Efficient and Thread-Safe Hashmap

Tyler Hostler-Mathis, Ashley Voglewede, Chris Herrera, David Florez

Abstract

In this work we introduce the idea of an efficient and thread-safe hashmap. The standard hashmaps currently used in software development are designed to be sequential, which brought about the idea of making the standard hashmap faster through parallelization.

Our development of a concurrent hashmap was primarily focused around three properties: Simple, Extensible, and Correct. We created an API that was simple enough for developers to use, supporting idiomatic method calls from multiple threads. The API easily allows for creation of a generic hashmap permitting simultaneous insertions, gets, and deletes[Appendix A]. Our hashmap is extensible in that it leaves room for improvement on its underlying containers, allowing for developers to easily test out other data structures. Our hashmap is also correct, being that it is fully linearizable with each method call having a strict linearization point.

To further reduce complexity, the hashmap utilizes separate chaining for item insertions of the same hashcode. This allows for all items of the same hashcode to be in a single container within the hashmap, thus only the container itself needs to be thread safe. With our implementation of a thread-safe hash map, we are seeing runtimes that are significantly faster than the STL hashmap, proving that parallelization can indeed be a faster alternative to the standard implementation.

Index Terms

IEEE, IEEEtran, LaTeX, paper, Parallel processing, Hashmap, Multithreading

CONTENTS

I. Introduction

THE hashmap is one of the most frequently used data structures that is available in most libraries. It balances quick writes with quick lookups, and allows the user to solve many problems that would be difficult without it. Although the writes and reads are quick, they do have some problems when lots of data is being written. Primarily, if there are many collisions, they must spend some time finding an open space for the data before returning control to the main thread.

Our proposal is to implement a parallelized hashmap that allows multiple threads to write and read at the same time. This directly decreases runtime, as other non-colliding threads can continue their work while collisions are being resolved.

II. Prior Research

ALTHOUGH multithreading is a relatively new field, there has already been significant work done in this area. Below we have summarized four different influential and relevant research papers on the parallelization of hashmaps.

A. "A Wait-Free Hash Map" - Damian Dechev, Steven Feldman, Pierre Laborde

In this paper, researchers at the University of Central Florida explored a wait-free hashmap with atomic operations that allow the hashmap to be utilized by data-intensive applications. A hash map is classified as wait-free if it guarantees that each process makes progress in a finite number of steps. This wait-free design avoids deadlock and performance bottlenecks that accompany traditional blocking implementations, such as mutual exclusion locks. The design also supports concurrent expansion of the hash map, avoiding the overhead that comes with global resizes. The hashmap structure is a multi-level array similar to a tree, with each position being either a single node (dataNode or markedDataNode) or an array of nodes (arrayNodes). Their experimental results proved their design to be 7 times faster than the traditional blocking design and 15 times faster than the best available alternative non-blocking designs. [1]

B. "Lock-free parallel dynamic programming" - Alex Stivala, Peter Stuckey, Maria Garcia de le Banda, Manuel Hermenegildo, Anthony Wirth

This research paper demonstrates a real world and useful application of a thread-safe hash-map. Here dynamic programming problems are solved using a lock-free shared hash table that randomizes the order in which the sub-problems are computed. The research demonstrates application in well known knapsack and shortest path problems. [2]

C. "Java Implementation of a Lock-Free HashMap" - Ali Mizan, Brandon McMillan, Vaibhav Dedhia and Ebin Scaria

In this paper, researchers from the University of Central Florida build upon the prior works of "A Wait-Free Hash Map" listed in section A. They use Feldman's concurrent hashmap as a basis for their java implementation, and also account for memory management unlike Feldman's design. They also used logical bit flags to solve the ABA problem. However, their implementation of a wait-free hashmap with 3 operations did not outperform java's ConcurrentHashMap with 64 different operations. [3]

D. "High Performance Dynamic Lock-Free Hash Tables and List-Based Sets" - Maged M. Michael

In this paper from the IBM Thomas J. Watson Research Center, they present the first CAS-based lock-free list-based set algorithm as their underlying algorithm for lock-free hashtables. This algorithm is dynamic, linearizable, and space-efficient, with their experimental results showing that it outperforms the lock-based implementations significantly. [4]

III. Implementation

THERE are a number of languages with extensive support for multithreading, but with our focus on efficiency C++ was a clear choice. We will begin by discussing the fundamental design decisions that were made before any code was written, decisions that defined the progress of the rest of our project.

