

# Lightweight Parallel Accumulators Using C++ Templates

Yossi Lev  
Oracle Labs  
yossi.lev@oracle.com

Mark Moir  
Oracle Labs  
mark.moir@oracle.com

## ABSTRACT

The Boost.Accumulators framework provides C++ template-based support for incremental computation of many important statistical functions, such as maximum, minimum, mean, count, variance, etc. Basic accumulators can be combined to build more sophisticated ones. We explore how this framework can be extended to implement lightweight *parallel* accumulators that allow multiple threads to `Store` sample data, and support concurrent `GetResult` operations that incrementally compute desired functions over the data. Our evaluation shows that our parallel accumulators are scalable and can effectively exploit programmer-supplied knowledge to achieve significant optimizations for some important cases.

## Categories and Subject Descriptors

D.1.3 [PROGRAMMING TECHNIQUES]: Concurrent Programming—*Parallel programming*

## General Terms

Algorithms, Performance, Design

## Keywords

Concurrent programming, template metaprogramming

## 1. INTRODUCTION

The Boost.Accumulators framework [11] is a component of the widely-used C++ Boost libraries that supports computation of various statistical summary functions over data. These functions are computed *incrementally*: the desired function is computed over the data pushed into an accumulator so far, without waiting until all data has been pushed. A flexible framework allows programmers to specify new accumulators, and to combine them to yield more sophisticated accumulators. The programmer can then define an *accumulator set*, that calculates any number of accumulator functions while sharing common components between them.

Given the rapid shift to multicore computing, a shortcoming of the Boost.Accumulators framework is that it requires all operations

on the accumulator set (pushing new data or computing the summary function) to be executed sequentially. This paper explores how to extend the library to allow multiple operations to be executed in parallel for accumulators whose summary functions are both associative and commutative, thereby providing parallel implementations of a large class of important statistical functions.

A key contribution of our work is that our framework allows the summary function to be computed over the data pushed so far, even concurrently with new data being pushed. However, we optimize for the case of data being pushed in the absence of such concurrent requests, as we believe that typical usage will push many samples, but only occasionally request the current result. For example, Troyer et. al. use accumulators for parallel Monte Carlo simulations [13], and require occasional evaluation of the summary function to check the current statistical error, and terminate the simulation when certain error is achieved.

We provide even more efficient push operations for several cases of interest: when support for computing the summary function concurrently is not needed, when the programmer indicates that the desired data and summary functions have the “NRV” property discussed in Section 2, or when only a single thread uses the accumulator. In this last case, we aim to achieve performance comparable with a handcrafted single-thread solution. Finally, we provide support for functions that are not associative and commutative by allowing programmers to provide additional synchronization code.

Our framework supports the atomic evaluation of multiple accumulators. At least for associative and commutative summary functions, this is achieved without the programmer writing any concurrent synchronization code. As in Boost.Accumulators, we use C++ template programming in order to share common functionality between accumulators by tracking the dependencies between them, and to apply various compile-time optimizations when applicable.

Below we present a detailed example of how parallel accumulators can be defined and used with our framework. In Section 2, we introduce the concept of Non Repeating Value (NRV) variables, and explain how they can be exploited to implement efficient parallel accumulators and to compose them. Section 3 describes the library support for our framework, including some important optimizations and extensions that can be enabled by additional synchronization code provided by the programmer. We evaluate our implementation in Section 4, and conclude in Section 5.

### 1.1 Usage example

The following example shows how parallel accumulators are used, and how the framework can be used to define new ones.

Figure 1 shows a single thread computing the count and sum of two samples using a parallel accumulator. As with the serial version of Boost.Accumulators, we declare an accumulator set type `accSet` with the sample type (`int`) and the *tags* (names) of the functions we

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICSE '11, May 21–28, 2011, Waikiki, Honolulu, HI, USA  
Copyright 2011 ACM 978-1-4503-0577-8/11/05 ...\$10.00.

```

1  typedef
2  ParallelAccSet<int, TypeList<Sum, Count>> accSet_t;
3
4  accSet_t mySet;
5
6  // Each thread:
7  //
8  accSet::Tid tid = mySet.Register();
9
10 mySet.Store(tid, 2);
11 mySet.Store(tid, 3);
12
13 cout << "Sum: " << mySet.GetResult<Sum>() << endl;
14 cout << "Count: " << mySet.GetResult<Count>() << endl;

```

**Figure 1: Using the Sum and Count parallel accumulators.**

would like to compute (line 2), allocate an instance (`mySet`) of that type (line 4), and then push samples into it (lines 10–11). With the parallel version, many threads can push samples concurrently. Each thread must call `Register` (line 8) before pushing its first element, in order to get an identifier (`tid`) that it uses with subsequent `Store` operations. A thread can also call the `GetResult<tag>` method (lines 13 and 14), which computes the function specified by `tag` for all samples pushed so far by all threads.

Next, Figure 2 demonstrates how the library can be extended with additional parallel accumulators by showing how the `Sum` accumulator, as well as another accumulator `Mean` that uses `Sum` and `Count` to calculate the mean of the samples, are defined.

Like in the serial version, defining an accumulator consists of two main steps: defining a structure that implements the accumulator (`SumImp` and `MeanImp` in our example), and defining the accumulator tag (`Sum` and `Mean`) that identifies the implementation and specifies any dependencies it has on other accumulators.

