CMSC 23010 Homework 3b Programming Final

Ruolin Zheng

May 20, 2019

The svn submission in the `hw3b` directory includes:

- `Makefile` - instructions for the compiler
- `include/` - a folder containing header files for `driver, common`
- `src/driver.c` - the main program which parses command line arguments and the checksum program with specified STRATEGY and LOCK
- `src/common.c` - functions shared by the main program, `driver.c` and the test program, `driver_output.c`
- `tests/driver_output.c driver_output.h` - the same program as `src/driver.c` except that it outputs the checksum for a specified number of sources and packets. Please see **Correctness Testing** below for details
- `tests/test_driver_out.py` - automated Python script for running `test/driver_output` and comparing the results of SERIAL, LOCKFREE and the possible strategy-lock combinations of MUTEX, CLH with HOMEQUEUE, AWESOME
- `result/` - data from running on SLURM stored in `.txt` and `.csv`
- `plot.ipynb` - Jupyter Notebook file for generating plots

1 Running the Program and the Tests

Please run `make clean` and `make` in the `hw3b` directory. This will create the following binary executables: `./driver, tests/driver_output`.

1.1 The Program: driver

Usage:
`./driver -m NUM_MILLISECONDS -n NUM_SOURCES -w MEAN -u [0|1] -e SEED -d QUEUE_DEPTH -l [mutex|clh] -s [lockfree|homequeue|awesome]`
To run SERIAL: `./driver -m 2000 -n 7 -w 1000 -u 1`
To run MUTEX AWESOME: `./driver -m 2000 -n 7 -w 1000 -u 1 -d 8 -l mutex -s awesome`

1.2 Correctness Testing: driver_output.c and test_driver_output.py

`driver_output` accepts the same parameters as `driver` for configuring the program type, lock, and strategy, except that it accepts `-t NUM_PACKET_PER_SOURCE` instead of `-m NUM_MILLISECOND`. For Correctness Testing, `driver_output` outputs to a `.txt` the sum of all checksums from the $n \times T$ packets (expecting overflow as a defined behavior.) `test_driver_output.py` is an automated script that launches `driver_output` with all possible programs types (SERIAL, PARALLEL), locks (CLH, MUTEX), and strategies (LOCKFREE, HOMEQUEUE, AWESOME). It then compares `serial_res.txt` with all the output from all other combination of program types. This generic Correctness Testing framework applies to all combination of program types.

1.2.1 Design Choices for Correct Testing and Reasoning

1. The program accepts T and n, the number of packets per source and the number of sources. Once the Dispatcher completes the double for-loops, it sets the `timeout_flag`. The Workers only exit when all $T \times n$ packets have been processed, all queues appear empty, and the `timeout_flag` is set.

2. To keep track of the accumulated checksum for all the packets from all the sources, each Worker keeps a local accumulated checksum of all packets it has processed. Upon joining the threads, the Dispatcher collects the local return values into the global checksum. This approach is feasible since **a signed long overflow is a defined behavoir** according to https://en.wikipedia.org/wiki/Integer_overflow.

2 Performance Experiment Results

2.0.1 Notes about Design and Changes in Design

1. In my original design document, I specified that I would compare ALOCK and TAS to determine the effect of lock fairness on load balancing. I also stated that I would exchange ALOCK for CLH if the latter proves more efficient. From my experiments in **HW3a**, CLH is the only lock that ever performs as well as MUTEX. Therefore, for the

purpose of implementing a high-throughput AWESOME strategy, I decided to instead use CLH and MUTEX for this assignment.

2. In accordance with my lock design choices in HW3a, my CLH implementation needs to dynamically allocate memory as threads come in to acquire the locks. This may decrease the program's performance relative to MUTEX.

Each of the following experiments run for 2 seconds. I use the median of 5 runs for Uniform Packets and that of 11 runs for Exponential Packets. The raw data are stored in `result/master*.txt, overhead*.csv, uniform*.csv, exponential*.csv`.

2.1 Idle Lock Overhead

2.1.1 Original Hypothesis Adapted for CLH and MUTEX

1. Because there is no contention, and both CLH and MUTEX are rather fair locks, they incur similar amounts of overhead. Both their Speedup and Worker rate plots should follow similar downward-sloping trends. There shouldn't be a significant difference between their Speedup or Worker Rate.

2. Because of the lock overhead, the Speedup should be consistently below 1.0. As W increases, the Speedup should gradually increase and eventually tend to 1.0. This is because with heavier packets, the number of packets HOMEQUEUE and LOCKFREE can handle both decreases. HOMEQUEUE access the locks less often and incurs proportionally less overhead.

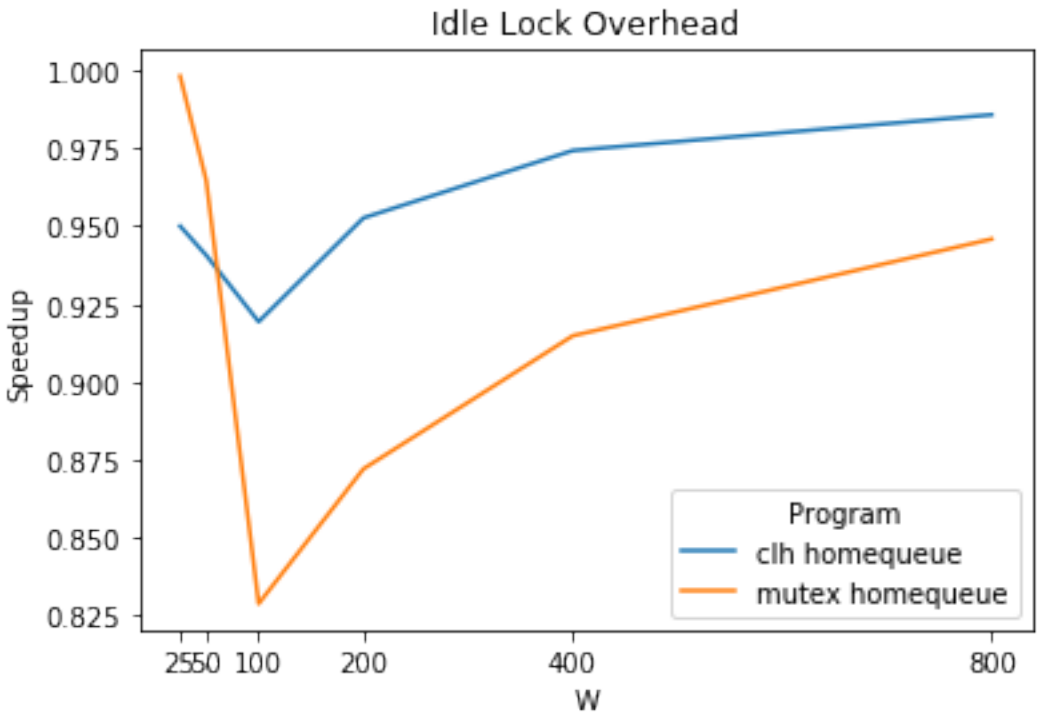
3. For both CLH and MUTEX, the Worker Rate decreases as W increases, since we have fixed time M. Under our assumption that CLH and MUTEX have similar overhead, the amount of decrease in their Worker Rate from $W = 25$ to 800 should be approximately equal.

