Introduction

This is an implementation of producer/consumer system, borrowing existing concepts of multiple buffering to significantly enhance throughput by reducing the overhead of thread synchronization. A buffer of N locations can be viewed as a buffer of N rows and 1 column. The access to each location should be protected by a lock (mutex or atomic) in order to prevent a data race. In the modified data structure, the buffer is viewed as N = R x C, where R is number of rows and C is number of columns. Each thread will work on a single row of C locations at a stretch rather than one by one. This also helps increasing cache coherency as a thread can work on contiguous locations. The results are illustrated with timing statistics. A brief informal analysis is presented to explain performance improvement.

Multi-buffer

Multi-buffer here is just another name for two-dimensional array. The batching of multiple contiguous locations for synchronization by producer and consumer is effectively used in Disruptor design (http://mechanitis.blogspot.co.uk/2011/07/dissecting-disruptor-writing-to-ring.html). However in the multi-buffer implementation, where the buffer size (that is row size) is fixed, the concept is borrowed from traditional “double buffering” in computer animation (https://en.wikipedia.org/wiki/Multiple_buffering) . When an image is being displayed, the next image is created in a separate ‘back buffer’. Once the back image is rendered, it will be copied to the ‘front buffer’ that is displayed. Making use of display time for generating the next image engenders artifact-free smooth animation. In our case, when one thread is reading from a buffer, another thread can be writing to another.

Timing statistics

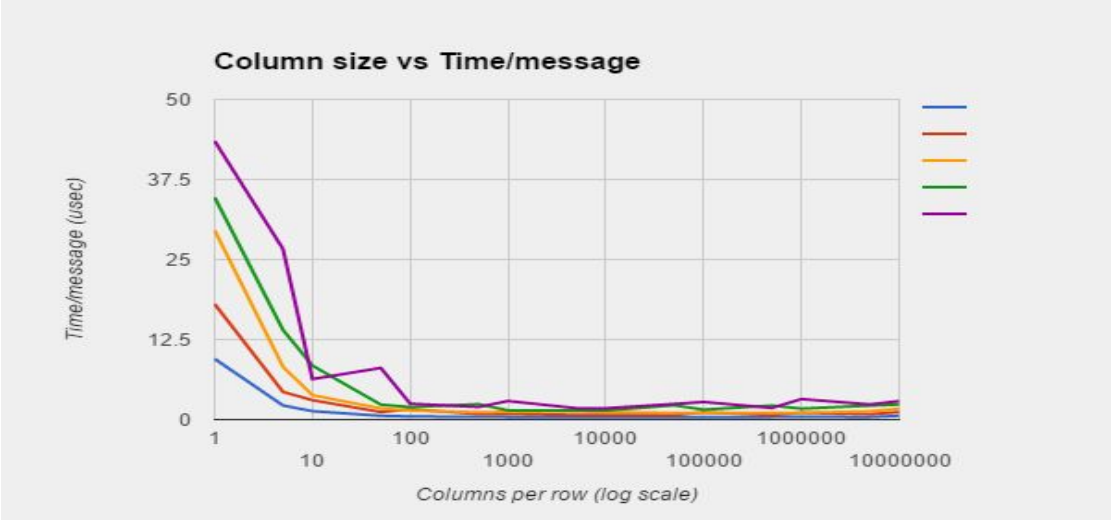
The code is developed in C++ using Visual Studio Community 2015 running on Intel i7-5500 2.4GHz dual core processor hosting Windows 10 64-bit OS. The machine has 8GB RAM. The synchronization is done using **lockless** primitives, rather than more expensive mutexes.

The tests are done with an integer ring buffer of 10,000,000 locations. Producer(s) and consumer(s) are run for about 5000 milliseconds. The number of locations processed by each thread are counted and per message timing measured. Processing involves producer writing an integer value and consumer reading it from the location. Consistency checks are made to ensure that what is read is identical to what was written.

