A Generic Parallel Collection Framework

Aleksandar Prokopec, Tiark Rompf, Phil Bagwell, Martin Odersky
EPFL
{firstname}.{lastname}@epfl.ch

Abstract

Most applications manipulate structured data. Modern languages and platforms provide collection frameworks with basic data structures like lists, hashtables and trees. These data structures come with a range of predefined operations which include sorting, filtering or finding elements. Such bulk operations usually traverse the entire collection and process the elements sequentially. Their implementation often 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, which can be 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.

Categories and Subject Descriptors D.1.3 [*Programming techniques*]: Concurrent programming—Parallel programming

General Terms Parallel programming, Collection libraries

Keywords parallel collections, parallel data structures, Scala

1. Introduction

With the arrival of new multicore computer architectures, parallel programming is becoming more and more widespread. Multiprocessor programming is more complex than programming uniprocessor machines and often requires platform awareness. Parallel programs are harder to produce and maintain. There are many approaches to solve this problem. One is to offer programmers existing programming abstractions and implement them using parallel algorithms under the hood. In this way, the programmer is relieved of the low-level details such as synchronization and load-balancing. General purpose programming languages often have rich collection libraries which provide data structures such as arrays, lists, trees, hashtables or priority queues. Some modern frameworks also have lock-free versions of these data structures which allow concurrent access without resorting to classical means of synchronization such as locks and monitors [23].

Many collection frameworks have collection class hierarchies with common bulk operations which include sorting, filtering, partitioning, finding elements or applying user-specific functions on elements as is the case with the map/reduce operation. Functional programming encourages the use of predefined combinators when writing a program, which is particularly well-suited for parallel operations because a set of well chosen operations provided by the library or a framework can allow the user to write efficient parallel programs. The problem, however, is that frameworks often define a set of operations common to all collections and these have to be reimplemented for each collection anew, which can make implementation and addition of new collection classes cumbersome. So far, collection frameworks have solved this problem by implementing all of their operations in terms of iterators or a generalized `foreach` method. However, due to their sequential nature, they are not applicable to parallel collection operations which require splitting data across multiple processors and assembling results. This paper describes how a wide set of parallel operations can be implemented in divide and conquer style algorithms which rely on two abstractions implemented in concrete collections - splitting and combining.

General purpose programming languages and the accompanying platforms currently provide various forms of library support for parallel programming. Most platforms offer multithreading support. However, starting and initializing a thread can be computationally expensive due to stack creation, limiting scalability. It also usually involves a lot of boilerplate on the part of the programmer, so some languages support other constructs for parallel programming. For instance, .NET langugages have support for common parallel programming patterns, such as parallel looping constructs, aggregations and the map/reduce pattern [4]. These constructs relieve the programmer of having to reimplement low-level details such as correct load-balancing between processors each time a parallel application is written. .NET Parallel LINQ provides parallelized implementations of .NET query operators. Another example is the Java `ParallelArray`, which is an extension to the JSR166 package [9]. It is an efficient parallel array implementation with many operations. These operations rely on the fact that the underlying data structure is an array, which makes them efficient, but also inapplicable to data representations for trees or hash maps. Groovy Parallel Systems [21] uses this parallel array to implement parallel collection processing, but is currently limited to collections based on arrays. Data Parallel Haskell has a parallel array implementation with parallel bulk operations [24].

Our parallel collection framework is generic and can be applied to a multitude of different data structures. It enhances collections with a large number of operations that allow efficient parallel processing of elements within the collection, giving direct support for parallel programming patterns such as map/reduce or parallel looping. Some of these operations return another collection as their return value. For instance, the `filter` method will return a new collection comprising of elements in the collection that satisfy a given predicate. Our solution adresses not only parallel traversal and processing of elements in a parallel collection, but also parallel construction of various data structures. We have benchmarked its performance and we show experimental results. It is fully compliant with the preexisting Scala collection framework in terms of its operations and integration into the class hierarchy, meaning that users do not have to switch to a different programming model to use them. It also allows straightforward definition of custom parallel collections.

Our contributions are the following:

- Our framework relies on *splitter* and *combiner* abstractions which are used to implement a variety of operations. This design allows extensions of the framework to new collection classes with a minimum amount of boilerplate.
- We apply our approach to the implementation of specific collection classes such as a parallel hash maps, describing a solution of merging them in parallel. We are not aware of this solution prior to our own.
- We present benchmark results which compare parallel collections to their sequential variants and other implementations.
- Our framework relieves the programmer of the burden of synchronization, load-balancing and other low-level details. Due to the backwards compatibility with regular collections, existing applications can use our collection framework and improve their performance on multicore architectures.

The rest of the paper is organized as follows. Section 2 gives an overview of Scala collection framework. Section 3 compares approaches to parallelizing operations on data and describes adaptive work stealing, a technique used to load-balance work between different processors. Section 4 describes abstractions used to implement parallel collection operations, and gives several case studies on concrete parallel collection classes. Section 5 contains a set of benchmarks used to measure performance of parallel collections, and Section 6 concludes.

