Parallel optimization of the set data structure

Gustav Karlsson

Gustav Karlsson
Spring 2015
Bachelor's thesis, 15 hp
Supervisor: Jerry Eriksson
Examiner: Pedher Johansson
Bachelor's program in computer science, 180 hp
Abstract

The Set data structure is a powerful and popular programmer’s tool based on set theory. Bulk operations such as addAll provide a simple way of working with big collections of elements but can severely limit the performance of a multi-processor system when invoked on big sets if performed sequentially.

Parallel processing is a technique that can significantly shorten the execution time of otherwise sequentially executed tasks. In this thesis, a new type of set is proposed that uses multiple threads to employ parallel processing for bulk operations.

A new set is designed based on a non-blocking hash table with a high level of concurrency. The parallelization of the set makes use of parallel Java 8 streams to allow for quick iteration of the source collection elements, resulting in a speedup in processing of bulk operations.

Testing the parallel set shows a significant increase in execution speed on bulk operations when operating on more than 10000 elements.
Acknowledgements

I would like to thank my supervisor Jerry Eriksson at Umeå University for helping me keep this work on track. I would also like to thank Pedher Johansson at Umeå University for giving me some valuable assistance and feedback.

Finally, thank you Judith for your support, patience, and for putting up with my ramblings and rants.
Contents

1 Introduction 1

2 The set data structure 1

  2.1 Operations 2

  2.1.1 Inspecting operations 2

  2.1.2 Mutating operations 2

  2.1.3 Set creation operations 2

  2.1.4 Bulk operations 3

  2.2 Performance and common implementations 3

  2.2.1 Array 3

  2.2.2 Linked list 4

  2.2.3 Binary search tree 4

  2.2.4 Hash table 4

3 Concurrency and parallelism 5

  3.1 Turning concurrent into parallel 8

  3.2 Bulk operations 9

  3.3 The producer – consumer bottleneck 9

4 A case for parallel sets 9

5 Method 10

  5.1 Java 8 as a test platform 10

  5.2 Designing a parallel set 10

    5.2.1 Cliff Click’s non-blocking hash table 11

    5.2.2 Achieving non-blocking concurrency 11

    5.2.3 Turning the hash table into a set 12

    5.2.4 Parallelization 12

  5.3 Testing 13

    5.3.1 Parameters 13

    5.3.2 Test setup 14

6 Results 15

7 Conclusion 18

8 Discussion 18

  8.1 CPU scaling 18

  8.2 Number of elements scaling 18
1 Introduction

Data structures are foundational in computer science and come in many different forms such as lists, maps, trees, queues, stacks and sets. They are used for many different purposes such as assuring the flow of data, controlling relations between elements, or to simply act as containers. An interesting aspect of data structures is how they are represented in computer programs. The elements held in a data structure must be mapped to the computer’s main memory and as they grow in size, performance becomes an increasingly interesting topic of research. Execution time for lookups and modifications is one of the most important performance metrics of a data structure and can vary greatly depending on how it is implemented.

As time goes and development continues, computer systems get increasingly faster, but in recent years, processor speed is increasing slower and slower due to difficulties in the manufacturing process known as the “power wall” [1]. Instead, focus has shifted towards increasing the number of processors (or processor cores). This has forced programmers to make better use of multiple processors in computations – a difficult task that requires careful design to avoid issues such as race conditions, deadlocks and resource starvation.

One way of simplifying the use of multiple processors is to hide the parallelization behind a layer of abstraction such as a class in object oriented design. By encapsulating the complex parts, the solution can be reused in other projects to help spread the use of parallel computing.

Goal

The goal of this work is to design and implement a mutable set\(^1\) capable of executing a selection of its operations in parallel by encapsulating the parallelism in the implementation of the set. Sets are targeted both to limit the scope of the work and because they are interesting from a performance standpoint. The resulting set is tested to answer the following questions:

- How does a parallel set perform in comparison to a sequential set when running on a multi-processor system?
- How does the performance of a parallel set scale with the number of used processors?
- How does the performance of a parallel set scale with the size of the operation?