2.1.2 Data and Graphs

Below is a table displaying the throughput ratio of HOMEQUEUE vs. LOCKFREE and a graph of their Speedup relative to LOCKFREE against W.

W	clh homequeue	mutex homequeue
25	0.950015	0.998141
50	0.940621	0.964571
100	0.919337	0.828886
200	0.952626	0.872104
400	0.974253	0.914832
800	0.985727	0.945864

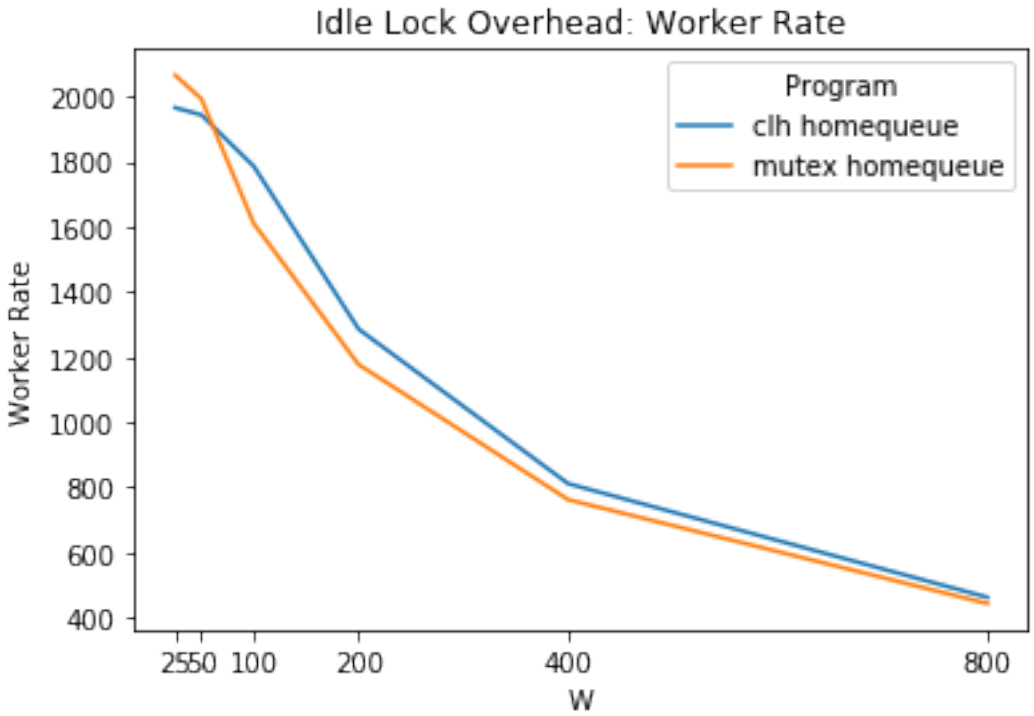
Table 1: HOMEQUEUE Speedup Relative to LOCKFREE



Below is a table displaying the **Worker Rate** of HOMEQUEUE and a graph of Worker Rate against W.

W	clh homequeue	mutex homequeue
25	1966.0	2066.0
50	1944.0	1993.0
100	1786.0	1610.0
200	1286.0	1177.0
400	811.0	762.0
800	462.0	443.0

Table 2: HOMEQUEUE Worker Rate



2.1.3 Observation and Analysis

1. From the graph, the Speedup is indeed consistently below 1.0 as we hypothesized.

2. We hypothesized that CLH and MUTEX should incur a similar amount of overhead. The data show that MUTEX incurs 4% more overhead than CLH at $W = 800$ and about 10% at $W = 100$ or 200. This is not quite significant, but is actually consistent with our observation in **HW3a**: We found that in the case of idle locks and $n = 1$, CLH has about 2x the throughput of MUTEX. As we explained in HW3a, this is possibly because the industrial-level code for MUTEX is not designed or optimized for contention-free situation.

3. For W below 100, the Speedup decreases as W increases. I didn't expect this in my original hypothesis, but this Piazza post (https://piazza.com/class/jtypws5zkcj3i5?cid=157) inspired me to find with an explanation for this case. Essentially, with uniform packets, the performance is much dependent on the coordination in pace between the Dispatcher and the Worker. The Dispatcher needs to generate enough packets to keep all Workers fed, but not too much to fill up the Workers' queues. The Workers need to work fast enough so that the Dispatcher does not spin on their full queues, but not too fast in case they repeated access empty queues, dequeue `NULL` packets, and then yield the CPU.

4. With smaller W, the Workers are able to work faster and prevent the Dispatcher from spinning. Moreover, the time they take to process the small packets may coincidentally match the time it takes for the Dispatcher to cycle through all queues and visit particular queues a second time. However, as W increases from 25 to 100, this intricate balance is disturbed, bringing us back to the point of start of our hypothesized trend.

