Adding Approximate Counters

GUY L. STEELE JR. and JEAN-BAPTISTE TRISTAN, Oracle Labs

We describe a general framework for adding the values of two approximate counters to produce a new approximate counter value whose expected estimated value is equal to the sum of the expected estimated values of the given approximate counters. (To the best of our knowledge, this is the first published description of any algorithm for adding two approximate counters.) We then work out implementation details for five different kinds of approximate counter and provide optimized pseudocode. For three of them, we present proofs that the variance of a counter value produced by adding two counter values in this way is bounded, and in fact is no worse, or not much worse, than the variance of the value of a single counter to which the same total number of increment operations have been applied. Addition of approximate counters is useful in massively parallel divide-and-conquer algorithms that use a distributed representation for large arrays of counters. We describe two machine-learning algorithms for topic modeling that use millions of integer counters and confirm that replacing the integer counters with approximate counters is effective, speeding up a GPU-based implementation by over 65% and a CPU-based implementation by nearly 50%, as well as reducing memory requirements, without degrading their statistical effectiveness.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; E.2 [**Data Storage Representations**]; G.3 [**Probability and Statistics**]: Probabilistic Algorithms

General Terms: Algorithms, estimation, measurement, performance, theory

Additional Key Words and Phrases: Approximate counters, distributed computing, divide and conquer, multithreading, parallel computing, statistical counters

ACM Reference format:
Guy L. Steele Jr. and Jean-Baptiste Tristan. 2017. Adding Approximate Counters. *ACM Trans. Parallel Comput.* 4, 1, Article 5 (October 2017), 45 pages.
https://doi.org/10.1145/3132167

1 INTRODUCTION, BACKGROUND, AND RELATED WORK

We will say that a *counter* is an integer-valued variable whose value is initially 0 and to which two operations can be freely applied: *increment*, which replaces the current value k with $k + 1$, and *read*, which returns the current value of the variable. We assume that in a concurrent environment all the operations on a given counter are observed by all threads to have been performed as if in some specific sequential order, and so each operation may be regarded as atomic. It is then easy to see that the result of any *read* operation on a given counter will be the number of *increment* operations performed on that counter prior to the *read* operation.

Authors' addresses: G. L. Steele Jr. and J.-B. Tristan, Oracle Labs, 35 Network Drive UBUR02-313, Burlington, MA 01803; emails: {guy.steele, jean.baptiste.tristan}@oracle.com.

https://doi.org/10.1145/3132167

Morris (1978) introduced the notion of a *probabilistic counter*. His idea was to be able to use w bits to count more than 2^w things by actually increasing the counter value in response to only some *increment* operations rather than all of them; in one specific case, his algorithm increases a counter value of k with probability 2^{-k}, and a *read* operation that observes a counter value k returns $2^k - 1$ as a statistical estimate of the actual number of times the *increment* operation has been performed.

Morris, furthermore, provided a generalization of this algorithm as well as a statistical analysis. The probabilistic decision made by the *increment* operation can rely on the output of a random (or pseudorandom) number generator, and as Morris observes, "The random number generator can be of the simplest sort and no great demands are made on its properties." Flajolet (1985) provides a detailed statistical analysis of the Morris algorithms.

Cvetkovski (2007) describes Sampled-Log Approximate Counting, a variant that has better accuracy over most of the counting range and a lower expected number of actual advances of the counter. Csűrös (2010) provides a framework and analysis applicable to a more general class of counter representations and algorithms than those of Morris. Mitchell and Day (2011) introduce "flexible approximate counters," in which the probability of increasing the counter value may be controlled by a user-supplied table. Dice, Lev, and Moir (hereafter "DLM") (2013) examine the use of probabilistic counters in a concurrent environment and explore alternate representations for the counter state.

Tristan et al. (2015) report that replacing standard counters with approximate counters allows a machine-learning algorithm (topic modeling) using Latent Dirichlet Allocation with Gibbs sampling (LDA Gibbs) on a GPU to make more efficient use of the limited memory on a GPU and therefore process larger data sets. They also report that the statistical performance of the algorithm remains essentially unchanged, and that, surprisingly, the version using approximate counters runs *faster*, despite the fact that incrementing an approximate counter requires generation of a pseudorandom number as well as other calculations. (Their Figure 6 indicates an overall speed improvement of approximately 9%.) They suggest two possible reasons for the increase in speed: (1) probabilistic incrementation performs many fewer actual writes to memory, and (2) less memory bandwidth is needed when performing bulk reads of the counters.

We have now extended this prior work by implementing distributed versions of the same algorithm (using either normal integer counters or approximate counters), on eight GPU cards with an Ethernet interconnect. The arrays of counters are replicated so each GPU has a complete set of counters, but each GPU otherwise has just $\frac{1}{8}$ of the total dataset to be analyzed. On each iteration, each GPU clears its counters, processes its fraction of the dataset, and then uses a standard (hypercubelike) message-exchange pattern to sum elementwise the eight arrays of counter values, in such a way that every GPU receives all sums (as for the MPI operation `MPI_ALLREDUCE` (Snir et al. (1998), Section 4.11.4)); the version that uses approximate counters adds approximate counters by using the algorithms that we present in this article. The behavior is quite similar to that of the single-GPU version reported by Tristan et al.: replacing integer counters with approximate counters does not affect the statistical performance of the algorithm, but allows the same hardware to process larger datasets. In one test, either 32-bit integer counters were used or 8-bit approximate counters; using approximate counters improved the overall speed of each iteration of the distributed application by over 65%. We have also tested a distributed multiple-CPU implementation of another algorithm, organized as a stochastic cellular automaton, that addresses the same topic-modeling problem. We believe that approximate counters will be increasingly useful in future applications that perform statistical analysis of "Big Data" (whether using "machine learning" techniques or otherwise) and that large-scale distributed implementations of these applications will require the ability to add approximate counters.

Novel contributions of this article (an expanded version of Steele and Tristan 2016):

(1) a notational framework that encompasses many kinds of approximate counter representation and reconciles some conflicting notation in the literature;
(2) the presentation, using this framework, of a general algorithm for adding approximate counters so expected estimated value of the sum is equal to the sum of the expected estimated values of the given approximate counters (we believe that this is the first published algorithm for adding approximate counters);
(3) the derivation of optimized implementations of this general addition algorithm for five specific kinds of approximate counter;
(4) the presentation of "cookbook" versions of the algorithms (highlighted by framing boxes) that also check for overflow and therefore are suitable for transcription into actual programs;
(5) proofs for three kinds of counters that the variance remains bounded when counter values are added;
(6) speed measurements of multiple algorithms for adding approximate counters; and
(7) speed and quality measurements of distributed implementations of two machine-learning applications whose performance is greatly improved through the use of approximate counters and the ability to add approximate counters.

2 A GENERAL FRAMEWORK

Following Csűrös (2010), but generalizing his framework slightly to accommodate the alternate representations of DLM, we characterize a probabilistic counter that uses representation T, transition function $\tau : T \to T$, and transition probability function $Q : T \to [0, 1]$ as a variable that can contain values of type T whose value is initially S_0 (a specific value of type T). Two operations may be freely applied to such a counter: *increment*, which with probability $Q(s)$ replaces the current value s with $\tau(s)$ (and with probability $1 - Q(s)$ does not change the value of the variable), and *read*, which returns a value $f(s)$ that may be regarded (i.e., modeled) as a random variable whose expected value is the number of *increment* operations performed on that counter prior to the *read* operation. The function f is called the *unbiased estimator function*; Csűrös shows that f may be uniquely derived from Q.[1] We require the transition function τ never to cycle; that is, it satisfies the property that $\tau^i(S_0) \neq \tau^j(S_0)$ if $i \neq j$.[2] As a result, the possible states of a probabilistic counter produced by *increment* operations from the starting value S_0 are in one-to-one correspondence with the natural numbers, and the only reason to choose a representation T other than the natural numbers $\mathbb{N}$ is for "engineering purposes." For notational convenience, we will define $q_k = Q(\tau^k(S_0))$; this matches the meaning of "q_k" as used by Csűrös. We will also define $f_k = f(\tau^k(S_0))$, which corresponds to "$f(k)$" as used by Csűrös.

If we assume that the pseudo-function *random*() chooses a real number uniformly randomly (or uniformly pseudorandomly) from the half-open real interval $[0, 1)$, then the *increment* and *read* operations may be implemented as follows:

[1] Mitchell and Day (2011) take the opposite approach: their method represents the estimator function f (which they call "ϕ") as a monotonically increasing table and uses a solver to derive the transition probability function Q (which they call "p"), also represented as a table.

[2] In practice, it may be acceptable for an implementation of τ to signal an "overflow error" if it would otherwise be compelled to repeat a counter state. Some of the pseudocode we present explicitly checks for and signals overflow errors. One way to handle such a signal is to allow the counter to "saturate" once it reaches its highest possible value; our pseudocode shows the appropriate assignment statement (if needed) for this purpose.

```
1: procedure increment(var X: T)
2:     if random() < Q(X) then X ← τ(X)

1: procedure read(X: T)
2:     return f(X)
```

That is the essence of it; all the rest is engineering and the statistical analysis behind it. We may choose T and τ and Q so τ is easy to calculate, or Q is easy to calculate, or f is easy to calculate, or all three. It may be desirable to choose Q to guarantee certain statistical properties, for example, so the variance of values returned by f can be bounded (and this can be done in various ways), and perhaps also to choose Q so the dynamic range of the counter will be appropriate for a specific application.

In addition to the pseudofunction *random*() already described, which returns a real value (or floating-point approximation) in the range $[0, 1)$, we will also find it convenient to assume the availability of two additional pseudofunctions. First, *randomBits*(j) takes a nonnegative integer j and returns an integer value chosen uniformly randomly from the 2^j integers in the range $[0, 2^j)$ (which is equivalent to independently and uniformly choosing j bits at random and using them as binary digits to represent an integer value). Second, *allZeroRandomBits*(j) takes a nonnegative integer j and returns *true* with probability 2^{-j} and *false* with probability $1 - 2^{-j}$ (which is equivalent to independently and uniformly choosing j bits at random and testing whether they all happen to be 0). Note that *randomBits*(0) always returns 0 and *allZeroRandomBits*(0) always returns *true*.

3 ADDING APPROXIMATE COUNTERS

Generalizing the observation of Csűrös to our framework, f may be derived from Q as follows:

$$f_k = f(\tau^k(S_0)) = \sum_{0 \le i < k} \frac{1}{Q(\tau^i(S_0))} = \sum_{0 \le i < k} \frac{1}{q_i}$$

Note that $f_0 = 0$. Because each probability q_i lies in the range $[0, 1]$, we have $\frac{1}{q_i} \ge 1$, and therefore the values $f_k = f(\tau^k(S_0))$ are strictly monotonic in k, and then some: $f_k + 1 \le f_{k+1}$. Therefore, we can uniquely define a representation-finding function $\varphi : [0, +\infty) \to T$ that given any nonnegative real number v returns $\tau^K(S_0)$, where K is the unique integer such that $f_K \le v < f_{K+1}$. Because the function φ produces a "quantized" result, it is a left inverse to f but not a right inverse: $\varphi(f(\tau^K(S_0))) = \tau^K(S_0)$ for all $k \ge 0$, but in general it is not true that $f(\varphi(v)) = v$; the best we can say is that $f(\varphi(v)) \le v < f(\tau(\varphi(v)))$. A general implementation of φ is as follows:

```
1: procedure φ(v)
2:     let S ← S_0
3:     loop
4:         let S′ ← τ(S)
5:         if v < f(S′) then return S
6:         S ← S′
7:     end loop
```

However, specific counter representations typically permit a much faster (non-iterative) implementation. Just as an implementation of τ may in practice signal an overflow error, a specialized implementation of φ may likewise signal an overflow error if its argument is too large.

Given an implementation of φ for a specific sort of counter, we can "add" two counter representation values x and z so the expected estimated value of the sum $x \oplus z$ equals the sum of

their individual expected estimated values (provided that the processes that produced x and z are statistically independent):

$$x \oplus z = \begin{cases} \tau(\varphi(f(x) + f(z))) & \text{with probability } \Delta \\ \varphi(f(x) + f(z)) & \text{with probability } (1 - \Delta)' \end{cases}$$

$$\text{where } \Delta = \frac{(f(x) + f(z)) - f(\varphi(f(x) + f(z)))}{f(\tau(\varphi(f(x) + f(z)))) - f(\varphi(f(x) + f(z)))}.$$

The expected value of $f(x \oplus z)$ will then be

$$\begin{aligned} &(1 - \Delta) f(\varphi(f(x) + f(z))) + \Delta f(\tau(\varphi(f(x) + f(z)))) \\ &= f(\varphi(f(x) + f(z))) + \Delta\Big(f(\tau(\varphi(f(x) + f(z)))) - f(\varphi(f(x) + f(z)))\Big) \\ &= f(\varphi(f(x) + f(z))) + (f(x) + f(z)) - f(\varphi(f(x) + f(z))) \\ &= f(x) + f(z), \end{aligned}$$

as desired.

We can express this addition operation as an algorithm that modifies a counter X by adding into it the estimated value of a statistically independent counter Z (we do this so it will be similar in form to the *increment* operation):

```
1:  procedure add(var X: T, Z: T)
2:      let v ← f(X)
3:      let w ← f(Z)
4:      let S ← v + w
5:      let K ← φ(S)
6:      let V ← f(K)
7:      let W ← f(τ(K))
8:      let Δ ← (S−V)/(W−V)
9:      if random() < Δ then
10:         X ← τ(K)
11:     else
12:         X ← K
```

This algorithm is not terribly mysterious: it adds two counters X and Z by computing their expected values v and w, adding those values to get a sum S, then mapping that sum back to one of the two counter states whose expected values straddle S, in such a way that the expected value of the result is S. The real point is that we will use this general algorithm as a template for addition algorithms specialized to particular counter representations by inlining specific definitions of f and τ and φ, and then optimizing. This strategy guarantees that appropriate expected values are maintained. However, it is necessary to provide a separate proof for each kind of counter that the variance is bounded.

4 GENERAL MORRIS COUNTERS

For the general probabilistic counter defined by Morris,

$$\text{type } T \text{ is } \mathbb{N} \qquad S_0 = 0 \qquad \tau(x) = x + 1 \qquad Q(x) = q^{-x}$$

(where q is a fixed constant equal to $1 + \frac{1}{a}$, where a is the parameter used by Morris); it follows that

$$f(x) = \frac{q^x - 1}{q - 1}$$

(which is equivalent to the formula $a((1+\frac{1}{a})^x - 1)$ given by Morris), and

$$\varphi(v) = \left\lfloor \log_q((q-1)v+1) \right\rfloor .$$

Then, *increment* and *read* operations for such probabilistic counters may be described in the following manner:

```
1: procedure increment(var X: ℕ)
2:     if random() < q^{-X} then X ← X + 1
```

```
1: procedure read(var X: ℕ)
2:     return (q^X − 1)/⟨q−1⟩
```

We use angle brackets $\langle \cdots \rangle$ to indicate a constant ($q-1$ in this case) whose value can be computed at compile time.

If instead we let the counter representation type T be the set of values $\mathbb{Z}_{2^b} = \{0, 1, 2, \ldots, 2^b - 1\}$, representable in a b-bit word as unsigned binary integers, then the *increment* operation may perform overflow checking:

```
1: procedure increment(var X: ℤ_{2^b})
2:     if random() < q^{-X} then
3:         if X ≠ ⟨2^b − 1⟩ then X ← X + 1
4:         else overflow error
```

In some implementation contexts, when b is relatively small, it may be worthwhile to tabulate the functions Q and f—that is, to preconstruct two arrays Q' and f' containing the values of $Q(X)$ and $f(X)$ for all $0 \le X < 2^b$, but let $Q'[2^b - 1] = 0$ to serve as a sentinel that will effectively enforce saturation on overflow. Such use of tabulations is certainly worth considering when $b = 8$, and perhaps even when b is as large as 16. Then, the *increment* and *read* operations are simply:

```
1: procedure increment(var X: ℤ_{2^b})
2:     if random() < Q'[X] then X ← X + 1
1: procedure read(var X: ℤ_{2^b})
2:     return f'[X]
```

Counter Z may be added into counter X as follows:

```
1: procedure add(var X: ℕ, Z: ℕ)
2:     let v ← (q^X − 1)/⟨q−1⟩
3:     let w ← (q^Z − 1)/⟨q−1⟩
4:     let S ← v + w
5:     let K ← ⌊log_q(⟨q − 1⟩S + 1)⌋
6:     let V ← (q^K − 1)/⟨q−1⟩
7:     let W ← (q^{K+1} − 1)/⟨q−1⟩
8:     let Δ ← (S−V)/(W−V)
9:     if random() < Δ then X ← K + 1 else X ← K
```

If the computation of the base-q logarithm is not entirely accurate in the computation of K, then it may fall just under an integer value that it should have equaled or exceeded. If so, then K will be 1

smaller than it should have been. But this error is benign in this context, because then $\Delta > 1$, and so K will be incremented. There will be no further chance to increment K again, but that would have occurred only with probability commensurate with other floating-point errors.

However, in many practical situations, it is faster to avoid the computation of a logarithm entirely. If the implementation pretabulates the values of $f(x)$ in an array f' (as already described above for use by the *read* operation), then the array can be searched for the correct position. For this purpose it is best to make f' have length $2^b + 2$, with $f'[X] = f(X)$ for $0 \le X \le 2^b$, and place a sentinel value—either $+\infty$ or a very large finite value—in the last position at index $2^b + 1$. Also, note that $W - V$ can be simplified to q^k, so is it is helpful to pretabulate q^{-k} in a length-2^b array p. Then, for b-bit words with overflow checking, we have:

```
 1: procedure add(var X: ℤ_{2^b}, Z: ℤ_{2^b})
 2:   let S ← f'[X] + f'[Z]
 3:   let K ← max(X, Z)
 4:   while S ≥ f'[K + 1] do K ← K + 1
 5:   if K ≤ ⟨2^b − 1⟩ then
 6:     let V ← f'[K]
 7:     if random() < (p[K])(S − V) then
 8:       if K ≠ ⟨2^b − 1⟩ then X ← K + 1
 9:       else overflow error: X ← ⟨2^b − 1⟩
10:     else X ← K
11:   else overflow error: X ← ⟨2^b − 1⟩
```

If $q = 1.08$, for example, then the **while** loop should iterate no more than 8 times, so this may be much faster that the calculation of a logarithm in software. (In Section 13, we confirm this general intuition by presenting a C implementation of this technique and measurements of its use within one specific computational environment.)

Note that the explicit overflow checking only ensures that the value of K, once computed, will fit in a b-bit word. It is assumed that the computation of S will use an arithmetic with sufficient range to avoid overflow. This is typically not difficult in practice: if $b = 8$, and S is computed using standard IEEE 754 double-precision floating-point operations, then the computation of S will never suffer overflow.

Morris asserts (without proof) that the variance in the estimated value of a counter after n *increment* operations have been performed is $n(n-1)/2a = \frac{q-1}{2}n(n-1)$. In Section 11 of this article, our Lemma 11.1 provides an explicit proof of a generalization of this proposition, and Theorem 11.4 shows that when *add* operations are also used, the variance is bounded by $\frac{q-1}{2}n(n-1) + \rho$ where $\rho = \frac{1}{-2(q^2-4q+1)}$. Note that ρ lies in the range $[\frac{1}{6}, \frac{1}{4})$ for $1 < q \le 2$, so this is a reasonably tight bound on the variance even for small n.