In more detail, the `SumImp` template structure, parameterized by the sample type, provides the framework with the structures it needs to generate a parallel `Sum` implementation:

- `resultType` (line 4) is the type of the result produced by the `Sum` function. In this case, it is the same as the sample type.
- The `tldata` structure (lines 5–9) defines the variable(s) needed to accumulate samples. For `Sum`, this is simply a `sum` variable of the sample type (line 8).
- The `Store` method (lines 10–12) defines how to update the `tldata` variables when adding a new sample.
- The `Combine` method (lines 13–15) defines how to combine the data values represented by two `tldata` structures.
- The `Result` method (lines 16–20) specifies how the result (`Sum`) is determined from a `tldata` structure. (We describe the `DepResults` template parameter later.)

Observe that none of the code provided by the programmer to implement `Sum` involves concurrency or synchronization: it merely defines how to accumulate data values, combine accumulated values, and compute the summary function from accumulated values. In particular, the programmer does not specify how the summary function is computed concurrently with samples being stored.

Next, we define an accumulator tag for `Sum`, and associate it with the implementation structure by defining the `imp` type, which simply defines `imp` to derive from `SumImp<Sample>` (line 25). In addition, we provide a sequence of tags of other accumulators on which `Sum` depends (line 23); here the special empty sequence

```

1  // Sum implementation and accumulator tag
2  template <typename Sample>
3  struct SumImp {
4      typedef Sample resultType;
5      struct tldata {
6          tldata():sum(0) {
7              }
8          Sample sum;
9      };
10 void Store(const Sample& s, tldata& d) {
11     d.sum += s;
12 }
13 void Combine(const tldata& tld, tldata& accData) {
14     accData.sum += tld.sum;
15 }
16 template <typename DepResults>
17 resultType Result(const tldata& accData,
18                 DepResults& theDepResults) {
19     return accData.sum;
20 }
21 };
22 struct Sum {
23     typedef nil dependencies;
24     template <typename Sample>
25     struct imp : public SumImp<Sample> {
26     };
27 };
28
29 // Mean implementation, dependent on Sum and Count
30 template <typename Sample>
31 struct MeanImp {
32     typedef double resultType;
33     struct tldata {
34     };
35     template <typename DepResults>
36     resultType Result(const tldata& accData,
37                     DepResults& theDepResults) {
38         return (double)
39             theDepResults.template GetDepResult<Sum>() /
40             theDepResults.template GetDepResult<Count>();
41     }
42 };
43 struct Mean {
44     typedef TypeList<Sum, Count>.seq dependencies;
45     template <typename Sample>
46     struct imp : public MeanImp<Sample> {
47     };
48 };

```

**Figure 2: Defining the Sum and Mean parallel accumulators.**

`nil` indicates that `Sum` does not depend on any other accumulators. `CountImp` and `Count` (not shown) are defined similarly.

Finally, we define the implementation and tag for the `Mean` accumulator. Because the `Mean` accumulator simply uses the `Sum` and `Count` accumulators and calculates the ratio between their results, its implementation structure `MeanImp` does not accumulate any data of its own, and thus has an empty `tldata` structure (lines 33–34), and no `Store` or `Combine` method. To access the results of `Sum` and `Count` (on which `Mean` depends), the `Result` method of `MeanImp` calls the `GetDepResult` method of its `theDepResults` argument. The framework ensures that the results it provides for the `Sum` and `Count` accumulators via this argument are consistent with each other: these results were both “current” at some point during the execution of the `GetResult` operation that called `Result` (see Section 3). The tag for the `Mean` accumulator (lines 43–48) then indicates to the framework, via the `dependencies` sequence type, that `Mean` depends on `Sum` and `Count` (line 44), so that the library knows to use these accumulators even if the accumulator set type only explicitly specifies `Mean` as the function to be computed.

## 2. NRV-BASED SYNCHRONIZATION

The concept of Non-Repeating Values (NRV) is central to our approach. An NRV variable never changes to a value it has held before, and therefore if we read a value from it that we have read from it previously, we can conclude that it has contained that value continuously since the previous read.

NRVs arise naturally in many functions of interest. For example, as we add samples to a set of integers, the count, minimum, and maximum of the samples are all NRVs. In other cases, it is not difficult to derive an NRV function even for a variable whose value may repeat. An example is the sum of a set of integer samples. While the sum of the integers may repeat over time (due to the inclusion of negative numbers in the sample type), if we track sums of the positive and negative samples separately, the sum of their absolute values is an NRV. More generally, any variable can be transformed into an NRV by augmenting it with a counter that is incremented every time the variable is updated.<sup>1</sup>

Given a set of NRV variables, it is straightforward to take an atomic (instantaneous) snapshot of them by repeatedly scanning the variables until two successive scans yield the same values for all variables, providing a snapshot. By the NRV property, the values in the snapshot were all simultaneously current in the interval between the two scans. This technique is used in nonblocking snapshot algorithms [3] and for validating reads in some software transactional memory (STM) implementations (e.g., [8]).

The ability to take an atomic snapshot of a set of NRV variables suggests a way to parallelize an implementation of an associative and commutative NRV function: each thread maintains a thread-local NRV variable representing the contributions of its own Stores, and we get the function value by taking an atomic snapshot of them and combining the values. Having each variable updated only by a single thread makes lightweight solutions feasible.

In the above-described approach, a variable could change between every pair of scans, preventing the snapshot from ever completing. To address this issue, we can use an “anti-starvation” flag to request Store operations to wait until an ongoing snapshot completes, as illustrated by the following pseudocode.

```
Store(tid, s) {
    while (GetIsRunning); // spin
    UpdateTLD(tid, s); // update thread-local data
}

Get() {
    GetIsRunning = true;
    aggrData = Combine(GetSnapshot());
    GetIsRunning = false;
    return CalculateResult(aggrData);
}
```

The `GetIsRunning` flag is not required for correctness: it merely facilitates progress. The program is correct even if Stores ignore it, so expensive memory barriers are unnecessary.

