Atomic maximum/minimum

Proposal to extend atomic with maximum/minimum operations

Document number: P0493R0
Date: 2016-11-08
Authors: Al Grant (algrant@acm.org), Bronek Kozicki (brok@spamcop.net)
Audience: LEWG; SG1 - Concurrency

Introduction

This proposal extends the atomic operations library to add atomic maximum/minimum operations. These were originally proposed by N3696 as particular cases of a general priority update mechanism, where objects were tested and conditionally updated. In contrast to N3696, we propose atomic maximum/minimum operations that have the effect of unconditional memory updates with respect to memory ordering. A future proposal may reintroduce the concept of a conditionalized atomic update.

This paper benefited from discussion with Mario Torrecillas Rodriguez, Nigel Stephens and Nick Maclaren.

Background and motivation

Atomic maximum/minimum operations are useful in a variety of situations in multithreaded applications:

- optimal implementation of certain lock-free shared data structures - as in the motivating example later in this paper
- reductions in data-parallel applications: for example, OpenMP (https://computing.llnl.gov/tutorials/openMP/#REDUCTION) supports maximum/minimum as a reduction operation
- recording the maximum so far reached in an optimization process, to allow unproductive threads to terminate
- collecting statistics, such as the largest item of input encountered by any worker thread.

Atomic maximum/minimum operations already exist in several other programming environments, including OpenCL (https://www.khronos.org/registry/cl/specs/opencl-2.0-openclc.pdf), and in some hardware implementations. Application need, and availability, motivate providing these operations in C++.

Notes on the proposal

1. the proposed operation has the effect of a read-modify-write, irrespective of whether the value changes. This accords with existing atomic operations (even when, like `fetch_or(0)`, they never affect the value) and avoids the need to deal with data-dependent memory effects. Conditional stores, barriers etc. may be used to implement the operations as long as the overall effect with respect to the requested memory order is that of an atomic read-modify-write.
2. syntactically, maximum/minimum are not operators in C++, so those atomic functions which would normally be overloaded on compound operators are instead named as *`op`*`_fetch`. This ensures that the functionality provided for different operations is consistent, even though the naming is (necessarily) different. **These functions could be given an additional optional memory ordering parameter, and they could also be defined for existing operators (i.e. `add_fetch` etc.). We suggest the committee decide on both these points.**
3. whether types are signed or unsigned affects the result of maximum/minimum operations. The atomic type determines the type of the operation.
4. this paper proposes operations on integral and pointer types only. If both this proposal and floating-point atomics as proposed in P0020 are adopted then we propose that atomic floating-point maximum/minimum operations also be defined, in the obvious way.

The following text outlines the proposed changes, based on N4060 (Working Draft dated 2016-07-12).

29.2: Header <atomic> synopsis

```
namespace std {
  // 29.6.2, templated operations on atomic types
  ...
  template<class T> T atomic_fetch_max(volatile atomic<T>*, T) noexcept;
  template<class T> T atomic_fetch_max(atomic<T>*, T) noexcept;
  template<class T> T atomic_fetch_max_explicit(volatile atomic<T>*, T, memory_order) noexcept;
  template<class T> T atomic_fetch_max_explicit(atomic<T>*, T, memory_order) noexcept;
  template<class T> T atomic_fetch_min(volatile atomic<T>*, T) noexcept;
  template<class T> T atomic_fetch_min(atomic<T>*, T) noexcept;
  template<class T> T atomic_fetch_min_explicit(volatile atomic<T>*, T, memory_order) noexcept;
  template<class T> T atomic_fetch_min_explicit(atomic<T>*, T, memory_order) noexcept;
```

```
// 29.6.3, arithmetic operations on atomic types
  ...
  integral atomic_fetch_max(volatile atomic-integral*, integral) noexcept;
  integral atomic_fetch_max(atomic-integral*, integral) noexcept;
  integral atomic_fetch_max_explicit(volatile atomic-integral*, integral, memory_order) noexcept;
  integral atomic_fetch_max_explicit(atomic-integral*, integral, memory_order) noexcept;
  integral atomic_fetch_min(volatile atomic-integral*, integral) noexcept;
  integral atomic_fetch_min(atomic-integral*, integral) noexcept;
  integral atomic_fetch_min_explicit(volatile atomic-integral*, integral, memory_order) noexcept;
  integral atomic_fetch_min_explicit(atomic-integral*, integral, memory_order) noexcept;
  ...
}
```