2. Scala Collection Framework

Scala is a modern general purpose statically typed programming language which fuses object-oriented and functional programming [1]. It allows expressing common programming patterns in a concise, elegant and type-safe way. It integrates seamlessly with Java, and offers a range of features such as higher-order functions, local type inference, mixin composition and a rich type system which includes generics, path-dependent types, higher-kinded types and other. Of particular interest here are higher-order functions and traits. We summarize these below. After that, we shortly describe basic concepts of the Scala collection framework. Readers familiar with Scala and its collections may wish to skip this section. Those interested to learn more about Scala are referred to [2].

In Scala, functions are first-class objects – functions can be passed around as regular objects, 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, one could write:

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

Higher-order functions are useful for certain collection methods. For instance, the method `find` found in Scala collections 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've used some syntactic sugar in the last example. Since the `find` method expects a function from an integer to boolean, the local type inference mechanism will deduce that the argument of the provided function must be an integer. Since the argument appears only once in the body of the function, its occurence can be replaced by the placeholder symbol _, which makes the code much cleaner.

Throughout this paper we often refer to a construct called a *trait*. Traits are similar to Java interfaces in the sense that they may contain abstract methods and a class is allowed to inherit multiple traits. But traits, also known as rich interfaces, are less restrictive as they 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 in the Scala collection framework form a class hierarchy [3] [19]. A simplified version is shown in figure 1. The `Traversable` trait is a common ancestor of all collection classes. It defines an abstract method `foreach`, which traverses all of the elements of the collection and applies a specified higher-order function to each element. This style of traversal is known as the *push-style* - all elements are traversed at once. Other operations defined in `Traversable` are implemented using the `foreach` method. A comprehensive list of all operations can be found in [3].

Trait `Iterable` is a descendant of `Traversable`. It declares an abstract method `iterator` which returns an iterator used to traverse the elements. Iterators provide *pull-style* traversal - the next element is requested explicitly and not all elements have to be traversed. Method `foreach` in `Iterable` is implemented using the iterator[1]. Three other traits inherit from `Iterable` – `Set`, `Seq` and `Map`.

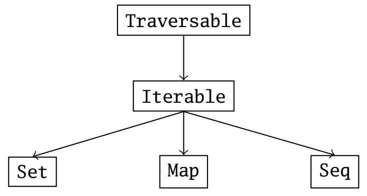

Figure 1. Collection base classes hierarchy

Trait `Seq` describes sequences – collections in which elements are assigned an index. These have an `apply` method taking an integer index and producing the corresponding element. Examples of sequences include `ArrayBuffer`, `List` and `Stream` classes. `Seq` trait defines operations specific to sequences such as `startsWith`, `indexWhere` and `reverse`.

Trait `Set` denotes `Iterable` collections which contain no duplicate elements. It defines abstract methods for adding and removing elements from the set, and checking if an element is contained in the set. It contains implementations for set operations like union, intersection and difference.

`Maps` are `Iterable` collections of pairs of keys and values. They define abstract methods for adding and removing entries to the map, and the `get` method used to lookup values associated with keys.

The collection framework is in the package `scala.collection`, which contains subpackages `mutable` and `immutable`. All of the traits described above have corresponding versions in each of the three packages. Traits in a subpackage inherit those in the root collection package. Collections in the `mutable` package additionally define destructive operations which allow in-place modifications

[1] The converse is not possible.

on collections – for instance, sequences define the `update` method to change the element at the specified index, and maps define the `put` method to associate a key to a new value. Collections in the `immutable` package cannot be modified – e.g. adding an element to the set produces a new set. These operations need not always copy the entire set. There exist efficient implementations for most immutable data structures [17].

Most collection operations are implemented in terms of element traversal, using the `foreach` method found in `Traversable` or iterators provided by `Iterable` collections. Some operations also return collections as their results. These use objects of type `Builder` to build the collections. Trait `Builder` is parametrized with the element type of the collection and the collection type it produces. It declares a method `+=` which is used to add elements to the builder. The method `result` is called after all the desired elements have been added to the builder and it returns a collection containing those elements. After calling `result` the contents of the builder are undefined and the builder cannot be used again before calling the `clear` method. Specific collection instances provide specific builders.

```
val withA = for {
  (n, s) <- names zip surnames
  if n startsWith "A"
} yield (n, s)
val groups = withA.groupBy(_._2)
println(groups.size)
for {
  (surname, pairs) <- groups
  if pairs.size <= 2
  (name, surname) <- pairs
} println(name + " " + surname)
```

Figure 2. Example program

We give a short example program to show how the collection framework is used. Assume we have two sequences `names` and `surnames` which contain names and corresponding surnames. We want to print out the number of distinct surnames for names starting with 'A' and then print a list of all such names and surnames for which there exists at most one other name with the same surname. Code in figure 2 shows how to do this.