A. Probing Vs. Separate Chaining

The first fork in the road of development came in the form of a critical design decision: advanced probing or separate chaining? Although advanced probing would likely prove to be marginally more efficient, we decided to implement separate chaining due to its high extensibility and plasticity. With a separate chaining approach, we have full freedom to implement, test, and benchmark many different underlying container implementations without having to modify critical Hashmap code. As detailed in listing 1, the user can even supply their desired underlying container at compile time via the third template argument of our hashmap.

```
tshm::Hashmap<std::string, int, ll::AddOnlyLockFreeLL> fast(1e4);
tshm::Hashmap<std::string, int, ll::LockableLL> supports_removal(1e4);
```

Listing 1. Polymorphic Containers Example

B. Fixed Vs. Dynamic Capacity

The efficiency of a hashmap is directly related to its capacity, but of course there is a tradeoff as larger capacities take a longer time to allocate and deallocate. Although the perfect solution to this problem would implement dynamic sizing in order to address this issue, we decided to implement fixed capacity in order to reduce scope. Once all core features are completed, we plan to move forwards with dynamic sizing. For more information refer to Appendix B.

C. Underlying Containers

Since we took the approach of bucketted hashmaps with separate chaining to resolve collisions, we first needed to design and implement the underlying container in a thread-safe manner. For ease of implementation, we began by paralellizing a linked-list.

1) Hand-Over-Hand Locking Linked List: Our first implementation of a thread-safe linked-list was rather naive, but it got the job done. When traversing the linked-list, we make sure to acquire a lock on the node that we are attempting to move to before unlocking our current node, also known as hand-over-hand locking. This ensures that all operations are operating on the nodes that they are expecting.

Important code chunks are included in listings 2, 3, and 4. Within these chunks of code, we find linearization points at lines 12 and 30 for addition, and 20 for removal. Although very safe, we expect this container to be the slowest, particularly when thread counts are high.

Theorem III.1 *All hand-over-hand locking linked list operations have defined linearization points.*

Corollary III.1.1 *The hand-over-hand locking linked list is linearizable.*

2) Add-Only Lock-Free Linked List: After locking down predictable behavior with the hand-over-hand locking linked list we moved on to support lock-free containers, starting with an add-only lock-free linked list. Removal is in the works, but will incur a runtime overhead over this implementation. To achieve thread-safe functionality, we iterate over the linked list optimistically and utilize a compare-and-set to apply modifications. If the compare-and-set fails, we can simply search again from the beginning of the list. These failures should be fairly rare and will not significantly affect the runtime. As before, important code is documented in listing 5 and 6. We find our linearization points at lines 12 and 27 while adding.

Theorem III.2 *All add-only lock-free linked list operations have defined linearization points.*

Corollary III.2.1 *The add-only lock-free linked list is linearizable.*

D. Thread-Safe Hashmap

Now that we have covered the underlying containers, lets take a look at the actual hashmap that utilizes them. Fortunately, the containers were most of the heavy lifting, and the resulting data structure was fairly simple and intuitive.

1) Hashmap: Since our underlying containers are strictly thread-safe, we can simply tell them to insert or fetch items after hashing. This results in very simple hashmap code which is detailed in listing 7. We simply acquire the hashed index, create a new entry, and insert it into the list. The underlying container takes care of all thread-safe ensurances. Our linearization points are defined by the underlying container.

Corollary III.2.2 *The hashmap is linearizable.*

2) Hashset: The hashset is a simple modification of the hashmap, inserting elements by value rather than key and value. Again, the important code is provided in listing 8.