The pseudocode above assumes only a single thread executes Get operations; it is straightforward to extend this approach to support Gets by multiple threads, for example using a shared counter implemented using compare-and-swap (CAS), or using a SNZI [5]. While this puts additional overhead on Get operations, it does not affect Store operations, consistent with our design goals. Furthermore, it is not difficult to extend these ideas so that Stores cannot be starved by frequent Gets, again without adding significant overhead to Stores. For example, if the anti-starvation flag is replaced by a counter, then Store operations can spin when the counter is odd, indicating that a Get is executing, but can distinguish between

<sup>1</sup>In principle, a counter may wrap around and is thus not a pure NRV. We assume large enough counters that this does not happen in practice, and do not address this issue further in this paper.

different Get operations, allowing them to spin only while one Get completes. This ensures that neither kind of operation can starve the other indefinitely. Because we assume Gets are infrequent, our basic implementation does not include this mechanism.

Next we discuss how we can build a parallel accumulator that depends on others, such as the example in Section 1. Although we can achieve atomic snapshots of the threads’ contributions to the Count and Sum accumulators using the technique described above, nothing guarantees that these snapshots will be consistent with each other. For example, if the 4th sample was already stored in the Sum accumulator but not in the Count accumulator, then a `GetResult<Mean>` operation may return an incorrect result, dividing the sum of 4 samples by 3. A similar problem occurs if the thread local NRV variable of an accumulator cannot be written atomically.

To address this problem, when a thread performs a Store operation that needs to update multiple thread local variables atomically, it increments a per-thread NRV counter before beginning these updates, and again after completing them. Thus, a Store operation that targets accumulators `Acc1...AccN` operates as follows:

```
Store(tid, s) {
    while (GetIsRunning); // spin
    Increment(NRVCounters[tid]);
    UpdateTLD<Acc1>(tid, s);
    UpdateTLD<Acc2>(tid, s);
    ...
    UpdateTLD<AccN>(tid, s);
    Increment(NRVCounters[tid]);
}
```

This way, Get can detect that a Store is in progress by observing that this counter is odd.<sup>2</sup> That is, Get iterates over all threads’ data and counters until it sees that none of the counters is odd or has changed since the last time it was read:

```
Get() {
    GetIsRunning = true;
    done = true;
    NRVCounter snapshot[MaxThreads]; // init 0
    TLD aggrData;
    do {
        NRVCounter c;
        aggrData = TLD(); // reset aggrData
        foreach (threadId tid) {
            Combine(GetTLD(tid), aggrData);

            // spin until even
            while ((c = NRVCounters[tid]) % 2);
            done = done && (snapshot[tid] == c);
            snapshot[tid] = c;
        }
    } while (!done)
    GetIsRunning = false;
    return CalculateResult(aggrData);
}
```

This way, there is no need for the thread local data of each accumulator to be of an NRV type, or be written atomically.

In the next section we describe how we used these algorithms to build our parallel accumulator set, which supports parallel Store and GetResult operations for any number of accumulator functions, of either NRV or non-NRV type.

## 3. IMPLEMENTATION

Figure 3 illustrates the implementation of the framework using pseudocode. An object of type

```
ParallelAccSet < sampleType, Seq < Acc1, ..., AccN >>
```

<sup>2</sup>The order of the NRV counter store and updates to the thread local data must be preserved; most modern architectures do so and this can be enforced on others using a store-store barrier.

```

1  typedef unsigned long long NRVCounter;
2
3  template <typename SampleType, typename AccSeq>
4  class ParallelAccSet {
5  public:
6
7      void Store(Tid tid, SampleType s) {
8          TLD tld = this->tldContainer.GetTLD(tid);
9          BeginStore(&tld);
10
11         ForEach(imp in this->allImp) {
12             imp.Store(s, Get<imp.tldata>(tld.allAccsTLD));
13         }
14
15         EndStore(&tld);
16     }
17
18     template <typename AccTag>
19     AccTag::imp::resultType
20     GetResult() {
21
22         while (!CAS(this->GetIsRunning, false, true)); // spin
23
24         // accsToCompute is a sequence consisting of
25         // AccTag and everything it depends on (see text).
26
27         NRVCounter snapshot[MaxThreads];
28         do {
29             done = true;
30             tldSeq aggTLDSeq;
31             int i = 0;
32             TLDContainer::Iter itr = this->tldContainer->Iter();
33             for (; !itr.End(); itr.next()) {
34                 TLD tld = *itr;
35                 ForEachType(acc in accsToCompute) {
36                     acc::imp& imp = Get<acc::imp>(this->allImp);
37                     imp.Combine(
38                         Get<imp.tldata>(tld.allAccsTLD),
39                         Get<imp.tldata>(aggTLDSeq)
40                     );
41                 }
42                 NRVCounter c;
43                 while ((c = tld.counter)%2 != 0) ; // spin
44                 done = done && (snapshot[i] == c);
45                 snapshot[i++] = c;
46             }
47         } while (!done);
48
49         this->GetIsRunning = false;
50
51         AccTag::imp& imp = Get<AccTag::imp>(this->allImp);
52         DepResults depResults(aggTLDSeq);
53         return
54             imp.Result(Get<imp.tldata>(aggTLDSeq), depResults);
55     }
56 private:
57     inline void BeginStore(TLD* tld) {
58         while (this->GetIsRunning) ; // spin
59         tld.counter++;
60     }
61
62     inline void EndStore(TLD* tld) {
63         tld.counter++;
64     }
65
66     struct TLD {
67         tldSeq allAccsTLD;
68         NRVCounter counter;
69         // Padding
70     };
71
72     allImpSeq allImp;
73     TLDContainer<TLD> tldContainer;
74     bool GetIsRunning;
75 };

```

Figure 3: Pseudocode illustrating library implementation.

supports three methods. `Register()` returns a unique thread id (`Tid`) to be passed to the `Store` method of this object; it should be called once per thread for each such object. `Store(tid, s)` pushes a sample `s`, of type `sampleType`, to each of the accumulators `Acc1...AccN`. `GetResult<AccTag>()` returns the accumulated result for `AccTag` (where `AccTag` is one of the `Acc1...AccN` types).