This table presents timing statistics (in microseconds/message) with number of columns gradually increased from 1 (10,000,00 rows) to 10,000,000 (single row). The number of producer and consumer threads is varied from 1 to 5 each

Columns/row	prod/cons 1	prod/cons 2	prod/cons 3	prod/cons 4	prod/cons 5
1	9.48326	18.0454	29.5644	34.7048	43.4996
5	2.20292	4.32257	8.23745	14.0021	26.6308
10	1.31265	3.02914	3.81599	8.42288	6.32188
50	0.58752	1.17786	1.7446	2.32926	8.0644
100	0.491919	1.61091	1.40518	1.95166	2.47732
500	0.420428	0.864207	1.13753	2.40795	1.97138
1000	0.409507	0.891153	1.21051	1.43177	2.89715
5000	0.391492	0.709989	1.25976	1.40151	1.78031
10000	0.401206	0.6957	1.17283	1.41461	1.76539
50000	0.409838	0.696577	1.03741	2.22827	2.43613
100000	0.399542	1.14114	1.05516	1.54845	2.75032
500000	0.410689	0.705494	1.04381	2.18656	1.80933
1000000	0.418593	1.08169	1.06539	1.71233	3.2017
5000000	0.406894	0.842637	1.30016	2.17918	2.36909
10000000	0.58836	1.17666	1.70513	2.35356	2.8749

Above table is plotted below, each colour representing the plot for specific number of threads. Identical number of producer and consumer threads are used. For example 3 in the above table means 3 producer and 3 consumer threads. Higher the number of threads, greater the time taken per message. It can be seen that increasing number of locations per row has significant impact on throughput by reducing time / message.



Analysis suggests that time/message is inversely proportional to columns/row

Analysis

There is significant improvement when multiple buffers are used, even with modest number of locations in each buffer.

Let '`s`' be amount of time spent in synchronization, acquiring a row.
Let '`w`' be amount of time writing a single location.
Let N be the total number of locations = Rows X Columns (N = 10,000,000 in our tests),

The total producer time T spent in producing whole buffer (consumer case is similar) = locking time + writing time

```
N =  number of rows(R)*locations per row(C)
N = R*C ---------------------- (1)
```

Locking is required for each row. Thus

```
Time to synchronize R rows = R*s ----------------- (2)
Time to write all locations = N*w ----------------- (3)
```

Total time to produce whole buffer

```
T = (2) + (3) = R*s + N*w ----------------------- (4)
```

Time/location (Y-axis of the plotted graph): t

```
t = T/N = R*s/N + N*w/N
```

```
R*s/N = s/C since N = R*C
```

`t = s/C  + w ---------------------- (5)`

`t` is inversely proportional to locations per row which is similar to a graph of the form :

```
y = 1/x
```

Limitation: If time taken to write each location 'w' dominates the total time, that is `w >> s/C`, then `t` degenerates to w.

Greater the number of columns per row, faster the production time, greater the throughput.

Special case: single row

When reducing the number of rows saves time by reducing locking time, what happens in the extreme case of using a single row, that is, C = N? The last item in the statistic tables represents this case. Should it not produce the best results? One notices that the last item in the tables that represents this case has inferior results. The explanation is that no other producer or consumer is able to run concurrently when a producer or consumer is working on the buffer. One needs at least two rows to benefit from concurrency.

Latency

Throughput can be increased by reducing the number of rows, thereby reducing locking, Reducing the number of rows means, increasing number of locations per row.This means latency suffers.

In each row, locations are produced sequentially, with 'w' as write time per location:
first location is produced at w, second at 2*w, third at 3*w, and so on and Cth location at C*w

Thus, the worst case latency in each row = C*w ---------- (6)
(the time taken to produce C-th location in a row)

If C =1, that is, each row with one location, the latency is best.

Smaller the number of columns per row, the greater the latency

Optimum number of columns per row for throughput and latency

Higher throughput favours maximum number of columns per row, but higher latency favours the opposite, seemingly irreconcilable.

Following is an attempt to get at some optimal C.

```
From (4) above T: total production time :
T = R*s + N*w
 = N*s/C + N*w ------------ (a)

From (6) worst case latency

L_worst = C*w   --------------- (b)

Optimize (a) + (b)
That is, optimize both total production time that determines throughput
as well as latency

f(C) = T + L_worst
     = N*s/C + N*w + C*w ---------------------- ( c )
```

Find minima of f(C), by taking 1st derivative w.r.t C and solve for 0

```
minima f(C) = -N*s/(C*C) + w = 0  (since N*w is constant)
              w*C*C = N*s
              C = sqrt(N*s/w) ------------------ (d)
              C ~ sqrt(N)
```

Optimal number of locations per row is proportional to sqrt of N.

Note that if 's' dominates, i.e. s >> w, it makes sense to have larger 'C' to reduce synchronization time, where as if w >> s, it means synchronization has relatively little cost and one could revert to C = 1, a traditional one dimensional buffer.