5 BINARY MORRIS COUNTERS

For the simplest sort of probabilistic counter defined by Morris, we choose $q = 2$ (which corresponds to $a = 1$ as used by Morris):

$$\text{type } T \text{ is } \mathbb{N} \qquad S_0 = 0 \qquad \tau(x) = x + 1 \qquad Q(x) = 2^{-x}.$$

It follows that

$$f(x) = 2^x - 1,$$
$$\varphi(v) = \lfloor \log_2(v+1) \rfloor .$$

This makes the *increment* and *read* operations especially simple:

```
1: procedure increment(var X: ℕ)
2:     if allZeroRandomBits(X) then X ← X + 1
```

```
1: procedure read(var X: ℕ)
2:     return 2^X − 1
```

Skilled programmers know various clever ways to compute $2^X - 1$, depending on whether the result is to be represented as an integer (perhaps by using a left-shift operation) or as a floating-point value (using a scaling operation to adjust the exponent field). For an integer result, the *read* operation might be simply:

```
1: procedure read(var X: ℕ)
2:     return (1 ≪ X) − 1
```

Note that in C, C++, the Java programming language, or other languages that have similar rules for the shift operator <<, it may be important to write the literal 1 in the expression 1 << X as, say, `1L` or `1ull` to ensure that the result has a specific (sufficiently large) type, such as (respectively) `long` or `unsigned long long`; alternatively, it may be appropriate to use an explicit cast, as in this example:

```
uint64_t read(uint8_t x) {
     return (((uint64_t) 1) << x) - 1;
}
```

It is especially easy to add counter Z into counter X: because $f(k+1) = 2^{k+1} - 1 > 2(2^k - 1) = 2f(k)$, it follows that V is always equal to $f(\max(X, Z))$, so $S - V = f(\min(X, Z))$, and therefore $\Delta = \frac{2^{\min(X,Z)}-1}{2^{\max(X,Z)}}$. This leads to a very simple procedure that does not need to implement the φ function explicitly:

```
1: procedure add(var X: ℕ, Z: ℕ)
2:     let K ← max(X, Z)
3:     let L ← min(X, Z)
4:     if allZeroRandomBits(K − L) then
5:         if ¬allZeroRandomBits(L) then X ← K + 1
6:         else X ← K
7:     else X ← K
```

With overflow checking, *increment* and *add* look like this:

```
1: procedure increment(var X: ℤ_{2^b})
2:     if allZeroRandomBits(X) then
3:         if X ≠ ⟨2^b − 1⟩ then X ← X + 1
4:         else overflow error
```

```
1: procedure add(var X: ℤ_{2^b}, Z: ℤ_{2^b})
2:   let K ← max(X, Z)
3:   let L ← min(X, Z)
4:   if allZeroRandomBits(K − L) then
5:     if ¬allZeroRandomBits(L) then
6:       if K ≠ ⟨2^b − 1⟩ then X ← K + 1
7:       else overflow error: X ← K
8:     else X ← K
9:   else X ← K
```

While the binary Morris approximate counter is in principle a special case of the general Morris approximate counter with $q = 2$, our addition algorithm is rather different from the one presented for the general case. Therefore, in Section 11, we also present Theorem 11.6, a separate proof of bounded variance for binary Morris approximate counters when *add* operations are used, showing that the variance is bounded by $\frac{n(n-1)}{2}$ (no additive constant ρ is needed).

6 CSŰRÖS FLOATING-POINT COUNTERS

For the general (i.e., scaled) floating-point counters defined by Csűrös (2010), which are parametrized not only by a base q ($1 < q \leq 2$) but also by $M = 2^s$ for some integer $s \geq 0$, we have

$$\text{type } T \text{ is } \mathbb{N} \qquad S_0 = 0 \qquad \tau(x) = x + 1 \qquad Q(x) = q^{-\lfloor x/M \rfloor}.$$

For convenience, let $\mu = \frac{M}{q-1}$; it follows that

$$f(x) = (\mu + (x \bmod M))q^{\lfloor x/M \rfloor} - \mu,$$
$$\varphi(v) = d \cdot M + \left\lfloor \frac{v + \mu}{q^d} - \mu \right\rfloor \text{ where } d = \left\lfloor \log_q \frac{v + \mu}{\mu} \right\rfloor.$$

In effect, all bits of an integer x *except* the s lower order bits are treated as a binary[3] exponent $e = \lfloor x/M \rfloor$, and the s low-order bits of x (i.e., $x \bmod M$) are treated as a floating-point significand with an implicit leading 1-bit (that's why you have to add μ before multiplying by $q^{-\lfloor x/M \rfloor}$); finally, μ is subtracted so the integer counter representation 0 will map to the estimated value 0 (and, most pleasantly, every integer representation x less than M maps to the estimated value x). Again the *increment* operation is very simple, and *read* is reasonably simple:

```
1: procedure increment(var X: ℕ)
2:   if random() < q^{−⌊X/M⌋} then X ← X + 1
```

```
1: procedure read(var X: ℕ)
2:   return (μ + (X mod M))q^{⌊X/M⌋} − μ
```

[3]Csűrös (2010) primarily assumes binary counters ($q = 2$), but also briefly discusses "scaled floating-point counters," relating them to earlier work by Stanojević (2007). These bear the same relationship to binary Csűrös floating-point counters that general Morris counters have to binary Morris counters. We wish to add the observation that the primary motivation for requiring M to be a power of 2 is to make it easy to compute $\lfloor x/M \rfloor$ and $x \bmod M$ using bit shifting and masking operations. For some applications, using base-2 Csűrös floating-point counters with M an integer that is not a power of 2 may well be preferable to limiting M to be some power of 2 and then choosing a value of q other than 2. For other applications, it may be convenient to choose some $q < 2$ but then choose M so μ is an integer. Our pseudocode assumes only that M is an integer unless otherwise indicated, and the proofs in Section 11 assume only that M is an integer.

And again, in some implementation contexts, if the range of values for X is relatively small, it may be worthwhile to tabulate the functions Q and f as arrays Q' and f' (but letting the last element of Q' be 0) so the *increment* and *read* operations are simply:

```
1: procedure increment(var X: ℤ_{2^b})
2:     if random() < Q'[X] then X ← X + 1
1: procedure read(var X: ℤ_{2^b})
2:     return f'[X]
```

But for $q = 2$ and $M = 2^s$, shifting and bitwise operations can be useful (we show a version of *increment* with overflow checking):

```
1: procedure increment(var X: ℤ_{2^b})
2:     if allZeroRandomBits(X ≫ s) then
3:         if X ≠ ⟨2^b − 1⟩ then X ← X + 1 else overflow error
1: procedure read(var X: ℕ)
2:     return ((M + (X & ⟨M − 1⟩)) ≪ (X ≫ s)) − M
```

In general (for any q and M), addition goes like this:

```
1: procedure add(var X: ℕ, Z: ℕ)
2:     let v ← (μ + (X mod M))q^⌊X/M⌋ − μ
3:     let w ← (μ + (Z mod M))q^⌊Z/M⌋ − μ
4:     let S ← v + w
5:     let d ← ⌊log_q (S+μ)/μ⌋
6:     let K ← d · M + ⌊(S+μ)/q^d − μ⌋
7:     let V ← (μ + (K mod M))q^⌊K/M⌋ − μ
8:     let W ← (μ + ((K + 1) mod M))q^⌊(K+1)/M⌋ − μ
9:     let Δ ← (S−V)/(W−V)
10:    if random() < Δ then X ← K + 1 else X ← K
```

Note that $d \geq 0$; indeed, $d \geq \max(\lfloor X/M \rfloor, \lfloor Z/M \rfloor)$.

Now, $1 < q \leq 2$ and $0 \leq r < 1$ together imply $q^r < 2$; therefore $\lfloor \frac{S+\mu}{2^d} - \mu \rfloor = \lfloor \frac{S+\mu}{q^{\lfloor \log_q \frac{S+\mu}{\mu} \rfloor}} - \mu \rfloor = \lfloor \frac{S+\mu}{q^{\log_q \frac{S+\mu}{\mu} - r}} - \mu \rfloor = \lfloor q^r \mu - \mu \rfloor \leq q^r \mu - \mu = (q^r - 1)\mu < \mu \leq M$. Therefore, $K \bmod M = \lfloor \frac{S+\mu}{q^d} - \mu \rfloor$, and so $\lfloor K/M \rfloor = d$ and we have

$$\begin{aligned} S - V &= S - \left((\mu + (K \bmod M)) q^{\lfloor K/M \rfloor} - \mu \right) \\ &= S - \left(\left(\mu + \left\lfloor \frac{S+\mu}{q^d} \right\rfloor - \mu \right) q^{\lfloor K/M \rfloor} - \mu \right) \\ &= S - \left(\left\lfloor \frac{S+\mu}{q^d} \right\rfloor q^d - \mu \right) \\ &= S - \left(((S+\mu) - ((S+\mu) \bmod q^d)) - \mu \right) \\ &= (S+\mu) \bmod q^d. \end{aligned}$$

Next, it's worth simplifying $W - V$ by case analysis as follows:

(a) If $(K \bmod M) < M-1$, then $\lfloor (K+1)/M \rfloor = \lfloor K/M \rfloor$, and

$$\begin{aligned} W - V &= \Big((\mu + ((K+1) \bmod M))q^{\lfloor (K+1)/M \rfloor} - \mu\Big) - \Big((\mu + (K \bmod M))q^{\lfloor K/M \rfloor} - \mu\Big) \\ &= \Big(\mu + ((K+1) \bmod M)\Big)q^{\lfloor K/M \rfloor} - \Big(\mu + (K \bmod M)\Big)q^{\lfloor K/M \rfloor} \\ &= \Big(((K+1) \bmod M) - (K \bmod M)\Big)q^{\lfloor K/M \rfloor} \\ &= \Big(((K+1) - K) \bmod M\Big)q^{\lfloor K/M \rfloor} = q^{\lfloor K/M \rfloor}. \end{aligned}$$

(b) If $(K \bmod M) = M-1$, then $\lfloor (K+1)/M \rfloor = \lfloor K/M \rfloor + 1$:

$$\begin{aligned} W - V &= \Big((\mu + ((K+1) \bmod M))q^{\lfloor (K+1)/M \rfloor} - \mu\Big) - \Big((\mu + (K \bmod M))q^{\lfloor K/M \rfloor} - \mu\Big) \\ &= \Big(\mu + ((K+1) \bmod M)\Big)q^{\lfloor K/M \rfloor + 1} - \Big(\mu + (K \bmod M)\Big)q^{\lfloor K/M \rfloor} \\ &= \Big(q\mu + q((K+1) \bmod M)\Big)q^{\lfloor K/M \rfloor} - \Big(\mu + (K \bmod M)\Big)q^{\lfloor K/M \rfloor} \\ &= \Big(q\mu + q \cdot 0 - \mu - (M-1)\Big)q^{\lfloor K/M \rfloor} = q^{\lfloor K/M \rfloor}. \end{aligned}$$

Therefore, in all cases, $W - V = q^{\lfloor K/M \rfloor} = q^d$.

So, after some simplification, we have:

```
1: procedure add(var X: ℕ, Z: ℕ)
2:     let S ← (μ + (X mod M))q^⌊X/M⌋ +
                (μ + (Z mod M))q^⌊Z/M⌋ − ⟨2μ⟩
3:     let d ← ⌊log_q (S+μ)/μ⌋
4:     let K ← d · M + ⌊(S+μ)/q^d − μ⌋
5:     if random() < (S+μ)/q^d mod 1 then X ← K + 1 else X ← K
```

However, in practice it may be faster to search an array f' as described in Section 4.

If we restrict our attention to $q = 2$ and $M = 2^s$, and assume the use of a machine word B bits wide for representing estimated values (in contrast to words b bits wide that may be used for counter representation values) and an instruction to count the number of leading zeros in a B-bit word, then we can recast the algorithm (with overflow checking) as follows (letting $S' = S + \mu > 0$):

```
1: procedure add(var X: ℤ_{2^b}, Z: ℤ_{2^b})
2:     let S' ← ((ℤ_{2^B})(M + (X & ⟨M − 1⟩)) ≪ (X ≫ s)) +
                ((ℤ_{2^B})(M + (Z & ⟨M − 1⟩)) ≪ (Z ≫ s)) − M
3:     let d ← ⟨B − (s + 1)⟩ − countLeadingZeros(S')
4:     let K ← (d ≪ s) + (S' ≫ d) − M
5:     if K ≤ ⟨2^b − 1⟩ then
6:         if randomBits(d) < (S' & ((1 ≪ d) − 1)) then
7:             if K ≠ ⟨2^b − 1⟩ then X ← K + 1
8:             else overflow error: X ← ⟨2^b − 1⟩
9:         else X ← K
10:    else overflow error: X ← ⟨2^b − 1⟩
```

Section 11 presents Theorem 11.10, a proof of bounded variance for scaled Csűrös approximate counters when *add* operations are used.

7 DLM PROBABILITY COUNTERS

For the probabilistic counter defined by DLM (2013), where the value in a counter is not approximately the logarithm of the number of increment operations but rather the probability that the next increment operation should change the counter:

$$\text{type } T \text{ is the closed real interval } [0,1] \qquad S_0 = 1.0 \qquad \tau(x) = \frac{x}{q} \qquad Q(x) = x.$$

If, as before, we let $q = 1 + \frac{1}{a}$ (thus $a = \frac{1}{q-1}$), then it follows that

$$f(x) = \frac{a}{x} - a$$

(which is equivalent to the formula $a(\frac{1}{x} - 1)$ given by DLM but has one fewer operation). Then, the *increment* and *read* operations may be implemented as follows:

```
1: procedure increment(var X: [0, 1])
2:     if random() < X then X ← ⟨q^{-1}⟩X
1: procedure read(var X: [0, 1])
2:     return a/X − a
```

The counter representation is not quantized to integers, only to floating-point approximations[4] of real numbers, so it may not be necessary in practice when implementing the φ function to choose randomly between two possible values, though for complete accuracy one would indeed perform the floating-point calculations to "infinite precision" and then do the final floating-point rounding in an appropriately probabilistic manner, as described by Forsythe (1959) and later by Callahan (1976) and Parker (1997, Section 6.4, 2000). Assuming no need to be that finicky, we pretend that φ is an exact inverse of f:

$$\varphi(v) = \frac{a}{v + a},$$

and so counter Z may be added into counter X as follows:

```
1: procedure add(var X: [0, 1], Z: [0, 1])
2:     let S ← (a/X − a) + (a/Z − a)
3:     let K ← a/(S+a)
4:     X ← K
```

After simplification, this becomes:

```
1: procedure add(var X: [0, 1], Z: [0, 1])
2:     X ← 1 / (1/X + 1/Z − 1)
```

[4]Although we cannot resist pointing out that fixed-point fractions may be a practical alternate representation for probabilities, with the advantage of allowing the use of a random number generator that generates random b-bit integers, which may be faster than generating random floating-point values.

The remarkable thing about this representation is that the parameter a is not required for computing the sum of two counters, *provided*, of course, that the two counters do use the same parameter a.

While DLM probability counters may not require fewer bits for their representation than ordinary integer counters, they do share with other kinds of approximate counters the property that the *increment* operation does not always perform a write to memory.

8 DLM FLOATING-POINT COUNTERS

For the floating-point counters defined by DLM (2013), which are parametrized by a constant $M = 2^s$ for some integer $s \geq 0$ and by a second constant Θ (which DLM calls `MantissaThreshold`) that is a positive even integer not greater than M, we have

$$\text{type } T \text{ is } \mathbb{N} \qquad S_0 = 0 \qquad Q(x) = 2^{-\lfloor x/M \rfloor},$$

$$\tau(x) = \textbf{if } (k \bmod M) + 1 < \Theta \textbf{ then } k + 1 \textbf{ else } k - (k \bmod M) + M + \tfrac{\Theta}{2}$$

and it follows that

$$f(x) = (k \bmod M)2^{\lfloor k/M \rfloor}.$$

In effect, all bits of an integer k *except* the s lower order bits are treated as a binary exponent $e = \lfloor k/M \rfloor$, and the s low-order bits of k (i.e., $k \bmod M$) are treated as a floating-point significand that does *not* have an implicit leading 1-bit and whose leading bit may or may not be 1 (i.e., the representation is not necessarily in normalized form). As a result, there is some redundancy in the representation: two or more counter representation values may map to the same estimated value, and φ is not a function but rather a relation. One similarity to the floating-point representation used by Csűrös is that every integer representation k less than M maps to the estimated value k. Again the *increment* operation is very simple, and *read* is reasonably simple:

```
1: procedure increment(var X: ℕ)
2:     if allZeroRandomBits(⌊X/M⌋) then
3:         if (X mod M) + 1 < Θ then X ← X + 1
4:         else X ← X − (X mod M) + ⟨M + Θ/2⟩
```

```
1: procedure read(var X: ℕ)
2:     return (X mod M)2^⌊X/M⌋
```

Again, we cleverly use shifting and bitwise operations:

```
1: procedure increment(var X: ℕ)
2:     if allZeroRandomBits(X ≫ s) then
3:         if (X & ⟨M − 1⟩) < ⟨Θ − 1⟩ then X ← X + 1
4:         else X ← (X & ⟨¬(M − 1)⟩) + ⟨M + Θ/2⟩
```

```
1: procedure read(var X: ℕ)
2:     return (X & ⟨M − 1⟩) ≪ (X ≫ s)
```

DLM (2013) points out that the purpose of using a redundant representation and allowing Θ to be smaller than M is to allow a choice of representations that are equivalent in value but differ in their probability of changing the counter during an *increment* operation; this flexibility is

used to address situations in which there appears to be high contention among multiple threads for a shared counter. We will assume that this flexibility is not needed when performing an *add* operation, and therefore we are free to choose an implementation for φ that does not take Θ into account:

```
1:  procedure add(var X: ℕ, Z: ℕ)
2:    let S ← (X mod M)2^⌊X/M⌋ + (Z mod M)2^⌊Z/M⌋
3:    if S < M then X ← S
4:    else
5:      let d ← ⌊log2(S + 1)⌋ − ⟨s + 1⟩
6:      let K ← d · 2^s + ⌊S / 2^d⌋
7:      let V ← (K mod M)2^⌊K/M⌋
8:      let W ← ((K + 1) mod M)2^⌊(K+1)/M⌋
9:      let Δ ← (S−V)/(W−V)
10:     if random() < Δ then
11:       if (K mod M) = ⟨M − 1⟩ then
12:         X ← (d + 1)2^s + ⟨M/2⟩
13:       else X ← K + 1
14:     else X ← K
```

After simplification and introduction of bitwise operations this is:

```
1:  procedure add(var X: ℕ, Z: ℕ)
2:    let S ← ((X & ⟨M − 1⟩) ≪ (X ≫ s)) +
                ((Z & ⟨M − 1⟩) ≪ (Z ≫ s))
3:    if S < M then X ← S
4:    else
5:      let d ← ⟨B − s⟩ − countLeadingZeros(S′)
6:      let K ← (d ≪ s) + (S ≫ d)
7:      if randomBits(d) < (S & ((1 ≪ d) − 1)) then
8:        if (K mod M) = ⟨M − 1⟩ then
9:          X ← K + ⟨M/2 + 1⟩
10:       else X ← K + 1
11:     else X ← K
```

With overflow checking, the *increment* and *add* operations look like this (we assume that one may disobey the Θ threshold if doing so will postpone an overflow error):

```
1:  procedure increment(var X: ℕ)
2:    if allZeroRandomBits(X ≫ s) then
3:      if (X & ⟨M − 1⟩) < ⟨Θ − 1⟩ then X ← X + 1
4:      else if (X ≫ s) = ⟨2^(b−s) − 1⟩ then
5:        if X ≠ ⟨2^b − 1⟩ then X ← X + 1
6:        else overflow error
7:      else X ← X & ⟨¬(M − 1)⟩) + ⟨M + Θ/2⟩
```

```
1:  procedure add(var X: ℕ, Z: ℕ)
2:    let S ← ((X & ⟨M − 1⟩) ≪ (X ≫ s)) +
                ((Z & ⟨M − 1⟩) ≪ (Z ≫ s))
3:    if S < M then X ← S
4:    else
5:      let d ← ⟨B − s⟩ − countLeadingZeros(S′)
6:      let K ← (d ≪ s) + (S ≫ d)
7:      if K ≤ ⟨2^b − 1⟩ then
8:        if randomBits(d) < (S & ((1 ≪ d) − 1)) then
9:          if K ≠ ⟨2^b − 1⟩ then
10:           if (K mod M) = ⟨M − 1⟩ then
11:             X ← K + ⟨M/2 + 1⟩
12:           else X ← K + 1
13:         else overflow error: X ← ⟨2^b − 1⟩
14:       else X ← K
15:     else overflow error: X ← ⟨2^b − 1⟩
```

9 ADDING COUNTERS OF DIFFERENT TYPES

If counter X is of type T_1 (with transition function τ_1, estimation function f_1, and representation-finding function φ_1), and we wish to add into it a counter Z of type T_2 (with estimation function f_2), then our general algorithm for the *add* operation accommodates this easily:

```
1:  procedure add(var X: T1, Z: T2)
2:    let v ← f1(X)
3:    let w ← f2(Z)
4:    let S ← v + w
5:    let K ← φ1(S)
6:    let V ← f1(K)
7:    let W ← f1(τ1(K))
8:    let Δ ← (S−V)/(W−V)
9:    if random() < Δ then X ← τ1(K) else X ← K
```

Whether a specific optimized version is worthwhile will depend on the two specific choices of counter representation.

