A Multithreaded, Transaction-Based, Read/Write Locking Strategy for Containers

Bob Steagall

C++Now 2019

Overview

- Sharing a container among multiple threads
- A motivating problem
- Some possible solutions
- A solution based on strict timestamp ordering (STO)
- Testing the STO-based solution
- Summary

Sharing a Container

- Sometimes a container must be shared between threads

- We desire to avoid race conditions during writes
  - Assume elements are themselves unsynchronized – i.e., susceptible to races
  - Exactly one thread may update an element at any given time
  - No other thread may read an element while the writer is updating it
  - More than one thread may read an element when no update is in progress

- We now have a wealth of concurrency tools at our disposal
  - Writing multi-threaded applications is easier than ever
    - (Maybe)

Sharing a Container – Avoiding Race Conditions

- If all threads are readers…
  - No locking is required

- If the number of reads is much larger than the number of writes…
  - We might be able to use a readers/writer lock (`std::shared_mutex`)

- What about the case where most operations are writes?
  - A per-element mutex strategy might work…
  - ... **if** a given write operation requires locking exactly one element

Sharing a Container – Avoiding Race Conditions

- What about the case where **all** operations are writes…

  -- and --

- Each element E to be updated is related to a set R_E of other elements
  - let's call this set *E's related group*
  - let's define $U_E = \{E \cup R_E\}$ and call it *E's update group*

  -- and --

- One or more elements in R_E must also be updated at the same time as, and consistently, with E

  -- and --

- The number of elements in R_E to be updated varies

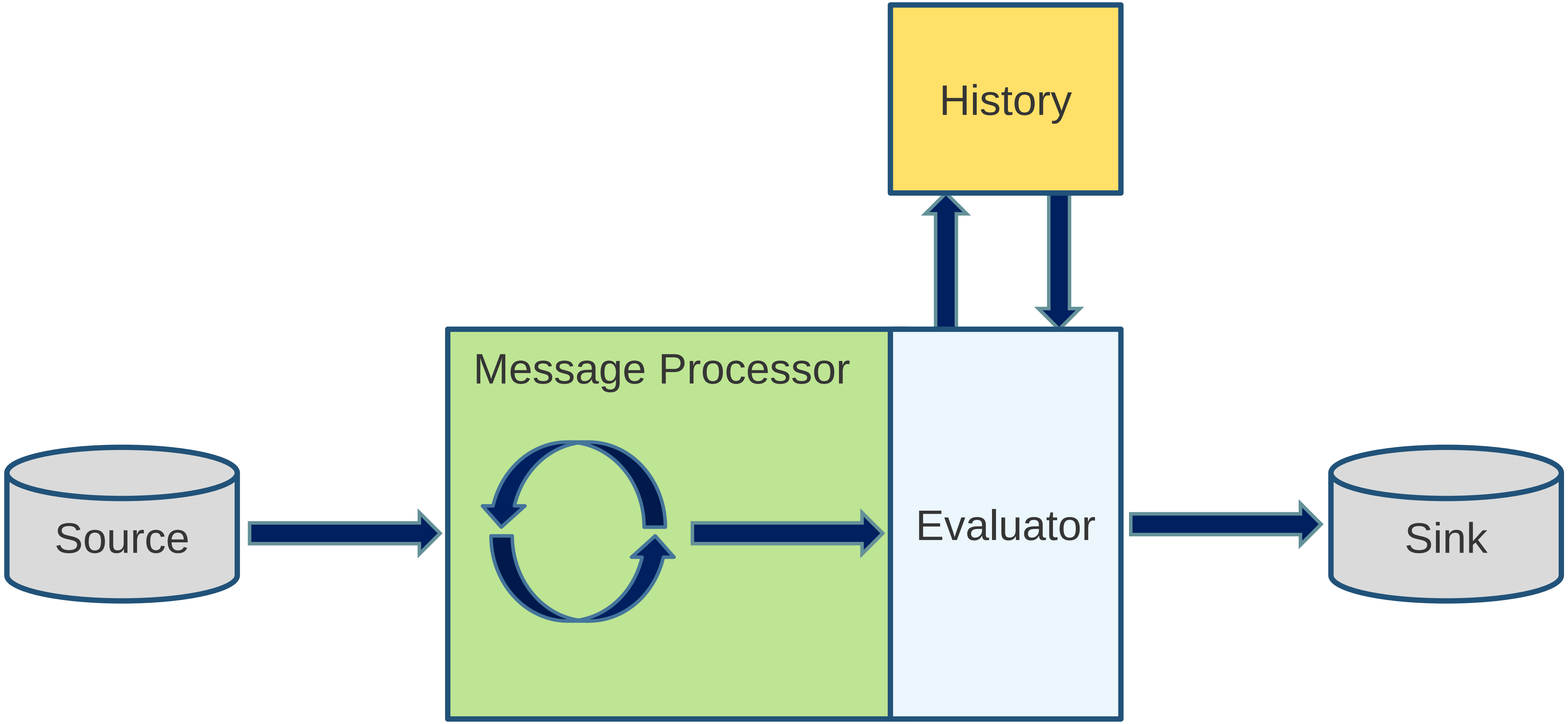
  -- and --

- The membership of R_E varies

Motivating Problem – Reactive Message Processor

- Receives a continuous stream of input messages
- Generates output only when something interesting happens
  - A **history** must be maintained to detect relevant changes
- Reading the state of the history is never required
- Every message input requires a write operation to the history
  - Which uses one or more containers

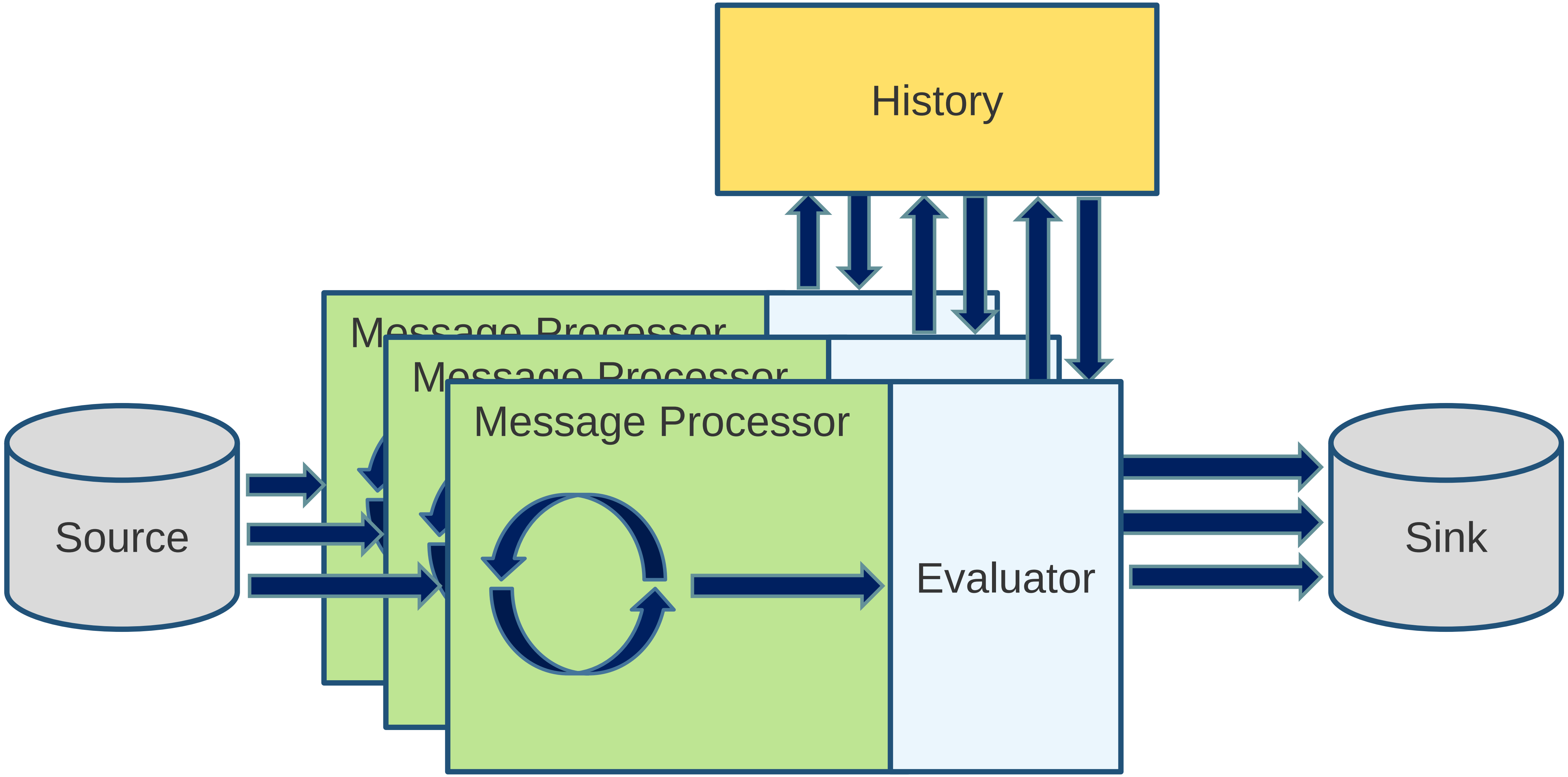
Motivating Problem – Reactive Message Processor



Motivating Problem – Reactive Message Processor

- Receives a continuous stream of input messages
- Generates output only when something special happens
  - History must be maintained to detect “special” changes
- Reading the state of the history is never required
- Every message input requires a write operation to a container
  - Which uses one or more containers
- **It must scale to multiple threads!**

Motivating Problem – Reactive Message Processor



Solution Requirements

- **Atomic:** Each modification of an update group is treated as a single transaction, which either succeeds completely, or fails completely
- **Consistent:** Each transaction can only bring the update group (and the enclosing container) from one valid state to another, maintaining all invariants
- **Isolated**: Each transaction must ensure that concurrent execution of other transactions leaves its update group (and the container) in the same state that would have been obtained as if the transactions were executed in some valid sequential order

Some Solutions - Sharding

- Divide the set of elements into individual *shards* such that the members of each element’s related group are also in the shard

- Updates to each shard are performed by only one thread dedicated to servicing that shard

- Upside
  - Good performance

- Downside
  - Increased complexity
  - It works IFF the data is amenable to sharding

Some Solutions – Per-Container Mutex

- Instantiate a per-container mutex and perform updates in a single critical section guarded by the mutex

- Upside
  - It works
  - Easy to understand and think about

- Downside
  - **Does not scale**

Some Solutions – Per-Element Mutex

- Instantiate a per-element mutex, acquire the mutexes for all the elements in the update group, then update and release them all

- Upside
  - Seems like it should work (at least for the case of exactly one element)
  - Slightly more difficult to understand and think about

- Downside
  - Some mutex implementations are not small
  - What if E_0's related group contains E_1, and E_1's related group contains E_0?
  - **Trap!**

Some Solutions – Per-Element Mutex

- Instantiate a per-element mutex, acquire the mutexes for all the elements in the update group, then update and release them all

- Upside
  - Seems like it should work
  - Slightly more difficult to understand and think about

- Downside
  - Some mutex implementations are not small
  - What if E_0's related group contains E_1, and E_1's related group contains E_0?
  - **Trap!**

Another Solution – Strict Timestamp Ordering (STO)

- STO is one of *many* database concurrency control algorithms

- It is a transactional, *timestamp-based* algorithm

- **Timestamp**
  - A monotonically increasing value that indicates the age of a transaction
  - A lower timestamp value (TSV) indicates an older transaction
  - A higher timestamp value indicates a newer transaction

- STO uses timestamps to serialize concurrent transactions

Strict Timestamp Ordering

- When each transaction begins it is assigned a unique timestamp from a universal timestamp source

- If two transactions are attempting to write the same update group at the same time, the transaction with the lower timestamp goes first
  - Younger transactions always wait for older transactions
  - Older transactions never wait for younger transactions, they give up (roll back) and try again

- STO operation schedules are serializable and deadlock-free
  - Price for deadlock-freedom is the potential restart of a transaction

Strict Timestamp Ordering

- A transaction `TX`$_0$ with timestamp `tsv`
  - Functions `begin()`, `commit()`, and `rollback()`
- An element `E`$_0$ with
  - A read timestamp `rd_tsv`
  - A write timestamp `wr_tsv`
- Function `update()` to update `E`$_0$
- Function `read()` to read from `E`$_0$

Strict Timestamp Ordering

- Transaction TX_0 calls $\texttt{read}(E_0)$

- If $[TX_0.\texttt{tsv} < E_0.\texttt{wr\_tsv}]$ then
  - a younger transaction has already written E_0, so call $\texttt{rollback}(TX_0)$

- If $[TX_0.\texttt{tsv} > E_0.\texttt{wr\_tsv}]$ then
  - delay TX_0 until TX_1 (the transaction that wrote E_0) is done
  - update $E_0.\texttt{rd\_tsv}$

Strict Timestamp Ordering

- Transaction $\mathtt{TX_0}$ calls $\mathtt{update(E_0)}$

- If $\mathtt{[TX_0.tsv < E_0.rd\_tsv]}$ then
  - a younger transaction has already read from $\mathtt{E_0}$, so call $\mathtt{rollback(TX_0)}$

- If $\mathtt{[TX_0.tsv < E_0.wr\_tsv]}$ then
  - a younger transaction has already written to $\mathtt{E_0}$, so call $\mathtt{rollback(TX_0)}$

- If $\mathtt{[TX_0.tsv > E_0.wr\_tsv]}$ then
  - delay $\mathtt{TX_0}$ until $\mathtt{TX_1}$ (the transaction that wrote $\mathtt{E_0}$) is done
  - update $\mathtt{E_0}$ and $\mathtt{E_0.wr\_tsv}$

Implementing STO in C++

- For our problem, assume all operations are updates
  - Then we need only one timestamp per element, E.tsv

- Transaction TX_0 calls update(E_0)

- If [TX_0.tsv < E_0.tsv] then
  - a younger transaction has already written to E_0, so call rollback(TX_0)

- If [TX_0.tsv > E_0.tsv] then
  - delay TX_0 until TX_1 (the transaction that wrote E_0) is done
  - update E_0 and E_0.tsv

Implementing STO in C++ – Design Choices

- Need some basic synchronization components
  - Mutex
  - Condition variable
  - Atomic pointer
  - Atomic compare and exchange
- Need a class that represents a lockable item (element)
- Need a class that represents a transaction

Implementing STO in C++ – Design Choices

- Threads share containers holding lockable items

- Threads will instantiate and own transactions

- A transaction’s key member functions (begin, acquire, commit, rollback) will only be called by the owning thread

- Transactions acquire lockable items on behalf of their owning thread
    - Owning thread performs actual write operations

- Transactions will acquire the entire update group before the thread applies any write operation to any member of that group

Implementing STO in C++ – Design Choices

```
thread_function(shared_collection)
    [Create a transaction object]

    while [Work remains to be done]
        [Begin a new transaction]
        [Find target element E in shared_collection]
        [Determine the membership of the update group to the extent possible]

        while [Update group members remain unacquired AND acquisitions have all succeeded]
            [Starting with E, attempt to acquire the next unacquired member of the update group]
            [Revise the membership of the update group, if necessary]
        endwhile

        if [All members of the update group were acquired]
            [Apply write operations to the members of the update group]
            [Commit the transaction, loop back to the top, get new work]
        else
            [Roll back the transaction, loop back to the top, and try again]
        endif
    endwhile
```

Implementing STO in C++ – Design Choices

```
thread_function(shared_collection)
    [Create a transaction object]

    while [Work remains to be done]
        [Begin a new transaction]
        [Find target element E in shared_collection]
        [Determine the membership of the update group to the extent possible]

        while [Update group members remain unacquired AND acquisitions have all succeeded]
            [Starting with E, attempt to acquire the next unacquired member of the update group]
            [Revise the membership of the update group, if necessary]
        endwhile

        if [All members of the update group were acquired]
            [Apply write operations to the members of the update group]
            [Commit the transaction, loop back to the top, get new work]
        else
            [Roll back the transaction, loop back to the top, and try again]
        endif
    endwhile
```

