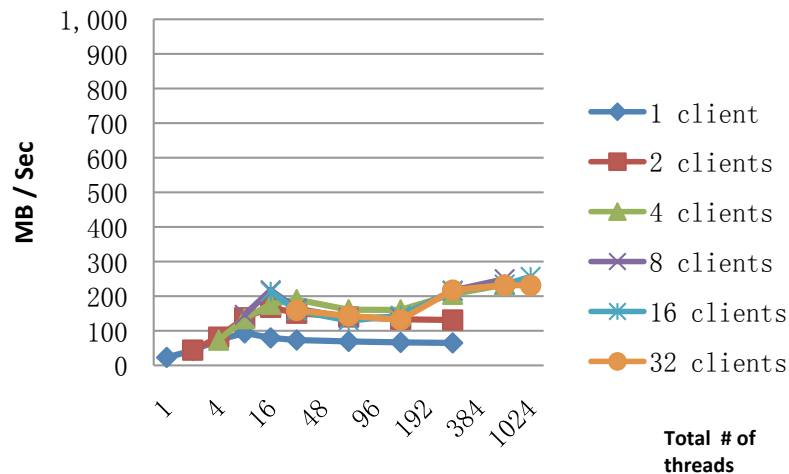
echo getattr



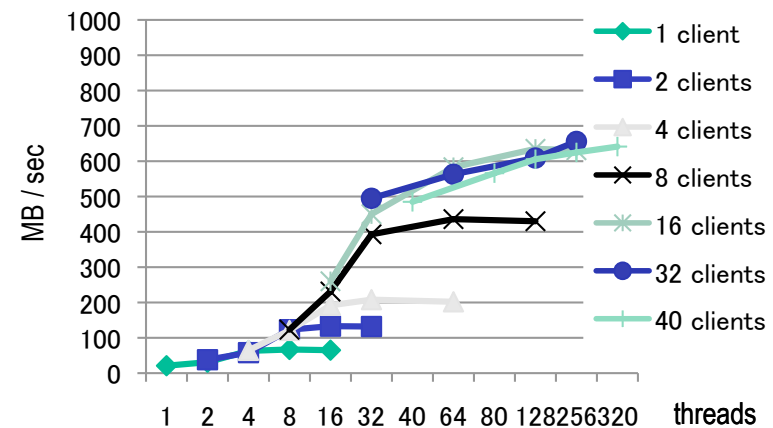
echo getattr



echo brw 4K read

