Implicit Parallelism in a Functional Subset of Scala

Shams Imam Robert Cartwright Vivek Sarkar
(shams, cork, vsarkar)@rice.edu
Department of Computer Science,
Rice University.

Abstract

In this paper, we describe our approach to implicitly parallelizing programs written in a functional subset of Scala characterized by the single assignment rule. Our functional programming model is supported by a functional subset of Scala based on our past experiences with the Sisal language and the functional language level of DrJava. We automatically parallelize functional Scala programs by inserting *futures* and *accumulators*. We avoid the resource limitations that arise when tasks block on futures by executing *delimited continuations* on a *cooperative scheduler* with a fixed number of worker threads. The functional structure of programs written in our Scala subset ensures deadlock freedom since such programs cannot generate cyclic dependences among expressions/futures. Our functional programming model exposes abundant opportunities for safe implicit fine-grain parallelism at the expression level. Such expressions can be represented as future constructs that compute the value of the expression in parallel with the rest of the computation. On the other hand, introducing parallelism adds overhead to manage and schedule execution of the futures and to avoid blocking worker threads when the results of the future are required due to data dependence constraints. To achieve high performance in executing such applications on many-core processors, we must partition the computation into largely independent tasks (contrained by data dependence) with appropriate granularity that is neither too fine nor too coarse and we must must efficiently implement the representation and execution of futures. We currently rely on hand-written code and runtime techniques to determine the appropriate granularity for parallel tasks and we use *delimited continuations* to represent computations waiting on futures without blocking worker threads.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming

General Terms Design, Languages, Performance

Keywords Applicative Language, Futures, Implicit Parallelism, Cooperative Scheduling

1. Introduction

The Scala language is an attractive platform for implementing applications because it smoothly integrates the object-oriented and functional programming models. While past work has focused on opportunities for exploiting task and actor parallelism [5, 9] in imperative, object-oriented Scala programs, there has been less effort devoted to identifying and exploiting parallelism in Scala programs written in a functional style. (A notable exception is the Scala parallel collections library [12].) In this paper, we describe our approach to implicitly parallelizing programs written in a functional subset of Scala characterized by the single assignment rule. Our functional programming model is supported by a functional subset of Scala based on our past experiences with the Sisal language and the functional language level of DrJava. This functional subset excludes imperative statements (rebinding local variables or object fields) and instead focuses on Scala's support for explicit function definitions and expressions written in terms of primitive functional operations and the rich Scala collection of functional classes and traits. Our subset also includes restricted versions of arrays, tuples, for-comprehensions, case-classes, pattern-matching, and `Option` / `Either` classes.

Our approach to automatic parallelization is based on the judicious insertion of *futures* and *accumulators*. To achieve high performance in executing such applications on many-core processors, we must partition the computation into largely independent tasks (constrained by data dependence) with appropriate granularity that is neither too fine nor too coarse and we must must efficiently implement the representation and execution of futures. We avoid resource limitations arising from tasks blocked on futures by executing *delimited continuations* (semantically equivalent to blocked

tasks) on a *cooperative scheduler* with a fixed number of worker threads. On the other hand, introducing parallelism adds overhead to manage and schedule execution of the futures and to avoid blocking worker threads when the results of the future are required due to data dependence constraints. Our preliminary experimental results on a system with 12-cores showed a speedup of 6.6× for a Matrix multiplication computation, 11.4× for a Smith-Waterman dynamic programming computation, and 2.6× for a Quicksort computation. In each case, the parallel version was obtained by only inserting annotations/wrappers for futures and accumulators, without changing the logical structure of the original functional program.

The rest of the paper is organized as follows. Section 2 summarizes the Sisal and DrJava projects that had a major influence on the identification of the functional subset of Scala used in this paper. That subset is subsequently defined in Section 3. Section 4 introduces parallel constructs that can be inserted into functional Scala programs without disturbing their semantics, and Section 5 describes optimizations that can reduce the overheads that can arise from excessive parallelism. Section 6 then describes the implementation of our runtime system to support the parallelism and optimizations discussed in Sections 4 and 5. Section 7 describes our preliminary experimental results; Section 8 discusses related work; and Section 9 contains our conclusions and plans for ongoing/future work.

2. Background

In this section, we briefly summarize two past projects that have had a major influence on the identification of the functional subset of Scala used in this paper.

2.1 Sisal

Sisal (Streams and Iterations in a Single-Assignment Language) [3, 11] is a *single-assignment* programming language with *value-oriented semantics*; it does not permit memory-update operations or any other side effects. Since Sisal is a side-effect free language, subexpressions may be evaluated in any order without effect on computed results, provided all data dependences are satisfied. For example, there is no requirement that the left operand of an operator be evaluated before the right operand; in fact, they can both be evaluated in parallel. Language constructs are provided for conditional and iterative expressions which implicitly contain control dependences. Iterative expressions have a set of initial value definitions, a corresponding set of redefinitions used to define new values on every iteration, a loop control, and a result value defined in terms of the final (or intermediate) values of the names in the redefinitions. The keyword **old** is used to refer to the value of a name in the previous iteration to differentiate it from the value of the name in the current iteration. Sisal also offers a **for** construct for expressing parallel iterations with no loop-carried dependences.

```
1  function split(data:array[integer]
2                 returns array[integer],
3                         array[integer])
4    let pivot := data[1]
5    in
6      for elem in data
7      returns
8        array of elem when elem <= pivot
9        array of elem when elem > pivot
10     end for
11   end let
12 end function

14 function quicksort(data:array[integer]
15                    returns array[integer])
16   if array_size(data) > 1 then
17     let l, r := split(data)
18     in quicksort(l) || quicksort(r)
19     end let
20   else
21     data
22   end if
23 end function
```

