

Atomic maximum/minimum

Proposal to extend atomic with maximum/minimum operations

Document number: P0493R1
Date: 2020-05-08
Reply-To: Al Grant (algrant@acm.org), Bronek Kozicki (brok@spamcop.net)
Audience: LEWG; SG1 - Concurrency

Abstract

Add integer 'max' and 'min' operations to the set of operations supported in `<atomic>`, with minor adjustments to function naming necessitated by the fact that 'max' and 'min' do not exist as infix operators. Memory ordering will be the same as for existing operations, i.e. writes are unconditional.

Revision history

- P0493R0 (2016-11-08): Original proposal
- P0493R1 (2020-05-08): Add motivation for defining new atomics as read-modify-write. Clarify status of proposal for new-value-returning operations. Align with C++17.

Introduction

This proposal extends the atomic operations library to add atomic maximum/minimum operations. These were originally proposed for C++ in 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, like the existing standard atomic operations, 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 addition (fetch-and-add) was introduced in the NYU Ultracomputer [Gottlieb 1982], has been implemented in a variety of hardware architectures, and has been standardized in C and C++. Atomic maximum/minimum operations (fetch-and-max) have a history almost as long as atomic addition, e.g. see [Lipovski 1988], and have also been implemented in various hardware architectures but are not currently standard in C and C++. This proposal fills the gap.

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

- optimal implementation of 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-opencl.pdf>), and in some hardware implementations. Application need, and availability, motivate providing these operations in C++.

The proposed language changes add atomic max/min to `<atomic>`, based on the existing atomic operations, with some adjustment due to the fact that C++ has no infix operators for max/min. Like the existing atomic operations, atomic max/min have the effect of a read-modify-write, irrespective of whether the value changes. This is done for several reasons:

- it matches the semantics of the other atomic operations, e.g. `fetch_add()`, `fetch_or()`, `test_and_set()`
- fetch-and-max intrinsics in several other programming models (OpenCL, CUDA, C++ AMP, HCC) define it as a read-modify-write
- several existing hardware architectures (ARM, RISC-V) define fetch-and-max as a read-modify-write

There is an alternative point of view that atomic max/min should be implemented as an atomic read-and-conditional-store, and at least one hardware architecture (POWER) implements it this way. We believe defining this in C++ differently from the existing atomics but with a similar name would be inconsistent. The new operations are not sufficiently different from the existing operations to justify this inconsistency. Existing operations may also result in no change to the value, yet are still defined as unconditional writes. For example `fetch_or(1)` results in no change to the target value if the least significant bit is already set, and `fetch_or(0)` never changes the target value. Similarly, `test_and_set()` is defined as a read-modify-write even if the flag is already set. The operation of 'max' on the lattice of integers is the counterpart of the operation of 'or' on the lattice of bit vectors; 'or' is bitwise 'max'. C++ has already defined that `fetch_or()` is an unconditional store, so for consistency, `fetch_max()` should also be an unconditional store. A future proposal might define conditionally-storing versions of atomic operations - i.e. conditionally storing equivalents of `fetch_or()`, `fetch_max()`, `test_and_set()` etc. - with a different name scheme.

On a hardware implementation where fetch-and-max is a conditional store, the read-modify-write operations proposed in this paper can be implemented using compare-exchange.

Summary of proposed additions to `<atomic>`

The current `<atomic>` provides atomic operations in several ways:

- as a named non-member function template e.g. `atomic_fetch_add()` returning the old value
- as a named member function template e.g. `atomic<T>::fetch_add()` returning the old value

- as an overloaded compound operator e.g. `atomic<T>::operator+=()` returning the new value

Adding 'max' and 'min' versions of the named functions is straightforward. Max/min operations exist in signed and unsigned flavors. The atomic type determines the operation. There is precedent for this in C, where all compound assignments on atomic variables are defined to be atomic, including sign-sensitive operations such as divide and right-shift.

The overloaded operator `atomic<T>::operator op=(n)` is defined to return the new value of the atomic object. This does not correspond directly to a named function. For max and min, we have no infix operators to overload. So if we want a function that returns the new value we would need to provide it as a named function. However, for all operators the new value can be obtained as `fetch_op(n) op n`, (the standard defines the compound operator overloads this way) while the reverse is not true for non-invertible operators like 'and' or 'max'. Thus the functions would add no significant functionality other than providing one-to-one equivalents to `<atomic>`'s existing compound operator overloads, and **we do not currently propose them** in P0493. **We suggest the committee decide whether these functions are needed, and if so, whether they should be given an additional optional memory ordering parameter, and whether they should also be defined for existing operations.** Following some of the early literature on atomic operations ([Kruskal 1988] citing [Draughon 1967]), we suggest that if required, names should have the form `replace_op`. We must stress that any inconsistency with the existing atomic functions is based on the lack of infix representation of these operations in the language syntax, rather than because of any difference in the nature of the operations in the language execution model.

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.

