Lightweight, Composable, Parallel Accumulators

Yossi Lev
Oracle Labs

Joint work with:

Mark Moir
Oracle Labs
Accumulators

• Incremental calculation of a function over samples

• Example: Sum, Mean, STDV, Min, Max

• Operations:
  • **Store**: push the data
  • **Result**: returns value calculated so far
Boost.Accumulators

- Generic accumulators framework
- **Accumulator Set:** calculates a set of functions over samples
• Generic accumulators framework
• Accumulator Set: calculates a set of functions over samples
• Example:

```cpp
AccSet<double, tags_seq<Mean, STDV, Min, Max> > accs;
accs.store(2.3);
accs.store(4.2);
...
cout << accs.GetResult<Mean>() << endl;
cout << accs.GetResult<STDV>() << endl;
```
• Generic accumulators framework

• **Accumulator Set:** calculates a set of functions over samples

• **Example:**

```cpp
AccSet<double, tags_seq<Mean, STDV, Min, Max> > accs;
accs.store(2.3);
accs.store(4.2);
...
cout << accs.GetResult<Mean>() << endl;
cout << accs.GetResult<STDV>() << endl;
```

• Library provides code for many important statistical functions

• **Extendable:** Allows the programmer to provide code for additional functions.
## Dependencies

<table>
<thead>
<tr>
<th>Sum</th>
<th>Count</th>
</tr>
</thead>
</table>

Wednesday, May 18, 2011
Dependencies

- Sum
- Count
- Mean
Dependencies

\[ \text{Mean} = \frac{\text{Sum}}{\text{Count}} \]
Dependencies

- To provide implementation for Mean
  - Specify dependency on Sum and Count

\[
\text{Mean} = \frac{\text{Sum}}{\text{Count}}
\]
Dependencies

- To provide implementation for Mean
- Specify dependency on Sum and Count
- Empty Store operation

```cpp
MeanImp::Store(sampleType s) {};
```

- Result uses the values that it depends on:

```cpp
resultType MeanImp::Result() {
    return GetDepResult<Sum>() / GetDepResult<Count>();
}
```
• When we specify:

```
AccSet<double, tags_seq<Mean, STDV> > acce;
```
Dependencies

• When we specify:

\[
\text{AccSet}\langle\text{double, tags\_seq}\langle\text{Mean, Count, Sum, STDV}\rangle\rangle > \text{accs};
\]
Dependencies

• When we specify:

    AccSet<double, tags_seq<Mean, Count, Sum, STDV> > accs;

• Boost automatically pulls in dependent accumulators
Dependencies

• When we specify:
  \[
  \text{AccSet}\langle\text{double}, \text{tags}\_\text{seq}\langle\text{Mean}, \text{Count}, \text{Sum}, \text{STDV}\rangle\rangle > \text{accs};
  \]

• Boost automatically pulls in dependent accumulators

• **Share** any components between accumulators
Dependencies

• When we specify:
  
  \[
  \text{AccSet<double, tags_seq\langle Mean, Count, Sum, STDV\rangle > accs;}
  \]

• Boost automatically pulls in dependent accumulators

• \textbf{Share} any components between accumulators

• Significantly simplifies extending the library

• Extremely efficient: all done at \textit{compile time}
Parallel Accumulators

- Associative and Commutative functions
- Goals:
  - Store samples in parallel efficiently
Parallel Accumulators

• Associative and Commutative functions

• Goals:
  • Store samples in parallel efficiently
  • Concurrent Result and Store ops
    
    minimal overhead for Store when Result is not running
Parallel Accumulators

• Associative and Commutative functions

• Goals:

  • Store samples in parallel efficiently

  • Concurrent Result and Store ops
    minimal overhead for Store when Result is not running

• Dependencies:

  Parallel Sum + Parallel Count → Parallel Mean
Parallel Accumulators

• Associative and Commutative functions

• Goals:
  • Store samples in parallel efficiently
  • Concurrent Result and Store ops
    minimal overhead for Store when Result is not running

• Dependencies:

  Parallel Sum + Parallel Count → Parallel Mean

composability: atomic{ Result<Sum>() / Result<Count>(); }
Parallel Accumulators

• Associative and Commutative functions

• Goals:
  • Store samples in parallel efficiently
  • Concurrent Result and Store ops
    minimal overhead for Store when Result is not running

• Dependencies:
  Parallel Sum + Parallel Count → Parallel Mean

**composability:** atomic{ Result<Sum>() / Result<Count>(); }
Previous Work: Store Samples In Parallel

- Intel’s TBB, Cilk++ Reducers
  - **Data parallel solution** each thread aggregates the samples it stores, combine aggregated data at end

- **Simple Interface:** only requires providing an additional Combine method.