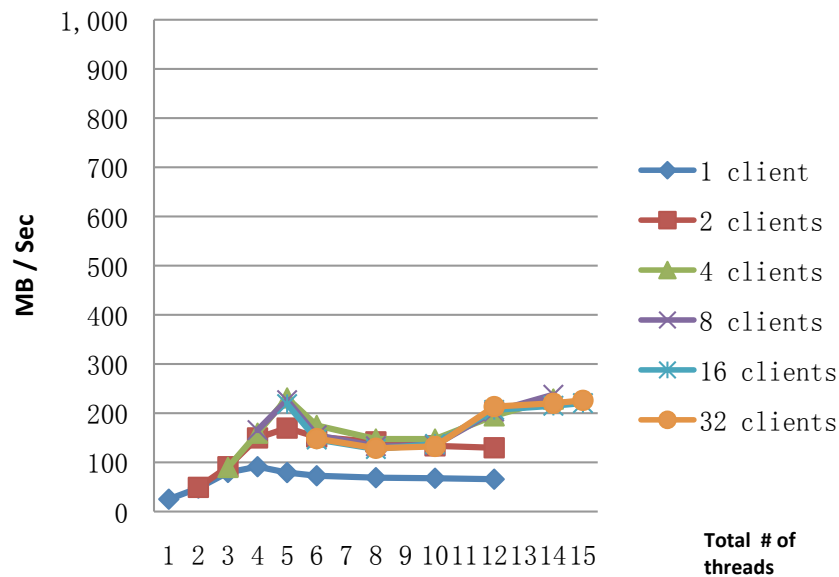


echo 4K read

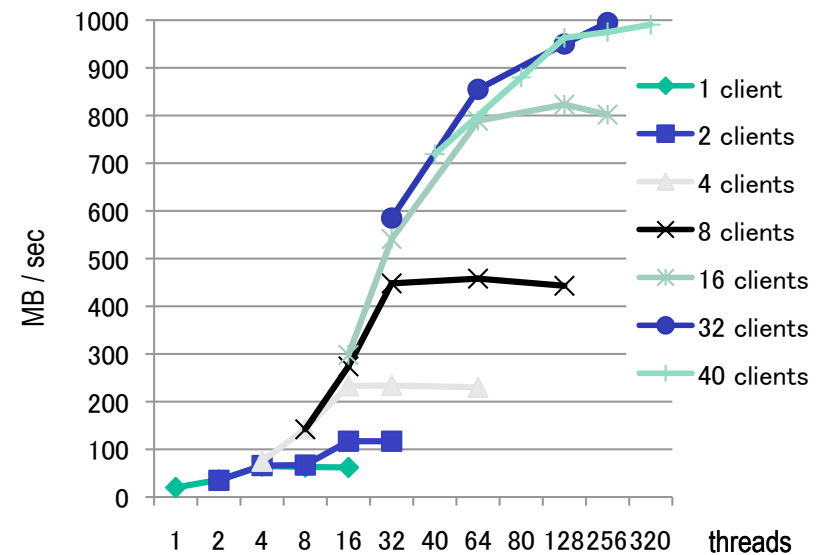


Lustre::ptlrpc (echo performance charts 2/2)

echo brw 4K write