It is even possible to add two counters of different types to produce a result in a third representation T_3 (with transition function τ_3, estimation function f_3, and representation-finding function φ_3):

```
1:  procedure add(X: T1, Z: T2): T3
2:    let v ← f1(X)
3:    let w ← f2(Z)
4:    let S ← v + w
5:    let K ← φ3(S)
6:    let V ← f3(K)
7:    let W ← f3(τ3(K))
8:    if random() < (S−V)/(W−V) then return τ3(K) else return K
```

At this time, we do not know of any specific practical application for this ability.

10 DISCUSSION

Using approximate counters (perhaps with addition) is easy: they are a direct replacement for increment-only integer counters. The API is trivial: *read*, *increment*, and perhaps *add* operations, plus a way to initialize them to zero. The question is under what circumstances (i.e., in what sort of algorithm) it is appropriate to do such a replacement. We speak to that briefly in the last paragraph of Section 16.

Approximate counters deliver only a (unbiased) statistical estimate of the true number of increment operations (Flajolet 1985). The standard deviation of this estimate is the square root of the variance (this is the very definition of "standard deviation"). If the statistical distribution of values taken on by an approximate counter after some number n of *increment* operations were a (continuous) normal distribution, then we could apply the standard "68-95-99.7 Rule" for normal distributions and expect the statistical estimate to be less than one standard deviation away from the true value about 68% of the time, to be less than two standard deviations away from the true value about 95% of the time, and to be less than three standard deviations away from the true value "nearly always" (about 99.7% of the time). However, the distribution of an approximate counter is actually a discrete distribution—which is, however, easily computed by recurrence. We have done so for eight choices of parameters q and M and for $0 \le n \le 1000$. (As examples, Figure 1 shows the discrete distributions for the estimated value of a general Morris approximate counter (for $q = 1.2$) after 3, 4, 5, 6, 7, 8, 9, and 10 *increment* operations, and Figure 2 shows the discrete distributions for the estimated value of a general Csűrös approximate counter (for $q = 1.2$, $M = 4$) after 6, 7, 8, 9, 10, 11, 12, and 13 *increment* operations.) For each of the 8,008 distributions, we computed the mean, standard deviation, and the fraction of probability mass in the distribution that is within one, two, or three standard deviations from the mean. The results are graphed in Figures 3 and 4. In each chart, the x-axis is the number of *increment* operations and the y-axis is the fraction of probability mass; the lowest data line corresponds to one standard deviation, the middle data line to two standard deviations, and the uppermost data line to three standard deviations. These data lines are "choppy" because of quantization effects of the discrete distributions; when the data line for one standard deviation dips down low, it is because a substantial chunk of probability mass happens to lie just outside that boundary, rather than at some great distance. Even so, that data line pretty much stays above 0.6 for all but the smallest values of n. It will be seen from this exemplary data that, at least for larger values of n, the 68-95-99.7 Rule does approximately describe these distributions despite the fact that they are discrete.

As a specific example, by our Theorem 11.4 (which we present below in Section 11.1), the variance of a general Morris counter is no larger than $\frac{q-1}{2}n(n-1) + \rho$ (where $\frac{1}{6} \le \rho < \frac{1}{4}$), so the standard deviation is the square root of that, and for large n this is approximately $(\sqrt{(q-1)/2})n$; for $q = 1.1$, the standard deviation is approximately $0.22n$. Therefore, we can expect a counter estimate to be within 22% of the true value about 2/3 of the time, to be within 44% of the true value about 95% of the time, and to be within 66% of the true value "nearly always."

Csűrös counters and Morris counters have similar behavior when the number of increments is large (not surprising, because Morris counters are the special case of Csűrös counters with $M = 1$). An advantage of Csűrös counters over Morris counters is that they are exact until the number of increments reaches M, but there is no free lunch: in exchange for achieving a variance of exactly 0 for small values of the counter, the upper bound on the variance of larger values of the counter is slightly worse. Consider an 8-bit general Csűrös counter with $M = M_C = 8$ and $q = q_C = 1.2$; the largest representable value is $(\frac{8}{1.2-1} + (255 \bmod 8))(1.2)^{\lfloor 255/8 \rfloor} - \frac{8}{1.2-1} = (40+7)(1.2)^{31} - 40 \approx 13348.02$. An 8-bit general Morris counter described by $q = q_M$ whose highest representable value is the same must satisfy $\frac{q_M^{255}-1}{q_M-1} \approx 13348.02$; solving for q_M gives $q_M \approx 1.022667$. So the coefficient

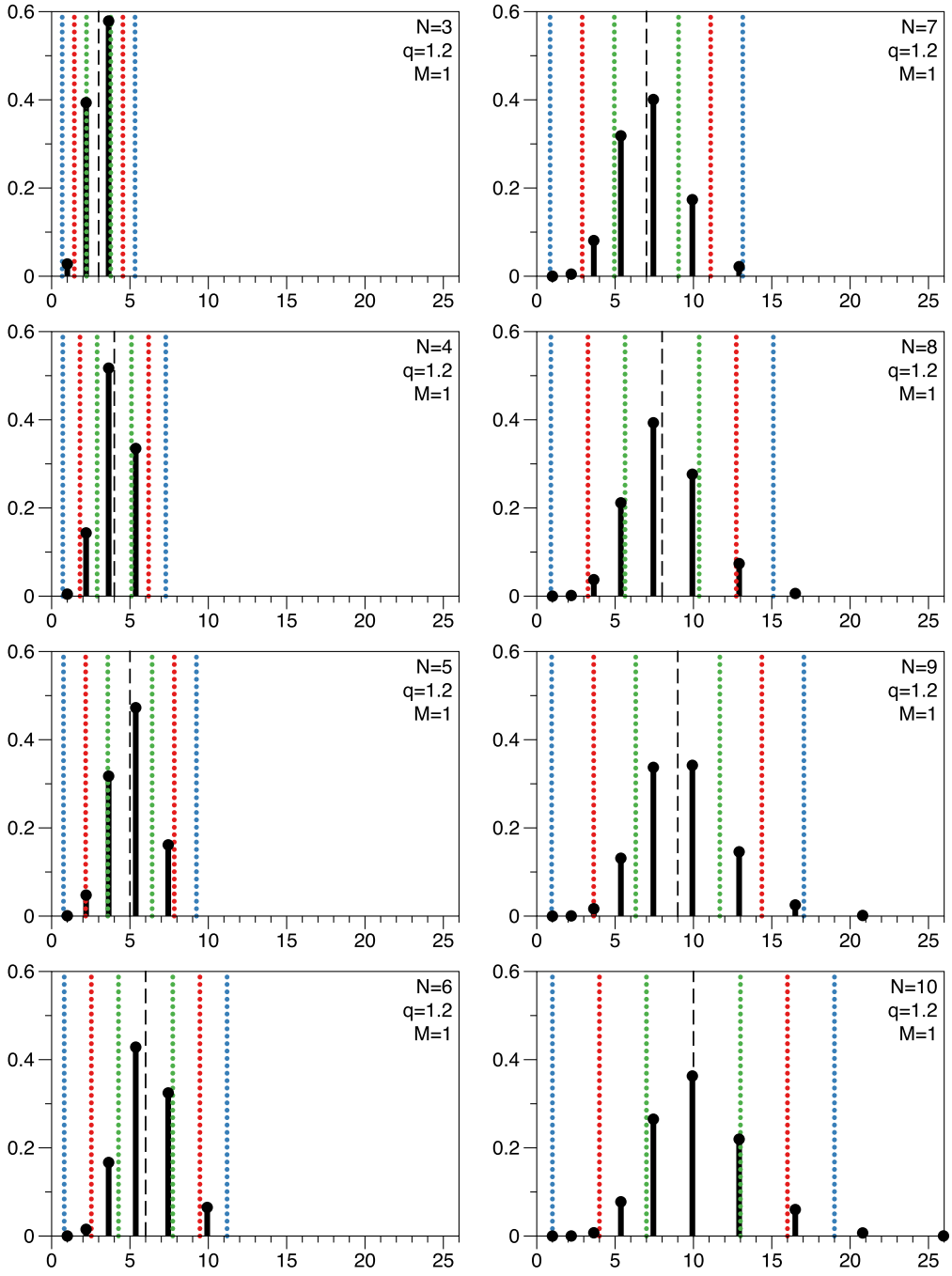

Fig. 1. The discrete statistical distributions of a general Morris approximate counter (for $q = 1.2$) after n *increment* operations (for $3 \leq n \leq 10$). The x-axis is the value of the estimator function f; the y-axis is probability mass. The vertical dashed line indicates the mean of the distribution (equal to n); the vertical dotted lines indicate distances of one (green), two (red), and three (blue) standard deviations from the mean.

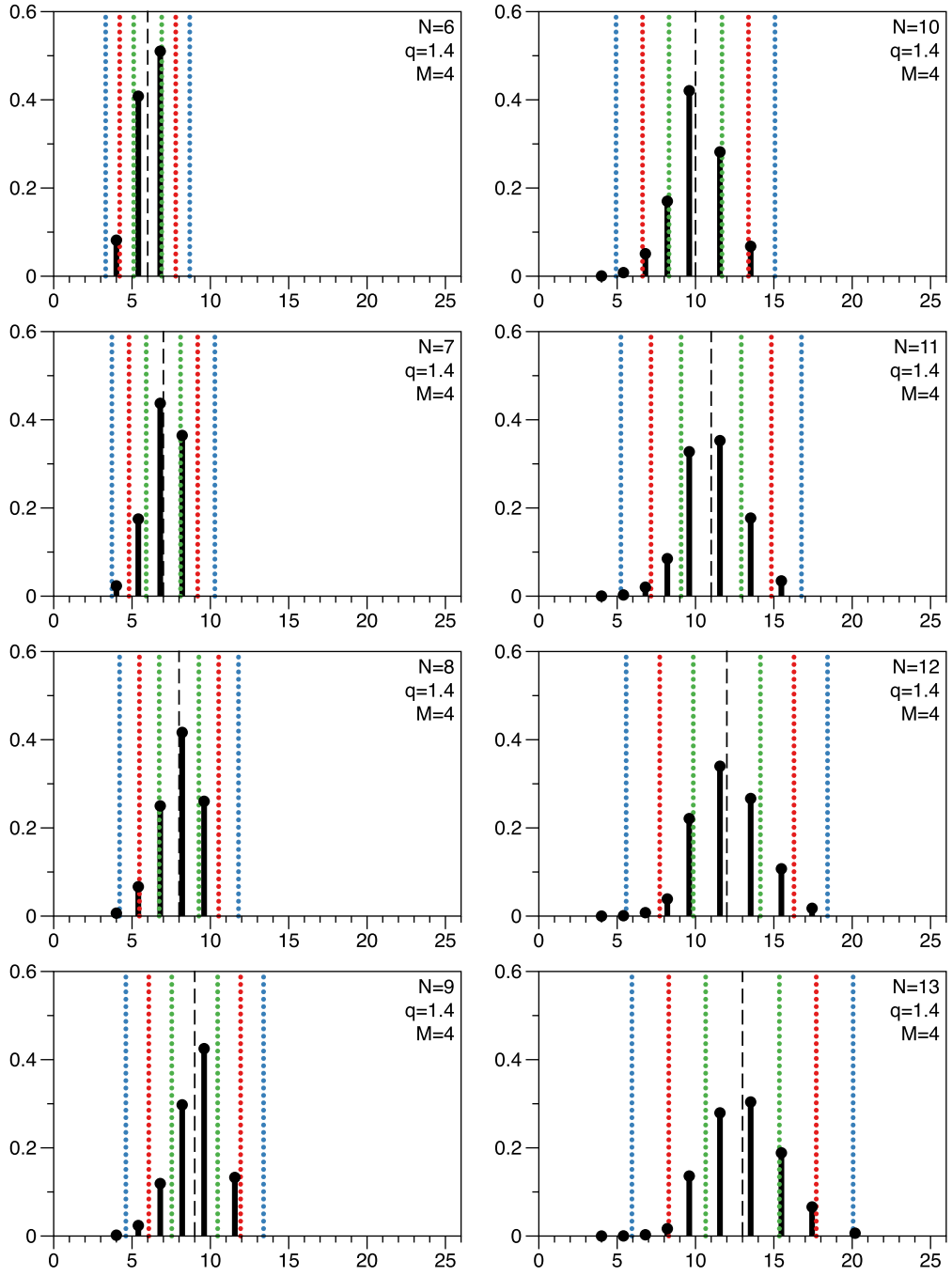

Fig. 2. The discrete statistical distributions of a general Csűrös approximate counter ($q = 1.4$, $M = 4$) after n *increment* operations (for $6 \leq n \leq 13$). The x-axis is the value of the estimator function f; the y-axis is probability mass. The vertical dashed line indicates the mean of the distribution (equal to n); the vertical dotted lines indicate distances of one (green), two (red), and three (blue) standard deviations from the mean.

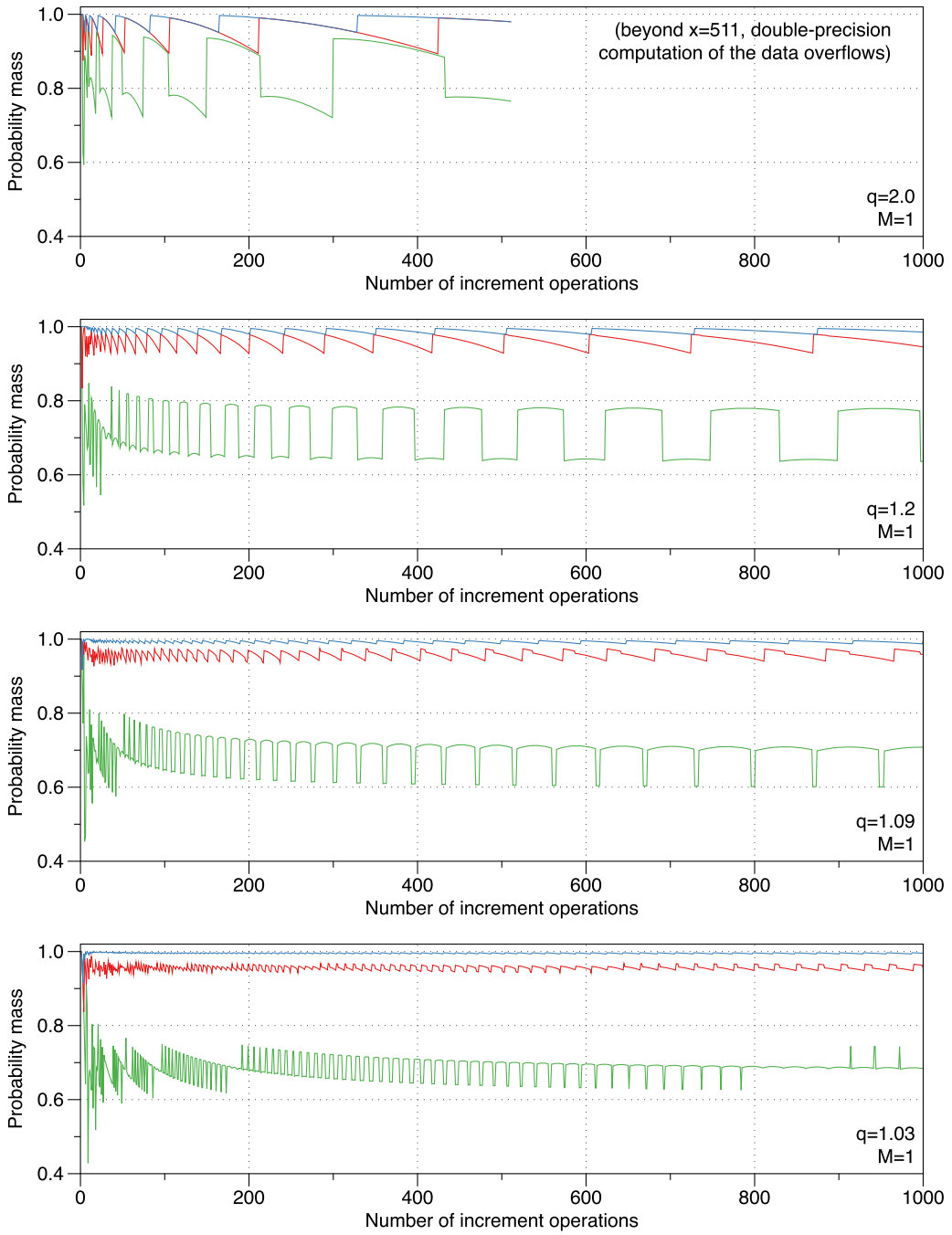

Fig. 3. The fraction of probability mass that is within one (lowest line, green), two (middle line, red), or three (highest line, blue) standard deviations from the mean after applying some number of increment operations to a zeroed Morris approximate counter (parameter $M = 1$) for four different values of the parameter q.Note that in all charts the lowest value shown for the y axis is 0.4, not 0.0.