- **Our contribution**
  - Concurrent Composable Result
  - Keep simple interface
  - Minimal overhead for Store operations
Previous Work: Store Samples In Parallel

- Intel’s TBB, Cilk++ Reducers

- **Data parallel solution**
  each thread aggregates the samples it stores, combine aggregated data at end

- **Simple Interface:** only requires providing an additional Combine method.

- Our contribution
  - Concurrent Composable Result
  - Keep simple interface
  - Minimal overhead for Store operations
Previous Work: Store Samples In Parallel

- Intel’s TBB, Cilk++ Reducers
  - Data parallel solution
    each thread aggregates the samples it stores, combine aggregated data at end
  - Simple Interface: only requires providing an additional Combine method.

- Our contribution
  - Concurrent Composable Result
  - Keep simple interface
  - Minimal overhead for Store operations
RW Lock Solution
(Straw man solution)

- **Mutual Exclusion** between Result and Store ops
- **Store** operations can run concurrently
- **Result** cannot run concurrently with Store
RW Lock Solution
(Straw man solution)

- Mutual Exclusion between Result and Store ops
- Store operations can run concurrently
  - Acquire lock in read mode
- Result cannot run concurrently with Store
RW Lock Solution
(Straw man solution)

• Mutual Exclusion between Result and Store ops
  • Store operations can run concurrently
    ➔ Acquire lock in read mode
  • Result cannot run concurrently with Store
    ➔ Acquire lock in write mode
RW Lock Solution
(Straw man solution)

- Mutual Exclusion between Result and Store ops
- Store operations can run concurrently
  - Acquire lock in read mode
- Result cannot run concurrently with Store
  - Acquire lock in write mode

High overhead for Store operations
NRV Based Synchronization

● **NRV**: Non Repeated Value variables: Never changes to a value it had before

● Value aggregated for many accumulator functions is NRV

● Examples: Count, Min, Max, Sum<unsigned>
NRV Based Synchronization

- **NRV**: Non Repeated Value variables: Never changes to a value it had before
- Value aggregated for many accumulator functions is NRV
  - Examples: Count, Min, Max, Sum<unsigned>
- Atomic snapshot of any number of NRVs
  - Scan repeatedly until two successive scans yield the same values
Solution for Single NRV Accumulator

Per Thread

Min +inf

Min 15

Min 5
Solution for Single NRV Accumulator

Store:

Min
+inf

Min
15

Min
5
Solution for Single NRV Accumulator

Store:

Per Thread

Min 10

Min 8

Min 5
Solution for Single NRV Accumulator

Result:

Per Thread

Min 10

Min 8

Min 5
Solution for Single NRV Accumulator

Result:

<table>
<thead>
<tr>
<th>Min</th>
<th>10</th>
<th>. . .</th>
<th>Min</th>
<th>8</th>
<th>. . .</th>
<th>Min</th>
<th>5</th>
</tr>
</thead>
</table>

Per Thread
Solution for Single NRV Accumulator

Result:

Per Thread

Min 10

Min 8

Min 5

==?