Figure 1: Sisal implementation of quicksort

A unique aspect of the Sisal language is that it was designed to make functional programming accessible to mainstream imperative programmers. We will use the Quicksort example in Figure 1 to illustrate some of its key features:

- *Let expressions:* Since all imperative programmers are familiar with assignment statements, a **let** expression can simply be viewed as a sequence of assignment statements as in lines 4 and 17. There are two important differences between a let expression and a block of assignments statements, however: *a*) the single-assignment rule prohibits the same name (variable) from occurring more than once on the left-hand-side, and *b*) the let expression must always conclude with an **in** claus that specifies the return value of the expression.
- *Multi-arity expressions:* Expressions and function return values are permitted to have an arity that is > 1. For example, the call to the split() function in line 17 returns an expression with arity = 2; the two components of the return value correspond to the left and right subarrays created by function split(). Multi-arity expressions avoid the programming complexity of having to create box/tuple data structures to communicate multiple values between function calls and other expressions.
- *Types:* Type checking is performed at compile-time since Sisal is a strongly typed language. Sisal requires that complete types be provided for all function parameters and return values, as in lines 1–3 and lines 14–15. However, no types need to be specified for any local names (variables) within a function.

  The data types of Sisal include the basic scalar types: boolean, character, integer, real, double real. Data struc-

ture values can be records, arrays, unions, or streams. Arrays in Sisal are dynamically sized. (The `||` operation in line 18 denotes array concatenation.) Recursive data types are also permitted, e.g., a recursive record containing a union type can be used to implement any acyclic pointer data structure.

- *For expressions:* The **for** expression in lines 6–10 can be viewed as a generalized array comprehension. As shown in lines 8 and 9, the **returns clause** can be very powerful and can include guarded conditions analogous to "where" clauses in query languages.
- *Conditional expressions:* Most mainstream programmers are familiar with the "*condition* ? *true-value* : *false-value*" syntax for conditional expressions in C and Java. The same concept can be used more broadly in Sisal using the if-then-else expression, as in lines 16–22.

2.2 Functional subset of Java

We also have experience identifying a functional subset of Java to make functional programming accessible to beginning programmers. This subset is supported as the *functional language level* in DrJava, a pedagogic IDE for Java. The pedagogy underlying this functional subset teaches program design in Java by using a hierarchy of language levels [8]. The simplest language level focuses on functional programming with immutable data by making all fields and local variables `final` and forbidding identity testing (using the `==` operator) on object values. In this restricted language, it is easy to define composite algebraic data types such as lists and trees augmenting the base functional types like `int`, `double` and `boolean`. In addition, functions over these algebraic types can be defined using the interpreter and visitor design patterns. Programs written in the functional subset of Scheme can readily be expressed in this language.

As a result, executing functional Java programs is analogous to performing ordinary algebraic simplification, which is a much simpler process than general computation in Java. This functional subset of Java includes all of the core constructs of Java including interfaces, static fields, package declarations, exceptions including try-catch statement and throw statements, anonymous classes, and explicit casts. Of these features, anonymous classes are the most important because they enable methods to pass "functions" (behavior) as data. In Java, anonymous classes play the same role as lambda expressions in functional languages and can be replaced by Java 8 lambda expressions in the future. Explicit threads are forbidden so functional Java programs are strictly sequential.

In determining our functional subset of Scala, we first identified the Scala subset corresponding to the the functional language level of DrJava. Then we added a few additional Scala constructs supporting immutable arrays akin to familiar Sisal constructs to help accommodate most high performance computing applications. Finally, we imposed some restrictions on the resulting language to ensure programs could be executed in parallel without affecting their semantic behavior.

The functional language level of Java in DrJava has a strictly sequential semantics in which the order of expression evaluation is revealed by computations that generate and catch exceptions. In our functional subset of Scala, we eliminate `try-catch` blocks and `throw` statements and collapse all exceptions thrown by Scala library classes to a single non-specific error value. Hence, any program that throws an exception simply returns this unique error value. As a result, we can safely execute computations in parallel without specifying how multiple exceptions generated in parallel are reconciled. If an exception is generated during the execution of a program in our parallel subset, the result of the computation is the error value. We defer further work on exceptions in our functional subset as future work.

3. Functional Scala

Scala unifies the object-oriented and functional programming paradigms within a single coherent language. The language provides a comprehensive collection of functional and object-oriented constructs including support for imperative programming. Since we want to restrict our attention to immutable data to simplify introducing parallelism, we rely on a limited subset of the Scala language. In particular, we disallow imperative statements that violate the single assignment property for both object fields and local variables. The resulting functional programs expose abundant opportunity for safe implicit fine-grain parallelism at the expression level that our compiler and runtime can exploit.