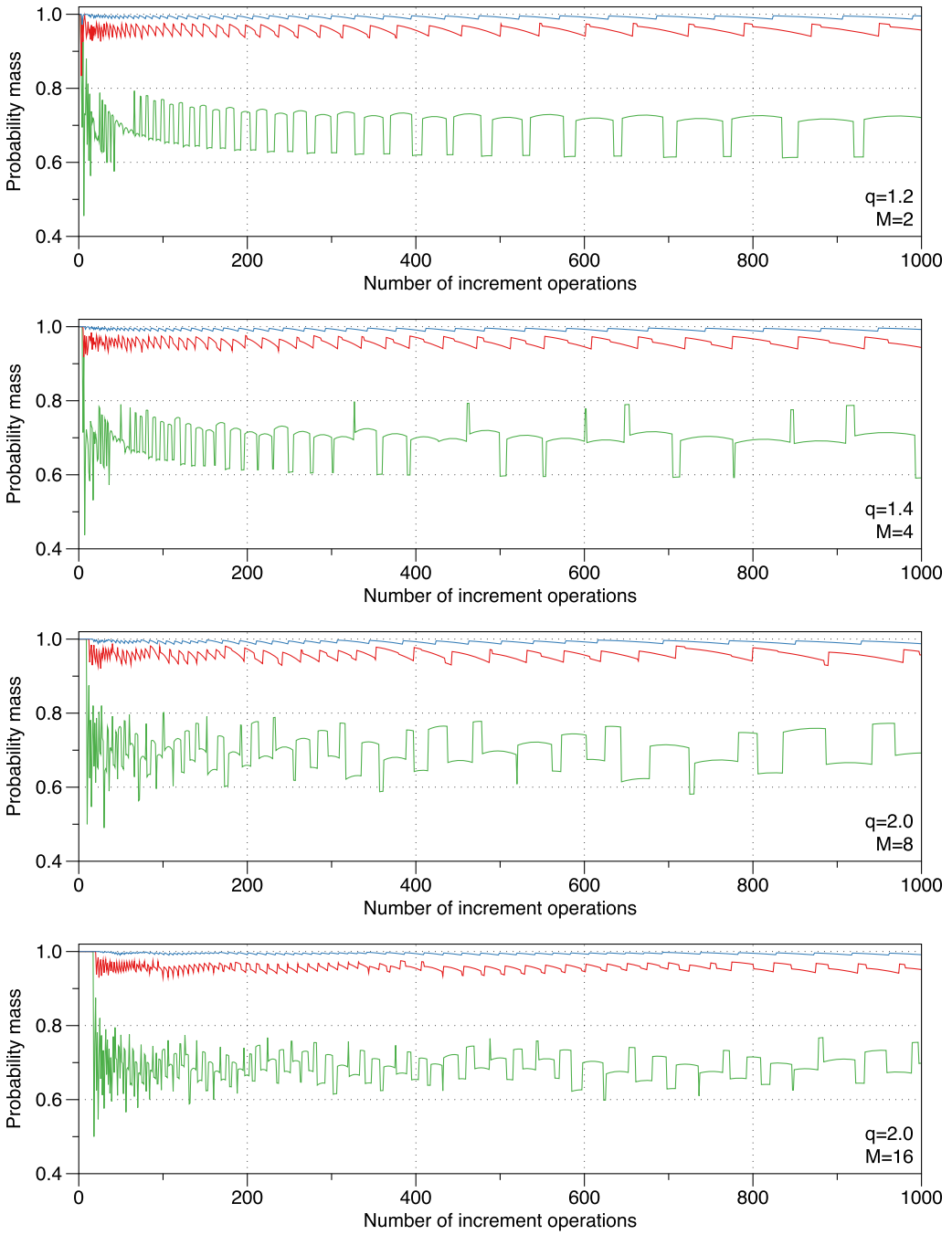

Fig. 4. The fraction of probability mass that is within one (lowest line, green), two (middle line, red), or three (highest line, blue) standard deviations from the mean after applying some number of increment operations to a zeroed Csűrös approximate counter for four different value combinations of the parameters q and M.Note that in all charts the lowest value shown for the y axis is 0.4, not 0.0.

of $n(n-1)$ in the formula given by Theorem 11.10 for the upper bound on the variance of the Csűrös counter is $\frac{1}{2\mu_C} = \frac{q_C-1}{2M_C} = \frac{0.2}{16} = 0.0125$, but the coefficient of $n(n-1)$ in the formula given by Theorem 11.4 for the upper bound on the variance of the Morris counter is only $\frac{q_M-1}{2} = \frac{0.022667}{2} = 0.011333$.

An advantage of a binary Csűrös counter over a non-binary Morris counter is that the incrementation code is cheaper. However, binary Csűrös counters support only a limited choice of maximum counter value. If your application happens to have a maximum counter value that well matches one of the possibilities for binary Csűrös counters, then that is a good choice. Otherwise, one might be better off using a general Morris counter with an appropriately calculated value for q, unless there is a specific desire to have small counter values be exact, in which case a general Csűrös counter may be a better choice, where one first chooses the necessary M and then calculates the appropriate value for q.

Table 1 shows the base-2 logarithm of the largest representable value of an 8-bit, 10-bit, 12-bit, or 16-bit general Csűrös counter for various values of q and M, truncated to one decimal place. (One can pack six 10-bit counters or five 12-bit counters into a 64-bit word.) This table is useful for getting an idea of what parameters might be appropriate for an approximate counter intended to replace a k-bit ordinary integer counter. For example, if an application uses 64-bit integer counters but there is reason to believe that no counter value ever exceeds 2^{40}, then it is reasonable to look for values just above 40 in the table. If we wish to use an 8-bit counter, then $M = 1$ (a general Morris counter) and $q = 1.11$ may be about right; we can also see that no 8-bit counter works for $M \geq 8$, but $M = 4$ and $q = 1.5$ is a possibility. If we are willing to use a 16-bit counter, then a binary Csűrös counter ($q = 2$) with $M = 2048$ should easily suffice, so this may be a better choice than $q = 1.08$ and $M = 256$.

11 PROOFS OF BOUNDED VARIANCE

In this section, we prove three theorems about bounds on the variance of approximate counters: one for general Morris counters, a slightly tighter bound for the special case of binary Morris counters, and one for general Csűrös counters. In each case, the result is that the variance of a counter whose expected value is n is bounded by an expression of the form $\xi\, n(n-1) + \eta$, where ξ and η are specific constants that depend on the counter represention parameters q and s. For each theorem the proof is inductive, following the structure of the computation that produced the counter by using *increment* and *add* operations. Each of the subsections below presents one theorem, preceded by relevant lemmas and their proofs.

11.1 Proof of Bounded Variance for General Morris Counters

Lemma 11.1 [Generalized from Morris (1978)]. *Let X be a general Morris approximate counter whose expected value is n, and let X' be the result of applying an increment operation to that counter. Let κ be an arbitrary constant.*

Assume $\mathrm{Var}\,(f(X)) \leq \frac{q-1}{2} n(n-1) + \kappa$; *then* $\mathrm{Var}\,(f(X')) \leq \frac{q-1}{2}(n+1)n + \kappa$. *Alternatively, assume* $\mathrm{Var}\,(f(X)) = \frac{q-1}{2} n(n-1) + \kappa$; *in that case,* $\mathrm{Var}\,(f(X')) = \frac{q-1}{2}(n+1)n + \kappa$.

Proof. Let $\Delta = q^{-X} = \frac{q-1}{(q^{X+1}-1)-(q^X-1)} = \frac{1}{f(X+1)-f(X)}$. Then:

$$\begin{aligned} &\mathrm{Var}\,(f(X')) \\ =&\mathbb{E}\left[(f(X'))^2\right] - (\mathbb{E}[f(X')])^2 \end{aligned}$$

Table 1. Base-2 Logarithms of Largest Representable Values for General Csűrös Counters

	8-bit counter: $\log_2 f(2^8-1)$						10-bit counter: $\log_2 f(2^{10}-1)$					
M	1	2	4	8	16	32	4	8	16	32	64	128
$q=2$	255.0	128.5	65.8	34.9	19.9	12.9	257.8	130.9	67.9	36.9	21.9	14.9
$q=1.9$	236.2	119.2	61.2	32.6	18.9	12.5	239.0	121.5	63.3	34.7	20.9	14.5
$q=1.8$	216.5	109.5	56.4	30.3	17.8	12.0	219.2	111.7	58.5	32.4	19.8	14.0
$q=1.7$	195.7	99.1	51.3	27.9	16.7	11.5	198.3	101.4	53.4	29.9	18.7	13.6
$q=1.6$	173.6	88.2	45.9	25.3	15.5	11.1	176.1	90.4	48.0	27.4	17.5	13.1
$q=1.5$	150.1	76.6	40.3	22.6	14.3	10.6	152.6	78.8	42.4	24.7	16.3	12.6
$q=1.4$	125.1	64.2	34.2	19.8	13.0	10.0	127.4	66.4	36.3	21.8	15.0	12.1
$q=1.3$	98.2	51.0	27.8	16.8	11.7	9.5	100.5	53.1	29.9	18.8	13.7	11.5
$q=1.2$	69.3	36.8	21.0	13.7	10.4	9.0	71.5	38.9	23.1	15.7	12.4	11.0
$q=1.1$	38.3	21.8	14.0	10.6	9.1	8.5	40.4	23.9	16.1	12.6	11.1	10.5
$q=1.09$	35.1	20.3	13.3	10.3	9.0	8.4	37.2	22.3	15.4	12.3	11.0	10.4
$q=1.08$	31.9	18.8	12.7	10.0	8.9	8.4	34.0	20.8	14.7	12.0	10.9	10.4
$q=1.07$	28.7	17.2	12.0	9.7	8.7	8.3	30.8	19.3	14.0	11.7	10.7	10.3
$q=1.06$	25.4	15.7	11.3	9.4	8.6	8.3	27.5	17.8	13.3	11.5	10.6	10.3
$q=1.05$	22.2	14.2	10.7	9.2	8.5	8.2	24.3	16.3	12.7	11.2	10.5	10.2
$q=1.04$	19.0	12.8	10.1	8.9	8.4	8.1	21.1	14.8	12.1	10.9	10.4	10.2
$q=1.03$	15.9	11.4	9.5	8.7	8.3	8.1	17.9	13.4	11.5	10.7	10.3	10.1
$q=1.02$	12.9	10.1	8.9	8.4	8.2	8.0	14.9	12.1	10.9	10.4	10.2	10.0
$q=1.01$	10.1	8.9	8.4	8.2	8.1	8.0	12.1	11.0	10.4	10.2	10.1	10.0

	12-bit counter: $\log_2 f(2^{12}-1)$						16-bit counter: $\log_2 f(2^{16}-1)$					
M	16	32	64	128	256	512	256	512	1024	2048	4096	8192
$q=2$	259.9	132.9	69.9	38.9	23.9	16.9	263.9	136.9	73.9	42.9	27.9	20.9
$q=1.9$	241.1	123.6	65.4	36.7	22.9	16.5	245.2	127.7	69.4	40.8	27.0	20.6
$q=1.8$	221.3	113.8	60.5	34.4	21.8	16.0	225.4	117.9	64.6	38.5	25.9	20.1
$q=1.7$	200.4	103.4	55.4	32.0	20.7	15.6	204.5	107.5	59.5	36.0	24.8	19.6
$q=1.6$	178.2	92.5	50.1	29.4	19.5	15.1	182.3	96.5	54.1	33.4	23.6	19.1
$q=1.5$	154.7	80.8	44.4	26.7	18.3	14.6	158.7	84.9	48.4	30.7	22.4	18.6
$q=1.4$	129.5	68.4	38.3	23.8	17.0	14.1	133.6	72.5	42.4	27.9	21.1	18.1
$q=1.3$	102.6	55.1	31.9	20.8	15.7	13.5	106.6	59.2	36.0	24.8	19.8	17.6
$q=1.2$	73.6	40.9	25.1	17.7	14.4	13.0	77.7	45.0	29.2	21.7	18.5	17.0
$q=1.1$	42.5	25.9	18.1	14.6	13.1	12.5	46.5	29.9	22.1	18.7	17.2	16.5
$q=1.09$	39.2	24.3	17.4	14.3	13.0	12.4	43.3	28.4	21.4	18.4	17.0	16.5
$q=1.08$	36.0	22.8	16.7	14.0	12.9	12.4	40.1	26.9	20.7	18.1	16.9	16.4
$q=1.07$	32.8	21.3	16.0	13.7	12.8	12.3	36.8	25.3	20.1	17.8	16.8	16.4
$q=1.06$	29.5	19.8	15.4	13.5	12.6	12.3	33.6	23.8	19.4	17.5	16.7	16.3
$q=1.05$	26.3	18.3	14.7	13.2	12.5	12.2	30.3	22.3	18.8	17.2	16.6	16.3
$q=1.04$	23.1	16.8	14.1	12.9	12.4	12.2	27.1	20.9	18.1	17.0	16.4	16.2
$q=1.03$	19.9	15.4	13.5	12.7	12.3	12.1	24.0	19.5	17.6	16.7	16.3	16.2
$q=1.02$	16.9	14.1	12.9	12.4	12.2	12.1	20.9	18.2	17.0	16.5	16.2	16.1
$q=1.01$	14.2	13.0	12.4	12.2	12.1	12.0	18.2	17.0	16.5	16.2	16.1	16.1

Each entry is the base-2 logarithm of the largest representable value of an 8-bit, 10-bit, 12-bit, or 16-bit general Csűrös counter for the specified value of q and M, truncated to one decimal place. A counter for which this value is k is comparable in range to an ordinary k-bit integer counter. Note that a Morris counter is the same as a Csűrös counter with $M=1$, and a binary counter is simply a general counter with $q=2$.

$$
\begin{aligned}
&= \mathbb{E}\Big[(1-\Delta)(f(X))^2 + \Delta(f(X+1))^2\Big] - (n+1)^2 \\
&= \mathbb{E}\left[(f(X))^2 + \Delta\Big((f(X+1))^2 - (f(X))^2\Big)\right] - (n+1)^2 \\
&= \mathbb{E}\Big[(f(X))^2\Big] + \mathbb{E}\left[f(X+1) + f(X)\right] - (n+1)^2 \\
&= \mathbb{E}\Big[(f(X))^2\Big] + \mathbb{E}\left[\frac{q((q-1)f(X)+1)-1}{q-1} + f(X)\right] - (n+1)^2
\end{aligned}
$$

$$
\begin{aligned}
&= \left(\mathrm{Var}\left(f(X)\right) + \left(\mathbb{E}\left[f(X)\right]\right)^2\right) + \mathbb{E}\left[(q+1)f(X)+1\right] - (n+1)^2 \\
&= \left(\mathrm{Var}\left(f(X)\right) + n^2\right) + (q+1)n + 1 - (n+1)^2 \\
&\leq \frac{q-1}{2}n(n-1) + \kappa + n^2 + (q+1)n + 1 - (n+1)^2 \\
&= \frac{q-1}{2}n(n+1) + \kappa,
\end{aligned}
$$

with equality holding when the equality $\mathrm{Var}\left(f(X)\right) = \frac{q-1}{2}n(n-1) + \kappa$ holds. □

Lemma 11.2. *Given values q and S such that $1 < q \leq 2$ and $S \geq 0$, let $K = \lfloor \log_q((q-1)S+1) \rfloor$, let $V = \frac{q^K-1}{q-1}$, let $W = \frac{q^{K+1}-1}{q-1}$, and let $A = (S-V)(W-S)$. Then, $A \leq \frac{((q-1)S+1)^2}{4q}$.*

Proof. From the definition of the floor operation $\lfloor \cdot \rfloor$, $K = \log_q((q-1)S+1) - r$ for some $0 \leq r < 1$. If we fix q and S and view V as a function of r, then it is easy to see that it is monotonic; if we then compute $L = V_{r=1} = \frac{q^{-1}((q-1)S+1)-1}{q-1}$, then it follows that $L < V \leq S$.

We will now recharacterize V as an element of the range $(L, S]$ using a more convenient parameter $\delta = \frac{V-L}{S-L}$ such that $0 < \delta \leq 1$. We write V in terms of S, q, and δ as $V = L + \delta(S-L) = q^{-1}(\delta((q-1)S+1) + S - 1)$ and $W = \frac{q((q-1)V+1)-1}{q-1} = \delta((q-1)S+1) + S$. Then, we have $A = (S-V)(W-S) = \delta(1-\delta)q^{-1}((q-1)S+1)^2$.

Regard A as a function of δ and ask what value of δ maximizes it; the answer, easily derived by solving $\frac{d}{d\delta}(\delta(1-\delta)) = 0$, is $\delta = \frac{1}{2}$, and the second derivative there is negative, so the maximum possible value for A is $\frac{1}{2}(1-\frac{1}{2})q^{-1}((q-1)S+1)^2$, and therefore for any value of δ in $(0,1]$, we have $A \leq \frac{((q-1)S+1)^2}{4q}$. □

Lemma 11.3. *Let X be a general Morris approximate counter with parameter q ($1 < q \leq 2$) whose expected value is n, let Z be a general Morris approximate counter with parameter q whose expected value is m, and assume that X and Z are statistically independent. Let $\rho = \frac{1}{-2(q^2-4q+1)}$, and also assume that $\mathrm{Var}\left(f(X)\right) \leq \frac{q-1}{2}n(n-1) + \rho$ and that $\mathrm{Var}\left(f(Z)\right) \leq \frac{q-1}{2}m(m-1) + \rho$; then $\mathrm{Var}\left(f(X \oplus Z)\right) \leq \frac{q-1}{2}(n+m)((n+m)-1) + \rho$.*

Proof. If $Z = 0$, then $m = \mathbb{E}\left[f(Z)\right] = 0$, $w = 0$, $S = v$, $V = v$, and $\Delta = 0$; therefore $f(X \oplus Z) = f(X)$ exactly, and it follows that $\mathrm{Var}\left(f(X \oplus Z)\right) = \mathrm{Var}\left(f(X)\right) \leq \frac{q-1}{2}n(n-1) \leq \frac{q-1}{2}(n+m)(n+m-1) + \rho$, as desired. Similarly, if $X = 0$, then $n = \mathbb{E}\left[f(Z)\right] = 0$, $v = 0$, $S = w$, $V = w$, and $(1-\Delta) = 0$; therefore $f(X \oplus Z) = f(Z)$ exactly, so $\mathrm{Var}\left(f(X \oplus Z)\right) = \mathrm{Var}\left(f(Z)\right) \leq \frac{q-1}{2}m(m-1) \leq \frac{q-1}{2}(n+m)(n+m-1) + \rho$.

