**Multicore Programming – Homework #4 - Practical**

**Dan Klein, 301217725**

**Guy Reuven, 301840591**

*General note:*

On each of the described tests, we present the **average of 4 runs** (as assigned to runs parameter), performed one after the other, and thus, hopefully performed under the same machine conditions.

When running uniformLoad or exponentialLoad tests, the number of runs was adjusted to 5 and 11 iterations accordingly.

**Counter Tests- Experiment #1: Idle Lock Overhead**

In this experiment we have compared the throughput of SerialCounter and ParallelCounter (single thread) implementations.  
Specifically, we calculated the speedup between the same running parameters under these two implementations.

Since counter test is serial bottleneck, performance degradation is obviously expected, and by observing the speedup we can learn on the overhead magnitude.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | TAS | Backoff | CLH | MCS |
| SpeedUp () | 0.254954271 | 0.259510386 | 0.079476652 | 0.155143101 |

As seen in the table, we can see major performance degradation in all locks, as expected. Out of all the locks, we can see that the CLH&MCS lock suffers the most degradation, which can be explained since these locks’ implementation is based on linked list data-structure, which is relatively expensive to manage.

On the other hand, TAS and Backoff locks has better performance (which is very similar, since Backoff lock with single thread is like TAS lock), which is also not surprising since these locks are simpler to manage in operations number perspective.

The table also presenting a difference between the two more complex locks (MCS and CLH), such that the CLH locks suffers more from the parallel overhead.

That behavior can be explained by looking at the code for a single thread:

* In the CLH lock() method the thread preform getAndSet on the tail and then it sets his pred to be the previous tail.   
  Then, he start sniping, asking about the condition only once, but it needs to read the lock creator memory that can be far and only then it realize that it can acquire the lock.
* On the other observing MCS lock() method, we can see that after the getAndSet if the predecessor is null the thread acquire the lock immediately.

Of course, the MCS operation is more efficient, hence the speedup difference.

**Counter Tests- Experiment #2: Lock Scaling**

*Tuning the Backoff Lock*

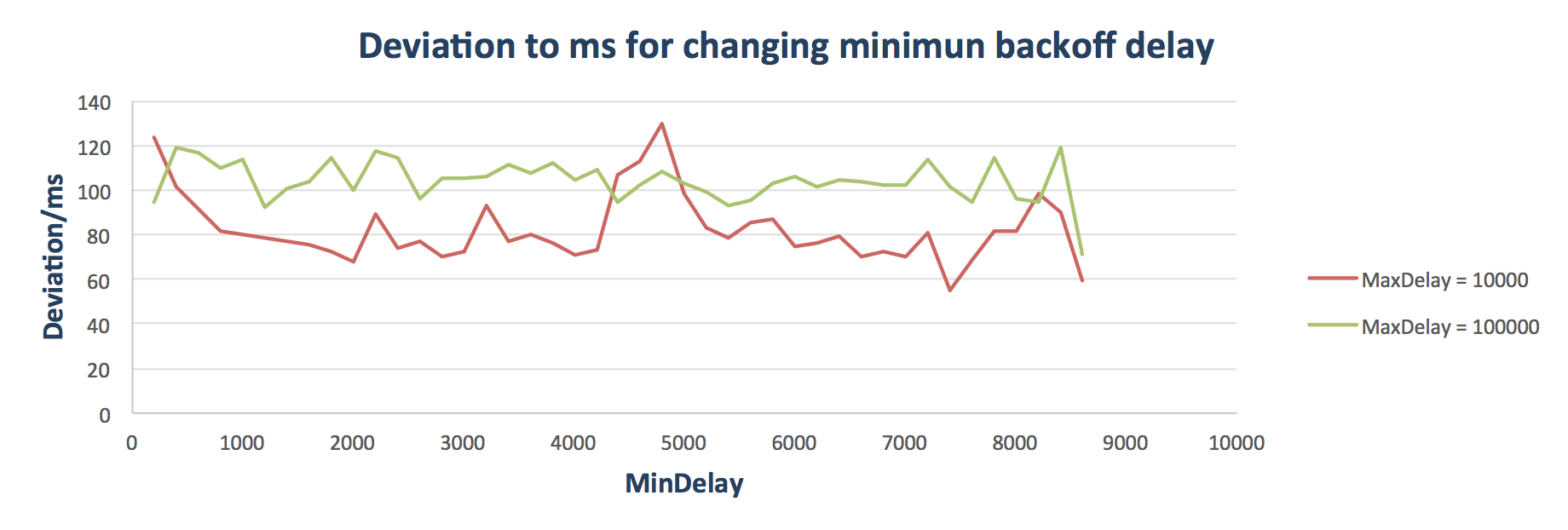
In this Pre-experiment, we tried to tune the MIN/MAX delay parameters of the Backoff lock.

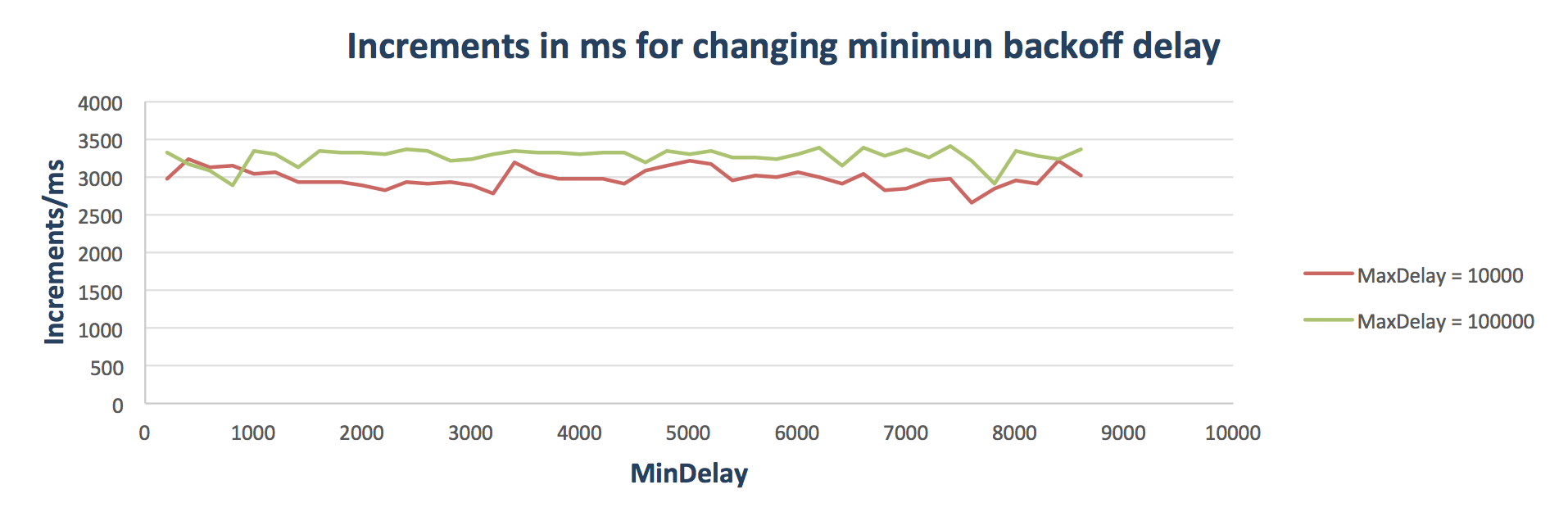
The goal of this assignment is very simple: achieve the highest increments/msec ratio, while keeping the standard deviation as low as possible, when running on university’s multicore machine.

