NUMA aware locks Implementation and Evaluation

Zhongyu Ma

ScM research project Spring 2012

1 Introduction

Programs running on NUMA machines are sensitive to memory access locality[3]. Acessing data on the local memory is significantly faster than remote memory. Thus, designing locks that can take advantage of this property would improve the performance. We review three papers related to this topic and study the behaviors of these NUMA aware locks. We implement these locks in C++, and evaluate them on the two different NUMA platforms–intel and sparc. We compared these locks each other and with non NUMA aware locks in terms of performance and fairness.

2 Implementation

Generally, all the locks that we implemented are spin locks, including backoff locks and queue locks. First, we built the Backoff TTAS(Test and Test and Set) lock, as well as the CLH and MCS queue lock. We built these simple locks not only because some of the NUMA aware locks are assembled by them, it is also interesting to compare the NUMA aware locks with them. Some NUMA aware lock use only the backoff lock or the queue lock, others comprise both these two kind of locks. In the following section, we describe the algorithms of these the NUMA aware locks and how we implemented them in detail.

2.1 Hierarchical Backoff Lock

The HBO locks are designed to take into account the architecture's memory hierarchy and access costs[5]. It is refined by the following three steps:

- The basic idea is simple, if a thread needs to backoff, it first checks whether it is on the node that is currently holding the lock. If so, it backoff a smaller constant, otherwise it backoff a larger constant. In this manner, the thread on the lock holding node is more likely to subsequently acquire the lock.
- Letting all the threads competing for the lock somehow increases the global coherence traffic overhead. To reduce it, the algorithm force the threads except the first one spinning on a local memory.
- It's possible to get starvation because some threads may be granted the lock repeatedly due to the short backoff constant. To improve the fairness, priority is introduced. If a thread fails to obtain the lock in certain times, it gets "angry". The angry node has the ability to prevent the coming threads on the cluster that holding the lock to compete the lock.

Corresponding to these three steps, we define three classes hierarchically. Here we only describe the last class. The class has a volatile integer member–state to indicate whether the lock is free or is obtained by a certain thread. Each thread maintains a thread local variable–node_id to record which node is the thread running, the "state" equals to the "node_id" of the thread holding the lcok. In the acquire method, the thread first try to CAS-ed the ”state” with its ”node_id”, if the lock was free right before this operation, the thread claims it obtain the lock and return from the acquire method. Otherwise the thread needs to backoff. The previous CAS operation return the ”node_id”, the id of the node that currently owns the lock. The thread compare this value with its own ”node_id”. If they are the same, the thread backoffs a small constant. Otherwise the thread backoff a larger constant and make another CAS operation on a local memory–is_spinning to see if it is the first thread in its node. The ”is_spinning” is initiated to a special value, so only the first node get the special value and is allowed to continue competing the lock, other threads have to spin on the local memory until the first thread obtain the lock. The first thread obtainning the lock will reset the ”is_spinning” to the special value so that all the threads on the node will stop spinning on it and continue to compete the lock. To deal with the starvation, if a thread backoff too many times, it becames an ”angry” node, who has the ability to set the ”is_spinning” of the holding lock node to be its ”node_id”, so that the coming threads on the lock-holding node have to spin on the ”is_spinning” and will not be able to compete the lock until the ”is_spinning” is reset. The class also maintains an array of ”node_id” to record the nodes that has been stopped to acquire the lock by this node. Once the ”angry” node obtains the lock, it reset all the stopped node’s ”is_spinning” to be the specail value. The release method is fairly simple, all it needs to do is set the ”state” to be the initiated special value.

2.2 Hierarchical CLH Queue Lock

The backoff locks inherently has the drawback of cache-coherence traffic overhead because all the threads are spinning on the same shared location, by no exception is the HBO lock. Although the HBO lock with the starvation avoidance somehow reduce the number of threads contending on the same memory, there are still multiple threads spinning on it. Here the CLH Queue Lock is extended to a Hierarchical version that could take advantage of memory locality without suffering the fairness issues.

