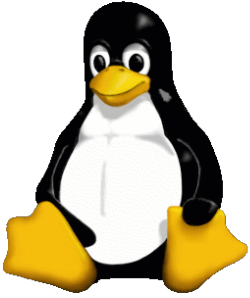
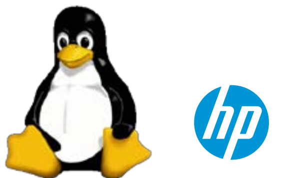


Optimizing Application Performance in Large Multi-core Systems

Waiman Long / Aug 19, 2015
HP Server Performance & Scalability Team
Version 1.1

Agenda

1. Why Optimizing for Multi-Core Systems
2. Non-Uniform Memory Access (NUMA)
3. Cacheline Contention
4. Best Practices
5. Q & A



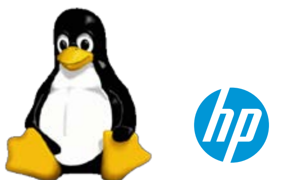
 This talk focuses mainly on performance related issues on x86 processors, though many of the lessons can be equally applied to other CPU architectures.

CPU Core Counts are Increasing

- The table below shows the progression in the maximum number of cores per CPU for different generations of Intel CPUs.

CPU Model	Max Core Count	Max thread Count in a 4P Server
Westmere	10	80
IvyBridge	15	120
Haswell	18	144
Broadwell	24	192
Skylake	28	224
Knight Landing	72	1152 (4 threads/core)

- Massive number of threads will be available for running applications.
- The question now is how to make full use of all these computing resources. Of course, virtualization and containerization are all useful ways to use them up. Even then, the typical size of a VM guest or container is also getting bigger and bigger with more vCPUs in it.



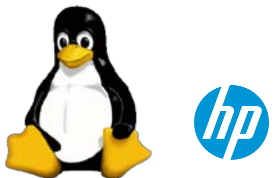
Multi-threaded Programming

This talk is **NOT** about how to do multi-threaded programming. There are a lot of resources available for that. Instead, it focuses mainly on the following 2 topics that have big impact on multi-threaded application performance on a multi-socket computer system:

1. NUMA-awareness
   Non-Uniform Memory Access (NUMA) means memory from different locations may have different access times. A multi-threaded application should try to access as much local memory as possible for the best possible performance.

2. Cacheline contention
   When two or more CPUs try to access and/or modify memory locations in the same cacheline, the cache coherency protocol will work to serialize the modification and access to ensure program correctness. However, excessive cache coherency traffic will slow down system performance by delaying operation and eating up valuable inter-processor bandwidth.

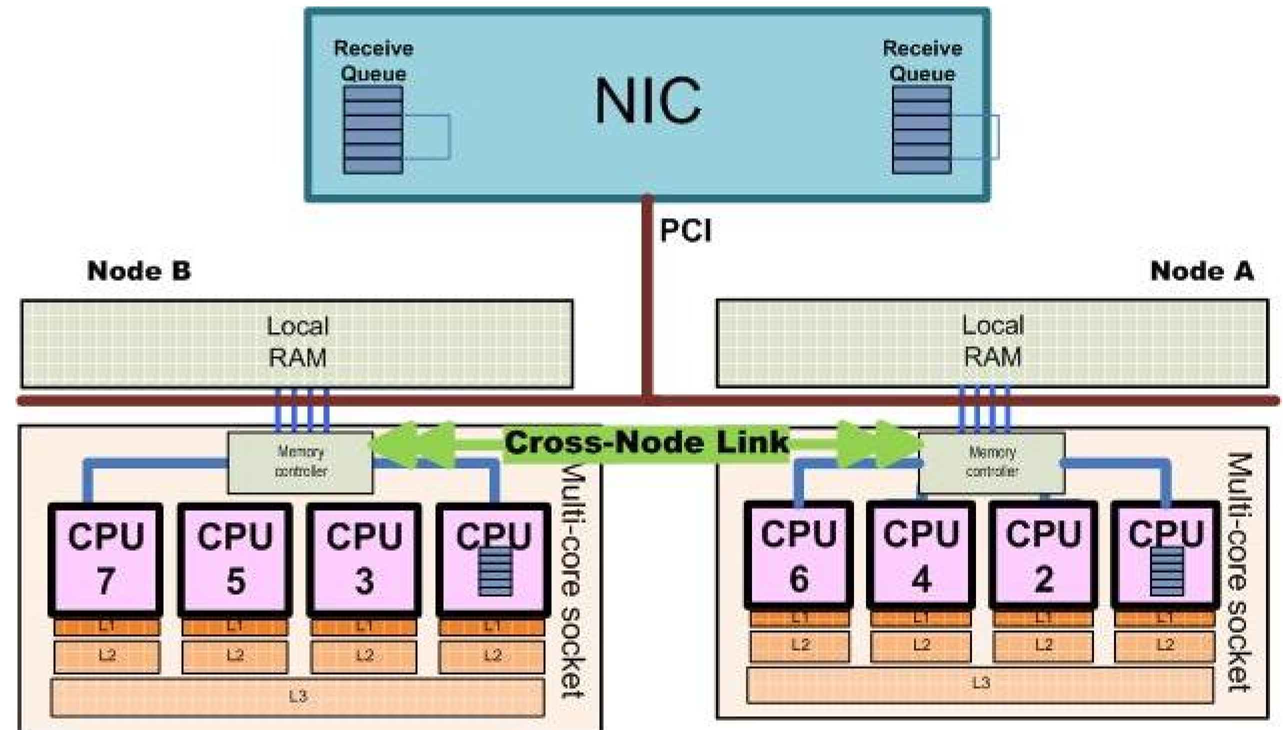
As long as a multi-thread application can be sufficiently parallelized without too much inter-thread synchronizations (as limited by the Amdahl's law), most of the performance problems we observed are due to the above two problems.