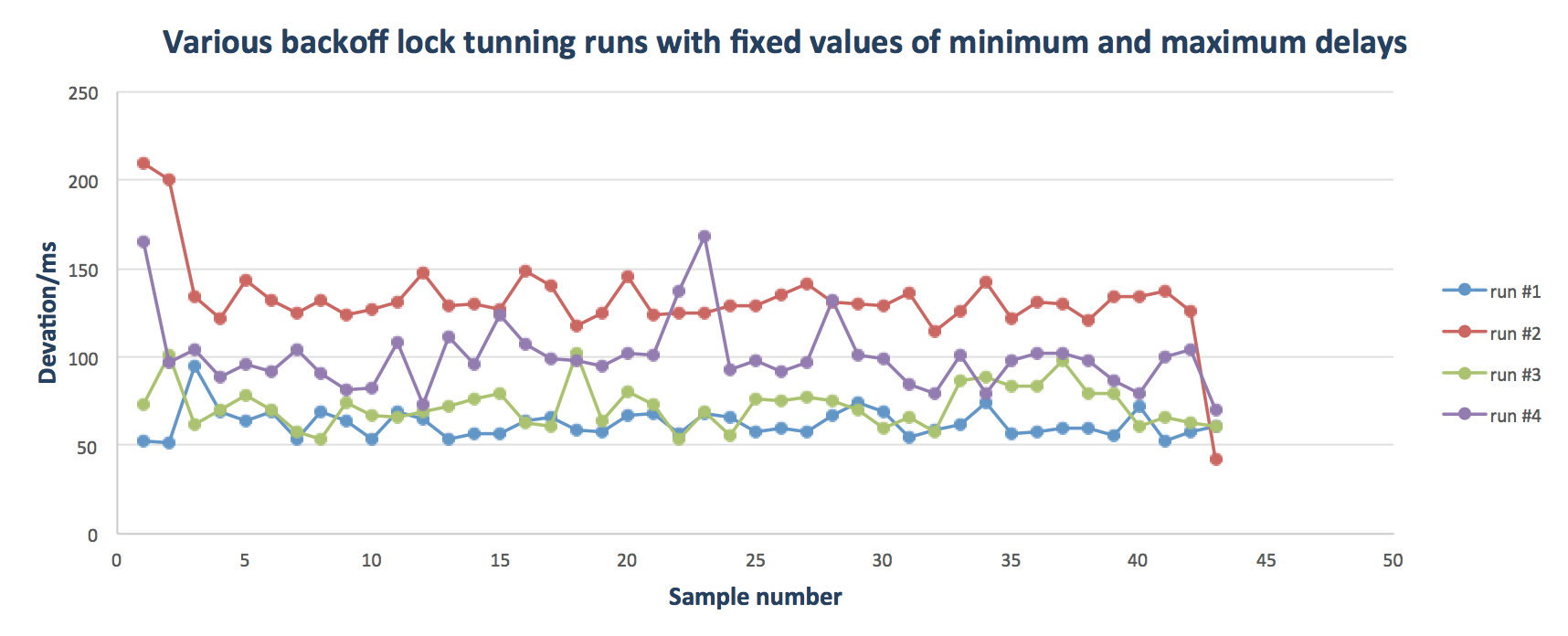
“Tuning the backoff lock could be a nightmare” [Shavit, 28/12/14]

However, we quickly found out the task isn’t that simple.  
We learned that even when picking greater values (small deviation, relatively high increment ratio), these results **aren’t consistent**, so running the very same test again resulting very different values.

The following graphs best demonstrates this catch[[1]](#footnote-1):