Scala provides an expressive functional collections library consisting of the classes and traits in the `scala.collection.immutable` package. These immutable collections provide functional analogs of familiar imperative operations like insertion or removal by building and returning new collections–leaving the input collections unchanged (copy-on-write semantics). Scala's support for first-class functions and a rich immutable collections API also enables us to express computations that traverse collections and generate new ones using higher-order functional operations like `fold`, `map`, and `filter` instead of imperative loops. To prevent mutation operations, we restrict the use of collections to classes from the `scala.collection.immutable` package. Since Scala uses immutable collections by default, this restriction is not burdensome.

Figure 2 shows we can write elegant programs that conform to our restrictions. In fact, essentially any functional program expressible in Scheme or ML (excluding programs that use `call/cc`) can easily be written in our Scala subset. The Quicksort example, inspired by he SISAL example in Figure 1, uses immutable lists. Our code uses the head of the list as the partition element and the Scala `partition` operation to create the left and right fragments returned as tuples.

We will revisit the Quicksort example in subsequent sections of the paper as we extend our functional subset to optimize parallel computations.

Note that our functional subset presumes that recursion is used as the principal control structure (perhaps along with dynamic dispatch). In our `quicksort` example, the function `quicksort` generates abundant opportunities for parallelism by recursively calling itself. The `partition` operation, which is left abstract in our code, may include similar internal parallelism.

```
def quicksort[T](
  list: List[T]): List[T] = {
 if (list.isEmpty) {
  list
 } else {
  val x = list.head
  val xs = list.tail
  val (lP, rP) = xs partition(_ < x)
  val left = quicksort(lP)
  val right = quicksort(rP)
  left ++ (x :: right)
 }
}
```

Figure 2: Quicksort using immutable `Lists` and the built-in `partition` operation. The higher-order `partition` function accepts an anonymous function as an argument, the underscore operator allowing for a concise syntax.

In addition to processing immutable collection classes via higher order traversal operations and explicit recursion, our functional subset also includes support for immutable arrays. In contrast to conventional Scala, we disallow operations (such as array element assignment) that can mutate arrays. Array handles are exposed only after all elements have been initialized. This restriction is enforced by requiring the user to use the `tabulate` function to create new array instances. Figure 3 shows how we preserve a functional philosophy by creating an array using the `tabulate` function to initialize the array. Once created, an array can be used for indexed access to its elements or used in functional operations like filter, fold, etc. We are currently working on extending our subset to support initializing array elements using previously initialized array elements provided the iteration space for the array indices is traversed in a *well-defined* manner that guarantees each element is initialized exactly once. This extension conforms to the single assignment restriction since each array element is bound exactly once.

To ensure that the fields of a class are assigned exactly once, we limit the concrete classes defined in program text to `case` classes. Similarly, no expression of type `Mutable` or type `None` is permitted and equality testing may not be performed on values of function type. Immutable collections classes, arrays, and case classes enable us to rely on structural equality to reflect a simple semantic model where distinguishable copies do not exist. Since our functional programs must be deterministic, we also prohibit any explicit reference to classes that represent or create threads, e.g., `Thread`, `Runnable`, any class in `java.util.concurrent`, any class in a Java or Scala GUI library, any class in the RMI libary, *etc.*

```
def multiplySeq(
  a: Array2d, b: Array2d): Array2d = {
 val (T, R, C) =
  (a(0).length, a.length, b(0).length)
 val res = Array.tabulate(R, C) {
  (row, col) =>
   (0 until T).foldLeft(0.0) {
    (acc, i) =>
     acc + (a(row)(i) * b(i)(col))
   }
 }
 res
}
```

Figure 3: Matrix multiplication using `Arrays`. We use the collection API (`foldLeft` reduction operation) inside the `tabulate` operation to retain the functional nature despite using arrays.

4. Parallel Constructs

The lack of side-effects in functional programs makes it suitable for parallelism since the immutable semantics guarantees the absence of data races. In addition, all the data dependences are explicit and can be computed using a simple dataflow analysis. This presents the opportunity for implicit parallelism at a very fine level within program expressions. In this section, we discuss such implicit constructs that can be introduced to safely parallelize a functional program. Of course, these constructs can be introduced at a very fine level of granularity where the overhead of managing the parallelism offsets any performance benefits from running the code in parallel. In the next section (Section 5) we discuss how we can optimize the placement of implicit parallel constructs to ensure sufficient granularity to make parallel improve performance in typical situations.