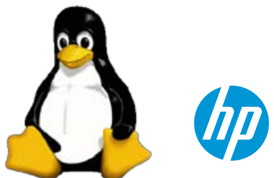


Non-Uniform Memory Access (NUMA)

Non-Uniform Memory Access (NUMA)

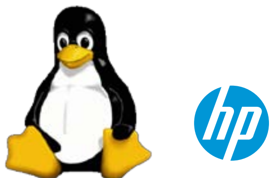
- Access to local memory is much faster than access to remote memory.
- Depending on the way the processors are interconnected (glue-less or glued), remote memory access latency can be two times or even three times as slow as local memory access latency.
- Inter-processor links are not just for memory traffic, it can be used for I/O and cache coherency traffic. So the bandwidth can also be smaller than from local memory.
- For an application that is memory bandwidth constrained, it may run up to 2-3 times slower when most of the memory accesses are remote instead of local.
- For a NUMA-blind application, the higher the number of processor sockets, the higher the chance of remote memory access leading to poorer performance.





NUMA Support in Linux

- On boot-up, the system firmware communicates the NUMA setup of the system by using ACPI (Advanced Configuration & Power Interface) tables.
- How memory is allocated under NUMA is controlled by the memory policy which can be grouped into two main types:
  1. Node local – allocation happens in the same node as the running process
  2. Interleave – allocation occurs round-robin over all the available nodes
- The default is node local after initial boot-up to ensure optimal performance for processes that don't need a lot of memory.
- The exact memory policy of a block of memory can be chosen with the mbind(2) system calls.
- The process-wide memory policy can be set or viewed with the set_mempolicy(2) and get_mempolicy(2) system calls.
- The NUMA memory allocation of a running process can be viewed from the file `/proc/<pid>/numa_maps`.

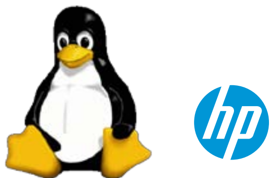


Linux Thread Migration

- By default, Linux kernel performs load balancing by moving threads from the busiest CPUs to the most idle CPUs.
- Such thread migrations is usually good for overall system performance, but may disrupt cache and memory locality of a running application affecting its performance.
- Migration of a task from one CPU to another CPU of the same socket won't usually have too much impact other than the need to refill the L1/L2 caches. It should have no effect on memory locality.
- Migration of a task from one CPU to anther CPU on a different socket, however, can have a significant adverse effect on its performance.
- To avoid this kind of disruption, the usual practice is to bind the thread to a given socket or NUMA node. This can be done by:
  - Use sched_setaffinity(2) for process or pthread_setaffinity_np(3) for thread, and taskset(1) from the command line.
  - Use cgroups like cpuset(7) to constrain the set of CPUs and/or memory nodes to use.
  - Use numactl(8) or libnuma(3) to control NUMA policy for processes/threads or shared memory.
- Before that, the application must be able to figure out the NUMA topology of the system it is running in. On the command level lscpu(1) can be used to find out the number of nodes and the CPU numbers on each of them. Alternatively, the application can parse the sysfs directory /sys/devices/system/node to find out how many CPUs and their numbers in each node.

