# Parallel and Distributed Systems Assignment 2

Sayak Chakraborti NetID-schakr11

February 2017

## Compilation and Running the code

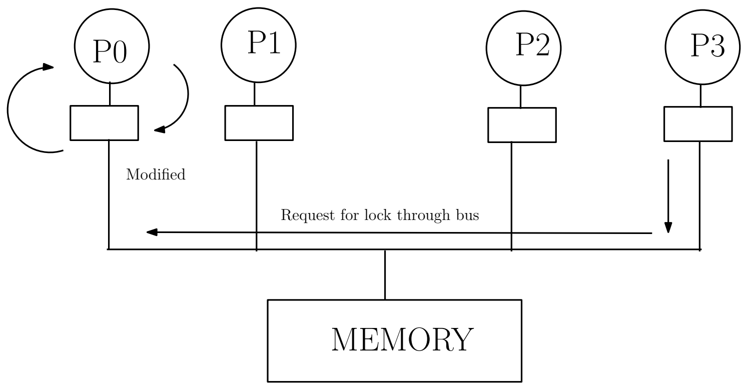
Compile using the make file.Type in make.If there are errors,then for explicit compilation use "g++ parcount2.cpp -std=c++0x -lpthread -o parcount2 -w" or "g++ parcount2.cpp -std=c++11 -lpthread -o parcount2 -w" whichever works on your compiler.
For running the code use "./parcount2 -t 5 -i 10000" or something similar with variable parameters.Extensive error checking has not been done for the code.

## Introduction

In this assignment we are supposed to compare the performance of some fair ,unfair,queue based and non-queue based spin locks and determine which of them perform how well under certain circumstances.The usage of locks is to ensure synchronisation among multiple threads/processes which want to access the same memory.This is done by ensuring mutual exclusion,which states that a particular point in time only one thread is allowed to execute it's critical section (which contains the shared variable).Mutual exclusion can be achieved by atomic read modify and write operations which are being exploited in some of the locks.Thus the general idea is to acquire a lock(indicating that no other thread/process should go into their critical section and only the thread/process acquiring the lock gets to go into it's critical section.After it is done with it's critical section it releases the lock by setting the lock value in a way that the other threads now have a chance to grab the lock to do their stuff.
This basic idea gets complicated when we worry about starvation free,fairness and deadlock/livelock avoidance strategies(where multiple threads/processes try to acquire the lock and each are unsuccessful in their attempts).

### The Locks

The lock is generally a global variable of a certain type.Once a thread has acquired a lock it continues into it's critical section and after it's done release the lock.But now the lock is in modified state in the local cache of this particular thread's core and it can grab onto it again and start executing it's critical section again unless there is an read request for the modified state from other core via the bus.Once a thread on a core has modified the lock invalidation signal is sent to the other cores who may have a local copy of the lock.So it may so happen that in large systems with multiple cores and thread can keep on acquiring and releasing the lock until some one else sends a request via the bus which takes considerable amount of time to reach this core.



From the book it is clear that the C++ and the test and set locks are not fair(explained below),they along with the queued locks and the ticket lock are not preemption tolerant.each lock has been discussed in some details below.

| | TAS (w/ backoff) | Ticket (w/ backoff) | MCS | CLH | MCS K42 | CLH K42 |
|---|---|---|---|---|---|---|
| Fairness | poor | good | good | good | good | good |
| Preemption Tolerance | fair | poor | poor | poor | poor | poor |
| Scalability | fair | fair | good | good | good | good |

Source:Shared Memory Synchronization

## C++ mutex

The mutex class in c++ allows mutual exclusion of access to a shared variable.It is the majorly used synchronization primitive for multi-threaded programs.A calling thread owns a mutex when the lock operation on it is successful until it calls an unlock.It is a single variable based lock and once the mutex is released every one can compete for the lock again making it an unfair lock.

### naive Test And Set

The test and set lock works on an atomic writable variable ,which induces bus traffic on every write the variable.The lock is pretty simple whoever wants to acquire the lock simply atomically checks the lock value if it is set to false or not.If it is set to false then no one is in his critical section and it simply sets the variable to true and return false the as the flag was not set previously and thus this thread enters into it's critical section while other threads read that the flag value is set and spin on the true return value.
Once the thread has executed it's critical section it sets the flag back to false,for other threads(as well as itself) to compete for the lock once again.But it's not a fair lock,as every time every thread competes for the lock(even the one that has recently had the lock).Also the read modify write to the same variable cause high contention.