5. From $W = 100$ to 800, the plots follow the trend I hypothesized. As W increases, for both MUTEX and CLH, the Speedup gradually increases. The Speedup eventually tend to 0.985 for CLH. This is plausible, since when we fix the runtime M, the program can handle fewer packets when they are heavier. Therefore, HOMEQUEUE accesses the locks less often and incurs lower overhead.

6. In summary, for larger W, the Idle Lock Overhead becomes less significant relative to the overall runtime. The overhead for CLH is on average 3% less than that of MUTEX. This is consistent with our Counter Test Idle Lock Overhead result in **HW3a.**

7. From the second graph, Worker Rate decreases as W increases, which aligns with my hypothesis. The Worker Rate for HOMEQUEUE with either lock is smaller than that in **HW2.** From $W = 25$ to 800, CLH's Worker Rate decreases by 76.5%, and MUTEX's Worker Rate decreases by 78.6%.

2.2 Speedup with Awesome

2.2.1 My Awesome Strategy

- I built upon the FULLQUEUE approach I mentioned in my design document by combining it with the LASTQUEUE strategy.

- The Dispatcher broadcasts an integer `last_queue_index` which helps Workers identify the queue and its lock. It updates the pointer in two cases: (1) When it is spinning on a full queue, it updates the index to the full queue; (2) When it has just finished enqueued into a queue, it updates the index to that queue.

- The Workers have access to all n queues, but still prioritize the queue corresponding to their index as in LOCKFREE and HOMEQUEUE. One Worker first tries to dequeue from its own queue, protected by a lock. If the queue is empty, the Worker tries to acquire the lock associated with the gloab `last_queue_index`. If it succeeds, it updates the index to `(index + 1) % n` since the Dispatcher may be just enqueuing into the `(index + 1)`-th queue; it yields the CPU after computing the checksum. If it fails to acquire the lock, it also directly yields the CPU.
- There is a reason why I require a Worker to reset the index after it has successfully acquired the lock of the globally-available queue: if the Worker does not reset the index and the Dispatcher has not been able to set it to a new value, some other available Workers may attempt to access this queue and fail.

Algorithm 1: Dispatcher

```
1  while time do
2      for queue in queues do
3          if enqueue fails then
4              index ← this queue;
5              spin;
6          index ← this queue;
```

Algorithm 2: Worker

```
1  while time do
2      lock;
3      packet ← dequeue my queue;
4      unlock;
5      if packet is NULL then
6          if trylock index succeeds then
7              index ← (index + 1) % n;
8              packet ← dequeue index;
9              if packet is not NULL then
10                 getFinger(packet)
11             else
12                 yield;
13     else
14         yield;
```

2.2.2 Design Choices and Possible Influence on Performance Testing Results

1. `%` is an expensive operation. Therefore, since we set d to 8 in this assignment, I used the bitwise operators and let my program reject the input if d is not 2^x for some x.

2. I hypothesize that this approach should work well for both UNIFORM and EXPONENTIAL packets, since Workers on empty queues can always identify a queue to work on. For EXPONENTIAL, it is also very likely that the particular queue is full and blocking the Dispatcher from assigning work to other available Workers.

3. Moreover, my strategy would remain very efficient even if the number of threads exceeds the number of cores. In LOCKFREE and HOMEQUEUE, workers are fixed to queues. When there are more threads than cores, the enforced serialization will cause some Workers to repeatedly encounter empty queues and yield, while other Workers' queues remain full. In my strategy, Workers can access the fullest queue or the queue the Dispatcher has just enqueued into. This reduces the problem of forced serialization. My strategy should be similarly efficient for both CLH and MUTEX.

4. My strategy ensures that the Dispatcher does not drop packets and that no single queue is "ignored" by all the Workers. This is because I require the Workers to prioritize their own queues before attempting the globally-available one.

I run the following tests for Uniform and Exponential Load for all combinations of locks and strategies, namely SERIAL, LOCKFREE, MUTEX HOMEQUEUE, MUTEX AWESOME, CLH HOMEQUEUE, CLH AWESOME.

2.3 Speedup with Uniform Load

2.3.1 Original Hypothesis Adapted for CLH and MUTEX

1. We have five curves for each W, LOCKFREE, MUTEX HOMEQUEUE, MUTEX AWESOME, CLH HOMEQUEUE, CLH AWESOME. As long as the number of threads is smaller than the number of cores, all these curves should slope upward.

2. Since the SLURM machines have 16 cores, we expect each curve to peak at $n+1 = 14$. There should be a sharp decrease in Speedup at $n = 27$, since the hardware constraint forces part of the program to execute serially.

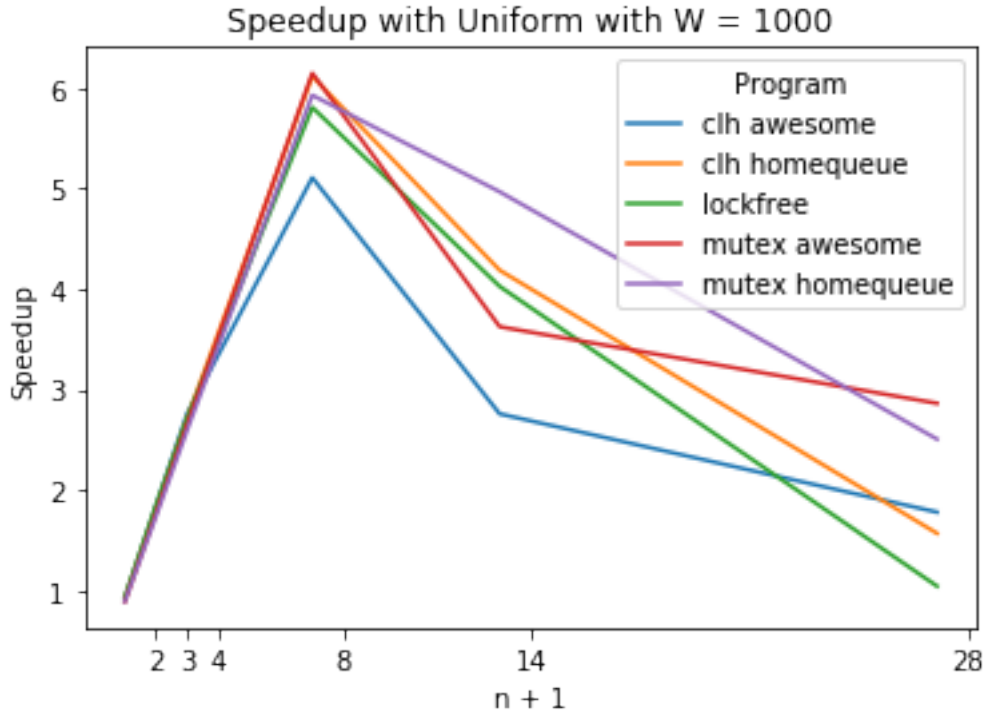
3. Our ideal Speedup is n-fold, where n is the number of threads. For either MUTEX and CLH, we expect LOCKFREE to be closer to the ideal than HOMEQUEUE. Since my AWESOME strategy in part reduces the chance of Dispatcher spinning or Worker yielding, I hypothesize that it should lie above LOCKFREE. Since both CLH and MUTEX scale well even with contention, their performance with the HOMEQUEUE strategy and their performance gain from AWESOME should be similar.