We've omitted how we obtained the actual sequences of names and surnames as this is not relevant for the example. We give a readable solution in terms of *for-comprehensions* that iterate over sequence of pairs of names and surnames obtained by calling `zip` and filter those which start with 'A'. We then group these pairs according to the surname (second element of the pair, which is referenced with `_._2`) by calling `groupBy` and print the number of distinct surnames. We then iterate over surname groups with 2 or less names and print them. The code in figure 2 is translated to code similar to the one shown in figure 3. Readers interested in exact rules of for-comprehensions are referred to [2].

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

Figure 3. Translated program

There are several approaches to adding parallel operations to the existing collections framework. The approach taken in Data Parallel Haskell is to define a new set of methods for parallel operations and give them separate names (e.g. names of their sequential counterparts suffixed with 'P') [24]. Method calls in existing programs have to be modified to use parallel operations with the same semantics. This clutters the namespace with new names and the new names cannot be used in existing for-comprehensions.

A different approach is to have parallel operations implemented in separate classes. The separate class approach solves the issues above as methods for parallel operations can have the same names as their sequential variants. Clients can thus only invoke parallel operations if their data is in a parallel collection class. For this reason 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 to sequential implementations. Furthermore, we define a separate hierarchy of parallel sequences, maps and sets, and have them inherit regular sequence, map and set traits.

3. Adaptive work stealing

When using multiple processors load-balancing techniques are required to divide work. 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 must be created and assigned a subset of elements to work on. However, to create and initialize a thread 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 [8]. 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.

There are many load-balancing techniques previously proposed [5] [6] [7] [11]. 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 stealing [8] [10] [11] is a lightweight and efficient load-balancing technique. In work stealing, work is divided into tasks and distributed among processors. Each processor maintains a task queue. Once a processor finishes with one task, it dequeues the next task from the queue. If the queue becomes empty, the processor tries to steal a task from another processor's queue.

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 [8]. Still, for work stealing to be effective work must be partitioned into tasks of a small enough granularity, which can lead to overheads if there are too many tasks. As stated earlier, most of the operations in parallel collections can be implemented in terms of a divide and conquer scheme, so we use computation trees to show the order in which computations are performed, as shown in figure 4.

Uniformly sized task approach in theory guarantees that the greatest amount of time that the processors stay idle is equal to the time it takes to process one task, assuming uniform amount of

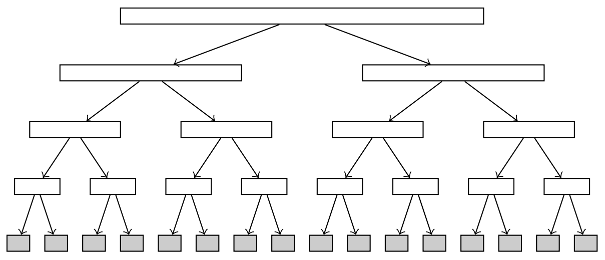

Figure 4. Fine-grained splitting

work per element. This can happen if all the processors finish with their last tasks at the time when there is one more task remaining. If the time needed to do the work sequentially is T, the number of processors is P 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.

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

What this simple analysis does not take into account is the overhead incurred by creating many tasks. In practice, fewer tasks can lead to significantly higher performance. But fewer tasks can also lead to poorer load-balancing as stated before. The technique we've used to solve these two issues is exponential task splitting, inspired by [12]. The basic idea is the following – if some worker thread completed its work, and there are still more tasks in its queue, then it means other workers are preoccupied with work of their own, so the worker thread could try to do more work with the next task. The heuristic used is to double the amount of work done next time. If the worker thread hasn't got more tasks in its queue, then it may steal tasks from other queues. 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 what task gets stolen depends on the order tasks were pushed to the queue (forked) – one can be selective about it.

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 in figure 5.

The worst case scenario that can happen with the above approach is that a processor gets assigned a biggest task it has processed so far at the exact moment when all other processors have

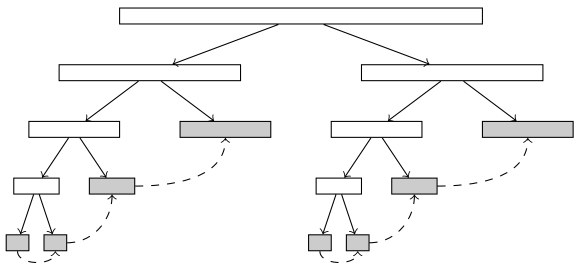

Figure 5. Exponential splitting

finished their work. This task has to come from the processor's own queue and not be stolen, otherwise it will be split into smaller tasks which will be pushed on its queue, enabling the other processors to steal tasks and not be idle. So, the task will be processed sequentially. 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 inequality 2 then gives a bound on the worst case speedup, assuming $P \ll N$ (the task size is fine-grained enough).

$$speedup = \frac{T}{T_P} = \frac{P+1}{2} \quad (2)$$

This estimate says that our task scheduling scheme never results in execution time more than twice as greater than the theoretical lower limit, given that the biggest number of tasks potentially generated is $N \gg P$.

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've 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 4. 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. 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 parallel operations into groups and describe how operations in different groups are implemented. Finally, we describe implementations of several concrete classes in our framework. The code examples we show are simplified with respect to actual code for purposes of clarity and brevity[2].

