Linux multi-core scalability

Oct 2009

Andi Kleen
Intel Corporation
andi@firstfloor.org

Overview

- Scalability theory
- Linux history
- Some common scalability trouble-spots
- Application workarounds

Motivation

- CPUs still getting faster single-threaded
  - But more performance available by going parallel

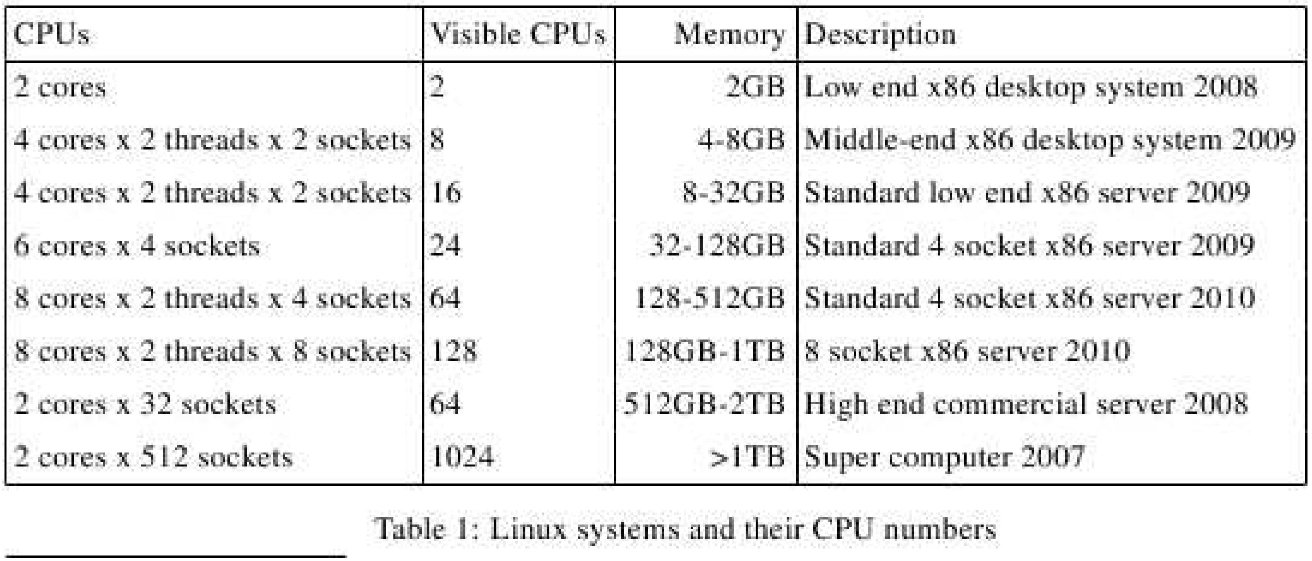
- threaded CPUs dual-core quad-core hexa-core octo-core ...
  - 64-128 logical CPUs on standard machines upcoming
    - Cannot cheat on scalability anymore
  - High end machines larger
    - Rely on limited workloads for now

- Memory sizes are growing
  - Each CPU thread needs enough memory for its data (~1GB/thread)
  - Multi-core servers support a lot of memory (64-128GB)
    - Servers systems going towards TBs of RAM maximum
  - Large memory size is a scalability problem
    - Especially with 4K pages
    - Some known problems in older kernels ("split LRU")

Terminology

- Cores
  - Core inside a CPU
- Threads (hardware)
  - Multiple logical CPU per threaded core
- Sockets
  - CPU package
- Nodes
  - NUMA node with same memory latency

Systems



CPUs	Visible CPUs	Memory	Description
2 cores	2	2GB	Low end x86 desktop system 2008
4 cores x 2 threads x 2 sockets	8	4-8GB	Middle-end x86 desktop system 2009
4 cores x 2 threads x 2 sockets	16	8-32GB	Standard low end x86 server 2009
6 cores x 4 sockets	24	32-128GB	Standard 4 socket x86 server 2009
8 cores x 2 threads x 4 sockets	64	128-512GB	Standard 4 socket x86 server 2010
8 cores x 2 threads x 8 sockets	128	128GB-1TB	8 socket x86 server 2010
2 cores x 32 sockets	64	512GB-2TB	High end commercial server 2008
2 cores x 512 sockets	1024	>1TB	Super computer 2007

Table 1: Linux systems and their CPU numbers

Laws

- Amdahl’s law:
  - Parallelization speedup limited by performance of serial part
- Amdahl assumes that data set size stays the same
- In practice we tend to be more guided by Gustafson’s law
  - More cores/memory allow to process larger datasets
  - Easier more coarse grained parallelization

Parallelization classification

- Single job improvements
  - For example weather model
  - Parallelization of long running algorithm
  - Not covered here

- "Library style" / "server style" of tuning
  - Providing short lived operations for many parallel users
  - Typical for kernels, network servers, some databases (OLTP)
    - "requests" "syscalls" "transactions"
  - Key is to parallelize access to shared data structures
    - Let individual operations run independently
  - Usually no need to parallelize inside individual operations

Parallel data access tuning stages

Goal: Let threads run independent

- Code locking "first step"
  - One single lock per subsystem acquired by all code
  - Limits scaling
- Coarse grained data locking "lock data not code"
  - More locks: object locks, hash table lock
  - Reference counters to handle object lifetime
- Fine grained data locking (optional)
  - Even more locks (multiple per object)
  - Per bucket lock in a hash
- Fancy locking (only for critical paths)
  - Minimize communication (avoid false sharing)
  - per-CPU data
  - NUMA locality
  - Lock less: relying on ordered updates, Read-Copy-Update (RCU)