Corollary III.2.3 *The hashset is linearizable.*

```
// Add new element to the linked list
void add(const T &val) {
  // Maintain lock on cur node
  LockableNode *mover = head;
  mover->lock();

  // Traverse the list
  bool isHead = true;
  while (true) {
    // If we find it, update
    if (!isHead && mover->val == val) {
      mover->val = val;
      mover->unlock();
      return;
    }

    // Lock our next node (if it exists)
    LockableNode *next = mover->getNextAndLock();
    // Stop if we've reached the end
    if (next == nullptr)
      break;

    // Unlock and move
    mover->unlock();
    mover = next;
    isHead = false;
  }

  // Insert at end
  mover->next = new LockableNode(val);
  mover->unlock();
  curSize++;
}
```

Listing 2. Hand-Over-Hand Locking Linked List Add

```
// Remove an element, return success
bool remove(T val) {
  // Maintain lock on current node
  LockableNode *mover = head;
  mover->lock();

  // Traverse, look for node to remove
  while (true) {
    // Get the next node
    LockableNode *next = mover->getNextAndLock();

    // Break if we're done
    if (next == nullptr) {
      mover->unlock();
      return false;
    }

    // Found it, remove
    if (next->val == val) {
      mover->next = next->next;
      mover->unlock();
      next->unlock();

      delete next;
      curSize--;

      return true;
    }

    // Unlock and move
    mover->unlock();
    mover = next;
  }

  mover->unlock();
  return false;
}
```

Listing 3. Hand-Over-Hand Locking Linked List Remove

```
// Return existence, store val in param
bool find(T &val) const {
  // Empty list
  if (head->next == nullptr)
    return false;

  // Maintain lock on current node
  LockableNode *mover = head->next;
  mover->lock();

  // Check for existence
  while (true) {
    if (mover->val == val) {
      val = mover->val;
      mover->unlock();
      return true;
    }

    // Get next and break if done
    LockableNode *next = mover->getNextAndLock();

    if (next == nullptr) {
      mover->unlock();
      return false;
    }

    // Move
    mover->unlock();
    mover = next;
  }
}
```

Listing 4. Hand-Over-Hand Locking Linked List Find

3) Managed Hashmap: In an effort to reduce the API's complexity, we attempted to create a hashmap that manages the callee's threads for them. We allow the user to specify a maximum worker thread count, and then the hashmap manages all multithreading from that point on. Unfortunately, this idea is better on paper than in implementation. The added overhead of spinning up a new thread for every operation is simply not worth it, and our experimental results make this clear. Regardless, we have provided the core components for documentation purposes in listing 9.

IV. Experimental Results

We will now provide experimental benchmark data for the data structures that we implemented as a component of our research. Some figures cited below reference a key, which represents the sample size used. The benchmark results are detailed in this section via graphs generated with Jupyter Notebooks. Runtime is measured in milliseconds.

A. Underlying Containers

1) Hand-Over-Hand Locking Linked List: As expected, the hand-over-hand locking linked list performed the worst of our containers. When many threads attempt to traverse the list, they could all be limited by a single thread attempting to remove

```
// Add new element to the list
void add(const T &val) {
  Node *toAdd = new Node(val);

  // Keep going till we find success
  while (true) {
    Node *pred = head, *curr = head->next;

    while (curr != nullptr) {
      // Found it, update
      if (curr->val == val) {
        curr->val = val;
        delete toAdd;
        return;
      }

      pred = curr;
      curr = curr->next;
    }

    // Connect new node
    toAdd->next = curr;

    // Add with CAS
    Node *expected = curr;
    Node *required = toAdd;
    if (pred->next.compare_exchange_weak(
      expected,
      required
    )) {
      curSize++;
      return;
    }
  }
}
```

Listing 5. Add-Only Lock-Free Linked List Add

```
bool find(T &val) const {
  Node *curr = head->next;

  while (curr != nullptr) {
    // Found it
    if (curr->val == val) {
      val = curr->val;
      return true;
    }

    curr = curr->next;
  }

  return false;
}
```

Listing 6. Add-Only Lock-Free Linked List Find

```
// Associate specified key with specified value
void put(const K &key, const V &val) {
    size_t index = getHashedIndex(key);
    hashmap[index].add(TypedEntry(key, val));
}

// Return the status of containment and value
std::pair<bool, V> get(const K &key) const {
  size_t index = getHashedIndex(key);

  TypedEntry entry(key);
  if (hashmap[index].find(entry))
    return {true, entry.val};

  return {false, V{}};
}
```

