On A Generic Parallel Collection Framework

Aleksandar Prokopec, Phil Bagwell, Tiark Rompf, Martin Odersky

June 27, 2011

Abstract

As the number of cores increases in modern multiprocessors, it is becoming increasingly difficult to write general purpose applications that efficiently utilize this computing power. This remains an open research problem.

Most applications manipulate structured data. Modern languages and platforms provide collection frameworks with basic data structures like lists, hashtables and trees. These data structures have a range of predefined operations which include mapping, filtering or finding elements. Such bulk operations traverse the collection and process the elements sequentially. Their implementation relies on iterators, which are not applicable to parallel operations due to their sequential nature.

We present an approach to parallelizing collection operations in a generic way, used to factor out common parallel operations in collection libraries. Our framework is easy to use and straightforward to extend to new collections. We show how to implement concrete parallel collections such as parallel arrays and parallel hash maps, proposing an efficient solution to parallel hash map construction. Finally, we give benchmarks showing the performance of parallel collection operations.

1 Introduction

Due to the arrival of new multicore computer architectures, parallel programming is becoming more and more widespread. Fundamental changes in computer architecture induce changes in the way we write and think about programs. Multiprocessor programming is more complex than programming uniprocessor machines and requires not only the understanding of new computational principles and algorithms, but also the intricacies of the underlying hardware. This makes efficient programs not only harder to produce, but also to maintain.

One programming approach is to implement existing programming abstractions using parallel algorithms under the hood. This omits low-level details such as synchronization and load-balancing from the program. Most programming languages have libraries which provide data structures such as arrays, trees, hashtables or priority queues. The challenge is to use them in parallel.

Collections come with bulk operations like mapping or traversing elements. Functional programming encourages the use of predefined combinators, which is

beneficial to parallel computations – a set of well chosen collection operations can serve as a programming model. These operations are common to all collections, making extensions difficult. In sequential programming common functionality is abstracted in terms of iterators or a generalized `foreach`. But, due to their sequential nature, these are not applicable to parallel computations which split data and assemble results [26]. This paper describes how parallel operations can be implemented with two abstractions – splitting and combining.

The approach of augmenting collection classes with a wide range of operations has been adopted in the Scala collection framework. These operations strongly resemble those found in functional languages such as Haskell. While developing Scala parallel collections, these operations had to be parallelized to make parallel collections compliant with existing collections.

Our parallel collection framework is generic and can be applied to different data structures. It enhances collections with operations executed in parallel, giving direct support for programming patterns such as map/reduce or parallel looping. Some of these operations produce new collections. Unlike other frameworks proposed so far, our solution adresses parallel construction without the aid of concurrent data structures. While data structures with concurrent access are crucial for many areas, we show an approach that avoids synchronization when constructing data structures in parallel from large datasets.

Our contributions are the following:

1. Our framework is generic in terms of *splitter* and *combiner* abstractions, used to implement a variety of parallel operations, allowing extensions to new collections with the least amount of boilerplate.

2. We apply our approach to specific collections like parallel hash tables. We do not use concurrent data structures. Instead, we structure the intermediate results and merge them in parallel. Specialized data structures with efficient merge operations exist, but pay a price in cache-locality and memory usage [28] [25]. We show how to merge existing data structures, allowing parallel construction and retaining the efficiency of the sequential access.

3. Our framework has both mutable and immutable (persistent) versions of each collection with efficient update operations.

4. We present benchmark results which compare parallel collections to their sequential variants and existing frameworks. We give benchmark results which justify the decision of not using concurrent data structures.

5. Our framework relieves the programmer of the burden of synchronization and load-balancing. It is implemented as an extension of the Scala collection framework. Due to the backwards compatibility with regular collections, existing applications can improve performance on multicore architectures.

The paper is organized as follows. Section 2 gives an overview of the Scala collection framework. Section 3 describes adaptive work stealing. Section 4 describes the design and several concrete parallel collections. Section 5 presents experimental results. Section 6 shows related work.

2 Scala Collection Framework

Scala is a modern general purpose statically typed programming language for the JVM which fuses object-oriented and functional programming [6]. Its features of interest for this paper are higher-order functions and traits. We summarize them below. These language features are not a prerequisite for parallel collections – they serve as a syntactic and design convenience. We shortly describe the basic concepts of the Scala collection framework. Readers familiar with Scala and its collections may wish to skip this section. Readers interested to learn more are referred to textbooks on Scala [7].

In Scala, functions are first-class objects – they can be assigned to variables or specified as arguments to other functions. For instance, to declare a function that increments a number and assign it to a variable:

```
var add = (n: Int) => n + 1
```

First-class functions are useful for collection methods. For example, method `find` returns the first element in the collection that satisfies some predicate. The following code finds the first even number in the list of integers `lst`:

```
lst.find(_ % 2 == 0)
```

We have used some syntactic sugar above. Since the `find` method expects a function from an integer to boolean, the local type inference mechanism will deduce that the function expects an integer. Since the argument appears only once in the body of the function, its occurence can be replaced by the placeholder symbol `_`, making the code cleaner. In languages like Java without first-class functions, anonymous classes can achieve the same effect.

Traits are similar to Java interfaces and may contain abstract methods. They also allow defining concrete methods. Multiple traits can be mixed together into a class using the **with** keyword. Here is an example of a trait describing an iterator:

```
trait Iterator[T] {
  def hasNext: Boolean
  def next: T
  def foreach[U](f: T => U) = while (hasNext) f(next)
}
```

Collections form a class hierarchy with the most general collection type `Traversable`, which is subclassed by `Iterable`, and further subclassed by `Set`,

`Seq` and `Map`, representing sets, sequences and maps, respectively [8] [18]. Collections are in the package `scala.collection`, with 2 subpackages. Collections in the `mutable` package additionally allow in-place modifications, while those in the `immutable` package cannot be modified – e.g. adding an element to the set produces a new set. There exist efficient implementations for most immutable data structures [17] [19]. Some operations (`filter`, `take` or `map`) produce collections as results. The requirement in the framework is that these methods return the same type of the collection as the receiver collection. This could be ensured by refining the return types of all the methods returning collections in every collection class, leading to low maintainability and a large amount of boilerplate. In order to avoid this, each collection type trait (such as `Traversable`, `Iterable` or `Seq`) has a corresponding template trait `TraversableLiek`, `IterableLike` or `SeqLike` with an additional representation type `Repr` which is instantiated to the concrete collection type once this template trait is mixed in with a concrete collection class. All methods returning a collection of the same type as the collection itself have their return type declared as `Repr`.

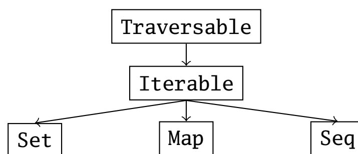

Figure 1: Collection base classes hierarchy

Collections use objects of type `Builder`. `Builder` declares a method `+=` for adding elements to the builder. Its method `result` is called after all the desired elements have been added and it returns the collection. After calling `result` the contents of the builder are undefined and the builder cannot be used again before calling the `clear` method. Each collection provides a specific builder.

We give a short example program (Fig. 2). Assume we have two sequences `names` and `surnames`. We want to group names starting with 'A' which have same surnames and print all such names and surnames for which there exists at most one other name with the same surname. We have omitted how we obtained the actual sequences of names and surnames as this is not relevant for the example. The example uses *for-comprehensions* [7] to iterate the sequence of pairs of names and surnames obtained by `zip` and filter those which start with 'A'. They are grouped according to the surname (second pair element) with `groupBy`. Surname groups with 2 or less names are printed. The sugared code on the left is translated to a sequence of method calls similar to the one shown on the right. PLINQ uses a similar approach of translating a query-based DSL into method calls. Readers interested in exact rules of for-comprehensions are referred to [7].

