N4283: Atomic View (Revision to N4142, Atomic Operations on a Very Large Array)

Carter Edwards hcedwar@sandia.gov
Hans Boehm hboehm@google.com
2015-01-26 (version 2)

1. Introduction

This paper proposes an extension to the atomic operations library [atomics] for atomic operations applied to non-atomic objects. The proposal is in three parts: (1) the concept of an atomic view, (2) application of this concept for atomic operations applied to members of a very large array in a high performance computing (HPC) code, and (3) application of this concept for atomic operations applied to an object in a large legacy code which cannot replace this object with a corresponding atomic<T> object.

```
namespace std {
namespace experimental {
  template< class T > atomic_array_view ;
  template< class T > atomic_concurrent_view ;
  template< class T > atomic_concurrent_view<T>
    make_atomic_concurrent_view( T & );
}
}
```

2. Atomic View Concept

A class conforming to the atomic view concept provides atomic operations for a referenced non-atomic object. An atomic view does not own (neither exclusively nor shared) the non-atomic object for which it provides atomic operations. Three groups of atomic operations are defined: (1) for any type, (2) additional operations for an integral type, and (3) additional operations for a floating point type.

A class conforming to the atomic view concept provides the following operations for all types.

```
template< class T > struct atomic-view-concept {

  bool is_lock_free() const noexcept ;

  void store( T , memory_order = memory_order_seq_cst ) const noexcept ;
  T load( memory_order = memory_order_seq_cst ) const noexcept ;
  T exchange( T , memory_order = memory_order_seq_cst ) const noexcept ;
  bool compare_exchange_weak( T & , T , memory_order , memory_order ) const
noexcept ;
  bool compare_exchange_strong( T & , T , memory_order , memory_order ) const
noexcept ;
  bool compare_exchange_weak( T & , T , memory_order = memory_order_seq_cst ) const
noexcept ;
  bool compare_exchange_strong( T & , T , memory_order = memory_order_seq_cst )
const noexcept ;

  // For compatibility with atomic<T>
  operator T() const noexcept ;
  T operator=(T) const noexcept ;
};
```

A class conforming to the atomic view concept shall also provide the following operations when T is an integral type.

```
template<> struct atomic-view-concept < integral > {

  integral fetch_add( integral , memory_order = memory_order_seq_cst) const
noexcept;
  integral fetch_sub( integral , memory_order = memory_order_seq_cst) const
noexcept;
  integral fetch_and( integral , memory_order = memory_order_seq_cst) const
noexcept;
  integral fetch_or(  integral , memory_order = memory_order_seq_cst) const
noexcept;
  integral fetch_xor( integral , memory_order = memory_order_seq_cst) const
noexcept;

  integral operator++(int) const noexcept;
  integral operator--(int) const noexcept;
  integral operator++() const noexcept;
  integral operator--() const noexcept;
  integral operator+=( integral ) const noexcept;
  integral operator-=( integral ) const noexcept;
  integral operator&=( integral ) const noexcept;
  integral operator|=( integral ) const noexcept;
  integral operator^=( integral ) const noexcept;
};
```

A class conforming to the atomic view concept shall also provide the following operations when T is a floating point (*fp*) type. This capability is critical for high performance computing (HPC) applications and domain libraries.

```
template<> struct atomic-view-concept < fp > {
  fp fetch_add(fp , memory_order = memory_order_seq_cst) const noexcept;
  fp operator+=( fp ) const noexcept;
};
```

3. Atomic View for a Very Large Array

HPC applications and domain libraries allocate very large arrays. Computations with these arrays typically have distinct phases that allocate and initialize members of the array, update members of the array, and read members of the array. Parallel algorithms for initialization (e.g., zero fill) have non-conflicting access when assigning member values. Parallel algorithms for updates have conflicting access to members which must be guarded by atomic operations. Parallel algorithms with read-only access require best-performing streaming read access, random read access, vectorization, or other guaranteed non-conflicting HPC pattern.

Usage Scenario

a) A very large array of trivially copyable members is allocated.
b) A parallel algorithm initializes members through non-conflicting assignments.
c) The array is wrapped by an atomic_array_view<T>.
d) One or more parallel algorithms update members of the array through atomic view operations.
e) The atomic_array_view<T> is destructed.
f) Parallel algorithms access array members through non-conflicting reads, writes, or updates.