4.1 Futures

One way to parallelize functional programs at the expression level is via the use of futures [7]. A future represents a single-producer multiple-consumer pattern and contains an immutable value that becomes available at a later point in program execution. Consumers who attempt to resolve the value of the future before it is available need to suspend, *i.e.*, wait for the data dependence to be satisfied. The value is usually computed asynchronously by a producer task thus providing parallelism with other independent expressions. When the producer task completes execution, it resolves the value of the future thus allowing the resumption of any previously suspended consumers. Consumers who force the value of the future after it has already been resolved can continue execution without being suspended. Figure 4 displays the

code for a possible parallelized version of quicksort from Figure 2 using futures. The future allows the left partition to be sorted in parallel with the right partition. When the results of the left and right fragments are required they are resolved (at line 13 and 14) before computing the resulting sorted list.

```
1  def quicksort[T](
2    list: List[T]): List[T] = {
3   if (list.isEmpty) {
4    list
5   } else {
6    val x = list.head
7    val xs = list.tail
8    val (lP, rP) = xs partition(_ < x)
9    val leftFuture = async quicksort(lP)
10   val rightFuture = async quicksort(rP)
11   leftFuture.resolve() ++
12     (x :: rightFuture.resolve())
13  }
14 }
```

Figure 4: Parallelized version of Quicksort from Figure 2 using futures. The `async` is used to create a future that is computed *async*hronously. The `resolve()` operation acts as a synchronization point and is used to resolve the value of the future.

Our view of futures is influenced by both Data-driven Futures [2, 17] in Habanero-Java (HJ) and the Scala Future API [6]. We treat futures as asynchronous tasks resolved by a specific producer task but also provide a means of creating Promise instances that are explicitly resolved by arbitrary producer tasks. Unlike Scala futures, our cooperative runtime (Section 6.1) ensures there is no thread-blocking involved when the future is resolved. This approach enables us to parallelize the user code with minimal changes and avoids the generation of *event-driven style* code via callbacks in our compiler plugin. It is important to note that we do not handle exceptions inside futures; if the user code wrapped by a future throws an exception the programs aborts with a runtime error by throwing the exception. We may later readdress our exception handling semantics in a manner similar to HJ, where the exception is re-thrown whenever consumers call the `get` operation. Note that supporting exceptions requires a protocol for combining exceptions thrown by tasks executing in parallel.

4.2 For Comprehensions

Our next construct for specifying implicit parallelism is a parallel `forall` loop that returns an array of the results computed by the loop iterations. In the implementation of this construct, the iterations of the `forall` loop can be executed in any order independent of the iteration index thus simplifying parallelism. As an example, the `tabulate` operation from Figure 3 can be converted to a parallel `forall` loop. The difference from traditional uses of parallel loops is that the result of the `forall` loop is an array whose elements have been initialized to the results of the individual iterations, these elements are computed in parallel. Figure 5 shows the simple change on line 4 that parallelizes the initialization of the two-dimensional array.

```
1  def multiplySeq(
2    a: Array2d, b: Array2d): Array2d = {
3   ...
4   val res: Array2d = forall(R, C) {
5    (row, col) =>
6     (0 until T).foldLeft(0.0) {
7       ...
8     }
9   }
10  ...
11 }
```

Figure 5: Matrix multiplication using `forall` loop for the array initialization.

We also support a generalization of the `forall` loop described above that allows references to values computed in other *earlier* iterations. As mentioned in Section 3, the body of the `forall` can contain references to previously initialized array members given a well-defined iteration order. The order guarantees deadlock dependence cycles cannot be created while initializing the array. If array accesses are attempted outside the defined iteration order an error is reported. We can support the access to other elements since each iteration is executed as a future. By using futures inside the `forall` body to resolve references to such elements, we are automatically assured that data dependences are respected while allowing maximum possible parallelism.

4.3 Scalar Reductions - Accumulators

Reductions represent a common pattern for computing a scalar value using an operation, such as summation, across multiple pieces of data from a collection of participating entities. If the operation is associative and/or commutative and we are interested only in the end result of the reduction, the computation can be parallelized using various constructs. When using structured fork-join task parallelism, the *finish accumulator* construct [16] can be used for user-defined deterministic reductions in parallel computations where the participating entities are the parallel tasks. In the case of data collections, these participating entities represent tasks computing values based on individual elements from the collection.

Figure 6 shows an example with parallel reductions based on built-in support for the sum operator to safely parallelize the reduction in the dot product computation of row and columns from the two input matrices. The accumulator registers on the `forall` loop in line 7. The individual parallel computations spawned from the `forall` loop all participate in the reduction by placing a value in the accumulator using the `put` operation. The end result of the computation is then safely read outside the `forall` loop in line 10 using the `get` operation. Attempts to read the value of the accumulator inside the `forall` body in line 8 would result in the

accumulator deterministically returning the initial seed value of `0.0`.

```
def multiplySeq(
  a: Array2d, b: Array2d): Array2d = {
 ...
 val res: Array2d = forall(R, C) {
  (row, col) =>
   val acc = new Accumulator(0.0)(_+_)
   forall(T)(acc) {
    acc.put(a(row)(i) * b(i)(col))
   }
   acc.get() // return the result
 }
 ...
}
```

Figure 6: Matrix multiplication using `forall` loop and finish accumulators to compute the dot product.

In our compilation and optimization process, we insert this form of parallelism when we can deduce it is applicable. The key issue is inferring that the reduction operation is commutative and/or associative.