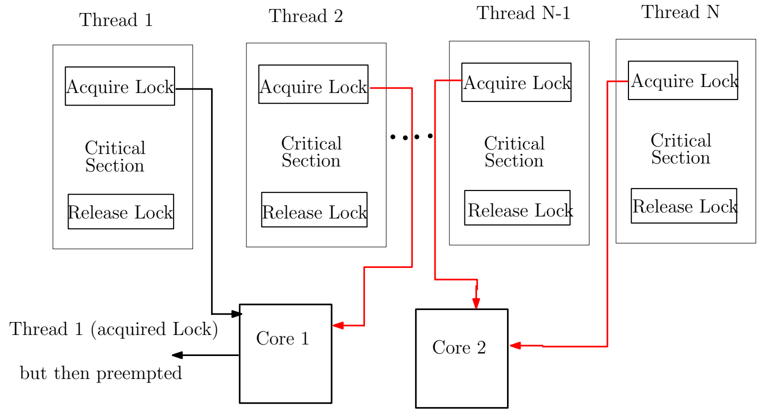
### Test And Set with exponential back off

This is the same as the test and set lock but with exponential back off.That once a thread has inspected the lock variable it should not inspect it right away as it is highly probable that the lock is still with the thread who had it and is working on it's critical section.Back off dependent on time required for critical section.For a well tuned TAS lock the it can be as much as twice as fast(or even higher) as the naive ticket lock.But it may not happen always tough.

### naive Ticket Lock

The ticket lock is a fair lock which allows equal opportunity for threads contending for the lock to grab it.It grants access to the competing locks in FIFO order.Each thread is issued a ticket based on the order in which it tries to grab the lock by calling the acquire function.In order that no two threads get the same ticket ,the ticket lock uses atomic fetch and increment to issue ticket whose magnitude increase with incoming requests.It uses two atomic variables next-ticket and now-serving to indicate the next ticket value that needs to be issued and the currently being served ticket respectively.Upon receiving a ticket number the thread checks whether the current serving ticket is it's ticket or not in a while loop.If it happens to be it's ticket then it breaks from the loop and acquires the lock and goes into it's critical section.If not then it keeps checking it.
For release we simply increment the now-serving and store it atomically so that the next ticket waiting can start.Ticket lock has no starvation unlike the test and set lock.**But may suffer when the thread that has acquired the lock**

**is preempted and the other threads are spinning helplessly in hope that the acquired thread is working on it's critical section and once it is done it will increment the ticket for the next thread to get going.This noticeable happens when the number of threads are more than the number of physical cores to run on and thus a particular point at least one of the thread is not running.It may also be cause by FIFO scheduler or something like that.**



### Ticket Lock with exponential back off

This is the same idea as of the ticket lock ,but instead of the thread trying to acquire to acquire the lock repeatedly ,it delays the next check for the now serving ticket against it's ticket by some amount that should be proportional to the amount of time required for the lock holding thread to complete it's critical section.The delay is implement with an empty loop that iterates the arguments number of times.On a well tuned (maybe tuned with some machine learning) ticket lock this helps to reduce the reads of this now-serving variable which redues contention and time.

### MCS Lock

The Mellor-Crummey and Scott(MCS) lock is a queue based lock which was inspired from other array based lock.Since it is a queue based lock it has no limitations as to how many threads can work with the lock.A queue is maintained that tries to grant access to the requesting threads in a First In First Out(FIFO) order.Each thread creates a local qnode which contains a pointer link to the next qnode and a boolean waiting value which suggests whether it should wait or move along into it's critical section.
When a thread tries to acquire the lock it sends it local qnode variable to the acquire function where first it is initializes the next pointer of the qnode to

NULL and the bollean variable is set to true(both atomically).Now we insert this qnode into the tail of the queue and check whether there was any previous entry to the queue or not with the prev pointer.If there wasn't any previous entry then this thread has the lock and can go ahead into it's critical section.If there was an previous entry into the queue(which may or may not have the lock now) then let it know that you are the next one and wait on your waiting variable until it is set the false by your predecessor.
For release find the successor of your qnode and if the successor is found then set his waiting variable to false so that he can execute his critical section.If the successor is NULL and thus it gives an impression that there are no successor who need to be given charge of the lock ,then atomically try to compare and swap the tail pointer back to NULL,if it is successful then there has been no new addition to the queue after our last check and we are good,but if it fails that means there has been a recent addition to the queue which is supposed to be our successor and we should wait until it modifies our next pointer to itself and we ca set it's waiting variable to false so that it can right away start it's critical section since I am done.
There are other benefits of the MCS lock like it is a fair and gives each requesting thread equal opportunity to deal with it's critical section once the preceding critical section is done.Also the requesting threads join the queue without waiting on anything ,that is, as soon as a thread requests the lock it is granted a position in the FIFO queue where it is supposed to be served in definite time,thus no problem of starvation.But the size of the queue may increase if the thread that currently has the lock is preempted and there are many lock acquisition requests that queue up.
Each thread waiting in the queue spins on it's local waiting variable ,thus there is no contention for it.The disadvantage of the MCS lock is that each thread has to maintain a local qnode variable and pass the pointer to the acquire and release function whenever needed.This can be a problem for a system which wants to have diverse locks available to it's users with the same set of functions parameters(may be some kind of polymorphism).This was taken care in the K42 version of this lock.