We want to run such programs in parallel, but new operations have to be integrated with the existing collections. Data Parallel Haskell defines a new set of

names for parallel operations [22]. Method calls in existing programs have to be modified to use corresponding parallel operations. This clutters the namespace with new names, the new names cannot be used in existing for-comprehensions and existing programs have to be modified. A different approach is implementing parallel operations in separate classes. We add a method `par` to regular collections which returns a parallel version of the collection pointing to the same underlying data. We also add a method `seq` to parallel collections to switch back. Furthermore, we define a separate hierarchy of parallel sequences, maps and sets which inherit corresponding general collection traits `GenSeq`, `GenMap` and `GenSet`.

```
val withA = for {
  (n, s) <- names zip surnames
  if n startsWith "A"
} yield (n, s)
val groups = withA.groupBy(_._2)
for {
  (surname, pairs) <- groups
  if pairs.size < 3
  (name, surname) <- pairs
} println(name, surname)
```

```
val groups = names.zip(surnames)
  .filter(_._1.startsWith("A"))
  .groupBy(_._2)
groups.filter(_._2.size < 3)
  .flatMap(_._2)
  .foreach(p => println(p))
```

Figure 2: Example program

3 Adaptive work stealing

When using multiple processors load-balancing techniques are required. In our case operations are performed on elements of the collection so dividing work can be done in straightforward way by partitioning the collection into element subsets. How partitioning is exactly done for an arbitrary collection is described later in the paper. Classes in collection frameworks often provide users with a method that performs some operation on every element of the collection – in the case of Scala collection framework this operation is known as the `foreach` method. Implementing a parallel `foreach` method requires that subsets of elements are assigned to different processors. Collection subsets can be assigned to different threads – each time a user invokes the `foreach` method on some collection, a thread is be created and assigned a subset of elements to work on. However, thread creation is expensive and can exceed the cost of the collection operation by several orders of magnitude. For this reason it makes sense to use a pool of worker threads in sleeping state and avoid thread creation each time a parallel operation is invoked.

There exists a number of frameworks that provide thread pools. One of them is the Java Fork/Join Framework [4]. It introduces an abstraction called a fork/join task which describes a unit of work to be done. This framework also manages a pool of worker threads, each being assigned a queue of fork/join

tasks. Each task may spawn new tasks (`fork`) and later wait for them to finish (`join`). Scala parallel collections use it to efficiently schedule tasks between processors.

The simplest way to schedule work between processors is to divide it in fixed-size chunks and schedule an equal part of these on each processor. The problem with this approach is twofold. First of all, if one chooses a small number of chunks, this can result in poor workload-balancing. In particular, at the end of the computation a processor may remain with a relatively large chunk, and all other processors may have to wait for it to finish. On the other hand, large number of chunks guarantees better granularity, but imposes a high overhead, since each chunk requires some scheduling resources. One can derive optimal expressions for optimal sizes of these chunks [1], but these are only appropriate for a large number of processors [3]. Other approaches include techniques such as *guided self scheduling* [2] or *factoring* [3], which were originally devised for computers with a large number of processors. An optimal execution schedule may depend not only on the number of processors and data size, but also on irregularities in the data and processor availability. Because these circumstances cannot be anticipated in advance, it makes sense to use adaptive scheduling. Work is divided to tasks and distributed among processors. Each processor maintains a task queue. Once a processor completes a task, it dequeues the next one. If the queue is empty, it tries to steal a task from another processor's queue. This technique is known as work stealing [11] [5]. We use the Java fork-join framework to schedule tasks [4]. The fork/join pool abstraction can be implemented in a number of ways, including work stealing, as it is the case with Java Fork/Join Framework [4]. For effectiveness, work must be partitioned into tasks that are small enough, which leads to overheads if there are too many tasks.

Assuming uniform amount of work per element, equally sized tasks guarantee that the longest idle time is equal to the time to process one task. This happens if all the processors complete when there is one more task remaining. If the number of processors is P, the work time for $P = 1$ is T and the number of tasks is N, then equation 1 denotes the theoretical speedup in the worst case. Thread wake-up times, synchronization and other aspects have been omitted from this idealized analysis.

$$speedup = \frac{T}{(T - T/N)/P + T/N} \underset{P \to \infty}{\to} N \tag{1}$$

In practice, there is an overhead with each created task – fewer tasks can lead to better performance. But this can also lead to worse load-balancing. This is why we've used exponential task splitting [12]. If a worker thread completes its work with more tasks in its queue that means other workers are preoccupied with work of their own, so the worker thread does more work with the next task. The heuristic is to double the amount of work (Fig. 3). If the worker thread hasn't got more tasks in its queue, then it steals tasks. The stolen task is always the biggest task on a queue. There are two points worth mentioning here. First, stealing tasks is generally more expensive than just popping them

from the thread's own queue. Second, the fork/join framework allows only the oldest tasks on the queue to be stolen. The former means the less times stealing occurs, the better – so we will want to steal bigger tasks. The latter means that which task gets stolen depends on the order tasks were pushed to the queue (forked) – one can be selective about it. Stolen tasks are split until reaching threshold size – the need to steal indicates that other workers may be short on tasks too. This is illustrated in Fig. 3.

Once a method is invoked on a collection, the collection is split into two parts. For one of these parts, a task is created and forked. Forking a task means that the task gets pushed on the processor's task queue. The other part gets split again in the same manner until a threshold is reached – at that point that subset of the elements in the collection is operated on sequentially. After finishing with one task, the processor pops a task of its queue if it is nonempty. Since tasks are pushed to the queue, the last (smallest) task pushed will be the first task popped. At any time the processor tries to pop a task, it will be assigned an amount of work equal to the total work done since it started with the leaf. On the other hand, if there is a processor without work on its queue, it will steal from the opposite side of the queue were the first pushed task is. When a processor steals a task, it divides the subset of the collection assigned to that task until it reaches threshold size of the subset. To summarize – stolen tasks are divided into exponentially smaller tasks until a threshold is reached and then handled sequentially starting from the smallest one, while tasks that came from the processor's own queue are handled sequentially straight away. An example of exponential splitting with 2 processors is shown on the right in figure 3.

The worst case scenario is a worker being assigned the biggest task it processed so far when that task is the last remaining. We know this task came from the processor's own queue (otherwise it would have been split, enabling the other processors to steal and not be idle). At this point the processor will continue working for some time T_L. We assume input data is uniform, so T_L must be equal to the time spent up to that moment. If the task size is fine-grained enough to be divided among P processors, work up to that moment took $(T - T_L)/P$, so $T_L = T/(P+1)$. Total time for P processors is then $T_P = 2T_L$. The equation 2 gives a bound on the worst case speedup, assuming $P \ll N$:

$$speedup = \frac{T}{T_P} = \frac{P+1}{2} \tag{2}$$

This estimate says that the execution time is never more than twice as great as the lower limit, given that the biggest number of tasks generated is $N \gg P$. To ensure this, we define the minimum task size as $threshold = \max(1, n/8P)$, where n is the number of elements to process.

Two further optimizations have been applied in our implementation. When splitting a task into two tasks we do not fork both tasks, pushing them both to the queue only to pop one of them [12]. Instead, we only push one of the tasks to the queue, and operate on the other one directly. Since pushing and popping to the queue involves synchronization, this leads to performance improvements.

