Assignment 4 Analysis

Justin Cohler
5/7/2019

Background

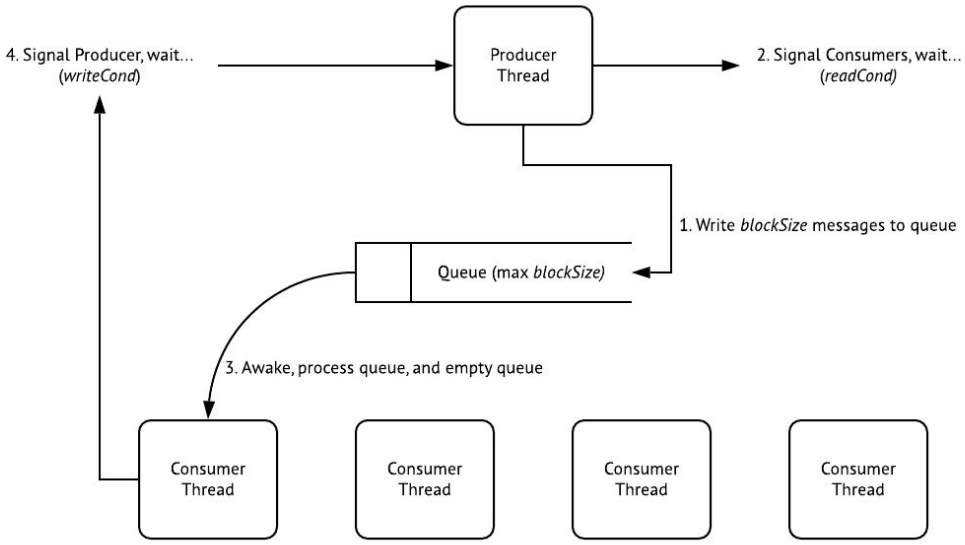
In approaching block reading twitter messages, this project design employed the use of a queue (a Go slice in practice). I first created a global *TaskContext* structure, with the following tooling facilities encapsulated:

```
type TaskContext struct {
    sync.Mutex
    readCond      *sync.Cond
    writeCond     *sync.Cond
    blockSize     int
    queue         []string
    terminateFlag bool
    wg            sync.WaitGroup
}
```

Among some helpful flags and metadata, the above context contains the queue where commands are placed, as well as two *Conditions*. The first condition is used by the producer thread to signal/broadcast *n* asynchronous consumers to act. The second condition is for a given selected consumer to alert the producer thread to continue block writing. The *terminateFlag* is used by the producer thread to alert all consumers to wrap-up via a *broadcast* sent after all records have been written from stdin-to-queue.

On program start, the main thread generates *n* asynchronous threads (as defined by command-line argument), all consumers of the queue, each of which initially signal a *write Condition* before entering their own *read Condition* wait. Additionally, the program generates one asynchronous producer thread, which initially awaits an awakening signal.

When the producer awakes, acquiring the shared *mutex* on the *TaskContext*, it reads a *block* of lines from the file and appends them to the queue. It then signals one consumer to wake up and process the batch of lines placed on the queue. The ordering of the cycle can be represented in large part by the below diagram (although there are a few additional checks happening at each step).



Benchmark Specifications

The aforementioned benchmarking was performed on a commodity hardware laptop, with the following specifications:

- **CPU:** Intel(R) Core i5-6200U CPU @2.30GHz
    - 2 cores
    - 4 logical hardware threads (2 per core)
- **Memory:** 8GB DDR3
- **OS:** Microsoft Windows 10 Home Edition

The benchmarking was performed via bash script run several times (to take the average in Excel) running through the cartesian product of all options listed:

- Thread Count: {0 (Sequential), 1, 2, 4, 6, 8}
- # of Operations: {50k, 100k, 500k, 1M}
- Operation Type Breakdown: {99% Remove/Write, 99% Contains, Even Split}