*Deviation comparison[[2]](#footnote-2):*

*Increment comparison:*

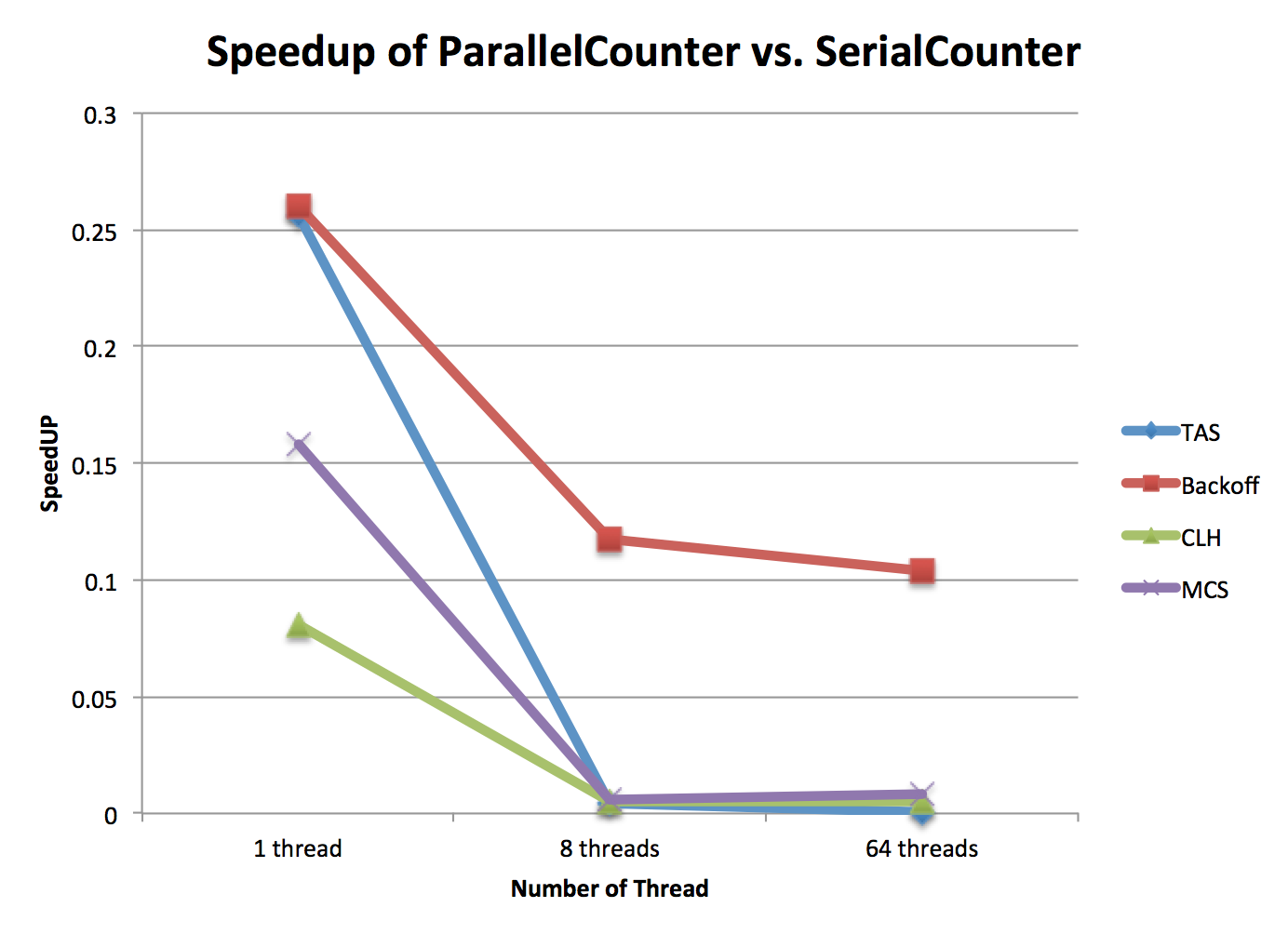
*Fixed delay values running over and over again*:

On the last graph we can see that even when running the **very same delay values** approximately 180 times (~ 45 runs in a batch) we received great difference, **without changing anything in the code itself**.

Based on this understanding, we derived that on our multicore machine, changing the MIN/MAX parameters from its predefined values doesn’t contribute consistently, hence decided to keep the original values unchanged.

*Lock scaling test*

In this part, we tested all locking algorithms in ParallelCounter and compared it to SerialCounter test.

As we mentioned earlier, since the counter is a serial bottleneck, we expected performance degradation, causing the speedup values to be lower than 1.

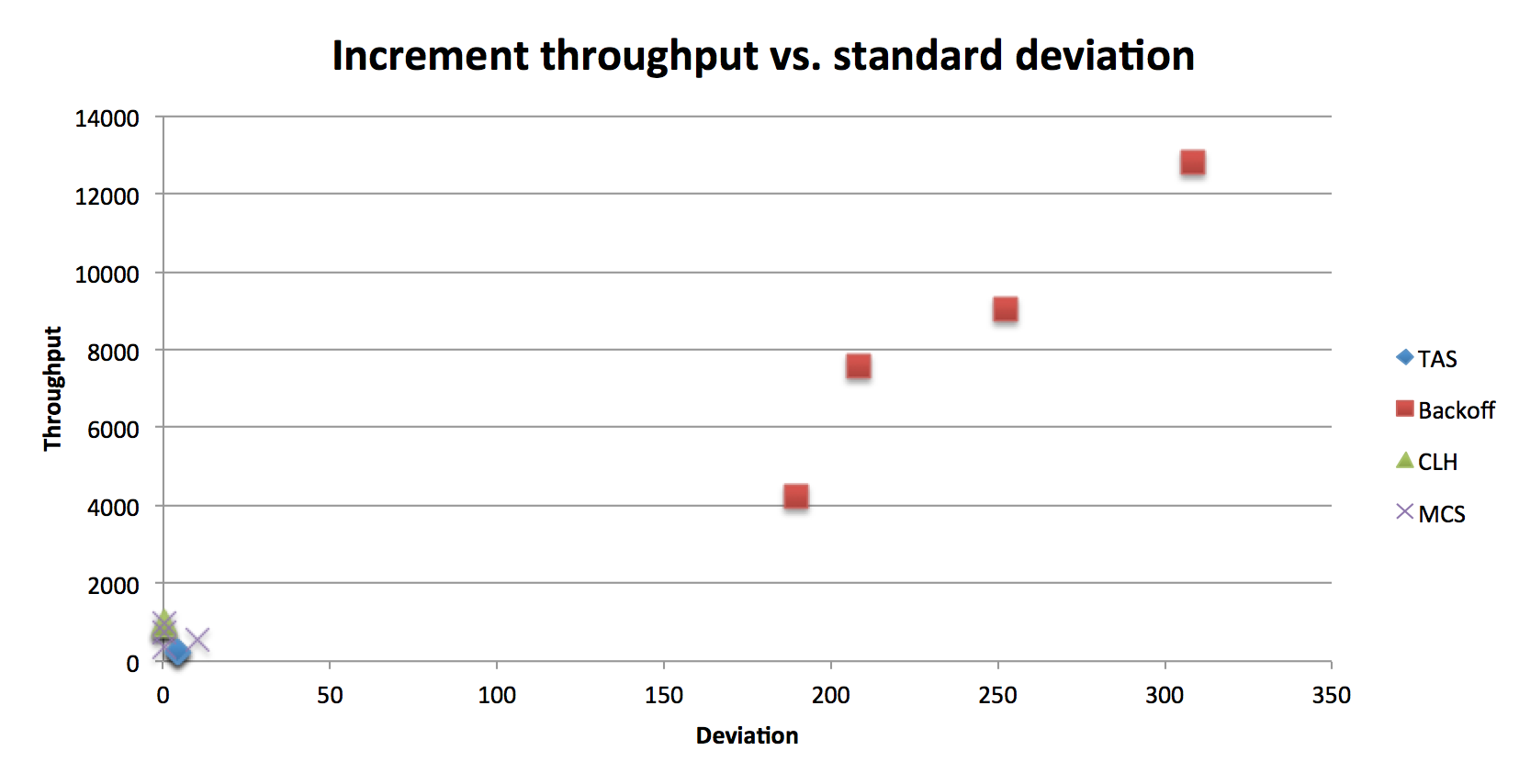
The received results are:

Analyzing the graph, we can immediately see that the Backoff lock achieved that lowest degradation out of all the locks.   
This can be explained since as we’ve seen in the first part of this section, and as we’ll see in the next one, the deviation (which we translated into fairness) of this lock are poor, causing some threads to get much more processor time than others, hence acting more like a serial counter than a parallel one.

