A Generic Parallel Collection Framework

Aleksandar Prokopec, Phil Bagwell, Tiark Rompf, and Martin Odersky

École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland

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 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

With the arrival of multicore architectures, parallel programming is becoming more widespread. 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 [18]. This paper describes how parallel operations can be implemented with two abstractions – splitting and combining.

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

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6853, Part II, pp. 136–147, 2011.

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 [20] [17]. 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. Sect. 2 gives an overview of the Scala collection framework. Sect. 3 describes adaptive work stealing. Sect. 4 describes the design and several concrete parallel collections. Sect. 5 presents experimental results. Sect. 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 [3]. Readers interested to learn more are referred to textbooks on Scala [4].

Its features of interest for this paper are higher-order functions and traits. These language features are not a prerequisite for parallel collections – they serve as a convenience. Our approach can be applied to other general purpose languages as well. Functions are first-class objects – they can be assigned to variables or specified as arguments to other functions. For instance, to find the first even number in the list of integers `lst`, we write: `lst.find(_ % 2 == 0)`. 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.

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 [5]. Some operations (`filter`, `take` or `map`) produce collections as results. They 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. Each collection provides a specific builder.

We give a short example program (Fig. 1). 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. The example uses *for-comprehensions* [4] 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.

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 [14]. Method calls in existing programs have to be modified to use corresponding parallel operations. 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))
```

Fig. 1. Example program

3 Adaptive Work Stealing

When using multiple processors load-balancing techniques are required. 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 [8] [2]. We use the Java fork-join framework to schedule tasks [1]. For effectiveness, work must be partitioned into tasks that are small enough, which leads to overheads if there are too many tasks.

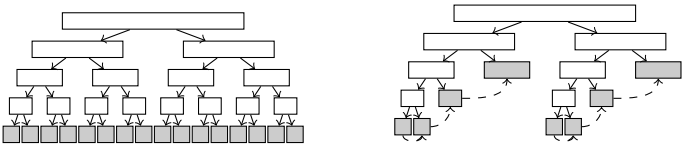

Fig. 2. Fine-grained and exponential task splitting

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.

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

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 [9]. 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. 2). 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. Stolen tasks are split until reaching threshold size – the need to steal indicates that other workers may be short on tasks too.

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.

4 Design and Implementation

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. 1) in terms of a few abstractions. 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.

```
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]
}
```

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. Combining results from different tasks occurs more than once during a parallel operation in a tree-like manner (Fig. 2).

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 Parallel Array

Arrays are mutable sequences – class `ParArray` stores the elements in an array.

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.

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[1] 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). 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))))
}
```

4.3 Parallel Rope

To avoid the copying step altogether, a data structure such as a *rope* is used to provide efficient splitting and concatenation [10]. 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.

Combiners may use the append operation for `+=`, but this results in unbalanced ropes [10]. 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 [10].

4.4 Parallel Hash Table

Associative containers implemented as hash tables guarantee $O(1)$ access with high probability. There is plenty of literature available on concurrent hash tables [13]. 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`).