29.5: Atomic types

```
namespace std {
  template<> struct atomic<integral> {
    ...
    integral fetch_max(integral, memory_order = memory_order_seq_cst) volatile noexcept;
    integral fetch_max(integral, memory_order = memory_order_seq_cst) noexcept;
    integral max_fetch(integral) volatile noexcept;
    integral max_fetch(integral) noexcept;
    integral fetch_min(integral, memory_order = memory_order_seq_cst) volatile noexcept;
    integral fetch_min(integral, memory_order = memory_order_seq_cst) noexcept;
    integral min_fetch(integral) volatile noexcept;
    integral min_fetch(integral) noexcept;
    ...
  };
  template<class T> struct atomic<T*> {
    ...
    T* fetch_max(T*, memory_order = memory_order_seq_cst) volatile noexcept;
    T* fetch_max(T*, memory_order = memory_order_seq_cst) noexcept;
    T* max_fetch(T*) volatile noexcept;
    T* max_fetch(T*) noexcept;
    T* fetch_min(T*, memory_order = memory_order_seq_cst) volatile noexcept;
    T* fetch_min(T*, memory_order = memory_order_seq_cst) noexcept;
    T* min_fetch(T*) volatile noexcept;
    T* min_fetch(T*) noexcept;
  };
}
```

29.6: Operations on atomic types

29.6.5: Requirements for operations on atomic types

In table 148, add the following entries:

Key	Op	Computation
`max`		maximum as computed by `std::max` from `<algorithm>`
`min`		minimum as computed by `std::min` from `<algorithm>`

Add:

```
C A::key_fetch(M operand) volatile noexcept;
C A::key_fetch(M operand) noexcept;
```

These operations are only defined for keys '`max`' and '`min`'.

Effects: `A::fetch_`*`key`*`(operand)`

Returns: `std::`*`key`*`(A::fetch_`*`key`*`(operand), operand)`

Motivating example

The following example implements a lockfree queue, used in the following way:

```
int main()
{
    queue<int> q(16);
    assert(q.post(42));
    int d;
    assert(q.read(d));
    assert(d == 42);
    assert(not q.read(d));
}
```

A naive implementation of the queue follows:

```
#include <atomic>
#include <utility>
#include <cstddef>

template <typename T>
class queue
{
    // Rounded up logarithm with base of 2
    static int log2(int s)
    {
        --s;
        int r = 0;
        while (s)
        {
            s >>= 1;
            r += 1;
        };
        return r;
    }

    // Actual data storage, contains queued value and data stamp for this slot
    struct slot
    {
        slot() : value(), stamp(0)
        { }

        T value;
        std::atomic_long stamp;
    };

public:
    queue(const queue&) = delete;
    queue& operator=(const queue&) = delete;

    explicit queue(int s) : head_(0) , tail_(0) , bits_(log2(s)) , size_(1 << bits_) , buffer_(nullptr)
    {
        buffer_ = new slot[size_];
    }

    ~queue()
    {
        // Must not be called when either post() or read() are running in other
        // threads. Such calls must be completed before destruction
        delete[] buffer_;
    }

    bool post(T&& v) noexcept(true)
    {
        slot* ptr = nullptr;    // Store the data to here
        long expected = 0;      // compared against ptr->stamp
        unsigned long head = head_.load();
        for (;;)
        {
            ptr = &buffer_[index(head)];
            expected = stamp(head);
            const long newstamp = expected + 1;
            const long oldstamp = ptr->stamp.load();
            if (oldstamp == expected)
            {
                const unsigned long next = head + 1ul;
                // Try to claim ownership of the slot
                if (head_.compare_exchange_weak(head, next))
                {
                    ptr->stamp = newstamp;
                    break;
                }
```

```
                // else head has been updated
            }
            else if (oldstamp > expected)
                head = head_.load(); // claimed by another thread already
            else
                return false; // overflowing, i.e. ptr is to be read yet
        }

        ptr->value = std::move(v);
        ptr->stamp = expected + 2;
        return true;
    }

    bool read(T& v) noexcept(true)
    {
        slot* ptr = nullptr;       // Read the data from here
        long expected = 0;         // compared against ptr->stamp
        unsigned long tail = tail_.load();
        for (;;)
        {
            // Optimize for case when data needs to be read, but check that
            // there is actually anything in there.
            if (tail == head_.load())
                break; // Must not advance tail beyond head

            ptr = &buffer_[index(tail)];
            expected = stamp(tail) + 2;
            const long newstamp = expected + 1; // = stamp(tail) + 3
            const long oldstamp = ptr->stamp.load();
            if (oldstamp == expected)
            {
                const unsigned long next = tail + 1ul;
                // Try to claim ownership of the slot
                if (tail_.compare_exchange_weak(tail, next))
                {
                    ptr->stamp = newstamp;
                    break;
                }
                // else tail has been updated
            }
            else
                tail = tail_.load(); // claimed by another thread already

            ptr = nullptr;
        }

        if (ptr)
        {
            v = std::move(ptr->value);
            ptr->stamp = expected + 2;
            return true;
        }

        return false;
    }

private:
    // Calculate head/tail position inside buffer_ array
    constexpr int index(unsigned long h) const
    {
        return (h & (size_ - 1ul));
    }

    // Calculate lap number for high bits in slot->stamp
    constexpr long stamp(unsigned long h) const
    {
        return (h & ~(size_ - 1ul)) >> (bits_ - 2);
    }

    std::atomic_ulong           head_; // slot being written
    std::atomic_ulong           tail_; // slot being read
    const int                   bits_; // = log2(size_)
```

```
    const int                    size_; // must be power of 2
    slot*                        buffer_;
};
```