10 • • • 8 • • • • 5

Copyright 2011 Oracle Labs

Wednesday, May 18, 2011
Solution for Single NRV Accumulator

Result:

10  •  •  •  •  8  •  •  •  •  5
Solution for Single NRV Accumulator

Result:

Per Thread

Min 10
Min 8
Min 5

Combine: 5
Solution for Single NRV Accumulator

TLD: thread local data for the aggregation

Store(Sample s) {
    Update TLD;
}

Result() {
    aggData = Combine(GetSnapshot(TLDs));
    return CalculateResult(aggData);
}
Solution for Single NRV Accumulator

TLD: thread local data for the aggregation

Store(Sample s) {
    Update TLD;
}

Result() {
    aggData = Combine(GetSnapshot(TLDs));
    return CalculateResult(aggData);
}

Starvation???
Solution for Single NRV Accumulator

Handling Starvation: anti starvation flag

TLD: thread local data for the aggregation

Store(Sample s) {
    while (ReadingResult);
    Update TLD;
}

Result() {
    ReadingResult = true;
    aggData = Combine(GetSnapshot(TLDs));
    ReadingResult = false;
    return CalculateResult(aggData);
}
Solution for Single NRV Accumulator

Handling Starvation: anti starvation flag

TLD: thread local data for the aggregation

```java
Solution for Single NRV Accumulator

Handling Starvation: anti starvation flag

TLD: thread local data for the aggregation

```java
Store(Sample s) {
    while (ReadingResult);
    Update TLD;
}

Result() {
    ReadingResult = true;
    aggData = Combine(GetSnapshot(TLDs));
    ReadingResult = false;
    return CalculateResult(aggData);
}

• Similar to mutual exclusion but...

• We **ask, don’t tell**, Store not to run in parallel
  • Read/Write races are handled correctly because read data is NRV

  ➡ Avoid expensive synchronization primitives (CAS, memory barrier)

Copyright 2011 Oracle Labs

Wednesday, May 18, 2011
Solution for Single NRV Accumulator

Handling Starvation: anti starvation flag

TLD: thread local data for the aggregation

Store(Sample s) {
    while (ReadingResult);
    Update TLD;
}

Result() {
    ReadingResult = true;
    aggData = Combine(GetSnapshot(TLDs));
    ReadingResult = false;
    return CalculateResult(aggData);
}

- Similar to mutual exclusion but...
- We **ask, don't tell**, Store not to run in parallel
  - Read/Write races are handled correctly because read data is NRV
  - Avoid expensive synchronization primitives (CAS, memory barrier)
General Solution

Per Thread

Count 10  Sum 45  Count 12  Sum 42
General Solution

Separate NRV from data:

Per Thread

<table>
<thead>
<tr>
<th>Ver</th>
<th>Count</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>10</td>
<td>45</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ver</th>
<th>Count</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>12</td>
<td>42</td>
</tr>
</tbody>
</table>

Copyright 2011 Oracle Labs

Wednesday, May 18, 2011
General Solution

Separate NRV from data:

Store: increment ver before and after updating TLD

Per Thread

<table>
<thead>
<tr>
<th>Ver</th>
<th>Count</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>10</td>
<td>45</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ver</th>
<th>Count</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>12</td>
<td>42</td>
</tr>
</tbody>
</table>
General Solution

Separate NRV from data:

Store: increment ver before and after updating TLD
Result: snapshot ver + TLD, spin on odd version
General Solution

Separate NRV from data:

**Store:** increment ver before and after updating TLD

**Result:** snapshot ver + TLD, spin on odd version

Per Thread

<table>
<thead>
<tr>
<th>Ver</th>
<th>Count</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>10</td>
<td>45</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ver</th>
<th>Count</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>12</td>
<td>42</td>
</tr>
</tbody>
</table>

20

20
General Solution

Separate NRV from data:

**Store:** increment ver before and after updating TLD

**Result:** snapshot ver + TLD, spin on odd version

Per Thread

<table>
<thead>
<tr>
<th>Ver</th>
<th>Count</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>10</td>
<td>45</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ver</th>
<th>Count</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>12</td>
<td>42</td>
</tr>
</tbody>
</table>

Wednesday, May 18, 2011
General Solution

Separate NRV from data:

Store: increment ver before and after updating TLD

Result: snapshot ver + TLD, spin on odd version

Per Thread

Ver 20
Count 10
Sum 45

Ver 24
Count 12
Sum 42

20 10 45

24 12 42
General Solution

Separate NRV from data:

**Store**: increment ver before and after updating TLD

**Result**: snapshot ver + TLD, spin on odd version

---

Per Thread

<table>
<thead>
<tr>
<th>Ver</th>
<th>Count</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>10</td>
<td>45</td>
</tr>
</tbody>
</table>

==?  