The `Store` method pushes the specified sample to *each* of the accumulators in the set. Each of these appears to take effect instantaneously at some point during the execution of the `Store` method, but they are *not* guaranteed to all occur atomically together. If this is required for some subset of the accumulators, the programmer must explicitly construct an accumulator that depends on each member of the subset, so the implementation guarantees that the `GetResult` method for that accumulator returns a result based on a consistent view of the accumulators on which it depends.

To reduce overhead, especially for `Store` operations, we employ template metaprogramming [2] techniques, similar to those used in `Boost.MPL` [7] and `Boost.TypeTraits` [1], to specialize code *at compile time* depending on the types with which `ParallelAccSet` is instantiated. Briefly, our metaprogram operates on *type sequences*, which are collections of types determined at compile time. Abstractly, a type sequence can be thought of as a list whose elements are of different types; in practice, we implement a type sequence as a structure whose fields correspond to the sequence elements, and use template structures and methods to access these elements, and to transform one type sequence to another.

We begin by describing the main compile-time algorithms and type sequences that we use, and then continue with the description of the `ParallelAccSet` method implementations.

### 3.1 Types and data

When a `ParallelAccSet` is instantiated with accumulators `Acc1...AccN`, the compiler recursively follows the dependencies between them and defines the `allAccSeq` sequence, which consists of the `Acc1...AccN` tags and all accumulator tags they depend on (duplicates are removed). Next, we define the `allImpSeq` sequence type, consisting of the accumulator implementation structure `imp` of each accumulator tag in `allAccSeq`.

The compiler also creates the `tldSeq` type, comprising a sequence of `imp::tldata` structures, one for each implementation `imp` in `allImpSeq`. This way, we can allocate the thread local data for different accumulators together, improving both cache locality and memory usage (because we do not need padding to avoid false sharing between different data owned by the same thread). We then define the `TLD` structure (lines 66-70) which consists of a field of type `tldSeq`, an `NRVCounter`, and padding. (Note that the illustration in the Section 2 showed the counter as separate from other thread-local data, while in the library it is part of the `TLD` structure.)

A `ParallelAccSet` object consists of an instance `allImp` of the `allImpSeq` type described above, an anti-starvation flag `GetIsRunning`, and a container `tldContainer` holding a `TLD` object for each registered thread (lines 72-74).

### 3.2 Methods

`Register` (not shown) allocates a `TLD` instance and inserts it into `tldContainer`, allowing `GetResult` to iterate over the thread-local data and counters of all registered threads. The `counter` field of the `TLD` instance allocated by a `Register` call that returns `tid` is analogous to `NRVCounters[tid]` in Section 2.

The `Store` method (lines 7-16) encloses updates to the thread local data for all accumulators between calls to `BeginStore()` and `EndStore()` (lines 9 and 15); these functions increment the per-thread counter (lines 60 and 63) and `BeginStore()` waits for `GetIs-`

Running to be false (line 59), thus respecting the anti-starvation flag, as described in Section 2.

The per-thread `tldata` structures are updated for each accumulator in the set by using the `ForEach` template method (line 11), which calls the `Store` method of each implementation in the `allImp` sequence (line 12), passing the appropriate `tldata` field from the `allAccsTLD` sequence (the `Get<imp.tldata>` template method yields a reference to the `tldata` field for implementation `imp`).

In practice, the `ForEach` method is not really a loop, and the `Get` method does not really iterate over the `allAccsTLD` sequence at run time: because the type and positions of all elements in the sequence are known at compile time, these constructs simply instruct the compiler to generate code to access each of these elements.

The `GetResult` method begins by setting the anti-starvation flag `GetsRunning` (line 22), waiting if it is already set by another thread (here we show a simple version that allows only one `GetResult` operation to execute at a time, as discussed in Section 2).

It then aggregates per-thread data from all threads (lines 28-47) for the required accumulators into the `aggTLDSeq` variable, ensuring that the result is based on a consistent snapshot of this data. The accumulators for which a `GetResult<AccTag>()` call collects data are `AccTag` as well as any accumulators on which `AccTag` depends (directly or indirectly); the compiler facilitates this by defining a sequence `accsToCompute`, allowing `GetResult` to collect data from each relevant accumulator by calling its `Combine` method (lines 35–39). As before, the loop uses the `Get` method to access required fields in the `allAccsTLD` and `aggTLDSeq` sequences.

To ensure that the aggregated data is based on a consistent snapshot, a `snapshot` array records the last value seen for each per-thread NRV counter. After collecting the data for a given thread, we compare its NRV counter to the previous, recording whether any comparisons have failed (line 44), and storing the current value for comparison in the next scan in case some comparisons did fail (line 45). We never record an odd NRV counter value, but rather spin until it is even (line 43); this ensures that we successfully exit the loop only when we have iterated over all the thread's TLD structures while no `Store` was in progress (see lines 60 and 63).