To implement the algorithm, we define the HCLH_QNode class and the HCLHLock class. The HCLH_QNode has three volatile members: the ”cluster_id” indicates where the thread is running, the ”successor_must_wait” is a bool varaible that is set to be true either when the corresponding thread has obtained the lock or is waiting for the lock, the ”tail_when_spliced” is also a bool varaible to notify its direct sucessor to splice the local queue to the global queue. The HCLH lock maintains a local queue for each cluster and a single global queue[4]. Correspondingly, the HCLHLock class a global HCLH_Qnode pointer and an pointer to an array of the HCLH_Qnode pointers. Besides, each thread has two local variables, my_qnode and my_pred, which are both pointers to the HCLH_Qnode. Each thread also has an integer variable to record the id of the node where the thread is running on, this variable can be used as the index to the local Queue. In the acquire method, each thread first initialize the thread local HCLH_Qnode, setting the ”successor_must_wait”to be true to indicate that it is waiting the lock. Then it CAS the local queue tail with its HCLH_Qnode, and simultaneously get its predecessor queue node. Then, the thread has to spin until either one of the folowing conditions is true:

- The predecessor is from the same node, and its bool variables ”successor_must_wait” and ”tail_when_sp liced” are both false.
- The predecessor is not from the same node or its bool variable ”tail_when_spliced” is true.

The first case indicates that the current thread is already in the global queue so what it needs to do is waiting its predecessor to release the lock in order to access the critical section. The second case is a little complicated, it means the current thread is at the head of the local queue, so it has to splice the local queue into the global queue. Before the splicing, the thread spleep for a certain time to wait its successor to join into the local queue. Then, the thread CAS the global queue tail to be its local queue tail, and simultaneously get its global predecessor. At the same, the thread needs to set the local queue tail’s bool variable ”tail_when_spliced” to be true, meaning that the first successors in the local queue take the responsiblity to splice the local queue into the global queue. After the splicing, the current thread has been in the global queue, so it spin on its global predecessor's ”successor_must_wait” until this bool variable turn true. In the release method, it reset the ”successor_must_wait” to be false, indicating it leave the critical section so that its successor could obtain the lock.

2.3 Lock Cohorting: A General Technique for Designing NUMA Locks

The idea of the cohort lock is the same as the previous two locks: passing the lock to the threads on the same cluster so that it allows sequences of threads(on the same cluster) to execute consecutively with little overhead and require little tuning beyond the locks used to create the cohort locks[2]. A lock is thread-oblivious when the acquiring thread and the releasing thread can be different. We call a lock a cohort lock when a thread holding the lock can detect whether there are other threads on the same cluster competing the lock. To access the critical section a thread must first acquires its local lock, and based on the state of the local lock, decides if it can immediately enter the critical section or must compete for the global lock.

C-BO-BO lock: the local and global locks are both simple backoff lock. The CBOBOLock class defines member variable of a BackoffLock pointer and a member variable of an array of BackoffLock pointer. These two variables are used as the global lock and the local locks respectively. The class also has two array of volatile bool variables–successor_exist and globalAcquired. The former is a flag showing whether there is other threads in the same node are waiting for the lock; the latter indicates whether the global lock has been obtained by this node. Besides, each thread needs to maintain a thread local variable ”t_node_num” to record the id of the node that the thread is running on. The ”t_node_num” variable is also used to index the local lock. In the acquire method, the thread first calls the local lock's acquire method. Upon success, the thread check if the ”globalAcquired” is true, if so, the thread return from the acquire method and claims owning the lock. Otherwise, it continue to call the global lock's acquire method, after that, it marks its ”globalAcquired” to be true. In the release method, it first check the ”successor_exist”, if it is true, then the thread only release the local lock and return. Otherwise, it means there is no thread in this node competing the lock, so it first release the global lock, and then release the local lock by calling their release method respectively.