<table>
<thead>
<tr>
<th>Ver</th>
<th>Count</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>12</td>
<td>42</td>
</tr>
</tbody>
</table>

==?

---

Also takes care of composability:
we never see a partial effect of Store
Implementation: ParallelAccSet
Implementation: ParallelAccSet

• Like the sequential AccSet of Boost.Accumulators
  
  • **Extendable:** programmer specifies how to store samples and how to calculate the result.
  
  • **Supports dependencies:** can implement one accumulator using others, common components are shared.

• A lot is done at **compile time**.
Implementation: ParallelAccSet

- Like the sequential AccSet of Boost.Accumulators
  - **Extendable:** programmer specifies how to store samples and how to calculate the result.
  - **Supports dependencies:** can implement one accumulator using others, common components are shared.
  - A lot is done at **compile time**.
- Similar to TBB/Reducers, programmer also provides Combine method
- **Supports composable concurrent Result with no interface overhead.**
Optimizations

- Store overhead:
  - Incrementing NRV version twice
  - Checking anti-starvation flag
- May not be negligible for very simple accumulators like Count
Optimizations

- Store overhead:
  - Incrementing NRV version twice
  - Checking anti-starvation flag
- May not be negligible for very simple accumulators like Count
- Can avoid a separate NRV version if:
  - All accumulators in set are of NRV type, and
  - None are dependent on others
- **Example:** ParallelAccSet<Min, Max, Count>
Optimizations

- Store overhead:
  - Incrementing NRV version twice
  - Checking anti-starvation flag
- May not be negligible for very simple accumulators like Count
- Can avoid a separate NRV version if:
  - All accumulators in set are of NRV type, and
  - None are dependent on others
- Example: ParallelAccSet<Min, Max, Count>

  Programmer specifies which accumulators are NRV, Library tracks dependencies, and eliminates NRV version if possible.
Optimizations

• Store overhead:
  • Incrementing NRV version twice
  • Checking anti-starvation flag
• May not be negligible for very simple accumulators like Count
• Can avoid a separate NRV version if:
  • All accumulators in set are of NRV type, and
  • None are dependent on others
• Example: ParallelAccSet<Min, Max, Count>

  Programmer specifies which accumulators are NRV,
  Library tracks dependencies,
  and eliminates NRV version if possible.
• At set level: Serial mode (at compile time)
Plugging in External Accumulators

- So far: lightweight composable solution for associative & commutative accumulators
Plugging in External Accumulators

- So far: lightweight composable solution for associative & commutative accumulators
- Given an external parallel accumulator that is not necessarily associative/commutative
  - E.g. Median
Plugging in External Accumulators

• So far: lightweight composable solution for associative & commutative accumulators

• Given an *external parallel accumulator* that is not necessarily associative/commutative
  
  • E.g. Median

• Can we “plug it into” our framework, have other accumulators depend on it and share it?
  
  • E.g. compute the ratio between median and mean?
Plugging in External Accumulators

• So far: lightweight composable solution for associative & commutative accumulators

• Given an external parallel accumulator that is not necessarily associative/commutative
  • E.g. Median

• Can we “plug it into” our framework, have other accumulators depend on it and share it?
  • E.g. compute the ratio between median and mean?

➤ Requires composability with external atomic Store and Result operations
Plugging in External Accumulators

- So far: lightweight composable solution for associative & commutative accumulators
- Given an external parallel accumulator that is not necessarily associative/commutative
  - E.g. Median
- Can we “plug it into” our framework, have other accumulators depend on it and share it?
  - E.g. Compute the ratio between median and mean?
- Requires composability with external atomic Store and Result operations

Copyright 2011 Oracle Labs

Wednesday, May 18, 2011
Plugging in External Accumulators

- Store calls external accumulator atomic Store
- Result calls external atomic Result between passes
Plugging in External Accumulators

- Store calls external accumulator atomic Store
- Result calls external atomic Result between passes

Per Thread:

- Ver: 20
- Count: 10
- Sum: 45

- Ver: 24
- Count: 12
- Sum: 42
Plugging in External Accumulators

- Store calls external accumulator atomic Store
- Result calls external atomic Result between passes

Per Thread

<table>
<thead>
<tr>
<th>Ver</th>
<th>Count</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>10</td>
<td>45</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ver</th>
<th>Count</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>12</td>
<td>42</td>
</tr>
</tbody>
</table>

No TLD has changed between end of one before last scan, and beginning of last scan.