### K42 MCS Lock

The idea here is that the threads don't explicitly need to have local qnodes for each of them whose pointer they are supposed to pass to the acquire and release of the MCS. We can do this because the only job of the of the thread's qnode which has acquired the lock is to hold the pointer to the next qnode of some other thread in the queue so that it can set it's waiting variable appropriately and let it know once it's done.
So now the lock instead of being a tail pointer as in MCS is a qnode struct with two fields (first is the tail pointer which serves the same purpose as in MCS and the other one is the next one which refers to the next pointer in the waiting queue who has to be indicated when the lock holding thread finishes).
In order to acquire the lock the first thread which observes tail to be NULL,uses

atomic compare and swap to change the tail value to point to the lock itself(that is when there is no previous qnode ,prev value is NULL).Thus the thread now acquires the lock and the other threads know that someone has acquired the lock as the tail pointer is no longer NULL and no other threads are waiting as the tail pointer points back to the lock.When a new thread wants to acquire the lock ,it updates the next link of the previous node (which incidentally is the lock itself and not the first threads qnode) and waits for the lock to be free(on it's tail pointer which is set to boolean true upon initialization).If any other thread comes to acquire the lock then it sets it's previous pointers(the last one waiting) link to it's qnode.
For release the releasing thread(the one which currently holds the lock and is going to release it) checks for the next waiting thread in queue by accessing the next pointer of the q and if there is a successor then it changes it's tail pointer to NULL,thus setting it free from waiting and actually acquiring the lock.
The advantages here are we actually don't need to keep track of the qnode of the node that is actually running(has the lock) and modify it's next pointer and so on as in MCS.The global lock has two pointers one to the immediate waiting thread next(uses the next field) and to keep track of the latest waiting thread it uses the tail pointer(so that upon further request it can add them to the end of the queue).Apart from this each waiting thread also has it's next pointer connected to the next waiting thread.So parsing from q.next to q.tail will (using the next field in each node) will give us all the waiting threads in order.
As suggested in the book,that the entry into the acquire waiting queue is lock-free and not wait free ,thus it may cause starvation as some thread may always get a chance to acquire the lock when it probes in lock-free condition while other might not be that lucky.This leads to poorer performance compared to MCS but gets rid of each thread having to pass local pointers to the acquire and release functions.

### CLH Lock

The main difference here from the MCS algorithm is that each thread instead of spinning on it's own variable in it's local qnode it spins on the waiting variable of the previous/predecessor qnode's variable.In comparison to MCS the spin will still be on a local variable to the thread that is spinning as the other threads variable will be copied into the local cache of the successor thread.On the other hand the qnode of the previous thread that has potentially finished it's critical section will now have to keep it's qnode available for the successor to spin on.Thus the same qnode shouldn't be allowed to release(as there may be a dependent thread (successor) that is waiting on this qnode).Hence the solution is to provide a fresh qnode to acquire and a different qnode to release.
Here the lock itself is a pointer to a qnode which has qnode pointer prev and a boolean successor must wait flag.Initially the prev pointer is set to NULL and the boolean flag is set to FALSE indicating the the first acquiring thread doesn't

need to wait.When a thread wants to acquire the lock it set's it successor must wait flag to true(so that the successor waits for it to finish) and it places itself at the tail of the queue and gets it's predecessor's qnode pointer(which it stores in it's prev pointer).And now it waits on it's predecessor's successor must wait flag(which for the first acquiring thread would be false as it's predecessor is the lock itself).
Upon completion of it's critical section it calls the release with a different node pointer (so as to preserve the acquire pointer for the successor thread to wait on).It accesses the prev pointer to get the previous node and set it's own successor must wait to false(so that the successor can get it's critical section which has been waiting on this flag).And points it's pointer to the predecessor.

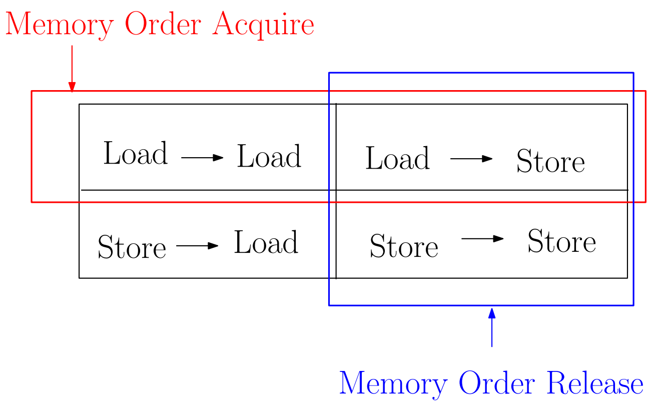
### K42 CLH Lock

The basic idea here is that locks be released in the reverse order than which they are acquired to accommodate nested critical sections.For this it is suggested that we have a global array of thread-qnode-pointers which would be indexed by thread IDs.The lock itself is a head and a tail pointer of type qnode. each qnode struct has only one single boolean variable that is successor must wait flag as in CLH but no prev pointer.Initially tail points to a dummy qnode where the successor must wait is set to flas(which is quite evident).
In order to acquire the lock the thread-qnode-ptr of the particular thread is accessed and the successor must wait flag is set to true(as in case of CLH so that the successor waits till I complete).This node is then swapped with the tail pointer and threads qnode.Wait on the predecessors flag(which for the first acquiring thread is false).Now when the thread has the lock it stores it's pointer as head and modifies it's array pointer to the predecessor pointer.
For release it simply sets the head pointer's successor must wait to false allowing the next waiting thread to get into it's critical section.