Now, suppose $X > 0$ and $Z > 0$ (therefore $n = \mathbb{E}\left[f(X)\right] \geq 1$ and $m = \mathbb{E}\left[f(Z)\right] \geq 1$):

$$
\begin{aligned}
&\mathrm{Var}\left(f(X \oplus Z)\right) \\
&\quad = \mathbb{E}\left[\left(f(X \oplus Z)\right)^2\right] - \left(\mathbb{E}\left[f(X \oplus Z)\right]\right)^2 \\
&\quad = \mathbb{E}[(1-\Delta)V^2 + \Delta W^2] - \mathbb{E}[S^2] \\
&\quad = \mathbb{E}\left[\frac{W-S}{W-V}V^2 + \frac{S-V}{W-V}W^2 - S^2\right] \\
&\quad = \mathbb{E}\left[(S-V)(W-S)\right] \\
&\quad \leq \mathbb{E}\left[\frac{((q-1)S+1)^2}{4q}\right] \qquad \text{[by Lemma 11.2]}
\end{aligned}
$$

$$
\begin{aligned}
&= \mathbb{E}\left[\frac{(q-1)^2S^2 + 2(q-1)S + 1}{4q}\right] \\
&= \mathbb{E}\left[\frac{(q-1)^2}{4q}(f(X)+f(Z))^2 + \frac{q-1}{2q}(f(X)+f(Z)) + \frac{1}{4q}\right] \\
&= \frac{(q-1)^2}{4q}\Big(\mathbb{E}\Big[(f(X))^2\Big] + \mathbb{E}[2f(X)f(Z)] + \mathbb{E}\Big[(f(Z))^2\Big]\Big) + \frac{q-1}{2q}\mathbb{E}[f(X)+f(Z)] + \frac{1}{4q} \\
&= \frac{(q-1)^2}{4q}\Big(\mathrm{Var}\,(f(X)) + (\mathbb{E}\,[f(X)])^2 + \mathbb{E}\,[2f(X)f(Z)] + \mathrm{Var}\,(f(Z)) + (\mathbb{E}\,[f(Z)])^2\Big) \\
&+ \frac{q-1}{2q}(\mathbb{E}[f(X)+f(Z)]) + \frac{1}{4q} \\
&= \frac{(q-1)^2}{4q}\Big(\mathrm{Var}\,(f(X)) + n^2 + 2nm + \mathrm{Var}\,(f(Z)) + m^2\Big) + \frac{q-1}{2q}(n+m) + \frac{1}{4q} \\
&= \frac{(q-1)^2}{4}\Big(\mathrm{Var}\,(f(X)) + \mathrm{Var}\,(f(Z)) + (n+m)^2\Big) + \frac{q-1}{2q}(n+m) + \frac{1}{4q} \\
&\le \frac{(q-1)^2}{4q}\Big(\frac{q-1}{2}n(n-1) + \rho + \frac{q-1}{2}m(m-1) + \rho + (n+m)^2\Big) + \frac{q-1}{2q}(n+m) + \frac{1}{4q} \\
&= \frac{(q-1)(q^2-1)}{8q}(n^2+m^2) + \frac{(q-1)^2}{4q}(2nm) - \frac{(q-1)(q+1)(q-3)}{8q}(n+m) + \frac{(q-1)^2}{2q}\rho + \frac{1}{4q} \\
&= \frac{(q-1)(q^2-1)}{8q}(n^2+m^2) + \frac{(q-1)^2}{4q}(2nm) - \frac{(q-1)(q+1)(q-3)}{8q}(n+m) + \rho.
\end{aligned}
$$

We wish to show that this last quantity is less than or equal to $\frac{q-1}{2}(n+m)(n+m-1)+\rho$; we do so by showing that $\frac{8q}{q-1}$ times their difference is nonpositive (note that $\frac{8q}{q-1} > 0$).

$$
\begin{aligned}
&\frac{8q}{q-1}\left(\frac{(q-1)(q^2-1)}{8q}(n^2+m^2) + \frac{(q-1)^2}{4q}(2nm) - \frac{(q-1)(q+1)(q-3)}{8q}(n+m) + \rho\right. \\
&\left.-\left(\frac{q-1}{2}(n+m)(n+m-1)+\rho\right)\right) \\
&= (q^2-4q-1)n^2 - (q^2-6q-3)n - 2(q+1) + (q^2-4q-1)m^2 - (q^2-6q-3)m - 2(q+1)nm.
\end{aligned}
$$

We now need only show that

$$(q^2-4q-1)n^2 - (q^2-6q-3)n - 2(q+1)nm \le 0$$

and

$$(q^2-4q-1)m^2 - (q^2-6q-3)m - 2(q+1)nm \le 0.$$

The two proofs are identical in form; here, we present just one of the proofs. Because $2(q+1)n > 0$ and $m \ge 1$,

$$
\begin{aligned}
&(q^2-4q-1)n^2 - (q^2-6q-3)n - 2(q+1)nm \\
&\le (q^2-4q-1)n^2 - (q^2-6q-3)n - (2q+2)n \\
&= (q^2-4q-1)n^2 - (q^2-4q-1)n \\
&= (q^2-4q-1)n(n-1) \le 0,
\end{aligned}
$$

because $n(n-1) \geq 0$ for all $n \geq 1$ and $q^2 - 4q - 1 < 0$ for all $1 < q \leq 2$ (the roots of $q^2 - 4q - 1 = 1$ are $2 - \sqrt{5} \approx -0.236\ldots$ and $2 + \sqrt{5} \approx 4.236\ldots$). □

THEOREM 11.4. *Let $1 < q \leq 2$, let $\rho = \frac{1}{-2(q^2-4q+1)}$, and let X be a general Morris approximate counter with parameter q whose expected value is n. Then,* $\mathrm{Var}\,(f(X)) \leq \frac{q-1}{2}n(n-1) + \rho$.

PROOF. We proceed by structural induction on the computation that created X.

Basis case: Suppose that X is a newly created counter. Then, its value is definitely 0, and therefore its expected value n is 0 and its variance is 0, and so $\mathrm{Var}\,(f(X)) = 0 = \frac{0(0-1)}{2} \leq \frac{n(n-1)}{2} + \rho$.

Inductive case 1 (incrementation): Suppose that X was produced by incrementing a counter Y having expected value $n-1$. By our inductive hypothesis, we have $\mathrm{Var}\,(f(Y)) \leq \frac{(n-1)(n-2)}{2} + \rho$, and therefore by Lemma 11.1 with $\kappa = \rho$, we have $\mathrm{Var}\,(f(X)) \leq \frac{n(n-1)}{2} + \rho$.

Inductive case 2 (addition): Suppose that X was produced by adding a counter Y with expected value y and a statistically independent counter Z with expected value z, such that $y + z = n$. By inductive hypothesis, we have $\mathrm{Var}\,(f(Y)) \leq \frac{q-1}{2}y(y-1) + \rho$ and $\mathrm{Var}\,(f(Z)) \leq \frac{q-1}{2}z(z-1) + \rho$, and therefore by Lemma 11.3, we have $\mathrm{Var}\,(f(X)) \leq \frac{q-1}{2}n(n-1) + \rho$.

Therefore, in all cases $\mathrm{Var}\,(f(X)) \leq \frac{q-1}{2}n(n-1) + \rho$. □

11.2 Proof of Bounded Variance for Binary Morris Counters

LEMMA 11.5. *Let X be a binary Morris approximate counter whose expected value is n, and Z be a statistically independent binary Morris approximate counter whose expected value is m. Without loss of generality assume $m \leq n$. Assume also that* $\mathrm{Var}\,(f(X)) \leq \frac{n(n-1)}{2}$*; then, we have* $\mathrm{Var}\,(f(X \oplus Z)) \leq \frac{(n+m)((n+m)-1)}{2}$*. As an alternative, assume also that $m \leq 1$ and* $\mathrm{Var}\,(f(X)) = \frac{n(n-1)}{2}$*; then* $\mathrm{Var}\,(f(X \oplus Z)) = \frac{(n+m)((n+m)-1)}{2}$.

PROOF. Let $\Delta = \frac{f(Z)}{f(X+1)-f(X)}$. Then:

$$
\begin{aligned}
&\mathrm{Var}\,(f(X \oplus Z)) \\
&= \mathbb{E}\left[(f(X \oplus Z))^2\right] - (\mathbb{E}[f(X \oplus Z)])^2 \\
&= \mathbb{E}\left[(1-\Delta)(f(X))^2 + \Delta(f(X+1))^2\right] - (n+m)^2 \\
&= \mathbb{E}\left[(f(X))^2 + \Delta\left((f(X+1))^2 - (f(X))^2\right)\right] - (n+m)^2 \\
&= \mathbb{E}\left[(f(X))^2\right] + \mathbb{E}[f(Z)(f(X+1) + f(X))] - (n+m)^2 \\
&= \left(\mathrm{Var}\,(f(X)) + (\mathbb{E}\,[f(X)])^2\right) + \mathbb{E}[f(Z)(3f(X)+1)] - (n+m)^2 \\
&\leq \frac{n(n-1)}{2} + n^2 + \mathbb{E}[f(Z)(3f(X)+1)] - (n+m)^2 \\
&= \frac{n(n-1)}{2} + n^2 + m(3n+1) - (n+m)^2 \\
&= \frac{n^2}{2} + nm - m^2 - \frac{n}{2} + m \\
&\leq \frac{n^2}{2} + nm - m^2 - \frac{n}{2} + m + \frac{3}{2}(m^2 - m) \\
&= \frac{(n+m)((n+m)-1)}{2}
\end{aligned}
$$

(because $\frac{3}{2}(m^2 - m) \geq 0$ for all integer $m \geq 0$), with equality holding when $m \leq 1$ and $\mathrm{Var}\,(f(X)) = \frac{n(n-1)}{2}$ (because we have $\frac{3}{2}(m^2 - m) = 0$ when $m = 0$ or $m = 1$). □

THEOREM 11.6. *Let X be a binary Morris approximate counter whose expected value is n. Then,* $\mathrm{Var}\,(f(X)) \leq \frac{n(n-1)}{2}$.

PROOF. We proceed by structural induction on the computation that created X.

Basis case: Suppose that X a newly created counter. Then, its value is definitely 0, and therefore its expected value is 0 and its variance is 0, and so $\mathrm{Var}\,(f(X)) = 0 = \frac{0(0-1)}{2} = \frac{n(n-1)}{2}$.

Inductive case 1 (incrementation): Let us suppose that X was produced by incrementing counter Y having expected value $n - 1$. By our inductive hypothesis, we have $\mathrm{Var}\,(f(Y)) \leq \frac{(n-1)(n-2)}{2}$, and so by Lemma 11.1 with $\kappa = 0$, we have $\mathrm{Var}\,(f(X)) \leq \frac{n(n-1)}{2}$.

Inductive case 2 (addition): Suppose that X was produced by adding a counter Y with expected value y and a statistically independent counter Z with expected value z, such that $z \leq y$ and $y + z = n$. By inductive hypothesis, we have $\mathrm{Var}\,(f(Y)) \leq \frac{y(y-1)}{2}$, and therefore by Lemma 11.5, we have $\mathrm{Var}\,(f(X)) \leq \frac{n(n-1)}{2}$.

Therefore, in all cases $\mathrm{Var}\,(f(X)) \leq \frac{n(n-1)}{2}$. □

11.3 Proof of Bounded Variance for General Csűrös Counters

LEMMA 11.7. *Given real $S \geq 0$, real $1 < q \leq 2$, and integer $M \geq 1$, let $\mu = \frac{M}{q-1}$, $d = \lfloor \log_q \frac{S+\mu}{\mu} \rfloor$, $\mathcal{V} = M\frac{q^d-1}{q-1}$, $\mathcal{W} = M\frac{q^{d+1}-1}{q-1}$, $\omega = \frac{\mathcal{W}-\mathcal{V}}{M}$, $j = \lfloor \frac{S-\mathcal{V}}{\omega} \rfloor$, $V = \mathcal{V} + j\omega$, $W = V + \omega$, and $A = (S - V)(W - S)$. Then, $A \leq \frac{((q-1)S+M)^2}{4M(M+(q-1))} = \frac{(S+\mu)^2}{4\mu(\mu+1)}$.*

PROOF. The interval $[\mathcal{V}, \mathcal{W}]$ is divided into M subintervals of width ω; subinterval j is the one that contains S. (If S falls right on the boundary between two subintervals, then we can regard it as being in either subinterval; then either $S = V$ or $S = W$, and so $(S - V)(W - S) = 0$.)

If S is in subinterval j ($0 \leq j < M$), then the smallest value $\mathcal{V}_{\min}$ that $\mathcal{V}$ could have (holding q, M, and j fixed while letting S vary) occurs when S is at the upper end of subinterval j, and similarly $\mathcal{W}_{\min} = q\mathcal{V}_{\min} + M$ and $\omega_{\min} = \frac{(q-1)\mathcal{V}_{\min}+M}{M}$. But where is $\mathcal{V}_{\min}$ with respect to S? If S is at the upper end of subinterval j, then $S - \mathcal{V}_{\min} = (j + 1)\omega_{\min}$ and $\mathcal{W}_{\min} - S = (q\mathcal{V}_{\min} + M) - S = (M - (j + 1))\omega_{\min}$ similarly. (Note that $\mathcal{W}$ and ω are minimized exactly when $\mathcal{V}$ is minimized, which is what justifies using the "min" subscripts on $\mathcal{W}_{\min}$ and $\omega_{\min}$.) Eliminating $\omega_{\min}$ gives us $(M - (j + 1))(S - \mathcal{V}_{\min}) = (j + 1)((q\mathcal{V}_{\min} + M) - S)$. Solving this for $\mathcal{V}_{\min}$ gives $\mathcal{V}_{\min} = \frac{M(S-(j+1))}{M+(q-1)(j+1)} = \frac{\mu(S-(j+1))}{\mu+(j+1)}$.

Now $V_{\min}$, the lowest possible value for V (which is the lower end of the subinterval containing S), is j subinterval-widths above $\mathcal{V}_{\min}$ and so $V_{\min} = (\mathcal{V}_{\min} + j\omega_{\min}) = \frac{((q-1)jS+M(S-1)}{M+(q-1)(j+1)} = \frac{(jS+\mu(S-1)}{\mu+(j+1)}$. (Note that $V_{\min}$ corresponds to L as used in the proof of Lemma 11.2.) We characterize V using a parameter δ as

$$V = ((1 - \delta)V_{\min} + \delta S) = \frac{\delta(M + (q - 1)S) + (q - 1)jS + M(S - 1)}{M + (q - 1)(j + 1)} = \frac{\delta(\mu + S) + jS + \mu(S - 1)}{\mu + (j + 1)}.$$

Now, we have a formula for V in terms of S and δ (and fixed q, M, and j); in exchange for introducing the parameter δ, we have gotten rid of those pesky floor functions.

Once V has been defined in this way and regarded as the actual lower end of subinterval j, we can calculate a corresponding $\mathcal{V}_V$ that is exactly j subinterval-widths below V by solving

$(V - \mathcal{V}_V)(M - j) = ((q\mathcal{V}_V + M) - V)j$ to get the result

$$\begin{aligned}\mathcal{V}_V &= \frac{M(\delta(M + (q-1)S) - (q-1)j^2 + j(q-1)(S-1) + MS - (j+1)M)}{(M + (q-1)j)(M + (q-1)(j+1))} \\ &= \frac{\mu(\delta(\mu + S) - j^2 + j(S-1) + \mu S - (j+1)\mu)}{(\mu + j)(\mu + (j+1))}.\end{aligned}$$

The width of each subinterval is $\omega_V = \frac{(q-1)\mathcal{V}_V + M}{M}$ and, therefore, we have $W = V + \omega_V = S + \delta\frac{M+(q-1)S}{M+(q-1)j} = S + \delta\frac{\mu+S}{\mu+j}$.

Then, $(S - V)(W - S) = \delta(1-\delta)\frac{((q-1)S+M)^2}{(M+(q-1)j)(M+(q-1)(j+1))}$. We now have $(S - V)(W - S)$ in terms of S and δ and q and M and j. Now, switch gears: hold S and q and M fixed, and allow δ and j to vary. Then, $(S - V)(W - S)$ is maximal when $\delta = \frac{1}{2}$ and $(\mu + j)(\mu + (j+1))$ is minimal. Now $(2\mu + 1)$ is positive for $1 < q \le 2$ and $M \ge 1$; therefore for $j \ge 0$, $(\mu + j)(\mu + (j+1)) = j^2 + (2\mu + 1)j + \mu(\mu + 1)$ is minimal when $j = 0$, and so $A = (S - V)(W - S) \le \frac{((q-1)S+M)^2}{4M(M+(q-1))} = \frac{(S+\mu)^2}{4\mu(\mu+1)}$. □

Lemma 11.8. *Let X be a Csűrös approximate counter with integer parameter M ($M \ge 1$) and real parameter q ($1 < q \le 2$) whose expected value is n, let Z be a Csűrös approximate counter with the same parameters M and q whose expected value is m, and assume that X and Z are statistically independent. Let $\mu = \frac{M}{q-1}$ and $\rho = \frac{\mu^2}{4\mu^2+4\mu-2}$, and assume that* $\mathrm{Var}\,(f(X)) \le \frac{1}{2\mu}n(n-1) + \rho$ *and that* $\mathrm{Var}\,(f(Z)) \le \frac{1}{2\mu}m(m-1) + \rho$*; then,* $\mathrm{Var}\,(f(X \oplus Z)) \le \frac{1}{2\mu}(n+m)((n+m)-1) + \rho$.

Proof. As in the proof of Lemma 11.3, if $Z = 0$, then $m = \mathbb{E}\,[f(Z)] = 0$, $w = 0$, $S = v$, $V = v$, and $\Delta = 0$; therefore, $f(X \oplus Z) = f(X)$, so $\mathrm{Var}\,(f(X \oplus Z)) = \mathrm{Var}\,(f(X)) \le \frac{1}{2\mu}n(n-1) \le \frac{1}{2\mu}(n + m)(n + m - 1) + \rho$, as desired. In the same way, if $X = 0$, then $n = \mathbb{E}\,[f(Z)] = 0$, $v = 0$, $S = w$, $V = w$, and $(1 - \Delta) = 0$; therefore, $f(X \oplus Z) = f(Z)$, so $\mathrm{Var}\,(f(X \oplus Z)) = \mathrm{Var}\,(f(Z)) \le \frac{1}{2\mu}m(m-1) \le \frac{1}{2\mu}(n+m)(n+m-1) + \rho$.

Now, suppose $X > 0$ and $Z > 0$ (therefore, $n = \mathbb{E}\,[f(X)] \ge 1$ and $m = \mathbb{E}\,[f(Z)] \ge 1$):

$$\begin{aligned}&\mathrm{Var}\,(f(X \oplus Z)) \\ &= \mathbb{E}\,[(S - V)(W - S)] && \text{[as in proof of Lemma 11.3]} \\ &\le \mathbb{E}\left[\frac{(S+\mu)^2}{4\mu(\mu+1)}\right] && \text{[by Lemma 11.7]} \\ &= \mathbb{E}\left[\frac{S^2 + 2\mu S + \mu^2}{4\mu(\mu+1)}\right] \\ &= \frac{1}{4\mu(\mu+1)}\left(\mathbb{E}\left[(f(X))^2\right] + \mathbb{E}[2f(X)f(Z)] + \mathbb{E}\left[(f(Z))^2\right]\right) + \frac{1}{2(\mu+1)}\mathbb{E}[f(X) + f(Z)] + \frac{\mu}{4(\mu+1)} \\ &= \frac{1}{4\mu(\mu+1)}\left(\mathrm{Var}(f(X)) + (\mathbb{E}[f(X)])^2 + \mathbb{E}[2f(X)f(Z)] + \mathrm{Var}(f(Z)) + (\mathbb{E}[f(Z)])^2\right) \\ &\quad + \frac{1}{2(\mu+1)}\left(\mathbb{E}[f(X) + f(Z)]\right) + \frac{\mu}{4(\mu+1)} \\ &= \frac{1}{4\mu(\mu+1)}\left(\mathrm{Var}(f(X)) + n^2 + 2nm + \mathrm{Var}(f(Z)) + m^2\right) + \frac{1}{2(\mu+1)}(n+m) + \frac{\mu}{4(\mu+1)}\end{aligned}$$

$$
\begin{aligned}
&= \frac{1}{4\mu(\mu+1)}\left(\mathrm{Var}\left(f(X)\right) + \mathrm{Var}\left(f(Z)\right) + (n+m)^2\right) + \frac{1}{2(\mu+1)}(n+m) + \frac{\mu}{4(\mu+1)} \\
&\le \frac{1}{4\mu(\mu+1)}\left(\frac{n(n-1)}{2\mu} + \rho + \frac{m(m-1)}{2\mu} + \rho + (n+m)^2\right) + \frac{1}{2(\mu+1)}(n+m) + \frac{\mu}{4(\mu+1)} \\
&= \frac{1}{4\mu(\mu+1)}\left(\frac{1}{2\mu}+1\right)(n^2+m^2) + \frac{1}{4\mu(\mu+1)}(2nm) + \left(\frac{1}{2(\mu+1)} - \frac{1}{4\mu^2(\mu+1)^2}\right)(n+m) \\
&\quad + \frac{1}{2\mu(\mu+1)}\rho + \frac{\mu}{4(\mu+1)} \\
&= \frac{2\mu+1}{8\mu^2(\mu+1)^2}(n^2+m^2) + \frac{1}{4\mu(\mu+1)}(2nm) + \frac{4\mu^2-1}{8\mu^2(\mu+1)^2}(n+m) + \rho.
\end{aligned}
$$

We wish to show that this last quantity is less than or equal to $\frac{1}{2\mu}(n+m)(n+m-1)+\rho$, by showing that $8\mu^2(\mu+1)^2$ times their difference is nonpositive (note that for $1 < q \le 2$, $\mu = \frac{M}{q-1} \ge M \ge 1$ and that $8\mu^2(\mu+1)^2 > 0$):

$$
\begin{aligned}
&8\mu^2(\mu+1)^2\left(\frac{2\mu+1}{8\mu^2(\mu+1)^2}(n^2+m^2) + \frac{1}{4\mu(\mu+1)}(2nm) + \frac{4\mu^2-1}{8\mu^2(\mu+1)^2}(n+m) + \rho\right. \\
&\qquad \left. -\left(\frac{1}{2\mu}(n+m)(n+m-1)+\rho\right)\right) \\
&= (-4\mu^2-2\mu+1)(n^2+m^2) + (8\mu^2+4\mu-1)(n+m) + (-4\mu^2-2\mu)(2nm).
\end{aligned}
$$

As in the proof of Lemma 11.3, we will exhibit only a proof that

$$(-4\mu^2-2\mu+1)n^2 + (8\mu^2+4\mu-1)n + (-4\mu^2-2\mu)nm \le 0.$$

Because $(-4\mu^2-2\mu) < 0$ and $m \ge 1$,

$$
\begin{aligned}
&(-4\mu^2-2\mu+1)n^2 + (8\mu^2+4\mu-1)n + (-4\mu^2-2\mu)nm \\
&\le (-4\mu^2-2\mu+1)n^2 + (8\mu^2+4\mu-1)n + (-4\mu^2-2\mu)n \\
&= (-4\mu^2-2\mu+1)n^2 + (4\mu^2+2\mu-1)n \\
&= (-4\mu^2-2\mu+1)n(n-1) \le 0,
\end{aligned}
$$

because $n(n-1) \ge 0$ for all $n \ge 1$ and $(-4\mu^2-2\mu+1) < 0$ for all $\mu \ge 1$. □

Lemma 11.9. *Let X be a Csűrös approximate counter with integer parameter M ($M \ge 1$) and real parameter q ($1 < q \le 2$) whose expected value is n, let X' be the result of applying an increment operation to that counter, and let $\mu = \frac{M}{q-1}$ and $\rho = \frac{\mu^2}{4\mu^2+4\mu-2}$. Assume* $\mathrm{Var}\left(f(X)\right) \le \frac{1}{M(M+1)}n(n-1) + \rho$*; then,* $\mathrm{Var}\left(f(X')\right) \le \frac{1}{M(M+1)}(n+1)n + \rho$.