Automatic NUMA Balancing (AutoNUMA)

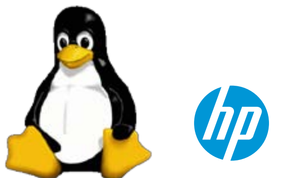
- Newer Linux kernels (3.8 or later) has a scheduler feature called Automatic NUMA Balancing which, when enabled, will try to migrate the memory associated with the processes to the nodes where those processes are running.
- Depending on the applications and their memory access pattern, this feature may help or hurt performance. So the mileage can vary. You really need to try it out to see if it helps.
- For long running processes with infrequent node-to-node CPU migration, AutoNUMA should be able to help improving performance. For relatively short running processes with frequent node-to-node CPU migration, however, AutoNUMA may hurt.
- AutoNUMA usually does not perform as good as with explicit NUMA policy from the numactl(8) command, for example.
- For applications that are fully NUMA-aware and do their own balancing, it is usually better to turn this feature off.



Cacheline Contention

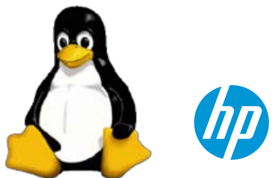
Cache Coherency Protocols

- In a multi-processor system, it is important that all the CPUs have the same view on all the data in the system no matter if the data reside in memory or in caches.
- The cache coherency protocol is the mechanism to maintain consistency of shared resource data that ends up stored in multiple local caches.
- Intel processors use the MESIF protocol which consists of five states: Modified (M), Exclusive (E), Shared (S), Invalid (I) and Forward (F).
- AMD processors use the MOESI protocol which consists of five states: Modified (M), Owned(O) Exclusive (E), Shared (S) and Invalid (I).
- 2-socket and sometimes 4-socket systems can use snooping/snarfing as the coherency mechanism. Larger system typically use directory-based coherency mechanism as it scales better than the others.
- Many performance problems of multi-threaded applications, especially on large systems with many sockets and cores, are caused by cacheline contention due to either true and/or false sharing.
- True cacheline sharing is when multiple threads are trying to access and modify the same data.
- False cacheline sharing is when multiple threads are trying to access and modify different data that happen to reside in the same cacheline.



Impact of Cacheline Contention

- To illustrate the impact of cacheline contention, two type of spinlocks are used – ticket spinlock and queued spinlock.
- Ticket spinlock is the spinlock implementation used in the Linux kernel prior to 4.2. A lock waiter gets a ticket number and spin on the lock cacheline until it sees its ticket number. By then, it becomes the lock owner and enters the critical section.
- Queued spinlock is the new spinlock implementation used in 4.2 Linux kernel and beyond. A lock waiter goes into a queue and spins in its own cacheline until it becomes the queue head. By then, it can spin on the lock cacheline and attempt to get the lock.
- For ticket spinlocks, all the lock waiters will spin on the lock cacheline (mostly read). For queued spinlocks, only the queue head will spin on the lock cacheline.
- The charts in the next 4 pages show the 2 sets of locking rates (the total number of lock/unlock operations that can be performed per second) as reported by a micro-benchmark with various number of locking threads running. The first set is with an empty critical section (no load) whereas the second set has an atomic addition in the same lock cacheline in the critical section (1 load). The test system was a 16-socket 240-core IvyBridge-EX (Superdome X) system with 15 cores/socket and hyperthreading off.



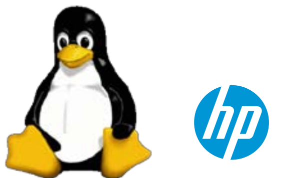
Ticket Lock vs. Queued Lock (2-20 Threads, No Load)

Locking Rate (millions/s)

25
20
15
10
5
0

0 2 4 6 8 10 12 14 16 18 20

Ticket Lock
Queued Lock



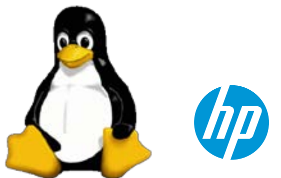
Ticket Lock vs. Queued Lock (16-240 Threads, No Load)

Locking Rate (millions/s)

7
6
5
4
3
2
1
0

0 15 30 45 60 75 90 105 120 135 150 165 180 195 210 225 240

Ticket Lock
Queued Lock



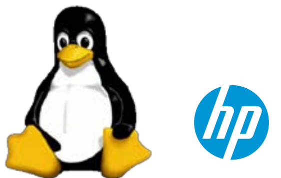
Ticket Lock vs. Queued Lock (2-20 Threads, 1 Load)

Locking Rate (Millions/s)

16
14
12
10
8
6
4
2
0

0 2 4 6 8 10 12 14 16 18 20

Ticket Lock   Queued Lock



Ticket Lock vs. Queued Lock (16-240 Threads, 1 Load)