Scanning the NRV counters and combining all `tldata` structures continues until all of the NRV counters are observed not to have changed since they were last recorded (see lines 29, 44, and 47). Then we can conclude that no `Store` by any thread occurred while the data was being aggregated, and thus that the aggregation for all accumulators was based on the same set of sample data.

We have optimized our implementation based on the observation that, because the counters we snapshot never decrease, it suffices to compare their sums, rather than their individual components. However, this optimization applies only to `GetResult` performance. We have foregone various other optimizations to `GetResult` for the sake of simplicity, as we are most interested in optimizing `Store` performance. For example, taking a snapshot immediately before scanning the threads' data would significantly increase the chances of avoiding collecting the data a second time.

After acquiring a snapshot, the anti-starvation flag is released (line 49), and the `Result` method of `AccTag`'s implementation object is called to determine the value of the summary function computed by `AccTag` (line 53). Because this value may depend on the results of other accumulators on which `AccTag` depends, the aggregated data for these accumulators is also made available to the `Result` method associated with `AccTag`. This is achieved using the `depResults` object (line 52), which provides a `GetDepResult` method for each accumulator `acc` on which `AccTag` depends; this method simply calls the `Result` method of `acc`'s implementation, passing the data aggregated for `acc`, and the `depResults` object.

### 3.3 Custom synchronization

Having `Store` operations update only per-thread data, as described above, is key to the efficient parallelization of accumulators. Nonetheless, synchronization between `Stores` is useful in some cases.

Consider the density function of `Boost.Accumulators`, which requires the choice of histogram bins to be determined based on the first  $I$  samples for some specified  $I$ . This functionality *requires* synchronization between threads, because otherwise `Store` operations by one thread do not know how many `Store` operations *in total* have been executed. This functionality can be supported effectively using our framework using custom synchronization.

Specifically, the `Store` method of the implementation structure for an accumulator can use one or more additional variables declared as part of that structure for synchronization between threads. (Note that the framework allocates only one instance of the implementation structure for each accumulator, via the `allImp` field.)

In our example, the `density` implementation can maintain an additional shared counter that records how many of the initial  $I$  samples have been stored, and additional information required to determine the histogram bins. Once the bins are established, a thread needs no longer modify the shared state stored in the implementation structure, and can simply increment the counter of the appropriate bin by modifying its `tldata` structure. Various optimizations can improve the scalability of the first  $I$  stores, such as replacing the shared counter with a scalable barrier [9].

Some care is required if the shared state is accessed in the `Result` method, because it may not be consistent with the aggregated data with which `Result` is called. For example, an *incorrect* approach would be to provide an implementation of a `Min` accumulator that just uses CAS to update a shared `min` variable. Having `Result` simply return the value of this variable may lead to incorrect behavior if another accumulator is dependent on `Min`: the result returned by `Min::Result` is based on data that may not be consistent with that used for the accumulator that depends on `Min`. Nonetheless, as discussed below, a “standalone” accumulator (one that is independent of all other accumulators in the set) can use arbitrary synchronization mechanisms in any of its methods, and still be used with the library, because it is not required to be consistent with any other accumulator.

### 3.4 Optimizations

While the overhead of checking the anti-starvation flag and incrementing the NRV counter in the `Store` operation is very low, it might still be non-negligible compared to the other cost of some simple accumulators, like `Count` or `Sum`. Here we describe a few cases where this overhead can be further reduced or even eliminated, especially for simple accumulation functions.

**Limited concurrency** If a `ParallelAccSet` instance is for single-threaded use only, or if it does not need to support `GetResult` running concurrently with `Stores`, then the `Store` operation need not update the NRV counters or check the anti-starvation flag. `ParallelAccSet` supports two optional template arguments, `MaxThreads` (default 64) and `ConsistentGet` (default true), and provides empty `StoreBegin` and `StoreEnd` methods if either `MaxThreads=1` or `ConsistentGet` is false. Thus, programmers can declare both single threaded and multithreaded variants of the same `ParallelAccSet`, simply by specifying different template arguments.

**No external NRV counter** A set in which *all* accumulators are independent of each other allows us to elide the NRV counters in two cases. One is when the accumulator uses custom synchronization, and does not require any thread local data; an example is the CAS-

based `Min` mentioned above, which is not safe to use when other accumulators depend on it, but can be used when evaluated independently. The other case is when the accumulator implementation itself can provide an NRV value. We discuss some examples below.

To enable an accumulator to benefit from this optimization when circumstances allow, while still allowing it to be used when the optimization is not applicable, an optional `standAloneImp` type can be specified in the tag for an accumulator, in addition to the regular `imp` type that specifies the regular implementation. At compile time, the library chooses the appropriate implementation based on the dependencies between the set's accumulators.

A standalone implementation must either a) include its own synchronization code to make the `Result` method work correctly despite concurrent `Store` operations, or b) provide its own NRV value, to be used by the `GetResult` method when aggregating thread local data. In the self synchronized case, the implementation can still use a `tldata` structure as before, but no `Combine` method should be provided, and the `Result` method should get no arguments.

An accumulator that provides its own NRV value based on its thread-local data does so via an additional `SelfNRV` method. For functions like `Count` and `Max`, this method can simply return the value stored in the thread-local data. For a `Sum` accumulator, we can split the thread-local `sum` variable into two variables, say `sumPos` and `sumNeg`; change the `Store` method to add positive data values to `sumPos` and negative ones to `sumNeg`; and change the `Result` method to return the sum of `sumPos` and `sumNeg`. With these changes, the `SelfNRV` method can return the sum of the *absolute values* of `sumPos` and `sumNeg`, an NRV value.