Proof. Analysis of the *increment* and *add* algorithms presented in Section 6 shows that *increment*(X) is equivalent in its behavior to *add*$(X, 1)$. Moreover, the variance of a Csűrös counter with expected value 1 is 0. Therefore, we can apply Lemma 11.8. □

Theorem 11.10. *Let X be a Csűrös approximate counter with integer parameter M ($M \ge 1$) and real parameter q ($1 < q \le 2$) whose expected value is n, and let $\mu = \frac{M}{q-1}$ and $\rho = \frac{\mu^2}{4\mu^2+4\mu-2}$. Then,* $\mathrm{Var}\left(f(X)\right) \le \frac{1}{2\mu}n(n-1) + \rho$.

Proof. We proceed by structural induction on the computation that created X.

Basis case: Suppose that X a newly created counter. Then, its value is definitely 0, and therefore its expected value is 0 and its variance is 0, and so $\text{Var}(f(X)) = 0 \leq \frac{1}{2\mu}0(0-1) + \rho = \frac{1}{2\mu}n(n-1) + \rho$.

Inductive case 1 (incrementation): Suppose that X was produced by incrementing a counter Y having expected value $n-1$. By inductive hypothesis, we have $\text{Var}(f(Y)) \leq \frac{1}{2\mu}y(y-1) + \rho$, and therefore by Lemma 11.9, we have $\text{Var}(f(X)) \leq \frac{1}{2\mu}n(n-1) + \rho$.

Inductive case 2 (addition): Suppose that X was produced by adding a counter Y with expected value y and a counter Z with expected value z, such that $z \leq y$ and $y + z = n$. By inductive hypothesis, we have $\text{Var}(f(Y)) \leq \frac{1}{2\mu}y(y-1) + \rho$ and $\text{Var}(f(Z)) \leq \frac{1}{2\mu}z(z-1) + \rho$, and therefore by Lemma 11.8, we have $\text{Var}(f(X)) \leq \frac{1}{2\mu}n(n-1) + \rho$.

Therefore, in all cases $\text{Var}(f(X)) \leq \frac{1}{2\mu}n(n-1) + \rho$. □

Note that $\frac{1}{6} \leq \rho < \frac{1}{4}$, so the bound on the variance of Csűrös counters with addition, no matter what the values of M and q, is just as reasonably tight as the bound on the variance of Morris counters.

12 MEASUREMENTS OF TWO APPLICATIONS

We implemented a distributed version of the LDA Gibbs topic-modeling algorithm described by Tristan et al. (2015). The algorithm is presented in Figure 5.

The Gibbs sampler for LDA has the following parameters: I is the number of iterations to perform, M is the number of documents, V is the size of the vocabulary, K is the number of topics, $N[M]$ is an integer array of size M that describes the shape of w, α is a parameter that controls how concentrated the distributions of topics per documents should be, β is a parameter that controls how concentrated the distributions of words per topics should be, $w[M][N]$ is ragged array containing the document data (where subarray $w[m]$ has length $N[m]$), $\theta[M][K]$ is an $M \times K$ matrix where $\theta[m][k]$ is the probability of topic k in document m, and $\phi[V][K]$ is a $V \times K$ matrix where $\phi[v][k]$ is the probability of word v in topic k. Each element $w[m][n]$ is a nonnegative integer less than V, indicating which word in the vocabulary is at position n in document m. The matrices θ and ϕ are typically initialized by the caller to randomly chosen distributions of topics for each document and words for each topic; these same arrays serve to deliver "improved" distributions back to the caller.

The algorithm uses three local data structures to store various statistics about the model (lines 2–4): $tpd[M][K]$ is an $M \times K$ matrix where $tpd[m][k]$ is the number of times topic k is used in document m, $wpt[V][K]$ is an $V \times K$ matrix where $wpt[v][k]$ is the number of times word v is assigned to topic k, and $wt[K]$ is an array of size K where $wt[k]$ is the total number of time topic k is in use.

The algorithm works by iterating over the dataset I times (loop starting on line 5 and ending on line 33). An iteration starts by clearing the local data structures to zero (lines 6–8). Each iteration is composed of two phases. In the first phase (lines 10–21), we assume that the values of θ and ϕ are fixed, and to every word occurrence $w[m][n]$ in the document data we assign a new topic, randomly chosen according to a distribution computed from θ and ϕ, and tally these choices in wpt, tpd, and wt. In the second phase (lines 23–32), we assume that the statistics wpt, tpd, and wt are fixed, and we compute new values for θ and ϕ; each new value is the mean of a Dirichlet distribution induced by the statistics. We now review these two phases in detail.

In phase 1, we iterate over all the documents m (line 10) and all the words n in each document (line 11). To each word, we need to associate a topic. A topic is chosen randomly according to the following formula: the probability of associating a word $w[m][n]$ to topic k is proportional to the probability $\theta[m][k]$ that topic k is associated to document m times the probability $\phi[w[m][n]][k]$

```
 1: procedure LDA_Gibbs(int I, int M, int V, int K, int N[M], float α, float β,
                        int w[M][N], var float θ[M][K], var float φ[V][K])
 2:   local array int tpd[M][K]
 3:   local array int wpt[V][K]
 4:   local array int wt[K]
 5:   for i from 0 through I − 1 do
 6:     clear array tpd                                  ▷ Set every element to 0
 7:     clear array wpt                                  ▷ Set every element to 0
 8:     clear array wt                                   ▷ Set every element to 0
 9:     ▷ Phase 1: tally statistics by sampling distributions computed from θ and φ
10:     for all 0 ≤ m < M do
11:       for all 0 ≤ n < N do
12:         local array float p[K]
13:         for all 0 ≤ k < K do
14:           p[k] ← θ[m][k] × φ[w[m][n]][k]
15:         end for
16:         let z ← sample(p)                            ▷ Now 0 ≤ z < K
17:         increment tpd[m][z]                          ▷ Increment counters
18:         increment wpt[w[m][n]][z]                    ▷   (which may be integer
19:         increment wt[z]                              ▷   or approximate)
20:       end for
21:     end for
22:     ▷ Phase 2: Compute new θ and φ arrays from the tallied statistics
23:     for all 0 ≤ m < M do
24:       for all 0 ≤ k < K do
25:         θ[m][k] ← (tpd[m][k] + α)/(N[m] + K × α)
26:       end for
27:     end for
28:     for all 0 ≤ v < V do
29:       for all 0 ≤ k < K do
30:         φ[v][k] ← (wpt[v][k] + β)/(wt[k] + V × β)
31:       end for
32:     end for
33:   end for
```

Fig. 5. Pseudocode for the LDA Gibbs topic-modeling algorithm.

that vocabulary word word $w[n][m]$ is associated to topic k (lines 12–15). From the resulting array p of relative (unnormalized) probabilities, we sample a new topic z (line 16) and update the statistics *wpt*, *tpd*, and *wt* accordingly (lines 17–19).

In phase 2, for every document m, we compute the mean of the Dirichlet distribution induced by the per-document statistics $tpd[m]$ (lines 23–27). Then, for every vocabulary word v, we compute the mean of the Dirichlet distribution induced by the per-word statistics $wpt[v]$ (lines 28–32).

The algorithm was implemented on a cluster of four nodes (connected by 1Gb/s Ethernet), each with an Intel Core-i7 4820k CPU and two NVIDIA Titan Black GPU cards, for a total of 8 GPUs. The code is written in C++ and CUDA. We use MPI for internode communication. The behavior is quite similar to that of the single-GPU version reported by Tristan et al.: replacing integer counters with approximate counters does not affect the statistical performance of the algorithm, but allows the same hardware to process larger datasets. Moreover, the speed of the algorithm is markedly increased, largely because of a third effect: less data needs to be pushed through the

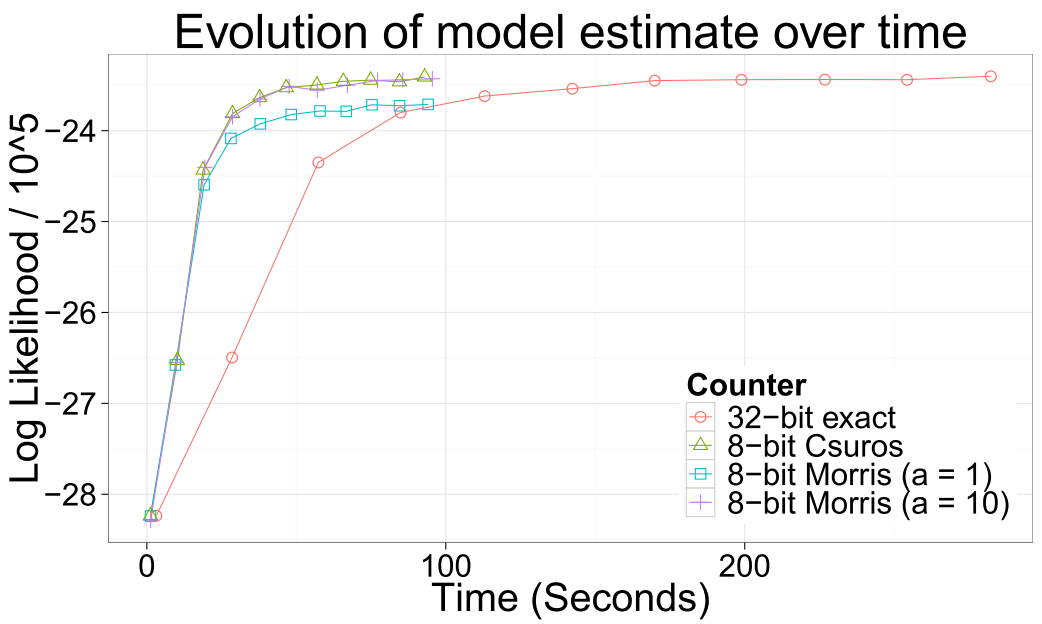

Fig. 6. 32-bit integer counters vs. 8-bit approximate counters used in a distributed (multiple-GPU) application (LDA Gibbs topic modeling) coded in CUDA using MPI, compared by log likelihood (*higher is better*).

network connecting the nodes. In one test, either 32-bit integer counters were used, or 8-bit approximate counters; using approximate counters improved the overall speed of each iteration of the distributed application by over 65% (see Figure 6, which shows measurements of time and log likelihood). Using 8-bit general Morris counters with $a = 10$ (i.e., $q = \frac{11}{10} = 1.1$) worked well, in that topics were produced using the same number of iterations as with 32-bit integer counters, without decreasing the log-likelihood measure of statistical performance. (Because the maximum counts for this particular dataset are not large, we could have used an even smaller value of q, say 1.03, to achieve smaller variance, but we found that $q = 1.1$ provided sufficient statistical quality and moreover was sufficient to handle the much larger counts, up to 2^{32} and beyond, of larger datasets.) Also effective were 8-bit binary Csűrös counters with $s = 5$. However, using 8-bit binary Morris counters ($a = 1$) caused the algorithm to converge to a smaller (less desirable) value of log likelihood; we conjecture that this occurs because an 8-bit binary Morris counter has huge dynamic range, much of which is wasted, so the part of the range that is actually used is too coarse to be fully effective (put another way, the variance is too large).

We have also tested a distributed implementation of a stochastic cellular automaton (SCA) for topic modeling as described by Zaheer et al. (2015, 2016). The algorithm is presented in Figure 7.

The SCA algorithm is best understood in comparison with the Gibbs sampler we just detailed. Like the Gibbs algorithm, the SCA algorithm iterates over the data to compute statistics for the topics (loop starting on line 9 and ending on line 31). However, SCA does not explicitly represent the entire probability matrices θ and ϕ. Since these matrices are the output of the algorithm, they need to be computed in a post-processing phase that follows the iterative phase (lines 33–42). In this post-processing phase, we compute the θ and ϕ distributions as the means of Dirichlet distributions induced by the statistics.

In the iterative phase of SCA, the values of θ and ϕ, which are necessary to compute the topic proportions, are computed on the fly (lines 20 and 21). Unlike the Gibbs algorithm, where on each

```
1:  procedure SCA(int I, int M, int V, int K, int N[M], float α, float β, int w[M][N])
2:    local array int tpd[M][K]
3:    local array int wpt[V][K]
4:    local array int wt[K]
5:    initialize array tpd[0]                          ▷ Randomly chosen distributions
6:    initialize array wpt[0]                          ▷ Randomly chosen distributions
7:    initialize array wt[0]                           ▷ Randomly chosen distributions
8:    ▷ The main iteration
9:    for i from 0 through (I ÷ 2) − 1 do
10:     for r from 0 through 1 do
11:       clear array tpd[1 − r]                       ▷ Set every element to 0
12:       clear array wpt[1 − r]                       ▷ Set every element to 0
13:       clear array wt[1 − r]                        ▷ Set every element to 0
14:       ▷ Compute new statistics by sampling distributions
15:       ▷   that are computed from old statistics
16:       for all 0 ≤ m < M do
17:         for all 0 ≤ n < N do
18:           local array float p[K]
19:           for all 0 ≤ k < K do
20:             let θ ← (tpd[r][m][k] + α)/(N[m] + K × α)
21:             let φ ← (wpt[r][v][k] + β)/(wt[r][k] + V × β)
22:             p[k] ← θ × φ
23:           end for
24:           let z ← sample(p)                        ▷ Now 0 ≤ z < K
25:           increment tpd[1 − r][m][z]               ▷ Increment counters
26:           increment wpt[1 − r][w[m][n]][z]         ▷   (which may be integer
27:           increment wt[1 − r][z]                   ▷   or approximate)
28:         end for
29:       end for
30:     end for
31:   end for
32:   ▷ Final output pass
33:   for all 0 ≤ m < M do
34:     for all 0 ≤ k < K do
35:       write θ[m][k] as (tpd[0][m][k] + α)/(N[m] + K × α)
36:     end for
37:   end for
38:   for all 0 ≤ v < V do
39:     for all 0 ≤ k < K do
40:       write φ[v][k] as (wpt[0][v][k] + β)/(wt[0][k] + V × β)
41:     end for
42:   end for
```

Fig. 7. Pseudocode for the Stochastic Cellular Automaton algorithm.

iteration we have a back-and-forth between two phases, where one reads θ and ϕ to update the statistics and the other reads the statistics to update θ and ϕ, SCA performs the back-and-forth between two copies of the statistics. Therefore, the number of iterations is halved (line 9), and each iteration has two subiterations (line 10), one that reads *tpd*[0], *wpt*[0], and *wt*[0] to write *tpd*[1], *wpt*[1], and *wt*[1], then one that reads *tpd*[1], *wpt*[1], and *wt*[1] to write *tpd*[0], *wpt*[0], and *wt*[0]. One key advantage of this algorithm over the Gibbs algorithm is that *all* the in-memory arrays are counters, so using approximate counters leads to an even smaller memory footprint and even better memory bandwidth usage than for LDA Gibbs.

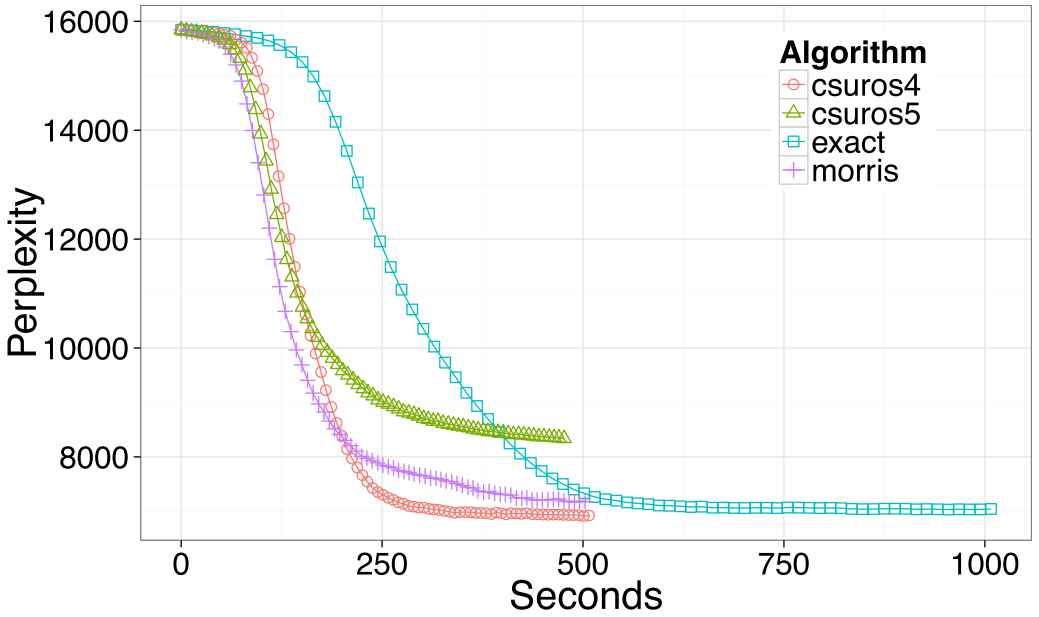

Fig. 8. 32-bit integer counters vs. 8-bit approximate counters used in a distributed (multiple-CPU) application (SCA topic modeling) coded in Java using MPI, compared by perplexity (*lower is better*).

We tested a version of the SCA algorithm implemented in the Java programming language. To achieve good performance, we use only arrays of primitive types and pre-allocate all arrays before learning starts. We implement multithreaded parallelization within a node using the work-stealing Fork/Join framework (tasks are recursively subdivided until the number of data points to be processed is less than 10^5). Internode communication uses the Java binding to a version of OpenMPI 1.8.7 that we modified to support the use of approximate-counter addition operations as arguments to the `allReduce` method (see Section 13). We use a sparse array representation for the counts of topics per document θ and use Walker's alias method (Walker 1974) to draw from discrete distributions. We run our experiments on a small cluster of 16 nodes (connected by 10Gb/s Ethernet), each with two 8-core Intel Xeon E5 processors (some nodes have Ivy Bridge processors while others have Sandy bridge processors) for a total of 32 hardware threads per node and 256GB of memory. We run 32 JVM instances (one per Xeon socket), assigning each one 20GB of memory. The number of topics (K) is 100, the number of training iterations is 75, and $\alpha = \beta = 0.1$. We use two datasets, both of which are cleaned by removing stop words and rare words: we use the English Wikipedia dataset for training and the Reuters RCV1 dataset for testing. Our Wikipedia dataset has 6,749,797 documents comprising 6,749,797 tokens with a vocabulary of 291,561 words. Our Reuters dataset has 806,791 documents comprising 105,989,213 tokens with a vocabulary of 43,962 words.