[2] Variance and bounds annotations have been omitted, as well as implicit parameters. Only crucial classes in the hierarchy are discussed. Source code can be obtained at `http://lampsvn.epfl.ch/svn-repos/scala/scala/trunk/src/library/scala/collection/parallel/`.

4.1 Initial approach

For the benefit of easy definition of new collection classes and easier maintenance we want to define most operations in terms of a few abstract methods. 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 – it influences the choice of the underlying data structure.

So, we can define one required abstract operation like this:

```
def split: Seq[ParIterable[T]]
```

where `Seq[ParIterable[T]]` is the return type of the method and denotes a sequence of objects of type `ParIterable[T]`, which represents parallel collections. There is more than one way to implement `split`. A trivial approach of copying elements is not very efficient. A more refined approach is to produce several views which iterate over parts of the collection. Certain data structures such as balanced trees and binomial heaps have the property that they can be efficiently split.

While `split` allows assigning collection subsets to different processors, there are operations that return collections as their result (e.g. `map`). Parts of the collection produced by different tasks should be combined together into the final result. It is not clear how to split an arbitrary collection, so we introduce:

```
def combine(c: ParIterable[T]): ParIterable[T]
```

This method returns a collection of type `ParIterable[T]` which contains elements of both collections. A trivial implementation of copying the elements of the receiver and `c` into a new collection is unacceptable in most cases. It is more efficient to implement `combine` lazily and copy the elements when the final collection needs to be evaluated. One way to evaluate lazily is to simply chain results of leaf computations together and return them wrapped within the collection. Once the final collection is needed, it is allocated and elements are copied. We show later how to do this for parallel arrays and parallel hash maps.

One might argue that the resulting collection like an array could be allocated prior to operation invocation, and leaf computations could simply copy the elements. This is true for methods such as `map` where the result size is known a priori, but not for `filter` where result size depends on the predicate. It is possible to invoke the predicate twice – once to count the elements and once more to copy them. This has several problems. First, the predicate may not be pure, so invoking it twice may change semantics[3]. Second, invoking twice might be more expensive than copying twice. Third, while copying is applicable to arrays, it is unclear how to apply it to immutable collections.

Another implementation approach is to use the properties of data structures at hand to efficiently merge operations. Trees, skiplists and certain types of heaps are particularly suitable for this [13] [14].

Once the subsets of the collection have been assigned to different processors, method `seq` can be invoked to process the sequentially. We will show that these methods are specific enough to allow implementation of all other parallel operations in our framework.

4.2 Further refinement

The *split-combine* approach allows implementation of collection operations in terms of abstract methods, but has several downsides.

[3] Although operators with non-disjoint side-effects passed to parallel operations as arguments are subject to data races, they still could be synchronized.

One issue arises with the `combine` method. If implemented by copying or some fast operation inherent to the underlying data structure, then such a design works well. However, if it is done lazily, the `combine` method approach requires either a new class that holds the unevaluated version of the final data structure, or embedding the logic for lazy evaluation into the main collection class. The former requires another collection class, while the latter makes the code less comprehensible.

When multiple classes are defined for the same collection, proper semantics for their methods become unclear. For instance, it is unclear what calling `split` on a sequential view of the parallel collection should do, since it is not required. In the same way, if we use a separate class for lazy combining, its `split` and `seq` methods are not invoked.

Problems described above cause a lot of boilerplate when implementing a collection. Abstract methods are not required in all the contexts – `combine` is not invoked while splitting collections subsets to different processors. Similarly, `split` method is never invoked while reassembling results.

We now present a solution that addresses these problems. Following the *split-combine* idea proposed earlier, we define two new abstractions – splitting iterators (or splitters) and combiners. We first describe the concept of splitters.

A splitter is an iterator with standard methods such as `next` and `hasNext` used to iterate elements of the collection. It has an additional method `split` which returns a sequence of splitters that iterate over disjunct subsets of the collection. The original iterator becomes invalidated after calling `split`.

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

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. If the splitter is empty or has only one element, then the `split` method returns a sequence with the receiver. In other cases, the partition does not contain empty splitters and has at least two splitters. Implementations partition the elements into subsets in a ratio that depends on the collection splitter iterates. However, the number of subsets in the partition should not be very large and their sizes should not differ greatly, otherwise load-balancing mechanism described earlier could be compromised. 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.

One example is a splitter over a collection called `ParArray` – splitting it yields two splitters, one iterating over the elements in the first half of the array and the other iterating elements in the second half. This is implemented by storing the index bounds inside the splitter.

Next, we describe combiners, a generalization of the builders described in an earlier section. Each parallel collection provides a separate combiner, just as regular collections provide a builder implementation. While a builder can be added elements and produce the collection with the `result` method, a combiner has a method `combine` that takes another combiner and produces a combiner that contains the union of their elements. Semantics of `combine` define both builders to become invalidated after its invocation.

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

A combiner takes two type parameters `Elem` and `To` which denote the element type and the type of the resulting collection, respectively. A combiner for the `ParArray` holds array chunks in a singly-linked list – `combine` links the internal lists of two combiners together. Once `result` is invoked, the number of elements contained in these array chunks is known, so an array is allocated and elements are copied in parallel into it using fast array copy operations.