4. I hypothesized in **1. Idle Lock Overhead** that the overhead becomes less significant as W increases. Therefore, with increasing W, we expect the Speedup to be closer to the ideal n-fold for both CLH and MUTEX.

2.3.2 Data and Graphs

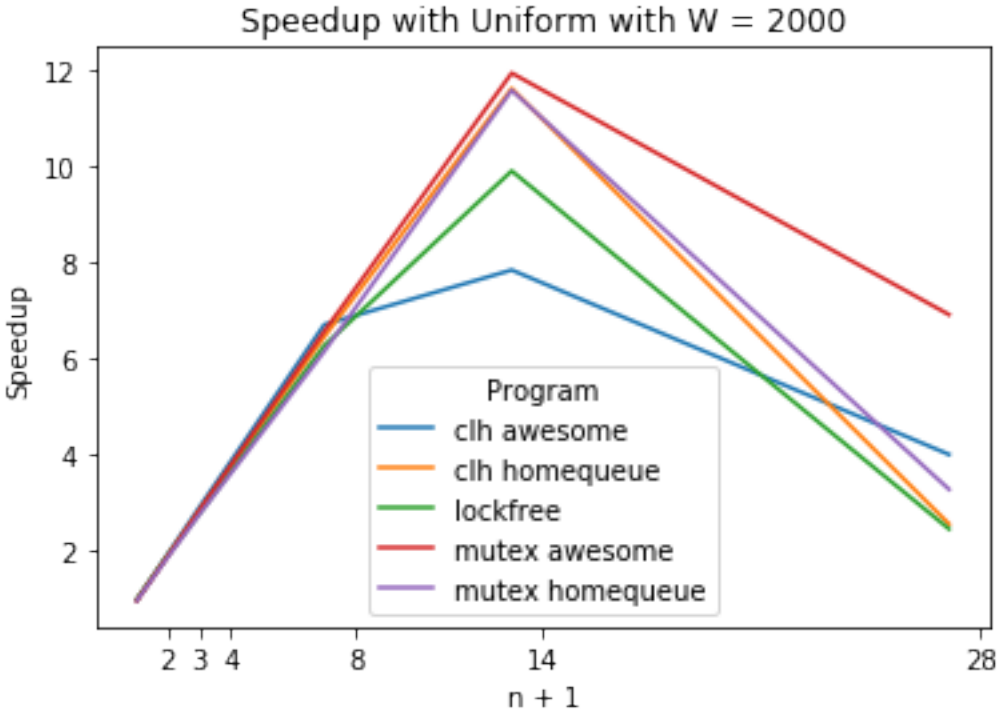
n	clh awesome	clh homequeue	lockfree	mutex awesome	mutex homequeue
1	0.925826	0.925262	0.936230	0.892713	0.893472
2	1.849138	1.825568	1.850476	1.775845	1.742910
3	2.763411	2.719987	2.721474	2.661429	2.601850
7	5.106652	6.114621	5.805225	6.145542	5.926584
13	2.758132	4.186333	4.025248	3.624687	4.966193
27	1.780912	1.570155	1.045049	2.862886	2.506850

Table 3: Uniform, $W = 1000$



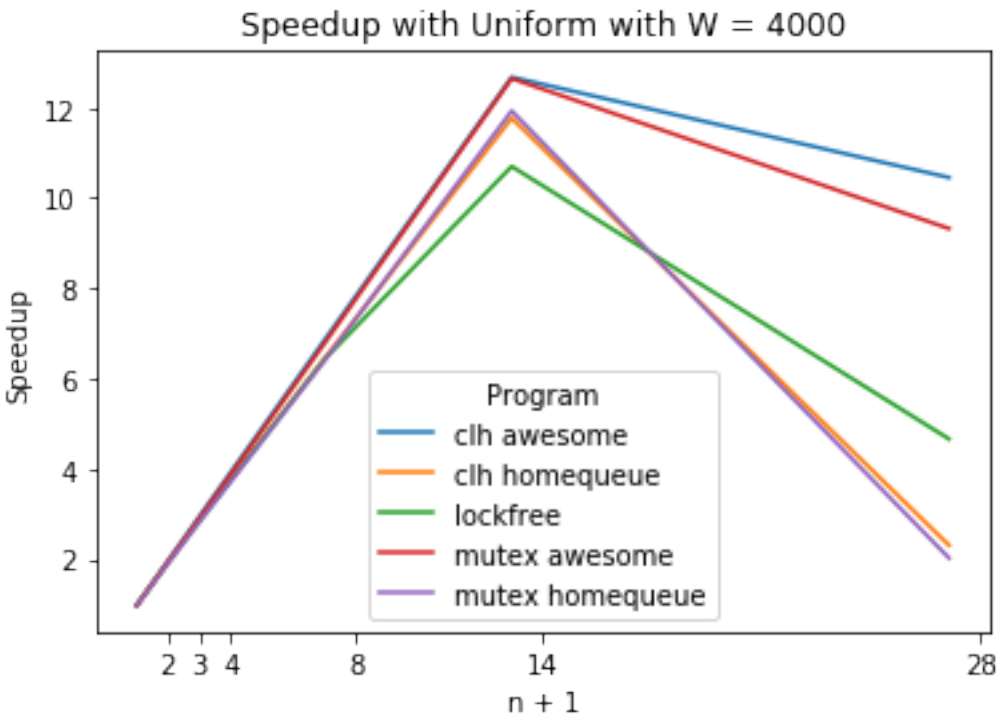
n	clh awesome	clh homequeue	lockfree	mutex awesome	mutex homequeue
1	0.961614	0.961590	0.967127	0.942283	0.942981
2	1.921313	1.889776	1.899202	1.887126	1.856701
3	2.888571	2.813527	2.758725	2.829613	2.736442
7	6.698386	6.444914	6.260009	6.560967	6.141476
13	7.836589	11.621205	9.902005	11.942618	11.583396
27	3.989753	2.540748	2.436809	6.905633	3.263506