Listing 7. Hashmap

```
// Associate specified key with specified value
void insert(const T &item) {
  size_t index = getHashedIndex(item);
  hashset[index].add(item);
}

// Returns whether the item is in the Hashset
bool contains(T item) const {
  size_t index = getHashedIndex(item);
  return hashset[index].find(item);
}
```

Listing 8. Hashset

an early node. Results can be seen in figure 1. Note that the largest case we run is seeing a runtime improvement of over 300ms when comparing one thread to four.

Although hand-over-hand locking implementation is clearly thread-safe, there are certainly improvements that can be made. By forcing the threads to obtain locks before moving, we create significant bottlenecks for threads attempting to traverse the linked-list at the same time. Notably, we see a worst case runtime of almost four seconds at five threads.

2) Add-Only Lock-Free Linked List: The next attempt took the form of a lock-free linked list which optimistically traverses the linked-list, utilizing a compare-and-set to ensure that the add goes smoothly. Results can be seen in figure 2. This is the default underlying container for our hashmap, as after analyzing hashmap usage, we determined that the average user does not

```cpp
// Associate specified key with specified value
void put(const K &key, const V &val) {
  // Acquire the thread semaphore
  threadLock.acquire();

  // Spawn new thread
  std::thread t([&] (TypedEntry entry) {
    size_t index = getHashedIndex(entry.key);
    hashmap[index].add(entry);
    threadLock.release();
  }, TypedEntry(key, val));

  // Detach the thread
  t.detach();
}
```

```cpp
// Get the resulting value
std::pair<bool, V> get(const K &key) const {
  // Spin until all puts are done
  while (threadLock.active);

  // Get item
  size_t index = getHashedIndex(key);
  TypedEntry entry(key);
  if (hashmap[index].find(entry))
    return {true, entry.val};

  return {false, V{}};
}
```