Having defined splitters and combiners, we now describe the base trait for all parallel collections – `ParIterable`. It has the same operations like the `Iterable` trait of the Scala collection framework and methods specific to parallel iterable collections. `ParIterable` has an abstract method `parallelIterator` which returns a splitter and an abstract method `newCombiner` which returns the combiner for the collection. All operations are implemented in terms of splitters and combiners.

Certain collection operations need to pass information between processors, as we'll show later. Each parallel operation invocation creates an object of type `Signalling` for this purpose. A splitter is assigned a `Signalling` object and its children obtain the same `Signalling` object when it is split.

Each collection operation is implemented in a different task inner class of `ParIterable`. We use the *cake pattern* [18] to implement task scheduling logic in a different layer which defines an abstract type `Task`. Tasks for different operations inherit this abstract task, defining leaf computations, how data is split and results are combined. Such a design allows swapping scheduling implementation.

Finally, the method used to calculate the threshold used for adaptive work stealing is called `threshold` and simply returns an integer value according to equation 3. Specific collection classes may choose to override it if necessary.

Subtraits `ParSeq`, `ParMap` and `ParSet` define parallel sequences, maps and sets respectively. We do not describe them here in detail, but refer readers to the accompanying source code.

4.3 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 the Scala collection framework is the `foreach` method [3].

```
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[4]. 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 section 3. To split the elements of the collection into subsets, it invokes the `split` method of its splitter. 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
```

[4] 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` does not hold is found. Parallel collections have to communicate that the computation may stop. The `Signalling` object mentioned earlier allows tasks 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.

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` 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]. 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` and `splitAt` 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, `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` and `span`.

Method `psplit` of `PreciseSplitter` for parallel sequences is more general than `split`. It allows splitting the sequence into arbitrary subsequences. 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`.

4.4 Parallel array

A collection that's used often is an array. `ParArray` found in the Scala parallel collection framework stores the elements in an underlying array. It is a parallel sequence and extends the `ParSeq` trait.

Method `split` is implemented to return two splitters with different bounds pointing to the same underlying array. This makes `split` an $O(1)$ method in terms of the size of the array. Method `psplit` is implemented similarly.

`ParArray` combiner internally maintains a list of array chunks. Parallel array combiners are combined simply by concatenating their lists of arrays. Once the root task in the computation tree finishes, the size of the resulting array is known. The array is allocated and elements are copied into it. Most platforms support fast array copying operations. Furthermore, copying can be parallelized as well, as is the case with `ParArray`. 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.

As stated earlier, methods producing parallel arrays that know their sizes in advance are optimized to allocate an array and work on it directly. These methods do not use lazy building schemes described above and avoid copying the elements twice. To avoid copying altogether, a data structure such as a *rope* can be used to provide efficient splitting and concatenation [15].

4.5 Parallel hash trie

Parallel collections rely on operations that efficiently split elements into subsets and merge them back into collections. For an arbitrary collection type it is not obvious how to do so efficiently. For instance, merging two hash tables can be done in linear time, which could be unacceptable for large hash tables. Another problem with this is that such construction cannot be parallelized efficiently. We've tried using concurrent hash maps which allow concurrent construction by several different processes simultaneously, but this often lead to poor performance.

Inspired by hash trees we've implemented an efficient alternative to hash tables we call a parallel hash trie. A regular hash trie works as follows. It constructs a root hash table of 2^n elements which holds key/value pairs, where n is typically 5. Adding a key/-value pair to the hash trie amounts to computing the hash code of the key, taking first n bits of the hash code and using them as an

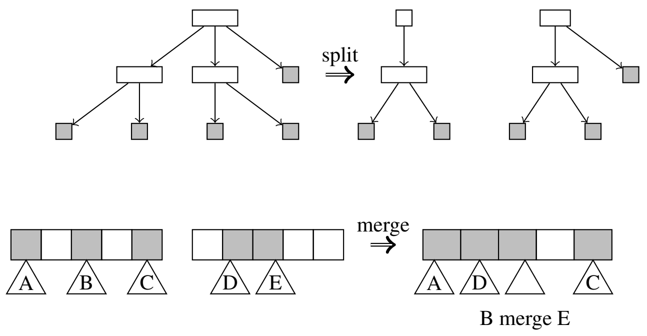

Figure 6. Hash trie operations

index in the array in which we place the key/value pair. In case of a collision we simply create a new array and put it as a subtrie in the corresponding entry in the root hash table. The key/value pairs which collide are put into the new array to indices that correspond to the next n bits of their hash codes, and this is repeated recursively as long as there are collisions. The resulting data structure forms very shallow tree, so only a few hops are required to find the correct element. A further optimisation is to use a bitmap in each node of the tree to denote which hashcodes are used in the node. Hash tries are described in detail in [20]. We've found hash tries to be comparable in performance to hash tables, providing faster iteration and slightly slower construction.