Implementing STO in C++ – Design Choices

```
thread_function(shared_collection)
    [Create a transaction object]

    while [Work remains to be done]
        [Begin a new transaction]
        [Find target element E in shared_collection]
        [Determine the membership of the update group to the extent possible]

        while [Update group members remain unacquired AND acquisitions have all succeeded]
            [Starting with E, attempt to acquire the next unacquired member of the update group]
            [Revise the membership of the update group, if necessary]
        endwhile

        if [All members of the update group were acquired]
            [Apply write operations to the members of the update group]
            [Commit the transaction, loop back to the top, get new work]
        else
            [Roll back the transaction, loop back to the top, and try again]
        endif
    endwhile
```

Implementing STO in C++ – Design Choices

```
thread_function(shared_collection)
    [Create a transaction object]

    while [Work remains to be done]
        [Begin a new transaction]
        [Find target element E in shared_collection]
        [Determine the membership of the update group to the extent possible]

        while [Update group members remain unacquired AND acquisitions have all succeeded]
            [Starting with E, attempt to acquire the next unacquired member of the update group]
            [Revise the membership of the update group, if necessary]
        endwhile

        if [All members of the update group were acquired]
            [Apply write operations to the members of the update group]
            [Commit the transaction, loop back to the top, get new work]
        else
            [Roll back the transaction, loop back to the top, and try again]
        endif
    endwhile
```

Implementing STO in C++ – Design Choices

```
thread_function(shared_collection)
    [Create a transaction object]

    while [Work remains to be done]
        [Begin a new transaction]
        [Find target element E in shared_collection]
        [Determine the membership of the update group to the extent possible]

        while [Update group members remain unacquired AND acquisitions have all succeeded]
            [Starting with E, attempt to acquire the next unacquired member of the update group]
            [Revise the membership of the update group, if necessary]
        endwhile

        if [All members of the update group were acquired]
            [Apply write operations to the members of the update group]
            [Commit the transaction, loop back to the top, get new work]
        else
            [Roll back the transaction, loop back to the top, and try again]
        endif
    endwhile
```

Implementing STO in C++ – Design Choices

```
thread_function(shared_collection)
    [Create a transaction object]

    while [Work remains to be done]
        [Begin a new transaction]
        [Find target element E in shared_collection]
        [Determine the membership of the update group to the extent possible]

        while [Update group members remain unacquired AND acquisitions have all succeeded]
            [Starting with E, attempt to acquire the next unacquired member of the update group]
            [Revise the membership of the update group, if necessary]
        endwhile

        if [All members of the update group were acquired]
            [Apply write operations to the members of the update group]
            [Commit the transaction, loop back to the top, get new work]
        else
            [Roll back the transaction, loop back to the top, and try again]
        endif
    endwhile
```

Implementing STO in C++ – Design Choices

```
thread_function(shared_collection)
    [Create a transaction object]

    while [Work remains to be done]
        [Begin a new transaction]
        [Find target element E in shared_collection]
        [Determine the membership of the update group to the extent possible]

        while [Update group members remain unacquired AND acquisitions have all succeeded]
            [Starting with E, attempt to acquire the next unacquired member of the update group]
            [Revise the membership of the update group, if necessary]
        endwhile

        if [All members of the update group were acquired]
            [Apply write operations to the members of the update group]
            [Commit the transaction, loop back to the top, get new work]
        else
            [Roll back the transaction, loop back to the top, and try again]
        endif
    endwhile
```

Implementing STO in C++ – Design Choices

```
thread_function(shared_collection)
    [Create a transaction object]

    while [Work remains to be done]
        [Begin a new transaction]
        [Find target element E in shared_collection]
        [Determine the membership of the update group to the extent possible]

        while [Update group members remain unacquired AND acquisitions have all succeeded]
            [Starting with E, attempt to acquire the next unacquired member of the update group]
            [Revise the membership of the update group, if necessary]
        endwhile

        if [All members of the update group were acquired]
            [Apply write operations to the members of the update group]
            [Commit the transaction, loop back to the top, get new work]
        else
            [Roll back the transaction, loop back to the top, and try again]
        endif
    endwhile
```

Implementing STO in C++ – Design Choices

```
thread_function(shared_collection)
    [Create a transaction object]

    while [Work remains to be done]
        [Begin a new transaction]
        [Find target element E in shared_collection]
        [Determine the membership of the update group to the extent possible]

        while [Update group members remain unacquired AND acquisitions have all succeeded]
            [Starting with E, attempt to acquire the next unacquired member of the update group]
            [Revise the membership of the update group, if necessary]
        endwhile

        if [All members of the update group were acquired]
            [Apply write operations to the members of the update group]
            [Commit the transaction, loop back to the top, get new work]
        else
            [Roll back the transaction, loop back to the top, and try again]
        endif
    endwhile
```

Implementing STO in C++ – Design Choices

```
thread_function(shared_collection)
    [Create a transaction object]

    while [Work remains to be done]
        [Begin a new transaction]
        [Find target element E in shared_collection]
        [Determine the membership of the update group to the extent possible]

        while [Update group members remain unacquired AND acquisitions have all succeeded]
            [Starting with E, attempt to acquire the next unacquired member of the update group]
            [Revise the membership of the update group, if necessary]
        endwhile

        if [All members of the update group were acquired]
            [Apply write operations to the members of the update group]
            [Commit the transaction, loop back to the top, get new work]
        else
            [Roll back the transaction, loop back to the top, and try again]
        endif
    endwhile
```

Implementing STO in C++ – Design Choices

```
thread_function(shared_collection)
    [Create a transaction object]

    while [Work remains to be done]
        [Begin a new transaction]
        [Find target element E in shared_collection]
        [Determine the membership of the update group to the extent possible]

        while [Update group members remain unacquired AND acquisitions have all succeeded]
            [Starting with E, attempt to acquire the next unacquired member of the update group]
            [Revise the membership of the update group, if necessary]
        endwhile

        if [All members of the update group were acquired]
            [Apply write operations to the members of the update group]
            [Commit the transaction, loop back to the top, get new work]
        else
            [Roll back the transaction, loop back to the top, and try again]
        endif
    endwhile
```

Implementing STO in C++ – Design Choices

```
thread_function(shared_collection)
    [Create a transaction object]

    while [Work remains to be done]
        [Begin a new transaction]
        [Find target element E in shared_collection]
        [Determine the membership of the update group to the extent possible]

        while [Update group members remain unacquired AND acquisitions have all succeeded]
            [Starting with E, attempt to acquire the next unacquired member of the update group]
            [Revise the membership of the update group, if necessary]
        endwhile

        if [All members of the update group were acquired]
            [Apply write operations to the members of the update group]
            [Commit the transaction, loop back to the top, get new work]
        else
            [Roll back the transaction, loop back to the top, and try again]
        endif
    endwhile
```

Implementing STO in C++ – Design Choices

```
thread_function(shared_collection)
    [Create a transaction object]

    while [Work remains to be done]
        [Begin a new transaction]
        [Find target element E in shared_collection]
        [Determine the membership of the update group to the extent possible]

        while [Update group members remain unacquired AND acquisitions have all succeeded]
            [Starting with E, attempt to acquire the next unacquired member of the update group]
            [Revise the membership of the update group, if necessary]
        endwhile

        if [All members of the update group were acquired]
            [Apply write operations to the members of the update group]
            [Commit the transaction, loop back to the top, get new work]
        else
            [Roll back the transaction, loop back to the top, and try again]
        endif
    endwhile
```

Implementing STO in C++ – Design Choices

```
thread_function(shared_collection)
    [Create a transaction object]

    while [Work remains to be done]
        [Begin a new transaction]
        [Find target element E in shared_collection]
        [Determine the membership of the update group to the extent possible]

        while [Update group members remain unacquired AND acquisitions have all succeeded]
            [Starting with E, attempt to acquire the next unacquired member of the update group]
            [Revise the membership of the update group, if necessary]
        endwhile

        if [All members of the update group were acquired]
            [Apply write operations to the members of the update group]
            [Commit the transaction, loop back to the top, get new work]
        else
            [Roll back the transaction, loop back to the top, and try again]
        endif
    endwhile
```

Prerequisites

Inclusions, Type Aliases, Forward Declarations

```
//- Stuff we need.
//
#include <cstring>
#include <atomic>
#include <chrono>
#include <condition_variable>
#include <functional>
#include <future>
#include <mutex>
#include <random>
#include <thread>
#include <type_traits>
#include <vector>

using tsv_type     = uint64_t;      //- Timestamp value
using tx_id_type   = uint64_t;      //- Transaction ID
using item_id_type = uint64_t;      //- Item ID

class transaction;
class lockable_item;
class stopwatch;
```

Class Overview – stopwatch

```
//- Simple class to provide timing in test functions.
//
class stopwatch
{
  public:
    ~stopwatch() = default;

    stopwatch();
    stopwatch(stopwatch&&) = default;
    stopwatch(stopwatch const&) = default;

    stopwatch&  operator =(stopwatch&&) = default;
    stopwatch&  operator =(stopwatch const&) = default;

    template<class T>  T  seconds_elapsed() const;
    template<class T>  T  milliseconds_elapsed() const;

    void    start();
    void    stop();

  private:
    ...
};
```

Class `lockable_item`

Class Overview – lockable_item

```
class lockable_item
{
  public:
    lockable_item();

    item_id_type    id() const noexcept;
    tsv_type        last_tsv() const noexcept;

  private:
    friend class transaction;

    using atomic_tx_pointer = std::atomic<transaction*>;
    using atomic_item_id    = std::atomic<item_id_type>;

    atomic_tx_pointer   mp_owning_tx;   //- Pointer to transaction object that owns this object
    tsv_type            m_last_tsv;     //- Timestamp of last owner
    item_id_type        m_item_id;      //- For debugging/tracking/logging

    static  atomic_item_id      sm_item_id_generator;
};
```

Member Functions – lockable_item

```
inline
lockable_item::lockable_item()
:   mp_owning_tx(nullptr), m_last_tsv(0), m_item_id(++sm_item_id_generator)
{}


inline item_id_type
lockable_item::id() const noexcept
{
    return m_item_id;
}


inline tsv_type
lockable_item::last_tsv() const noexcept
{
    return m_last_tsv;
}


lockable_item::atomic_item_id   lockable_item::sm_item_id_generator = 0;
```

Class `transaction`

Class Overview – transaction

```
class transaction
{
  public:
    ~transaction();
    transaction(int log_level, FILE* fp=nullptr);

    tx_id_type  id() const noexcept;
    tsv_type    tsv() const noexcept;

    void    begin();
    void    commit();
    void    rollback();

    bool    acquire(lockable_item& item);

  private:
    ...

};
```

Class Overview – transaction

```
  ...

private:
  using item_ptr_list = std::vector<lockable_item*>;
  using mutex         = std::mutex;
  using tx_lock       = std::unique_lock<std::mutex>;
  using cond_var      = std::condition_variable;
  using atomic_tsv    = std::atomic<tsv_type>;
  using atomic_tx_id  = std::atomic<tx_id_type>;

  tx_id_type      m_tx_id;
  tsv_type        m_tx_tsv;
  item_ptr_list   m_item_ptrs;
  mutex           m_mutex;
  cond_var        m_cond;
  FILE*           m_fp;
  int             m_log_level;

  static  atomic_tsv      sm_tsv_generator;
  static  atomic_tx_id    sm_tx_id_generator;
  ...
```

Class Overview – transaction

```
    ...

  private:
    using item_ptr_list = std::vector<lockable_item*>;
    using mutex         = std::mutex;
    using tx_lock       = std::unique_lock<std::mutex>;
    using cond_var      = std::condition_variable;
    using atomic_tsv    = std::atomic<tsv_type>;
    using atomic_tx_id  = std::atomic<tx_id_type>;

    tx_id_type      m_tx_id;
    tsv_type        m_tx_tsv;
    item_ptr_list   m_item_ptrs;
    mutex           m_mutex;
    cond_var        m_cond;
    FILE*           m_fp;
    int             m_log_level;

    static  atomic_tsv      sm_tsv_generator;
    static  atomic_tx_id    sm_tx_id_generator;
    ...
```

Class Overview – transaction

```
    ...

  private:
    void    log_begin() const;
    void    log_commit() const;
    void    log_rollback() const;
    void    log_acquisition_success(lockable_item const& item) const;
    void    log_acquisition_failure(lockable_item const& item) const;
    void    log_acquisition_same(lockable_item const& item) const;
    void    log_acquisition_waiting(lockable_item const& item, transaction* p_curr_tx) const;
};


transaction::atomic_tsv     transaction::sm_tsv_generator   = 0;
transaction::atomic_tx_id   transaction::sm_tx_id_generator = 0;
```

Member Functions – transaction

```
transaction::transaction(int log_level, FILE* fp)
:   m_tx_id(++sm_tx_id_generator)
,   m_tx_tsv(0)
,   m_item_ptrs()
,   m_mutex()
,   m_cond()
,   m_fp(fp)
,   m_log_level(log_level)
{
    m_item_ptrs.reserve(100u);
}
```

Member Functions – `transaction`

```
inline tx_id_type
transaction::id() const noexcept
{
    return m_tx_id;
}


inline tsv_type
transaction::tsv() const noexcept
{
    return m_tx_tsv;
}
```

Member Functions – transaction::begin()

```
void
transaction::begin()
{
    log_begin();
    m_mutex.lock();
    m_tx_tsv = ++sm_tsv_generator;
    m_mutex.unlock();
}
```

Member Functions – `transaction::begin()`

```
void
transaction::begin()
{
    log_begin();
    m_mutex.lock();
    m_tx_tsv = ++sm_tsv_generator;
    m_mutex.unlock();
}
```

Member Functions – `transaction::begin()`

```
void
transaction::begin()
{
    log_begin();
    m_mutex.lock();
    m_tx_tsv = ++sm_tsv_generator;
    m_mutex.unlock();
}
```

Member Functions – `transaction::commit()`

```
void
transaction::commit()
{
    log_commit();

    tx_lock     lock(m_mutex);

    while (m_item_ptrs.size() != 0)
    {
        m_item_ptrs.back()->mp_owning_tx.store(nullptr);
        m_item_ptrs.pop_back();
    }
    m_cond.notify_all();
}
```

Member Functions – `transaction::commit()`

```
void
transaction::commit()
{
    log_commit();

    tx_lock     lock(m_mutex);

    while (m_item_ptrs.size() != 0)
    {
        m_item_ptrs.back()->mp_owning_tx.store(nullptr);
        m_item_ptrs.pop_back();
    }
    m_cond.notify_all();
}
```

Member Functions – `transaction::commit()`

```
void
transaction::commit()
{
    log_commit();

    tx_lock     lock(m_mutex);

    while (m_item_ptrs.size() != 0)
    {
        m_item_ptrs.back()->mp_owning_tx.store(nullptr);
        m_item_ptrs.pop_back();
    }
    m_cond.notify_all();
}
```

Member Functions – transaction::commit()

```
void
transaction::commit()
{
    log_commit();

    tx_lock     lock(m_mutex);

    while (m_item_ptrs.size() != 0)
    {
        m_item_ptrs.back()->mp_owning_tx.store(nullptr);
        m_item_ptrs.pop_back();
    }
    m_cond.notify_all();
}
```

Member Functions – transaction::commit()

```
void
transaction::commit()
{
    log_commit();

    tx_lock     lock(m_mutex);

    while (m_item_ptrs.size() != 0)
    {
        m_item_ptrs.back()->mp_owning_tx.store(nullptr);
        m_item_ptrs.pop_back();
    }
    m_cond.notify_all();
}
```

Member Functions – `transaction::commit()`

```
void
transaction::commit()
{
    log_commit();

    tx_lock     lock(m_mutex);

    while (m_item_ptrs.size() != 0)
    {
        m_item_ptrs.back()->mp_owning_tx.store(nullptr);
        m_item_ptrs.pop_back();
    }
    m_cond.notify_all();
}
```

