CMSC 23010 Homework 2 Theory

Ruolin Zheng

April 24, 2019

Q25

No. From the definition of linearizability, L1 requires that all method calls must have returned in the sequential history S; L2 requires that there is a sequential history S that preserves the order that we observe in a program execution history H. This is similar to the property of program order. Sequential consistency requires that: the method calls are sequential; and, the method calls should appear to take effect in program order. By dropping L2, we are making no assumption about the program order. Therefore, the definition of linearizability without L2 is too weak to equal sequential consistency.

Q29

Yes. By definition, an object is wait-free if its method calls are wait-free; and a method call is wait-free if every call finishes its execution in a finite number of steps. The statement claims that every thread completes an infinite number of method calls. FSOC, suppose x is not wait-free and there is some call that does not finish its execution in a finite number of steps. Then, at least one thread would not be able to complete an infinite number of method calls - a contradiction. Therefore, this definition is equivalent to saying that object x is wait-free.

Q30

Yes. By definition, an object is lock-free if its method calls are lock-free; and a method call is lock-free if it guarantees that infinitely often some method call finishes in a finite number of steps. The statement only claims that an infinite number of method calls are completed. FSOC, suppose x is not lock-free and there is no guarantee that any method call finishes in a finite number of steps. Then, it is not possible that an infinite number of method calls are completed - a contradiction. Therefore, this definition is equivalent to saying that object x is lock-free.

Q31

Assume that a thread calls method m a finite number of times, this method is indeed wait-free because every call finishes in a finite number (2^i) of steps; this method is not bounded wait-free because 2^i is not bounded by a constant and can grow arbitrarily large for large values of i.

Q32

Consider two writers, T_0 and T_1, and one reader, T_2. T_0 calls `enq(x)`, and T_1 calls `enq(y)`. In the example below, the two calls to `enq` is not linearized in the order that T_0 and T_1 execute line 15.
T_0 executes line 15, gets `i = 0` and increments `tail` by 1. `tail` is now 1.
T_1 executes line 15, gets `i = 1` and increments `tail` by 1. `tail` is now 2.
T_1 executes line 16, writes `y` into `items[1]`.
T_2 calls `deq`, reads `items[0]` to be `null`, and therefore returns `items[1]` which has value `y`.
T_0 executes line 16, writes `x` into `items[0]`.
Although T_0 executes line 15 before T_1, we fail to linearize the execution history `enq(x), enq(y), deq(y)`.
Similarly, in the example below, the two calls to `enq` is not linearized in the order that T_0 and T_1 execute line 16.
T_0 executes line 15, gets `i = 0` and increments `tail` by 1. `tail` is now 1.
T_1 executes line 15, gets `i = 1` and increments `tail` by 1. `tail` is now 2.
T_1 executes line 16, writes `y` into `items[1]`.
T_0 executes line 16, writes `x` into `items[0]`.
T_2 calls `deq`, dequeues `items[0]` which has value `x`.
T_2 calls `deq`, dequeues `items[1]` which has value `y`.
Although T_1 executes line 16 before T_0, we fail to linearize the execution history `enq(y), enq(x), deq(x), deq(y)`.
This does not mean that `enq` is not linearizable, because linearizability depends on the specific execution history. We cannot define a linearization point that satisifies all possible method call sequences.

CMSC 23010 Homework 2 Programming Final

Ruolin Zheng

April 27, 2019

The svn submission in the `hw2` directory includes:
`Makefile` - compilation instructions
`include/` - a folder containing header files `driver.h` and `lamport_queue.h`
`src/driver.c` - the main program which parses command line arguments and invoke SERIAL, PARALLEL and SERIAL-QUEUE accordingly
`src/lamport_queue.c` - the Lamport Queue data structure
`tests/driver_output.c driver_output.h` - the same program as `src/driver.c` except that it writes the resultant checksum matrix to `[s|p|q]_res.txt`
`tests/test_driver_out.py` - automated Python script for running multiple trials of `test/driver_output` and comparing the results among SERIAL, PARALLEL and SERIAL-QUEUE
`tests/test_queue.c` - correctness tests for the `lamport_queue` using the `criterion` framework
`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 `hw2` directory. This will create the following binary executables: `./driver, tests/driver_output, tests/test_queue`.