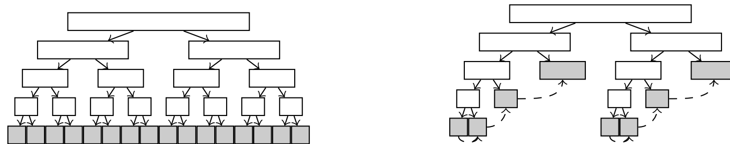

Figure 3: Fine-grained and exponential task splitting

Furthermore, Java Fork/Join Framework supports unforking tasks that have been put to the task queue. After a processor finishes with one of the tasks, it tries to unfork the task that was pushed to the queue before the last one. It does so by invoking the `tryUnfork` method. This proved to have a significant impact on performance.

Equation 2 has the consequence that the threshold size must depend on the number of processors. We define it according to 3, were n is the number of elements in the collection and P is the number of processors. This means that the number of tasks produced will be one order of magnitude greater than the number of processors if no work stealing occurs. We found that this rule of the thumb works well in practice.

$$threshold = \max(1, \frac{n}{8P}) \quad (3)$$

An important thing to notice here is that depending on the threshold one can control the maximum number of tasks that get created. Even if the biggest tasks from each task queue get stolen each time, the execution degenerates to the balanced computation tree shown in figure 3. The likelihood of this to happen has shown to be extremely small in practice and exponential splitting generates less tasks than dividing the collection into equal parts.

4 Design and implementation

We now describe how Scala parallel collections are implemented. We describe abstract operations on parallel collections that are needed to implement other parallel operations. We then classify operations into groups and describe how operations in different groups are implemented. Finally, we describe implementations of several concrete classes in our framework. The sequential versions of these classes are the representatives of the sequential collection framework, which is why we chose to implement parallel versions of these. The code examples we show are simplified with respect to actual code for purposes of clarity and brevity[1].

[1] Variance and bounds annotations have been omitted, as well as implicit parameters. Only crucial classes in the hierarchy are discussed. In some places, we have simplified the code by avoiding pattern matching. Complete and correct source code can be obtained at `http://lampsvn.epfl.ch/svn-repos/scala/scala/trunk/src/library/scala/collection/parallel/`.

4.1 Splitters and combiners

For the benefits of easy extension and maintenance we want to define most operations (such as `filter` or `flatMap` from Fig. 2) in terms of a few abstractions. The usual approach is to use an abstract `foreach` method or iterators. Due to their sequential nature, they are not applicable to parallel operations. In addition to element traversal, we need a split operation that returns a non trivial partition of the elements of the collection. The overhead induced by splitting the collection should be as small as possible – this influences the choice of the underlying data structure. We define *splitters* – iterators which have operations `next` and `hasNext` used to traverse. In addition, a splitter has a method `split` which returns a sequence of splitters iterating over disjunct subsets of elements. This allows parallel traversal. The original iterator becomes invalidated after calling `split`.

```
trait Splitter[T] extends Iterator[T] {
  def split: Seq[Splitter[T]]
}

trait Combiner[T, Coll] extends Builder[T, Coll] {
  def combine(other: Combiner[T, Coll]): Combiner[T, Coll]
}
```

Method `split` returns a sequence of splitters such that the union of the elements they iterate over contains all the elements remaining in the original splitter. All these splitters are disjoint. Parallel sequences define a more specific splitter `PreciseSplitter` which inherits `Splitter` and allows splitting the elements into subsets of arbitrary sizes, which is required to implement certain sequence operations.

Some operations produce collections (e.g. `filter`). Collection parts produced by different workers must be combined into the final result and *combiners* abstract this. Type parameter `T` is the element type, and `Coll` is the collection type. Parallel collections provide combiners, just as regular collections provide builders. Method `combine` takes another combiner and produces a combiner containing the union of their elements. Both combiners become invalidated after its invocation. Combining results from different tasks occurs more than once during a parallel operation in a tree-like manner (Fig. 3). The combine operation ideally has complexity $O(1)$ and no more than $O(\log n)$, where n is the number of elements in the combiners.

The parallel collection base trait `ParIterable` extends the `GenIterable` trait. It defines operations `splitter` and `newCombiner` which return a new splitter and a new combiner, respectively. Subtraits `ParSeq`, `ParMap` and `ParSet` define parallel sequences, maps and sets.

```
class Map[S](f: T => S, s: Splitter[T]) extends Task {
  var cb = newCombiner
  def split = s.split.map(subspl => new Map[S](f, subspl))
  def leaf() = while (s.hasNext) cb += f(s.next)
```

```
  def merge(that: Map[S]) = cb = cb.combine(that.cb)
}
```

Parallel operations are implemented within tasks, corresponding to those described previously. Tasks define `split`, `merge` and `leaf`. For example, the `Map` task is given a mapping function `f` of type `T => S` and a splitter `s`. Tasks are split to achieve better load balancing – the `split` typically calls `split` on the splitter and maps subsplitters into subtasks. Once the threshold size is reached, `leaf` is called, mapping the elements and adding them into a combiner. Results from different processors are merged hierarchically using the `merge` method, which merges combiners. In the computation root `cb` is evaluated into a collection. More than 40 collection operations were parallelized and some tasks are more complex – they handle exceptions, can abort or communicate with other tasks, splitting and merging them is often more involved, but they follow this pattern.

4.2 Common operations

Scala collections come with a wide range of operations. We divide them into groups, and show how to implement operations using abstract operations provided by specific collections.

One of the simplest operations found in our collection framework is the `foreach` method [8].

```
def foreach[U](f: T => U): Unit
```

It takes a higher-order function `f` and invokes that function on each element. The return value of `f` is ignored. The `foreach` method has two properties. First is that there are no dependencies between processors working on different collection subsets. The second is that it returns no value[2]. In other words, `foreach` is trivially parallelizable.

When `foreach` is invoked, a new task is created and submitted to the fork/join pool. This task behaves as described in the previous section. To split the elements of the collection into subsets, it invokes the `split` method of its splitter. Note that `split` may return more than two subiterators (its return type is `Seq[Splitter]`). This is more general than what was described in section 3 as each internal node in the computation tree can have more than two children, but the concept remains the same. The splitting and forking new tasks continues until splitter sizes reach a threshold size. At that point splitters are used to traverse the elements – function `f` is invoked on elements of each splitter. Once that is done, the task ends. Another example of a method that does not return a value is `copyToArray`.

Most other methods return a result. For instance, the `reduce` applies a binary associative operator to elements of the collection to obtain a result:

```
def reduce[U >: T](op: (U, U) => U): U
```

[2]Type `Unit` in Scala is the equivalent of `void` in Java and denotes no value.

It takes a binary function `op` which takes two elements of the collection and returns a new element. If the elements of the collection are numbers, `reduce` can take a function that adds its arguments. Another example is concatenation for collections that hold strings or lists. Operator `op` must be associative, because the order in which subsets of elements are partitioned and results brought together is undeterministic. Relative order is preserved – the operator does not have to be commutative. The `reduce` operation is implemented like `foreach`, but once a task ends, it returns its result to the parent task. Once the parent task is joined its children in the computation tree, it uses the `op` to merge the results. Other methods implemented in a similar manner are `aggregate`, `fold`, `count`, `max`, `min`, `sum` and `product`.

So far different collection subsets have been processed independently. For some methods results obtained by one of the tasks can influence the results of other tasks. One example is the `forall` method:

```
def forall(p: T => Boolean): Boolean
```

This method only returns `true` if the predicate argument `p` returns `true` for all elements. Sequential collections may take advantage of this fact by ceasing to traverse the elements once an element for which `p` returns `false` is found. Parallel collections have to communicate that the computation may stop. The `Signalling` trait mixed in with each splitter allows tasks using splitters obtained from the same root splitter to send messages to each other. It contains a flag which denotes whether a computation may stop. When the `forall` encounteres an element for which the predicate is not satisfied, it sets the flag. Other tasks periodically check the flag and stop processing elements if it is set. Every splitter has a reference to an instance of this trait called a context. It provides methods such as accessing an internal flag which denotes whether or not the computation should stop. This internal flag is implemented as a *volatile* boolean variable. When the `forall` method encounteres an element for which the predicate is not satisfied, it sets the flag. Other tasks periodically check the flag. Once they detect it is false, they stop processing the elements and return. This can lead to performance gains.