Member Functions – transaction::acquire()

```
acquire(lockable_item item)
    while (true)
        if [I acquire item]
            [Add item to the list of items that I own]

            if [My timestamp is newer than item’s last timestamp]
                [Set item’s timestamp to my timestamp]
                [Success – return and caller keeps going]
            else
                [Failure – return and caller rolls back]
            endif
        else
            if [I already own item]
                [Success – return and caller keeps going]
            else
                [I’ll wait until the current owner releases item and then re-try]
            endif
        endif
    endwhile
```

Member Functions – transaction::acquire()

```
acquire(lockable_item item)
    while (true)
        if [I acquire item]
            [Add item to the list of items that I own]

            if [My timestamp is newer than item’s last timestamp]
                [Set item’s timestamp to my timestamp]
                [Success – return and caller keeps going]
            else
                [Failure – return and caller rolls back]
            endif
        else
            if [I already own item]
                [Success – return and caller keeps going]
            else
                [I’ll wait until the current owner releases item and then re-try]
            endif
        endif
    endwhile
```

Member Functions – transaction::acquire()

```
acquire(lockable_item item)
    while (true)
        if [I acquire item]
            [Add item to the list of items that I own]

            if [My timestamp is newer than item’s last timestamp]
                [Set item’s timestamp to my timestamp]
                [Success – return and caller keeps going]
            else
                [Failure – return and caller rolls back]
            endif
        else
            if [I already own item]
                [Success – return and caller keeps going]
            else
                [I’ll wait until the current owner releases item and then re-try]
            endif
        endif
    endwhile
```

Member Functions – transaction::acquire()

```
acquire(lockable_item item)
    while (true)
        if [I acquire item]
            [Add item to the list of items that I own]

            if [My timestamp is newer than item’s last timestamp]
                [Set item’s timestamp to my timestamp]
                [Success – return and caller keeps going]
            else
                [Failure – return and caller rolls back]
            endif
        else
            if [I already own item]
                [Success – return and caller keeps going]
            else
                [I’ll wait until the current owner releases item and then re-try]
            endif
        endif
    endwhile
```

Member Functions – transaction::acquire()

```
acquire(lockable_item item)
    while (true)
        if [I acquire item]
            [Add item to the list of items that I own]

            if [My timestamp is newer than item’s last timestamp]
                [Set item’s timestamp to my timestamp]
                [Success – return and caller keeps going]
            else
                [Failure – return and caller rolls back]
            endif
        else
            if [I already own item]
                [Success – return and caller keeps going]
            else
                [I’ll wait until the current owner releases item and then re-try]
            endif
        endif
    endwhile
```

Member Functions – transaction::acquire()

```
acquire(lockable_item item)
    while (true)
        if [I acquire item]
            [Add item to the list of items that I own]

            if [My timestamp is newer than item’s last timestamp]
                [Set item’s timestamp to my timestamp]
                [Success – return and caller keeps going]
            else
                [Failure – return and caller rolls back]
            endif
        else
            if [I already own item]
                [Success – return and caller keeps going]
            else
                [I’ll wait until the current owner releases item and then re-try]
            endif
        endif
    endwhile
```

Member Functions – transaction::acquire()

```
acquire(lockable_item item)
    while (true)
        if [I acquire item]
            [Add item to the list of items that I own]

            if [My timestamp is newer than item’s last timestamp]
                [Set item’s timestamp to my timestamp]
                [Success – return and caller keeps going]
            else
                [Failure – return and caller rolls back]
            endif
        else
            if [I already own item]
                [Success – return and caller keeps going]
            else
                [I’ll wait until the current owner releases item and then re-try]
            endif
        endif
    endwhile
```

Member Functions – transaction::acquire()

```
acquire(lockable_item item)
    while (true)
        if [I acquire item]
            [Add item to the list of items that I own]

            if [My timestamp is newer than item’s last timestamp]
                [Set item’s timestamp to my timestamp]
                [Success – return and caller keeps going]
            else
                [Failure – return and caller rolls back]
            endif
        else
            if [I already own item]
                [Success – return and caller keeps going]
            else
                [I’ll wait until the current owner releases item and then re-try]
            endif
        endif
    endwhile
```

Member Functions – `transaction::acquire()`

```
acquire(lockable_item item)
    while (true)
        if [I acquire item]
            [Add item to the list of items that I own]

            if [My timestamp is newer than item’s last timestamp]
                [Set item’s timestamp to my timestamp]
                [Success – return and caller keeps going]
            else
                [Failure – return and caller rolls back]
            endif
        else
            if [I already own item]
                [Success – return and caller keeps going]
            else
                [I’ll wait until the current owner releases item and then re-try]
            endif
        endif
    endwhile
```

Member Functions – `transaction::acquire()`

```
acquire(lockable_item item)
    while (true)
        if [I acquire item]
            [Add item to the list of items that I own]

            if [My timestamp is newer than item’s last timestamp]
                [Set item’s timestamp to my timestamp]
                [Success – return and caller keeps going]
            else
                [Failure – return and caller rolls back]
            endif
        else
            if [I already own item]
                [Success – return and caller keeps going]
            else
                [I’ll wait until the current owner releases item and then re-try]
            endif
        endif
    endwhile
```

Member Functions – transaction::acquire()

```
acquire(lockable_item item)
    while (true)
        if [I acquire item]
            [Add item to the list of items that I own]

            if [My timestamp is newer than item’s last timestamp]
                [Set item’s timestamp to my timestamp]
                [Success – return and caller keeps going]
            else
                [Failure – return and caller rolls back]
            endif
        else
            if [I already own item]
                [Success – return and caller keeps going]
            else
                [I’ll wait until the current owner releases item and then re-try]
            endif
        endif
    endwhile
```

Member Functions – transaction::acquire()

```
bool
transaction::acquire(lockable_item& item)
{
    while (true)
    {
        transaction*    p_curr_tx = nullptr;

        if (item.mp_owning_tx.compare_exchange_strong(p_curr_tx, this))
        {
            m_item_ptrs.push_back(&item);

            if (m_tx_tsv > item.m_last_tsv)
            {
                log_acquisition_success(item);
                item.m_last_tsv = m_tx_tsv;
                return true;
            }

            ...
```

Member Functions – transaction::acquire()

```
acquire(lockable_item item)
    while (true)
        if [I acquire item]
            [Add item to the list of items that I own]

            if [My timestamp is newer than item’s last timestamp]
                [Set item’s timestamp to my timestamp]
                [Success – return and caller keeps going]
            else
                [Failure – return and caller rolls back]
            endif
        else
            if [I already own item]
                [Success – return and caller keeps going]
            else
                [I’ll wait until the current owner releases item and then re-try]
            endif
        endif
    endwhile
```

Member Functions – transaction::acquire()

```
    while (true)
    {
        transaction*    p_curr_tx = nullptr;

        if (item.mp_owning_tx.compare_exchange_strong(p_curr_tx, this))
        {
            m_item_ptrs.push_back(&item);

            if (m_tx_tsv > item.m_last_tsv)
            {
                log_acquisition_success(item);
                item.m_last_tsv = m_tx_tsv;
                return true;
            }
            else
            {
                log_acquisition_failure(item);
                return false;
            }
        }
        else
        ...
```

Member Functions – transaction::acquire()

```
    while (true)
    {
        transaction*    p_curr_tx = nullptr;

        if (item.mp_owning_tx.compare_exchange_strong(p_curr_tx, this))
        {
            m_item_ptrs.push_back(&item);

            if (m_tx_tsv > item.m_last_tsv)
            {
                log_acquisition_success(item);
                item.m_last_tsv = m_tx_tsv;
                return true;
            }
            else
            {
                log_acquisition_failure(item);
                return false;
            }
        }
        else
        ...
```

Atomic Compare and Exchange

```
temp<class T>
inline bool
atomic<T>::compare_exchange_strong(T& expected, T desired,
                                   std::memory_order mo = std::memory_order_seq_cst ) noexcept;
```

- Paraphrasing cppreference.com:
  - Atomically compares the representation of `*this` with that of `expected`, and if those are bitwise-equal, replaces the representation of `*this` with `desired` and returns `true`;
  - Otherwise, loads the actual value stored in `*this` into `expected` and returns `false`.

Member Functions – transaction::acquire()

```
while (true)
{
    transaction*    p_curr_tx = nullptr;

    if (item.mp_owning_tx.compare_exchange_strong(p_curr_tx, this))
    {
        m_item_ptrs.push_back(&item);

        if (m_tx_tsv > item.m_last_tsv)
        {
            log_acquisition_success(item);
            item.m_last_tsv = m_tx_tsv;
            return true;
        }
        else
        {
            log_acquisition_failure(item);
            return false;
        }
    }
    else
    ...
```

Member Functions – `transaction::acquire()`

```
while (true)
{
    transaction*    p_curr_tx = nullptr;

    if (item.mp_owning_tx.compare_exchange_strong(p_curr_tx, this))
    {
        m_item_ptrs.push_back(&item);

        if (m_tx_tsv > item.m_last_tsv)
        {
            log_acquisition_success(item);
            item.m_last_tsv = m_tx_tsv;
            return true;
        }
        else
        {
            log_acquisition_failure(item);
            return false;
        }
    }
    else
    ...
```

Member Functions – transaction::acquire()

```
acquire(lockable_item item)
    while (true)
        if [I acquire item]
            [Add item to the list of items that I own]

            if [My timestamp is newer than item’s last timestamp]
                [Set item’s timestamp to my timestamp]
                [Success – return and caller keeps going]
            else
                [Failure – return and caller rolls back]
            endif
        else
            if [I already own item]
                [Success – return and caller keeps going]
            else
                [I’ll wait until the current owner releases item and then re-try]
            endif
        endif
    endwhile
```

Member Functions – transaction::acquire()

```
        ...

        if (m_tx_tsv > item.m_last_tsv)
        {
            log_acquisition_success(item);
            item.m_last_tsv = m_tx_tsv;
            return true;
        }
        else
        {
            log_acquisition_failure(item);
            return false;
        }
    }
    else
    {
        if (p_curr_tx == this)
        {
            log_acquisition_same(item);
            return true;
        }
        ...
```

Member Functions – transaction::acquire()

```
        ...

        if (m_tx_tsv > item.m_last_tsv)
        {
            log_acquisition_success(item);
            item.m_last_tsv = m_tx_tsv;
            return true;
        }
        else
        {
            log_acquisition_failure(item);
            return false;
        }
    }
    else
    {
        if (p_curr_tx == this)
        {
            log_acquisition_same(item);
            return true;
        }
        ...
```

Member Functions – `transaction::acquire()`

```
            ...

            if (m_tx_tsv > item.m_last_tsv)
            {
                log_acquisition_success(item);
                item.m_last_tsv = m_tx_tsv;
                return true;
            }
            else
            {
                log_acquisition_failure(item);
                return false;
            }
        }
        else
        {
            if (p_curr_tx == this)
            {
                log_acquisition_same(item);
                return true;
            }
            ...
```

Member Functions – transaction::acquire()

```
        ...

        if (m_tx_tsv > item.m_last_tsv)
        {
            log_acquisition_success(item);
            item.m_last_tsv = m_tx_tsv;
            return true;
        }
        else
        {
            log_acquisition_failure(item);
            return false;
        }
    }
    else
    {
        if (p_curr_tx == this)
        {
            log_acquisition_same(item);
            return true;
        }
        ...
```

Member Functions – transaction::acquire()

```
acquire(lockable_item item)
    while (true)
        if [I acquire item]
            [Add item to the list of items that I own]

            if [My timestamp is newer than item’s last timestamp]
                [Set item’s timestamp to my timestamp]
                [Success – return and caller keeps going]
            else
                [Failure – return and caller rolls back]
            endif
        else
            if [I already own item]
                [Success – return and caller keeps going]
            else
                [I’ll wait until the current owner releases item and then re-try]
            endif
        endif
    endwhile
```

Member Functions – `transaction::acquire()`

```
        ...

        if (m_tx_tsv > item.m_last_tsv)
        {
            log_acquisition_success(item);
            item.m_last_tsv = m_tx_tsv;
            return true;
        }
        else
        {
            log_acquisition_failure(item);
            return false;
        }
    }
    else
    {
        if (p_curr_tx == this)
        {
            log_acquisition_same(item);
            return true;
        }
        ...
```

Member Functions – transaction::acquire()

```
...
if (p_curr_tx == this)
{
    log_acquisition_same(item);
    return true;
}
else
{
    log_acquisition_waiting(item, p_curr_tx);

    tx_lock     lock(p_curr_tx->m_mutex);

    while (item.mp_owning_tx.load() == p_curr_tx)
    {
        if (p_curr_tx->m_tx_tsv > m_tx_tsv)
        {
            log_acquisition_failure(item);
            return false;
        }
        p_curr_tx->m_cond.wait(lock);
    }
}
```

Member Functions – transaction::acquire()

```
...
if (p_curr_tx == this)
{
    log_acquisition_same(item);
    return true;
}
else
{
    log_acquisition_waiting(item, p_curr_tx);

    tx_lock     lock(p_curr_tx->m_mutex);

    while (item.mp_owning_tx.load() == p_curr_tx)
    {
        if (p_curr_tx->m_tx_tsv > m_tx_tsv)
        {
            log_acquisition_failure(item);
            return false;
        }
        p_curr_tx->m_cond.wait(lock);
    }
}
```

Member Functions – transaction::acquire()

```
...
if (p_curr_tx == this)
{
    log_acquisition_same(item);
    return true;
}
else
{
    log_acquisition_waiting(item, p_curr_tx);

    tx_lock     lock(p_curr_tx->m_mutex);

    while (item.mp_owning_tx.load() == p_curr_tx)
    {
        if (p_curr_tx->m_tx_tsv > m_tx_tsv)
        {
            log_acquisition_failure(item);
            return false;
        }
        p_curr_tx->m_cond.wait(lock);
    }
}
```

Member Functions – `transaction::acquire()`

```
...
if (p_curr_tx == this)
{
    log_acquisition_same(item);
    return true;
}
else
{
    log_acquisition_waiting(item, p_curr_tx);

    tx_lock     lock(p_curr_tx->m_mutex);

    while (item.mp_owning_tx.load() == p_curr_tx)
    {
        if (p_curr_tx->m_tx_tsv > m_tx_tsv)
        {
            log_acquisition_failure(item);
            return false;
        }
        p_curr_tx->m_cond.wait(lock);
    }
}
```

Member Functions – transaction::acquire()

```
...
if (p_curr_tx == this)
{
    log_acquisition_same(item);
    return true;
}
else
{
    log_acquisition_waiting(item, p_curr_tx);

    tx_lock     lock(p_curr_tx->m_mutex);

    while (item.mp_owning_tx.load() == p_curr_tx)
    {
        if (p_curr_tx->m_tx_tsv > m_tx_tsv)
        {
            log_acquisition_failure(item);
            return false;
        }
        p_curr_tx->m_cond.wait(lock);
    }
}
```

Member Functions – `transaction::acquire()`

```
...
if (p_curr_tx == this)
{
    log_acquisition_same(item);
    return true;
}
else
{
    log_acquisition_waiting(item, p_curr_tx);

    tx_lock     lock(p_curr_tx->m_mutex);

    while (item.mp_owning_tx.load() == p_curr_tx)
    {
        if (p_curr_tx->m_tx_tsv > m_tx_tsv)
        {
            log_acquisition_failure(item);
            return false;
        }
        p_curr_tx->m_cond.wait(lock);
    }
}
```

Member Functions – transaction::acquire()

```
...
if (p_curr_tx == this)
{
    log_acquisition_same(item);
    return true;
}
else
{
    log_acquisition_waiting(item, p_curr_tx);

    tx_lock     lock(p_curr_tx->m_mutex);

    while (item.mp_owning_tx.load() == p_curr_tx)
    {
        if (p_curr_tx->m_tx_tsv > m_tx_tsv)
        {
            log_acquisition_failure(item);
            return false;
        }
        p_curr_tx->m_cond.wait(lock);
    }
}
```

Member Functions – transaction::acquire()

```
...
if (p_curr_tx == this)
{
    log_acquisition_same(item);
    return true;
}
else
{
    log_acquisition_waiting(item, p_curr_tx);

    tx_lock     lock(p_curr_tx->m_mutex);

    while (item.mp_owning_tx.load() == p_curr_tx)
    {
        if (p_curr_tx->m_tx_tsv > m_tx_tsv)
        {
            log_acquisition_failure(item);
            return false;
        }
        p_curr_tx->m_cond.wait(lock);
    }
}
```