## Memory ordering

In this assignment it has been noted that the memory ordering would matter for the locks to work appropriately ,especially the queued locks.The general consistency model that the underlying architecture provides determine the behavior of the code if it doesn't have proper memory orderings.The consistency models that I am familiar with are ,sequential consistency model which respects all possible memory orders ,that means it does reorder anything.Then if we allow the store to load reordering (via the store buffer for better performance )then it falls on to becoming the TSO(Total Store Order) and now if we allow store to store reordering then it falls to PSO(Partial Store Order) and if other re-orderings are allowed it is called relaxed order.
So inorder to ensure that whichever machine we run it on (node2x18a (X86)

or node-ibm-822(POWER)) we have our code following strict ordering as the acquire and release require (in most cases that others) a store to happen before a particular load ,we impose ordering using memory order acquire and memory order release.The functioning of these two are depicted below.



Also I found some memory ordering of these two machines ,that might explain certain things.

Memory Ordering

| Type | POWER | x86 |
|---|---|---|
| Loads reordered after loads | ✓ | |
| Stores reorder after loads | ✓ | ✓ |
| Stores reorder after stores | ✓ | |
| Loads reordered after stores | ✓ | |
| Atomic reordered with loads | ✓ | |
| Atomic reordered with stores | ✓ | |
| Dependent loads reordered | | |
| Incoherent instruction cache pipeline | ✓ | ✓ |

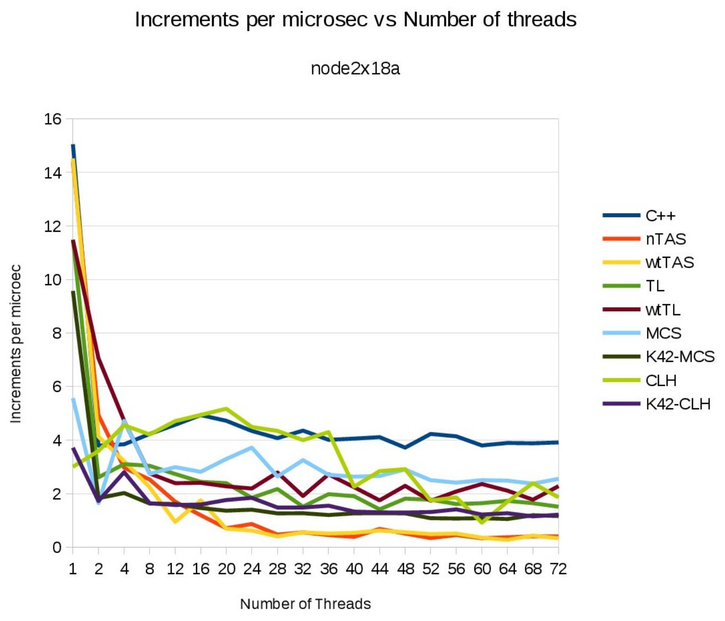
Source:wikipedia

## Experimental Results

The iteration value has been taken as such that the execution time for each of the lock implementation is at-least as close a 1 sec.For higher number of thread

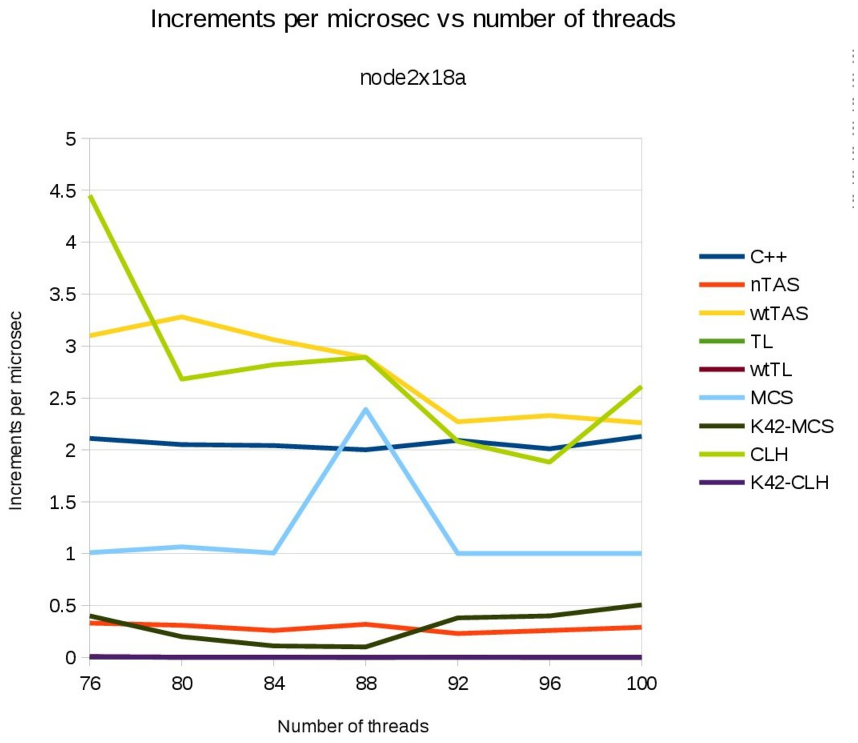
values the execution time for queued locks is more than 1 sec near 3-4 secs approximately.Since my code called each function(for a particular lock) after the end of the previous one,and the order was C++,TAS,wtTAS,ticket lock ,well trained ticket lock ,MCS ,K42-MCS,CLH and K42-CLH,it may so happen that for higher iteration values and higher number of threads(greater than 72 for the node2x18a) than the physical cores ,the ticket lock spends a lot of time which seems like forever and thus I am unable to report the other results as well.
For the node2x18a machine I got the following result:-