This version suffers from a performance problem, because `read()` will not be able to skip over the slot still-being-written to following it slots which are ready for read. The following improved version uses `atomic_fetch_max`:

```
#include <atomic>
#include <utility>
#include <cstddef>

template <typename T>
class queue
{
    // Rounded up logarithm with base of 2
    static int log2(int s)
    {
        --s;
        int r = 0;
        while (s)
        {
            s >>= 1;
            r += 1;
        };
        return r;
    }

    // Actual data storage, contains queued value and data stamp for this slot
    struct slot
    {
        slot() : value(), stamp(0)
        { }

        T value;
        std::atomic_long stamp;
    };

public:
    queue(const queue&) = delete;
    queue& operator=(const queue&) = delete;

    explicit queue(int s) : head_(0) , tail_(0) , bits_(log2(s)) , size_(1 << bits_) , buffer_(nullptr)
    {
        buffer_ = new slot[size_];
    }

    ~queue()
    {
        // Must not be called when either post() or read() are running in other
        // threads. Such calls must be completed before destruction
        delete[] buffer_;
    }

    bool post(T&& v) noexcept(true)
    {
        slot* ptr = nullptr;    // Store the data to here
        long expected = 0;      // CAS against ptr->stamp
        unsigned long head = head_.load();
        unsigned long next = 0; // Next value of head
        for (;;)
        {
            next = head + 1ul;
            ptr = &buffer_[index(head)];
            expected = stamp(head);
            const long newstamp = expected + 1;
            // Not going to revisit this slot in next iteration, so "strong" is required
            if (ptr->stamp.compare_exchange_strong(expected, newstamp))
                break;

            // Advance to next slot if this was claimed by another thread
            if (expected >= newstamp)
                head = next;
            else
                return false; // overflowing, i.e. ptr is to be read yet
        }
        atomic_fetch_max(head_, next);
```

```
        ptr->value = std::move(v);
        ptr->stamp = expected + 2;
        return true;
    }

    bool read(T& v) noexcept(true)
    {
        slot* ptr = nullptr;       // Read the data from here
        long expected = 0;         // CAS against ptr->stamp
        unsigned long tail = tail_.load();
        unsigned long next = tail; // Next value of tail
        for (;;)
        {
            // Optimize for case when data needs to be read, but check that
            // there is actually anything in there.
            const unsigned long head = head_.load();
            if (tail == head)
                break; // Must not advance tail beyond head

            ptr = &buffer_[index(tail)];
            expected = stamp(tail) + 2;
            const long newstamp = expected + 1; // = stamp(tail) + 3
            // Not going to revisit this slot in next iteration, so "strong" is required
            if (ptr->stamp.compare_exchange_strong(expected, newstamp))
            {
                // Advance tail if no slot was being written
                if (next == tail)
                    next = tail + 1ul;
                break;
            }

            ptr = nullptr;
            // Advance tail if no slot was being written.
            if (expected >= newstamp && next == tail)
                next = tail + 1ul;
            tail += 1ul;
        }
        atomic_fetch_max(tail_, next);

        if (ptr)
        {
            v = std::move(ptr->value);
            ptr->stamp = expected + 2;
            return true;
        }

        return false;
    }

private:
    // Calculate head/tail position inside buffer_ array
    constexpr int index(unsigned long h) const
    {
        return (h & (size_ - 1ul));
    }

    // Calculate lap number for high bits in slot->stamp
    constexpr long stamp(unsigned long h) const
    {
        return (h & ~(size_ - 1ul)) >> (bits_ - 2);
    }

    std::atomic_ulong           head_; // slot being written
    std::atomic_ulong           tail_; // slot being read
    const int                   bits_; // = log2(size_)
```

```
    const int                   size_; // must be power of 2
    slot*                       buffer_;
};
```