Member Functions – transaction::acquire()

```
...
if (p_curr_tx == this)
{
    log_acquisition_same(item);
    return true;
}
else
{
    log_acquisition_waiting(item, p_curr_tx);

    tx_lock     lock(p_curr_tx->m_mutex);

    while (item.mp_owning_tx.load() == p_curr_tx)
    {
        if (p_curr_tx->m_tx_tsv > m_tx_tsv)
        {
            log_acquisition_failure(item);
            return false;
        }
        p_curr_tx->m_cond.wait(lock);
    }
}
```

Member Functions – transaction::commit()

```
void
transaction::commit()
{
    log_commit();

    tx_lock     lock(m_mutex);

    while (m_item_ptrs.size() != 0)
    {
        m_item_ptrs.back()->mp_owning_tx.store(nullptr);
        m_item_ptrs.pop_back();
    }
    m_cond.notify_all();
}
```

Member Functions – `transaction::acquire()`

```
            ...

            else
            {
                log_acquisition_waiting(item, p_curr_tx);

                tx_lock lock(p_curr_tx->m_mutex);

                while (item.mp_owning_tx.load() == p_curr_tx)
                {
                    if (p_curr_tx->m_tx_tsv > m_tx_tsv)
                    {
                        log_acquisition_failure(item);
                        return false;
                    }
                    p_curr_tx->m_cond.wait(lock);
                }
            }
        }
    }
}
```

Member Functions – transaction::acquire()

```
while (true)
{
    transaction*    p_curr_tx = nullptr;

    if (item.mp_owning_tx.compare_exchange_strong(p_curr_tx, this))
    {
        m_item_ptrs.push_back(&item);

        if (m_tx_tsv > item.m_last_tsv)
        {
            log_acquisition_success(item);
            item.m_last_tsv = m_tx_tsv;
            return true;
        }
        else
        {
            log_acquisition_failure(item);
            return false;
        }
    }
    else
    ...
```

Testing

Test Strategy

- Create functions to update a collection of shared items
  - Make sure that data races are possible and can be detected
- Measure single-threaded updates – baseline
- Measure multi-threaded updates with induced data races
- Measure multi-threaded updates guarded by a single critical section
- Measure multi-threaded transactional updates

Testing – Includes, etc.

```cpp
#include "transaction.hpp"

using namespace tx;
using namespace std;
using namespace std::chrono_literals;

//- Forward declarations and common type aliases common to the test functions below().
//
struct test_item;

using item_list  = std::vector<test_item>;
using index_list = std::vector<size_t>;

using entropy  = random_device;
using prn_gen  = mt19937_64;
using int_dist = uniform_int_distribution<>;
using hasher   = hash<string_view>;
```

Testing – test_item

```
//- An updatable type susceptible to data races.
//
struct test_item : public lockable_item
{
    static constexpr size_t     buf_size = 32;

    char    ma_chars[buf_size];

    void    st_update(FILE* fp, prn_gen& gen, int_dist& char_dist);
    void    tx_update(transaction const& tx, FILE* fp, prn_gen& gen, int_dist& char_dist);
};
```

Testing – test_item::st_update()

```
//- Updates on-board data for the non-STO tests.  Checks to see if a race has occurred.
//
void
test_item::st_update(FILE* fp, prn_gen& gen, int_dist& char_dist)
{
    char                local_chars[buf_size];
    string_view         local_view(local_chars, buf_size);
    string_view         shared_view(ma_chars, buf_size);
    hasher              hash;

    for (size_t i = 0;  i < buf_size;  ++i)
    {
        local_chars[i] = ma_chars[i] = (char) char_dist(gen);
    }

    if (hash(shared_view) != hash(local_view))
    {
        fprintf(fp, "RACE FOUND!,  item %zd\n", this->id());
    }
}
```

Testing – test_item::st_update()

```cpp
//- Updates on-board data for the non-STO tests.  Checks to see if a race has occurred.
//
void
test_item::st_update(FILE* fp, prn_gen& gen, int_dist& char_dist)
{
    char                local_chars[buf_size];
    string_view         local_view(local_chars, buf_size);
    string_view         shared_view(ma_chars, buf_size);
    hasher              hash;

    for (size_t i = 0;  i < buf_size;  ++i)
    {
        local_chars[i] = ma_chars[i] = (char) char_dist(gen);
    }

    if (hash(shared_view) != hash(local_view))
    {
        fprintf(fp, "RACE FOUND!,  item %zd\n", this->id());
    }
}
```

Testing – test_item::st_update()

```
//- Updates on-board data for the non-STO tests.  Checks to see if a race has occurred.
//
void
test_item::st_update(FILE* fp, prn_gen& gen, int_dist& char_dist)
{
    char                local_chars[buf_size];
    string_view         local_view(local_chars, buf_size);
    string_view         shared_view(ma_chars, buf_size);
    hasher              hash;

    for (size_t i = 0;  i < buf_size;  ++i)
    {
        local_chars[i] = ma_chars[i] = (char) char_dist(gen);
    }

    if (hash(shared_view) != hash(local_view))
    {
        fprintf(fp, "RACE FOUND!,  item %zd\n", this->id());
    }
}
```

Testing – test_item::st_update()

```
//- Updates on-board data for the non-STO tests.  Checks to see if a race has occurred.
//
void
test_item::st_update(FILE* fp, prn_gen& gen, int_dist& char_dist)
{
    char                local_chars[buf_size];
    string_view         local_view(local_chars, buf_size);
    string_view         shared_view(ma_chars, buf_size);
    hasher              hash;

    for (size_t i = 0;  i < buf_size;  ++i)
    {
        local_chars[i] = ma_chars[i] = (char) char_dist(gen);
    }

    if (hash(shared_view) != hash(local_view))
    {
        fprintf(fp, "RACE FOUND!,  item %zd\n", this->id());
    }
}
```

Testing – test_item::tx_update()

```
//- Updates on-board data for the STO tests.  Checks to see if a race has occurred.
//
void
test_item::tx_update(transaction const& tx, FILE* fp, prn_gen& gen, int_dist& char_dist)
{
    char            local_chars[buf_size];
    string_view     local_view(local_chars, buf_size);
    string_view     shared_view(ma_chars, buf_size);
    hasher          hash;

    for (size_t i = 0;  i < buf_size;  ++i)
    {
        local_chars[i] = ma_chars[i] = (char) char_dist(gen);
    }

    if (hash(shared_view) != hash(local_view))
    {
        fprintf(fp, "RACE FOUND!,  TX %zd  item %zd\n", tx.id(), this->id());
    }
}
```

Testing – test_item::tx_update()

```
//- Updates on-board data for the STO tests.  Checks to see if a race has occurred.
//
void
test_item::tx_update(transaction const& tx, FILE* fp, prn_gen& gen, int_dist& char_dist)
{
    char            local_chars[buf_size];
    string_view     local_view(local_chars, buf_size);
    string_view     shared_view(ma_chars, buf_size);
    hasher          hash;

    for (size_t i = 0;  i < buf_size;  ++i)
    {
        local_chars[i] = ma_chars[i] = (char) char_dist(gen);
    }

    if (hash(shared_view) != hash(local_view))
    {
        fprintf(fp, "RACE FOUND!,  TX %zd  item %zd\n", tx.id(), this->id());
    }
}
```

Testing – test_item::tx_update()

```
//- Updates on-board data for the STO tests.  Checks to see if a race has occurred.
//
void
test_item::tx_update(transaction const& tx, FILE* fp, prn_gen& gen, int_dist& char_dist)
{
    char            local_chars[buf_size];
    string_view     local_view(local_chars, buf_size);
    string_view     shared_view(ma_chars, buf_size);
    hasher          hash;

    for (size_t i = 0;  i < buf_size;  ++i)
    {
        local_chars[i] = ma_chars[i] = (char) char_dist(gen);
    }

    if (hash(shared_view) != hash(local_view))
    {
        fprintf(fp, "RACE FOUND!,  TX %zd  item %zd\n", tx.id(), this->id());
    }
}
```

Testing – test_item::tx_update()

```
//- Updates on-board data for the STO tests.  Checks to see if a race has occurred.
//
void
test_item::tx_update(transaction const& tx, FILE* fp, prn_gen& gen, int_dist& char_dist)
{
    char                local_chars[buf_size];
    string_view         local_view(local_chars, buf_size);
    string_view         shared_view(ma_chars, buf_size);
    hasher              hash;

    for (size_t i = 0;  i < buf_size;  ++i)
    {
        local_chars[i] = ma_chars[i] = (char) char_dist(gen);
    }

    if (hash(shared_view) != hash(local_view))
    {
        fprintf(fp, "RACE FOUND!,  TX %zd  item %zd\n", tx.id(), this->id());
    }
}
```

Testing – test_item::tx_update()

```
//- Updates on-board data for the STO tests.  Checks to see if a race has occurred.
//
void
test_item::tx_update(transaction const& tx, FILE* fp, prn_gen& gen, int_dist& char_dist)
{
    char            local_chars[buf_size];
    string_view     local_view(local_chars, buf_size);
    string_view     shared_view(ma_chars, buf_size);
    hasher          hash;

    for (size_t i = 0;  i < buf_size;  ++i)
    {
        local_chars[i] = ma_chars[i] = (char) char_dist(gen);
    }

    if (hash(shared_view) != hash(local_view))
    {
        fprintf(fp, "RACE FOUND!,  TX %zd  item %zd\n", tx.id(), this->id());
    }
}
```

Single-threaded Element Access Test

Single-threaded Element Access Test

```
void
st_access_test(item_list& items, FILE* fp, size_t tx_count, size_t refs_count)
{
    entropy     rd;
    prn_gen     gen(rd());
    int_dist    refs_index_dist(0, (int)(items.size()-1));
    int_dist    refs_count_dist(1, (int) refs_count);
    int_dist    char_dist(0, 127);

    stopwatch   sw;
    index_list  indices;
    size_t      index;

    sw.start();

    for (size_t i = 0;  i < tx_count;  ++i)
    {
        //- Compute the size of the update group
        //
        indices.clear();
        refs_count = refs_count_dist(gen);
        ...
```

Single-threaded Element Access Test

```
void
st_access_test(item_list& items, FILE* fp, size_t tx_count, size_t refs_count)
{
    entropy     rd;
    prn_gen     gen(rd());
    int_dist    refs_index_dist(0, (int)(items.size()-1));
    int_dist    refs_count_dist(1, (int) refs_count);
    int_dist    char_dist(0, 127);

    stopwatch   sw;
    index_list  indices;
    size_t      index;

    sw.start();

    for (size_t i = 0;  i < tx_count;  ++i)
    {
        //- Compute the size of the update group
        //
        indices.clear();
        refs_count = refs_count_dist(gen);
        ...
```

Single-threaded Element Access Test

```
    ...

    for (size_t i = 0;  i < tx_count;  ++i)
    {
        //- Compute the size of the update group
        //
        indices.clear();
        refs_count = refs_count_dist(gen);

        //- Compute the membership of the update group
        //
        for (size_t j = 0;  j < refs_count;  ++j)
        {
            index = refs_index_dist(gen);
            indices.push_back(index);
        }

        ...
```

Single-threaded Element Access Test

```
...

for (size_t i = 0;  i < tx_count;  ++i)
{
    //- Compute the size of the update group
    //
    indices.clear();
    refs_count = refs_count_dist(gen);

    //- Compute the membership of the update group
    //
    for (size_t j = 0;  j < refs_count;  ++j)
    {
        index = refs_index_dist(gen);
        indices.push_back(index);
    }

    ...
```

Single-threaded Element Access Test

```
        ...

        //- Compute the membership of the update group
        //
        for (size_t j = 0;  j < refs_count;  ++j)
        {
            index = refs_index_dist(gen);
            indices.push_back(index);
        }

        //- Modify the members of the update group
        //
        for (size_t j = 0;  j < refs_count;  ++j)
        {
            index = indices[j];
            items[index].st_update(fp, gen, char_dist);
        }
    }

    sw.stop();
    fprintf(fp, "TX 0 took %d msec\n", sw.milliseconds_elapsed<int>());
}
```

Multi-threaded Element Access Test

Multi-threaded Element Access Test

```
void
mx_access_test(item_list& items, FILE* fp, size_t tx_count, size_t refs_count)
{
    static mutex    mtx;

    entropy     rd;
    prn_gen     gen(rd());
    int_dist    refs_index_dist(0, (int)(items.size()-1));
    int_dist    refs_count_dist(1, (int) refs_count);
    int_dist    char_dist(0, 127);

    stopwatch   sw;
    index_list  indices;
    size_t      index;

    sw.start();

    for (size_t i = 0;  i < tx_count;  ++i)
    {
        ...
```

Multi-threaded Element Access Test

```
void
mx_access_test(item_list& items, FILE* fp, size_t tx_count, size_t refs_count)
{
    static mutex    mtx;

    entropy     rd;
    prn_gen     gen(rd());
    int_dist    refs_index_dist(0, (int)(items.size()-1));
    int_dist    refs_count_dist(1, (int) refs_count);
    int_dist    char_dist(0, 127);

    stopwatch   sw;
    index_list  indices;
    size_t      index;

    sw.start();

    for (size_t i = 0;  i < tx_count;  ++i)
    {
        ...
```

Multi-threaded Element Access Test

```
void
mx_access_test(item_list& items, FILE* fp, size_t tx_count, size_t refs_count)
{
    static mutex    mtx;

    entropy     rd;
    prn_gen     gen(rd());
    int_dist    refs_index_dist(0, (int)(items.size()-1));
    int_dist    refs_count_dist(1, (int) refs_count);
    int_dist    char_dist(0, 127);

    stopwatch   sw;
    index_list  indices;
    size_t      index;

    sw.start();

    for (size_t i = 0;  i < tx_count;  ++i)
    {
        ...
```

Multi-threaded Element Access Test

```
...

for (size_t i = 0;  i < tx_count;  ++i)
{
    //- Compute the size of the update group
    //
    indices.clear();
    refs_count = refs_count_dist(gen);

    //- Compute the membership of the update group
    //
    for (size_t j = 0;  j < refs_count;  ++j)
    {
        index = refs_index_dist(gen);
        indices.push_back(index);
    }

    ...
```

Multi-threaded Element Access Test

```
...

for (size_t i = 0;  i < tx_count;  ++i)
{
    //- Compute the size of the update group
    //
    indices.clear();
    refs_count = refs_count_dist(gen);

    //- Compute the membership of the update group
    //
    for (size_t j = 0;  j < refs_count;  ++j)
    {
        index = refs_index_dist(gen);
        indices.push_back(index);
    }

    ...
```

Multi-threaded Element Access Test

```
...

//- Compute the membership of the update group
//
for (size_t j = 0;  j < refs_count;  ++j)
{
    index = refs_index_dist(gen);
    indices.push_back(index);
}

mtx.lock();

//- Modify the members of the update group
//
for (size_t j = 0;  j < refs_count;  ++j)
{
    index = indices[j];
    items[index].st_update(fp, gen, char_dist);
}

mtx.unlock();
...
```

Multi-threaded Element Access Test

```
...

//- Compute the membership of the update group
//
for (size_t j = 0;  j < refs_count;  ++j)
{
    index = refs_index_dist(gen);
    indices.push_back(index);
}

mtx.lock();

//- Modify the members of the update group
//
for (size_t j = 0;  j < refs_count;  ++j)
{
    index = indices[j];
    items[index].st_update(fp, gen, char_dist);
}

mtx.unlock();
...
```

Multi-threaded Element Access Test

```
        mtx.lock();

        //- Modify the members of the update group
        //
        for (size_t j = 0;  j < refs_count;  ++j)
        {
            index = indices[j];
            items[index].st_update(fp, gen, char_dist);
        }

        mtx.unlock();
    }

    sw.stop();
    fprintf(fp, "TX 0 took %d msec\n", sw.milliseconds_elapsed<int>());
}
```