1.1 The Program: driver

Usage:
`./driver -n NUM_SOURCE -t NUM_PACKET -d DEPTH -w MEAN -p PACKET_TYPE[c|u|e]`
`-r PROGRAM_TYPE[s|p|q] -s SEED`
Example:
`./driver -n 3 -t 50 -d 32 -w 25 -p c -r q` runs with SERIAL-QUEUE with constant packets, 3-1=2 sources, 50 packets per source, a queue of depth 32, and average workload 25.

1.2 Queue Correctness Unit Testing: test_queue

Please run this binary located in `tests/` as `./test_queue --verbose`.
This program tests for error conditions, serial correctness, and multiple scenarios which include one reader and one writer. These unit tests together guarantee the correctness of the concurrent lock-free queue implementation.

1.3 Program Correctness Testing: test_driver_output.py

Please make sure there is a binary named `driver_output` inside `tests` and then run `test_driver_output.py` in the same directory.
`driver_output` accepts the same parameter as does `driver`, but writes the checksum to a $T * (n-1)$ matrix for correctness testing. Running SERIAL, PARALLEL, SERIAL-QUEUE will generate `s_res.txt, p_res.txt, q_res.txt`, respectively.
The automated script, `test_driver_output.py`, will evoke `driver_output` with `-n 9 -t 4000 -d 32 -w 200` with exponentially distributed packets for each of SERIAL, PARALLEL, SERIAL-QUEUE. Subsequently, it checks for difference between each pair of the output files. This integration test guarantees the correctness of the whole program.

2 Performance Experiment Results

Note: For the following tests, data with Constant and Uniform Packets are collected over 5 trials; and data with Exponential Packets are collected over 11 trials.

2.1 Parallel Overhead

2.1.1 Original Hypothesis

(Copied from the Design Document with moderate clarification)

1. Due to the overhead, the ratio of SERIAL-QUEUE runtime to SERIAL runtime should be consistently above 1.0.

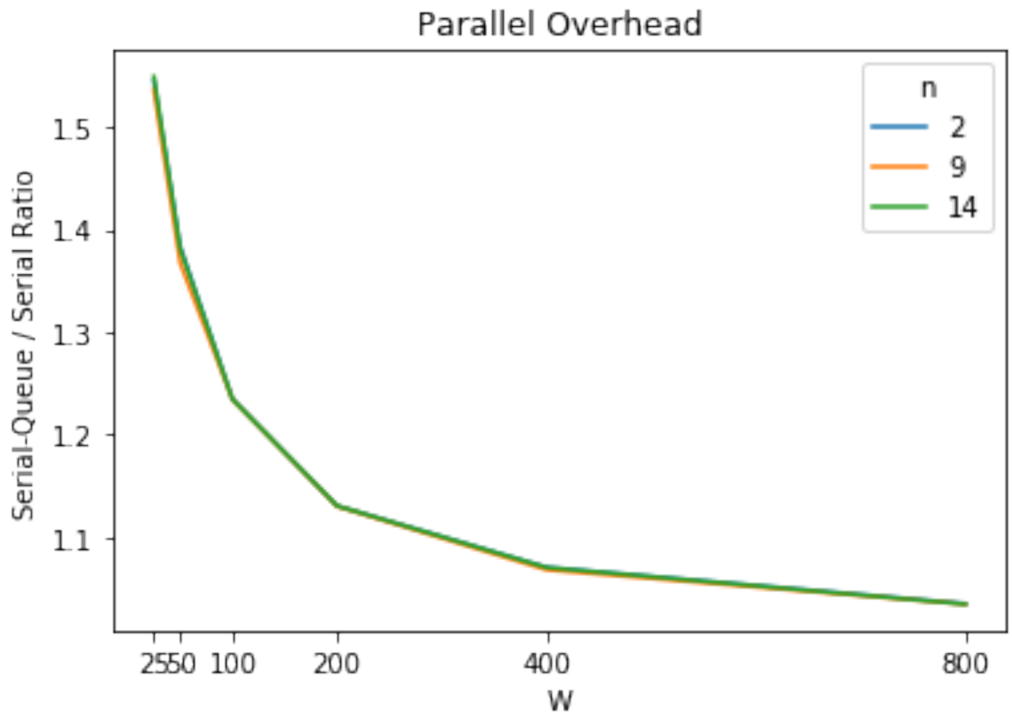
2. As W increases, the increase in overall runtime downplays the overhead, and the ratio should decrease and eventually tend to 1.0.