Locking Rate (Millions/s)

7
6
5
4
3
2
1
0

0 15 30 45 60 75 90 105 120 135 150 165 180 195 210 225 240

Ticket Lock — Queued Lock

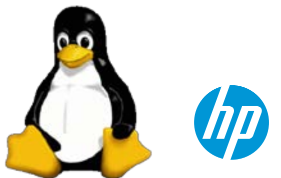
Ticket Lock (16-240 Threads, No Load vs. 1 Load)

Locking Rate (Millions/s)

2.4
2.2
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0

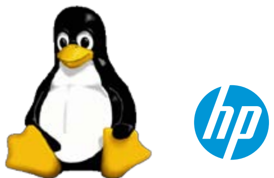
0 15 30 45 60 75 90 105 120 135 150 165 180 195 210 225 240

Ticket Lock (No Load) Ticket Lock (1 Load)



True Cacheline Sharing

- The previous pages shows an example of true cacheline sharing where all the threads need to acquire the same lock before going into their critical sections.
- With ticket spinlocks, all the lock waiters are spinning on the lock cacheline to wait for their turns. The higher the number of threads spinning , the slower the performance will be. Also, if the lock holder is doing some additional read/write operations on the same lock cacheline, there will be more drop in performance.
- With queued spinlocks, all the lock waiters except the queue head are spinning on their own cacheline. Within a single socket, the locking performance is essentially flat irrespective of the number of threads and the amount of additional load on the lock cacheline. Crossing the socket boundary, however, causes a drop in performance primarily due to the fact the cacheline transfer latency is much higher for two cores on different sockets than on the same socket. Increasing thread count causes increases in the proportion of inter-socket cacheline line transfer versus intra-socket cacheline transfer.



False Cacheline Sharing

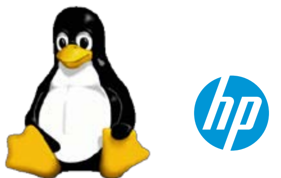
Average Execution Time (in second)

7
6
5
4
3
2
1
0

0 10 20 30 40 50 60 70 80

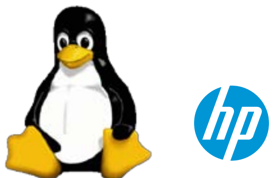
To illustrate the effect of false cacheline sharing, the chart at the left hand side shows the average thread execution time of a multi-thread micro-benchmark where each thread executes 10 million read-modify-write operations on the same cacheline as every other threads.

The execution time increases from 29.7ms for 1 thread, 191ms for 10 threads and up to 6520ms for 80 threads on a 8-socket 80-core Westmere-EX system. This is an increase of 220X.



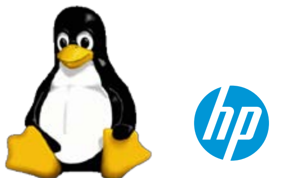
Indirect Measurement of Cacheline Contention

- The Linux perf performance analysis tool can be used to detect cacheline contention in the code.
- Use “perf record –e cycles:pp” to record its execution profile. PEBS (Precise Event Based Sampling) should be used, if supported, to have more accurate data on where the performance bottleneck is. With the perf profile data, use the annotate function of the “perf report” command to see how much times are spent in each instruction of the selected function.
- The next page has a portion of the sample annotation output for the mutex_spin_on_owner() function in the Linux kernel.
- In the annotated output, more than half of the cycles are spent by the “cmp” instruction before the “je”. With debuginfo data, that corresponds to the access of “lock->owner” variable.
- Having so much time spent on one instruction is an indication that the cacheline pointed to by the “lock” variable is heavily contended.
- By looking for instructions that consume a disproportionate amount of cycles time, we can have a pretty good idea of which cachelines are the performance bottleneck. We can then look for ways to reduce the contention and improve the performance of the applications.



Sample Perf Command Annotation Output (mutex_spin_on_owner)

```
       │      static __always_inline int constant_test_bit(unsigned int nr, const
       │      {
       │              return ((1UL << (nr % BITS_PER_LONG)) &
       │                      (addr[nr / BITS_PER_LONG])) != 0;
  1.48 │2f:   mov     -0x3fc8(%rcx),%rdx
       │                      if (need_resched())
  5.33 │      and     $0x8,%edx
       │    ↓ jne     43
       │      }
       │
       │      /* REP NOP (PAUSE) is a good thing to insert into busy-wait loops.
       │      static inline void rep_nop(void)
       │      {
       │              asm volatile("rep; nop" ::: "memory");
 31.46 │      pause
       │       * Mutex spinning code migrated from kernel/sched/core.c
       │       */
       │
       │      static inline bool owner_running(struct mutex *lock, struct task_st
       │      {
       │              if (lock->owner != owner)
  0.33 │      cmp     %rax,0x18(%rdi)
 58.06 │    ↑ je      28
```