Advantages of hash tries are not only their low space overhead and good cache-locality, but also the fact that operations on them can be easily parallelized. Each parallel hash trie iterator has a reference to a hash trie data structure. To implement `split` we simply take the root hash table and put half of the subtries into a new hash table and the other half in the other new hash table. We obtain two hash tries and assign each of them to a new iterator. This is shown in figure 6 above, where the gray elements denote the actual key/value pairs. Such a split operation is both straightforward and cheap. Since parallel hash tries are used to implement maps and sets, and not sequences, there is no need to implement the `psplit` method.

Combiners can be implemented to also internally contain hash tries. To implement the combine method for combiners, one needs to merge their internal hash tries. Merging the hash tries is illustrated in figure 6. For simplicity, the hash trie nodes are shown to contain only five entries. The elements in the root hash table are copied from either one hash table or the other for, unless there is a collision, as is the case with subtries B and E in the figure. Subtries that collide are recursively merged and the result is put in the root hash table of the resulting hash trie. This technique turned out to be much more efficient than sequentially building a hash trie or even an ordinary hash table. However, in a typical invocation of a parallel operation, combine methods of combiners are invoked more than once (see figure 5). If the amount of work done per collection element is big enough, then merging cost may be acceptable. In particular, for the least possible amount of work per element, we have observed slowdowns of up to 6 times compared to sequential execution. Merging can be done in parallel. Whenever two subtries collide, we can spawn a new task to merge the colliding tries.

We obtained much better performance using a lazy evaluation approach – by postponing the actual evaluation of the hash trie until the final hash trie is requested, and at that point using its tree properties to construct it in parallel. Our combiners do not contain hash tries. Instead, a parallel hash trie combiner contains an array of 32 buckets, each holding elements with the same 5 bit hashcode prefix[5]. Buckets are implemented as unrolled linked lists – a list of concatenated array chunks which are more space-efficient, cache-local and less expensive to add elements to. Adding an element starts by computing its hashcode and taking its prefix to find the appropriate bucket. It is then appended to an unrolled list – 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. 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.

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've 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've 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 – having a better cache coherence than a single processor that builds the entire trie and inserts new elements all throughout it.

4.6 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))
```

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.

4.7 Parallel views

Assume that the user wants to increase numbers in some collection `c` by 10, filter positive numbers in the first half of a collection and then sum these together. Using operations provided by Scala collections, he could write something like this:

```
c.map(_ + 10).take(c.size / 2).filter(_ > 0)
  .reduce(_ + _)
```

Even with all the operations parallelized, there is an inherent performance problem above, since each of the operations produces a new collection. The Scala collection framework provides a special type of collections called *views*. A view is a wrapper around another collection that can be used to traverse the elements of the original collection in some way. For instance, a `Filtered` view will only iterate over the elements of the original collection which satisfy a given predicate, while a `Mapped` view will iterate over all of the elements, but will apply the specified mapping function to each element before processing it. Invoking any of the methods that produce collections on a view will results in creating a new view. It is possible to stack views – each view holds a reference to the view it was created from. If the user wants to produce a concrete collection from the elements of the view at some point, he can do so by invoking the `force` method.

In the above example, calling `view` on the collection `c` and then all the subsequent methods would produce wrapper views on top of each other, until `reduce` gets invoked. Method `reduce` does not produce a collection, so the elements of the view would be traversed to produce a concrete result.

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 iterators 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.

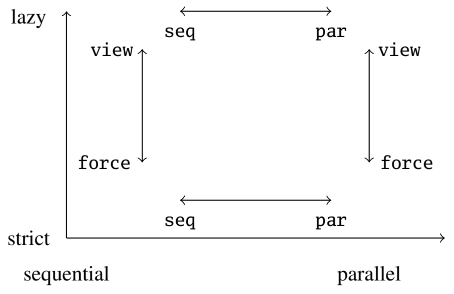

Figure 7. Strict-lazy and parallel-sequential conversions

Since regular collections also have views, the entire collection framework now provides a nice way to switch between strict and lazy collections on one axis, and sequential and parallel implementation on the other. Methods `par` and `seq` allow the collection to switch from a sequential to a parallel version and vice versa, respectively. Method `view` produces a view of a collection, while the method `force` produces a strict collection from a view. This is illustrated in figure 7.

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. The benchmarks shown were performed for parallel arrays and parallel hash tries on a machine with 4 Dual-Core AMD

[5] We could have had buckets hold elements with n bits in general, meaning combiners would have to hold 2^n buckets, but we've found that 5 bits work well in practice.

Opteron 2.8 MHz processor with hyper-threading, and are shown in figure 9. The number of processors used is displayed at the horizontal axis, the time in milliseconds needed is on the vertical axis. All tests were performed for large collections, and the size of the collection is shown for each benchmark. Each method was invoked few hundred times, and the time was measured for this batch of invocations – this was repeated until this value stabilized[6]. Each benchmark was made on a separate JVM invocation.

