Thread-safe histograms

Jim Pivarski

Princeton University – DIANA

June 1, 2017

Overheard at a CMS Core meeting:

DAQ makes thousands of histograms.

To safely fill them in parallel threads, they're scattered into thread-local copies, gathered by hsitogram-addition.

But this uses so much working memory, some tasks can't be performed.

What is the cost of a thread-safe histogram?

How could it be implemented?

Idea #1: make each histogram an actor with a thread-safe queue as a mailbox. “Fill” sends a message.

Idea #2: block write access at the granularity of *bins.*

using locks?

compare-and-swap?

`std::atomic`?

How could it be implemented?

Idea #1: make each histogram an actor with a thread-safe queue as a mailbox. “Fill” sends a message.

No. These things work when load is variable, allowing the receiver to catch up on old messages during low load without slowing down the sender. Doing this in a steady-load environment doesn't make sense.

Idea #2: block write access at the granularity of *bins.*

using locks?

compare-and-swap?

`std::atomic`?

How could it be implemented?

Idea #1: make each histogram an actor with a thread-safe queue as a mailbox. “Fill” sends a message.

No. These things work when load is variable, allowing the receiver to catch up on old messages during low load without slowing down the sender. Doing this in a steady-load environment doesn’t make sense.

Idea #2: block write access at the granularity of *bins.*

using locks?

No. The memory overhead of a lock per bin would defeat the purpose (saving memory), and a lock per histogram is too coarse.

compare-and-swap?

`std::atomic?`

How could it be implemented?

Idea #1: make each histogram an actor with a thread-safe queue as a mailbox. "Fill" sends a message.

No. These things work when load is variable, allowing the receiver to catch up on old messages during low load without slowing down the sender. Doing this in a steady-load environment doesn't make sense.

Idea #2: block write access at the granularity of *bins*.

using locks?

No. The memory overhead of a lock per bin would defeat the purpose (saving memory), and a lock per histogram is too coarse.

compare-and-swap?

Sounds good: per-bin contention is low and the work to be repeated in the case of a collision is minimal. No extra memory required.

`std::atomic`?

How could it be implemented?

Idea #1: make each histogram an actor with a thread-safe queue as a mailbox. “Fill” sends a message.

No. These things work when load is variable, allowing the receiver to catch up on old messages during low load without slowing down the sender. Doing this in a steady-load environment doesn’t make sense.

Idea #2: block write access at the granularity of *bins.*

using locks?

No. The memory overhead of a lock per bin would defeat the purpose (saving memory), and a lock per histogram is too coarse.

compare-and-swap?

Sounds good: per-bin contention is low and the work to be repeated in the case of a collision is minimal. No extra memory required.

`std::atomic`?

I originally thought this did locking, but I was wrong.

Naive implementation: unsafe baseline

Python script sets up many threads, feeds them all the *same* block of memory (`fillme`), and starts them all at the same time.

```
double fill(long *fillme, long size, long trials, long cardinality, long *collisions) {
  struct timeval startTime, endTime;
  int shift = (int)floor(log2((double)size / cardinality));   // control collision rate

  std::mt19937 rng;                                      // thread-local random number generator
  rng.seed(std::random_device()());
  std::uniform_int_distribution<long> distribution(0, cardinality - 1);

  gettimeofday(&startTime, 0);                           // start the stopwatch
  for (long i = 0;  i < trials;  i++) {
    long value = distribution(rng) << shift;             // drop LSBs to get more collisions

    fillme[value]++;                                     // naive increment bin

  }
  gettimeofday(&endTime, 0);                             // stop the stopwatch and return time
  return (1000L * 1000L * (endTime.tv_sec - startTime.tv_sec) +
              (endTime.tv_usec - startTime.tv_usec)) / 1000.0 / 1000.0;
}
```

Compare-and-swap implementation

Python script sets up many threads, feeds them all the *same* block of memory (`fillme`), and starts them all at the same time.

```
double fill(long *fillme, long size, long trials, long cardinality, long *collisions) {
  struct timeval startTime, endTime;
  int shift = (int)floor(log2((double)size / cardinality));   // control collision rate

  std::mt19937 rng;                                      // thread-local random number generator
  rng.seed(std::random_device()());
  std::uniform_int_distribution<long> distribution(0, cardinality - 1);

  gettimeofday(&startTime, 0);                           // start the stopwatch
  for (long i = 0;  i < trials;  i++) {
    long value = distribution(rng) << shift;             // drop LSBs to get more collisions

    long *ptr = &fillme[value];
    long oldval = *ptr;                                  // try...
    long newval = oldval + 1;
    while (CAS(ptr, oldval, newval) != oldval) {
      oldval = *ptr;                                     // ...try again
      newval = oldval + 1;
      (*collisions)++;                                   // measure actual collision rate
    }

  }
  gettimeofday(&endTime, 0);                             // stop the stopwatch and return time
  return (1000L * 1000L * (endTime.tv_sec - startTime.tv_sec) +
            (endTime.tv_usec - startTime.tv_usec)) / 1000.0 / 1000.0;
}
```