| Number of threads | Counter value | Iteration value | Increments per 1000 milliseconds | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | C++ | TAS | wtTAS | TL | wtTL | MCS | K42-MCS | CLH | K-42 CLH |
| 1 | 100000 | 100000 | 15.05 | 14.39 | 14.52 | 11.49 | 11.47 | 5.56 | 9.57 | 3.00 | 3.71 |
| 2 | 200000 | 100000 | 7.64 | 8.303 | 14.82 | 4.96 | 11.97 | 1.64 | 2.75 | 3.95 | 2.72 |
| 4 | 400000 | 100000 | 6.64 | 3.96 | 4.21 | 4.14 | 1.99 | 4.72 | 2.02 | 4.55 | 2.80 |
| 8 | 800000 | 100000 | 4.08 | 2.53 | 2.43 | 3.04 | 2.75 | 2.74 | 1.63 | 4.21 | 1.64 |
| 12 | 1200000 | 100000 | 4.57 | 1.70 | 0.94 | 2.73 | 2.39 | 2.99 | 1.61 | 4.71 | 1.57 |
| 16 | 1600000 | 100000 | 4.93 | 1.19 | 1.75 | 2.44 | 2.40 | 2.81 | 1.46 | 4.94 | 1.59 |
| 20 | 2000000 | 100000 | 4.72 | 0.71 | 0.69 | 2.39 | 2.27 | 3.29 | 1.36 | 5.17 | 1.76 |
| 24 | 2400000 | 100000 | 4.34 | 0.86 | 0.62 | 1.84 | 2.19 | 3.72 | 1.40 | 4.49 | 1.84 |
| 28 | 2800000 | 100000 | 4.07 | 0.47 | 0.40 | 2.17 | 2.79 | 2.64 | 1.26 | 4.34 | 1.48 |
| 32 | 3200000 | 100000 | 4.35 | 0.55 | 0.55 | 1.51 | 1.91 | 3.25 | 1.27 | 4.00 | 1.48 |
| 36 | 3600000 | 100000 | 4.01 | 0.45 | 0.51 | 1.98 | 2.72 | 2.70 | 1.20 | 4.30 | 1.55 |
| 40 | 4000000 | 100000 | 4.05 | 0.38 | 0.53 | 1.90 | 2.24 | 2.63 | 1.26 | 2.26 | 1.33 |
| 44 | 4400000 | 100000 | 4.11 | 0.68 | 0.62 | 1.41 | 1.75 | 2.66 | 1.27 | 2.84 | 1.30 |
| 48 | 4800000 | 100000 | 3.72 | 0.51 | 0.56 | 1.81 | 2.29 | 2.90 | 1.26 | 2.91 | 1.29 |
| 52 | 5200000 | 100000 | 4.23 | 0.34 | 0.49 | 1.77 | 1.73 | 2.50 | 1.08 | 1.74 | 1.31 |
| 56 | 5600000 | 100000 | 4.14 | 0.45 | 0.51 | 1.61 | 2.08 | 2.41 | 1.07 | 1.86 | 1.41 |
| 60 | 6000000 | 100000 | 3.80 | 0.33 | 0.35 | 1.64 | 2.36 | 2.50 | 1.08 | 0.91 | 1.22 |
| 64 | 6400000 | 100000 | 3.89 | 0.37 | 0.27 | 1.73 | 2.11 | 2.48 | 1.05 | 1.70 | 1.27 |
| 68 | 6800000 | 100000 | 3.88 | 0.40 | 0.43 | 1.64 | 1.77 | 2.37 | 1.19 | 2.39 | 1.15 |
| 72 | 7200000 | 100000 | 3.91 | 0.40 | 0.33 | 1.51 | 2.28 | 2.55 | 1.16 | 1.86 | 1.21 |
| 76 | 7600000 | 100000 | 3.84 | 0.37 | 0.31 | | | | | | |