```
class Matrix[T: Numeric: Manifest](n: Int) {
  val array = new Array[T](n * n)

  def apply(y: Int, x: Int) = {
    array(y * n + x)
  }

  def update(y: Int, x: Int, elem: T) {
    array(y * n + x) = elem
  }

  def *(b: Matrix[T]) = {
    val m = new Matrix[T](n)
    m.setProd(this, b)
    m
  }

  def setProd(a: Matrix[T], b: Matrix[T]) {
    for (i <- (0 until n*n).par)
      this(i/n, i%n) = prod(a, b, i/n, i%n);
  }

  private def prod(a: Matrix[T],
    b: Matrix[T], y: Int, x: Int): T =
  {
    var sum = zero
    for (i <- 0 until n) {
      sum += a(y, i) * b(i, x)
    }
    sum
  }
}
```

Figure 8. Matrix multiplication

We've used computationally cheap operators for the benchmarks. For instance, the argument function for the `foreach` method simply changes a field in the object assigned to a position in the array. The function for method `map` simply returns the argument, and the predicate for `find` simply compares two numbers. By showing that parallel collections exhibit good performance for such fine-grained operators compared to which per element processing is high, we show that they can be applied to operators that are computationally more expensive and for which per element processing overhead is negligible, as shown in the benchmark in which integer primality is checked for each element.

The entry *Sequential* denotes benchmarks for sequential loops implementing the operation in question. All operations in the Java parallel array (*extra166*) are specialized since the underlying structure is an array. Operations in our framework are implemented without any specific knowledge about the underlying data structures which leads to more indirections and other overheads. This is why our parallel array is slightly outperformed in some tests. The entry *HashMap* denotes regular flat hash tables based on linear hashing. If the per-element amount of work is increased, data

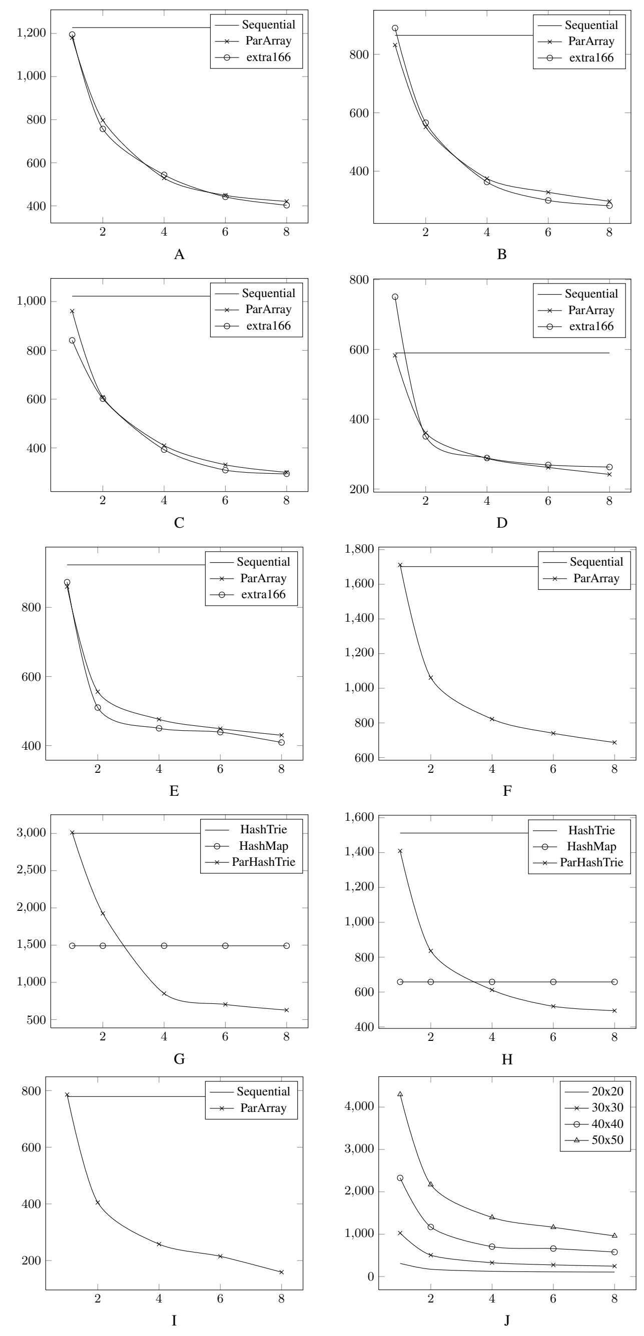

Figure 9. Benchmarks: (A) `ParArray.foreach`, 200k; (B) `ParArray.reduce`, 200k; (C) `ParArray.find`, 200k; (D) `ParArray.filter`, 100k; (E) `ParArray.map`, 100k; (F) `ParArray.flatMap`, 10k; (G) `ParHashTrie.reduce`, 50k; (H) `ParHashTrie.map`, 40k; (I) `ParArray.foreach`, primality, 200; (J) matrix multiplication

[6] The reason for this kind of benchmarking lies in the way the JVM works. Due to in-place compilations, inlining and other optimizations, the JVM goes through a heat-up phase after which the figures get more reliable.

structure handling cost becomes negligible and parallel hash tries outperform hash tables even for two processors.

We should comment on the results of the `filter` benchmark. Java's parallel array first counts the number of elements satisfying the predicate, then allocates the array and copies the elements. Our parallel array assembles the results as it applies the predicate and copies the elements into the array afterwards using fast array copy operations. When using only one processor the entire array is processed at once, so the combiner contains only one chunk – no copying is required in this case, hence the reason for its high performance in that particular benchmark.

The `map` benchmark for parallel arrays uses the optimized version of the method which allocates the array and avoids copying, since the number of elements is known in advance. Benchmark for `flatMap` includes only comparison with the sequential variant, as there is currently no corresponding method in other implementations.

A larger example is shown in figure 8, where a matrix class uses parallel ranges to implement trivial matrix multiplication. The matrix internally holds an array of object of type `T`, which have a `Numeric` context bound. This means that the matrix is generic, but can only hold elements that are numbers, so it is applicable to `Int`, `Double`, `BigDecimal`, etc. The `apply` and `update` methods get and set matrix entries, respectively. Method `*` allocates a new empty matrix and assigns it the product of the receiver and the argument matrix. This method makes the syntax for matrix multiplication transparent with respect to ordinary multiplication. Method `setProd` is where the magic happens. We traverse all of the elements of the array as with ordinary ranges, but we invoke `par` to make the operation parallel. Each element in the resulting matrix is then computed in parallel within method `prod`. Figure 9 shows benchmark results.