2 The set data structure

The set data structure is useful tool for programmers and originates from set theory – a branch of mathematical logic – in which it would more accurately be described as a finite set\(^2\).

\(^1\)A mutable set can be modified by inserting or removing elements.

\(^2\)A finite set has a limited number of elements, as opposed to an infinite set such as the set of all positive integers.
A set is a collection of elements where any given element may only occur once. The elements of a set are not ordered\(^3\) so the only two states an element can have in relation to a set is being a member of the set, or not being a member of the set.

### 2.1 Operations

Sets can be operated on in a number of ways depending on the type of set and what the intended purpose is. Some operations only inspect the state of the set, while some make modifications to it or create new sets from existing sets.

#### 2.1.1 Inspecting operations

Inspecting operations are used to inspect the state of the set. The most basic operations are:

- **contains** Checks whether the set contains a given element.
- **size** Returns the cardinality (number of contained elements) of the set.

#### 2.1.2 Mutating operations

Mutable sets must provide at least the following operations for the set to be “fully mutable”\(^4\):

- **add** Adds a given element to the set. The element is rejected if it is already a member of the set. Many implementations return a boolean value indicating if the insertion was accepted or rejected.
- **remove** Removes a given element from the set. Many implementations return a boolean value indicating if the element was removed or if it is not a member of the set.

#### 2.1.3 Set creation operations

In addition to operations performed on a set, some operations can be used to create new sets from multiple input sets.

- **union** Creates a set from elements that are members of any of the input sets.
- **intersection** Creates a set from elements that are members of all input sets.
- **difference** Creates a set from two existing sets \(A\) and \(B\), consisting of elements from \(A\) that are not members of \(B\).

---

\(^3\)Although some implementations order elements to provide a predictable iteration order or faster lookups.

\(^4\)**Fully mutable** in this case means that a set can always go from any possible state to any other possible state through mutable operations.
2.1.4 Bulk operations

For the convenience of the programmer, some implementations provide bulk versions of single element operations. Bulk operations do not provide any extra functionality but instead act as a replacement for the typical “for each item in A, do X on B” approach. They take as input an arbitrary number of elements (often packed in a data structure of their own) and perform an operation on the target set with each of the input elements.

- **addAll** Adds all elements of the input collection to the set. If the input is also a set, the target set will become the *union* of both sets.
- **removeAll** Removes all elements of the input collection from the set. If the input is also a set, the target set will become the *difference* between the target set and the input set.
- **containsAll** Checks whether the set contains all elements in the input collection. If the input is also a set, this will effectively determine if the input set is a *subset* of the target set.
- **retainAll** Removes all elements from the set that are not in the input collection. This is the inverse of removeAll.

2.2 Performance and common implementations

The restriction on sets to only contain a single instance of any element leads to an interesting observation: When adding an element to a set, the implementation must have some mechanism to rule out every other member as a duplicate of the element being inserted before accepting it. This can have a considerable impact on performance and a poorly designed implementation might need to traverse the full set on every insertion.

Another interesting property is the disorderliness of sets. Since elements are not ordered, there is no straightforward way of finding an element and the full set might have to be traversed in order to find the desired element.

Ultimately, this means that calls to add, remove and contains are $O(n)$ for any trivial implementation.

There are numerous ways to implement sets. Implementations come with different characteristics that make them suitable for different situations. Some of the most common approaches are:

2.2.1 Array

Elements are kept in a fixed size array where empty slots hold a null reference. When searching for an element, the array is traversed until a matching element is found. To add elements, the full array must be traversed to make sure the element is not already a member of the set. Removal sets the matching element’s index to null, creating a gap in the array unless it is the last element in the array.

If the array is full when an element is about to be inserted, a new and bigger array must be allocated and all elements must be transferred to the new array. Since transferring elements
to a new array is expensive, the new array is often created with capacity for additional ele-
ments, thus avoiding additional resizings for some time. The downside of a bigger capacity
is that more memory must be allocated. This balance between memory and computation is
important for the performance of the set.