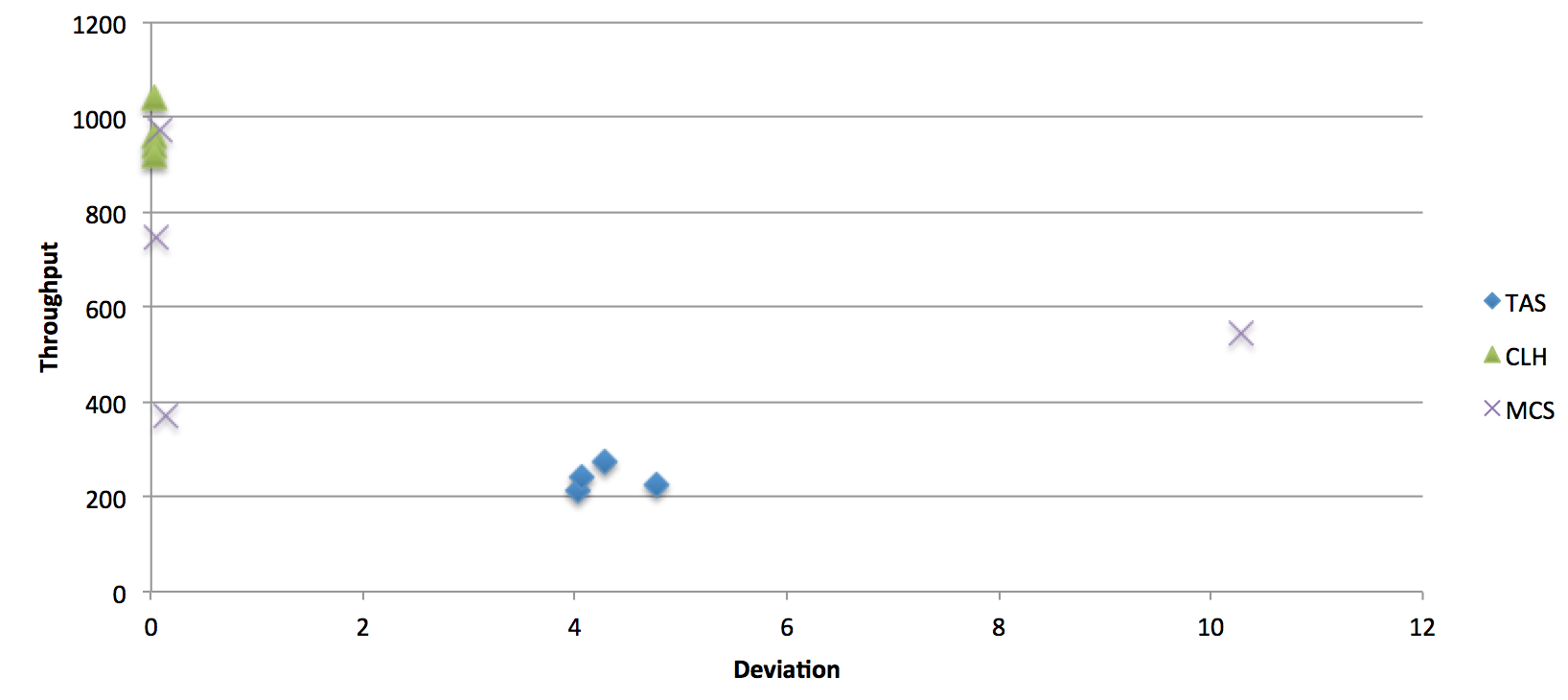
Also, we can see that once a multi-thread process starts, all locks’ (apart from Backoff lock) performance drops drastically.   
This can be explained since our multicore machine is a NUMA machine, all of these locks’ implementation built with threads trying to access each other’s associated memory[[3]](#footnote-3), therefore causing delays and impairs speedup.

On top, once a thread acquires the lock, he needs to transfer data to its associated memory. For many threads, this creates ”ping-pong” of data, in which each of these operation is a costly one.

**Counter Tests- Experiment #3: Fairness**

In this experiment, we tried and compared the standard deviation between the locking algorithms, where in Backoff lock, we fixed the delay values from previous section.

The received results are:



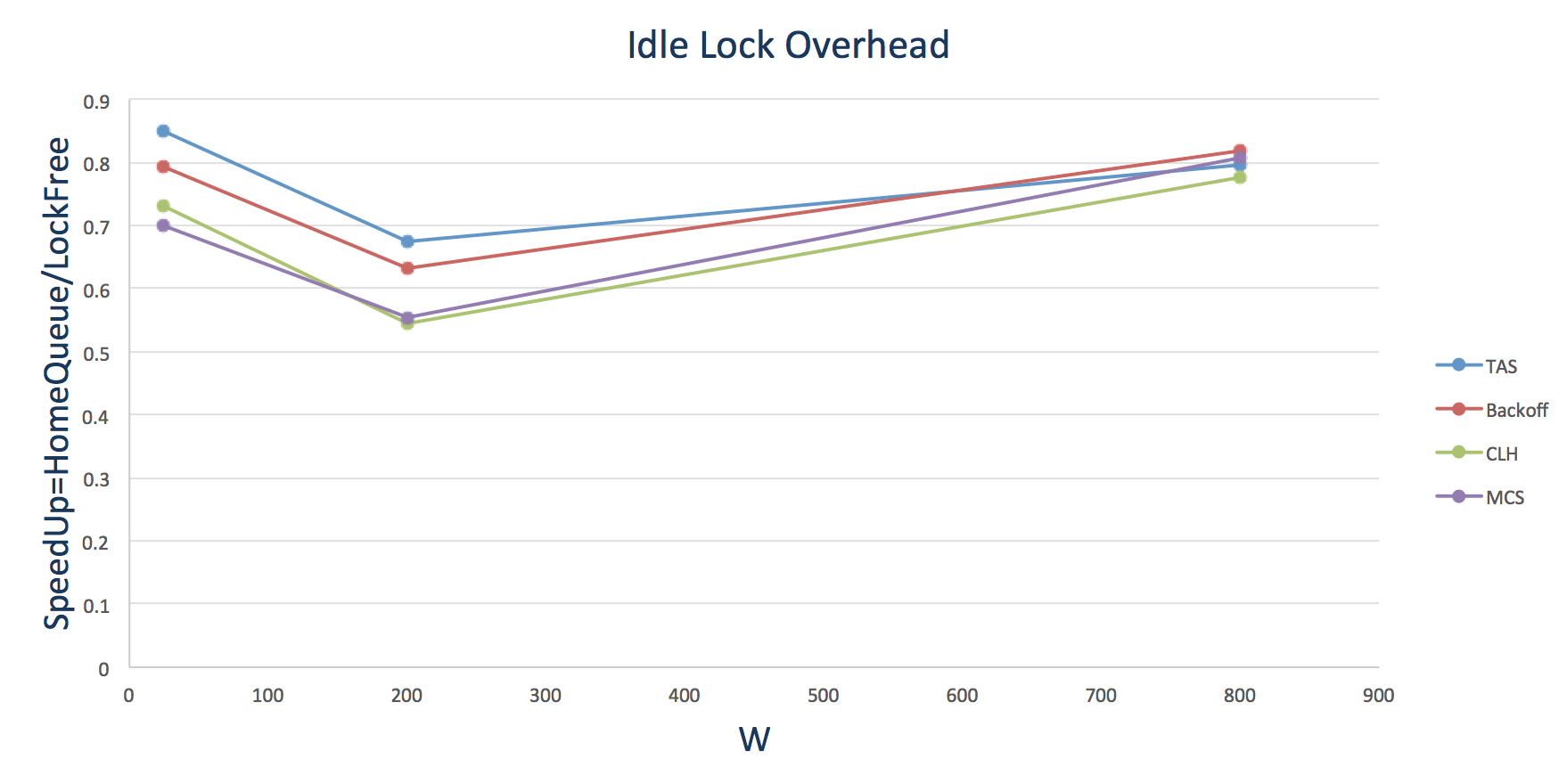
As we can see, the Backoff lock is certainly pops out since its deviation is much bigger than the rest of the locks. We’ve covered all the reasons for that already, so let’s go over to the rest of the locks (2nd graph).

