N4142: Atomic Operations on a Very Large Array

Carter Edwards hcedwar@sandia.gov
2014-09-08 (version 3)

1. Introduction

This paper identifies requirements for atomic operations on a very large array within high performance computing (HPC) applications and domain libraries, and proposes an extension to the atomic operations library [atomics] to satisfy these requirements.

2. Motivation / Requirements

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 will 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 through an allocation mechanism such as new or malloc.
b) A serial or parallel algorithm is used to initialize member values through non-conflicting assignments.
c) The array is wrapped by a constructed atomic_array<T> object.
d) A parallel algorithm updates member values exclusively through the atomic_array<T> interface.
e) The atomic_array<T> object is destructed.
f) Parallel algorithms access array member values through non-conflicting reads, writes, or updates.

3. Proposal for atomic_array<T>

Extend the atomic operations library with atomic_array<T> that wraps a pre-allocated and pre-initialized array of trivially copyable type T. Appropriate atomic<T> methods are translated to atomic_array<T> with the addition of an additional array offset argument to specify to which member of the array the atomic method is being applied. Specifying an offset outside of the size of the array is erroneous and has undefined behavior.

3.1. atomic_array<T>

```
template< class T > struct atomic_array {
  bool is_lock_free() const noexcept ;
  void store( size_t , T , memory_order = memory_order_seq_cst ) const noexcept ;
  T load( size_t , memory_order = memory_order_seq_cst ) const noexcept ;
  T exchange( size_t , memory_order = memory_order_seq_cst ) const noexcept ;
  bool compare_exchange_weak( size_t , T & , T , memory_order , memory_order )
const noexcept ;
  bool compare_exchange_strong( size_t , T & , T , memory_order , memory_order )
const noexcept ;
  bool compare_exchange_weak( size_t , T & , T , memory_order =
memory_order_seq_cst ) const noexcept ;
  bool compare_exchange_strong( size_t , T & , T , memory_order =
memory_order_seq_cst ) const noexcept ;

  size_t size() const noexcept ;
  ~atomic_array() noexcept ;
  atomic_array( T * , size_t N ) /* NOT noexcept */ ;

  atomic_array( const atomic_array & ) noexcept ;
  atomic_array( atomic_array && ) noexcept ;

  atomic_array() = delete ;
  atomic_array & operator = ( const atomic_array & ) = delete ;
};
```

3.2. atomic_array<*integral*>

```
template<> struct atomic_array<integral> {
  // additions to atomic_array<T>
  integral fetch_add( size_t , integral , memory_order = memory_order_seq_cst)
const noexcept;
  integral fetch_sub( size_t , integral , memory_order = memory_order_seq_cst)
const noexcept;
  integral fetch_and( size_t , integral , memory_order = memory_order_seq_cst)
const noexcept;
  integral fetch_or(  size_t , integral , memory_order = memory_order_seq_cst)
const noexcept;
  integral fetch_xor( size_t , integral , memory_order = memory_order_seq_cst)
const noexcept;
  // fetch_add one, fetch_sub one; replaces ++ and -- functionality
  integral fetch_inc( size_t , memory_order = memory_order_seq_cst) const noexcept;
  integral fetch_dec( size_t , memory_order = memory_order_seq_cst) const noexcept;

  // NOT atomic<integral> operator overloads: ++, --, +=, -=, &=, |=, ^=
};
```

3.3. atomic_array<*floating_point*>

A common HPC application and domain library pattern is to update an array of fundamental floating point type members through summation operations. This pattern results from numerical methods based upon superposition principle. Requirements for this operation have been acknowledged by some architectures providing hardware support for atomic floating point addition.

```
template<> struct atomic_array<fp> {
  // additions to atomic_array<T>
  fp fetch_add( size_t , fp , memory_order = memory_order_seq_cst) const noexcept;
};
```

3.4. Birth-to-Death Lifecycle: atomic_array(T*,size_t) through ~atomic_array()

The sole constructor **atomic_array<T>(T*,size_t)** initializes an atomic_array object for subsequent atomic operations on members of the wrapped array. As long as the atomic_array objects exists accessing members of the array through any other mechanism (e.g., dereferencing the original pointer) results in undefined behavior. Concurrent existence of more than one atomic_array object for a given array range results in undefined behavior.

The constructor and destructor may fence all outstanding memory accesses to members of the array. The constructor and destructor may allocate and deallocate auxiliary resources (e.g., locks) necessary to support the atomic member access operations. A poor-quality implementation could allocate an internal array of atomic<T> and initialize this internal array from the input array on construction, forward all atomic operations to the internal array, and then copy the internal array to the input array upon destruction.

The constructor may verify that the array has a desired alignment and adapt subsequent member access operations accordingly. The purpose for atomic_array is to enable support for the best-performing atomic operations on the wrapped array. The construct is allowed to throw an exception if the array is improperly aligned for the type **T**.

3.5. Shared semantics: atomic_array(const atomic_array &) and atomic_array(atomic_array &&)

An atomic_array object is a wrapper on an existing array, not a container. Allow atomic_array objects to be captured by value in lambda expressions and functors with guarantee of minimal (at most reference counting) overhead. If an implementation allocates and deallocates auxiliary resources these allocations are managed according to shared pointer semantics.

Default construction and assignment operations are deleted. An atomic_array object is bound to an array at construction and cannot be resassigned to wrap another array.

Member access methods modify members, not the atomic_array object. As such member access methods are const. This allows an atomic_array object to be captured as a const value.