Communication latency

- For highly tuned parallel code often latency is the limiter
  - Time to bounce the lock/refcount cache line from core A to B
    - Cost depends on distance
  - Adds up with fine-grained locking
  - Physical limitations due to signal propagation delays
  - Solution is to localize data or do less locks

- Good news is that in the multi core future latencies are lower
  - Compared to traditional large MP systems

- Multi-core has very fast communication inside the chip
  - "shared caches"
  - Modern interconnects are faster, lower latency
    - But going off-chip is still very costly
  - Lower latencies tolerate more communication
  - Modern multi-core system of equivalent size is easier to program

Problems & Solutions

- Parallelization leads to more complexity, more bugs
  - Adds overhead for single thread
  - Better debugging tools to find problems
    - lockdep, tracing, kmemleak
- Locks, atomic operations add overhead
  - Atomic operations are slow and synchronization costs
  - Number of locks taken for simple syscalls high and growing
- Compile time options (for embedded), code patching
  - Problem: small multi-core vs large MP system
  - Still doesn’t solve inherent complexity
- Lock less techniques (help scaling, but even more complex)
- Code patching for atomic operations

The locking cliff

- Still could fall off the locking cliff
  - Overhead of locking, complexity gets worse with more tuning
  - Can make further development difficult

- Sometimes solution is to not tune further
  - If use case is not important enough
  - Or speedup not large enough

- Or use new techniques
  - lock-less approaches
  - Radically new algorithms

Linux scalability history

- 2.0 big kernel lock for everything
- 2.2 big kernel lock for most of kernel, interrupts own locks
  - First usage on larger systems (16 CPUs)
- 2.4 more fine grained locking, still several common global locks
  - a lot of distributions back ported specific fixes
- 2.6 serious tuning, ongoing
  - New subsystems (multi queue scheduler, multi flow networking)
  - Very few big kernel lock users left
  - A few problematic locks like dcache, mm_sem
  - Advanced lock-less tuning (Read-Copy-Update, others)
- For more details see paper

Big Kernel Lock (BKL)

- Special lock that simulates old "explicit sleeping" semantics
  - Still some users left in 2.6.31
  - But usually not a serious problem (except on RT)

- File descriptor locking (flock et.al.)
- Some file systems (NFS, reiser)
- ioctls, some drivers, some VFS operations

- Not worth fixing for old drivers

VFS

- In general most IO is parallel
  - Depending on the file system, block driver

- namespace operations (dcache, icache) still have code locks
  - When creating path names for example
  - inode_lock / dcache_lock
  - Some fast paths in dcache (nearly) lock-less when nothing changes
    - Read only open faster
    - Still significant cache line bouncing
    - Can significantly limit scalability

- Effort under way to fine grain dcache/inode locking
  - Difficult because lock coverage is not clearly defined
  - Adds complexity

Memory management scaling

- In general scales well between processes
  - On older kernels make sure to have enough memory/core

- Coarse grained locking inside a process (struct mm_struct)
  - mm_sem semaphore to protect virtual memory mapping list
  - page_table_lock to protect page tables
  - Problems with parallel page faults, parallel brk/mmap

- mm_sem is a sleeping lock
  - Most page fault operations (including zeroing) hold
  - Convoying problems

- Problem for threaded HPC jobs, postgresql

Network scaling

- 1Gbit/s can be handled by single core on PC class
  - ... unless you use encryption
  - But 10Gbit/s still challenging
- Traditional single send queue, single receive queue per network card
  - Serializes sending, receiving
- Modern network cards support multi-queue
  - Multiple send (TX) queues to avoid contention while sending
  - Multiple receive (RX) queues to spread flows over CPUs
- Ongoing work in the network stack for better multi queue
  - RX spreading requires some manual tuning for now
  - Not supported in common production kernels (RHEL5)

Application workarounds I

- Scaling a non parallel program
  - Use Gustafson’s law! Work on more data files
  - gcc: make -j$(getconfig _NPROCESSORS_ONLN)
    - Requires proper Makefile dependencies
  - media encoder for more files:
    - find -name ’*.foo’ | xargs -n1 -P$(getconf _NPROCESSORS_ONLN) encoder
  - Renderer:
    - render multiple pictures

- Multi threaded program that does not scale to system size
  - For example popular open source database
  - Limit parallelism to its scaling limit
    - Requires load tests to find out
  - Possibly run multiple instances

Application workarounds II

- Run multiple instances ("cluster in a box")
  - Can use containers or virtualization
  - Or just use multiple processes

- Run different programs on same system
  - "server consolidation"
  - Saves power and is easier to administrate
  - Often more reliable (but single point of failure too)

- Or keep cores idle until needed
  - Some spare capacity for peak loads is always a good idea
  - Not that costly with modern power saving

Conclusions

- Multi-core is hard

- Linux kernel is well prepared
  - but still some more work to do

- Application tuning is the biggest challenge
  - Is your application well prepared for multi-core?

- Standard toolbox of tuning techniques available

Resources

- Paper: http://halobates.de/lk09-scalability.pdf
  - Has more details in some areas
- Linux kernel source
- A lot of literature on parallelization available
- andi@firstfloor.org

Backup

Parallelization tuning cycle

- Measurement
  - Profilers: oprofile, lockstat
- Analysis
  - Identify locking, cache line bouncing hot spots
- Simple tuning
  - Move to next tuning stage
- Measure again
  - Stop or repeat with fancier tuning