5. Parallelization Optimizations

Functional programming exposes abundant opportunities for implicit fine-grain parallelism at the expression level. Such expressions can be represented as future constructs that compute the values of the expression in parallel with the rest of the computation. However, introducing parallelism creates overhead to manage and schedule the execution of the futures (tasks) and to avoid the blocking of resources (worker threads) at synchronization points where the results of futures are required by data dependence constraints. To achieve high performance on a parallel machine, such computations must be partitioned into a set of tasks that are neither too fine nor too coarse. If the compiler/runtime needs to coarsen the granularity of parallel tasks to achieve a balance between the degree of parallelism (number of tasks executing simultaneously) and the overhead required to manage parallel execution, a variety of transformations can be performed to achieve this restructuring. We discuss some of these transformations in this section.

5.1 Granularity

Despite the opportunity to parallelize the execution of essentially every expression, the overheads from scheduling and synchronization constraints typically dictate that we use coarser grain parallelism than the theoretical maximum in order to effectively amortize the overhead incurred in managing parallel tasks. Of course, we must also ensure that load balancing is not adversely affected. We control the granularity of the exposed parallelism using two main techniques: *a*) loop chunking and *b*) parallelism thresholds.

5.1.1 Chunking

Chunking a parallel `forall` loop converts the loop to a form where each task consists of multiple iterations instead of a single iteration. This transformation is widely used in explicit task parallel programs to increase the computation performed in each task. This technique also works well in the presence of synchronization constraints, such as resolving futures inside the loop body. In explicitly parallel languages like Cilk, Habanero-Scala, and Habanero-Java, the size of the chunks can be specified statically (at compile time) or discovered at runtime based on the computation load. The compiler and run-time for our implicitly parallel subset of Scala try to ensure the number of chunks created is divisible by the number of workers so that load balance does not become an issue.

5.1.2 Thresholds

We also use thresholds to determine when a fragment of work is too small to benefit from parallelization. In explicitly parallel languages, the usual cost of this approach is maintaining both a serial and a parallel version of the program. However, given that the user of our Scala subset provides a simple functional program that we automatically translate to explicitly parallel form, generating two separate versions of the code is not an issue. As shown in Figure 7, we use the `asyncSeq` construct to specify whether to perform the computation asynchronously or to perform the computation synchronously. A future is returned in either case, but the value of the future is already resolved when it is computed synchronously.

```
def quicksortParallel[T](
  list: List[T]): List[T] = {
 if (list.isEmpty) {
  list
 } else {
  val x = list.head
  val xs = list.tail
  val (lP, rP) = xs partition(_ < x)
  val leftFuture = asyncSeq(
    someThresholdCond) {
   quicksortSerial(lP)
  } {
   quicksortParallel(lP)
  }
  val rightFuture = ...
  leftFuture.resolve() ++
    (x :: rightFuture.resolve())
 }
}
```

Figure 7: Parallel version of Quicksort from Figure 2 using futures along with thresholds.

5.2 Memoization

Functional programming promotes recursion as the primary control structure governing program execution. Con-

sequently, our translator must support the automatic parallelization of recursive calls. As we saw in the Quicksort example in Figure 7, we can use futures to wrap results from recursive calls and perform computations in parallel. We can further optimize functional programs by avoiding redundant computations if we know that multiple recursive calls share the same inputs and hence produce the same results. The trick is to use memoization, where we make progress top-down while solving a problem and maintaining a transparent cache of unique futures to represent solutions to common sub-problems, *i.e.*, when the function is called with the same arguments. For performance reasons, we expect to limit the use of memoization to functions that take scalar values as inputs as opposed to functions like `quicksort` which take collections as input arguments.

5.3 Redundant Futures

We also detect instances of futures introducing synchronizations that are probably redundant because they are evaluated almost immediately. We call such futures *redundant futures*. In the Quicksort example in Figure 7, futures are created for the left and right fragments. However, the future for the right fragment is immediately resolved in the next statement (line 18). As a result, we avoid computing the right fragment asynchronously as a future by computing it directly in the main thread. Avoiding the creation of such futures reduces the number of tasks spawned and minimizes the amount of synchronizations that must be dynamically handled by the runtime.

5.4 Synchronization with Finish Blocks

In structured fork-join task parallelism, a *parent* task can fork off one or more *child* tasks that logically run in parallel with the parent task. The parent task then waits, by joining on all of its transitively spawned children as they complete execution. We represent such synchronization using *finish* blocks. The end of a *finish* block serves as the control point for the join operation. A finish block provides a concise mechanism for ensuring that all futures spawned within a computation have completed without explicitly resolving all of the futures. Finish blocks are also useful around data parallel `forall` loops (see Section 4.2) where tasks are spawned to handle chunks of iterations, but further progress must wait until all such tasks complete. This model can be easily extended to support nested fork-join parallelism, such as the example where nested `forall` loops exposed parallelism at two levels enclosed in nested finish blocks.

6. Implementation

Our parallel runtime is inspired from the async-finish task parallel model from Habanero-Java [2]. Our runtime implementation, however, exploits Scala's support for continuations to implement an efficient version of the various parallel constructs, avoiding thread blocking operations. Our compiler relies on the task parallel model exposed by the runtime to introduce explicit parallelism in the application programs.