The library is modified as follows to support standalone implementations. At compile time, the library first checks to see if the specified accumulators are all independent, and *all* have standalone implementations specified. If this is not the case, normal compilation is used, and the optimization is not applied (because in this case we have to pay for the external NRV counters anyway).

If the accumulator set does satisfy the conditions for the optimization, a constant `allStandAlone` is set to specify that all accumulators in `allAccSeq` are standalone, and the `allImpSeq` sequence is constructed with the standalone implementations instead of the regular ones. In this case, a constant `NoNRV` is also set if none of the standalone implementations has a `SelfNRV` method.

The NRV counter is no longer needed if `allStandAlone` is set. Moreover, if `NoNRV` is set, `BeginStore` also avoids checking the anti-starvation flag, because in that case all accumulators' implementations are self synchronized, and `GetResult` does not iterate over the thread local data. (Depending on the synchronization used, similar mechanisms may be needed to facilitate progress, in which case they can be included with the self synchronization code.) Note that these tests are all based on compile-time constants, so they do not impose runtime overhead.

If the standalone implementation of an accumulator `Acc` has no `SelfNRV` method, `GetResult<Acc>` simply calls the `Result` method of the accumulator implementation (because it must be self synchronized). Otherwise, it iterates over all threads' TLDs, as in lines 32–47 of Figure 3, but instead of snapshotting the NRV counters, it snapshots the result of the `SelfNRV` method.

Finally, it is possible in some cases to combine the thread local data in a single pass, even while it is changing due to concurrent `Stores`, and still provide a consistent `Result` method. An example is a `Count` accumulator that is only incremented by one; in this case the sum obtained by a single pass is the correct sum of the thread-local counters at some point during the scan. To see why, note that because all thread-local counters are always increasing, the resulting sum must be between the sum of these counters at the

point the first of them is read, and their sum at the point the last is read. Because the counter can only be incremented by one, it must have taken any intermediate values between these two sums, and it is therefore correct to return the value obtained by the scan. The single-pass optimization can be achieved simply by providing a `SelfNRV` method that always returns zero (the initial value), and thus never causes the values to be scanned a second time.

We note that these optimizations are not all compatible with each other in all cases, and in particular that some apply only to accumulators that do not depend on others and/or that do not have others depending on them. Our approach allows the framework to determine which optimizations are applicable in which cases, without limiting the generality of the solution.

### 3.5 Using arbitrary external accumulators

We have described how our framework supports parallel implementations of associative and commutative accumulation functions, and how it can compose them and evaluate several accumulation functions together atomically. Some accumulation functions, however, may not be able to benefit from the parallel evaluation mechanism provided by the library, for example because they are not associative and/or commutative, or because they cannot easily be computed by having each thread access only a small portion of the data. Examples include the Boost median and rolling-mean accumulators, which compute the median of the samples, and the mean of the last `N` samples, respectively.

While our framework does not provide the means to parallelize the evaluation of such functions, it is not difficult to extend it to allow *external* parallel implementations to be “plugged in”. An external accumulator implementation provides thread-safe `Store` and `Result` operations, say using locks. While such accumulators do not rely on the framework to support their desired functionality, the framework can still allow other accumulators to depend on them and use their results. For example, given a thread safe implementation of rolling-mean, one can write an accumulator that computes the ratio between the overall mean and the rolling mean, knowing that the two values provided to that accumulator's `Result` method are consistent with each other.

The key observation enabling this powerful extension is that our algorithm guarantees that the snapshot of all thread-local data, obtained by the multiple passes done by `GetResult`, is consistent with the state of the memory between the read of the last NRV counter in the second-to-last pass, and the read of the first NRV counter in the last pass. Thus, calling the `Result` method of an external accumulator during this interval is guaranteed to return a result that is consistent with those of the other accumulators evaluated by the library. Note that in practice we do not know when a pass is going to be the last, but we can optimistically call the external `Result` method at the beginning of each pass (before reading the first NRV counter), and only use the result obtained in the last pass. Finally, if the library's `Store` method calls all external accumulators' `Store` methods one after another between the `BeginStore` and `EndStore` calls, then it is also guaranteed that the results obtained from different external accumulators are consistent with each other (because all NRV counters are guaranteed to have an even value that is not changed during that interval). Therefore it is safe to have an accumulator that depends on any number of external accumulators, as well as on accumulators evaluated by the library.

## 4. EVALUATION

We have evaluated our framework on a 64-core Niagara 2 system running at 1167 MHz using accumulators for `Min`, `Count`, `Sum`, `Mean`, and `Variance`. While `Min`, `Count` and `Sum` have no de-

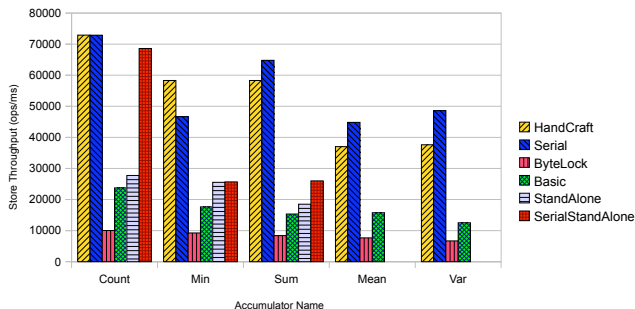


Figure 4: Single-threaded Store throughput.

dependencies, both Mean and Variance depend on Count, and on another accumulator  $\text{Moment}\langle K \rangle$ , which computes the  $K$ th moment of the samples. For the accumulators with no dependencies, we also provide standalone implementations: self-synchronized for Min (using CAS), and `SelfNRV` for Count and Sum.