Direct Measurement of Cacheline Contention

- There is an outstanding Linux kernel patch from Red Hat that adds a new feature called c2c (cache-to-cache) to the perf command - http://lwn.net/Articles/588866/.
- This new feature will enable direct measurement of cacheline contention of an application on a NUMA system.
- Sample command and partial output from the tool:

```
# perf c2c record -g ./futextest
# perf c2c report
  …
                           Shared Data Cache Line Table

                                        Total      %All                 Total
   Index              Phys Adrs      Records    Ld Miss       %hitm     Loads
=====================================================================
       0   0xffffc900275e6e00       248334     55.76%      67.49%    225428
       1             0x60a000          885      6.44%       7.79%       885
       2             0x60a000       214492      6.25%       7.56%    114654

                     Shared Cache Line Distribution Pareto

     ---- All ----   -- Shared --    ---- HITM ----
      Data Misses   Data Misses   Remote     Local   -- Store Refs --

 Num  %dist  %cumm  %dist  %cumm  LLCmiss   LLChit   L1 hit  L1 Miss       Data Address    Pid
===============================================================================================
-------------------------------------------------------------------------------------------------
   0  55.8%  55.8%  67.5%  67.5%     3569     1470    19563     3343 0xffffc900275e6e00  99649
-------------------------------------------------------------------------------------------------
```

Best Practices

Best Practices for NUMA and Cacheline Contention

- It is much faster to access local memory than remote memory from another NUMA node (socket).
- Cacheline contention within a NUMA node is also much less severe than among cores from different NUMA nodes.
- The followings are some of the best practices to build a cacheline contention and NUMA-aware multi-thread application:
  1. Partitions the data set into largely independent sub-groups each of which would then be put into memory of one NUMA node.
  2. Schedules one or more worker threads to process data for each of the sub-groups and bind them to the same NUMA node that has the data. This will prevent the tasks from being migrated to a different NUMA node causing all kind of performance issue. For maximum performance on a system with *n* threads per NUMA node, you will have to schedule at least *n* worker threads. You may need to schedule more if the worker threads have significant I/O wait time.
  3. Be aware of the cacheline placement of the variables used by each thread. X86 CPUs have a cacheline size of 64 bytes. For gcc, you can append the type attribute "__attribute__ ((__aligned__(64)))" to force a variable to be aligned on 64-byte boundary.
  4. Use cgroup (e.g. cpusets) to constraint the application to different NUMA node sizes to measure node scaling and use performance measuring tools like perf to look for performance bottleneck.

Case Study: Oracle Database

- The System Global Area (SGA) in an Oracle database is a group of shared memory areas that act as cache and hold various kind of information.
- In the past, Oracle recommended turning off NUMA support for their database for better performance. That is no longer true with the latest database versions and Linux OSes which did a better job of supporting NUMA. You may not see too much performance difference in 2-socket systems. Starting from 4-socket systems and above, enabling NUMA should have a positive performance impact.
- The _enable_NUMA_support parameter should be set to “True” to enable NUMA support in Oracle. There are a few visible changes after turning this parameter on.
  1. There is one SGA shared memory region per NUMA node instead of one or more SGAs each of which contains pages from multiple nodes.
  2. Each of the log writer processes will bind to a particular node and use the SGA from this node.
- There may also be other less visible changes under the hood.
- The more nodes or sockets the system have, the bigger the performance improvement will be. On a 8-socket system, you can see up to 10-20% performance improvement.
- Other changes that improve performance include using huge pages (hugetlbfs) and multiple instances each of which runs on separate sets of NUMA nodes.

Thank you

Q & A

Waiman.Long@hp.com

Backup Slides

Linux Kernel Performance Patches

The HPS Linux kernel performance & scalability team are responsible for solving Linux kernel performance and scalability problems to improve application performance.

Below are some of the Linux kernel performance patches that our team had helped merging into the upstream kernel over the years:

- Mutex performance tuning & MCS-lock based optimistic spinning.
- R/W semaphore performance tuning & MCS-lock based optimistic spinning.
- Lockref (lockless reference count update) in dentry cache.
- Queued spinlocks and rwlocks
- Finer granularity locking in System V IPC mechanism & huge page instantiation code.
- Better idle balancing code & other scheduling related performance fixes.
- Futex performance tuning and fixes