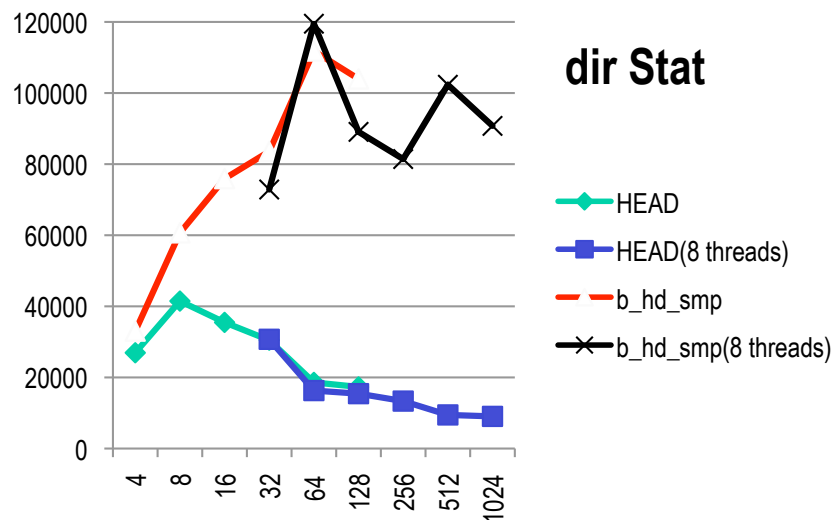
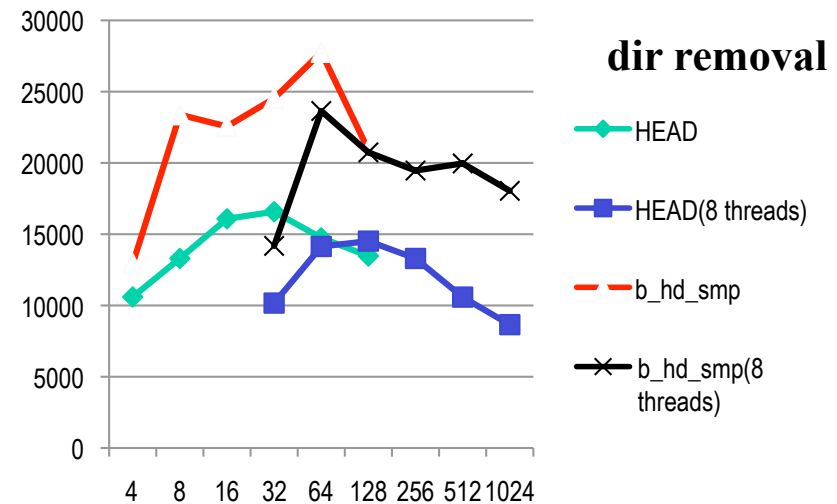
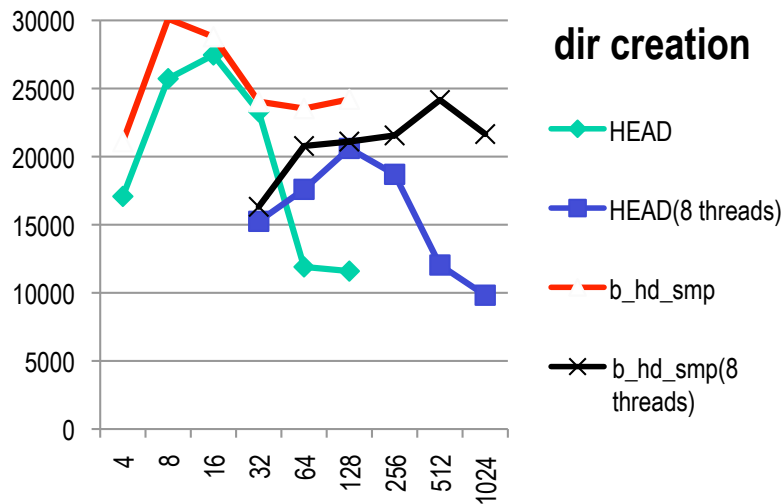
echo 4K write