An atomic_array_view<T> object is used to perform atomic operations on the non-atomic members of an array. The intent is for atomic_array_view<T> to enable support for the best-performing implementations of atomic operations on members of the wrapped array.

```
template< class T > struct atomic_array_view {

  bool is_lock_free() const noexcept ;

  atomic_array_view( T * , size_t ) /* NOT noexcept */ ;
  atomic_array_view( atomic_array_view && ) noexcept ;
  atomic_array_view( atomic_array_view & ) noexcept ;
  ~atomic_array_view() noexcept ;

  atomic_array_view() = delete ;
  atomic_array_view( const atomic_array_view & ) = delete ;
  atomic_array_view & operator = ( const atomic_array_view & ) = delete ;

  size_t size() const noexcept ;

  typedef  implementation-defined-proxy-type  reference ;

  reference operator[]( size_t ) const noexcept ;
};
```

`atomic_array_view<T>::atomic_array_view( T * ptr , size_t N ) /* NOT noexcept */ ;`

This *wrapping constructor* wraps an array [ptr .. ptr+N) of trivially copyable type T. When a user wraps an array the user make the following guarantees as long as an atomic_array_view exists for that array. Violation of these usage guarantees results in undefined behavior.

1) Members of the wrapped array will not be accessed through any mechanism other than atomic_array_view; e.g., will not be directly accessed by dereferencing a pointer to the member as **`ptr[i]`**.
2) An overlapping array will not be concurrently wrapped by another atomic_array_view.
3) A member of the wrapped array will not be concurrently wrapped with an atomic_concurrent_view.

The wrapping constructor may acquire resources, such as locks, as necessary to support atomic operations on its members. The wrapping constructor is allowed to throw an exception; e.g., if auxiliary resources could not be acquired or if the array to be wrapped is not properly aligned for the underlying architecture's best-performing atomic operation capability. The wrapping constructor must throw an exception if atomic<T> is lock-free but element-operations on atomic_array_view<T> are not lock-free; e.g., due to improper alignment.

Implementation note: All non-atomic accesses of the wrapped object that appear before the wrapping constructor must happen before the wrapping constructor completes. For example:

```
void foo( int * i , int n ) {
  i[0] = 42 ;
  i[n-1] = 42 ;
  // The previous assignments must happen before
  // the following wrapper constructor completes.
  atomic_array_view<int> ai(i,n);
```

A wrapping constructor of the form (T*begin, T*end) would be valid. However, the (T*ptr, size_t N) version is preferred to minimize potential confusion with construction from non-contiguous iterators. Wrapping constructors for standard contiguous containers would also be valid. However, such constructors could have potential confusion as to whether the atomic_array_view would or would not track resizing operations applied to the input container.

`atomic_array_view<T>::atomic_array_view( atomic_array_view & ) noexcept ;`

This copy constructor creates an additional view to the original wrapped array. Atomic operations on an array member are atomic with respect to any other atomic operation performed to the same array member through a copy-constructed atomic_array_view. If a wrapping constructor acquired resources then those resources are shared by all copy-constructed atomic_array_view objects. The copy constructor allows atomic_array_view objects to be passed by value to concurrent functions or captured by value in concurrently executed lambdas.

`atomic_array_view<T>::~atomic_array_view() noexcept ;`

The destructor of the final copy of an atomic_array_view object releases resources which may have been acquired by the wrapping constructor. Outstanding atomic operations associated with the wrapped array must happen before destruction of the final copy of an atomic_array_view. After the final copy is destroyed the user is free to directly access members of the array.

Notes for implementers regarding wrapping construction and final destruction

Wrapping construction and final destruction of an atomic_array_view provides an opportunity to amortize the time and space overhead potentially associated with acquiring and releasing auxiliary resources on a per-member or per-member-per-access basis.

A poor-quality implementation could allocate an internal array of atomic<T> in the wrapper constructor, initialize this internal array from the source array, apply all atomic operations to this internal array, copy the internal array values to the source array upon final destruction, and then deallocate the internal array.

`atomic_array_view<T>::reference`
`atomic_array_view<T>::operator[]( size_t ) const noexcept ;`

The member access operator returns an implementation defined proxy type that conforms to the atomic view concept for the selected member object. Selecting a member outside of the array is erroneous and has undefined behavior. The implementation defined proxy type may hold a reference to the selected element and reference to associated auxiliary data (e.g., lock) as needed to support atomic operations. A temporary value of this type may be declared with the intent to perform multiple atomic operations on that member.