3. Moreover, for a fixed W, the total number of packets across all sources is approximately $T(n-1) \approx 2^{30}/W$; with larger n, SERIAL-QUEUE needs to perform more reads and write, thus incurs more overhead and increases runtime. Therefore, the ratio should also increase with increasing n, which means that the curve of $n = 14$ should lie above $n = 9$ and $n = 2$ at a fixed W.

Note: In order to obtain meaningful statistics instead of system noise, I set $T \approx 2^{30}/(nW)$ so that each run takes at least one second.

2.1.2 Data and Plot

n	W	T	Serial(millisec)	Serial-Queue(millisec)	speedup	Worker(#Packets/millisec)
2	25	21474836	2624.146	4053.854	1.544828	10594.775480
2	50	10737418	1874.280	2595.224	1.383789	8274.752576
2	100	5368709	1507.243	1864.783	1.235149	5757.998781
2	200	2684354	1330.752	1507.512	1.130834	3561.304401
2	400	1342177	1234.567	1323.613	1.070951	2028.050918
2	800	671088	1192.949	1235.213	1.035592	1086.595818
9	25	4772185	4651.683	7147.226	1.536482	6009.278699
9	50	2386092	3334.937	4573.808	1.367886	4695.176641
9	100	1193046	2685.471	3310.480	1.234476	3243.462652
9	200	596523	2354.252	2664.932	1.130338	2014.576402
9	400	298261	2196.707	2346.118	1.068564	1144.168605
9	800	149130	2116.272	2191.213	1.035074	612.527071
14	25	3067833	4856.618	7536.381	1.549092	5698.978457
14	50	1533916	3476.885	4789.550	1.381716	4483.685624
14	100	766958	2799.394	3462.973	1.234794	3100.635852
14	200	383479	2462.192	2787.673	1.130800	1925.874778
14	400	191739	2294.119	2457.878	1.070743	1092.143125
14	800	95869	2209.799	2289.653	1.035099	586.192441



2.1.3 Observation and Analysis

1. From the plot, the SERIAL-QUEUE to SERIAL runtime ratio is consistently above 1.0, which aligns with my hypothesis about the overhead.

2. Moreover, as W increases, the ratio declines and eventually tends to 1.0, which aligns with my hypothesis: increasing overall runtime downplays the effect of overhead.

3. For a fixed W, there doesn't seem to be any difference among different n values. This helps to correct my original hypothesis. For a fixed W, the total number of packets is also fixed at $T(n-1) \approx 2^{30}/W$, meaning that the same quantity of reading from and writing to the queue would take place in SERIAL-QUEUE regardless of n. Therefore, the overhead we incur by creating and communicating via the queue is independent of n. Hence, at any fixed W, the n curves should approximately overlap, as they do in the plot.

2.1.4 Worker Rate

With the total number of packets being $T(n-1) \approx 2^{30}/W$, the Worker Rate is calculated as $numPackets/runtime = 2^{30}/(W \times runtime)$. Please refer to the above table for the calculated results. My original hypothesis is that the worker rate decreases as W increases. The statistics support my hypothesis: for a fixed n, the worker rate decreases as W increases; for a fixed W, as n increases, the runtime increases, and the worker rate decreases.