C-BO-MCS lock: it comprises a global backoff lock and local MCS locks. Correspondingly, the CBOMCSLock class has a BackoffLock variable and a pointer variable to an array of MCSLock.The same as the CBOBOLock, the class has an array variable–globalAcquired to record if the node has obtained the global lock. Besides, the class also maintain two thread local variables, the first one is a QNode pointer–myLocalNode, the second one is an integer–t_node_num, which is the id of the node that the thread currently running on. In the acquire method, the thread first call its local MCS lock's acquire, passing the ”myLocalNode” thread local variable as the argument. Upon success, it check its ”globalAcquired”, if true, return from the method, otherwise it continues to call the global BackoffLock's acquire method then set the ”globalAcquired” to be true and return. In the release method, it first check if there is other threads on the same node that are wating for the lock by checking if ”myLocalNode”'s next pointer is not NULL. If no such threads, then it call the local MCSLock's release method and return, otherwse it frist call the global BackoffLock's release method and set the ”globalAcquired” to be false before releasing the local MCS lock.

C-MCS-MCS lock: it comprises a global MCS lock and local MCS lock. So that the CMCSMCSLock class has a MCS lock pointer variable and a MCS lock pointer array variable.To compete the global lock, each node should have its QNode pointer, so the CMCSMCSLock class maintain an array pointer of QNode. Each thread has its own thread local QNode pointer to join the local queue. Similar with the above two cohort lock, the the CMCSMC class define a ”globalAcquired” bool array varialbe to let each node record whther it has obtained the global lock. In the acquire method, the thread first call the local MCS lock's acquire method by passing its thread local QNode pointer as the argument. After returning from it, the thread check if this node has obtained the global lock, if so, it returns. Otherwise, it call the global MCS locks's acquire method by passing the current node's Qnode pointer. Upon success, it set the "globalAcquired" to be true and return. In the release method, the thread first check whether there is other threads on the same node waiting for the lock by checking if the local Qnode's next pointer is not NULL, if so, it only release the local MCS lock and return, otherwise, it first call the global MCS lock's release method and set the "globalAcquired" to be false before releasing the local MCS lock.

All the above NUMA aware lock are implemented mostly followed the paper and the textbook, there is a few pointers that not stricly match the original algorithm:

- in the pseudo of the HBO in the paper, line 39 is "is_spinning[my_node_id] = L". Then it is possible that more than one thread are allowed to spinning on the lock. In my implementation, I revised this line to be "if(CAS(is_spinning[my_node_id], DUMMY, L) == L) goto restart;". In this manner, only the first thread can continue to spin on the lock, others have to spin on the "is_spinning[my_node_id]", which is a local memory.
- In the SPARC architecture, the CAS is defined as atomic_cas_32(uint32_t*, uint32_t, uint32_t), while in the Queue lock implementation, we need something like CAS(Qnode**, Qnode*, Qnode*). So we have to convert the Qnode pointer to a uint32_t. The way to make this convertion is defining an union that has two member, the Qnode pointer and the uint32_t, and before the CAS operation, we first initialize an union variable and assign the Qnode pointer to its corresponding member, then in the CAS operation, we fectch the uint32_t member.

The algorithms of these locks are straightforward to understand. While the most difficult aspect that I found to implement these locks is debugging–making sure the locks running correctly in mutual exclusive way and without deadlock. In some cases, it's pretty subtle to claim that the lock is in deadlock. It's not always obvious to defferentiate if the implementation is really in deadlock or it is just running too slow. After identifing the deadlock or mutual exclusive issue, locating where exactlly the problem is another techique. To fix the issues, it is important to carefully review the logic of the implementation and work through all the possible concurrent cases.

3 Evaluation

In this section, we evaluated the NUMA aware locks, comparing them with each other as well as the traditional non NUMA aware locks including the TTAS Backoff lock, the MCS and CLH queue lock. We used a fairly simple benchmark that randomly perform three kinds of stack operations–pop, push and peek concurrently. All these experiments were conducted on two machines: a Intel(R) Xeon(R) CPU E7- 4870 @ 2.40GHz with 2 clusters(20 cores per cluster), and a Sun dual-chip machine, powered by two UltraSPARC T2 Plus (Niagara II) chips[1]. The Niagara II is a chip multithreading (CMT) processor, with 8 1.165 GHz in-order cores with 8 hardware strands per core, for a total of 64 hardware strands per chip.