`bool atomic_array_view<T>::is_lock_free() const noexcept ;`

Returns whether atomic operations on the implementation defined proxy type are lock free.

Note: Under the HPC use case the member access operator, proxy type constructor, or proxy type destructor will be frequently invoked; therefore, an implementation should trade off decreased overhead in these operations versus increased overhead in the wrapper constructor and final destructor.

Example use:

```
// atomic array view wrapper constructor:
atomic_array_view<T> array( ptr , N );

// atomic operation on a member:
array[i].atomic-operation(...);

// atomic operations through a temporary value
// within a concurrent function:
auto x = array[i];
x.atomic-operation-a(...);
x.atomic-operation-b(...);
```

Suggested interface for the implementation defined proxy type.

```
struct implementation-defined-proxy-type {
  // conforms to atomic view concept

  // Construction limited to move
  implementation-defined-proxy-type(implementation-defined-proxy-type && ) =
noexcept ;
  ~implementation-defined-proxy-type();

  implementation-defined-proxy-type() = delete ;
  implementation-defined-proxy-type( const implementation-defined-proxy-type & ) =
delete ;
  implementation-defined-proxy-type &
    operator = ( const implementation-defined-proxy-type & ) = delete ;
};
```

4. Atomic View for a Single Non-atomic Object

Introducing concurrency within legacy codes can require replacing access operations to existing non-atomic objects with atomic access operations. The atomic_concurent_view<T> is intended to provide this functionality.

```
template< class T > struct atomic_concurrent_view {

  // conforms to atomic view concept

  atomic_concurrent_view( T & ) /* NOT noexcept */ ;
  atomic_concurrent_view( atomic_concurrent_view && ) noexcept ;
  ~atomic_concurrent_view() noexcept ;

  atomic_concurrent_view() = delete ;
  atomic_concurrent_view( const atomic_concurrent_view & ) = delete ;
  atomic_concurrent_view & operator = ( const atomic_concurrent__view & ) = delete
;
};

template< class T >
atomic_concurrent_view<T> make_atomic_concurrent_view( T & );
```

`atomic_concurrent_view<T>::atomic_concurrent_view( T & ) /* NOT noexcept */ ;`

This *wrapping constructor* wraps a trivially copyable object of type T. When a user wraps an object the user make the following guarantee as long as an atomic_concurrent_view may exist for that object. Violation of these usage guarantees results in undefined behavior.

1) The wrapped object shall not be accessed through any mechanism other than atomic_concurrent_view.
2) The wrapped object shall not be cast and subsequently wrapped by another atomic_concurrent_view of a different type T.
3) The wrapped object shall not be a member of an array wrapped by an atomic_array_view.
4) If T is a compound type (struct or statically sized array) then a subset of that object (member, array range) shall not be concurrently wrapped.
5) If the object is a member of a compound type then a superset of that object (enclosing struct or larger array range) shall not be concurrently wrapped.

In contrast to the atomic_array_view, many atomic_concurrent_view may be concurrently constructed to wrap a particular object. Atomic operations performed through these wrappers shall be atomic with respect to all corresponding wrappers for the particular object.

The wrapping constructor may acquire resources, such as retrieving a lock associated with the objects' address, as necessary to support atomic operations on the wrapped. The wrapping constructor is allowed to throw an exception; e.g., if auxiliary resources could not be acquired or if the object to be wrapped is not properly aligned. The wrapping constructor must throw an exception if atomic<T> is lock-free but atomic_concurrent_view<T> is not lock-free; e.g., due to improper alignment.

Implementation note: All non-atomic accesses of the wrapped object that appear before the wrapping constructor must complete before the wrapping constructor completes. For example:

```
void foo( int & i ) {
  i = 42 ;
  // The referenced value is 42 before the following statement completes.
  auto ai = make_atomic_concurrent_view(i);
```

atomic_concurrent_view<T>::~atomic_concurrent_view() noexcept ;

The destructor releases resources which may have been acquired by the wrapping constructor. After all concurrent wrappers for a given object are destroyed the user may directly access the object.

Example Usage

```
// atomic operation on an object:
make_atomic_concurrent_view(x).atomic-operation(...);

// When multiple atomic operations are performed the cost of
// constructing and destructing the atomic view can be amortized
// through a temporary atomic view object.
{
  auto ax = make_atomic_concurrent_view(x);
  ax.atomic-operation-a(...);
  ax.atomic-operation-b(...);
}
```