2.2 Dispatcher Rate

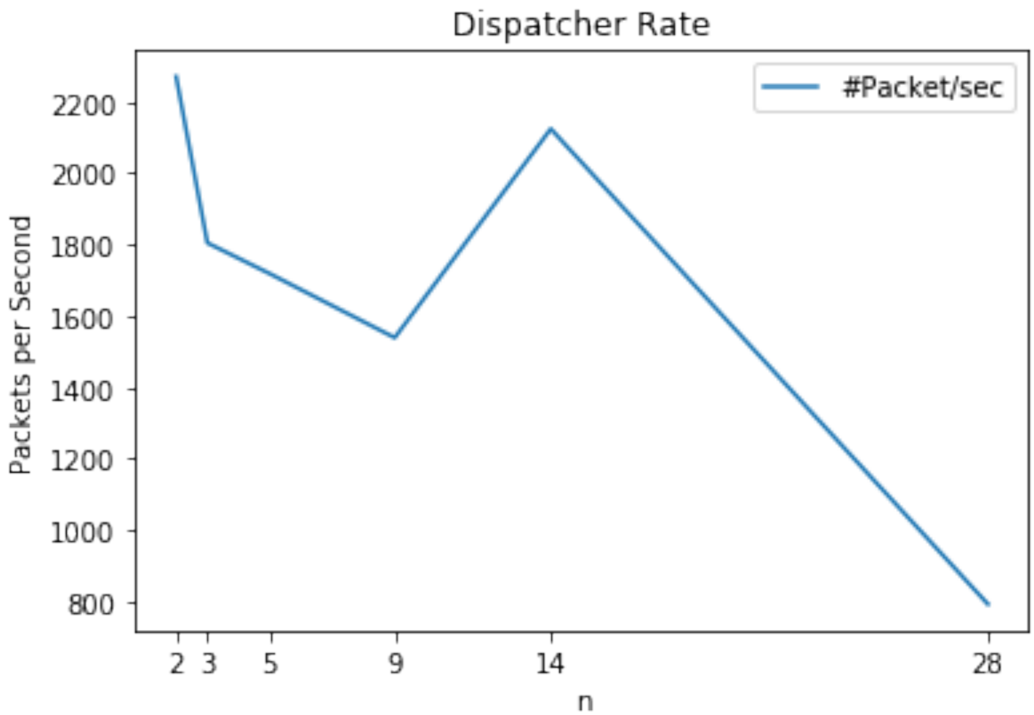
2.2.1 Original Hypothesis

(Copied from the Design Document) The total amount of work is approximately $T(n-1) = 2^{20}$, hence the **Dispatcher Rate** is $2^{20}/runtime$. Since the overhead of creating and joining $n-1$ threads, and of creating and communicating over $n-1$ queues increases in proportion to n, the runtime should increase as n increase; therefore, as n increases, the dispatcher rate becomes lower.

Note: In order to obtain meaningful statistics instead of system noise, I set $T \approx 2^{22}/(n-1)$ so that each run lasts over one second.

2.2.2 Data and Plot

n	W	T	Parallel(millisec)	Dispatcher Rate(#Packet/millisec)
2	1	4194304	1846.831	2271.082
3	1	2097152	2323.815	1804.922
5	1	1048576	2440.656	1718.515
9	1	524288	2725.052	1539.165
14	1	322638	1974.250	2124.500
28	1	155344	5294.299	792.227



2.2.3 Observation and Analysis

With $T = 2^2 2$, I expect the dispatcher rate to be $2^{22}/runtime$ and decrease with increasing n. The plot aligns with my hypothesis for the most part, except the outlier at $n = 14$. I've tried running more trials but still obtain non-regular pattern, which leads me to suspect that $n = 14$ happens to make very good use of parallelism and drastically decreases the runtime. From the table, the runtime at $n = 14$ is almost as low as $n = 2$.

Note: $T = 2^{17} = 131072$ for all three tests below: Constant, Uniform, and Exponential; thus T is not included in the tables.

2.3 Speedup with Constant Load

2.3.1 Original Hypothesis

(Copied from the Design Document with moderate clarification)

1. Ideally, the PARALLEL vs. SERIAL speedup should be $1/(n-1)$ as we have $n-1$ threads instead of one to handle the $n-1$ sources.