Transactional Element Access Test

Transactional Element Access Test

```
void
tx_access_test(item_list& items, FILE* fp, size_t tx_count, size_t refs_count)
{
    entropy     rd;
    prn_gen     gen(rd());
    int_dist    refs_index_dist(0, (int)(items.size()-1));
    int_dist    refs_count_dist(1, (int) refs_count);
    int_dist    char_dist(0, 127);

    stopwatch   sw;
    index_list  indices;
    size_t      index;

    transaction tx(1, fp);
    bool        acquired;

    sw.start();

    for (size_t i = 0;  i < tx_count;  ++i)
    {
        ...
```

Transactional Element Access Test

```
void
tx_access_test(item_list& items, FILE* fp, size_t tx_count, size_t refs_count)
{
    entropy     rd;
    prn_gen     gen(rd());
    int_dist    refs_index_dist(0, (int)(items.size()-1));
    int_dist    refs_count_dist(1, (int) refs_count);
    int_dist    char_dist(0, 127);

    stopwatch   sw;
    index_list  indices;
    size_t      index;

    transaction tx(1, fp);
    bool        acquired;

    sw.start();

    for (size_t i = 0;  i < tx_count;  ++i)
    {
        ...
```

Transactional Element Access Test

```
void
tx_access_test(item_list& items, FILE* fp, size_t tx_count, size_t refs_count)
{
    entropy     rd;
    prn_gen     gen(rd());
    int_dist    refs_index_dist(0, (int)(items.size()-1));
    int_dist    refs_count_dist(1, (int) refs_count);
    int_dist    char_dist(0, 127);

    stopwatch   sw;
    index_list  indices;
    size_t      index;

    transaction tx(1, fp);
    bool        acquired;

    sw.start();

    for (size_t i = 0;  i < tx_count;  ++i)
    {
        ...
```

Transactional Element Access Test

```
    ...

    for (size_t i = 0;  i < tx_count;  ++i)
    {
        //- Compute the size of the update group
        //
        indices.clear();
        refs_count = refs_count_dist(gen);

        //- Compute the membership of the update group
        //
        for (size_t j = 0;  j < refs_count;  ++j)
        {
            index = refs_index_dist(gen);
            indices.push_back(index);
        }

        ...
```

Transactional Element Access Test

```
...

for (size_t i = 0;  i < tx_count;  ++i)
{
    //- Compute the size of the update group
    //
    indices.clear();
    refs_count = refs_count_dist(gen);

    //- Compute the membership of the update group
    //
    for (size_t j = 0;  j < refs_count;  ++j)
    {
        index = refs_index_dist(gen);
        indices.push_back(index);
    }

    ...
```

Transactional Element Access Test

```
...

//- Compute the membership of the update group
//
for (size_t j = 0;  j < refs_count;  ++j)
{
    index = refs_index_dist(gen);
    indices.push_back(index);
}

tx.begin();
acquired = true;

//- Acquire the members of the update group
//
for (size_t j = 0;  acquired  &&  j < refs_count;  ++j)
{
    index    = indices[j];
    acquired = (acquired && tx.acquire(items[index]));
}

...
```

Transactional Element Access Test

```
...

//- Acquire the members of the update group
//
for (size_t j = 0;  acquired  &&  j < refs_count;  ++j)
{
    index    = indices[j];
    acquired = (acquired && tx.acquire(items[index]));
}

//- Modify the members of the update group
//
if (acquired)
{
    for (size_t j = 0;  j < refs_count;  ++j)
    {
        index = indices[j];
        items[index].tx_update(tx, fp, gen, char_dist);
    }
    tx.commit();
}
...
```

Transactional Element Access Test

```
...

//- Acquire the members of the update group
//
for (size_t j = 0;  acquired  &&  j < refs_count;  ++j)
{
    index    = indices[j];
    acquired = (acquired && tx.acquire(items[index]));
}

//- Modify the members of the update group
//
if (acquired)
{
    for (size_t j = 0;  j < refs_count;  ++j)
    {
        index = indices[j];
        items[index].tx_update(tx, fp, gen, char_dist);
    }
    tx.commit();
}
...
```

Transactional Element Access Test

```
...

//- Acquire the members of the update group
//
for (size_t j = 0;  acquired  &&  j < refs_count;  ++j)
{
    index    = indices[j];
    acquired = (acquired && tx.acquire(items[index]));
}

//- Modify the members of the update group
//
if (acquired)
{
    for (size_t j = 0;  j < refs_count;  ++j)
    {
        index = indices[j];
        items[index].tx_update(tx, fp, gen, char_dist);
    }
    tx.commit();
}
...
```

Transactional Element Access Test

```
...

//- Acquire the members of the update group
//
for (size_t j = 0;  acquired  &&  j < refs_count;  ++j)
{
    index    = indices[j];
    acquired = (acquired && tx.acquire(items[index]));
}

//- Modify the members of the update group
//
if (acquired)
{
    for (size_t j = 0;  j < refs_count;  ++j)
    {
        index = indices[j];
        items[index].tx_update(tx, fp, gen, char_dist);
    }
    tx.commit();
}
...
```

Transactional Element Access Test

```
...


//- Modify the members of the update group
//
if (acquired)
{
    for (size_t j = 0;  j < refs_count;  ++j)
    {
        index = indices[j];
        items[index].tx_update(tx, fp, gen, char_dist);
    }
    tx.commit();
}
else
{
    tx.rollback();
}
...
```

Transactional Element Access Test

```
...


//- Modify the members of the update group
//
if (acquired)
{
    for (size_t j = 0;  j < refs_count;  ++j)
    {
        index = indices[j];
        items[index].tx_update(tx, fp, gen, char_dist);
    }
    tx.commit();
}
else
{
    tx.rollback();
}
...
```

Transactional Element Access Test

```
        ...

        //- Modify the members of the update group
        //
        if (acquired)
        {
            for (size_t j = 0;  j < refs_count;  ++j)
            {
                index = indices[j];
                items[index].tx_update(tx, fp, gen, char_dist);
            }
            tx.commit();
        }
        else
        {
            tx.rollback();
        }
    }

    sw.stop();
    fprintf(fp, "TX %zd took %d msec\n", tx.id(), sw.milliseconds_elapsed<int>());
}
```

Main Test Driver

Test Driver – test_tx()

```
void
test_tx(FILE* fp, size_t item_count, size_t thread_count,
        size_t tx_count, size_t refs_count, size_t mode)
{
    using future_list = std::vector<std::future<void>>;

    stopwatch   sw;
    future_list fv;

    //- Mode 0 is a single-threaded run, in order to gather a baseline performance number.
    //
    if (mode == 0)
    {
        st_access_test(items, fp, tx_count, refs_count);
    }

    ...
```

Test Driver – `test_tx()`

```
void
test_tx(FILE* fp, size_t item_count, size_t thread_count,
        size_t tx_count, size_t refs_count, size_t mode)
{
    using future_list = std::vector<std::future<void>>;

    stopwatch   sw;
    future_list fv;

    //- Mode 0 is a single-threaded run, in order to gather a baseline performance number.
    //
    if (mode == 0)
    {
        st_access_test(items, fp, tx_count, refs_count);
    }

    ...
```

Test Driver – test_tx()

```
void
test_tx(FILE* fp, size_t item_count, size_t thread_count,
        size_t tx_count, size_t refs_count, size_t mode)
{
    using future_list = std::vector<std::future<void>>;

    stopwatch   sw;
    future_list fv;

    //- Mode 0 is a single-threaded run, in order to gather a baseline performance number.
    //
    if (mode == 0)
    {
        st_access_test(items, fp, tx_count, refs_count);
    }

    ...
```

Test Driver – test_tx()

```
...

//- Mode 1 is a multi-threaded, transaction-based.  This tests the algorithm.
//
else if (mode == 1)
{
    for (size_t i = 0;  i < thread_count;  ++i)
    {
        fv.push_back(std::async(std::launch::async,
                                std::bind(&tx_access_test, std::ref(items), fp,
                                          tx_count, refs_count)));
    }

    for (size_t i = 0;  i < thread_count;  ++i)
    {
        fv[i].wait();
    }
}

...
```

Test Driver – test_tx()

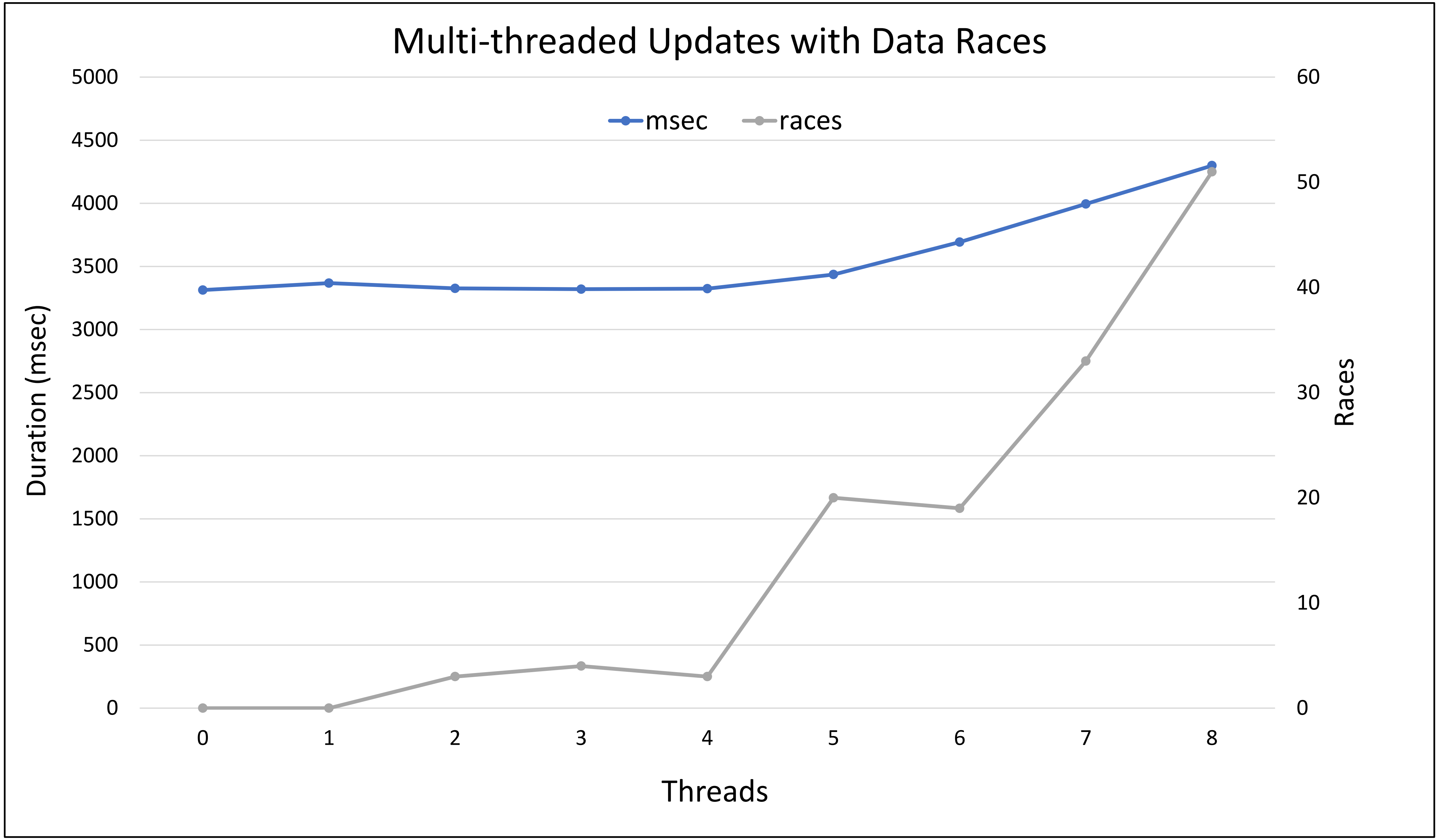
```
...

//- Mode 2 is a multi-threaded, but with a single mutex guarding all updates.
//
else if (mode == 2)
{
    for (size_t i = 0;  i < thread_count;  ++i)
    {
        fv.push_back(std::async(std::launch::async,
                                std::bind(&mx_access_test, std::ref(items), fp,
                                          tx_count, refs_count)));
    }

    for (size_t i = 0;  i < thread_count;  ++i)
    {
        fv[i].wait();
    }
 }

...
```

Test Driver – test_tx()

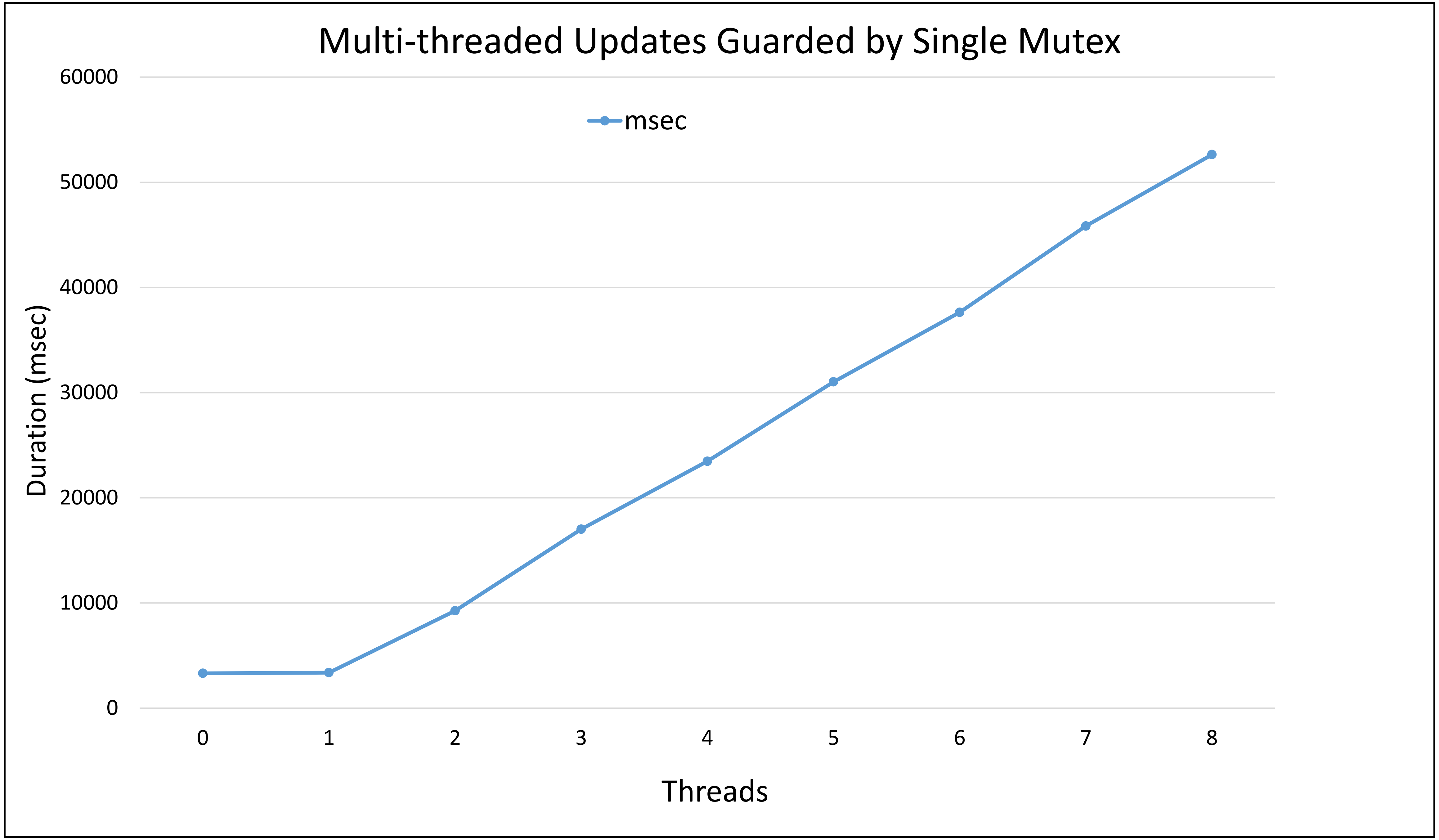
```
    ...

    //- Mode 3 is a multi-threaded, but with no protection.  Its purpose is to demonstrate
    //  that data races can occur in the absence of concurrency control.
    //
    else if (mode == 3)
    {
        for (size_t i = 0;  i < thread_count;  ++i)
        {
            fv.push_back(std::async(std::launch::async,
                                    std::bind(&st_access_test, std::ref(items), fp,
                                              tx_count, refs_count)));
        }

        for (size_t i = 0;  i < thread_count;  ++i)
        {
            fv[i].wait();
        }
    }
}
```