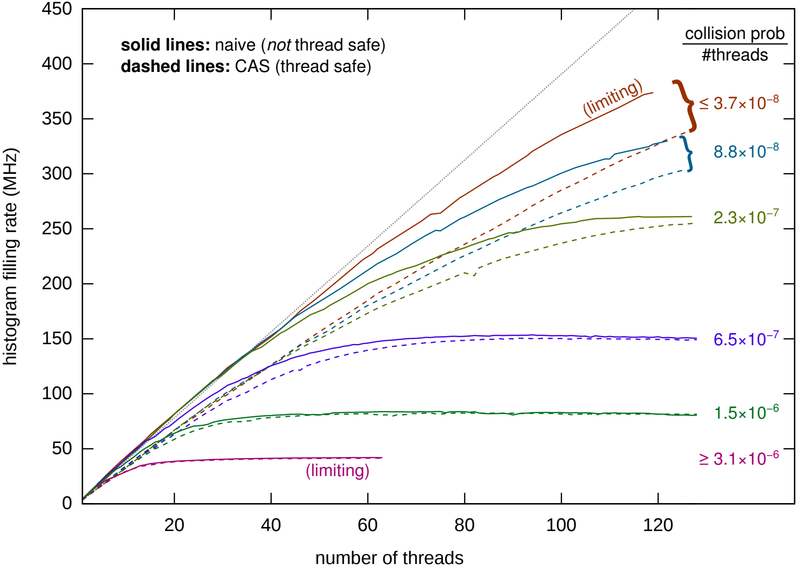
Python script sets up many threads, feeds them all the *same* block of memory (`fillme`), and starts them all at the same time.

```
double fill(long *fillme, long size, long trials, long cardinality, long *collisions) {
  struct timeval startTime, endTime;
  int shift = (int)floor(log2((double)size / cardinality));   // control collision rate

  // reinterpret the block of memory as an array of atomics
  std::atomic<long>* fillme2 = reinterpret_cast<std::atomic<long>*>(fillme);

  std::mt19937 rng;                                       // thread-local random number generator
  rng.seed(std::random_device()());
  std::uniform_int_distribution<long> distribution(0, cardinality - 1);

  gettimeofday(&startTime, 0);                            // start the stopwatch
  for (long i = 0;  i < trials;  i++) {
    long value = distribution(rng) << shift;              // drop LSBs to get more collisions

    // fancy fill method
    fillme2[value].fetch_add(1, std::memory_order_relaxed);

  }
  gettimeofday(&endTime, 0);                              // stop the stopwatch and return time
  return (1000L * 1000L * (endTime.tv_sec - startTime.tv_sec) +
              (endTime.tv_usec - startTime.tv_usec)) / 1000.0 / 1000.0;
}
```

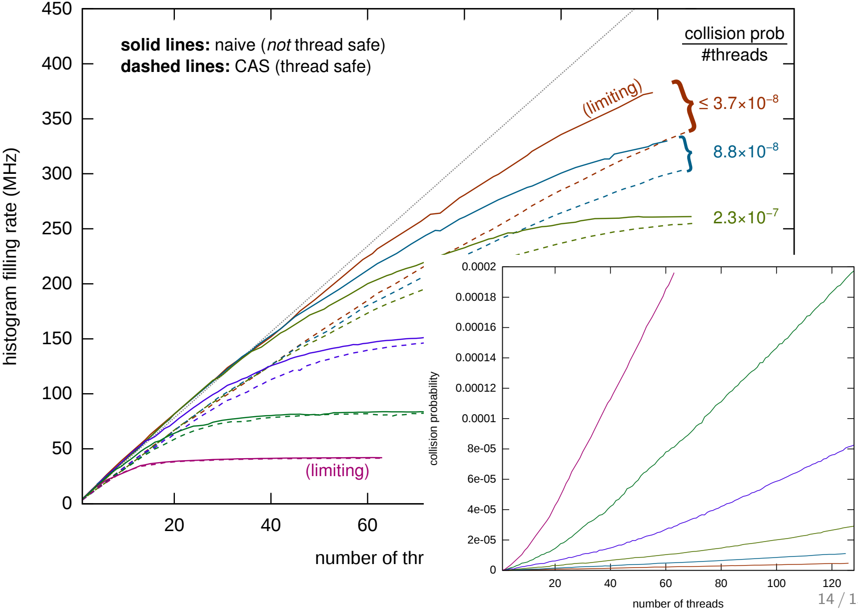
Experimental conditions

- Knight's Landing (for 128 threads), memory block in normal DRAM.
- Number of threads and naive vs. safe performed in a random order.
- Before execution, threads pinned to a random subset of CPUs.
- Collision rate controlled by bit shift, then measured in situ.

Compare-and-swap results



Compare-and-swap results



`std::atomic` results