Listing 9. Managed Hashmap

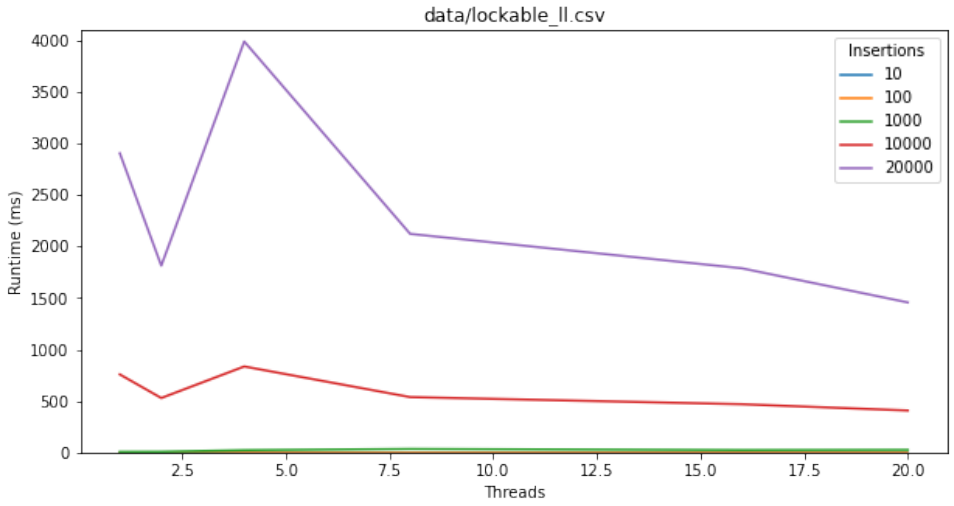

Fig. 1. Hand-Over-Hand Locking Linked List

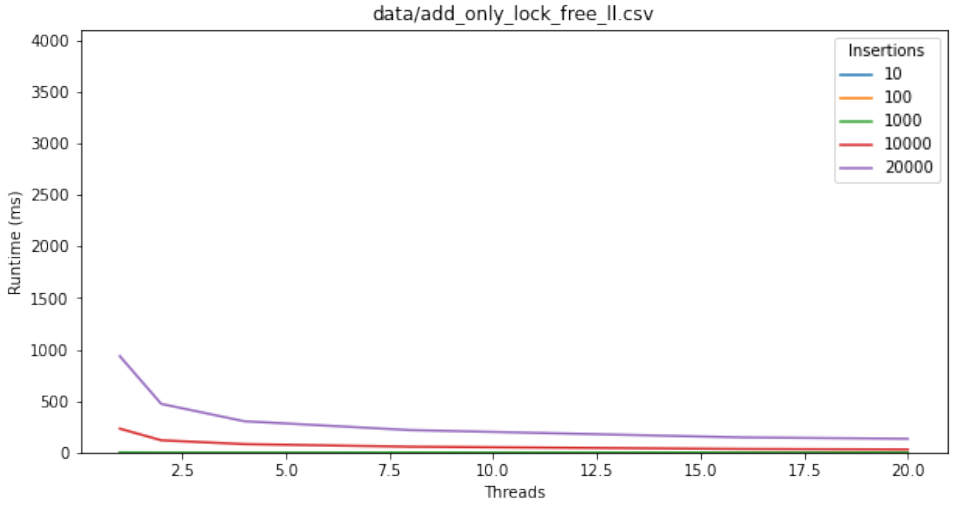

Fig. 2. Add Only Lock-Free Linked List

need to add to a linked-list.

This attempt gave a significant runtime improvement over the hand-over-hand locking linked list, with $20,000$ insertions taking roughly one second on one thread, and dropping logarithmically with relation to thread count afterwards.