Results

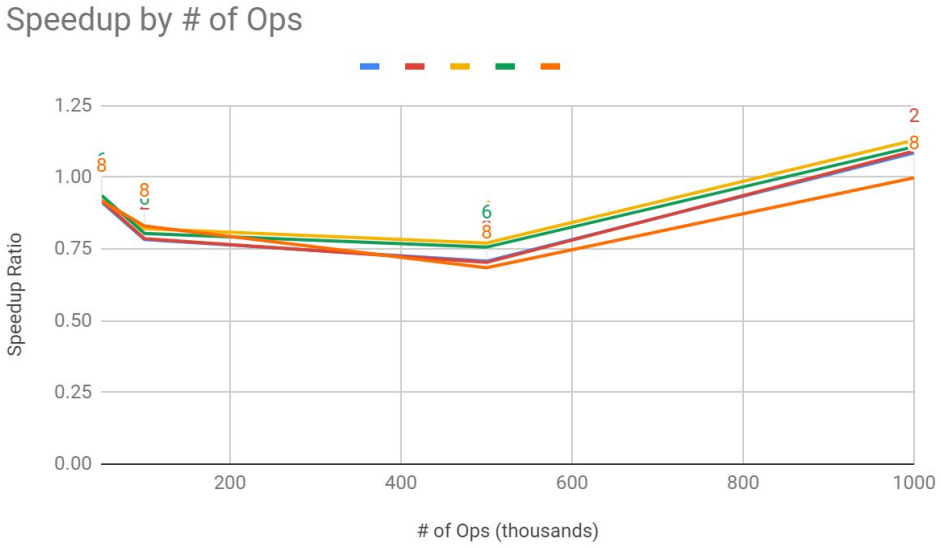
- How do increased numbers of threads affect the speedup ratio?

    In short, there is no speedup as thread counts increase in the current implementation of this program. The reason for this is straightforward: this program locks the queue to write a *blockSize* of records to the queue, then allows only one thread to process those messages, to ensure correctness is consistent, avoiding cases where a *WRITE* with an earlier timestamp is processed after a *REMOVE*

with a later timestamp, which would fail. As a result, only one thread is performing an operation at a given time. Therefore our parallelizability percentage is 0%. Plugging into Amdahl's Law:

$$speedup\ ratio\ =\ 1/(1+0/N)\ =\ 1.0$$

This is exactly what we see in results, where across thread-counts and operation-counts, the speedup continuously hovers around 1.0. In fact, as threads increase, the speedup may be statistically significantly lower than lower thread counts. Note that in the plot below, the eight thread solution (in orange) performs significantly worse consistently than the two and four thread solutions (in red and yellow).

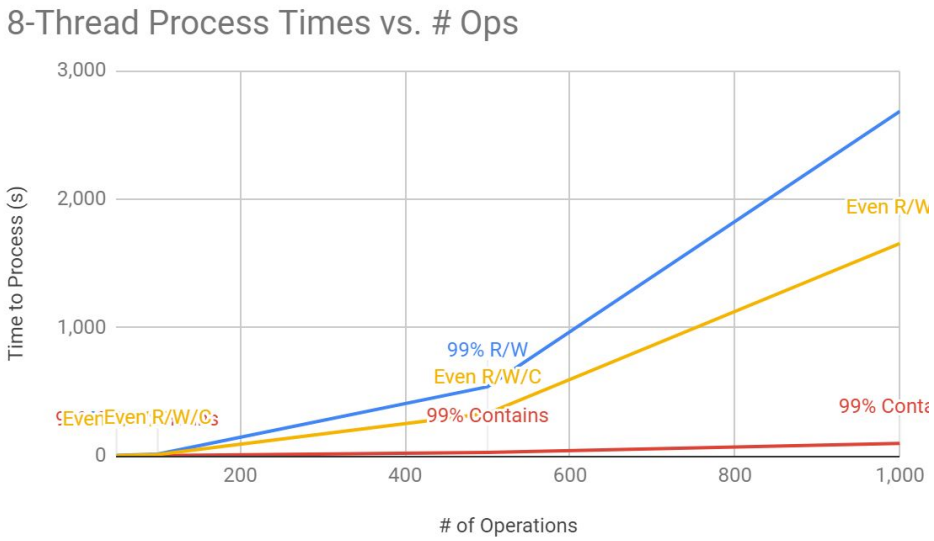


The reason for this decrease in performance is due to the overhead of context switching. Having to save state and context switch across large numbers of threads adds useless handoffs, which take time but do not increase parallelizability across hardware threads. As a result, this reduces the performance of the serial execution.

- How does Operation Type affect processing speed?

  As shown above, there is essentially zero (or negative) speedup in processing *n* operations as number of threads increases, as a result of sequential processing with the added overhead of context switching. Therefore, I arbitrarily chose to fix on the 8-thread data to compare processing times across different operation distributions. The three distributions of operation types were as follows:

- **99% R/W:** 49% Writes, 50% Removes, 1% Contains, 0% Strings
- **99% Contains:** 1% Writes, 0% Removes, 99% Contains, 0% Strings
- **Even R/W/C**: 33% Writes, 33% Removes, 34% Contains



Plotting the results of runtimes vs number of ops (from 50k to 1M), it is apparent that *Contains* operations are significantly faster than *Write/Remove* operations. This stands to reason, as *Contains* operations require only a read-lock, which can be acquired multiple times by readers. A write-lock, on the other hand must spin until all readers and any other writers have completed their operations in order to acquire the lock. This may result in significantly more spinning for writes to occur than for operations requiring only a read-lock, which do not need additional spins when other readers are active.