Test Results

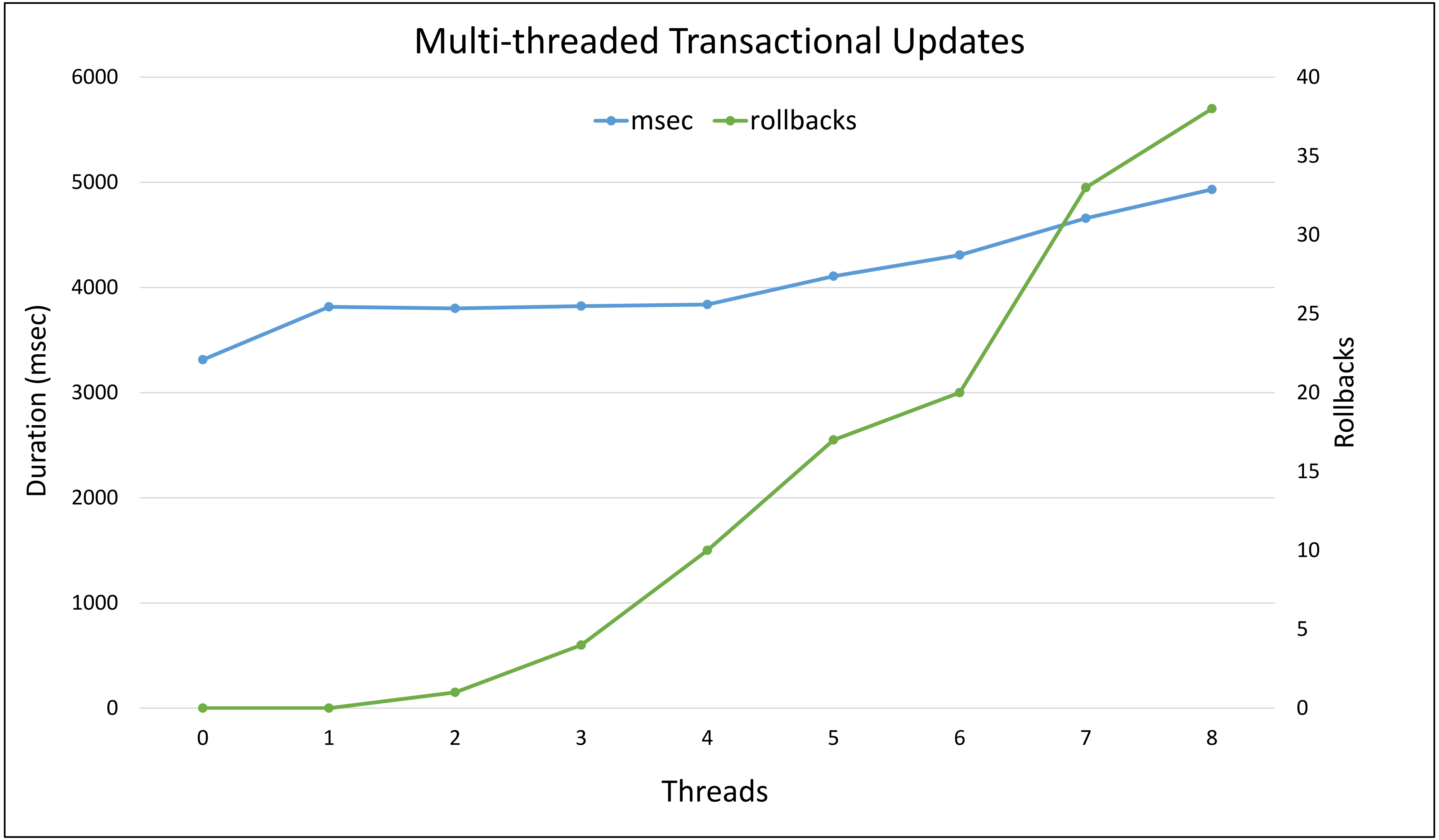
Results – 10M items / 1M transactions / 20 refs / 8 threads



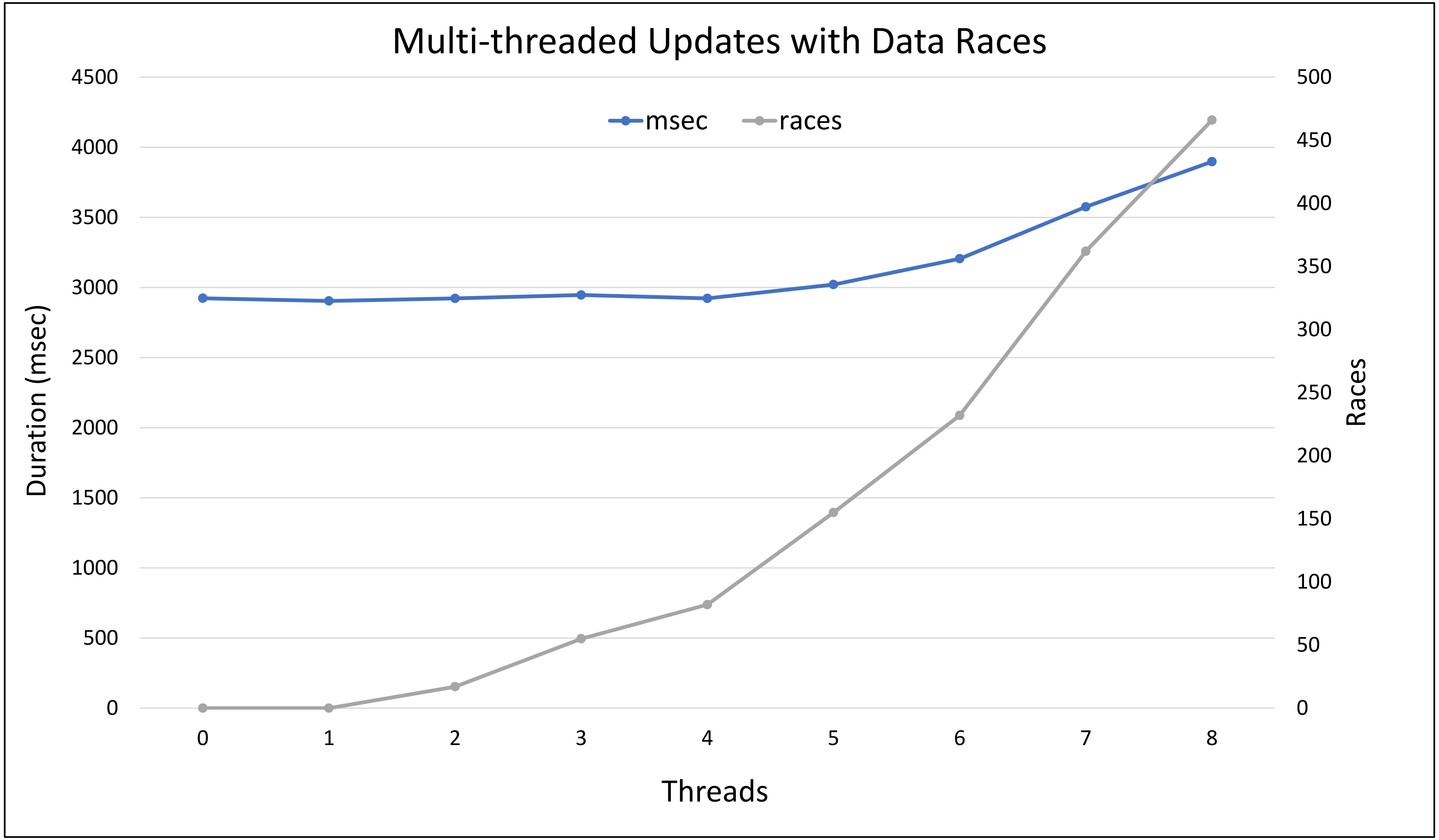
Results – 10M items / 1M transactions / 20 refs / 8 threads



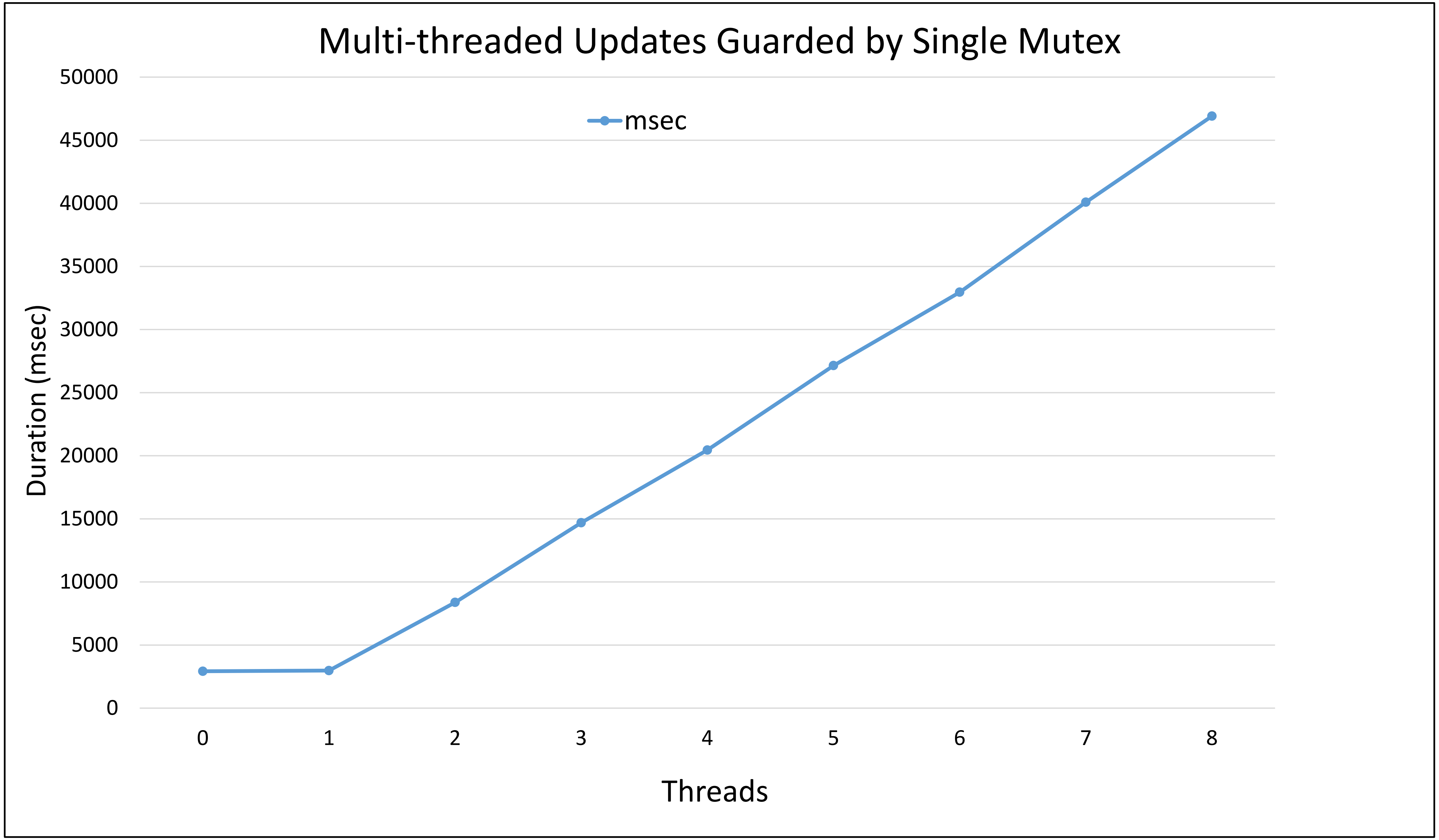
Results – 10M items / 1M transactions / 20 refs / 8 threads



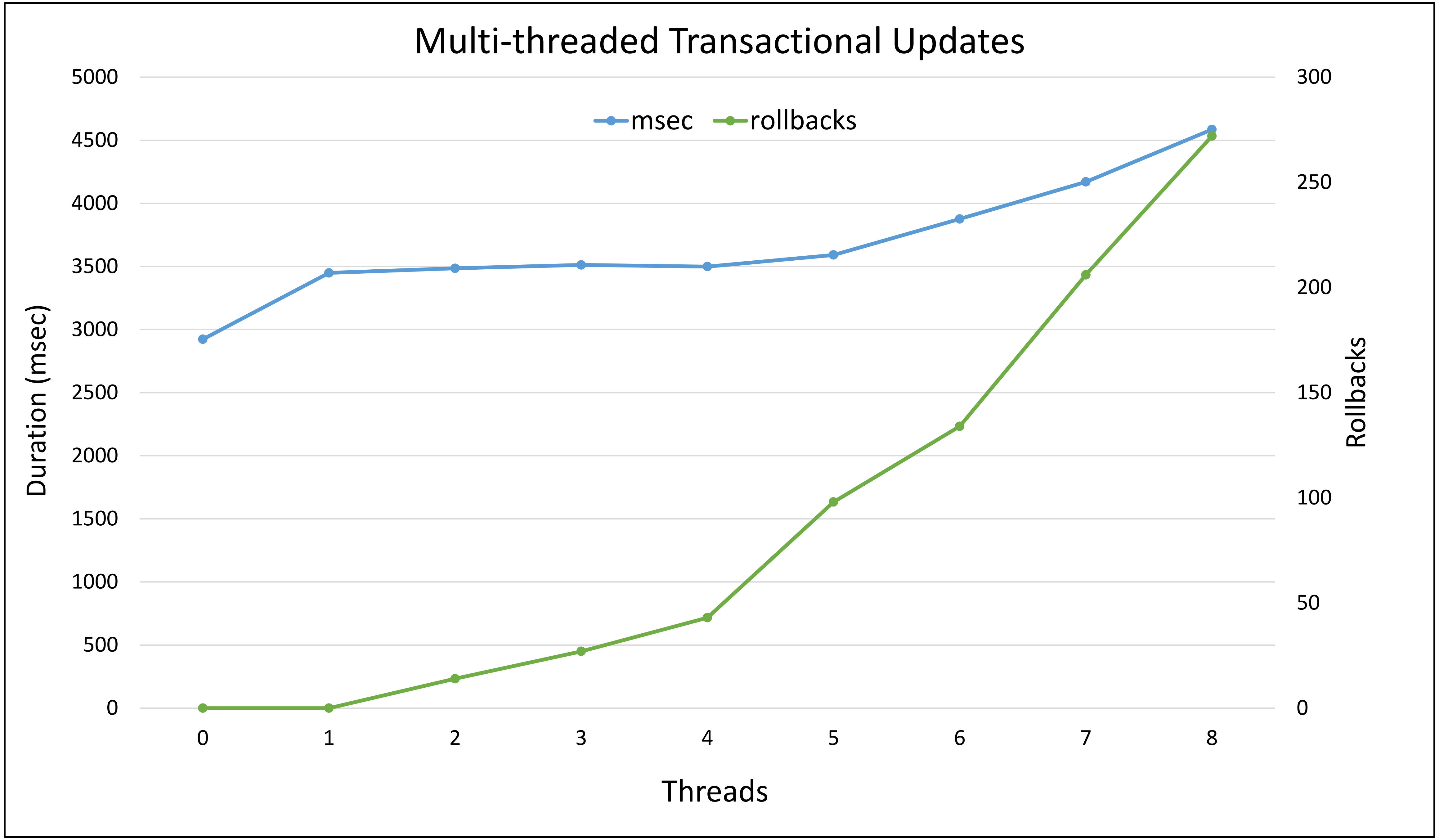
Results – 1M items / 1M transactions / 20 refs / 8 threads