Tasks like `exists`, `find`, `startsWith`, `endsWith`, `sameElements` and `corresponds` use the same mechanism to detect if the computation can end before processing all the elements. Merging the results of these tasks usually amounts to a logical operation. One other method we examine here is `prefixLength`:

```
def prefixLength(p: T => Boolean): Int
```

which takes a predicate and returns the number of initial elements in the sequence that satisfy the predicate. Once some task finds an element e that does not satisfy the predicate, not all tasks can stop. Tasks that operate on parts of the sequence preceding e may still find prefix length to be shorter, while tasks operating on the following subsequences cannot influence the result and may terminate. To share information about the element's exact position, `Signalling` has an integer flag that can be set by different processors using a

compare and swap operation. Since changes to the flag are monotonic, there is no risk of the ABA problem [16]. What the `Signalling` trait provides is an integer flag and methods to access and modify it atomically. The method `setIndexFlagIfLesser` displayed below implements a lock-free decrement of the atomic flag using compare-and-set operation provided by Java `AtomicInteger` class [3]:

```
def setIndexFlagIfLesser(f: Int) = {
  var loop = true
  do {
    val old = flag.get
    if (f >= old) loop = false
    else if (flag.CAS(old, f)) loop = false
  } while (loop);
}
```

This method decrements the flag if the provided value is smaller than the flag. If the atomic compare-and-set operation detects that the value has changed in the meanwhile, the procedure is repeated again. Note that changing the integer flag with other methods that the `Signalling` provides (such as unconditional set) could potentially lead to the ABA problem [16], where a reference is read once by one processor, changed once by other processors, then changed back again. The changes remain undetected for the original processor. However, our operations limit themselves to using only monotonic changes of the integer flag so there is no risk of this.

The `prefixLength` method uses the atomic integer flag to decrement it if possible. This means that if there is some other task which found a preceeding element not satisfying the predicate, the flag will not be set. Other tasks can read this flag periodically and decide whether or not they should terminate. The only question remaining is how to merge two task results in the computation tree. The way this is done is the following – if the left task in the computation tree returned the prefix length smaller than the number of elements it processed, then that is the result. Otherwise, their results are summed together. Other methods that use integer flags to relay information include `takeWhile`, `dropWhile`, `span`, `segmentLength`, `indexWhere` and `lastIndexWhere`.

Many methods have collections as result types. A typical example of these is the `filter` method:

```
def filter(p: T => Boolean): Repr
```

which returns a collection containing elements for which `p` holds. Tasks in the computation tree must merge combiners returned by their subtasks by invoking `combine`. Methods such as `map`, `take`, `drop`, `slice`, `splitAt`, `zip` and `scan` have the additional property that the resulting collection size is known in advance. This information can be used in specific collection classes to override default implementations in order to increase performance. For instance,

[3] The actual name of the method `compareAndSet` is not used here for brevity.

`ParArray` is optimized to perform these operations by first allocating the internal array and then passing the reference to all the tasks to work on it and modify it directly, instead of using a combiner. Methods that cannot predict the size of the resulting collection include `flatMap`, `partialMap`, `partition`, `takeWhile`, `dropWhile`, `span` and `groupBy`. Some of these will not just trivially merge the two combiners produced by the subtasks, but process them further in some way, such as the `span`. Method `span` returns a pair of two collections a and b – first contains the longest prefix of elements that satisfy a predicate, and the second contains the rest. Merging results of two tasks T_1 and T_2 that have combiner pairs results (a_1, b_1) and (a_2, b_2), respectively, depends on whether the T_1 found only elements satisfying the predicate – if so, then the result should be the pair $(a_1 a_2, b_2)$, where concatenation denotes merging combiners. Otherwise, the result is $(a_1, b_1 a_2 b_2)$.

Parallel sequences described by the trait `ParSeq` refine the return type of their `splitter` method – they return objects of type `PreciseSplitter`. Method `psplit` of the `Splitter` subclass `PreciseSplitter` for parallel sequences is more general than `split`. It allows splitting the sequence into subsequences of arbitrary length. Sequences in Scala are collections where each element is assigned an integer, so splitting produces splitters the concatenation of which traverses all the elements of the original splitter in order. Some methods rely on this. An example is:

```
def zip[S](that: ParSeq[S]): ParSeq[(T, S)]
```

which returns a sequence composed of corresponding pairs of elements belonging to the receiver and `that`. The regular `split` method would make implementation of this method quite difficult, since it only guarantees to split elements into subsets of any sizes – `that` may be a parallel sequence of a different type. Different splitters may split into differently sized subsequences, so it is no longer straightforward to determine which are the corresponding elements of the collections that the leaf tasks should create pairs of – they may reside in different splitters. The refined `psplit` method allows both sequences to be split into subsequences of the same size. Other methods that rely on the refined split are `startsWith`, `endsWith`, `patch`, `sameElements` and `corresponds`.

4.3 Parallel array

Arrays are mutable sequences – class `ParArray` stores the elements in an array. It is a parallel sequence and extends the `ParSeq` trait. We now show how to implement splitters and combiners for it.

Splitters. A splitter contains a reference to the array, and two indices for iteration bounds. Method `split` divides the iteration range in 2 equal parts, the second splitter starting where the first ends. This makes `split` an $O(1)$ method. We only show method `split` below:

```
class ArraySplitter[T](a: Array[T], i: Int, until: Int)
extends Splitter[T] {
```

```
  def split = Seq(
    new ArraySplitter(a, i, (i + until) / 2),
    new ArraySplitter(a, (i + until) / 2, until))
}
```

Combiners do not know the final array size (e.g. `flatMap`), so they construct the array lazily. They keep a linked list of buffers holding elements. A buffer is either a dynamic array[4] or an unrolled linked list. Method `+=` adds the element to the last buffer and `combine` concatenates the linked lists (an $O(1)$ operation). Method `result` allocates the array and executes the `Copy` task which copies the chunks into the target array (we omit the complete code here). Copying is thus parallelized as well.. To copy the elements from the chained arrays into the resulting array a new set of tasks is created which form another computation tree. An effect known as *false sharing* may occur in situations where different processors write to memory locations that are close or overlap and thus cause overheads in cache coherence protocols [16]. In our case, only a small part of an array could be falsely shared at the bounds of different chunks and writes from different chunks go left to right. False sharing is unlikely given that chunk sizes are evenly distributed.

When the size is not known a priori, evaluation is a two-step process. Intermediate results are stored in chunks, an array is allocated and elements copied in parallel.

```
class ArrayCombiner[T] extends Combiner[T, ParArray[T]] {
  val chunks = LinkedList[Buffer[T]]() += Buffer[T]()
  def +=(elem: T) = chunks.last += elem
  def combine(that: ArrayCombiner[T]) = chunks append that.chunks
  def result = exec(new Copy(chunks, new Array[T](chunks.fold(0)(_+_.size))))
}
```

Operations creating parallel arrays that know their sizes in advance (e.g. `map`) are overridden for `ParArray` to allocate an array and work on it directly. These methods do not use lazy building schemes described above and avoid the two step process described above.

4.4 Parallel rope

To avoid the copying step altogether, a data structure such as a *rope* is used to provide efficient splitting and concatenation [15]. Ropes are binary trees whose leaves are arrays of elements. They are used as an immutable sequence which is a counterpart to the `ParArray`. Indexing an element, appending or splitting the rope is $O(\log n)$, while concatenation is $O(1)$. However, iterative concatenations leave the tree unbalanced. Rebalancing can be called selectively.