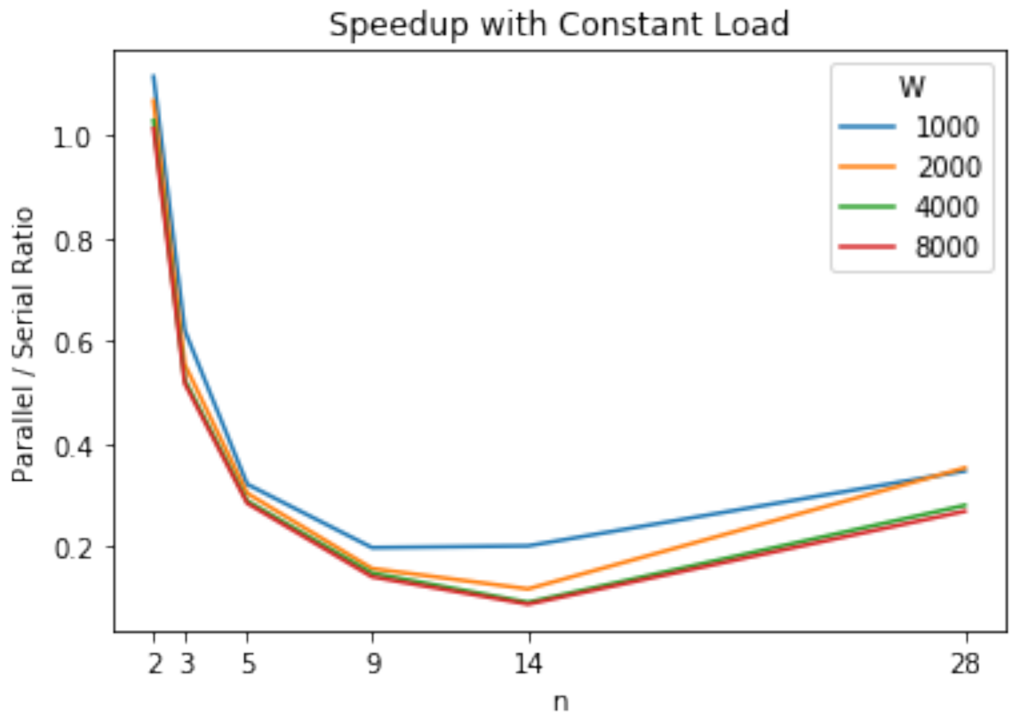
2. In reality, the overhead of creating and communication via queues, of spawning and joining threads, is proportional to n. The actual runtime of PARALLEL should be greater than ideal, and thus the speedup should be more than $1/(n-1)$.

3. As n increases, PARALLEL becomes significantly more efficient compared to SERIAL because the overhead gets downplayed. The curves should be downward-sloping.

4. Moreover, for any fixed n in PARALLEL, the amount of work per thread is exactly WT; assume PARALLEL runtime is a multiple of WT, then SERIAL runtime should be a multiple of $WT(n-1)$. This gives exactly our ideal speedup of $1/(n-1)$. Hence, there shouldn't be much variability among different values of W, and their curves should approximately overlap.

2.3.2 Data and Plot

n	W	Serial(millisec)	Parallel(millisec)	speedup
2	1000	292.007	325.439	1.114490
2	2000	572.277	608.243	1.067523
2	4000	1128.500	1162.073	1.029193
2	8000	2242.901	2273.204	1.013581
3	1000	575.844	359.941	0.620863
3	2000	1133.538	628.894	0.554806
3	4000	2252.506	1177.656	0.524782
3	8000	4477.932	2313.432	0.518063
5	1000	1151.174	372.224	0.322072
5	2000	2268.048	689.392	0.304599
5	4000	4493.259	1301.431	0.289586
5	8000	8949.366	2547.689	0.285111
9	1000	2290.665	455.253	0.198761
9	2000	4522.554	714.207	0.158079
9	4000	8984.202	1339.564	0.149082
9	8000	17903.551	2543.862	0.142085
14	1000	3720.190	751.702	0.202149
14	2000	7347.641	867.759	0.118471
14	4000	14597.049	1348.615	0.092410
14	8000	29076.768	2585.090	0.088877
28	1000	7728.419	2683.516	0.347787
28	2000	15261.901	5397.506	0.353796
28	4000	30311.205	8461.193	0.281005
28	8000	60419.168	16292.993	0.269551



2.3.3 Observation and Analysis

1. From the plot and the table, we do observe a speedup slightly higher than $1/(n-1)$: for example, at $n = 3, W = 1000$, we have a speedup of 0.58, slightly greater than the ideal $1/(3-1) = 0.5$. This aligns with my hypothesis about the difference between the ideal and the reality, as a result of the overhead.