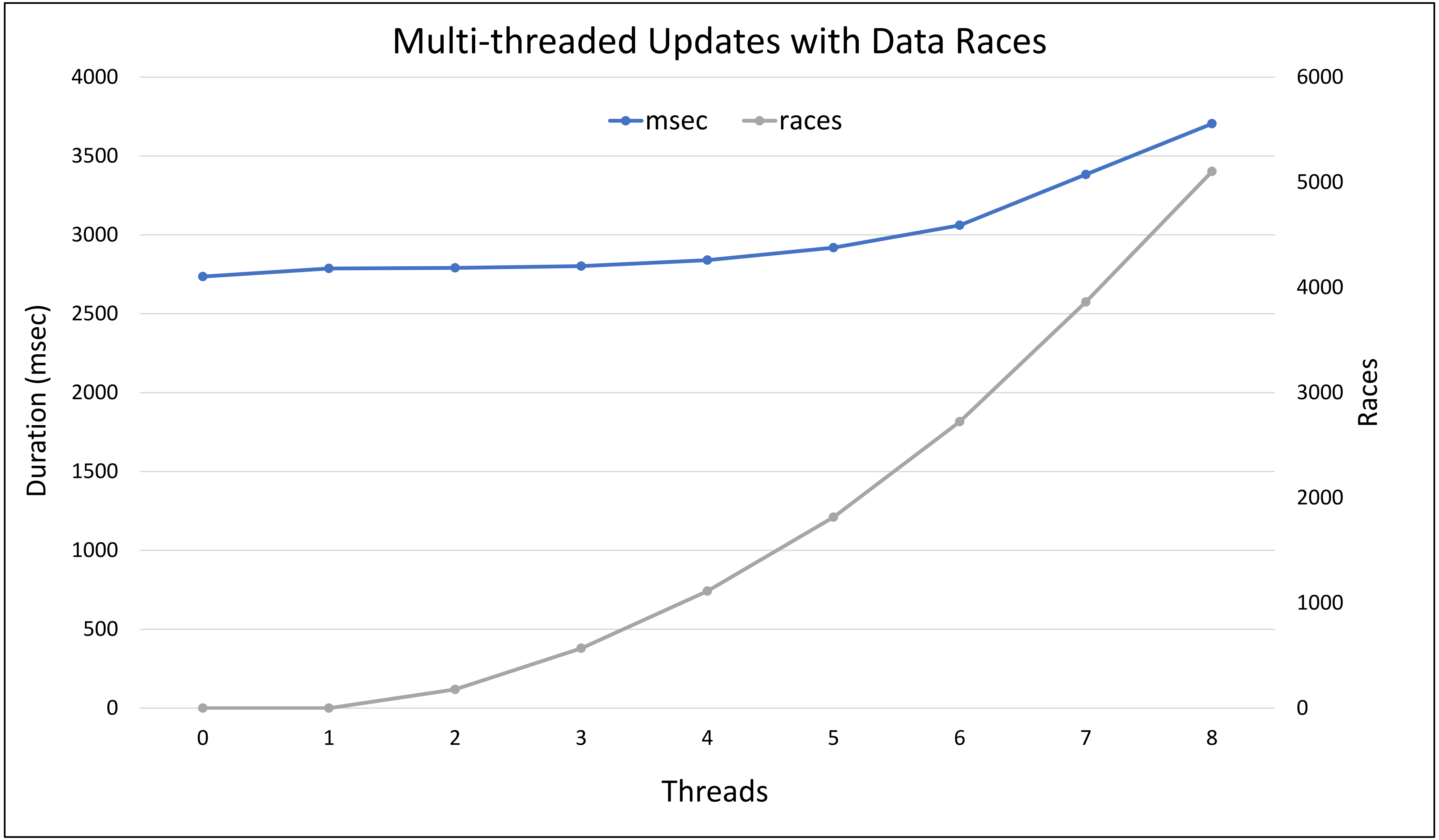
Results – 1M items / 1M transactions / 20 refs / 8 threads



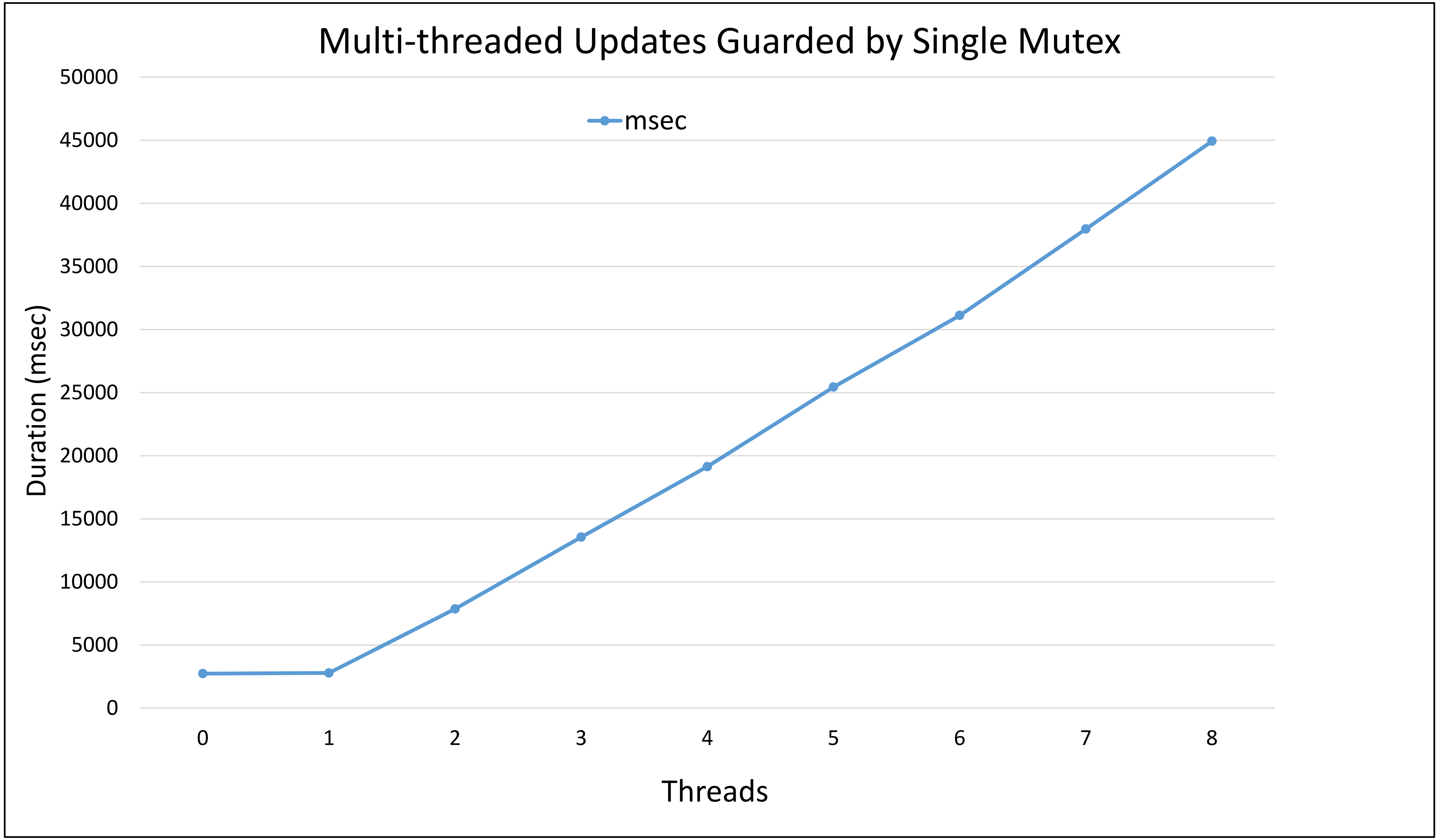
Results – 1M items / 1M transactions / 20 refs / 8 threads



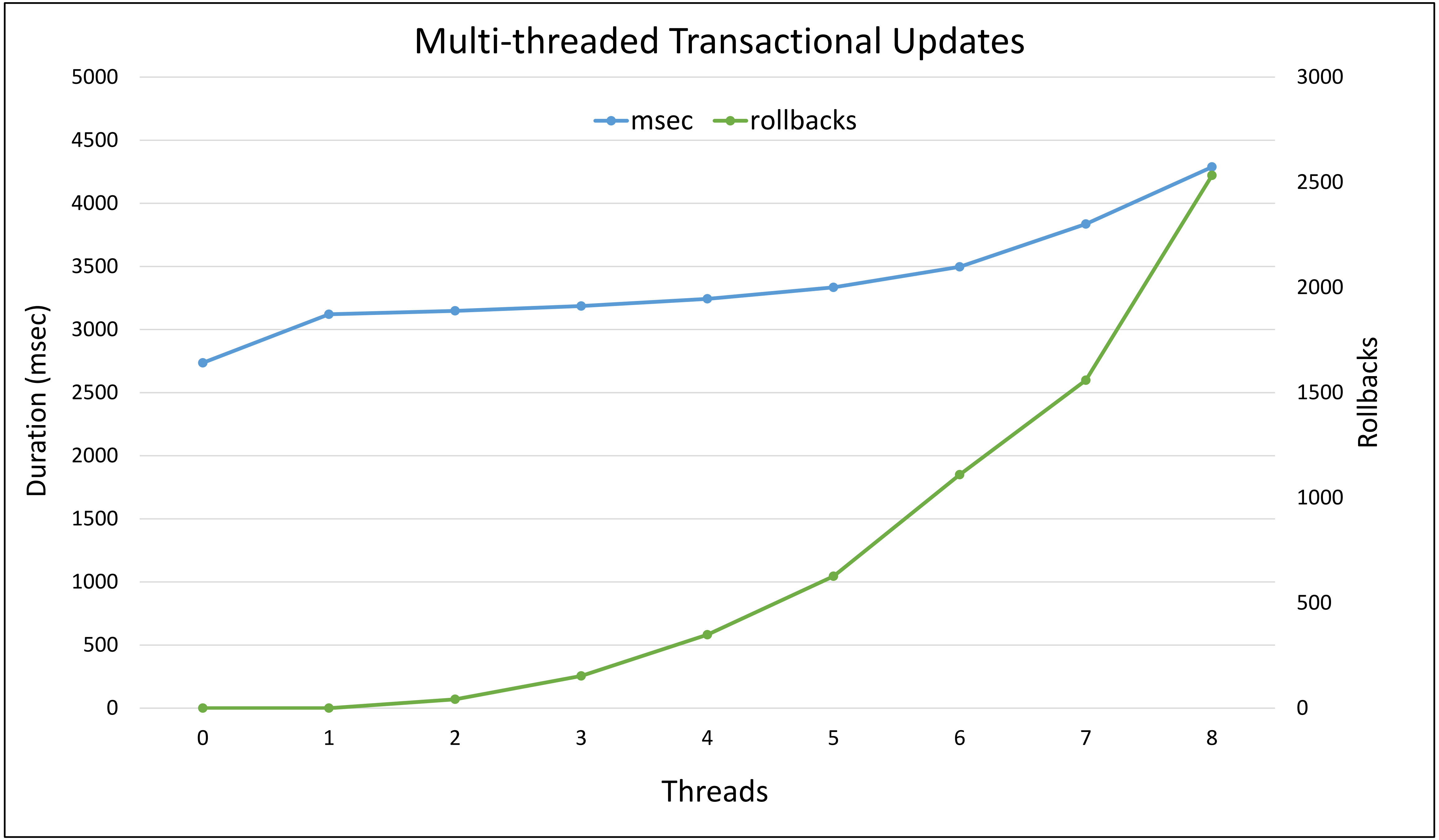
Results – 100K items / 1M transactions / 20 refs / 8 threads



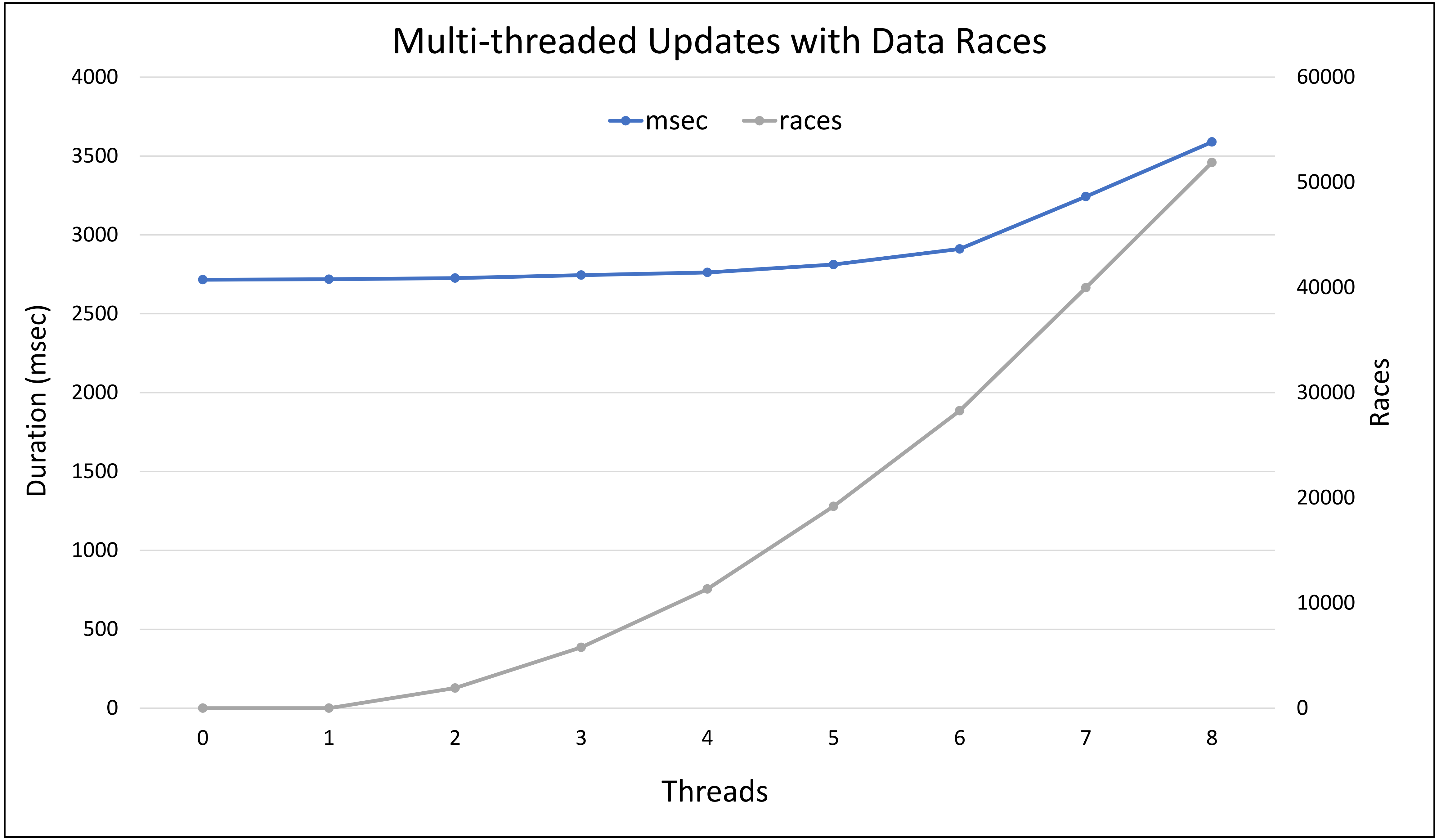
Results – 100K items / 1M transactions / 20 refs / 8 threads