Table 4: Uniform, $W = 2000$



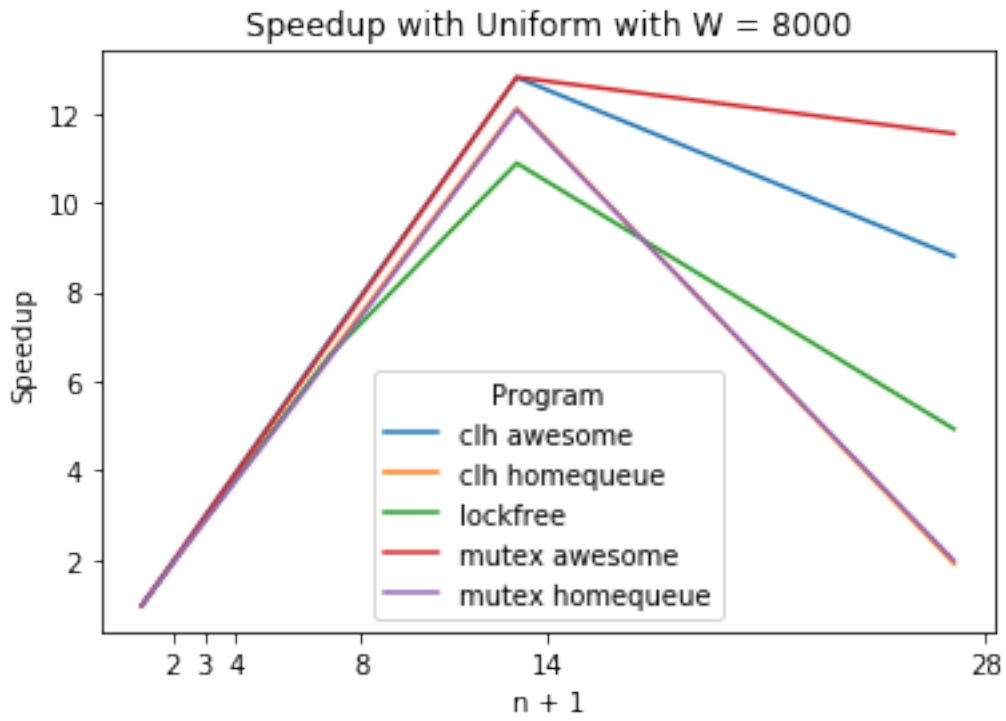
n	clh awesome	clh homequeue	lockfree	mutex awesome	mutex homequeue
1	0.980399	0.980047	0.983183	0.970306	0.970340
2	1.960869	1.926218	1.929480	1.943171	1.904749
3	2.941808	2.862845	2.844075	2.900276	2.817688
7	6.846854	6.438685	6.424442	6.775235	6.391859
13	**12.679513**	11.770613	10.701477	**12.649972**	11.932156
27	**10.455430**	2.303699	4.660276	**9.323940**	2.021633

Table 5: Uniform, $W = 4000$



n	clh awesome	clh homequeue	lockfree	mutex awesome	mutex homequeue
1	0.990192	0.990040	0.991320	0.985074	0.985008
2	1.975139	1.942849	1.939588	1.967067	1.931865
3	2.964873	2.890340	2.870226	2.950012	2.850092
7	6.920391	6.574005	6.562817	6.871013	6.497902
13	12.796185	12.107381	10.881164	12.802238	12.065562
27	**8.795047**	1.923408	4.931509	**11.539343**	1.990861

Table 6: Uniform, $W = 8000$



2.3.3 Observation and Analysis

1. From the graph, LOCKFREE appears to have a lower Speedup than HOMEQUEUE, which is quite unexpected. However, as I've mentioned before, the added overhead of acquiring the uncontended lock may be able to tune the Dispatcher-Worker pace. This may potentially results in the Worker wasting less time dequeuing `NULL` packets and

2. As I hypothesized, when W is above 1000, all curves slope upward and peak at $n = 13$, the number of threads the SLURM hardware supports. As for $W = 1000$, it is possible that when n is as large as 13, the Workers work too fast. The Dispatcher cannot keep all Workers fed. From our result in **HW2**, with $W = 1$, the Dispatcher Rate is 1539 packets per millisecond when $n = 9$ and 2124 when $n = 14$. The number of packets per millisecond for each worker is $1539/9 = 171$ and $2124/14 = 152$.

3. There is a sharp decrease in Speedup at $n = 27$ as I hypothesized, due to hardware constraint. Noticeably, at $n = 27$, the AWESOME strategies are significantly better than HOMEQUEUE with the corresponding lock. The most noteworthy cases occur at larger W, in alignment with my strategy design. When $W = 4000$, CLH AWESOME achieved 4.6x the Speedup of CLH HOMEQUEUE, and MUTEX AWESOME also achieved 4.6x the Speedup of MUTEX HOMEQUEUE. When $W = 8000$, CLH AWESOME achieved 4.6x the Speedup of CLH HOMEQUEUE, and MUTEX AWESOME also achieved 4.6x the Speedup of MUTEX HOMEQUEUE.