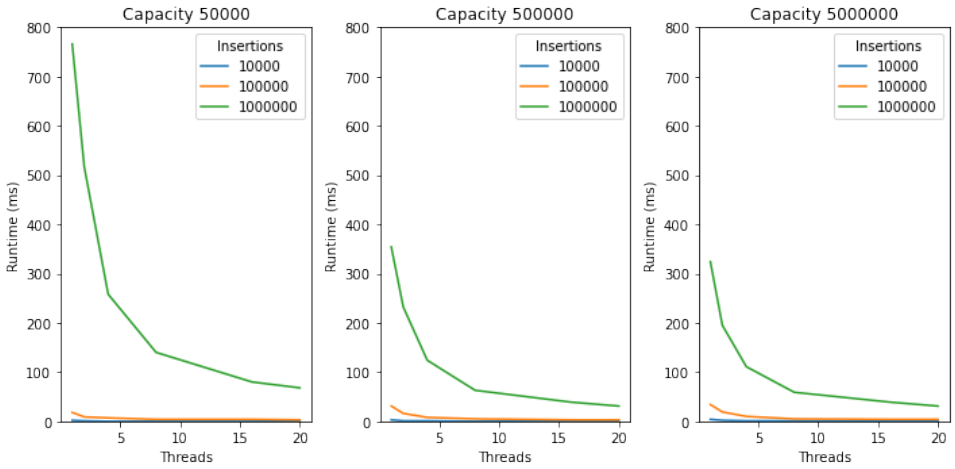

Fig. 3. Thread Safe Hashmap

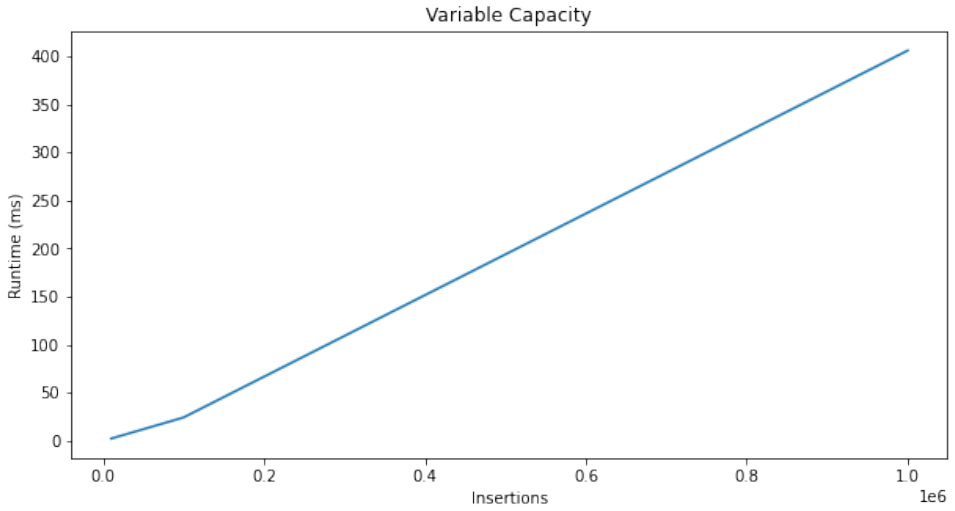

Fig. 4. STL Hashmap

B. Hashmaps

Now that our underlying container's efficiency has been covered, let's move into the hashmap itself. Our hashmap's runtime is defined by its capacity, as a smaller capacity will clearly create more collisions and thus incur a larger runtime, so we have separated the results into several graphs for ease of understanding.

1) Thread-Safe Hashmap: Our thread-safe hashmap's benchmarks blew us away, with incredible performance over expected. We see large improvements on runtime as threadcount increases. Benchmarks were run up to 10^6 insertions with 8 threads. Results are depicted in figure 3. As expected, we see larger capacity maps running significantly faster, although their allocation and deallocation scales linearly with their capacity.

C. STL Hashmap

We compared our hashmap with STL (C++ Standard Template Library), and were surprised to find that we improve over STL with a pseudo-linear relationship that decays into logarithmic after roughly 4 threads. The STL benchmark can be found in figure 4, immediately followed by a comparison of our thread-safe hashmap against it in 5.

D. Managed Hashmap

As expected, the managed hashmap's performance was abysmal. We see a linear speed increase when swapping from one thread to two; however, all efforts of thread increases afterwards proved fruitless. The managed hashmap simply has significant issues managing its threads in an efficient way. Benchmarks are located in figure 6.