We compare basic and standalone implementations using our framework, as well as using a lightweight readers-writer lock based on the bytelock technique [4]. The readers-writer lock is used such that Store operations are “readers” and a `GetResult` operation is a “writer”, because multiple concurrent Stores are allowed, while our current implementation supports only one `GetResult` at a time. To optimize for Store performance, our bytelock implementation allocates enough slots so that a “reader” will never need to use a CAS, only a store-load memory barrier, to acquire the lock.

In our experiments, a number of Store threads repeatedly store a value chosen using a lightweight thread-local random number generator, and we measure Store throughput; in some experiments, an additional thread executes `GetResult` operations periodically.

We first focus on single thread performance with an experiment that has only one Store thread. In addition to the bytelock version, and basic and standalone versions of our `ParAccSet` implementations, we added “Serial” versions of the basic and standalone accumulators, for which we specified 1 for the optional `MaxThreads` parameter, thereby enabling single-threaded optimizations as discussed in Section 3.4. For comparison, we also directly implemented simple “HandCraft” code for all accumulators, which does not support concurrent operations.

Figure 4 groups results by accumulator type. For all accumulators, all concurrent solutions entail significant overhead relative to the single-threaded versions; using a bytelock (`ByteLock`) results in substantially worse throughput than any of our methods. For example, with Count, our Basic implementation (without standalone or single-thread optimizations) achieves more than 2x the throughput of `ByteLock`; for Variance the factor is slightly less than 2. The standalone optimization (`StandAlone`), where applicable, widens the gap even further. The standalone version of Min provides the greatest improvement relative to the basic version because it is the only one that avoids the anti-starvation flag test (as it is fully self-synchronized, while the others use the `SelfNRV` method). This gap emphasizes how low the overhead of our solutions is; even with a single thread, a lightweight solution such as a bytelock—which does not perform any CAS in Store operations—achieves less than half of the throughput of our basic solution.

Going one step further, simply by specifying `MaxThreads=1`, our implementations get substantially faster still. For example, with Count, the serial version (`Serial`) is 3x faster than the basic version, and is competitive with the handcrafted solution. One of the main contributors to the large performance gap between Basic

and `Serial` is the additional level of indirection required to access the thread local data in a multithreaded solution. We believe that the reason `Serial` slightly outperforms the handcrafted versions in some cases is an artifact of different compiler optimizations being applied in the two cases, and not an algorithmic difference.

We observe that the `SerialStandAlone` variants typically perform worse than the `Serial` ones. This is because they add overhead in order to support optimizations that are beneficial only when multiple threads are supported (Sum separates the thread local sum into two variables, and Min uses CAS to modify the shared min variable). Count does not require any additional work to use the `SelfNRV` method, and thus it is competitive with the `Serial` variant. It is easy to disable the use of the standalone versions for serial sets without imposing additional cost.

Next, we explore the scalability of our solution with more Store threads (still without any `GetResult` operations). Results are shown in Figure 5 (top), on a log-log graph. Results for Mean are omitted as they were qualitatively similar to those for Variance.

For all accumulators, the bytelock solutions are slower than all others and scale up to only 4 threads. At 63 threads `ByteLock-Count` is 7x slower than `Basic-Count`. The improvement at 32 threads is because the slots in the byte lock happen to span two cache lines, so we get a slight increase in Store throughput when “readers” (Store operations) start using slots in the second one.

In contrast, both the basic and standalone implementations of all accumulators scale well with our framework up to 32 threads. Because of our successful efforts to avoid memory contention by parallelizing the accumulators, we have essentially made the workload compute-bound. Such workloads often do not scale well beyond 32 cores on this system because pairs of cores share a single pipeline. Therefore, we believe that the lack of continued scaling past 32 threads is an artifact of the system, not the algorithms.

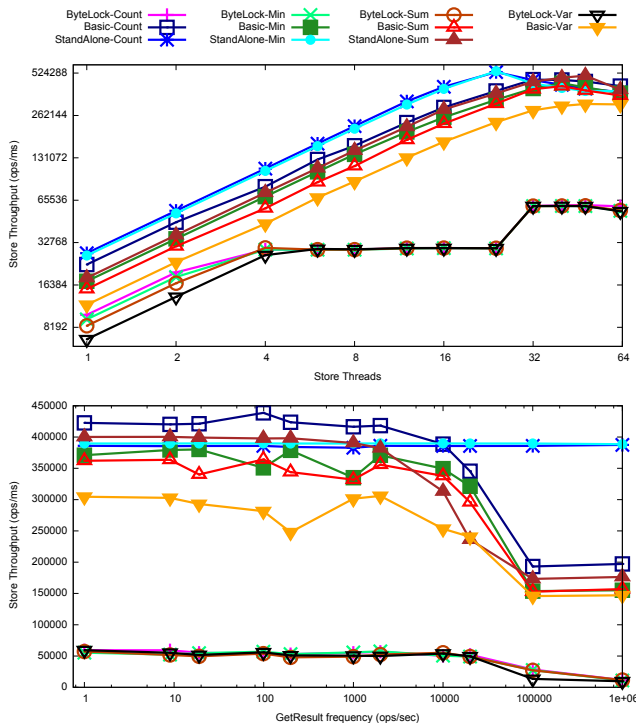
The standalone implementations achieve higher throughput than their non-standalone counterparts. The most significant improvement is again achieved by `StandAlone-Min`, as it is the only one that avoids both the NRV counter increment and testing the anti-starvation flag. `StandAlone-Min` is up to 1.6x faster than `Basic-Min`, whereas `StandAlone-Count` and `StandAlone-Sum` are only up to 1.4x faster than `Basic-Count` and `Basic-Sum`, respectively.