Sets backed by arrays offer a simple solution suitable for learning purposes, but should not
be used in real world scenarios since lookups are $O(n)$.

2.2.2 Linked list

Every element is linked to another element, creating a gapless chain throughout the whole
set. As with using an array, a full traversal must be performed when inserting elements, but
in contrast, a linked list does not leave gaps when removing elements. Instead, the previous
element is just redirected to link to the next element, skipping the removed element [2].

Linked lists can also grow without the need to transfer all data when a threshold is reached.
No memory is “wasted” on pre-allocation but the solution still uses more memory since
every “slot” contains two references – one for the held element and one for the link to the
next element.

Performance is similar to sets based on arrays ($O(n)$) since finding an element requires a
brute-force search [2].

2.2.3 Binary search tree

If the elements of the set are sortable\textsuperscript{5}, the set can be implemented as a binary search tree.
A binary search tree uses a tree structure with an element at every node. Every node can
have up to two edges leading to sub-trees. One of the sub-trees contains “smaller” elements
and one contains “larger” elements. Lookups in a binary search tree are performed by
comparing an input element with the root node to see if it is the desired element. If it is not,
the algorithm continues to search until the element is found or the appropriate sub-tree does
not exist. Which sub-tree is chosen depends on if the input element is “smaller” or “larger”
than the examined node. Figure 1 illustrates a binary search tree.

Binary search trees are $O(\log n)$ on average [2], but in order to actually perform well, the
tree must be well balanced. A tree is fully balanced when the height of its sub-trees differ
at most by one, and gets “less balanced” the more that depth differs.

A poorly balanced tree is more expensive to traverse, but “balancing” a tree is an expensive
process as well since multiple modifications might be necessary for each relocated element.

2.2.4 Hash table

Hash tables consist of a number of “buckets” that can store multiple elements. On insertion,
the element is run through a hash function to determine what bucket it should be stored in.
When looking up an element, the hash code is used to determine in what bucket to look in.

The number of buckets in relation to the number of elements is crucial to a hash table’s

\textsuperscript{5}Any two elements can be compared to determine an order.
Figure 1: A binary search tree with integer elements

performance. Having too few buckets leads to collisions\(^6\) which harms performance, while too many buckets lead to unnecessary memory usage (similar to over-allocating capacity in arrays). The ratio of elements-to-buckets in a hash table is called load factor.

Another critical aspect when working with hash tables is the quality of the hash function. A bad hash function will fail to spread the elements evenly over the buckets, resulting in collisions even if the load factor is low.

If the number of buckets is reasonable and the hash function is well designed, lookups in hash tables are \(O(1)\) [2].

3 Concurrency and parallelism

Parallel computing is an efficient way to improve performance. In parallel computing, concurrency and parallelism are two commonly used terms that describe programs and their execution. However, they are often used interchangeably and can mean different things in different contexts, so for the sake of clarity, this is how they are defined for the purpose of this study:

Concurrency is the possibility of multiple threads calling an operation simultaneously. Two calls are simultaneous if one call is started before the other one has returned.

Non-concurrent Operations on the structure can only be called by a single thread at a time. This is often enforced by a lock, forcing other threads to wait until the currently executing thread returns. If no lock exists, attempts at concurrent access can break the data structure.

Concurrent Operations on the structure can be called by multiple threads at the same time. Multiple modifications of the structure can therefore occur simultaneously along with reads. The structure must be carefully designed to handle concurrent access in order

\(^6\)A collision occurs when an element is assigned to an already occupied bucket and must find a free slot in that bucket.
to gain the desired performance without breaking the structure.

Parallelism is an operation’s ability to split its work into multiple tasks that can run in parallel on separated threads.

Sequential  Executes called operations using a single thread (usually the thread calling the operation).

Parallel  Executes called operation using multiple threads. Multiple threads are either spawned or fetched from a pool. The calling thread either blocks or takes part in the execution, but returns only when the operation is fully completed.

Combining the possibilities of concurrency and parallelism results in four possible outcomes. These are described in the following designs:

Non-concurrent and sequential
The most straightforward approach. One thread can access the structure at a time and that same thread executes the operation.