Time and perplexity measurements of the SCA algorithm are shown in Figure 8. Versions that use 8-bit approximate counters (Morris with $q = 1.08$, and Csűrös with either $s = 4$ or $s = 5$) are approximately twice as fast as the baseline that uses 32-bit integer counters. Csűrös counters with $s = 4$ achieved the best (lowest) perplexity in these tests. Csűrös counters with $s = 5$ do not have enough range for the application, and we observe that its asymptotic perplexity score is markedly worse.

```
#define APPROX_ADD(name,kind,s,counter,value,random,                    \
                   COMPUTE_S,COMPUTE_K,COMPUTE_V,COMPUTE_G)             \
    counter##_t name##kind(counter##_t x, counter##_t y) {              \
        COMPUTE_S(name,s,counter,value);                                \
        COMPUTE_K(name,s,counter,value);                                \
        COMPUTE_V(name,s,counter,value);                                \
        COMPUTE_G(name,s,counter,value,random);                         \
        if (((next_random_##random() * G) < (S - V))                    \
            && (K < (counter##_t)(-1))) { ++K; }                        \
        return K;                                                       \
    }
```

Fig. 9. C macro template code for defining addition functions on approximate counters.

13 MODIFYING OPENMPI TO SUPPORT ADDITION OF APPROXIMATE COUNTERS

In an effort to get the best possible speed on the stochastic cellular automaton application described in Section 12, we modified the source code of OpenMPI 1.8.7 (obtained from `https://www.open-mpi.org`) to provide operations that could be used with the `allReduce` method to combine arrays of approximate counters by adding them elementwise using approximate-counter addition. To test the relative speeds of various implementation strategies, we used C macros to systematically produce 264 distinct implementations of approximate-counter addition. Most of the modifications were made in the files `ompi/op/op.c` and `ompi/mca/op/base/op_base_functions.c`; we also found it necessary to extend certain other tables (especially in the Java code that provides the Java binding to OpenMPI) and to arrange for the MPI initialization code to precompute additional tables.

We implemented addition operations for seven distinct 8-bit representations of approximate counters: binary Csűrös counters for $s = 3$, $s = 4$, $s = 5$, $s = 6$, $s = 7$, and $s = 8$, and general Morris counters with $q = 1.08$. For each of the Csűrös counters, we chose the smallest possible integer type for representing estimated values in the intermediate calculations, or the smallest possible floating-point type if no integer type would suffice. For the general Morris counters, we implemented two versions, one using type `float` and one using type `double` for estimated values. We also needed to decide whether to use random numbers of type `float` or type `double`. The resulting eight combinations that we used may be seen in the header lines of Table 2 (see page 36. Each of these combinations was then provided multiple implementations.

Each of the 264 implementations was generated by the C macro `APPROX_ADD` shown in Figure 9. The basic idea is that addition of two approximate counters, as described in Section 3, given two counter representation values `x` and `y`, consists of five steps: computing S, computing K, computing V, computing $G = W - V$, and finally deciding whether to add 1 to K before returning it. We always implement the last step in the same way, but each of the first four steps may be carried out in more than one way; therefore, `APPROX_ADD` has four parameters `COMPUTE_S`, `COMPUTE_K`, `COMPUTE_V`, and `COMPUTE_G`. For each of these, the actual argument text will be the name of another C macro. The parameter `name` will be replaced by one of eight identifiers, indicating which of the eight operations is being implemented; the parameter `kind` is another identifier that (redundantly) encodes which actual C macros are to be used for `COMPUTE_S`, `COMPUTE_K`, `COMPUTE_V`, and `COMPUTE_G`. The parameter `s` is the s value for a Csűrös counter and is not used for a Morris counter. Each of the three parameters `counter`, `value`, and `random` indicates a type (one of `uint8`, `uint16`, `uint32`, `uint64`, `float`, `double`) to be used for the representation of respectively the approximate

```
APPROX_ADD(approx8add3,CBACC,3,uint8,uint16,float,                          \
           S_CSUROS_SUM,K_BINARY_SEARCH_CSUROS,V_CSUROS,G_FROM_W_CSUROS)
APPROX_ADD(approx8add4,CBACC,4,uint8,uint32,double,                         \
           S_CSUROS_SUM,K_BINARY_SEARCH_CSUROS,V_CSUROS,G_FROM_W_CSUROS)
APPROX_ADD(approx8add5,CBACC,5,uint8,uint64,double,                         \
           S_CSUROS_SUM,K_BINARY_SEARCH_CSUROS,V_CSUROS,G_FROM_W_CSUROS)
APPROX_ADD(approx8add6,CBACC,6,uint8,float,float,                           \
           S_CSUROS_SUM,K_BINARY_SEARCH_CSUROS,V_CSUROS,G_FROM_W_CSUROS)
APPROX_ADD(approx8add7,CBACC,7,uint8,double,double,                         \
           S_CSUROS_SUM,K_BINARY_SEARCH_CSUROS,V_CSUROS,G_FROM_W_CSUROS)
APPROX_ADD(approx8add8,CBACC,8,uint8,double,double,                         \
           S_CSUROS_SUM,K_BINARY_SEARCH_CSUROS,V_CSUROS,G_FROM_W_CSUROS)

                                    ⋮

APPROX_ADD(approx8add3,TMTW,3,uint8,uint16,float,                           \
           S_TABLE_SUM,K_MATRIX_LOOKUP,V_TABLE,G_FROM_W_TABLE)
APPROX_ADD(approx8add4,TMTW,4,uint8,uint32,double,                          \
           S_TABLE_SUM,K_MATRIX_LOOKUP,V_TABLE,G_FROM_W_TABLE)
APPROX_ADD(approx8add5,TMTW,5,uint8,uint64,double,                          \
           S_TABLE_SUM,K_MATRIX_LOOKUP,V_TABLE,G_FROM_W_TABLE)
APPROX_ADD(approx8add6,TMTW,6,uint8,float,float,                            \
           S_TABLE_SUM,K_MATRIX_LOOKUP,V_TABLE,G_FROM_W_TABLE)
APPROX_ADD(approx8add7,TMTW,7,uint8,double,double,                          \
           S_TABLE_SUM,K_MATRIX_LOOKUP,V_TABLE,G_FROM_W_TABLE)
APPROX_ADD(approx8add8,TMTW,8,uint8,double,double,                          \
           S_TABLE_SUM,K_MATRIX_LOOKUP,V_TABLE,G_FROM_W_TABLE)
APPROX_ADD(approx8addg,TMTW,(%unused%),uint8,double,double,                 \
           S_TABLE_SUM,K_MATRIX_LOOKUP,V_TABLE,G_FROM_W_TABLE)
APPROX_ADD(approx8addf,TMTW,(%unused%),uint8,float,float,                   \
           S_TABLE_SUM,K_MATRIX_LOOKUP,V_TABLE,G_FROM_W_TABLE)
```

Fig. 10. C macro invocations for defining addition functions on approximate counters (14 of 264 shown).

counters, their estimated values, and generated pseudorandom numbers. Figure 10 shows the first six and the last eight of the 264 specific invocations of the `APPROX_ADD` macro that we used in our experiments. (In all 264 cases, the `counter` type was `uint8`; the `APPROX_ADD` macro was designed also to support implementation of 16-bit approximate counters, but we have not yet investigated such cases.)

Figure 11 shows C macros that define names for some useful constants, namely sentinel values (of type `float` and `double`) that are much larger than any actual table entries, and the constant 1 of various data types (especially useful as left-hand operand of the C left-shift operator `<<`).

Figure 12 shows C macros that, given a representation value for a Csűrös counter and the value of s for that counter representation, computes the estimated value of the counter. Five versions are provided, one for each of the possible types `uint16`, `uint32`, `uint64`, `float`, `double` that might be used for representing the estimated value.

Figure 13 shows C macros that, given a representation value `x` for a Csűrös counter and its corresponding estimated value `v` and the value of s for that counter representation, computes the estimated value of the counter representation value `x+1`. This can be done more cheaply than

```
#define APPROX_ADD_SENTINEL_float ((float)(ldexp(1.0, 70)))

#define APPROX_ADD_SENTINEL_double (ldexp(1.0, 500))

#define ONE16 ((uint16_t) 1)

#define ONE32 ((uint32_t) 1)

#define ONE64 ((uint64_t) 1)
```

Fig. 11. C macros for certain useful constants.

```
#define INTERPRET_BINARY_CSUROS_uint16(x,s)                                  \
    ((uint16_t)(INTERPRET_BINARY_CSUROS_uint32(x,s)))

#define INTERPRET_BINARY_CSUROS_uint32(x,s)                                  \
    ((((ONE32 << (s)) + ((x) & ((ONE32 << (s)) - ONE32))) << ((x) >> (s)))\
     - (ONE32 << (s)))

#define INTERPRET_BINARY_CSUROS_uint64(x,s)                                  \
    ((((ONE64 << (s)) + ((x) & ((ONE64 << (s)) - ONE64))) << ((x) >> (s)))\
     - (ONE64 << (s)))

#define INTERPRET_BINARY_CSUROS_double(x,s)                                  \
    (ldexp((ONE64 << (s)) + ((x) & ((ONE64 << (s)) - ONE64)), (x) >> (s)) \
     - (ONE64 << (s)))

#define INTERPRET_BINARY_CSUROS_float(x,s)                                   \
    ((float_t)(ldexp((ONE64 << (s)) + ((x) & ((ONE64 << (s)) - ONE64)),   \
                     (x) >> (s))                                         \
               - (ONE64 << (s))))
```

Fig. 12. C macros for calculating the estimated value from a Csűrös counter representation x.

```
#define INCREMENTALLY_UPDATE_BINARY_CSUROS_uint16(x,v,s)                     \
    ((v) + (ONE16 << ((x) >> (s))))

#define INCREMENTALLY_UPDATE_BINARY_CSUROS_uint32(x,v,s)                     \
    ((v) + (ONE32 << ((x) >> (s))))

#define INCREMENTALLY_UPDATE_BINARY_CSUROS_uint64(x,v,s)                     \
    ((v) + (ONE64 << ((x) >> (s))))

#define INCREMENTALLY_UPDATE_BINARY_CSUROS_double(x,v,s)                     \
    ((v) + ldexp(1.0, ((x) >> (s))))

#define INCREMENTALLY_UPDATE_BINARY_CSUROS_float(x,v,s)                      \
    ((v) + (float_t)ldexp(1.0, ((x) >> (s))))
```

Fig. 13. C macros for calculating the estimated value for x + 1 given x and its estimated value v.

```
#define S_CSUROS_SUM(name,s,counter,value)                              \
    value##_t S = (INTERPRET_BINARY_CSUROS_##value(x,s) +               \
                   INTERPRET_BINARY_CSUROS_##value(y,s));
```

(C) Compute S by adding Csűrös values calculated from x and y

```
#define S_TABLE_SUM(name,s,counter,value)                               \
    value##_t S = (name##_interpret_##value[x] +                        \
                   name##_interpret_##value[y]);
```

(T) Compute S by adding value-table entries indexed by x and y

Fig. 14. C macros for calculating S from x and y.

computing the estimated value for x+1 from scratch. Again five versions are provided, one for each of the possible types uint16, uint32, uint64, float, double that might be used for representing the estimated value.

Figure 14 shows two different ways of computing S. The C macro S_CSUROS_SUM (also identified in abbreviated contexts, such as in Table 2, by the letter C) uses the relevant estimate-calculation macro from Figure 12 twice (once on x and once on y) and adds the results. The C macro S_TABLE_SUM (also identified by the letter T) instead assumes there is a precomputed table, whose name is constructed by the phrase name##_interpret_##value, which can be indexed by x and then by y to fetch two values to be added. The choice between these two macros for computing S reflects a potential space-time tradeoff that we wished to measure.

Figures 15 and 16 together show seven different ways of computing K. The C macro K_CSUROS_CALCULATE (also identified by C) in Figure 15 uses the relevant type-specific C macro (also shown in that same figure) to calculate K from S by using logarithms (computed using __builtin_clz if S has an integer representation, or using frexpf or frexp if S has a floating-point representation). Five of the six alternative C macros in Figure 16 use another approach, starting from the larger of x and y and searching upward to find the appropriate value of K; the last alternative instead uses a large two-dimensional table. The choices among the seven macros for computing K reflect potential space-time tradeoffs that we wished to measure.

The C macro K_LINEAR_SEARCH_CSUROS (also identified by LA) sets K equal to the larger of x and y and then repeatedly increments K until the calculated estimated value for $K + 1$ is not larger than S. (This technique depends on the fact that K is represented using a sufficiently large data type that the calculation of K cannot overflow.)

The C macro K_LINEAR_INCR_CSUROS (also identified by LI) likewise sets K equal to the larger of x and y and then repeatedly increments K until the estimated value for $K + 1$ is not larger than S, but it calculates estimated values using the incremental technique of the macros in Figure 13.

The C macro K_BINARY_SEARCH_CSUROS (also identified by BA) sets K equal to the larger of x and y and then repeatedly adds 8 to K until the calculated estimated value for $K + 8$ is not larger than S; then, it successively tries increments of 4, 2, and 1. The next effect is to search linearly for an appropriate block of 15 values, then perform binary search on that block. Our intuition was that in practice the loop would nearly always perform at most one iteration, so this technique would nearly always behave as a straight binary search; the question was, would it beat a linear search? (This technique depends on the fact that K is represented using a sufficiently large data type that the calculation of K cannot overflow.)

```
#define K_CSUROS_CALCULATE(name,s,counter,value)                          \
    K_CSUROS_CALCULATE_##value(s)
```

(C) Calculate K from S (by dispatching to a type-specific macro below)

```
#define K_CSUROS_CALCULATE_uint16(s)                                      \
    K_CSUROS_CALCULATE_uint32(s)
```

How to calculate K from an S value of type `uint16`

```
#define K_CSUROS_CALCULATE_uint32(s)                                      \
    uint32_t Sprime = S + (1 << s);                                       \
    /* Note: __builtin_clz operates on a 32-bit value */                  \
    uint32_t d = (31 - (1 << s)) - __builtin_clz(Sprime);                 \
    uint32_t K = (d << s) + (Sprime >> d) - (1 << s);
```

How to calculate K from an S value of type `uint32`

```
#define K_CSUROS_CALCULATE_uint64(s)                                      \
    uint64_t Sprime = S + (1 << s);                                       \
    /* Note: __builtin_clz operates on a 32-bit value */                  \
    uint32_t d = (Sprime >= (((uint64_t)1)<<32)) ?                        \
        (63 - (1 << s)) - __builtin_clz((uint32_t)(Sprime >> 32)) :       \
        (31 - (1 << s)) - __builtin_clz((uint32_t)Sprime);                \
    uint32_t K = (d << s) + ((uint32_t)(Sprime >> d)) - (1 << s);
```

How to calculate K from an S value of type `uint64`

```
#define K_CSUROS_CALCULATE_float(s)                                       \
    float_t Sprime = S + (1 << s);                                        \
    int d;                                                                \
    (void) frexpf(Sprime, &d);                                            \
    d -= ((1 << s) + 1);                                                  \
    uint32_t K = (d << s) + (uint32_t)ldexpf(Sprime, -d) - (1 << s);
```

How to calculate K from an S value of type `float`

```
#define K_CSUROS_CALCULATE_double(s)                                      \
    double_t Sprime = S + (1 << s);                                       \
    int d;                                                                \
    (void) frexp(Sprime, &d);                                             \
    d -= ((1 << s) + 1);                                                  \
    uint32_t K = (d << s) + (uint32_t)ldexp(Sprime, -d) - (1 << s);
```

How to calculate K from an S value of type `double`

Note: `__builtin_clz` takes a 32-bit integer and returns the number of leading zero bits in the binary representation of the operand. The definition of `K_CSUROS_CALCULATE_uint64` could be greatly simplified if a 64-bit version of this count-leading-zeroes operation were available.

Fig. 15. C macros for calculating K from either S or x and y (part 1 of 2).

```
#define K_LINEAR_SEARCH_CSUROS(name,s,counter,value)                          \
    uint32_t K = (x > y) ? x : y;                                             \
    while (S >= INTERPRET_BINARY_CSUROS_##value(K+1,s)) ++K;
```

(LA) Linear search of Absolutely calculated Csűrös values

```
#define K_LINEAR_INCR_CSUROS(name,s,counter,value)                            \
    uint32_t K = (x > y) ? x : y;                                             \
    value##_t Z = INTERPRET_BINARY_CSUROS_##value(K+1,s);                     \
    while ((S >= Z) && (K != (counter##_t)(-1))) {                            \
        ++K;                                                                  \
        Z = INCREMENTALLY_UPDATE_BINARY_CSUROS_##value(K,Z,s)                 \
    }
```

(LI) Linear search of Incrementally calculated Csűrös values

```
#define K_BINARY_SEARCH_CSUROS(name,s,counter,value)                          \
    uint32_t K = (x > y) ? x : y;                                             \
    /* This loop and binary search cannot blow past the 8 sentinels */        \
    while (S >= INTERPRET_BINARY_CSUROS_##value(K+8,s)) K += 8;               \
    if (S >= INTERPRET_BINARY_CSUROS_##value(K+4,s)) K += 4;                  \
    if (S >= INTERPRET_BINARY_CSUROS_##value(K+2,s)) K += 2;                  \
    if (S >= INTERPRET_BINARY_CSUROS_##value(K+1,s)) K += 1;
```

(BA) Binary search after linear search of every eighth Absolutely calculated value

```
#define K_LINEAR_SEARCH_TABLE(name,s,counter,value)                           \
    uint32_t K = (x > y) ? x : y;                                             \
    /* This loop cannot blow past the sentinel */                             \
    while (S >= name##_interpret_##value[K+1]) ++K;
```

(LT) Linear search of entries in value Table

```
#define K_BINARY_SEARCH_TABLE(name,s,counter,value)                           \
    uint32_t K = (x > y) ? x : y;                                             \
    /* This loop and binary search cannot blow past the 8 sentinels */        \
    while (S >= name##_interpret_##value[K+8]) K += 8;                        \
    if (S >= name##_interpret_##value[K+4]) K += 4;                           \
    if (S >= name##_interpret_##value[K+2]) K += 2;                           \
    if (S >= name##_interpret_##value[K+1]) K += 1;
```

(BT) Binary search after linear search of every eighth entry in value Table

```
#define K_MATRIX_LOOKUP(name,s,counter,value)                                 \
    uint32_t K = name##_lookup[x][y];
```

(M) lookup in a 2-D Matrix using x and y (and ignoring S)

Fig. 16. C macros for calculating K from either S or x and y (part 2 of 2).