Splitters are implemented similarly to `ParArray` splitters. They maintain a reference to the original rope, and the position in the rope. Splitting divides the

[4] In Scala, this collection is available in the standard library and called *ArrayBuffer*. In Java, for example, it is called an *ArrayList*.

rope into several parts, assigning each part to a new splitter. This operation is bound by the depth of the tree, making it logarithmic.

Combiners may use the append operation for `+=`, but this results in unbalanced ropes [15]. Instead, combiners internally maintain a concatenable list of array chunks. Method `+=` adds to the last chunk. The rope is constructed at the end from the chunks using the rebalancing procedure [15].

4.5 Parallel hash table

Associative containers, or maps, are another widely available collection. Associative containers implemented as hash tables guarantee $O(1)$ access with high probability. There is plenty of literature available on concurrent hash tables [21]. We describe a technique that constructs array-based hash tables in parallel by assigning non-overlapping element subsets to workers, avoiding the need for synchronization. This technique is applicable both to chained hash tables (used for `ParHashMap`) and linear hashing (used for `ParHashSet`).

Splitters maintain a reference to the hash table and two indices for iteration range. Splitting divides the range in 2 equal parts. For chained hash tables, a splitter additionally contains a pointer into the bucket. Since buckets have a probabilistic bound on lengths, splitting a bucket remains an $O(1)$ operation.

Combiners do not contain hash tables, since `combine` would require traversing both hash tables and merging corresponding buckets, thus having linear complexity. We want to ensure that copying occurs as least as possible and that it can be parallelized.

In the case of a `ParArray`, we parallelize the copying step by assigning different ranges to different processes. We want to achieve the same with parallel hash maps, to avoid synchronization. Given a set of elements, we want to construct a hash table using multiple processors. Subsets of elements will have to be assigned to different processors and will have to occupy a contiguous block of memory to avoid *false sharing*. To achieve this, elements are partitioned by their hashcode prefixes, which divide the table into logical blocks. This will ensure that they end up in different blocks, independently of the final table size. The resulting table is filled in parallel. We know describe this basic intuition in more detail.

```
class TableCombiner[K](ttk: Int = 32) extends Combiner[K, ParHashTable[K]] {
  val buckets = new Array[Unrolled[K]](ttk)
  def +=(elem: K) = buckets(elem.hashCode & (ttk - 1)) += elem
  def combine(that: TableCombiner[K]) = for (i <- 0 until ttk)
    buckets(i) append that.buckets(i)
  private def total = buckets.fold(0)(_ + _.size)
  def result = exec(new Fill(buckets, new Array[K](nextPower2(total / lf)))
}
```

Combiners keep an array of 2^k buckets, where k is a constant such that 2^k is greater than the number of processors to ensure good load balancing (from experiments, $k = 5$ works well for up to 8 processors). Buckets are unrolled

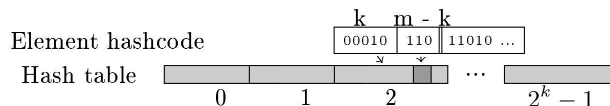

Figure 4: Hash code mapping

linked lists. Method `+=` computes the element hashcode and adds it to the bucket indexed by the k-bit hashcode prefix. Unrolled list tail insertion amounts to incrementing an index and storing an element into an array in most cases, occasionally allocating a new node. We used $n = 32$ for the node size. Method `combine` concatenates all the unrolled lists – for a fixed 2^k, this is an $O(1)$ operation.

Once the first step of the computation completes and reaches the root of the task tree, we have all the elements that will appear in the final hash map grouped into buckets according to their hashcode prefix. Method `result` is called at this point – the total number of elements `total` is obtained from bucket sizes. The required table size is computed by dividing `total` with the load factor `lf` and rounding to the next power of 2. The table is allocated and the `Fill` task is run, which can be split in up to 2^k subtasks, each responsible for one bucket. It stores the elements from different buckets into the hash table. Assume table size is $sz = 2^m$. The position in the table corresponds to the first m bits of the hashcode. The first k bits denote the index of the table block, and the remaining $m - k$ bits denote the position within that block (Fig. 4). Elements of a bucket have their first k bits the same and are all added to the same block – writes to different blocks are not synchronized. With linear hashing, elements occasionally "spill" to the next block. The `Fill` task records and inserts them into the next block in the merging step. The average number of spills is equal to average collision lengths – a few elements.

4.6 Parallel hash trie

A hash trie is an immutable map or set implementation with efficient element lookups and updates ($O(\log_{32} n)$) [19]. Updates do not modify existing tries, but create new versions which share parts of the data structure. Hash tries consist of a root table of 2^k elements. Adding an element computes the hash code and takes the first k bits for the table index i. In the case of a collision a new array is allocated and stored into entry i. Colliding elements are stored in the new array using the next k bits. This is repeated as long as there are collisions. The resulting data structure forms very shallow tree, so only $\log_{32} n$ indirections are required to find the correct element. To ensure low space consumption, each node has a 2^k bitmap to index its table (typically $k = 5$) [19]. We found hash tries to be comparable in performance to hash tables, providing faster iteration and somewhat slower construction and lookup, additionally being a persistent data structure. Hash tries have low space overheads and good cache-locality.

Splitters maintain a reference to the hash trie data structure. Method `split` divides the root table into 2 new root tables, assigning each to a new splitter (an $O(1)$ operation). This is shown in figure 5. Since parallel hash tries are used to implement maps and sets, and not sequences, there is no need to implement the `psplit` method.

Combiners contain hash tries. Method `combine` could merge the hash tries (figure 5). For simplicity, the hash trie root nodes are shown to contain only five entries. The elements in the root table are copied from either of the root tables, unless there is a collision, as with subtries B and E which are recursively merged. This technique turns out to be more efficient than sequentially building a trie – we observed speedups of up to 6 times. We compare the performance recursive merging against hash table merging and sequentially building tries in figure 6. Recursive merging can also be done in parallel. Whenever two subtries collide, we can spawn a new task to merge the colliding tries. Elements in the two colliding tries have the property that they all share the common hashcode prefix, meaning they will all end up in the same subtrie – the merge can be done completely independently of merging the rest of the tries. Parallel recursive merging is thus applicable only if the subtries merged in a different task are large enough. A problem that still remains is that we do more work than is actually necessary by merging more than once – a single element may be copied more than once while doing 2 subsequent recursive merges. Backed by experimental evidence presented in Fig. 6, we postulate that due to having to copy single elements more than once, although recursive merging requires less work than sequential construction, it still scales linearly with the trie size.

In a typical invocation of a parallel operation, combine method is invoked more than once (see figure 3), so invoking a recursive merge would still yield an unacceptable performance. This is why we use the two-step approach shown for hash tables, which results in better performance. Combiners maintain 2^k unrolled lists, holding elements with the same k-bit hashcode prefixes ($k = 5$). The difference is in the method `result`, which evaluates root subtries instead of filling table blocks. Each unrolled linked lists is a list of concatenated array chunks which are more space-efficient, cache-local and less expensive to add elements to. Adding an element amounts to computing its hashcode, taking its k bit prefix to find the appropriate bucket and appending it to the end – an array index is incremented and the element is stored in most cases. Occasionally, when an array chunk gets full, a new array chunk is allocated. To be able to append elements we keep a pointer to the end of the list. In general, unrolled lists have the downside that indexing an element in the middle has complexity $O(n/m)$ where n is the number of elements in the list and m is the chunk size, but this is not a problem in our case since we never index an element - we only traverse all of the elements once.