6.1 Cooperative Runtime

The runtime system uses a cooperative scheduler to select a task to be dequeued and executed by a particular worker thread. Our solution is based on the use of *one-shot delimited continuations* (DeConts, Section 6.1.1) and *single assignment data-driven controls* (DDCs, Section 6.1.2) to schedule tasks cooperatively in the presence of different synchronization constraints due to the use of the different parallel constructs from Section 4.

The use of DeConts enables the runtime to suspend a task at any point where another runtime may have forced the task to block a worker thread. Further, each suspended task is registered as waiting on one or more DDC objects. A suspended task can be resumed when appropriate synchronization events trigger the resolution of values in all of the DDCs that it is waiting on. Our approach is more efficient than schedulers that spawn additional worker threads to compensate for blocked worker threads (as well as approaches that leave worker threads blocked with/without spawning new worker threads).

6.1.1 One-shot Delimited Continuations

Delimited continuations (DeConts) represent the rest of the computation from a well-defined outer boundary, i.e. a subcomputation. This allows DeConts to return to their caller allowing the program to proceed at the call site. One-shot continuations refer to continuations that are resumed at most once, this guarantee makes them cheaper to implement as it makes it simpler to save the computation state. With one-shot DeConts we always make forward progress in the defined subcomputation every time the continuation is resumed. Our use of DeConts for cooperative scheduling builds on our previous work to systematically combine one-shot continuations and DeConts to support a task-parallel runtime [10]. We rely on the use of the Scala Continuations compiler plugin [13] support for delimited continuations in our runtime. However, the end user is not aware of the use of continuations, the annotations required for continuations are planned to be inserted by our compiler.

6.1.2 Data Driven Controls

Data-Driven Controls (DDCs) were first presented in [9] to support event-driven actors into a task parallel runtime. A DDC lazily binds a value and a list of closures (also known as a block of code) called the execution bodies (EBs). When a value is available, each EB from the list of EBs is executed using the provided value. The value follows the dynamic single assignment property ensuring data-race freedom. EBs that are registered with the DDC after the value has been set will trigger execution of the EB immediately. The EBs of the DDC may be executed either asynchronously or synchronously. For example, in a task parallel runtime the EB

could store book-keeping data and act as a synchronous callback into the runtime. The EB could trigger possible asynchronous actions, such as scheduling and execution of a task, by interacting with the runtime.

6.2 Implementing Parallel Constructs

6.2.1 Futures

Producer tasks are responsible for resolving the values inside DDCs while consumer tasks suspend until the value inside a DDC has not been resolved. A common case is the single-producer multiple-consumer case, they are also known as *futures* [7]. A future represents an immutable value, a DDC in our runtime, which will become available at a later point. This value is computed asynchronously by a producer task. When the producer task completes execution it resolves the value inside the DDC thus resuming any previously suspended consumers. Consumers who force the value of the future after it has already been resolved can continue execution without being suspended.

6.2.2 Finish Blocks

A DDC, which wraps a counter[1], is created for each finish block in a parent task. The counter is atomically incremented each time a child task is forked inside the finish block and atomically decremented as each child tasks completes execution, either normally or abnormally. When the count reaches zero, the value of the DDC is resolved. The join operation serves as a possible suspension point in our runtime and uses the DDC as its cause for suspending if it is invoked before the count reaches zero. If the count is zero when the join operation is called, execution of the parent task continues without the need for suspension. One thing to note is that when a task is resumed, it can be executed on any worker thread.

6.2.3 `forall` Loops

A `forall` loop is implemented by spawning asynchronous tasks using chunking for iterations to increase the granularity. A finish block encapsulates the spawned tasks and ensures the statement following the `forall` loop is executed only after all the spawned tasks have completed. When using the `forall` loop to initialize an array, the generated code ensures the results of each iteration are used to initialize the values of appropriate elements.

6.3 Compiler Extensions

The compiler extensions to automatically convert the user's sequential code into a parallel version has not been implemented yet. We plan to implement it using a compiler plugin to verify whether the user code subscribes to the functional subset we defined in Section 3. Once verified, the compiler plugin will then go ahead and parallelize the sequential code fragments using models to statically determine the granularity where possible, otherwise leaving the decision for granularity choice to the runtime. Some models which have been used successfully in the past in SISAL are described in the related work section (Section 8).

[1] Distributed counters can be used for increased scalability.

7. Preliminary Results

We present the results of our preliminary work where we have hand optimized functional programs for a few benchmarks. The logical structure of the original functional program remains unchanged during the parallelization. All implementations of each benchmark use the same algorithm, but parallel constructs, discussed in Section 4, are introduced to transform the original "sequential" version to an equivalent parallel version. Eventually, these hand optimizations will be performed by our compiler.