Figure 2: A non-concurrent and sequential design with a single thread calling and executing the operation
Non-concurrent and parallel

One thread can access the data structure at a time but the structure splits the work and executes using multiple inner threads.

Concurrent and sequential

Multiple threads can access the structure simultaneously. Operations are executed sequentially by the calling thread.
**Concurrent and parallel**

Multiple threads can access the structure simultaneously and each one splits the work and executes using multiple inner threads.

![Diagram](image)

**Figure 5:** A concurrent and parallel design with multiple calling threads and multiple executing inner threads

### 3.1 Turning concurrent into parallel

An interesting observation can be made regarding these designs: Any concurrent data structure can be transformed into a parallel data structure by wrapping the structure in an implementation that uses multiple threads for execution. This can be seen in Figure 6. The concurrent structure essentially becomes the inner structure of another structure.

![Diagram](image)

**Figure 6:** Transforming a concurrent and sequential structure into a non-concurrent and parallel structure
3.2 Bulk operations

In addition to single element operations, some sets also contain a number of bulk methods that accept collections of other elements as an argument. These methods are interesting since they combine iteration over every source element with an operation on the target collection, making them a target for parallel execution.

Bulk operations also implicitly provide some additional information like the size of the source input collection. Knowing the size allows the operation to perform necessary preparations such as resizing an underlying structure before a large number of elements are inserted. Resizing as early as possible saves time since fewer elements need to be relocated.

3.3 The producer – consumer bottleneck

Performing a bulk operation in parallel can however come with a problem: The input collection must be iterated fast enough to provide the inner threads with elements.

This essentially boils down to the producer – consumer problem, where the producer (an iterator traversing the input collection) must be able to match the rate of the consumers operating on the target collection.

Iteration over a collection is typically done in a predetermined order – starting at one end and finishing in the other. This means that using multiple iterators does not help if they all start at the same point. Some mechanism for iterating over a collection in parallel might therefore be necessary to avoid the bottleneck.

4 A case for parallel sets

Since bulk operations are typically sequential and often computationally expensive, a case can be made for parallelizing. A parallel implementation can have a single calling thread but still execute in parallel – fully utilizing all processors available to the system. If computation heavy operations like the bulk operations described in 2.1.4 could be executed in parallel, the execution time could be significantly reduced.

Comparison with concurrent sets

As seen in Figure 5, parallel execution can also be achieved by having multiple threads calling single-element operations. While this is certainly a possible way to gain performance, a parallel design has a number of benefits:

Portability A concurrent set must be externally parallelized. But with a parallel set, the parallelism is built-in and can easily be distributed in the form of a library.

Bulk operations A parallel set will execute operations faster up to a factor of p where p is the number of processors available to the system. A concurrent set will only use a single processor per operation call.

Constant size calculation Most collections use a counter to keep count of the number of elements they contain. A concurrent collection however, must synchronize the
counter which creates a bottleneck that hurts concurrency. Concurrent implementations can solve this by calculating the size by counting all elements in the set, but it brings performance down to $O(n)$. A parallel set can be non-concurrent and utilize a counter, making size calculation $O(1)$.

**Pre-insertion memory allocation** By using bulk operations, any collection backed by a fixed size array can detect and perform a resizing before actually adding elements, thus minimizing the number of elements that must be transferred to the new array. Transferring elements is extra expensive for hash tables since hash functions must be run again on every element.

5 Method

To evaluate the performance of parallel sets, a parallel set is created and tested along with other set implementations to evaluate its performance.

5.1 Java 8 as a test platform

Java 8 [3] is used as the platform of evaluation. There are a number of reasons for this:

Java offers a selection of well defined interfaces for various data structures as part of the Java Collections Framework [4]. The framework is part of the Java Standard Library and contains interfaces, implementations and tools for dealing with collections of elements. One of the interfaces is the Set interface, which defines operations like add, remove, contains and size among others.

The Set interface also offers bulk operations described in 2.1.4 and therefore allows for portability.