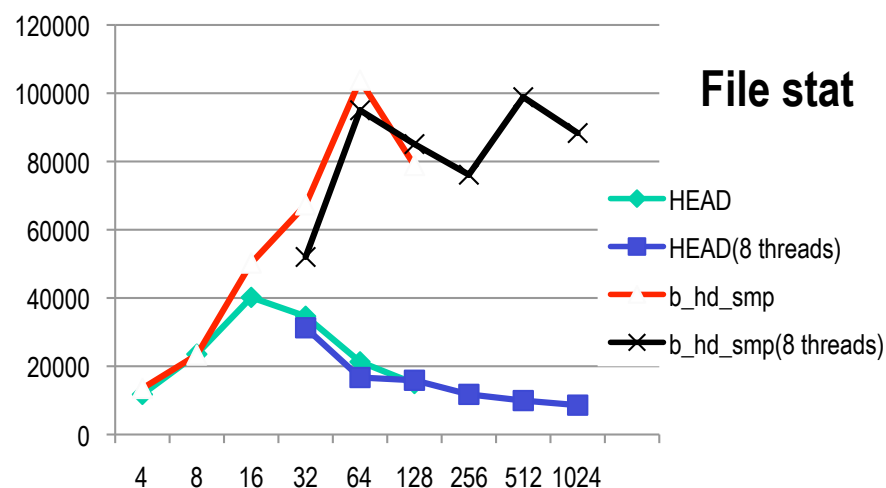
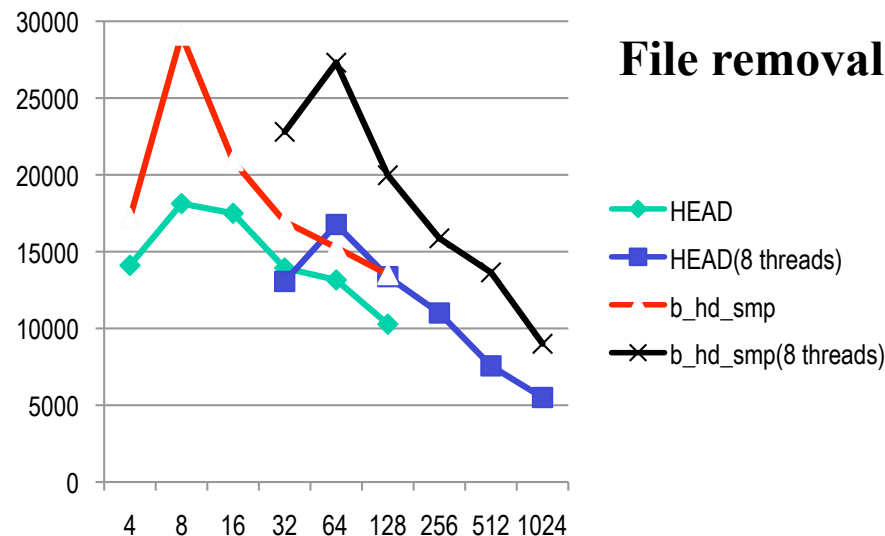
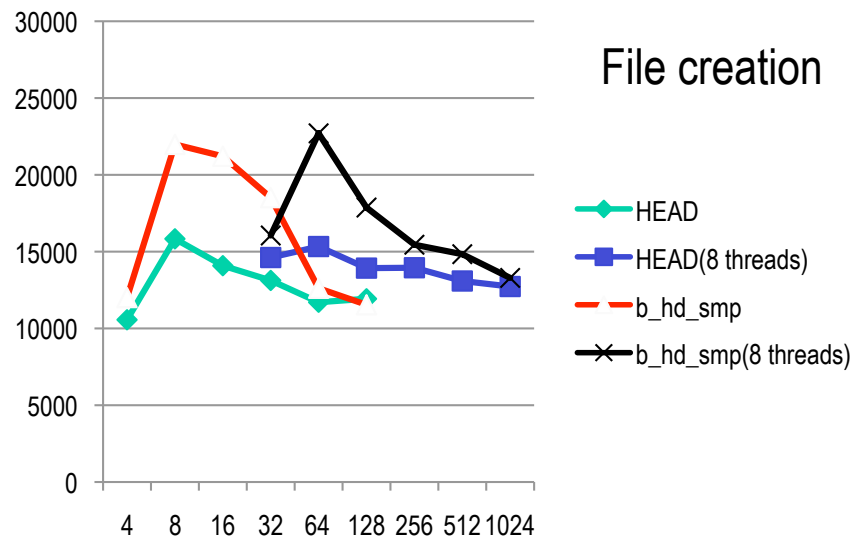
Lustre::ldlm