7.1 Experimental Setup

The benchmarks were run on a 12-core (two hex-cores) 2.8 GHz Intel Westmere SMP node with 48 GB of RAM per node (4 GB per core), running Red Hat Linux (RHEL 6.0). Each core had a 32 kB L1 cache and a 256 kB L2 cache. The software stack includes a Java Hotspot JDK 1.7, Scala-Sisal 1.0-SNAPSHOT, and Scala 2.10.1. Each benchmark used the same JVM configuration flags: *a*) `-Xms768m` *b*) `-Xmx8192m` *c*) `-Xss4m` *d*) `-XX:MaxPermSize=256m` *e*) `-XX:+UseParallelGC` *f*) `-XX:+UseParallelOldGC` *g*) `-XX:-UseGCOverheadLimit` and was run for fifteen iterations in ten separate JVM invocations, the arithmetic mean of fifty execution times (last five iterations from each invocation) are reported. This benchmarking methodology is based on work from [4] and we use the last five execution times to approximate the steady state behavior of the application.

7.2 Matrix Multiplication

Sequential Execution Time:				78471.67
	Our Implementation		Parallel Collections	
Threads	Time	Speed-up	Time	Speed-up
1	79770.04	0.984	90856.94	0.864
2	41847.58	1.875	43510.11	1.804
4	22967.85	3.417	23172.42	3.386
6	16254.40	4.828	18122.74	4.330
8	13252.36	5.921	14493.94	5.414
10	11626.78	6.749	13599.06	5.770
12	11929.76	6.578	12534.55	6.260

Table 1: Matrix Multiplication Results using chunk size of 50 and input matrix of size 1500 $\times$ 1500. Execution times are reported in milliseconds.

The first benchmark is a matrix multiplication program which uses three nested loops to compute the resulting matrix, the sequential code for which is shown in Figure 3. In

our parallel implementation we parallelize the outer loop using a `forall` loop and compute the individual elements of the result matrix using a `foldLeft` operation to compute the dot product. We compare our performance against the Scala parallel collections framework [12] by using an outer parallel collection to present the range $[0...(R-1)]$. While using the parallel collections we feed in a custom `ForkJoinPool` executor to control the number of threads used in the parallelization. Table 1 displays the results of our comparison against the sequential program where we see our implementation achieves speedup comparable to the parallel collections. Our implementation introduces lower overhead during parallelization and also avoids any thread blocking (due to the cooperative runtime) while waiting for the parallel loop to complete.

7.3 Smith-Waterman

Sequential Execution Time:		30020.76
Threads	Execution Time	Speed-up
1	30191.27	0.994
2	15186.69	1.977
4	7601.52	3.949
6	5096.49	5.890
8	3857.09	7.783
10	3119.40	9.624
12	2643.88	11.355

Table 2: Smith-Waterman Results using tile size of 640 and input strings of length 37120 $\times$ 38400. Execution times are reported in milliseconds.

The next benchmark is a Smith-Waterman computation which uses a dynamic programming approach to compute the optimal alignment of two input strings. The computation of each element relies on three other elements: top, left, and top-left and is usually implemented as two nested loops. The well-defined data dependences in the loop makes it an ideal benchmark to use for futures and a `forall` loop (incrementing index in inner loop before incrementing index in outer loop order). We create futures to perform the computation of each element. Inside the computation, futures for the three dependences are resolved before computation proceeds. This exposes the maximum parallelism (wavefront parallelism) while respecting the data dependences. In addition, our cooperative scheduling ensures suspensions of tasks waiting on unresolved futures is handled efficiently. As seen in Table 2, we achieve almost perfect linear speed-up as we increase the number of worker threads, getting 11.36$\times$ speed-up on 12 threads. Note that the speed-up is reported with respect a sequential memoized program and not a purely functional one where common subproblems would have been recomputed. Such a comparison would have reported even higher speed-ups.

7.4 Quicksort

Sequential Execution Time:		5908.45
Threads	Execution Time	Speed-up
1	7425.84	0.796
2	3979.64	1.485
4	3029.23	1.950
6	2622.73	2.253
8	2227.03	2.653
10	2915.70	2.026
12	2290.29	2.580

Table 3: Quicksort Results using input array size of 4000000 and recursive call depth threshold of 4. Execution times are reported in milliseconds.

We use the Quicksort benchmark, sequential code for which is in Figure 2, to test the use of call-depth thresholds during parallelization. We use the `asyncSeq` construct to parallelize the sorting of the left fragment (as displayed in Section 5.3) while sorting the right fragment in the same task. This avoids redundant future creation and helps with performance. However, the partitioning and appending of the results is still performed in serial and takes O(N) time which limits the benefits from parallelization. As a result, we see the extremely disappointing performance benefit of only 2.6$\times$ speed-up on 12 worker threads (in Table 3). The lack of performance is due to the data structure (immutable list) used which has poor performance due to serialization when using the `partition` and `append` operator.

8. Related Work

Our programming model uses a functional subset of Scala that is inspired by SISAL [11]. SISAL is a single assignment applicative language where its functional nature is used by the compiler to exploit implicit parallelism. We follow a similar philosophy in which we expose a functional subset of Scala to mimic the SISAL programming style in a more modern programming framework.

Our runtime implementation is heavily influenced by the cooperative runtime in Habanero-Java [2, 10]. We promote all tasks to represent futures and allow support for computing the value of futures synchronously or asynchronously. We also provide extensions to support data parallel `forall` loops, in addition to the notion of finish blocks and accumulators to optimize the computation of fold operations.