Combiners implement the `combine` method by simply going through all the buckets and concatenating the unrolled linked lists that represent the buckets, which is a constant time operation. Once the root combiner is produced the resulting hash trie is constructed in parallel – each processor takes a bucket and constructs subtrie sequentially, then stores it in the root array. We found this

technique to be particularly effective, since adding elements to unrolled lists is very efficient and avoids merging hash tries multiple times. Another advantage that we observed in benchmarks is that each of the subtries being constructed is on average one level less deep. Processor working on the subtrie will work only on a subset of all the elements and will never touch subtries of other processors. This means it will have to traverse one level less to construct the hash trie.

4.7 Parallel range

Most imperative languages implement loops using *for*-statements. Object-oriented languages such as Java and C# also provide a *foreach* statement to traverse the elements of a collection. In Scala, for-statements like:

```
for (elem <- list) process(elem)
```

are translated into a call to the `foreach` method of the object `list`, which does not necessarily have to be a collection:

```
list.foreach(elem => process(elem))
```

For-statements in Scala are much more expressive than this and also allow filtering, mapping and pattern matching the elements. See [7] for a more complete list of *for-comprehensions*.

To traverse over numbers like with ordinary for-loops, one must create an instance of the `Range` class, an immutable collection which contains information about the number range. The only data `Range` class has stored in memory are the lower and upper bound, and the traversal step. Scala provides implicit conversions which allow a more convenient syntax to create a range and traverse it:

```
for (i <- 0 until 100) process(i)
```

The `ParRange` collection is used to parallelize for-loops. To perform the loop in parallel, the user of can write:

```
for (i <- (0 until 100).par) process(i)
```

The `ParRange` is an immutable collection which can only contain numbers within certain bounds and with certain steps. It cannot contain an arbitrary collection of integers like other sequences, so it does not implement a combiner. It only implements the `split` which simply splits the range iterator into two ranges, one containing the integers in the first half of the range and the other integers in the second. The refined `split` method is implemented in a similar fashion.

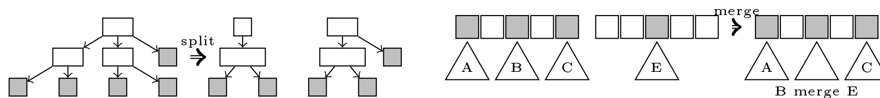

Figure 5: Hash trie operations

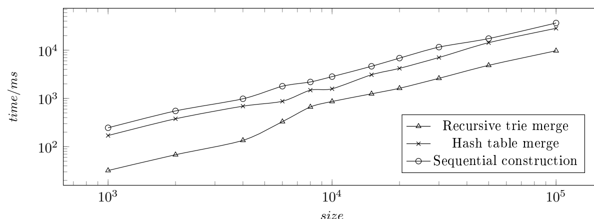

Figure 6: Recursive trie merge vs. Sequential construction

4.8 Parallel views

Assume we increment numbers in a collection `c`, take one half and sum positives:

```
c.map(_ + 1).take(c.size / 2).filter(_ > 0).reduce(_ + _)
```

Each operation produces an intermediate collection. To avoid this we provide *views*. For example, a `Filtered` view traverses elements satisfying a predicate, while a `Mapped` view maps elements before traversing them. Views can be stacked – each view points to its parent. Method `force` evaluates the view stack to a collection. In the example, calling `view` and the other methods on `c` stacks views until calling `reduce`. Reducing traverses the view to produce a concrete result. *Splitters* call `split` on their parents and wrap the subsplitters. The framework provides a way to switch between strict and lazy on one axis (`view` and `force`), and sequential and parallel on the other (`par` and `seq`), as illustrated in Fig. 7.

Parallel views reimplement behaviour of regular views in the Scala collection framework to do these non-stacking operations in parallel. They do so by extending the `ParIterable` trait and having their splitters implement the `split` method. Since their tranformer methods return views rather than collections, they do not implement combiners nor their `combine` method. Method `force` is also reimplemented to evaluate the collection represented by the view in parallel.

Every view iterator has a reference to the parent iterator it was created from. It's `next` and `hasNext` methods are implemented in terms of its parent iterator. For example, the iterator of the `Mapped` view calls the `next` method of the parent iterator to obtain an element and applies a user-defined function to it before returning. Splitters have to implement the `split` method in addition. We now give a summary of how to implement the `split` method in terms of the parent splitter for different view types.

The `Mapped` splitters are trivial – they start by splitting the parent splitter and then using the resulting splitters to produce a mapped splitter from each of them. `Taken` view splitters are parametrized by a parameter n which denotes how many initial elements of the parent collection are seen by the view. They split the parent into subsplitters, taking initial subsplitters that have the total of n or less elements. The next splitter is wrapped to return only the elements up to n, and the rest are ignored. `Sliced` and `Dropped` splitters are implemented in a similar manner. `Appended` views give an abstract view over the elements of two appended collections – their splitters implement split by simply returning two splitters, each traversing one of these two collections. `Patched` splitters are implemented in a similar manner. `Zipped` view splitters iterate over corresponding pairs of the elements from two collections and are split by splitting the parent splitters and zipping the subsplitters together.

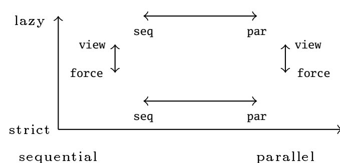

Figure 7: Strict-lazy and parallel-sequential conversions

4.9 Integration with the Scala collection framework

In terms of inheritance, parallel collections mirror the regular collection framework with corresponding traits. Parallel collection base trait `ParIterable` has descendants such as the `ParSeq`, `ParSet` and `ParMap` traits. These traits exist in the root parallel collection package `scala.collection.parallel`, and both in its `mutable` and `immutable` subpackages.

Although parallel collections should be integrated as tightly as possible with sequential collections to allow straightforward substitution of sequential collections, in a language which is not referentially transparent, a parallel collection should not be a subtype of a sequential collection. Programs written in languages that support side-effects within closures may in general (and in most cases will) produce different results depending on the order of elements the closure is invoked with. Consider the following example:

```
var num = -1
var set = false
for (x <- (0 until 100).par) if (CAS(set, false, true)) {
  num = x
}
```

Recall that this gets translated into the following:

```
var num = -1
var set = false
(0 until 100).par.foreach { x =>
  if (CAS(set, false, true)) {
    num = x
  }
}
```

The closure used within the for-comprehension, that is the `foreach` method, has a potential side-effect of writing to the variables `num` and `set` which are in scope. Using a sequential range in the for comprehension would always end the program so that `num` is set to 0. With a parallel collection, this is not so, as some processor may have started to concurrently process a different part of the range, setting `num` to something else.

One might assume that this difference is only crucial for parallel sequences and collections which guarantee traversal order, but side-effects pose a problem more generally – in a parallel collection closures are not only invoked out of order, but also concurrently by different processors, so their side-effects should be synchronized, as shown by the following example:

```
val ab = ArrayBuffer()
for (x <- (0 until 100).par) ab += x
```

As the array buffer class is not synchronized and it's accessed by different processors concurrently, this program may produce an array buffer in an invalid state.

Assuming that sequential collections guarantee one-at-a-time access, referential transparency is the necessary condition for allowing a parallel collection to be a subtype of a sequential collection and preserving correctness for all programs. Since Scala is not referentially transparent and allows side-effects, it follows that the program using a sequential collection may produce different results than the same program using a parallel collection at some point. If parallel collection types are subtypes of sequential collections, then this violates the Liskov substitution principle, as clients having references to sequential collections might not be able to use side-effects in closures freely.