- Ldlm_namespace
 - > Buckets lock instead of namespace lock
 - > Resources are not well hashed (like lu_site)
- Misc
 - > Cleanup global locks from hot-path
 - > Get rid of unnecessary local lock/unlock dance

Lustre mdtest performance 1/4 (individual directory for each thread)

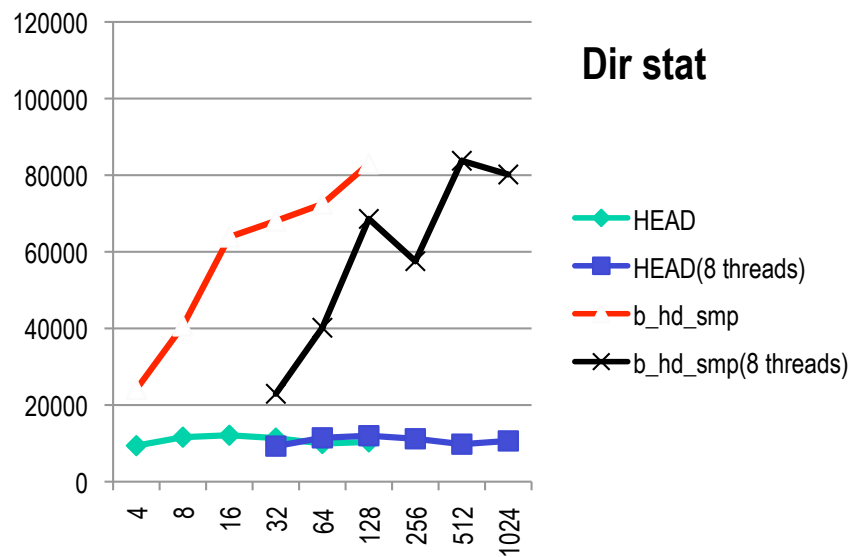
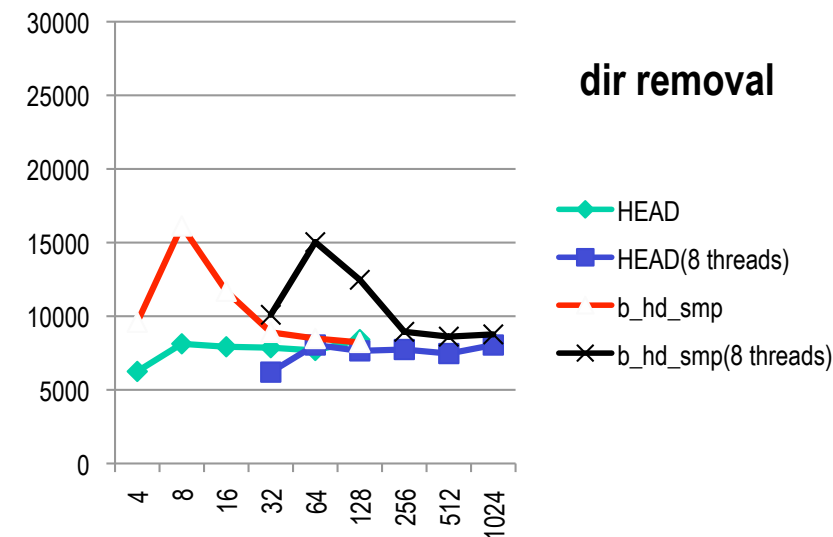
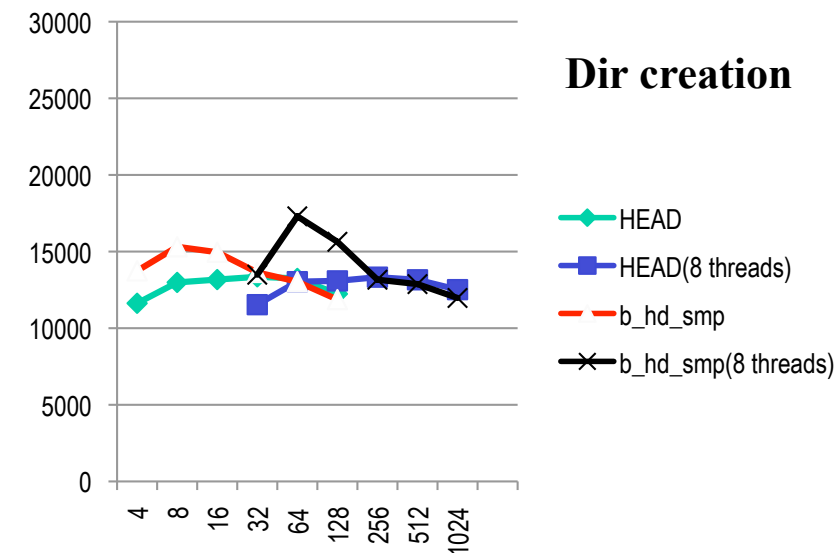


Lustre mdtest performance 2/4 (individual directory for each thread)