We can see that both CLH and MCS locks have vey minor deviation, if any. This is expected since these locks provide first-come-first-serve implementation, thus creating a **fair environment** for threads to run in.  
(note: we ignore the run with deviation=10 since he is simply not consistent with the rest of its kind, probably because of an interrupt from the OS).

Also, we observe that TAS lock has bigger deviation. This can be explained by the fact that TAS holds no precedence to any thread, or a starvation-free property, thus degrades the fairness.

**Packet Tests- Experiment #1: Idle Lock Overhead**

In this experiment we have compared the throughput of ParallelPacket with two different strategies: LockFree and HomeQueue.  
Specifically, we calculated the speedup between the same running parameters under these two implementations.

Unlike counter test #1, here we gave more weight to the parallel portion of the application, but compared between lock-free implementation and an implementation where locks are in used, but useless (HomeQueue) and as such we expected a smaller degradation so that the speedup ratio will get closer to 1.

The received results are:

We can see that as expected, none of the scenarios resulted a positive speedup (i.e. improvement) by applying the ParallelPacket HomeQueue implementation.

This is expected, since the HomeQueue implementation adds the overhead of the locks, but aren’t actually useful since we use 1-to-1 mapping strategy.

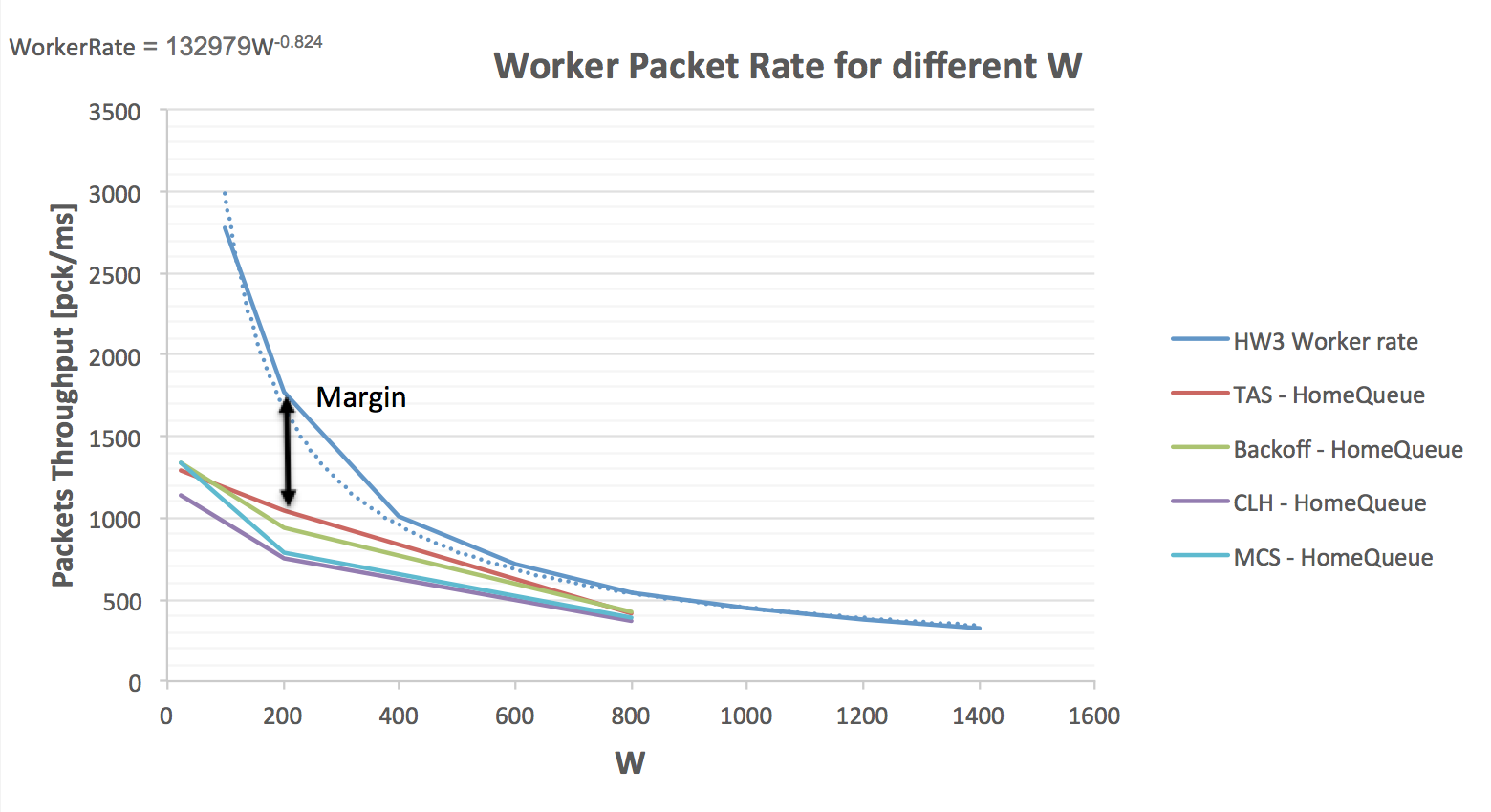
We can also notice that for larger W, the locking overhead takes less from the performance, as both implementations reaches speedup close to 1.