For these reasons, in order to be able to have a reference to a collection which may be either sequential or parallel, there has to exist a common supertype of both collection types. We implemented a general collection class hierarchy composed of `GenTraversable`, `GenIterable`, `GenSeq`, `GenMap` and `GenSet` traits which don't guarantee in-order or one-at-a-time traversal. Corresponding sequential or parallel traits inherit from these. For example, a `ParSeq` and `Seq` are both subtypes of a general sequence `GenSeq`, but they are in no inheritance relationship with respect to each other.

The new hierarchy is shown in Fig. 8, with maps and sets trait omitted for clarity.

Clients can now refer to sequential sequences using the `Seq` trait like before and to parallel sequences using the `ParSeq` trait. To refer to a sequence whose

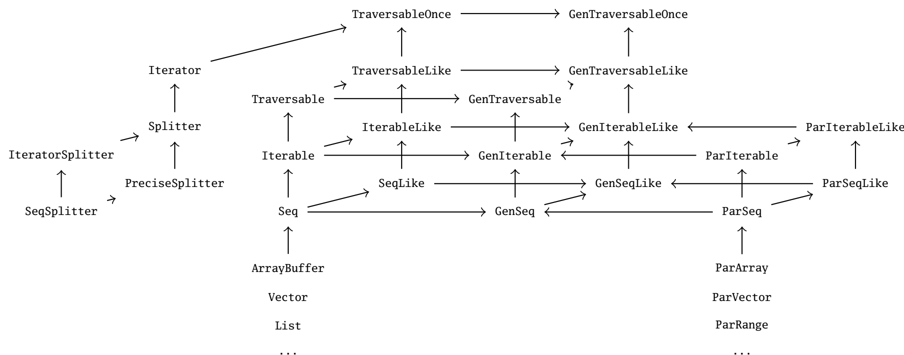

Figure 8: Hierarchy

implementation may be either parallel or sequential, clients can use the `GenSeq` trait. Note that this approach preserves source compatibility with existing code – the meaning of all existing programs remains the same.

The general collection traits provide the same methods as ordinary collections, but with fewer guarantees. Additionally, general collection traits introduce methods `seq` and `par` which return the corresponding sequential or parallel version of the collection, respectively.

Parallel collections require combiners defined by trait `Combiner` which extends the `Builder` trait. When used by regular collection methods, combiners have the same behaviour as normal builders do. Furthermore, as a counterpart to builder factories of type `CanBuildFrom` [8] [18], parallel collections have combiner factories of type `CanCombineFrom` which extends it, but returns combiners instead of ordinary builders. A combiner factory can be used anywhere in place of a builder factory.

5 Experimental results

Parallel collections were benchmarked and compared to both sequential versions and other currently available parallel collections, such as Doug Lea's `extra166.ParallelArray` for Java. We show here that their performance improves on that of regular collections and that it is comparable to different parallel collection implementations.

To measure performance, we follow established measurement methodologies [27]. Tests were done on a 2.8 GHz 4 Dual-core AMD Opteron and a 2.66 GHz Quad-core Intel i7. We first compare two JVM concurrent maps – `ConcurrentHashMap` and `ConcurrentSkipListMap` (both from the standard library) to justify our decision of avoiding concurrent containers. A total of n elements are inserted. Insertion is divided between p processors. This process is repeated over a sequence of 2000 runs on a single JVM invocation and the aver-

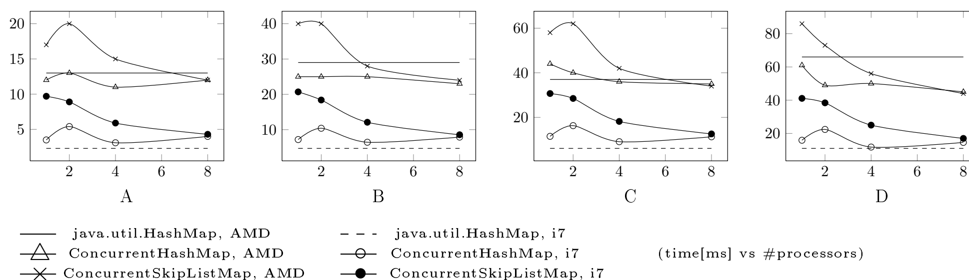

Figure 9: Concurrent insertion, total elements: (A) 50k; (B) 100k; (C) 150k; (D) 200k

age time is recorded. We compare against sequentially inserting n elements into a `java.util.HashMap`. The results are shown in figure 9 (no. processors vs. time in milliseconds), where a performance drop due to contention is observable. Concurrent data structures are general purpose and pay a performance penalty for this generality. Parallel hash tables are compared against `java.util.HashMap` in figure 10 I (mapping with a few arithmetic operations) and L (the identity function) – when no time is spent processing an element and entire time spent creating the table (L), hash maps are faster for 1 processor. For 2 or more, the parallel construction is faster.

Microbenchmarks A-L shown in Fig. 10 use inexpensive operators (e.g `foreach` writes to an array, `map` does a few arithmetic operations and the `find` predicate does a comparison). Good performance for fine-grained operators compared to which processing overhead is high means they work well for computationally expensive operators (shown in larger benchmarks M-O). Parallel array is compared against Doug Lea's `extra166y.ParallelArray` for Java.

Larger benchmarks[5] are shown at the end. The Coder benchmark brute-force searches a set of all sentences of english words for a given sequence of digits, where each digit corresponds to letters on a phone keypad (e.g. '2' represents 'A', 'B' and 'C'; '43' can be decoded as 'if' or 'he'). It was run on a 29 digit sequence and around 80 thousand words. The Grouping benchmark loads the words of the dictionary and groups words which have the same digit sequence.

6 Related work

General purpose programming languages and platforms provide various forms of parallel programming support. Most have multithreading support. However, starting a thread can be computationally expensive and high-level primitives for parallel computing are desired. We give a short overview of the related work in

[5]Complete source code is available at: `http://lampsvn.epfl.ch/svn-repos/scala/scala/trunk/`