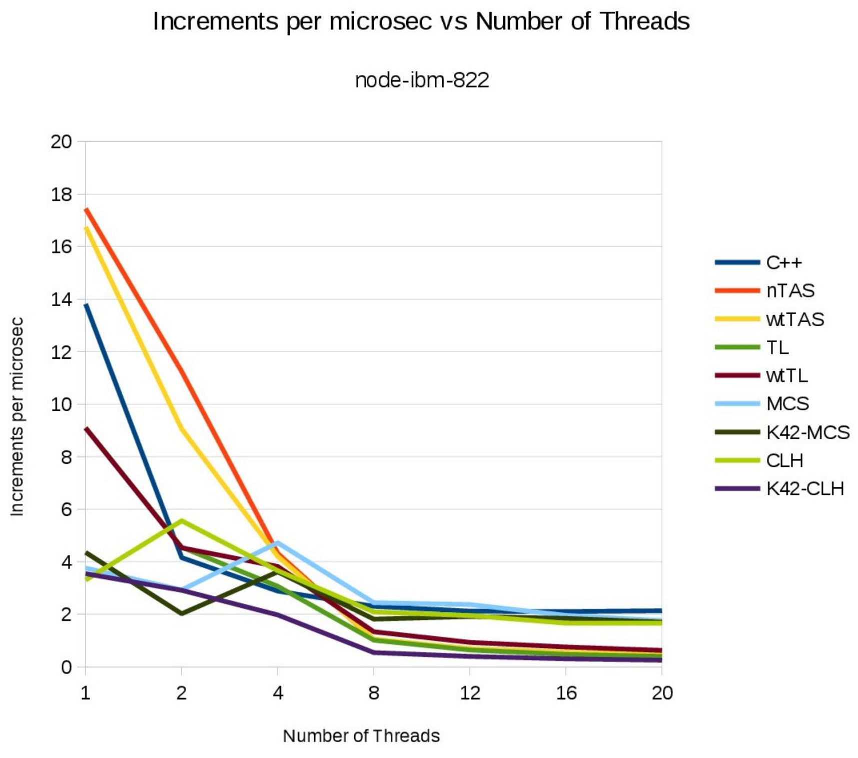
So for that reason I ran with lower iteration value for high number of threads.But still the performance of all the locks deteriorated like anything.The worst was the ticket lock.

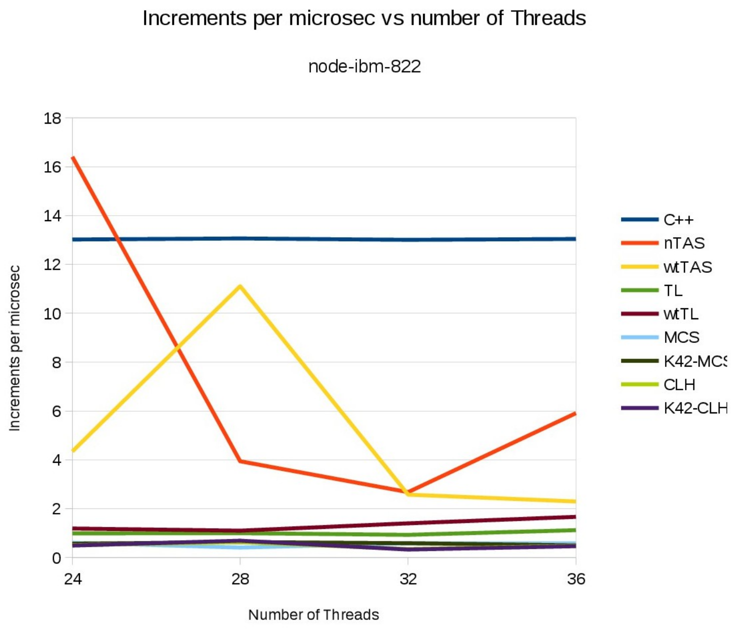
| Number of threads | Counter value | Iteration value | Increments per 1000 milliseconds | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | C++ | TAS | wtTAS | TL | wtTL | MCS | K42-MCS | CLH | K-42 CLH |
| 76 | 38000 | 500 | 2.11 | 0.33 | 3.10 | 0.0034 | 0.0081 | 1.010 | 0.4 | 4.45 | 0.004 |
| 80 | 40000 | 500 | 2.05 | 0.31 | 3.28 | 0.001 | 0.003 | 1.064 | 0.2 | 2.68 | 0.002 |
| 84 | 42000 | 500 | 2.04 | 0.26 | 3.06 | 0.002 | 0.003 | 1.006 | 0.11 | 2.82 | 0.001 |
| 88 | 44000 | 500 | 2.00 | 0.320 | 2.89 | 0.002 | 0.001 | 2.39 | 0.101 | 2.89 | 0.0008 |
| 92 | 46000 | 500 | 2.09 | 0.23 | 2.27 | 0.0021 | 0.0024 | 1.00095 | 0.38 | 2.08 | 0.0006 |
| 96 | 48000 | 500 | 2.01 | 0.26 | 2.33 | 0.0011 | 0.0010 | 1.002 | 0.40 | 1.88 | 0.0010 |
| 100 | 50000 | 500 | 2.13 | 0.29 | 2.26 | 0.0007 | 0.0019 | 1.0009 | 0.505 | 2.61 | 0.0008 |