Table 2. Average Benchmark Execution Time for 264 Implementation Variations of Approximate-counter Addition

				Csűrös	Csűrös	Csűrös	Csűrös	Csűrös	Csűrös	general Morris	general Morris
			estimates type:	$s = 3$	$s = 4$	$s = 5$	$s = 6$	$s = 7$	$s = 8$	$q = 1.08$	$q = 1.08$
			random type:	uint16	uint32	uint64	float	double	double	double	float
S	K	V	G	float	double	double	float	double	double	double	float
C	BA	C	C	9,097	9,073	8,715	25,935	24,607	26,963		
C	BA	T	T	8,610	8,651	8,453	20,512	19,000	19,627		
C	BA	T	W	8,082	8,250	8,056	21,774	19,692	21,345		
C	BT	C	C	7,711	7,958	7,515	15,794	15,542	13,452		
C	BT	T	T	7,265	7,236	7,305	10,813	10,782	9,711		
C	BT	T	W	7,310	6,962	6,892	11,019	10,606	9,726		
C	C	C	C	7,885	7,663	7,935	17,868	17,171	18,223		
C	C	T	T	7,382	7,375	7,685	13,132	12,544	12,159		
C	C	T	W	7,034	7,281	7,628	13,093	12,582	12,744		
C	LA	C	C	8,567	7,551	7,273	16,979	19,231	17,659		
C	LA	T	T	8,491	7,263	7,085	13,002	12,061	10,850		
C	LA	T	W	8,477	7,037	6,723	13,105	12,564	10,961		
C	LI	C	C	7,797	7,370	6,879	17,303	18,176	15,401		
C	LI	T	T	7,587	7,199	6,904	12,842	11,603	11,028		
C	LI	T	W	7,421	7,364	6,521	12,781	13,095	11,080		
C	LT	C	C	7,151	6,751	6,703	16,545	15,292	12,945		
C	LT	T	T	6,695	6,294	6,456	9,965	9,840	8,765		
C	LT	T	W	6,988	6,305	6,373	10,056	9,555	8,924		
C	M	C	C	6,831	6,761	6,347	16,447	13,718	13,420		
C	M	T	T	5,959	6,166	6,386	10,027	9,441	8,680		
C	M	T	W	6,227	6,173	6,318	9,849	9,625	8,906		
T	BA	C	C	8,692	8,442	8,441	22,182	21,189	21,172		
T	BA	T	T	8,204	7,920	7,949	18,050	14,820	15,848		
T	BA	T	W	7,822	7,897	7,745	18,176	15,022	16,043		
T	BT	C	C	7,554	7,333	7,189	13,031	11,290	9,904		
T	BT	T	T	6,946	6,776	6,879	6,836	6,345	6,014	6,842	6,860
T	BT	T	W	7,022	6,514	6,625	7,031	6,436	6,295	7,092	7,056
T	C	C	C	8,033	7,636	8,059	13,886	13,100	12,398		
T	C	T	T	7,160	7,220	7,654	8,922	8,595	8,540		
T	C	T	W	7,031	6,969	7,322	8,767	8,709	8,318		
T	LA	C	C	8,068	7,188	6,909	12,696	12,246	11,328		
T	LA	T	T	7,791	6,881	6,836	8,342	7,638	7,049		
T	LA	T	W	7,469	6,836	6,526	8,417	7,757	7,040		
T	LI	C	C	7,564	6,792	6,660	12,727	12,033	10,994		
T	LI	T	T	7,272	6,683	6,636	8,618	7,785	6,806		
T	LI	T	W	7,106	6,566	6,318	9,248	7,691	6,828		
T	LT	C	C	6,792	6,590	6,323	10,079	9,525	8,804		
T	LT	T	T	6,573	6,148	6,341	5,594	5,249	4,960	6,021	6,075
T	LT	T	W	6,643	6,118	6,218	5,608	5,265	5,007	6,096	6,092
T	M	C	C	6,445	6,373	6,220	9,794	9,494	8,728		
T	M	T	T	**5,929**	**5,721**	**5,894**	**5,555**	**5,173**	**4,856**	**5,839**	**5,711**
T	M	T	W	6,076	5,818	5,978	5,606	5,308	4,962	5,882	5,849

Each entry is a time in milliseconds, computed by making nine measurements of benchmark execution time, discarding the highest and lowest, and averaging the remaining seven. The maximum relative standard deviation in any set of seven averaged measurements was less than 0.122, and the average relative standard deviation over all 264 table entries was less than 0.022. The smallest time in each column is shown in **boldface**. These times may be compared to those in Table 3.

The C macro `K_LINEAR_SEARCH_TABLE` (also identified by `LT`) corresponds to the *add* pseudocode at the end of Section 4 (see page 7); it is similar to `K_LINEAR_SEARCH_CSUROS` but uses a table of estimated values rather than calculating them. (This technique depends on the additional fact that each table named `name##_interpret_##value` is padded at the end with one copy of a sentinel value, of the appropriate type, from Figure 11.)

The C macro `K_BINARY_SEARCH_TABLE` (also identified by `BT`) is similar to `K_BINARY_SEARCH_CSUROS` but uses a table of estimated values rather than calculating them. (This

Table 3. Average Benchmark Execution Time for 10 Standard MPI Combining Operations

MAX	MIN	SUM	PROD	LAND	BAND	LOR	BOR	LXOR	BXOR
1,913	1,924	1,869	1,875	1,891	1,862	1,887	1,854	1,940	1,854

Each entry is a time in milliseconds, computed by making nine measurements of benchmark execution time, discarding the highest and lowest, and averaging the remaining seven. Compare these to Table 2.

```
#define V_CSUROS(name,s,counter,value)                                  \
    value##_t V = INTERPRET_BINARY_CSUROS_##value(K,s);
```

(C) Compute V as a Csűrös value calculated from K

```
#define V_TABLE(name,s,counter,value)                                   \
    value##_t V = name##_interpret_##value[K];
```

(T) Compute V by using a value-table entry indexed by K

Fig. 17. C macros for calculating V from K.

technique depends on the additional fact that each table named `name##_interpret_##value` is padded at the end with *eight* copies of a sentinel value, of the appropriate type, from Figure 11.)

The last C macro in Figure 16, `K_MATRIX_LOOKUP` (also identified by `M`), ignores S and instead uses `x` and `y` to index into a precomputed two-dimensional table. This is expected to be quite fast, but the size of the table is fairly large (half a megabyte for the case of 8-bit approximate counters and a `double` representation for estimated values).

Figure 17 shows two different ways of computing V. The C macro `V_CSUROS` (also identified by `C`) uses the relevant estimate-calculation macro from Figure 12 on K. The C macro `V_TABLE` (also identified by `T`) instead assumes there is a precomputed table, whose name is constructed by the phrase `name##_interpret_##value`, which can be indexed by K to fetch the estimated value. The choice between these two macros for computing S reflects a potential space-time tradeoff that we wished to measure.

Figure 18 shows three different ways of computing G. The C macro `G_FROM_W_CSUROS` (also identified by `C`) uses the relevant estimate-calculation macro from Figure 12 on $K + 1$; then it subtracts V. The C macro `G_FROM_W_TABLE` (also identified by `W`) instead assumes there is a precomputed table, whose name is constructed by the phrase `name##_interpret_##value`, which can be indexed by $K + 1$ to fetch the estimated value; then it subtracts V. The C macro `G_TABLE` (also identified by `T`) bypasses the computation of W and instead assumes there is a second precomputed table, whose name is constructed by the phrase `name##_gap_##random`, which can be indexed by K to fetch the precomputed quantity $W - V$. The choice among these three macros for computing S reflects a potential space-time tradeoff that we wished to measure.

We have now exhibited two ways to compute S, seven ways to compute K, two ways to compute V, and three ways to compute W. However, in our experiments, we chose to take V from a table (using macro `V_TABLE`), if and only if G was also taken from a table (using either `G_FROM_W_TABLE` or `G_TABLE`). Therefore, we tried $2 \times 7 \times 3 = 42$ different implementation combinations. For each of the six Csűrös counter representations, we implemented all 42 combinations, but for the two implementations of Morris counters, only the 6 implementation combinations that use tables (thus avoiding any use of `INTERPRET_BINARY_CSUROS_xxx` macros) are applicable. Therefore, we produced a total of $6 \times 42 + 2 \times 6 = 252 + 12 = 264$ implementations of approximate-counter addition.

```
#define G_FROM_W_CSUROS(name,s,counter,value,random)                    \
    value##_t W = INTERPRET_BINARY_CSUROS_##value(K+1,s);               \
    value##_t G = W - V;
```

(C) Compute G by subtracting V from a Csűrös value calculated from $K + 1$

```
#define G_FROM_W_TABLE(name,s,counter,value,random)                     \
    value##_t W = name##_interpret_##value[K+1];                        \
    value##_t G = W - V;
```

(W) Compute G by subtracting V from a value-table entry indexed by $K + 1$

```
#define G_TABLE(name,s,counter,value,random)                            \
    value##_t G = name##_gap_##random[K];
```

(T) Compute G by using a gap-table entry indexed by K

Fig. 18. C macros for calculating G from K and possibly V.

To measure the relative speed of these implementations, we used an artificial benchmark that within each MPI process constructed an array of 8-bit approximate counters and then repeatedly did two things: initialize the array and then use the `allReduce` method to combine one array from each process, using one of the 264 implementations of approximate-counter addition as the combining operation. Only the execution time of the loop was measured. To provide a baseline for comparison, we also took measurements of the same benchmark using the standard MPI combining operators `MAX`, `MIN`, `SUM`, `PROD`, `LAND`, `BAND`, `LOR`, `BOR`, `LXOR`, and `BXOR` on 8-bit values.

We tried array lengths of 10,000, 100,000, and 1,000,000, adjusting the number of iterations to keep measured run times between 2 and 30s (most were at least 5s); we also tried various ways of initializing the arrays. These variations had some effect on the absolute measured run time, but had almost no effect at all on the relative comparisons. Therefore, we present just one set of measurements here, in Table 2, as being representative of all our observations. For these measurements the array length was 1,000,000, the number of iterations was 250, and every element of every array was initialized to 43. The measurements were taken on the same hardware (16 nodes containing two Xeon processors each) used for the SCA application measurements described in Section 12, using 32 JVM instances communicating with one another as 32 MPI processes. Each instance of the benchmark was run nine times; from each set of nine measurements, the lowest and highest values were discarded and the other seven averaged to produce the data presented in Table 2. For comparison, similarly processed measurements of 10 standard MPI combining operations applied to the same data are presented in Table 3.

The results are perhaps unsurprising. In the specific CPU-based hardware/software environment that we measured, we generally find that:

—Approximate-counter addition is slower than any of the 10 standard MPI combining operations.
—Incremental calculations during a linear search (`LI`) are faster than from-scratch absolute calculations during a linear search (`LA`).
—Linear search turns out to be faster than binary search.
—Table-based techniques are faster than making calculations.

Our conclusion, however, is not that any one implementation technique is superior in all circumstances, but rather that it may be worthwhile to compare the speed of several such techniques in each new computational environment. In a GPU-based environment, for example, it may be infeasible to use a very large table, or it may be that some form of calculation is faster than using even small tables, or it may be that binary search is faster than linear search, either because of SIMD constraints or because the value of q is small (very close to 1).

14 DON'T SUBTRACT (OR DECREMENT) APPROXIMATE COUNTERS

It is conceptually straightforward to define a subtraction operation for approximate counters: just take the pseudocode for the *add* operation in Section 3 and change the "+" to "−" in the computation of S in line 4:

```
1:  procedure subtract(var X: T, Z: T)
2:      let v ← f(X)
3:      let w ← f(Z)
4:      let S ← v − w                    ▷ We do not recommend use of this procedure!
5:      let K ← φ(S)
6:      let V ← f(K)
7:      let W ← f(τ(K))
8:      let Δ ← (S−V)/(W−V)
9:      if random() < Δ then
10:         X ← τ(K)
11:     else
12:         X ← K
```

It is similarly easy to define a *decrement* operation (perhaps in terms of *subtract*).

But this is *not* a good idea. Subtraction increases the variance of the result in the same manner as addition, but the result itself may be smaller in magnitude (absolute value) than either of its operands, so a series of addition and subtraction operations can cause the relative standard deviation to grow without bound.

We tried to obtain empirical confirmation of this theoretical observation by taking a well-known LDA algorithm that, rather than recalculating all counts from scratch on each iteration, instead updates counters incrementally: whenever the topic assigned to a word is changed, it decrements the words-per-topic count for the old topic and then increments the words-per-topic count for the new topic. We straightforwardly modified this algorithm to use approximate counters rather than standard integer counters. As expected, it behaved extremely poorly. It wasn't just that the log likelihood of the computed parameters failed to converge, even after an enormous number of iterations; there was gross misbehavior. We finally realized that the correctness of the algorithm depends critically on the fact that every counter always has a nonnegative value—but when incrementations and decrementations are only statistical, a probabilistic counter can take on negative values even if the number of decrementation attempts is always smaller than the number of incrementation attempts. Trying to address this naïvely by saturating at the lower end (i.e., refusing to decrement a counter that is already zero) introduced biases that violated other invariants of the algorithm, such as that the sum (or at least the expected sum) over an entire array should remain constant. While this particular experiment did not fully verify that the variance might grow without bound in practice, it did support the general hypothesis that pitfalls await the unwary who use subtraction on approximate counters.

15 MULTIPLICATION OF APPROXIMATE COUNTERS MIGHT BE WELL-BEHAVED

It *might* be reasonable to define a multiplication operation for approximate counters and to attempt to prove an appropriate bound on the variance of the product. However, we have not yet encountered or thought of a practical application for such an operation.

16 WHY HASN'T THIS BEEN DONE BEFORE?

When we set out to test a distributed version of the single-GPU LDA Gibbs algorithm with approximate counters, we recognized the need to replicate the counters—simple tests showed that trying to increment counters stored on a remote node would result in much slower performance—and therefore the need to add approximate counter values. We thought it would be easy to locate the necessary algorithm in the literature; approximate counters have been around for almost four decades, and it's an "obvious" operation to provide. But our best efforts uncovered no mention at all of this operation or anything like it.

We speculate that the need simply has not arisen until now, and offer a "just-so" story: If counters are stored in a central memory, then it never makes sense to use a replicated representation for the counters; if one can afford to store two copies of an 8-bit approximate counter, then one is better off using one 16-bit approximate counter, affording greater range or precision or both. So adding approximate counters makes sense only in a distributed setting. But, we have found that most uses of approximate counters in the literature have had to do with counting events (such as performance counters in a hardware processor); it does make sense, for example, for every processor in a cluster to have its own counters, but typically one clears the counters, runs a computation, and then gathers and aggregates the performance data just once, after the computation has completed. For database applications, past use has typically focused on counting features while making a single (possibly distributed) pass over the database. In such cases there is nothing to be gained by reducing the aggregated values back to the approximate-counter representation; rather, one communicates the approximate counter values, expands each to a full integer, and then sums the integers—that's all there is to it.

What motivates addition of approximate counters is an *iterative* distributed application that can use replicated approximate counters within each iteration, where each iteration also includes the aggregation and redistribution of such replicated counters. We found that machine-learning algorithms (such as topic modeling) and stochastic cellular automata fit this description and benefit accordingly. We admit that one benefit of adding approximate counters in such applications, namely the reduction in network traffic, could be had in other ways, such as applying a generic data-compression algorithm to an array of ordinary integer counters. However, the addition process is fairly quick, and there are other benefits to maintaining intermediate values in approximate-counter form, such as reduction of memory footprint.

17 CONCLUSIONS AND FUTURE WORK

Statistically independent approximate counters can be added, producing a result in the same representation, so the expected value of the result is the sum of the expected values of the operands, and the variance of the result is bounded. We present specific novel algorithms for adding five kinds of approximate counters in the literature; for three of them (general Morris, binary Morris, and Csűrös), we present proofs of bounded variance. We report that replacing integer counters with approximate counters maintains the statistical behavior of a distributed multiple-GPU implementation of a machine-learning application while improving its overall performance by almost a factor of 3, and of a distributed multiple-CPU implementation of another machine-learning application while improving its overall performance by almost a factor of 2.

It remains to produce proofs of bounded variance (if possible) for the other two kinds of approximate counters (DLM probability and DLM floating-point), to measure the relative speeds of various implementations of approximate-counter addition in other computational environments, and to investigate what other sorts of applications might benefit from adding approximate counters.

ACKNOWLEDGMENTS

We thank Yossi Lev for discussion of this material and for pointing us to important related work. We also thank Daniel Ford and the anonymous referees for helpful comments.

REFERENCES

A. C. Callahan. 1976. Random rounding: Some principles and applications. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP'76)*, Vol. 1. IEEE, New York, 501–504. DOI: http://dx.doi.org/10.1109/ICASSP.1976.1169938

Miklós Csűrös. 2010. Approximate counting with a floating-point counter. In *Proceedings of the 16th Annual International Conf. Computing and Combinatorics (COCOON'10)*. Springer-Verlag, Berlin, 358–367.

Andrej Cvetkovski. 2007. An algorithm for approximate counting using limited memory resources. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'07)*. ACM, New York, 181–190. DOI: http://dx.doi.org/10.1145/1254882.1254903

Dave Dice, Yossi Lev, and Mark Moir. 2013. Scalable statistics counters. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'13)*. ACM, New York, 307–308. DOI: http://dx.doi.org/10.1145/2442516.2442558

Philippe Flajolet. 1985. Approximate counting: A detailed analysis. *BIT Numeric. Math.* 25, 1 (1985), 113–134. DOI: http://dx.doi.org/10.1007/BF01934993

George E. Forsythe. 1959. Reprint of a note on rounding-off errors. *SIAM Rev.* 1, 1 (1959), 66–67. DOI: http://dx.doi.org/10.1137/1001011

Scott A. Mitchell and David M. Day. 2011. Flexible approximate counting. In *Proceedings of the 15th Symposium on International Database Engineering & Applications (IDEAS'11)*. ACM, New York, 233–239. DOI: http://dx.doi.org/10.1145/2076623.2076655

Robert Morris. 1978. Counting large numbers of events in small registers. *Commun. ACM* 21, 10 (Oct. 1978), 840–842. DOI: http://dx.doi.org/10.1145/359619.359627

D. Stott Parker. 1997. *Monte Carlo Arithmetic: Exploiting Randomness in Floating-Point Arithmetic*. Technical Report 970002. Computer Science Department, UCLA, Los Angeles, CA. Retrieved from http://fmdb.cs.ucla.edu/Treports/970002.pdf.

D. S. Parker, B. Pierce, and P. R. Eggert. 2000. Monte carlo arithmetic: How to gamble with floating point and win. *Comput. Sci. Eng.* 2, 4 (July 2000), 58–68. DOI: http://dx.doi.org/10.1109/5992.852391

Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. 1998. *MPI—The Complete Reference: Volume 1, The MPI Core* (2nd ed.). MIT Press, Cambridge, MA.

Rade Stanojevic. 2007. Small active counters. In *Proceedings of the 26th IEEE International Conference on Computer Communications (INFOCOM'07)*. IEEE, Piscataway, NJ, 2153–2161. DOI: http://dx.doi.org/10.1109/INFCOM.2007.249

Guy L. Steele Jr. and Jean-Baptiste Tristan. 2016. Adding approximate counters. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'16)*. ACM, New York, Article 15, 12 pages. DOI: http://dx.doi.org/10.1145/2851141.2851147

Jean-Baptiste Tristan, Joseph Tassarotti, and Guy L. Steele Jr. 2015. Efficient training of LDA on a GPU by mean-for-mode gibbs sampling. In *Proceedings of the 32nd International Conference on Machine Learning (ICML'15)*, Vol. 37. Journal of Machine Learning Research / Microtome Publishing, Brookline, MA, 10. Retrieved from http://jmlr.org/proceedings/papers/v37/tristan15.pdf.

A. J. Walker. 1974. Fast generation of uniformly distributed pseudorandom numbers with floating-point representation. *Electronics Lett.* 10, 25 (December 1974), 533–534. DOI: http://dx.doi.org/10.1049/el:19740423

Manzil Zaheer, Michael Wick, Jean-Baptiste Tristan, Alex Smola, and Guy L. Steele Jr. 2015. Exponential stochastic cellular automata for massively parallel inference. In *Proceedings of LearningSys: Workshop on Machine Learning Systems at Neural Information Processing Systems (NIPS'15)*. 10.

Manzil Zaheer, Michael Wick, Jean-Baptiste Tristan, Alex Smola, and Guy L. Steele Jr. 2016. Exponential stochastic cellular automata for massively parallel inference. In *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics (AISTATS'16). J. Mach. Learn. Res.*, 966–975.

Received January 2017; revised June 2017; accepted July 2017