2. As n increases from 2 to 14, we do observe the W curves to be downward-sloping. This aligns with my hypothesis that the increase in overall runtime amortize the overhead.

3. One noteworthy point is that, between $n = 14$ and 28, the speedup increases instead of decreases. This can be explained by the hardware constraint: the SLURM machines we are using support about 14 threads, so any thread number beyond that would run serially, and incur overhead of creating more queues and threads. Therefore, $n = 28$ would not experience the full speedup benefits of parallelism.

4. For a fixed n, we expect the different W curves to approximately overlap, which is illustrated in the plot. This aligns with my hypothesis that constant packets have low variability, thus PARALLEL's speedup should quite close to the ideal value $1/(n-1)$. A closer examinination of the table shows that, for a fixed n, as W increases, the speedup decreases by insignificant amounts. This can be explained by the fact that smaller W results in smaller runtime, and thus the effect of the overhead hasn't been

completely amortized. There is also more possibility for system noise. In all, we conclude from the plot that the W curves approximately overlap, which aligns with my original hypothesis.

5. To conclude, for **Constant Packets**, for $n = 2$ to 14, the measured speedup is close to the expected speedup. For $n = 28$, the measured speedup is significantly lower to the expected due to hardware constraint.

2.4 Speedup with Uniform Load

2.4.1 Original Hypothesis

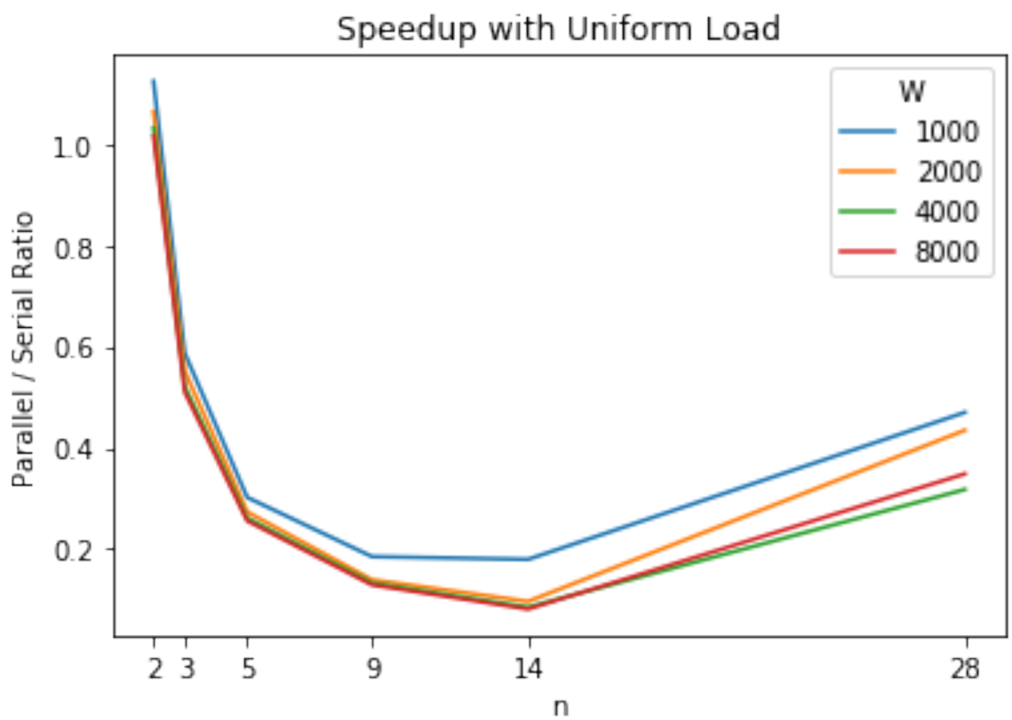
(Copied from the Design Document with moderate clarification)

1. The uniform distribution adds some variability to the system; for example, it is possible that one thread repeatedly gets heavy packets and increases the runtime of the entire system. With larger W, the range $[0, 2W]$ is wider, and the variability increases. (It may be worth noting that the probability shouldn't be too large when T is as great as 2^{17}.)