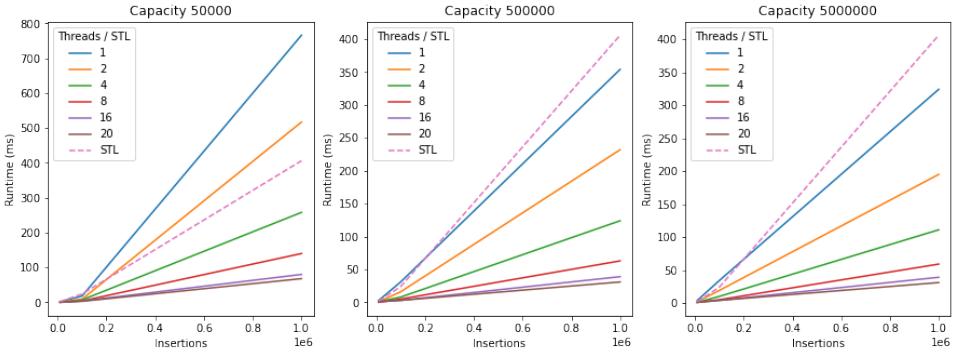

Fig. 5. TSHM vs STL

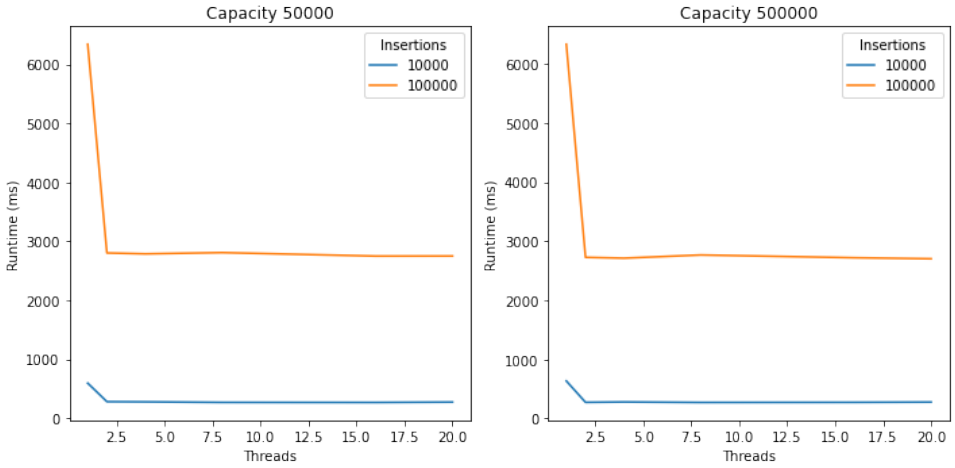

Fig. 6. Managed Hashmap

V. Conclusion

Our implementation of a parllelized hashmap was proven to have a dramatic speed and performance increase in comparison to the standard implementation. Our improvements in efficiency can be accredited to both the underlying container and the hashmap implementation itself.

By testing two different underlying containers, the hand-over-hand locking linked list and the add-only lock-free linked list, we were able to compare both methods and choose the most efficient implementation to further speed up our thread-safe hashmap. Upon testing, the add-only lock free linked list saw significant runtime improvements, had less overhead, and no bottlenecking. Our choice was clear to use the lock-free method as our underlying container.

As for our thread-safe hashmap, our benchmark tests showed a direct positive correlation between an increase in threads and a decrease in runtime. It had incredible results in comparison to the STL Hashmap, outperforming it in all reasonable[Appendix B] benchmarks.

Overall, our creation of a thread-safe hashmap was a success! With significant improvements for our parallelized data structure, we have further proven the power of parallel processing and the great potential it holds for the future of computing.

Appendix A
Removal

For future iterations of our parallelized hashmap, we plan on implementing removals and erasure. However, due to time constraints and to achieve maximum efficiency with better runtimes, we collectively decided to leave this feature out. Removals can significantly decrease the runtime due to its need for markable references, re-traversal upon compare and swap failure, and reference counting. We also determined that most use cases that use a hashmap do not require removals or erasure, and rely mainly on insertions and lookups. However, a version that supports these operations is currently in the works, and given more time and resources we are certain that we could accomplish this goal.

APPENDIX B
DYNAMIC SIZING

Since we currently use a fixed size for the hashmap, the user must know roughly how much space they will need to achieve their desired efficiency. This is fine for most cases, but users who overestimate their required space will see a significant overhead in allocating and deallocating the data structure.

In order to strike a balance between allocation runtime and hashed spread/efficiency, we will employ a powers-of-two resizing strategy. Given a capacity n, we will resize to $2n$ as soon as the hashmap fills $n/2$ elements. This will be accomplished by allocating a new underlying array, and re-hashing all previously hashed items.

APPENDIX C
SCALABILITY

We want to test on more threads to get a true picture of our scalability, to determine if there is a point where parallelization may actually hinder performance rather than improve. In our current benchmark tests, we used 4 threads for our linked list and 8 threads for our thread safe hashmap mainly due to resource restrictions. If given the time and hardware, we could provide more accurate results on scalability.

We would also like to experiment with more underlying containers to explore if there are other options that may be faster than our current implementation, like Red-Black trees. Our current underlying container is O(n) whereas we would want to explore data structures that are O(log n).

LISTINGS

LIST OF FIGURES

REFERENCES

[1] P. Laborde, S. Feldman, and D. Dechev, "A wait-free hash map," *International Journal of Parallel Programming*, vol. 45, 08 2015.
[2] A. Stivala, P. J. Stuckey, M. G. de la Banda, M. Hermenegildo, and A. Wirth, "Lock-free parallel dynamic programming," *Journal of Parallel and Distributed Computing*, vol. 70, no. 8, pp. 839–848, 2010.
[3] A. Mizan, B. McMillan, V. Dedhia, and E. Scaria, "Java implementation of a lock-free hashmap," *test*, 2001.
[4] M. M. Michael, "High performance dynamic lock-free hash tables and list-based sets," in *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, 2002, pp. 73–82.