4. When the number of threads is smaller than the number of cores, LOCKFREE and HOMEQUEUE can achieve quite decent Speedup, about $0.9n$ on average for large W such as 4000. Therefore, the performance gain of 1.08x with AWESOME instead of HOMEQUEUE does not appear as significant as in the point above.

5. As in my hypothesis, with large W, my AWESOME strategies approach the ideal n-fold Speedup. We observe this in the boldface table entries: Between $n = 2$ to 13, when $W = 4000$, CLH AWESOME's Speedup is $0.98n$ on average, and CLH AWESOME's Speedup is $0.97n$ on average. Similarly, when $W = 8000$, CLH AWESOME's Speedup is $0.985n$ on average, and CLH AWESOME's Speedup is $0.98n$ on average.

6. For $W = 1000, 2000$, CLH AWESOME has the lowest Speedup when the number of threads is smaller than the number of cores. However, it is worth pointing out that CLH AWESOME outperforms CLH HOMEQUEUE when $n = 27$. The reasoning is the same with my design for AWESOME: the Workers can access the fullest queue or the queue the Dispatcher has just enqueued into, reducing the problem of forced serialization.

7. In conclusion, these points prove that our AWESOME strategy is indeed very efficient for both CLH and MUTEX with UNIFORM packets, especially for larger W or when there are more threads than cores.

2.4 Speedup with Exponential Load

2.4.1 Original Hypothesis Adapted for CLH and MUTEX

1. The main difference between EXPONENTIAL and UNIFORM is the former's added variability. This leads to some Worker constantly getting heavy packets and maintaining full queues. This blocks the Dispatcher from assigning work to other available Workers.

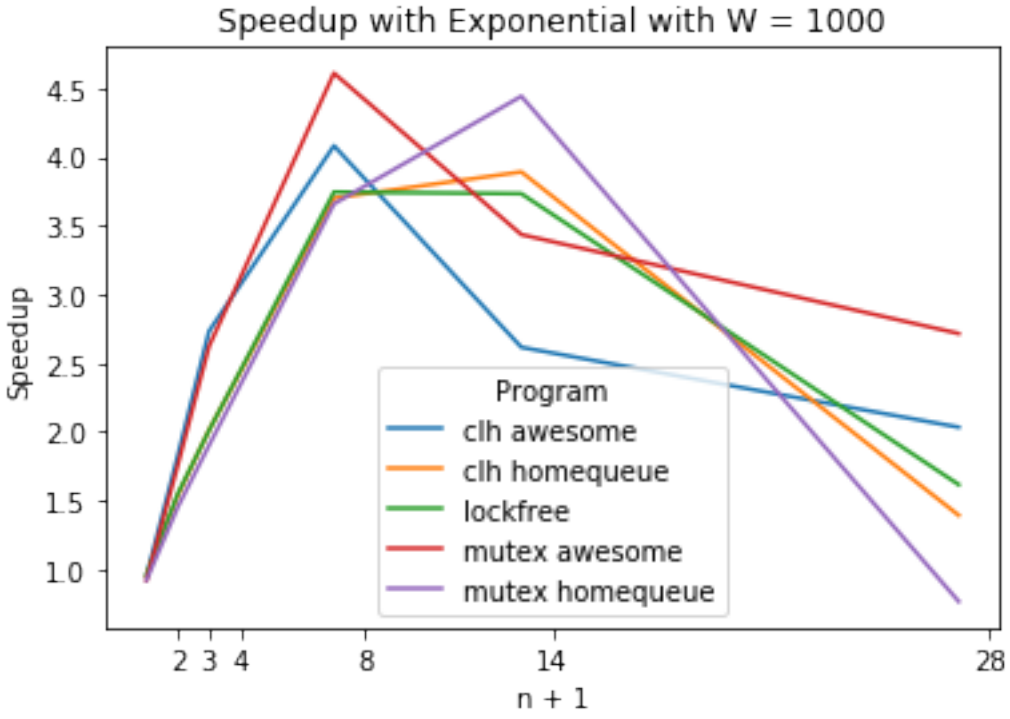
2. All the curves should still slope upward, and we still expect them to peak at $n = 13$. However, at larger values of W, the heavier packets become even heavier, which negatively impacts overall Speedup. Therefore, the peak Speedup of larger W should be smaller than that of smaller W.

3. As we've observed in **HW2**, the Speedup with exponential packets is smaller than that with uniform packets for the exact reason we have in the first point. This should still hold true for LOCKFREE and HOMEQUEUE. However, by the design of my AWESOME, I hypothesize that it will achieve peak Speedup similar to that with uniform packets at larger W such as 8000.

2.4.2 Data and Graphs

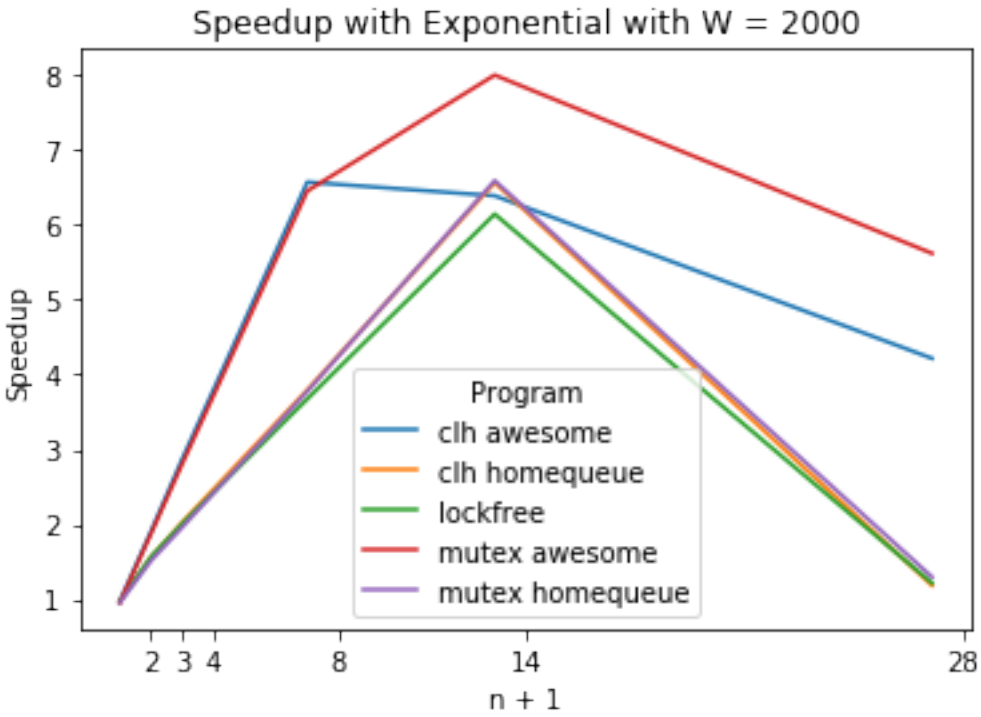
n	clh awesome	clh homequeue	lockfree	mutex awesome	mutex homequeue
1	0.948085	0.947746	0.958775	0.916876	0.918039
2	1.824119	1.534714	1.547302	1.754784	1.449354
3	2.731269	2.009284	2.008326	2.621170	1.899069
7	4.082885	3.701842	3.745670	4.611840	3.659626
13	2.614281	3.892269	3.733764	3.435594	4.443702
27	2.031875	1.390120	1.613430	2.714537	0.761885