**Solving the producer – consumer problem**

Java’s standard library up to Java 7 only allows for iterating through a generic collection sequentially using an iterator and can therefore suffer from the producer – consumer problem described in 3.3. Java 8 solves this by introducing streams and spliterators.

A stream is described in the Java 8 APIs as “A sequence of elements supporting sequential and parallel aggregate operations.” [5] Streams can be requested from any collection and can be processed in parallel if the collection supports it [6].

Streams use spliterators for parallel processing. A spliterator is an iterator that can split into smaller parts, each covering a distinct subset of the remaining elements. They can therefore be used by separate threads to maximize processor utilization.

5.2 Designing a parallel set

When designing for parallelism, a number of factors must be considered. Primarily, the underlying structure must be able to handle the concurrency. A hash table is chosen as the foundation for the parallel set for the following reasons:
Lookups and modifications are $O(1)$ on average.

- They have no requirement on elements being sortable.
- They can easily be made concurrent.

5.2.1 Cliff Click’s non-blocking hash table

The set is loosely based on Cliff Click’s non-blocking hash table. Click’s hash table is used since it offers very good scalability in multi-threaded environments [7, 8] and an otherwise simple design.

As seen in 2.2.4, hash tables traditionally store elements in “buckets”. Click’s hash table does not use buckets in the traditional sense. Instead, a single array holds all keys and values\(^7\), with keys on even indexes and values on odd indexes. Buckets still exist in a sense but start on different indexes (determined by the hash function) and reach all the way around the array, continuing at index 0 after the end of the array. When the array starts getting full, a new array is created, and the elements are copied to that one.

On insertion, a hash function is run on the element to determine an index in the array to start. This is roughly equivalent to finding a “bucket” in a typical hash table. If the slot is free, the element is inserted there. If the slot is occupied, the element currently occupying the slot is compared to the input element to see if it is the “same” element. Otherwise the next slot is tried, and the search continues until insertion is completed or rejected.

Removal works in a similar way, but instead of setting a slot to \texttt{null}, the slot is filled with a special \texttt{tombstone} object. The \texttt{tombstone} object is important because in contrast to \texttt{null} slots, the slot has previously been occupied. When traversing the array, getting to a \texttt{null} slot means that the key could not be found, while \texttt{tombstone} objects mean more slots must be tried.

5.2.2 Achieving non-blocking concurrency

In order to achieve non-blocking concurrency, Click views every slot as a state machine where every state can be handled by the algorithm. Slots contain a combination of the following tokens: Key (\(K\)), Value (\(V\)), Tombstone (\(T\)) and \texttt{null}. This leads to the following six possible states:

\[
\begin{align*}
\{\text{null, null}\} & \quad \text{Empty} \\
\{K, \text{null}\} & \quad \text{Partially inserted key/value} \\
\{K, V\} & \quad \text{Complete key/value pair} \\
\{K, T\} & \quad \text{Previously inserted, now deleted key} \\
\{\text{null, V}\} & \quad \text{Partially inserted key/value pair being read out-of-order} \\
\{\text{null, T}\} & \quad \text{Partially inserted key/value pair being read out-of-order}
\end{align*}
\]

\(^7\)This method is also called open addressing.
Since all these states can be handled, it does not matter if a slot is partially inserted, and therefore no synchronization is necessary.

For the actual assignment of keys and values, Click uses Compare-And-Swap (CAS) operations to ensure that no other thread just assigned that slot. This ensures that concurrent reads and writes can be performed at a high rate without locks.

5.2.3 Turning the hash table into a set

Taking inspiration from the non-blocking hash table, a set can be designed.

The underlying structure that holds all elements of the set is a special type of array called AtomicReferenceArray. This array is similar to a normal Java array but additionally allows atomic operations like CAS to be performed. Since keys are not used for sets, the number of possible states comes down to three:

{null} Empty
{V} Partially inserted key/value
{T} Fully functional key/value pair

Adding and removing elements is then only a matter of calculating an index using the hash function and walking through the structure until a desired slot is found. When he slot is found, an attempt is made to change the value to a new element or a tombstone with a CAS operation.