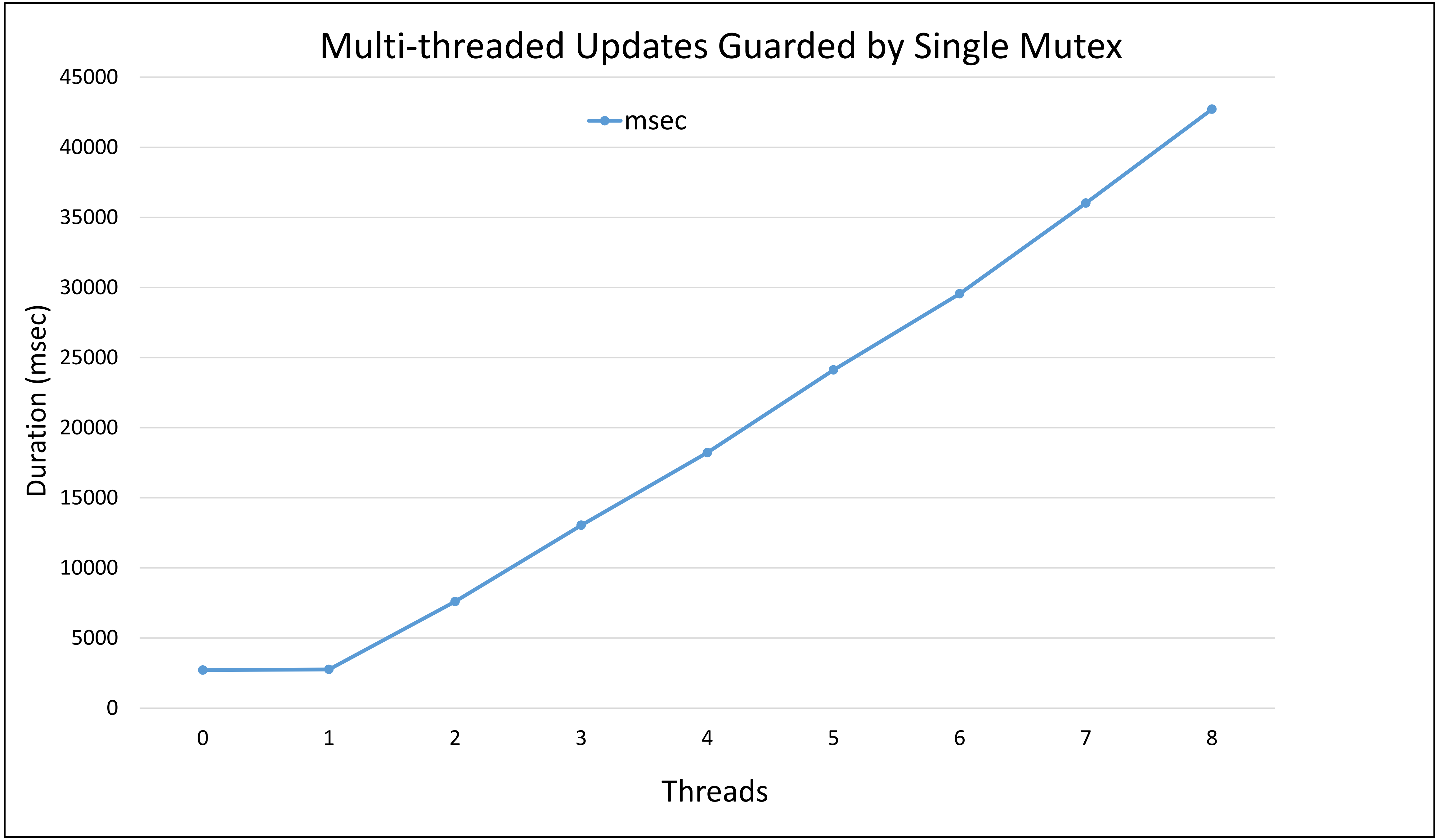
Results – 100K items / 1M transactions / 20 refs / 8 threads



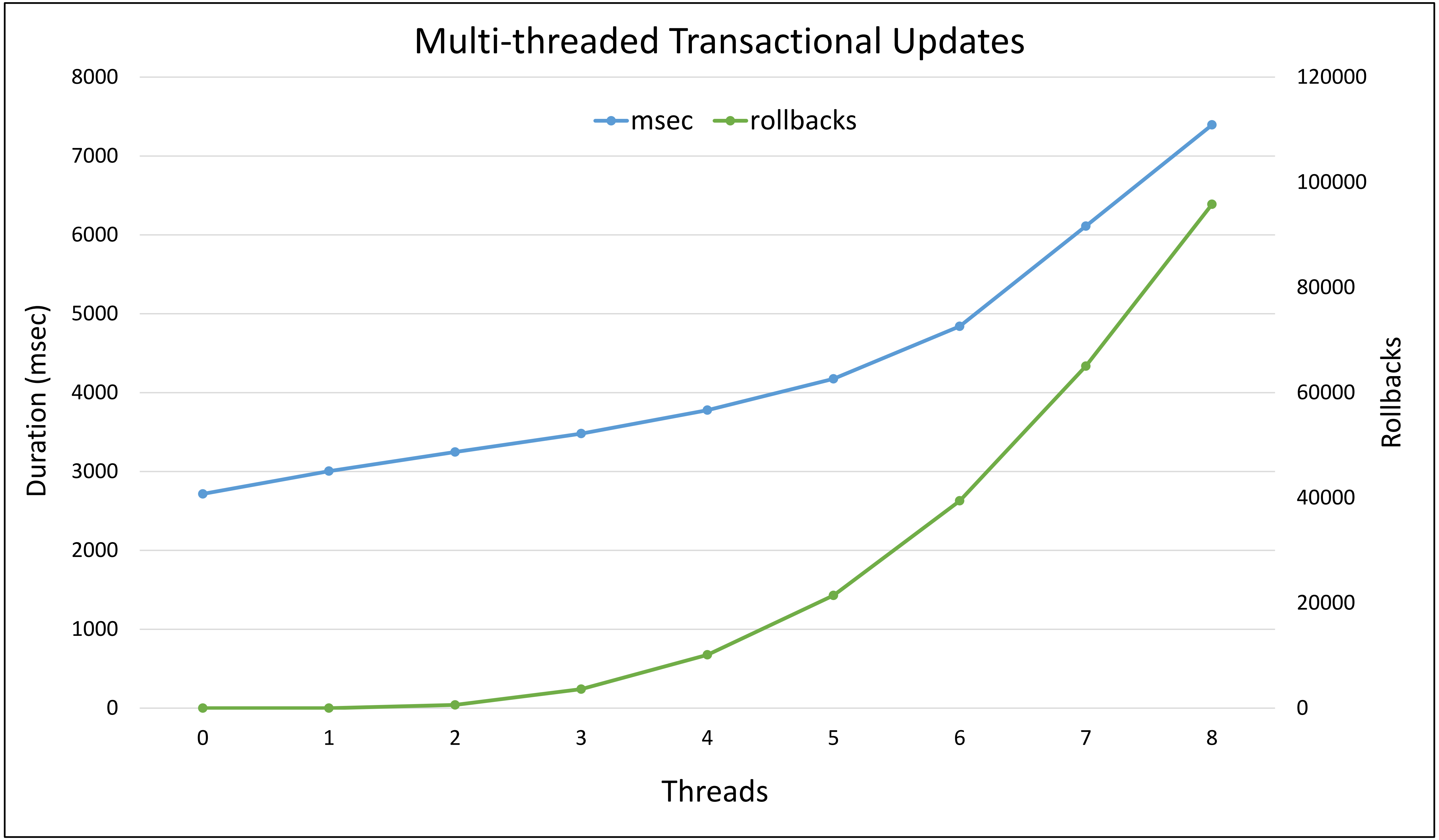
Results – 10K items / 1M transactions / 20 refs / 8 threads



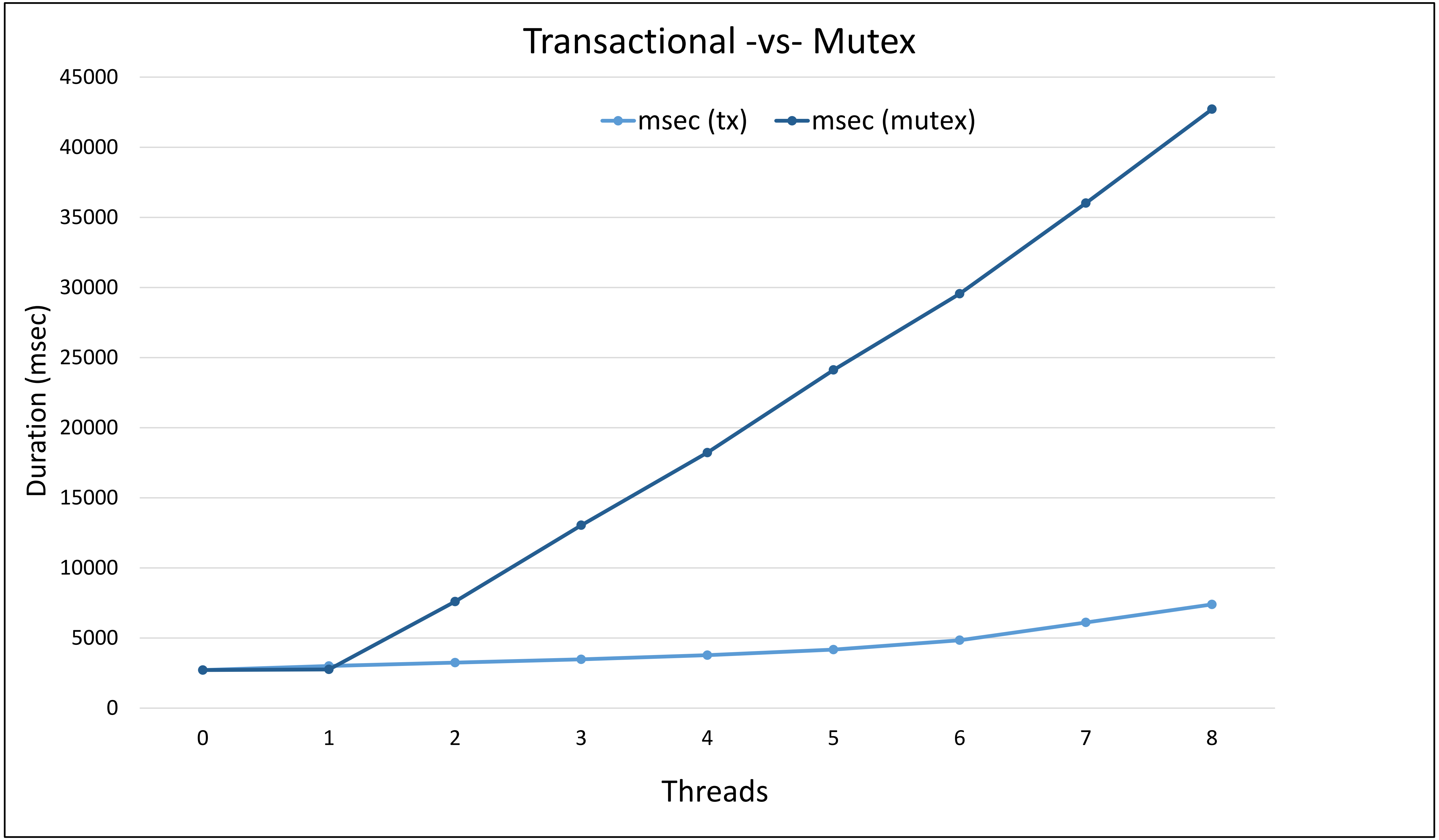
Results – 10K items / 1M transactions / 20 refs / 8 threads



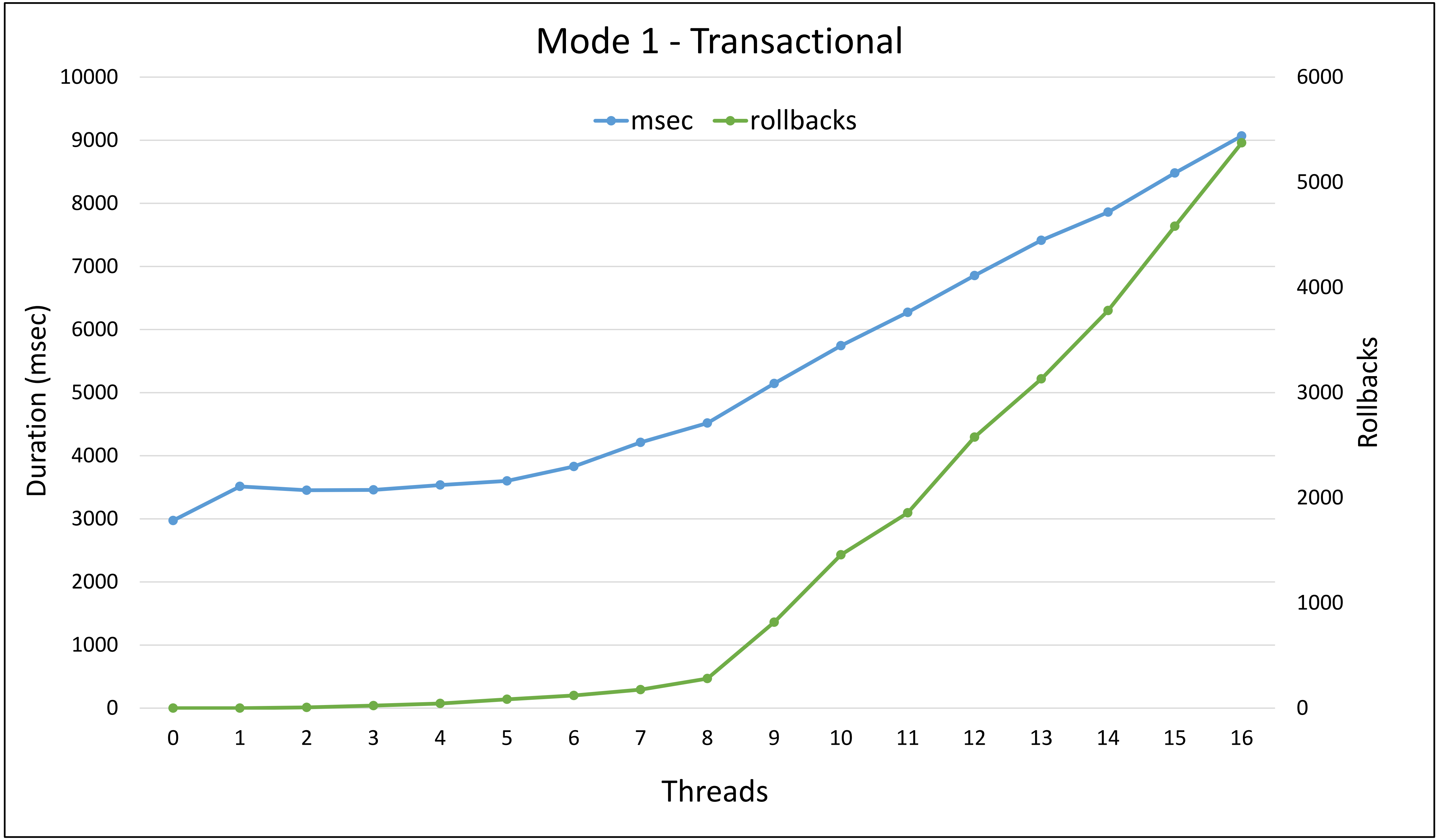
Results – 10K items / 1M transactions / 20 refs / 8 threads



Results – 10K items / 1M transactions / 20 refs / 8 threads



Results – 1M items / 1M transactions / 20 refs / 16 threads



Summary

Some Comments

- These tools operate upon containers and with elements, but don’t require changing the containers themselves

- There is an assumption that the container’s internal structure is unchanged while transactions are in progress
    - Consider the case of a `vector` resize
    - Consider the case of adding an element to a `map`
    - Could this be handled by a per-container shared mutex?

Some Comments

- So far, only `std::vector` has been used
  - The maximum number of elements is pre-allocated and resizes don’t occur

- To obtain a container that is resizable
  - Create a home-grown hash table using `std::vector`
  - Each element of the vector is a hash bucket
  - Hash buckets have member functions for adding, finding, erasing elements
  - Hash buckets are locked during transactions, and their contents updated
  - With a good hash function, lookup time is quite fast (but not as fast as indexing)

Some Comments

- Some threads could starve
    - Transactions might become stale
- Other container types may be amenable… ?
- Lots of room for more work

Questions?

Thank You for Attending!

Talk: `github.com/BobSteagall/CppNow2019`

Blog: `bobsteagall.com`