3.1 Performance

To measure the performance, we conduct two experiments to compute the throughput in a fixed time and the time to complete a fixed number of operations. In the Intel machine, the queue locks work better than the backoff locks, which makes sense because the backoff locks suffer from the cache coherence traffic issue because all the threads spin on the same shared location, while the queue locks spin on its own local memory. The left graphs in the Figure.1 and Figure shows this. While we notice that the HCLH lock performs worst

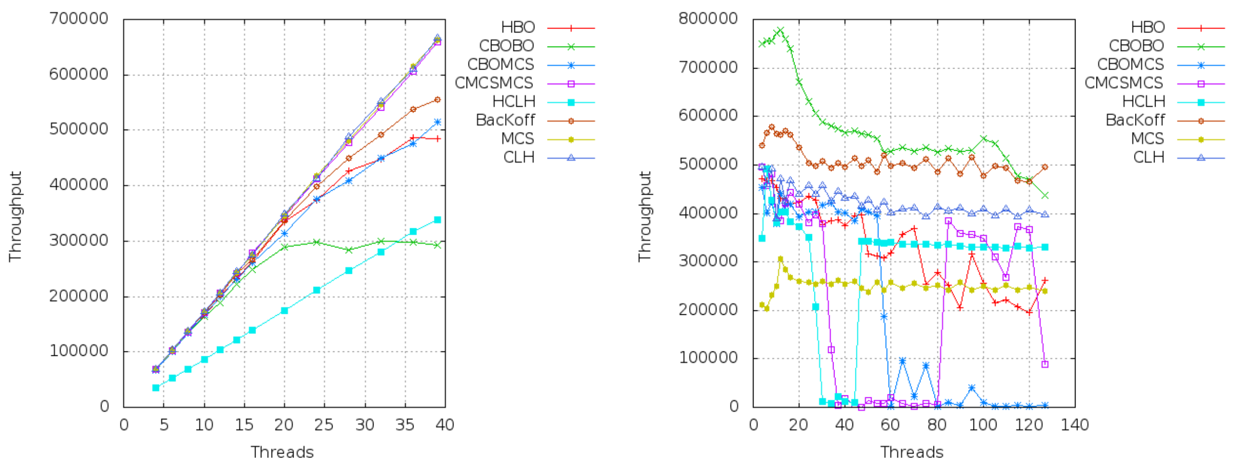

Figure 1: The graphs show the troughput of the operations for each lock running in 10 seconds. The left graph is in the Intel machine and the right graph is in the Sun Sparc machine.

in these eperiments in the intel machines. We suspect that the reason might be using the CAS operation on both local queue and splicing the local queue into the global queue.