References

- [Almasi]: "Highly Parallel Computing" 2nd ed., George S. Almasi and Allan Gottlieb,
- [Draughon 1967]: "Programming Considerations for Parallel Computers", E.R. Draughon et al., Courant Inst, 1967
- [Gong 1990]: "A Library of Concurrent Objects and Their Proofs of Correctness", Chun Gong and Jeanette M. Wing, 1990, <http://www.cs.cmu.edu/~wing/publications/CMU-CS-90-151.pdf>
- [Gottlieb 1982]: "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer", Gottlieb et al., ICCA, 1982
- [Kruskal 1988]: "Efficient Synchronization on Multiprocessors with Shared Memory", Clyde P. Kruskal et al., Ultracomputer Note #105, 1988
- [Lipovski 1988]: "A Fetch-And-Op Implementation for Parallel Computers", G.J. Lipovski and Paul Vaughan, 1988

Changes to the C++ standard

The following text outlines the proposed changes, based on N4660 (DIS 14882:2017).

32: Atomic operations library [atomics]

32.2: Header <atomic> synopsis [atomics.syn]

```
namespace std {
    // 32.7, non-member functions
    ...
    template<class T>
        T atomic_fetch_max(volatile atomic<T>*, typename atomic<T>::value_type) noexcept;
    template<class T>
        T atomic_fetch_max(atomic<T>*, typename atomic<T>::value_type) noexcept;
    template<class T>
        T atomic_fetch_max_explicit(volatile atomic<T>*, typename atomic<T>::value_type, memory_order) noexcept;
    template<class T>
        T atomic_fetch_max_explicit(atomic<T>*, typename atomic<T>::value_type, memory_order) noexcept;
    template<class T>
        T atomic_fetch_min(volatile atomic<T>*, typename atomic<T>::value_type) noexcept;
    template<class T>
        T atomic_fetch_min(atomic<T>*, typename atomic<T>::value_type) noexcept;
    template<class T>
        T atomic_fetch_min_explicit(volatile atomic<T>*, typename atomic<T>::value_type, memory_order) noexcept;
    template<class T>
        T atomic_fetch_min_explicit(atomic<T>*, typename atomic<T>::value_type, memory_order) noexcept;
    ...
}
```

32.6.2: Specializations for integers [atomic.types.int]

```
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 fetch_min(integral, memory_order = memory_order_seq_cst) volatile noexcept;
        integral fetch_min(integral, memory_order = memory_order_seq_cst) noexcept;
        ...
    };
}
```

In table 138, add the following entries:

Key	Op	Computation
-----	----	-------------

max		maximum as computed by <code>std::max</code> from <code><algorithm></code>
min		minimum as computed by <code>std::min</code> from <code><algorithm></code>

Add:

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

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

Effects: `A::fetch_key(operand)`

Returns: `std::key(A::fetch_key(operand), operand)`

After

```
T* operator op=(T operand) noexcept;
add "These operations are not defined for keys 'max' and 'min'."
```

32.6.3 Partial specialization for pointers [atomics.types.pointer]

```
namespace std {
  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* fetch_min(T*, memory_order = memory_order_seq_cst) volatile noexcept;
    T* fetch_min(T*, memory_order = memory_order_seq_cst) noexcept;
    ...
  };
}
```

In table 139, add the following entries:

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

Add:

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

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

Effects: `A::fetch_key(operand)`

Returns: `std::key(A::fetch_key(operand), operand)`

After

```
T* operator op=(T operand) noexcept;
add "These operations are not defined for keys 'max' and 'min'."
```

Motivating example

Atomic fetch-and-max can be used to implement a lockfree bounded queue, as explained in [Gong]:

```
typedef struct {
  elt item; /* a queue element */
  int tag; /* its generation number */
} entry;

typedef struct rep {
  entry elts[SIZE]; /* a bounded array */
  int back;
} reptype;

reptype queue;
```

```

void Enq(elt x) {
    int i;
    entry e, *olde;
    e.item = x;
    i = READ(&(queue.back)) + 1; /* set the new element's item to x */
    /* get a slot in the array for the new element */
    while (true) {
        e.tag = i / SIZE; /* set the new element's generation number */
        olde = EXCHANGE(&(queue.elts[i % SIZE]), -1, &e);
        /* exchange the new element with slot's
        value if that slot has not been used */
        if (olde->tag == -1) { /* if exchange is successful */
            break; /* get out of the loop */
        }
        ++i; /* otherwise, try the next slot */
    }
    FETCH_AND_MAX(&(queue.back), i); /* reset the value of back */
}

elt Deg() {
    entry e, *olde;
    int i, range;
    e.tag = -1; /* make e an empty entry */
    e.item = NULL;
    while (true) { /* keep trying until an element is found */
        range = READ(&(queue.back)) - 1; /* search up to back-1 slots */
        for (i = 0; i <= range; i++) {
            olde = EXCHANGE(&(queue.elts[i % SIZE]), i / SIZE, &e);
            /* check slot to see if it contains the oldest element */
            if (olde->tag != -1) { /* if so */
                return(olde->item); /* return the item in it */
            }
        } /* otherwise try the next one */
    }
}

```

A similar C++ example was used in the original version of this paper, due to Bronek Kozicki.

A queue class can be used as follows:

```

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 <cstdint>

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 <cstdint>

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_ulong 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_;
};

```