Resizing is performed in the beginning of any add or addAll operation if the resulting number of elements might exceed a certain limit. The limit is a fraction of the array size to avoid collisions and is based on a load factor property that can be set when creating the set.

5.2.4 Parallelization

The following operations are performed in parallel:

addAll Adds all elements of the input collection to this set.
removeAll Removes all elements of the input collection from this set.
containsAll Checks if this set contains all elements in the input collection.
retainAll Removes all elements from this set that are not in the input collection.
removeIf Removes all elements in this set that fulfill the input predicate.
toArray Returns an array containing all elements of this set.
hashCode Calculates a hash code from the elements in this set.

---

8 Compare-And-Swap is an atomic operation that compares the existing value with an expected value before assignment.
toString  Returns a string representation of this set.

Worth noting is that not only typical set operations are parallelized, but also some common Java methods like `hashCode` and `toString`.

All the parallelization is done using parallel streams. For operations with an input collection, the stream is created from the input collection, and a special concurrent version of a single element operation is invoked on the set.

Other operations such as `hashCode` are performed by creating a parallel stream from the set to collect the necessary individual hash codes and finally returning the sum of them.

5.3 Testing

To measure the performance of the parallel set, a custom built benchmarking suite is used. The suite allows test cases to be set up by adjusting a variety of parameters. Tests are then measured and their results are printed out.

A single test case consists of performing one operation on a set. The result of a test case is the time it takes to perform the operation.

5.3.1 Parameters

Tests are configured using a number of parameters that can be individually adjusted. The following parameters exist:

Number of threads

Determines how many inner threads are available during execution of an operation. Sequential operations will only ever use a single thread, while parallel operations can use multiple threads. The number of threads is used to measure how the implementation scales with the number of processors.

Operation

Determines what operation to execute. It is important to test both inspecting operations and mutable operations. Operations consist of two phases. A preparation phase used to set up the test environment, and an execution phase. When running a test, only the execution phase is measured.

The following three operations are included as test parameters as they are some of the most used bulk operations and also provide a good mix of reads and writes:

Implementation

The implementations that the operations were performed on. Both implementations are based on hash tables since they share the same performance complexity for common operations and require hash code calculation for elements. This should lead to less interference from unrelated performance factors in the test results. The following two implementations are tested:
<table>
<thead>
<tr>
<th>Operation</th>
<th>Preparation</th>
<th>Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>addAll</td>
<td>The set is initialized with a capacity of $n$</td>
<td>$n$ elements are added to the set</td>
</tr>
<tr>
<td>removeAll</td>
<td>The set is initialized with a capacity of $n$ and filled with $n$ elements</td>
<td>Every element is removed from the set</td>
</tr>
<tr>
<td>containsAll</td>
<td>The set is initialized with a capacity of $n$ and filled with $n$ elements</td>
<td>Every element is checked for its membership in the set</td>
</tr>
</tbody>
</table>

**ParallelConcurrentHashSet** A `ConcurrentHashSet` wrapped in an implementation that enables parallel execution for bulk operations. This is the same technique described in 3.1.

**ParallelHashSet** The implementation designed as part of this study.

**Number of elements**

The number of elements operated upon. Depending on the operation, it can be items in the source collection or the target set.

**5.3.2 Test setup**

All tests are performed on a system equipped with an Intel Core i7 2700K quad core processor and 16 GiB of RAM. The Java runtime environment is given an initial heap space of 8GiB and a maximum heap space of 12GiB to minimize the risk for garbage collection runs.

Test input data consists of randomly generated 26 strings of length 26 and is stored in an `ArrayList`. Each test is executed 10 times and the average execution time of those executions is recorded.
6 Results

Figure 7: Test results after invoking `addAll` with 1M elements on a set with variable number of threads. Results along the Y-axis are normalized to present speedup from a single thread with the given implementation.

Figure 8: Test results after invoking `removeAll` with 1M elements on a set with variable number of threads. Results along the Y-axis are normalized to present speedup from a single thread with the given implementation.
Figure 9: Test results after invoking `containsAll` with 1M elements on a set with variable number of threads. Results along the Y-axis are normalized to present speedup from a single thread with the given implementation.