Table 7: Exponential, $W = 1000$



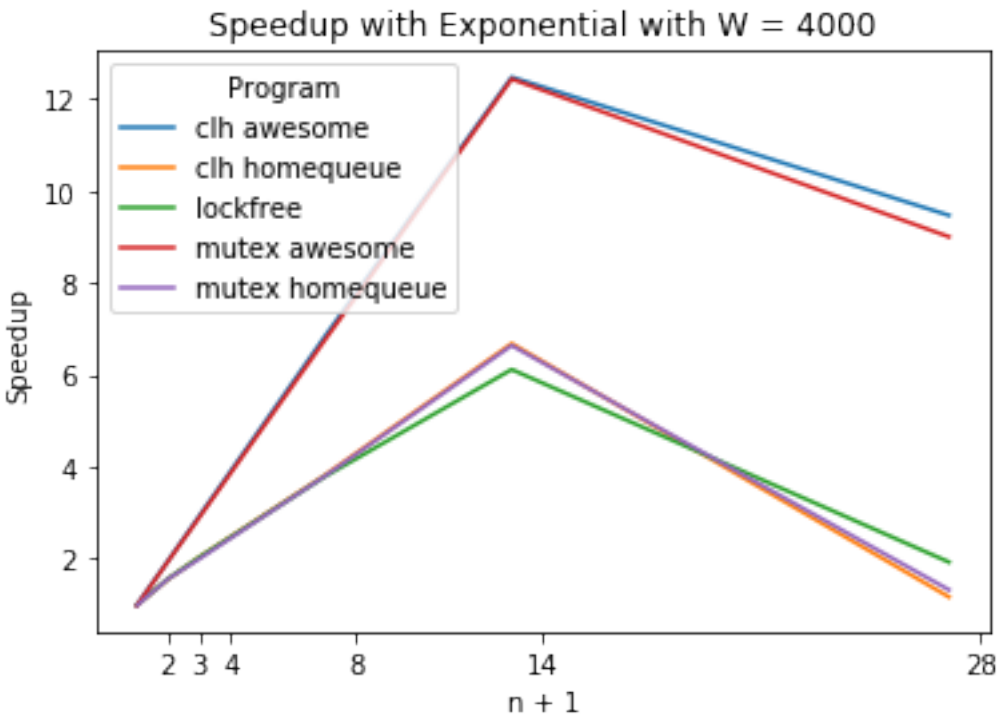
n	clh awesome	clh homequeue	lockfree	mutex awesome	mutex homequeue
1	0.973739	0.973426	0.978738	0.955031	0.955468
2	1.908980	1.561193	1.568615	1.879012	1.515982
3	2.876848	2.028083	2.012504	2.811538	1.963791
7	6.571582	3.795115	3.674800	6.452941	3.776603
13	6.388972	6.563701	6.144272	8.001153	6.591247
27	4.217360	1.184578	1.216607	5.619823	1.297463

Table 8: Exponential, $W = 2000$



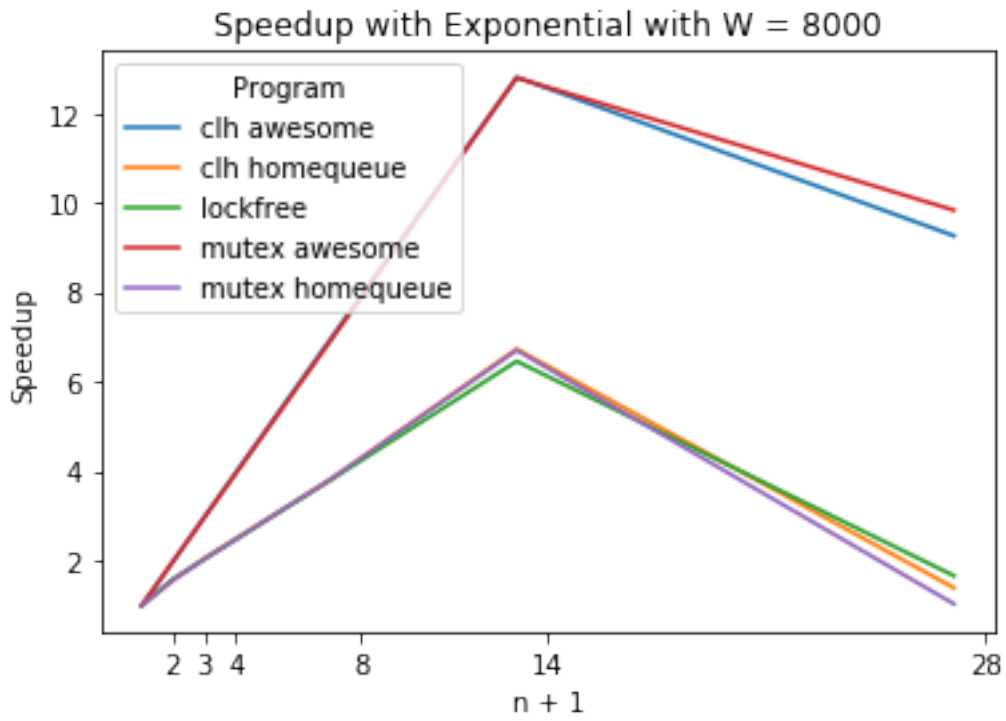
n	clh awesome	clh homequeue	lockfree	mutex awesome	mutex homequeue
1	0.986861	0.986422	0.989318	0.976476	0.976017
2	1.952196	1.549912	1.558603	1.936332	1.552434
3	2.925266	2.025884	2.025927	2.893234	1.989668
7	6.811406	3.800062	3.772593	6.731615	3.780357
13	**12.468637**	6.674215	6.109776	**12.428600**	6.631820
27	9.462146	1.161408	1.919387	8.995497	1.315278