Finally, to evaluate the impact of concurrent `GetResult` operations on Store performance, we added a thread that periodically performed `GetResults`. Figure 5 (bottom) shows results for 63 Store threads; the  $x$  axis shows target frequency<sup>3</sup> for `GetResult` operations on a log scale.

All solutions are mostly unaffected by the concurrent `GetResult` operations until their frequency exceeds 2000 operations per second; this is consistent with our design goal to optimize Store performance when `GetResult` operations are infrequent.

As expected, the Store throughput of `StandAlone-Min` remains unaffected, even when `GetResult` operations become frequent, because the Store operation of this fully self-synchronized solution never checks the anti-starvation flag. Interestingly, `StandAlone-Count` is also immune to the concurrent `GetResult` operations; this is because of the one-pass optimization for standalone Count (Section 3.4), and the fact that our `GetResult` implementation only sets the anti-starvation flag after it completes a first pass over all threads’ local data (recall that the anti-starvation flag is not required for correctness, it merely facilitates progress). With the one-

<sup>3</sup> The thread performing `GetResult` operations aims to execute them  $x$  times per second. However, when the target frequency becomes high enough, this may not be possible because one operation may not complete before the next is supposed to begin, in which case it begins immediately after the previous one ends.



**Figure 5: Scalability results. Top: without GetResult operations. Bottom: 63 threads performing Stores, 1 performing GetResults.**

pass optimization of StandAlone-Count, a second pass is never required, so the anti-starvation flag is never set.

In contrast, the basic solutions do degrade due to the frequent changes of the anti-starvation flag when GetResult frequency increases, and the Store throughput for all accumulators decreases by approximately a factor of two when GetResult is at its maximum frequency. With the bytelock solutions, the Store throughput degrades by more than a factor of five.

## 5. CONCLUDING REMARKS

We have explored how the Boost.Accumulators framework can be extended to provide *parallel* accumulators. Many of its functions can be parallelized using our framework without programmers writing any synchronization code; others can be supported with the use of custom synchronization. Our evaluation shows that our parallel accumulators are scalable and can exploit programmer-supplied knowledge to effectively optimize some important cases.

The framework we have described is similar in some ways to the Hyperobjects of [6]. Hyperobjects were developed for use in Cilk programs. Cilk [10] parallelizes a sequential program, and must therefore impose an order on the aggregation of data in order to preserve sequential semantics. As a result, it requires only associativity (and not commutativity) for its summary functions, and it does not support incremental evaluation of the summary function. Intel’s TBB reducers [12] similarly do not require commutativity, but do not support incremental evaluation.

In contrast, while our framework requires commutativity for basic accumulators, it does support concurrent and incremental evaluation of the summary function, composition of accumulators, and allows for the use of functions that are neither associative nor commutative by using custom synchronization code.

## Acknowledgments:

We are grateful to Dave Abrahams and Matthias Troyer for useful conversations, and to the anonymous referees for valuable feedback.

## References

- [1] D. Abrahams et al. Boost.type\_traits. [www.boost.org/doc/libs/release/libs/type\\_traits](http://www.boost.org/doc/libs/release/libs/type_traits), 2006.
- [2] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.
- [3] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *J. ACM*, 40:873–890, September 1993.
- [4] D. Dice and N. Shavit. Tlrw: Return of the read-write lock. In *Proc. 4th ACM SIGPLAN Workshop on Transactional Computing*, 2009. [labs.oracle.com/scalable/pubs/TLRW-SPAA-2010.pdf](http://labs.oracle.com/scalable/pubs/TLRW-SPAA-2010.pdf).
- [5] F. Ellen, Y. Lev, V. Luchangco, and M. Moir. SNZI: Scalable NonZero Indicators. In *26th ACM Symp. on Principles of Distributed Computing*, Aug. 2007.
- [6] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and other cilk++ hyperobjects. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA ’09, pages 79–90, New York, NY, USA, 2009. ACM.
- [7] A. Gurtovoy and D. Abrahams. The Boost MPL library. [www.boost.org/doc/libs/release/libs/mpl/doc/index.html](http://www.boost.org/doc/libs/release/libs/mpl/doc/index.html), 2004.
- [8] Y. Lev, V. Luchangco, V. J. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Anatomy of a Scalable Software Transactional Memory. In *Proc. 4th ACM SIGPLAN Workshop on Transactional Computing*, 2009. [labs.oracle.com/scalable/pubs/TRANSACT2009-ScalableSTMAatomy.pdf](http://labs.oracle.com/scalable/pubs/TRANSACT2009-ScalableSTMAatomy.pdf).
- [9] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9:21–65, February 1991.
- [10] MIT CSAIL Supertech Research Group. The Cilk project. <http://supertech.csail.mit.edu/cilk/>, 2010.
- [11] E. Niebler. Boost.accumulators. [www.boost.org/doc/libs/release/libs/accumulators/index.html](http://www.boost.org/doc/libs/release/libs/accumulators/index.html), 2006.
- [12] J. Reinders. *Intel threading building blocks*. O’Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.
- [13] M. Troyer, B. Ammon, and E. Heeb. Parallel object oriented monte carlo simulations. In *Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments*, ISCOPE ’98, pages 191–198, London, UK, 1998. Springer-Verlag.