Copyright 2011 Oracle Labs
Wednesday, May 18, 2011
Plugging in External Accumulators

- Store calls external accumulator atomic Store
- Result calls external atomic Result between passes

Per Thread

<table>
<thead>
<tr>
<th></th>
<th>Count</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ver</td>
<td>20</td>
<td>45</td>
</tr>
<tr>
<td>Count</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Sum</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Count</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ver</td>
<td>24</td>
<td>42</td>
</tr>
<tr>
<td>Count</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>Sum</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

No TLD has changed between end of one before last scan, and beginning of last scan.
Plugging in External Accumulators

• Store calls external accumulator atomic Store
  While storing thread's version number is odd (together with TLD update)
• Result calls external atomic Result between passes

Per Thread

<table>
<thead>
<tr>
<th>Ver</th>
<th>Count</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>10</td>
<td>45</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ver</th>
<th>Count</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>12</td>
<td>42</td>
</tr>
</tbody>
</table>

No TLD has changed between **end of one before last scan**, and **beginning of last scan**
Plugging in External Accumulators

- Store calls external accumulator atomic Store while storing thread’s version number is odd (together with TLD update)
- Result calls external atomic Result between passes

Per Thread

<table>
<thead>
<tr>
<th>Ver</th>
<th>Count</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>10</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>12</td>
<td>42</td>
</tr>
</tbody>
</table>

20 10 45 • • • 24 12 42

56 54

No TLD has changed between end of one before last scan, and beginning of last scan

Copyright 2011 Oracle Labs

Wednesday, May 18, 2011
Plugging in External Accumulators

- Store calls external accumulator atomic Store while storing thread’s version number is odd (together with TLD update).
- Result calls external atomic Result between passes.

- Allows combining any number of external and internal accumulators.
- Evaluate them atomically together.

No TLD has changed between end of one before last scan, and beginning of last scan.

Copyright 2011 Oracle Labs

Wednesday, May 18, 2011
• Implementations
  • ParallelAccSet
    • Basic
    • Serial: set instantiated for a single thread serial execution
    • Standalone (for some): optimized when all accumulators in set are independent
Evaluation

- Implementations
  - **ParallelAccSet**
    - **Basic**
    - **Serial**: set instantiated for a single thread serial execution
    - **Standalone** (for some): optimized when all accumulators in set are independent
  - **Mutual exclusion**: **ByteLock**
  - **Handcrafted** (serial)

Copyright 2011 Oracle Labs

Wednesday, May 18, 2011
Evaluation

- Implementations
  - ParallelAccSet
    - Basic
    - Serial: set instantiated for a single thread serial execution
    - Standalone (for some): optimized when all accumulators in set are independent
  - Mutual exclusion: ByteLock
  - Handcrafted (serial)

- Platform: 64-core Niagara II
Single Thread, Store only

Accumulator Name

Store Throughput (ops/ms)

- Count
- Min
- Sum
- Mean
- Var

Support parallelism

- HandCraft
- Serial
- ByteLock
- Basic
- StandAlone

Wednesday, May 18, 2011
Effect of Concurrent Result

- One thread calls Result periodically, 63 repeatedly run Store
- Measure Store throughput as Result frequency increases
Effect of Concurrent Result

- One thread calls Result periodically, 63 repeatedly run Store
- Measure Store throughput as Result frequency increases

Figure 5: Scalability results. Top: without GetResult operations. Bottom: 63 threads performing GetResult operations.
Effect of Concurrent Result

- One thread calls Result periodically, 63 repeatedly run Store
- Measure Store throughput as Result frequency increases
Summary

• Extendable framework for lightweight parallel accumulators
• Efficient Store operation
• Concurrent composable Result operation
• Allows using and composing external parallel accumulators