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

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. Given a set of elements, we want to construct a hash table using multiple processors. Subsets of elements are assigned to different processors and must 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.

```
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 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.

Method `result` is called in the computation root – 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. 3). 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.

4.5 Parallel Hash Trie

A hash trie is an immutable map or set implementation with efficient element lookups and updates ($O(\log_{32} n)$) [11]. Updates do not modify existing tries, but create new versions which share parts of the data structure. Hash tries consist

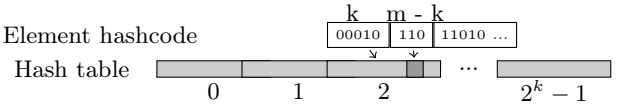

Fig. 3. Hash code mapping

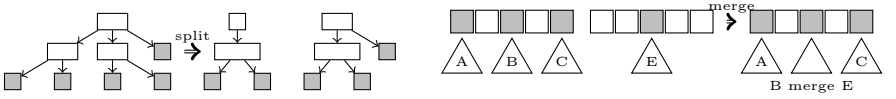

Fig. 4. Hash trie operations

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. To ensure low space consumption, each node has a 2^k bitmap to index its table (typically $k = 5$) [11]. 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.

Combiners can contain hash tries. Method `combine` could merge the hash tries (figure 4). 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 5. Although it requires less work, recursive merging scales linearly with the trie size. 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.

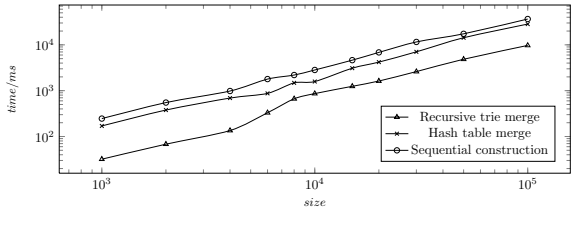

Fig. 5. Recursive trie merge vs. Sequential construction

4.6 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`).

5 Experimental Results

To measure performance, we follow established measurement methodologies [19]. 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 average time is recorded. We compare against sequentially inserting n elements into a `java.util.HashMap`.

Fig. 6 shows a performance drop due to contention. 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 7 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. 7 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[2] 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.

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

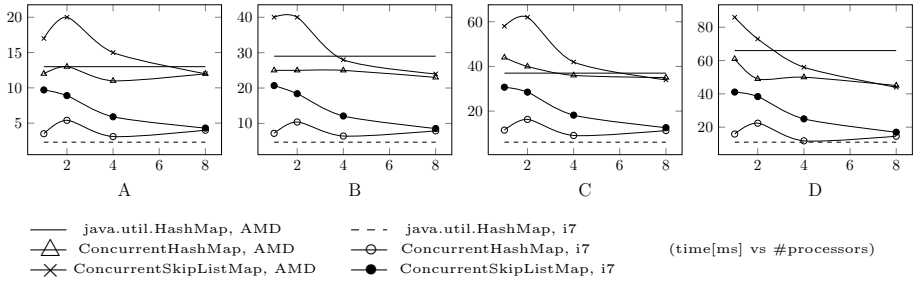

Fig. 6. Concurrent insertion, total elements: (A) 50k; (B) 100k; (C) 150k; (D) 200k

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 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 [13]. 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 [18].

.NET langugages support patterns such as parallel looping, aggregations and the map/reduce pattern [6]. .NET Parallel LINQ provides parallelized implementations query operators. On the JVM, one example of a data structure with parallel operations is the Java `ParallelArray` [7], 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 [14].

Some languages recognized the need for catenable data structures. Fortress introduces conc-lists, tree-like lists with efficient concatenation [17]. 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 [15]. STAPL for C++ has a similar approach – they provide thread-safe concurrent objects and iterators that can be split [16]. The STAPL project also implements distributed containers. Data structure construction is achieved by concurrent insertion, which requires synchronization.