2. Overall, As in **Speedup with Constant Load**, the speedup ratio still decreases with n, but the W curves may be different from each other; the curves of larger W might lie above those of smaller W, because it is more probable for threads to get uneven loads and increase the runtime of the system. Ideally the speedup should still be $1/(n-1)$, but the measured speedup should be lower due to the unbalanced load problem.

2.4.2 Data and Plot

n	W	Serial(millisec)	Parallel(millisec)	speedup
2	1000	318.642	358.941	1.124749
2	2000	625.686	665.982	1.064645
2	4000	1247.180	1287.615	1.032436
2	8000	2482.137	2522.479	1.016388
3	1000	636.453	373.271	0.586874
3	2000	1254.006	691.655	0.551544
3	4000	2493.042	1298.843	0.520992
3	8000	4961.769	2531.078	0.510116
5	1000	1273.798	384.494	0.301979
5	2000	2510.169	686.674	0.273250
5	4000	4981.447	1300.767	0.261113
5	8000	9922.451	2535.151	0.255499
9	1000	2546.762	470.724	0.184804
9	2000	5015.293	697.433	0.139064
9	4000	9955.759	1322.366	0.132824
9	8000	19840.668	2557.476	0.128890
14	1000	4139.328	742.448	0.179364
14	2000	8149.179	783.897	0.096217
14	4000	16184.746	1374.464	0.084917
14	8000	32240.371	2614.865	0.081114
28	1000	8596.008	4044.649	0.469928
28	2000	16928.953	7359.082	0.434704
28	4000	33601.812	10671.155	0.317577
28	8000	66932.602	23316.752	0.348393



2.4.3 Observation and Analysis

1. Similar to **Constant Packets**, as n increases, the W curves are downward-sloping and eventually tend to $1/(n-1)$. For example, at $n = 14, W = 8000$, the ideal speedup is $1/13 = 0.0769$ and the actual speedup is 0.0811. This aligns with my hypothesis that increasing overall runtime amortize the effect of overhead and variability in load.

2. Similar to **Constant Packets**, we again observe $n = 28$ to be an outlier due to hardware constraints.

3. I hypothesized that the curves of larger W might lie above those of smaller W. However, the plot shows that for n from 2 to 14, the curves lie very close to each other, as in **Constant Packets**. This is plausible, as with T as large as 2^{17}, the expected values from the uniform distribution should be approximately the same as from the constant distribution.

4. At $n = 28$, the curves do lie farther off from each other than in **Constant Packets**. This is plausible as a consequence of the uneven load problem, further exposed by the hardware constraint.

5. To conclude, for **Uniform Packets**, for $n = 2$ to 14, the measured speedup is close to the expected speedup. For $n = 28$, the measured speedup is significantly lower to the expected due to hardware constraint and variability from a uniform distribution.