Table 9: Exponential, $W = 4000$



n	clh awesome	clh homequeue	lockfree	mutex awesome	mutex homequeue
1	0.993422	0.993384	0.994607	0.988171	0.988313
2	1.980251	1.556865	1.580621	1.970490	1.558517
3	2.969815	2.040614	2.024908	2.950687	2.022094
7	6.913804	3.788477	3.776587	6.867604	3.790812
13	**12.805381**	6.724865	6.452046	**12.791752**	6.698045
27	**9.256955**	1.383013	1.652487	**9.830493**	1.022298

Table 10: Exponential, $W = 8000$



2.4.3 Observation and Analysis

1. At $W = 1000$, similar to uniform, the patterns in the graphs are not very clear. For example, from $n = 3$ to 8, AWESOME for either lock is on average 1.2x better than HOMEQUEUE. However, at $n = 13$, both HOMEQUEUE curves outperform AWESOME. At $n = 28$, MUTEX AWESOME is 1.3x better than MUTEX HOMEQUEUE and CLH AWESOME is 1.1x better than CLH HOMEQUEUE. Again, this is because the relatively light packets and the added variability give rise to the Dispatcher-Worker pacing problem. We may be able to reach more reliable and meaningful conclusions for larger W.

2. At $W = 2000$, as I hypothesized, MUTEX AWESOME is on average 1.3x better than MUTEX HOMEQUEUE, and CLH AWESOME is on average 1.2x better than CLH HOMEQUEUE.

3. The patterns for sufficiently large W such as 4000 and 8000 are almost identical for both locks. From $n = 3$ to 13, AWESOME's Speedup increases linearly at a faster rate (on average over 1.5x) than HOMEQUEUE's. Similar to uniform, AWESOME remains at 0.98% of our ideal n-fold Speedup. At $n = 27$, HOMEQUEUE's Speedup decreases drastically to about 20% of its value at $n = 13$, while AWESOME's only decreases to 70% of its peak value.

4. On a side note, CLH AWESOME actually outperforms MUTEX AWESOME at $n = 28$ by 1.05x at $W = 4000$. This may not be significant, but we did observe that CLH

outperformed MUTEX by 1.05x in **HW3a** when $n = 8$ and $BIG = 10^8$. Similar to our observaion in UNIFORM, the speedup with my AWESOME strategy appears to be constrained by the physical number of cores we are using, regardless of the packet types.

5. In summary, the load balancing scheme AWESOME played an important role in determining PARALLEL's scalability, regardless of whether we are using CLH or MUTEX.

2.5 Scalability Summaries for the Speedup with Awesome

Below is a summary of the Speedup from Awesome. Please refer back to the previous sections for specific details and analyses.

As I described in earlier sections, I designed my AWESOME strategy to work efficiently for locks that are not sensitive to contention such as CLH and MUTEX. As I expect, it works especially well for EXPONENTIAL, larger W, and situations in which threads outnumber the cores. As we saw from earlier data, in these situations, we achieve close to ideal n-fold Speedup with AWESOME.

n	clh	mutex	n	clh	mutex
1	1.000153	1.000067	1	1.000038	0.999856
2	1.016620	1.018222	2	1.271948	1.264337
3	1.025787	1.035058	3	1.455354	1.459223
7	1.052690	1.057420	7	1.824956	1.811645
13	1.056891	1.061056	13	1.904184	1.909774
27	**4.572638**	**5.796157**	27	**6.693323**	**9.616073**

Table 11: $W = 8000$, Awesome vs. HomeQueue Speedup Ratio, Left: Uniform, Right: Exponential

n	clh uniform	mutex uniform	clh exp	mutex exp
1	0.998862	0.993700	0.998809	0.993529
2	1.018329	1.014167	1.252831	1.246656
3	1.032976	1.027798	1.466642	1.457196
7	1.054485	1.046961	1.830702	1.818469
13	1.175994	1.176550	1.984701	1.982589
27	1.783439	2.339921	**5.601831**	**5.948907**

Table 12: $W = 8000$, Awesome vs. LockFree Speedup Ratio

2.6 Reflection and Possible Improvements

1. In my AWESOME strategy, I require the Worker who has acquired the lock to the globally-available queue to reset the index to `(index + 1) % n`. Randomizing the index may also be a feasible approach, and may be especially suitable in the case of added variability with exponential packets.

2. In my implementation, the Worker gives up accessing the globally-available queue by yielding the CPH if it fails its first `trylock` attempt. Alternatively, I can let the Worker sleep for a minimal amount with `usleep(10)` and then make more attempts. If I can fine-tune this interval parameter, some other Worker or the Dispatcher may be able to reset the index and allow the Worker to find some work to do. This could potentially increase the performance of my program.

3. The performance of AWESOME also depends on the amount of overhead. My CLH is almost as efficient as MUTEX, and thus they attain similar Speedup gains. ALOCK should also achieve similar Speedup as these two locks. It is possible to get better performance with an especially efficient lock such as a fine-tuned EXPONENTIAL BACKOFF LOCK. We will need to avoid contention-sensitive locks such as TAS and TTAS.