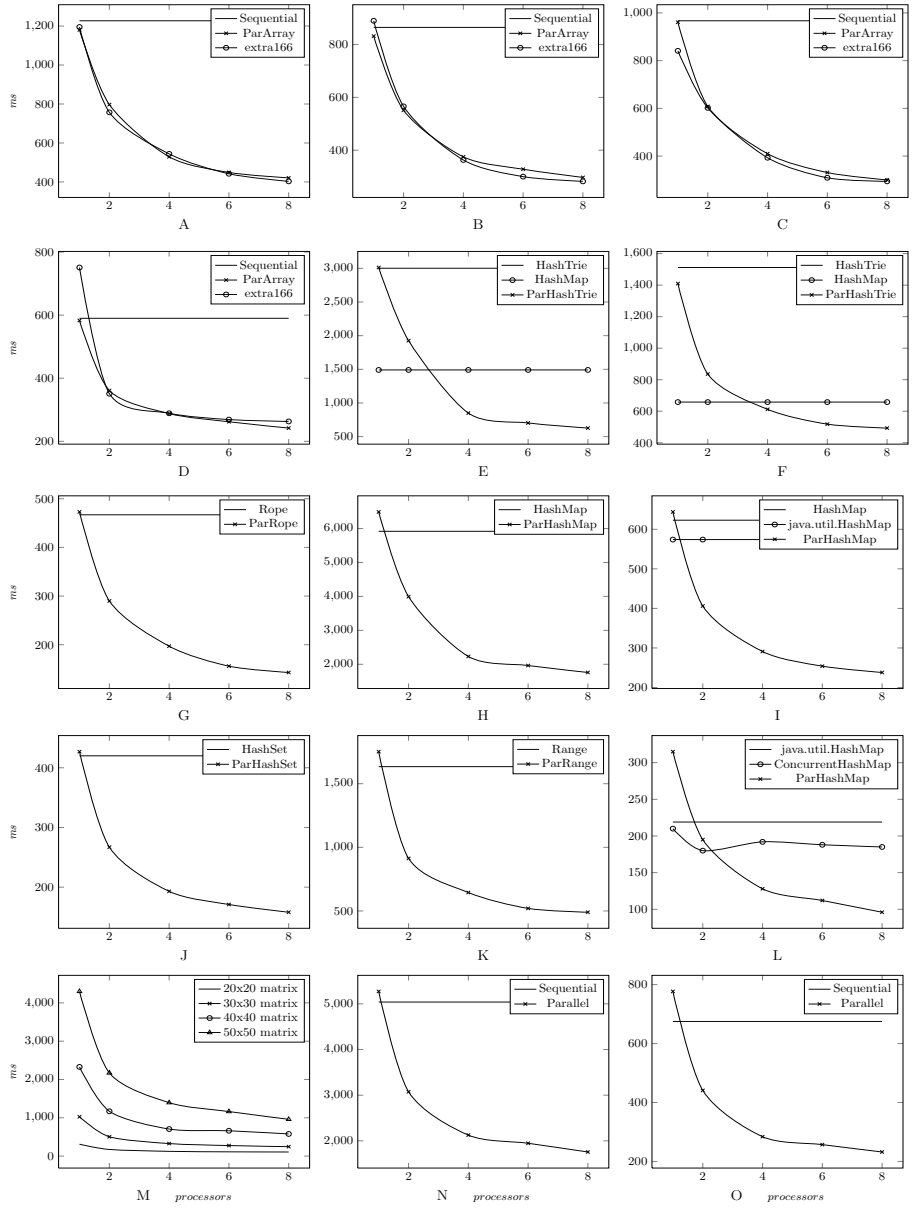

Fig. 7. Benchmarks (running time [ms] vs number of processors): (A) `ParArray.foreach`, 200k; (B) `ParArray.reduce`, 200k; (C) `ParArray.find`, 200k; (D) `ParArray.filter`, 100k; (E) `ParHashTrie.reduce`, 50k; (F) `ParHashTrie.map`, 40k; (G) `ParRope.map`, 50k; (H) `ParHashMap.reduce`, 25k; (I) `ParHashMap.map`, 40k; (J) `ParHashSet.map`, 50k; (K) `ParRange.map`, 10k; (L) `ParHashMap.map(id)`, 200k; (M) Matrix multiplication; (N) Coder; (O) Grouping

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 [12].

References

1. Lea, D.: A Java Fork/Join Framework (2000)
2. Traore, D., Roch, J.-L., Maillard, N., Gautier, T., Bernard, J.: Deque-free work-optimal parallel STL algorithms. In: Proceedings of the 14th Euro-Par Conference (2008)
3. Odersky, M., et al.: An Overview of the Scala Programming Language. Technical Report LAMP-REPORT-2006-001, EPFL (2006)
4. Odersky, M., Spoon, L., Venners, B.: Programming in Scala. Artima Press (2008)
5. Odersky, M.: Scala 2.8 collections. EPFL (2009)
6. Toub, S.: Patterns of Parallel Programming. Microsoft Corporation (2010)
7. Doug Lea's, Home page, `http://gee.cs.oswego.edu/`
8. Blumofe, R.D., Leiserson, C.E.: Scheduling Multithreaded Computations by Work Stealing. In: 35th IEEE Conference on Foundations of Computer Science (1994)
9. Cong, G., Kodali, S., Krishnamoorthy, S., Lea, D., Saraswat, V., Wen, T.: Solving Large, Irregular Graph Problems Using Adaptive Work Stealing. In: Proceedings of the 2008 37th International Conference on Parallel Processing (2008)
10. Boehm, H.-J., Atkinson, R., Plass, M.: Ropes: An Alternative to Strings. Software: Practice and Experience (1995)
11. Bagwell, P.: Ideal Hash Trees (2002)
12. Dragos, I., Odersky, M.: Compiling Generics Through User-Directed Type Specialization. In: Fourth ECOOP Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (2009)
13. Moir, M., Shavit, N.: Concurrent data structures. Handbook of Data Structures and Applications. Chapman and Hall, Boca Raton (2007)
14. Jones, S.P., Leshchinskiy, R., Keller, G., Chakravarty, M.M.T.: Harnessing the Multicores: Nested Data Parallelism in Haskell. Foundations of Software Technology and Theoretical Computer Science (2008)
15. Intel Thread Building Blocks: Tutorial (2010), `http://www.intel.com`
16. Buss, A., Harshvardhan, Papadopoulos, I., Tkachyshyn, O., Smith, T., Tanase, G., Thomas, N., Xu, X., Bianco, M., Amato, N.M., Rauchwerger, L.: STAPL: Standard Template Adaptive Parallel Library. In: Haifa Experimental Systems Conference (2010)
17. Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J.-W., Ryu, S., Steele Jr., G.L., Tobin-Hochstadt, S., et al.: The Fortress Language Specification (2008)
18. Steele Jr., G.L.: How to Think about Parallel Programming: Not! (2011), `http://www.infoq.com/presentations/Thinking-Parallel-Programming`
19. Georges, A., Buytaert, D., Eeckhout, L.: Statistically Rigorous Java Performance Evaluation. In: OOPSLA (2007)
20. Hinze, R., Paterson, R.: Finger Trees: A Simple General-purpose Data Structure. Journal of Functional Programming (2006)