2.5 Speedup with Exponentially Distributed Load

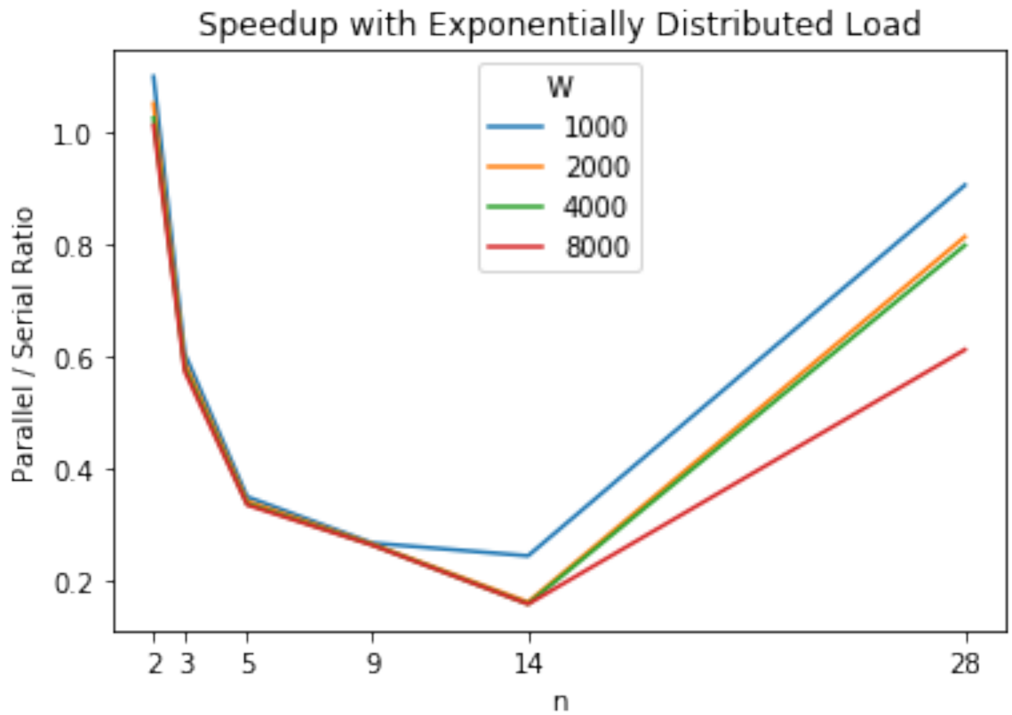
2.5.1 Original Hypothesis

(Copied from the Design Document with moderate clarification)

1. The exponential distribution introduces even more variability; and it is very likely that some thread constantly gets heavy packets and drastically increases the runtime of the entire system. With larger W, $\lambda = 1/W$ becomes smaller, and the individual W can vary more in value. Therefore, the curves of larger W should lie above those of smaller W because of this added imbalance of load distribution. The measured speedup should be a lot slower than $n-1$-fold.

2.5.2 Data and Plot

n	W	Serial(millisec)	Parallel(millisec)	speedup
2	1000	325.859	358.466	1.099864
2	2000	634.021	666.656	1.051629
2	4000	1250.453	1283.114	1.026015
2	8000	2483.611	2516.172	1.013040
3	1000	651.310	394.096	0.605050
3	2000	1268.089	747.321	0.589130
3	4000	2501.889	1451.118	0.579998
3	8000	4968.313	2848.891	0.573424
5	1000	1304.294	456.056	0.349665
5	2000	2540.272	868.236	0.341772
5	4000	5011.551	1690.381	0.337314
5	8000	9953.814	3335.548	0.335118
9	1000	2612.716	698.347	0.267190
9	2000	5084.639	1350.378	0.265560
9	4000	10032.797	2651.495	0.264280
9	8000	19926.408	5250.336	0.263486
14	1000	4243.362	1036.227	0.244284
14	2000	8263.844	1337.424	0.161824
14	4000	16298.538	2584.926	0.158603
14	8000	32372.576	5115.862	0.158033
28	1000	8809.014	7981.892	0.906140
28	2000	17154.600	13964.580	0.814002
28	4000	33834.855	27023.031	0.798657
28	8000	67207.453	41150.125	0.612323



2.5.3 Observation and Analysis

1. As in **Constant** and **Uniform**, my basic hypotheses about the general downward-sloping trend and my observation of $n = 28$ as an outlier still hold.

2. We do observe a speedup ratio significantly higher and worse than the ideal. To do a cross-comparison, at $n = 9, W = 8000$, the ideal speedup is $1/8 = 0.125$, **Constant** gives 0.142; **Uniform** gives 0.129; and **Exponential** gives 0.158.

3. Quite surprisingly, for $n = 2$ to 9, all W curves still lie close to each other despite the added variability. However, at $n = 14$, $W = 1000$ lies above all other curves, meaning that its performance is the worst. This is plausible since that we may happen to get very unbalanced loads, and cannot amortize that effect with such a small W.

4. At the outlier $n = 28$, we observe that the curves of smaller W are consistently above those of larger W. Similar to my last point, this is plausible as we might not have enough work to amortize the imbalance. Therefore, smaller W means more variability and thus the curves lie far off from each other.

5. This may lead to the conclusion that, when the number of threads is supported by the hardware, the unbalanced load problem is real, but not as influential as we may hypothesize.

6. To conclude, for **Exponentially Distributed Packets**, for $n = 2$ to 14, the measured speedup is lower than the expected speedup, compared to **Constant** and **Uniform**. For $n = 28$, the measured speedup is significantly lower to the expected due to hardware constraint and added variability from an exponential distribution.