Figure 10: Test results after invoking `addAll` with variable number of elements. Results along the Y-axis are normalized to present speedup going from 1 to 8 threads for the given implementation.
Figure 11: Test results after invoking `removeAll` with variable number of elements. Results along the Y-axis are normalized to present speedup going from 1 to 8 threads for the given implementation.

Figure 12: Test results after invoking `containsAll` with variable number of elements. Results along the Y-axis are normalized to present speedup going from 1 to 8 threads for the given implementation.
7 Conclusion

Looking at the results, the questions from Section 1 can now be answered.

How does a parallel set perform in comparison to a sequential set when running on a multi-processor system?

Parallelizing a data structure like the Set can provide a significant boost in performance on multiprocessor systems for large numbers of elements. Figures 10, 11 and 12 shows that the number of elements needs to be at least around 10000 before having any significant benefits.

How does the performance of a parallel set scale with the number of used processors?

Unfortunately, CPU scaling is far from linear. Figures 7, 8 and 9 all show diminishing returns in speedup. Ideally, the two implementation lines should lie close to the optimal line but instead dip rapidly when utilizing more than one thread.

How does the performance of a parallel set scale with the size of the operation?

Figures 10 and 11 shows that maximum speedup for write operations is achieved when using around 60000 elements or more. Read operations (Figure 12) scale up slightly faster and higher, reaching a speedup factor of 4.5 with ParallelHashMap as early as with 30000 elements. It also seems like using multiple threads when adding a relatively small number of elements can in some cases result in negative speedup.

8 Discussion

The parallel set shows some promise, but is suitable only in situations where data can be read quickly and in parallel, and when the number of elements is relatively high.

8.1 CPU scaling

Although performance was improved in these scenarios, CPU scaling was not as good as I had hoped. The reasons for this could be many, but I think the most likely reason is that the thread pool used by the parallel streams is too slow. It uses a work stealing algorithm that might be more suitable for bigger workloads than the ones in this task. Unfortunately this cannot be tweaked since it is built into the Java 8 Standard APIs.

Another possible explanation is the parallel read performance of the source. If data cannot be read quickly enough, the resulting bottleneck will harm performance. However, this seems unlikely since the source collection is based on a simple array which should not limit read performance.

8.2 Number of elements scaling

The results from figures 10, 11 and 12 are not surprising at all. As with all operations that requires some kind of preparation (work distribution to worker threads in this case),
some overhead should be expected. But as the size of the operation increases, the overhead becomes a smaller factor of the resulting execution time, and thus the scaling factor when using multiple threads can go up.

The result lines for the different operations all look similar, with Figure 12 standing out a bit by scaling up noticeably earlier and higher. I am not sure of the reason for this, but one possibility could be that the number of collisions for mutating operations increases with the element count.

The negative speedup seen with low element counts indicates that the work distribution costs more time than what is saved from using more processor threads. This is also not surprising but interesting since it can make choosing between sequential and parallel execution more difficult.

8.3 Future work

To pursue the topic of parallel data structures further, there are likely many interesting areas to explore. Some areas I could think of are:

- Other data structures such as lists, maps, queues and stacks can also be parallelized
- Other languages and libraries can provide other ways to implement the parallelization
- A custom built thread pool could allow for a more efficient work distribution that could decrease the overhead
- Testing elements with more expensive hash functions should increase overall execution time, making the overhead smaller and therefore improving the scaling
- A different underlying data structure (or a different implementation of hash tables) could perform better

8.4 Closing

Because of the diminishing returns in the results, I would only recommend these implementations when other CPU’s are known to have low utilization. Otherwise processing power might be “wasted” on the parallel set operations instead of being put to better use elsewhere.

Programming for multiprocessor systems can be difficult and time consuming. The convenience of using a class that encapsulates the parallel computation aspects while also implementing a commonly used interface makes it a useful tool and an interesting topic for further development.
References