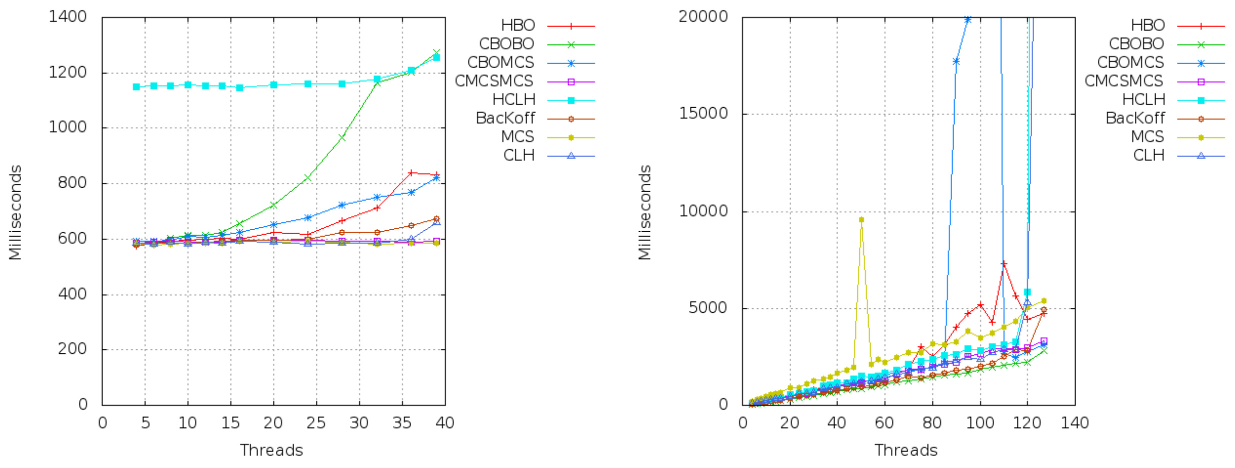

Figure 2: The graphs show the time to complete 10000 operations per thread.The left graph is in the Intel machine and the right graph is in the Sun Sparc machine

3.2 Locality

One of the important property of the NUMA aware lock is the locality, as the lock is more likely to pass from one thread to the other on the same cluster, so that we could expect threre should be less lock switching from one cluster to the other. We conducted an experiment to address this and show in Figure.3. We let the locks run in a fixed 10 seconds and record the total operations and the frequencies of the lock switching between the two clusters, then compute the opertations over the frequency. Thus, the larger value means the better locality the lock can archive. Show in the left graph in Figure.3, most all the NUMA aware locks have better locality than the traditional non NUMA aware lock. Especially, the HCLH lock out perform a lot than the others, which is reasonable because in HCLH lock, the master thread always waits a small piece

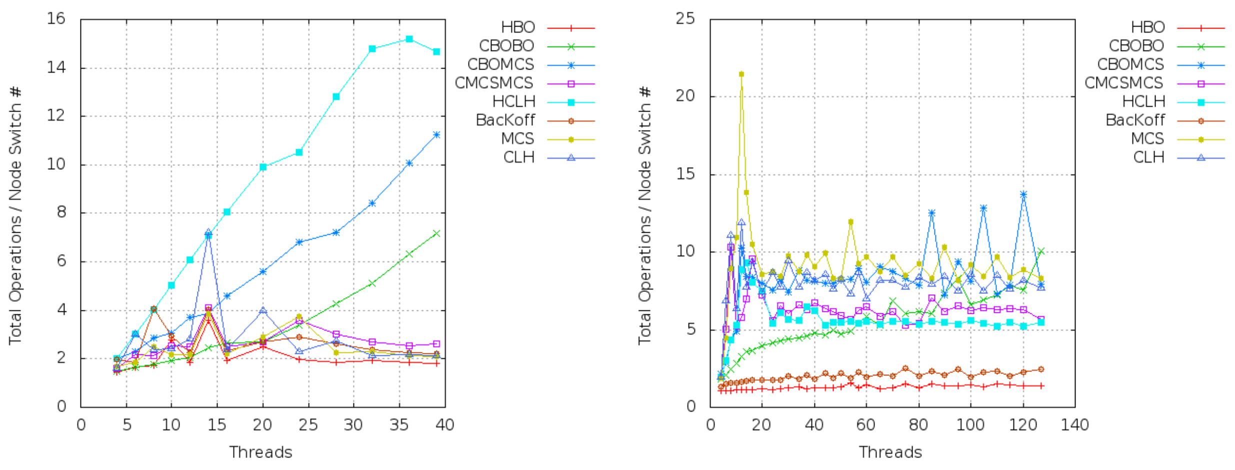

Figure 3: The graphs show the locality of the locks. We measure this by computer the total operations over the number of node switch during the locks running in a fixed 10 seconds.The left graph is in the Intel machine and the right graph is in the Sun Sparc machine

of time to let its neighbours shows up, so that once the master thread splice the local queue into the global queue, we expect the queue to have multiple nodes. Thus, a certain number of threads in the same cluster could execute in sequence.