A reasonable explanation for this behavior is that the locking overhead is not affected from W, but as W gets bigger, the relative part of the work overcomes the locking overhead.

These results are consistent with counter experiment #1, since on top of what we already explained, and from the same reasons, we can observe the same difference between the locks speedup, such that TAS and Backoff lock provides better performance comparing to CLH and MCS locks.

Comparing to the WorkerRate results from our last assignment, the rate extracted from this experiment (HomeQueue only) behaves as expected:

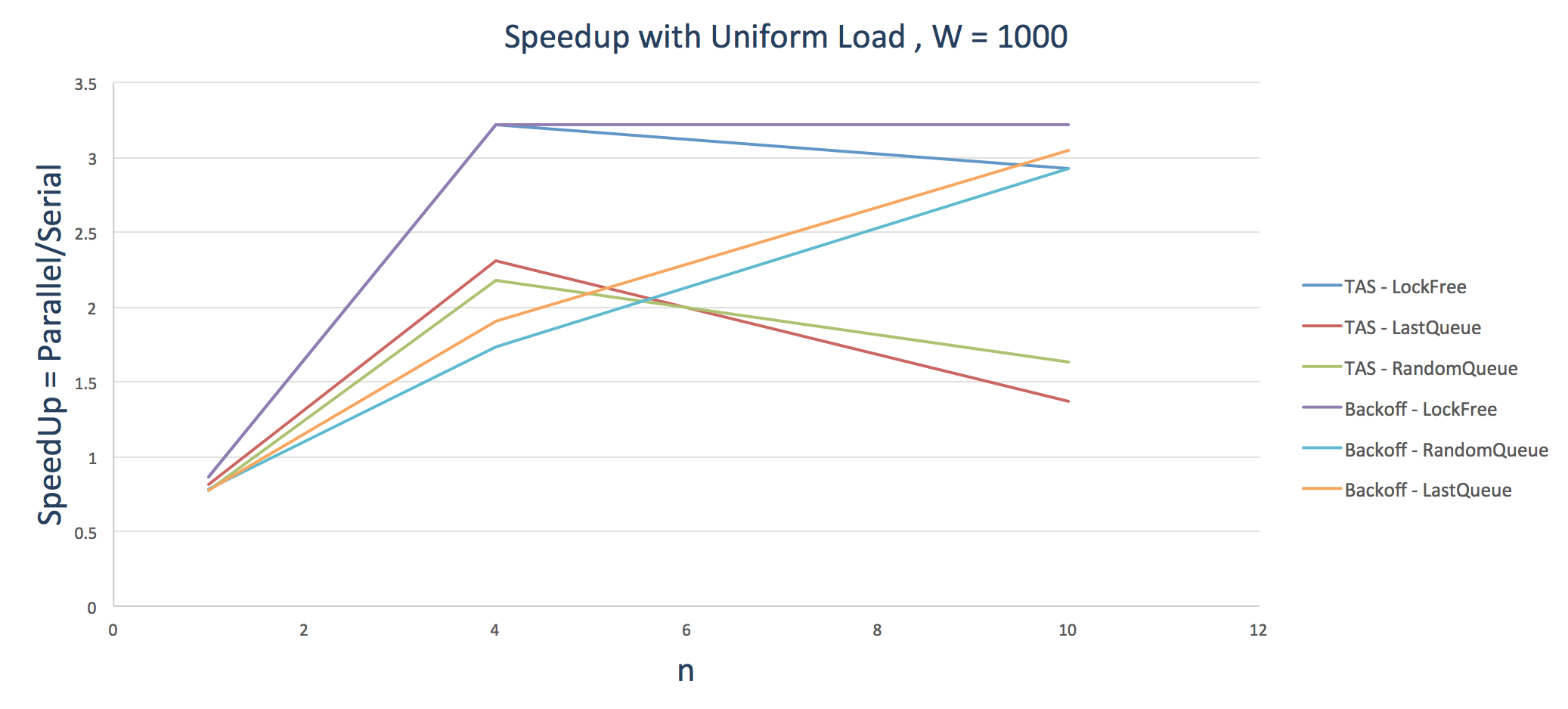
* As W gets lower, the relative portion of the locks gets bigger, eventually lowers the throughput.  
  This is causing a “margin” to be opened between the worker rates of the two assignments.
* As W get bigger, the relative portion of the locks gets smaller, eventually becoming negligible, so that for W large enough, all locks and no-locks worker rate will be equal.

****The following graph demonstrates the explanation above:

Finally, combining these two graphs we can derive that the speedup degradation at originates from the fact that the worker rate drops, while the parallel portion of the program isn’t big enough to compensate that.

**Packet Tests- Experiment #2: Speedup with Uniform Load**

In this experiment, we tested different strategies of locking in order create a better load balancing to the firewall. Specifically, we tested the performance of two locks- TAS and Backoff.