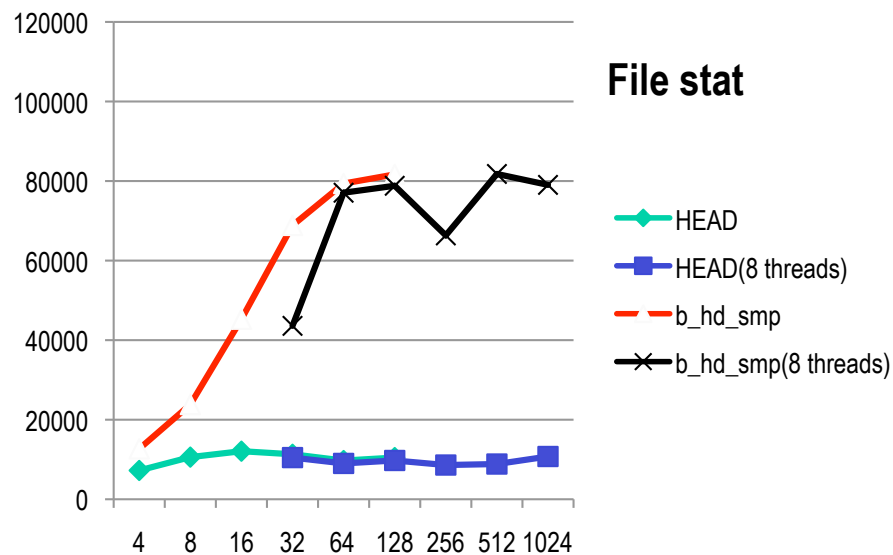
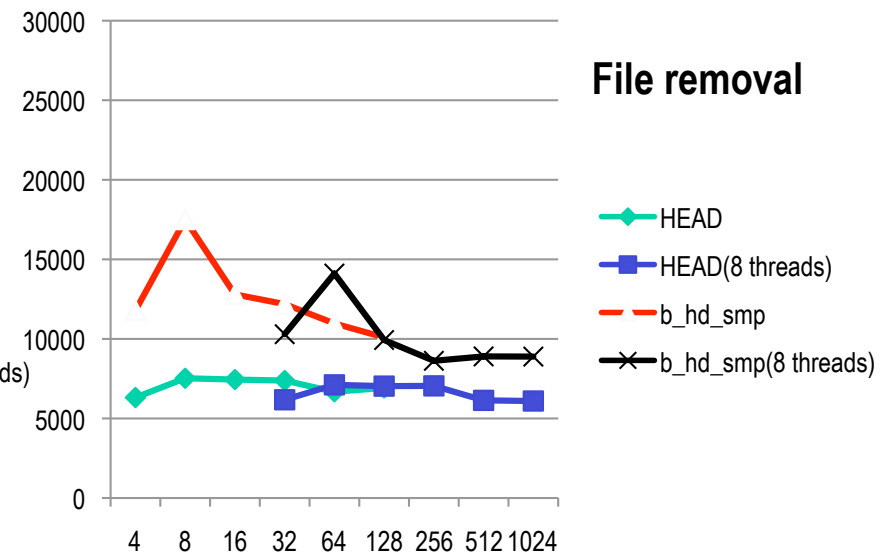
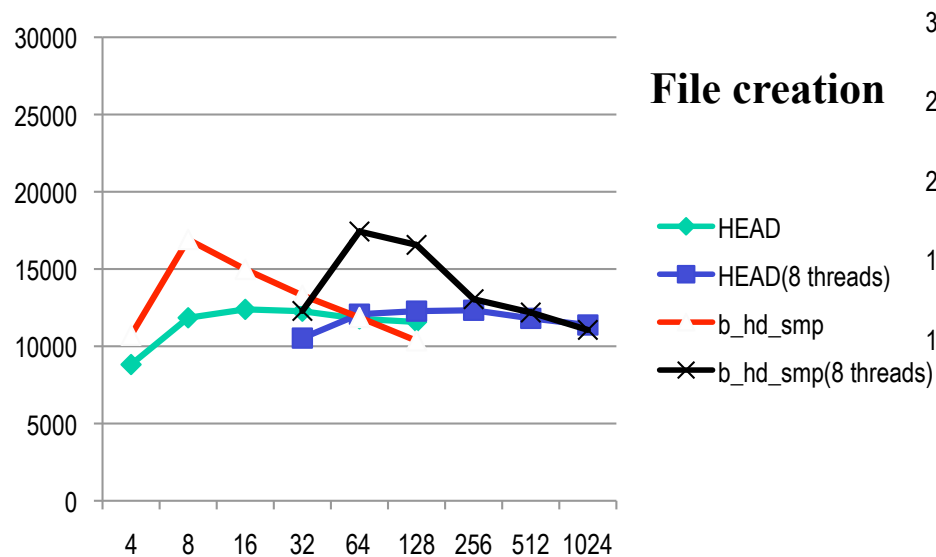


NB: test based on 1 OST, we can see better creation/removal performance if we add another OSS(1 OST)

Lustre mdtest performance 3/4 (shared directory for all threads)



Lustre mdtest performance 4/4 (shared directory for all threads)



Performance in real world is not fancy, why?

- Pdir?
- dynlock_lock?
- Transaction?