3.3 Fairness

Last, it's also interesting to measure the fariness of the NUMA aware locks and compare them with the regular locks. In our experiment, the machines both have two clusters, so we record the number of operations in each cluster in a fixed 10 seconds. Then compute the maximum value over the minimum value. If the lock is ideally fair, then this value sould be very close to one. In the intel machine, it's clear that the HBO lock along with the non NUMA aware locks perform worse than the others. The cohort locks and the HCLH locks are mosly between 1 and 2, meaning that both the two nodes realtively handle the same scale of works.

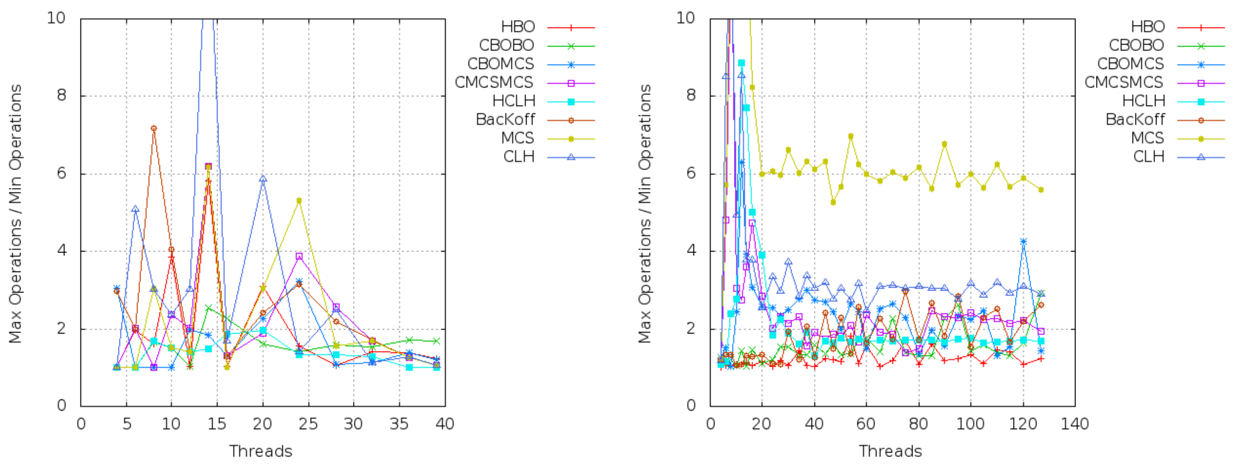

Figure 4: The graphs show fairness of the locks running in a fixed 10 seconds. We measure the fairness by computer the maximum operations node over the minimum operations, the larger the number, the less fair is the lock. The left graph is in the Intel machine and the right graph is in the Sun Sparc machine

4 Conclusions

Having review the recent NUMA aware lock algorithm, it's a good experience to implement play around them on the real NUMA machines in order to study their behavior. One of the motivation of this work is to realize how these locks could benifit the concurrent programs running on the NUMA machines. This is particular important as these NUMA machines become more and more popular nowadays. Although all the locks that we reviewed in this work are trying to take the advantage of memory locality and improve the performance, they behave quite different based on our experiments: the HCLH lock and the cohort locks perform much bettern in terms of the locality while most queue lock except the HCLH work better in the performance.

References

[1] *Opensparc t2 core microarchitecture specification*, `http://www.opensparc.net/pubs/t2/docs/OpenSPARCT2_Core_Micro_Arch.pdf`, 2007.

[2] Virendra J. Marathe David Dice and Nir Shavit, *Lock cohorting: A general technique for designing numa locks*, PPoPP'12 (2012), 247–256.

[3] Nir Shavit Maurice Herlihy, *The art of multiprocessor programming*, Morgan Kaufmann, 2008.

[4] D.Nussbaum V.Luchangco and N.Shavit, *A hierarchical clh queue lock*, Euro-Par (2006), 801–810.

[5] E.Hagersten Z.Radovic, *Hierarchical backoff locks for nonuniform communication architectures*, HPCA-9 2003. (2003), 241–252.