****The received results for are:

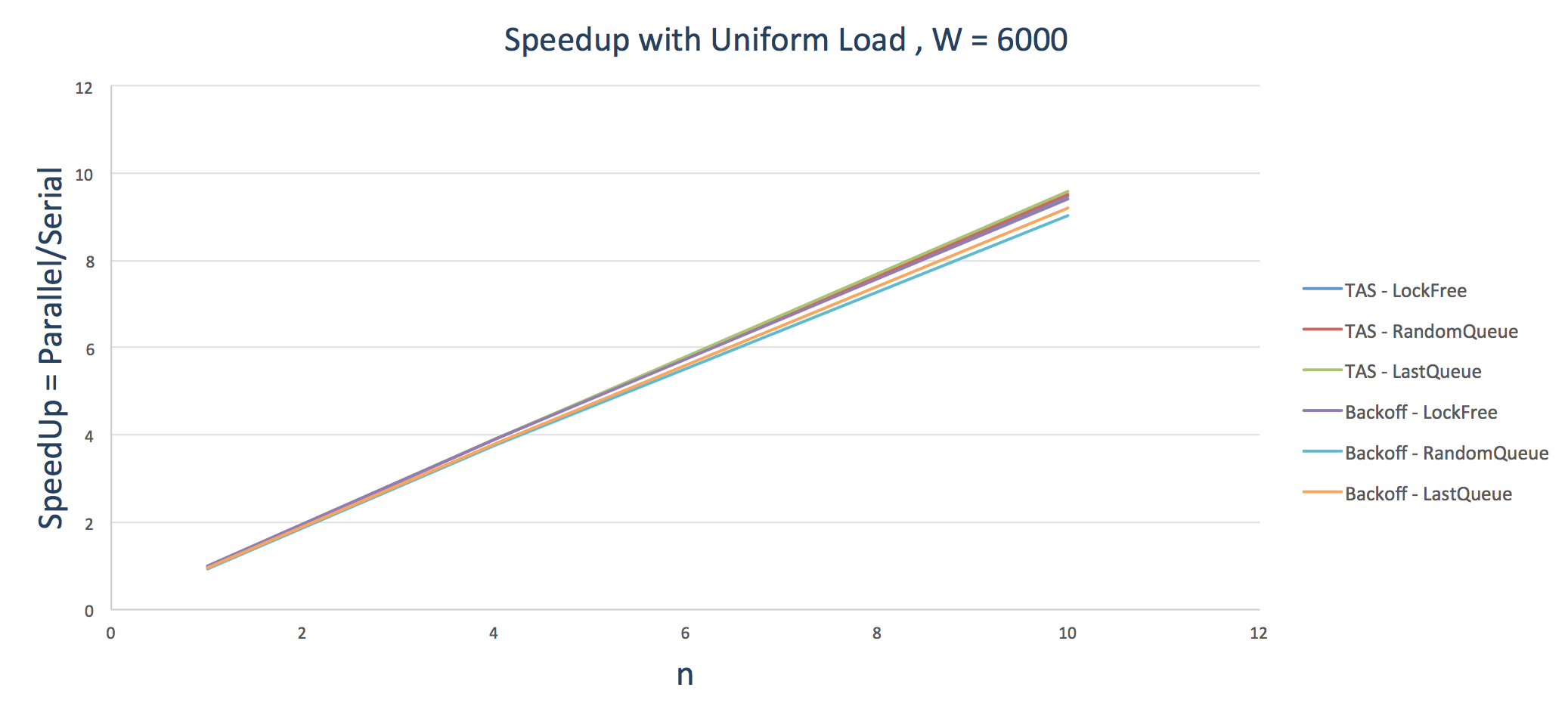
For , we can see that as expected and the biggest speedup achieved when using the LockFree implementation.

We can also see that since the mean processing time is relatively short, the speedup riches into a “ceiling” (~3.2 speedup). This behavior is no different than the one we’ve seen in the previous assignment, as the bottleneck in our system is moving towards the Dispatcher, which works in a serial way (Amdahl’s law).

Based on this “ceiling” value, we can also note that Backoff lock implementation reaches toward that value, whereas TAS implementation drops.

This can be explains since Backoff lock isn’t fair, hence it throws other threads to sleep and uses the timeslot efficiently (but it doesn’t necessary mean that all threads are carrying the same load).

On the other hand, TAS implementation drop because in this algorithm, as we add more threads to the “party”, the contention increases, thus creating a lot of dead time.   
At this processing time is short, more lock() and unlock() operations apply, presenting this issue very clearly.

****The received results for are:

Here, we can see an almost ideal speedup behavior.

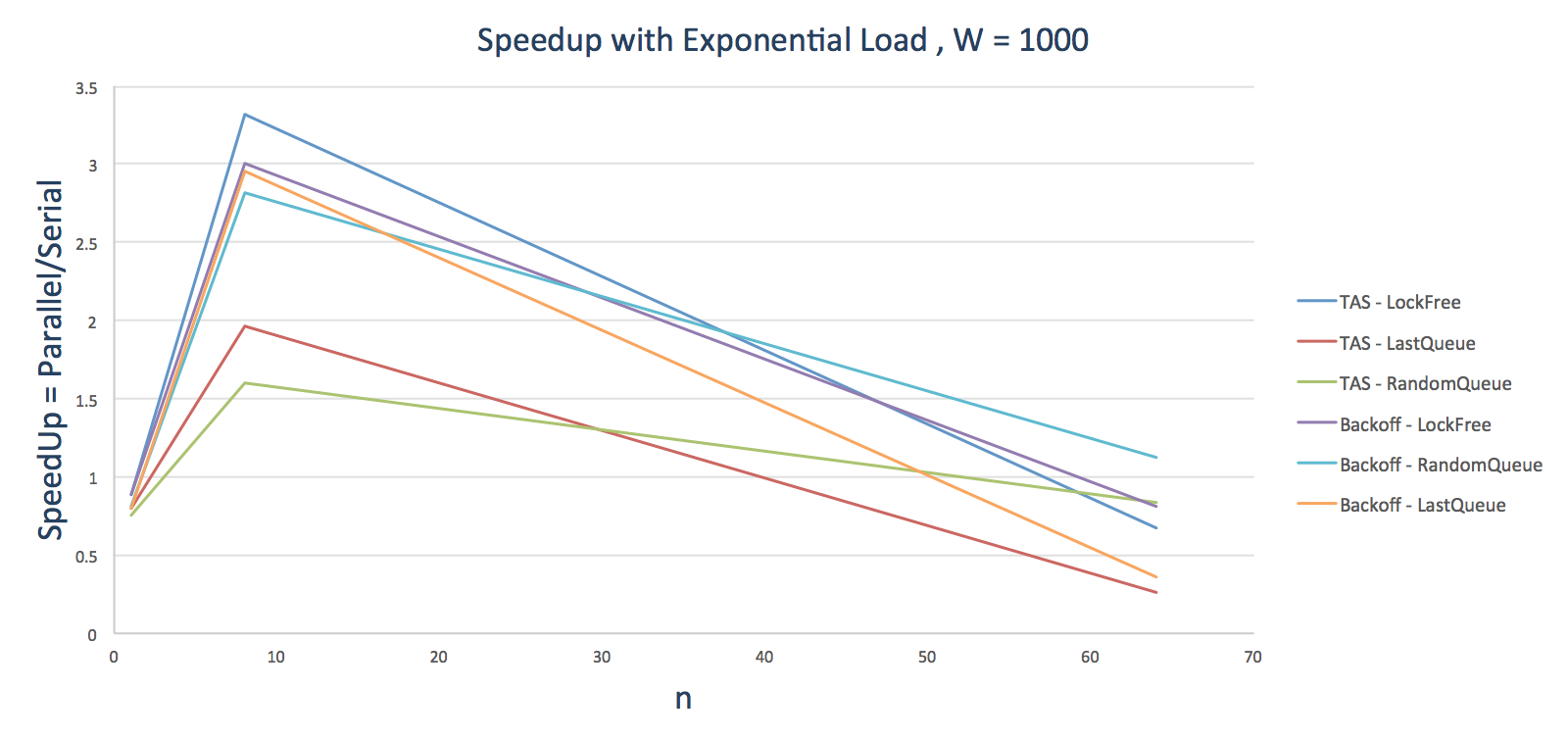
These results can be explained by the fact that similar to our last assignment explanation, the parallel portion of the application increases, thus decreases the ratio of locking overhead versus the rest of the application.