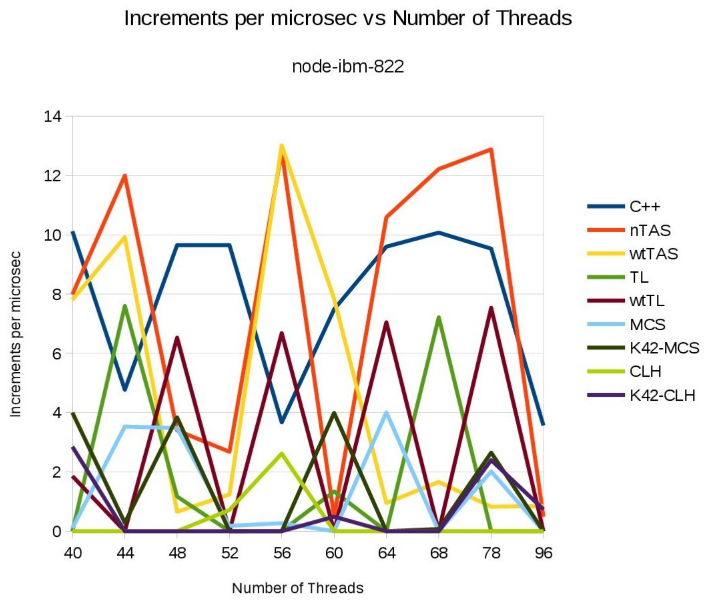


Node-ibm-822 is a IBM POWER architecture machine with 20 online cores and 140 off-line cores.It has four NUMA nodes each having 5 cores thus 20 sockets.Initially I started with a high iteration value of 400,000. But after reaching the numThread count of 20(which is incidentally the number of online cores,the MCS lock started to take a long time and thus I had to terminated the processes.On the IBM machine none of the queued locks work well.
For the node-ibm-822 I got the following results:-

| Number of threads | Counter value | Iteration value | Increments per 1000 milliseconds | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | C++ | TAS | wtTAS | TL | wtTL | MCS | K42-MCS | CLH | K-42 CLH |
| 1 | 400000 | 400000 | 13.81 | 17.44 | 16.75 | 9.10 | 9.09 | 3.76 | 4.35 | 3.30 | 3.54 |
| 2 | 800000 | 400000 | 4.15 | 11.24 | 9.05 | 4.55 | 4.53 | 2.93 | 2.02 | 5.56 | 2.91 |
| 4 | 1600000 | 400000 | 2.88 | 4.31 | 4.20 | 3.06 | 3.82 | 4.72 | 3.61 | 3.66 | 1.97 |
| 8 | 3200000 | 400000 | 2.30 | 1.02 | 1.06 | 1.009 | 1.33 | 2.44 | 1.81 | 2.09 | 0.54 |
| 12 | 4800000 | 400000 | 2.12 | 0.68 | 0.69 | 0.64 | 0.93 | 2.37 | 1.91 | 1.95 | 0.39 |
| 16 | 6400000 | 400000 | 2.10 | 0.53 | 0.55 | 0.47 | 0.75 | 1.96 | 1.84 | 1.66 | 0.30 |
| 20 | 8000000 | 400000 | 2.13 | 0.48 | 0.488 | 0.393 | 0.621 | 1.76 | 1.70 | 1.66 | 0.25 |
| 24 | 720000 | 3000 | 13.02 | 16.40 | 4.35 | 0.99 | 1.19 | 0.61 | 0.57 | 0.51 | 0.49 |
| 28 | 840000 | 3000 | 13.06 | 3.94 | 11.11 | - | - | 0.41 | 0.64 | - | 0.69 |
| 32 | 96000 | 3000 | 13.01 | 2.68 | 2.58 | - | - | 0.58 | 0.59 | 0.35 | 0.33 |
| 36 | 108000 | 3000 | 13.04 | 5.91 | 2.30 | 1.12 | 1.67 | 0.58 | 0.50 | 0.48 | 0.47 |
| 40 | 20000 | 500 | 10.12 | 7.99 | 7.81 | 0.0003 | 1.86124 | 0.11 | 4.00 | 0.001 | 2.85 |
| 44 | 22000 | 500 | 4.77 | 12.00 | 9.92 | 7.60 | 0.00084 | 3.53 | 0.28 | 0.00089 | 0.00036 |
| 48 | 24000 | 500 | 9.65 | 3.40 | 0.66 | 1.17 | 6.53 | 3.48 | 3.84 | 0.0003 | 0.00019 |
| 52 | 26000 | 500 | 9.65 | 2.68 | 1.24 | 0.000155 | 0.0027 | 0.181 | 0.002 | 0.714 | 0.0002 |
| 56 | 28000 | 500 | 3.67 | 12.83 | 13.02 | 0.004 | 6.683 | 0.27 | 0.00014 | 2.62 | 0.0003 |
| 60 | 30000 | 500 | 7.49 | 0.49 | 7.84 | 1.34 | 0.0004 | 0.00018 | 3.99 | 0.00021 | 0.49 |
| 64 | 32000 | 500 | 9.60 | 10.59 | 0.95 | 0.00030 | 7.05 | 4.01 | 0.00015 | 0.00016 | 0.00043 |
| 68 | 34000 | 500 | 10.07 | 12.22 | 1.66 | 7.22 | 0.00015 | 0.0003 | 0.0735 | 0.0001 | 0.00011 |
| 78 | 39000 | 500 | 9.53 | 12.88 | 0.83 | 0.00011 | 7.54 | 2.017 | 2.65 | 0.00029 | 2.40 |
| 96 | 48000 | 500 | 3.57 | 0.49 | 0.85 | 0.00013 | 0.00016 | 0.00018 | 0.00013 | 0.00011 | 0.74 |
| 100 | 10000 | 100 | 2.12 | 0.49 | 0.54 | 0.00009 | 0.00011 | 0.0003 | 0.00014 | 0.117 | 0.00009 |
| 102 | 10200 | 100 | 5.94 | 6.82 | 2.00 | - | - | 2.11 | 0.0008 | - | - |
| 105 | 10500 | 100 | 5.92 | 2.43 | 4.03 | - | - | 3.25 | 0.0003 | - | - |
| 110 | 11000 | 100 | 4.25 | 5.62 | 5.53 | - | - | 3.35 | 2.16 | - | - |
| 112 | 11200 | 100 | 4.24 | 3.60 | 5.10 | - | - | 2.88 | 2.14 | - | - |