6. Conclusion

We've provided parallel implementations for a wide range of operations found in the Scala collection library. We've done so by introducing two simple divide and conquer abstractions called *splitters* and *combiners* needed to implement most operations. These abstractions have been successfuly blended into the existing Scala collection framework. All parallel collections integrate smoothly and enchmarks show good performance.

One other task we plan to address in the future is to apply specialization to parallel arrays in order to achieve better performance for arrays of primitive types [22]. Currently, parallel arrays hold boxed objects instead of primitive integers and floats, which not only brings in boxing/unboxing overhead, but also means that these objects are distributed throughout the heap. This leads to cache misses and hinders performance.

In the future, we plan to extend our framework with new collection types. Also, future research includes finding and integrating new operations convenient for parallel applications. One example is to augment parallel collections with operations which allow expressing data dependencies in processing the elements of the collection. The challenge is how to do this in an expressive and efficient manner, which is applicable to a plethora of programs and algorithms. These and other tasks will be addressed in near future.

References

[1] Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean McDirmid, Stephane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Lex Spoon, Matthias Zenger, *An Overview of the Scala Programming Language*, Technical Report LAMP-REPORT-2006-001, EPFL, 2006.

[2] Martin Odersky, Lex Spoon, Bill Venners, *Programming in Scala*, Artima Press, 2008.

[3] Martin Odersky, *Scala 2.8 collections*. 2009.

[4] Stephen Toub, *Patterns of Parallel Programming*, Microsoft Corporation, 2010.

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

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

[7] Susan Flynn Hummel, Edith Schonberg, Lawrence E. Flynn, *Factoring: A Method for Schedulling Parallel Loops*, Communications of the ACM, 1992.

[8] Doug Lea, *A Java Fork/Join Framework*, 2000.

[9] Doug Lea's Home Page, *http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/src/extra166y/*

[10] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, Yuli Zhou, *Cilk: An Efficient Multithreaded Runtime System*, SIGPLAN Not., 1995.

[11] Robert D. Blumofe, Charles E. Leiserson, *Scheduling Multithreaded Computations by Work Stealing*, Proceedings 35th IEEE Conference on Foundations of Computer Science, 1994.

[12] Guojing Cong, Sreedhar Kodali, Sriram Krishnamoorthy, Doug Lea, Vijay Saraswat, Tong 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] Hans-J. Boehm, Russ Atkinson, Michael Plass, *Ropes: An Alternative to Strings*, Software: Practice and Experience, 1995.

[16] Nir Shavit, Maurice Herlihy, *The Art of Multiprocessor Programming*, Morgan Kaufmann, 2008.

[17] Chris Okasaki, *Purely Functional Data Structures*, Cambridge University Press, 1999.

[18] Martin Odersky, Matthias Zenger, *Scalable Component Abstractions*, Conference on Object Oriented Programming Systems Languages and Applications, 2005.

[19] Martin Odersky, Adriaan Moors, *Fighting Bitrot with Types*, Foundations of Software Technology and Theoretical Computer Science, 2009.

[20] Phil Bagwell, *Ideal Hash Trees*, 2002.

[21] Groovy Parallel Systems, *http://gpars.codehaus.org/*

[22] 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.

[23] Mark Moir, Nir Shavit, *Concurrent data structures*, Handbook of Data Structures and Applications, Chapman and Hall, 2007.

[24] Simon Peyton Jones, Roman Leshchinskiy, Gabriele Keller, Manuel M. T. Chakravarty, *Harnessing the Multicores: Nested Data Parallelism in Haskell*, Foundations of Software Technology and Theoretical Computer Science, 2008.