As a consequent, we observe almost no difference between the locks, and between the locks to the LockFree algorithm, which isn’t using any locks.

Also, we can note that since we’re using more threads to complete the packet dequeuing, hence achieve higher speedup as we add more threads to assist.

**Packet Tests- Experiment #3: Speedup with Exponential Load**

In this experiment, we repeated experiment #5 with two locks- TAS and Backoff, but with exponentially load packets.

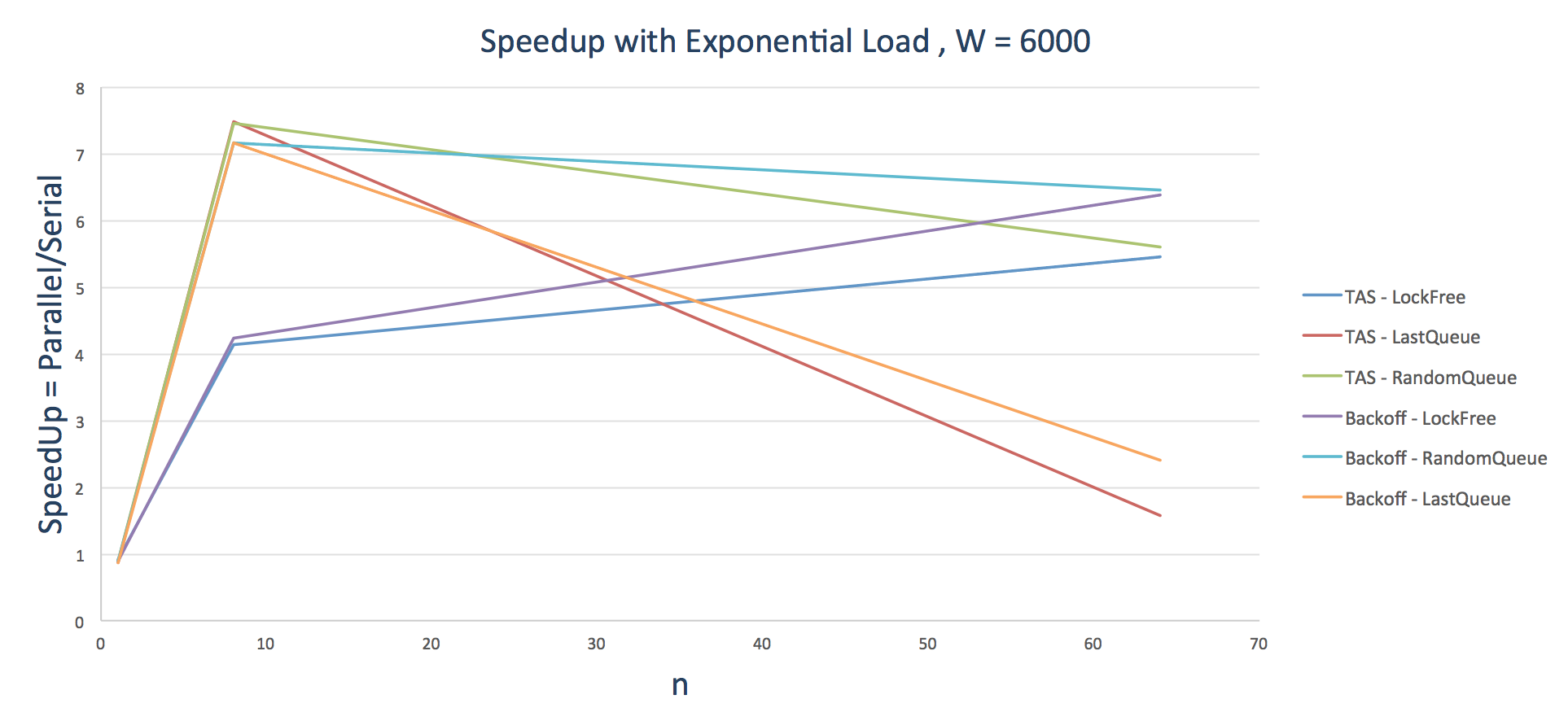
The received results for are:

In this experimsasasa

**Experiment #4: Speedup with Exponential Load**

First of all, we can see that the general behavior explained in previous assignment, in which for we suffer from bigger degradation as the number of threads climbs.   
Specifically, when integrating locks into the implementation, they have a bigger influence on the firewall’s performance.

Based on the graph we can observe two interesting points:

* At 8 threads, the worst implementations belong to TAS lock.  
  This is expected, since this lock has a big contention, so that for any implementation he will lose comparing lockFree and Backoff lock.  
  We’ve witness similar behavior in uniform load tests (10 threads results).
* At 64 threads, all of the implementation suffers from the previously explained degradation.  
  But, out of the implementation, we can see that the LastQueue has the worst performance.  
  This can be explained since LastQueue has the most overhead, and as more threads enters into the game, more and more threads will compete for the same queue, until it is finally free (we cannot see this as uniform load since we didn’t test for so many threads).

The received results for are:

Here we can see immediately better performance at higher threads count since the parallel portion is bigger, so that the bottleneck shifts from the dispatcher to the workers. Therefore, better parallelism-friendly algorithm strives more than those who don’t.

As in the case, we will pick two interesting points in the graph:

* At 64 threads, all of the implementation suffers from degradation, since every core in our machine shares its cache with another thread, hence suffers where both threads need to access memory operations.  
  Out of all implementations, we can see that again, LastQueue implementation suffers the most.  
  This can be explained by the same explanation as observed in , and in our **case the waiting time of each thread on a specific queue gets even longer** because of the higher processing time.
* At 8 threads ….

1. Each of the graphs shows a series where MAX delay has been fixed, and MIN delay changed incrementally [↑](#footnote-ref-1)
2. Deviation results were normalized in running time so that we can connect between increment rate and deviation [↑](#footnote-ref-2)
3. Some implementation more than others, as explained in experiment #1 answer [↑](#footnote-ref-3)