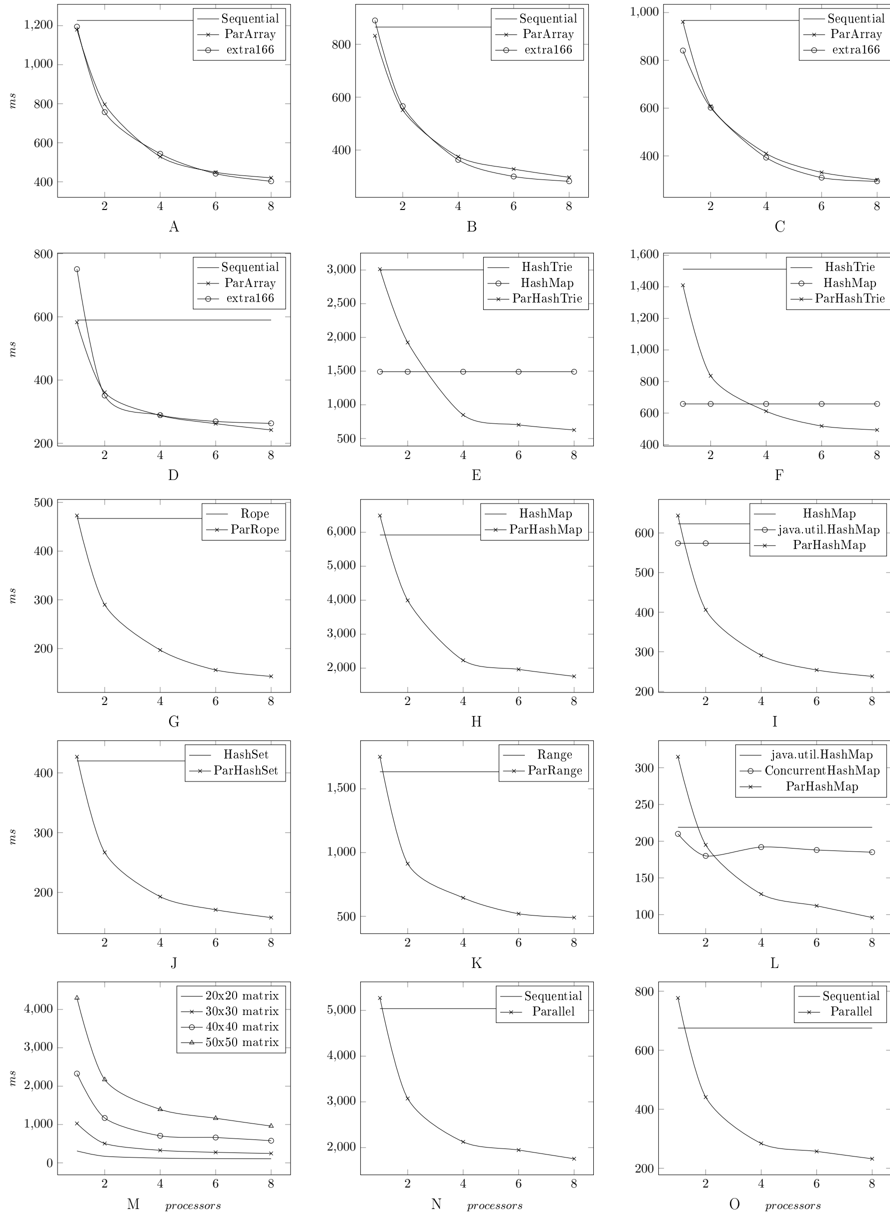

Figure 10: Benchmarks (running time [ms] vs number of processors): (A) `ParArray.foreach`, 200k; (B) `ParArray.reduce`, 200k; (C) `ParArray.find`, 200k; (D) `ParArray.filter`, 100k; (E) `ParHashTrie.reduce`, 50k; (F) `ParHashTrie.map`, 40k; (G) `ParRope.map`, 50k; (H) `ParHashMap.reduce`, 25k; (I) `ParHashMap.map`, 40k; (J) `ParHashSet.map`, 50k; (K) `ParRange.map`, 10k; (L) `ParHashMap.map(id)`, 200k; (M) Matrix multiplication; (N) Coder; (O) Grouping

the area of data parallel frameworks, which is by no means comprehensive.

There exists a body of work on data structures which allow access from several threads, either through locking or wait-free synchronization primitives [21]. They provide atomic operations such as insertion or lookup. Operations are guaranteed to be ordered, paying a price in performance – ordering is not always required for bulk parallel executions [26].

.NET langugages support patterns such as parallel looping, aggregations and the map/reduce pattern [9]. .NET Parallel LINQ provides parallelized implementations query operators. On the JVM, one example of a data structure with parallel operations is the Java `ParallelArray` [10], an efficient parallel array implementation. Its operations rely on the underlying array representation, which makes them efficient, but also inapplicable to other data representations. Data Parallel Haskell has a parallel array implementation with bulk operations [22].

Some languages recognized the need for catenable data structures. Fortress introduces conc-lists, tree-like lists with efficient concatenation [25] [28] [13] [14]. We generalize them to maps and sets, and both mutable and immutable data structures.

Intel TBB for C++ bases parallel traversal on iterators with splitting and uses concurrent containers. Operations on concurrent containers are slower than their sequential counterparts [23]. STAPL for C++ has a similar approach – they provide thread-safe concurrent objects and iterators that can be split [24]. The STAPL project also implements distributed containers. Data structure construction is achieved by concurrent insertion, which requires synchronization.

7 Conclusion

We provided parallel implementations for a wide range of operations found in the Scala collection library. We did so by introducing two divide and conquer abstractions called *splitters* and *combiners* needed to implement most operations.

In the future, we plan to implement bulk operations on concurrent containers. Currently, parallel arrays hold boxed objects instead of primitive integers and floats, which causes boxing overheads and keeps objects distributed throughout the heap, leading to cache misses. We plan to apply specialization to array-based data structures in order to achieve better performance for primitive types [20].

References

[1] C. P. Kruskall, A. Weiss: Allocating Independent Subtasks on Parallel Processors. IEEE Transactions on Software engineering, 1985.

[2] C. Polychronopolous, D. Kuck: Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. IEEE Transactions on Computers, 1987.

[3] S. F. Hummel, E. Schonberg, L. E. Flynn: Factoring: A Method for Schedulling Parallel Loops. Communications of the ACM, 1992.

[4] D. Lea: A Java Fork/Join Framework. 2000.

[5] D. Traore, J.-L. Roch, N. Maillard, T. Gautier, J. Bernard: Deque-free work-optimal parallel STL algorithms. Proceedings of the 14th Euro-Par conference, 2008.

[6] M. Odersky, et al.: An Overview of the Scala Programming Language. Technical Report LAMP-REPORT-2006-001, EPFL, 2006.

[7] M. Odersky, L. Spoon, B. Venners: Programming in Scala. Artima Press, 2008.

[8] M. Odersky: Scala 2.8 collections. EPFL, 2009.

[9] S. Toub: Patterns of Parallel Programming. Microsoft Corporation, 2010.

[10] Doug Lea's Home Page: http://gee.cs.oswego.edu/

[11] R. D. Blumofe, C. E. Leiserson: Scheduling Multithreaded Computations by Work Stealing. 35th IEEE Conference on Foundations of Computer Science, 1994.

[12] G. Cong, S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, T. Wen: Solving Large, Irregular Graph Problems Using Adaptive Work Stealing. Proceedings of the 2008 37th International Conference on Parallel Processing, 2008.

[13] William Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. Communications ACM, volume 33, 1990.

[14] Jean Vuillemin: A data structure for manipulating priority queues. Communications ACM, volume 21, 1978.

[15] H.-J. Boehm, R. Atkinson, M. Plass: Ropes: An Alternative to Strings. Software: Practice and Experience, 1995.

[16] N. Shavit, M. Herlihy: The Art of Multiprocessor Programming. Morgan Kaufmann, 2008.

[17] C. Okasaki: Purely Functional Data Structures. Cambridge University Press, 1999.

[18] M. Odersky, A. Moors: Fighting Bitrot with Types. Foundations of Software Technology and Theoretical Computer Science, 2009.

[19] P. Bagwell: Ideal Hash Trees. 2002.

[20] Iulian Dragos, Martin Odersky: Compiling Generics Through User-Directed Type Specialization. Fourth ECOOP Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems, 2009.

[21] M. Moir, N. Shavit: Concurrent data structures. Handbook of Data Structures and Applications, Chapman and Hall, 2007.

[22] S. P. Jones, R. Leshchinskiy, G. Keller, M. M. T. Chakravarty: Harnessing the Multicores: Nested Data Parallelism in Haskell. Foundations of Software Technology and Theoretical Computer Science, 2008.

[23] Intel Thread Building Blocks: Tutorial: http://www.intel.com. 2010.

[24] A. Buss, Harshvardhan, I. Papadopoulos, O. Tkachyshyn, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, L. Rauchwerger: STAPL: Standard Template Adaptive Parallel Library. Haifa Experimental Systems Conference, 2010.

[25] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr., S. Tobin-Hochstadt, et al.: The Fortress Language Specification. 2008.

[26] G. L. Steele Jr., How to Think about Parallel Programming: Not!: http://www.infoq.com/presentations/Thinking-Parallel-Programming. 2011.

[27] A. Georges, D. Buytaert, L. Eeckhout: Statistically Rigorous Java Performance Evaluation. OOPSLA, 2007.

[28] R. Hinze, R. Paterson: Finger Trees: A Simple General-purpose Data Structure. Journal of Functional Programming, 2006.