Increments per microsec vs number of Threads
node-ibm-822
Increments per microsec
Number of Threads
C++
nTAS
wtTAS
TL
wtTL
MCS
K42-MCS
CLH
K42-CLH


Increments per microsec vs Number of Threads
node-ibm-822
Increments per microsec
Number of Threads
C++
nTAS
wtTAS
TL
wtTL
MCS
K42-MCS
CLH
K42-CLH

## Observation

From the first experiment conducted on the node2x18a using a high value of a million as iteration number and number of threads up to the physical cores on node2x18a ,everything seems to go well,for smaller number of threads the test and set lock and ticket lock work pretty well with their exponential back off variant .The queued locks aren't that good at the beginning but they catch up soon and they have a steady increments per microseconds ,especially the CLH lock and the MCS locks.The reason the non-queue locks perform well when there are small number of threads is due to less contention,but as there are more and more threads there is more contention(even there is cache coherency overheads) and they are just not good enough.Also for the non-queued locks the locks are centralized and hence the cause of such contention.But the queued locks maintain their steady increments rate because they are highly scalable and the MCS is wait free and the CLH is lock free which is probably the case for their success. When the number of threads exceeds the number of cores things go really bad for the K42 variants of the queued locks as well as the ticket locks and the naive TAS.I think naive TAS suffers from too much contention and thus has such a performance ,on the other hand the well tuned TAS performs the rest and is in competition with the CLH lock which due to probably it's less complicated nature and allowing two separate pointers for release and acquire perform pretty well.Sadly this can't be said for the MCS lock where each thread spins on it's local waiting variable instead of the predecessor waiting variable as in CLH.**Thus it may so happen that when a thread is getting preempted and there are a lot of context switches each time that local variable has to be brought into the cache which consumes some time,but in case of the CLH it may so happen that the thread is scheduled on the same core as it's predecessor and thus it enjoys and exploits spacial locality and has high performance,might be some scheduler optimisation**.

Now coming to the IBM machine it is observed that for number of threads upto 20 which is incidentally the number of on line cores for the node-ibm-822 machine the locks work normally and follow similar pattern as the node2x18a results for the first few number of threads increments upto 20.Then the iteration value has to be reduced to get results in decent time but we see that the queued locks are not performing very well maybe ,this is because the IBM POWER machine has to implicitly check for and implement the memory orders which is not the case for the x86 machine and thus maybe it requires more time to get these memory orderings sorted out and provide memory fences which causes high execution time.
For number of threads greater than forty I observed a weird pattern where for each alternate execution the queued locks would perform better and then drop down again(may be due to some spacial and time locality issue),but when I tried to run for threads beyond 100 then I got segmentation faults for many of the times for CLH locks and hence wasn't able to report accurate value there.
The graph from the IBM machine looks like a work of Picasso(the painter) I

really can't figure out exactly why this pattern and what's happening.But the TAS locks seem to do pretty well and I am pretty sure that there is some underlying memory ordering going on and also when the code is trying to access the number of offline threads there may high latency and access times which are causing these irrational and abrupt results.
Some concepts pertaining to the locks have already been explained in the beginning and I hope my explanation is satisfactory.

## Conclusion

I have learnt a lot about atomic operations in c++ ,specially 'compareExchangeStrong' and also about different kinds of locks that may be used in parallel and distributed systems.I have struggled a bit with the code and some bizare results and the end of the day I hope that I have put up a good report.