Past work [1, 3] has shown that various compiler optimizations such as build-in-place analysis, update-in-place analysis, synchronization constraint insertion, etc. can be used to create efficient sequential and parallel implementations of a declarative language like SISAL. We will include similar analysis in transformations in the parallelization step of our compiler while determining granularity of computations.

We also plan to extend past work on partitioning Sisal programs [14, 15] in our compiler to statically determine the granularity at which parallelism is introduced into expressions. However, unlike their work we do not rely on the tasks to be able to run to completion without preemption, for example when a task needs to resolve a future prematurely. The efficiency of our technique will rely on the accuracy with which which we statically estimate the cost of executing statement blocks demarcated by suspension points. As in the past work, we will need to balance parallelism and overhead considerations when selecting an optimized partition.

9. Summary and Future Work

In this paper, we described our approach to implicitly parallelize programs written in a functional style that follows the single assignment rule. Our programming model uses a functional subset of Scala while our parallelization approach is based on automatic insertion of futures and accumulators. We avoid resource limitations arising from tasks blocked on futures, by executing delimited continuations on a cooperative scheduler with a fixed number of worker threads. Manual transformations of sequential user code using our parallel runtime is shown to provide good performance in some of our benchmarks.

Our current implementation has a fairly robust runtime where the different parallel constructs are implemented in the cooperative runtime. Next, we will implement our compiler plugin for Scala which will check that the user code is written in the functional subset of Scala we defined. The plugin will then go ahead and implicitly parallelize the code automatically determining the granularity to use along with other optimizations such as memoizing, avoiding redundant futures, etc. by analyzing the code and using techniques inspired from previous work on SISAL. Our goal is for our compiler to be able to generate code that competes with hand-coded versions on a wider variety of benchmark applications.

References

[1] D. Cann and J. Feo. SISAL versus FORTRAN - A Comparison Using the Livermore Loops. In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, Supercomputing '90, pages 626–636, Los Alamitos, CA, USA, 1990.

[2] V. Cavé, J. Zhao, Y. Guo, and V. Sarkar. Habanero-Java: the New Adventures of Old X10. *9th International Conference on the Principles and Practice of Programming in Java (PPPJ)*, August 2011.

[3] J.-L. Gaudiot, T. DeBoni, J. Feo, A. P. W. Böhm, W. A. Najjar, and P. Miller. The Sisal Project: Real World Functional Programming. In *Compiler Optimizations for Scalable Parallel Systems Languages*, pages 45–72, 2001.

[4] A. Georges, D. Buytaert, and L. Eeckhout. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 57–76, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-786-5.

[5] P. Haller and F. Sommers. *Actors in Scala*. Artima Incorporation, USA, 2012. ISBN 0981531652, 9780981531656.

[6] P. Haller, A. Prokopec, H. Miller, V. Klang, R. Kuhn, and V. Jovanovic. Futures and Promises - Scala Documentation. http://docs.scala-lang.org/overviews/core/futures.html, 2012.

[7] R. H. Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7:501–538, October 1985. ISSN 0164-0925. doi: 10.1145/4472.4478.

[8] J. I. Hsia, E. Simpson, D. Smith, and R. Cartwright. Taming Java for the Classroom. In *Proceedings of the 36th SIGCSE technical symposium on Computer science education*, SIGCSE '05, pages 327–331, New York, NY, USA, 2005. ACM. ISBN 1-58113-997-7.

[9] S. Imam and V. Sarkar. Integrating Task Parallelism with Actors. In *Proceedings of the ACM international conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 753–772, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1561-6.

[10] S. Imam and V. Sarkar. Efficient Cooperative Scheduling of Parallel Tasks with General Synchronization Patterns. Submitted for publication, 2013.

[11] J. Mcgraw, S. Skedzielewski, S. Allan, O. Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas. SISAL: Streams and Iterations in a Single Assignment Language, Language Reference Manual, Version 1.2. Technical Report M-146, LLNL, March 1985.

[12] A. Prokopec, P. Bagwell, T. Rompf, and M. Odersky. A Generic Parallel Collection Framework. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II*, Euro-Par'11, pages 136–147, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-23396-8.

[13] T. Rompf, I. Maier, and M. Odersky. Implementing First-Class Polymorphic Delimited Continuations by a Type-Directed Selective CPS-Transform. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 317–328, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-332-7.

[14] V. Sarkar and D. Cann. POSC A Partitioning and Optimizing SISAL Compiler. In *Proceedings of the 4th International Conference on Supercomputing*, ICS '90, pages 148–164, New York, NY, USA, 1990. ACM. ISBN 0-89791-369-8. doi: 10.1145/77726.255152.

[15] V. Sarkar and J. Hennessy. Partitioning Parallel Programs for Macro-Dataflow. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 202–211, New York, NY, USA, 1986. ACM.

[16] J. Shirako, V. Cavé, J. Zhao, and V. Sarkar. Finish Accumulators: a Deterministic Reduction Construct for Dynamic Task Parallelism. *The 4th Workshop on Determinism and Correctness in Parallel Programming (WoDet)*, March 2013.

[17] S. Taşırlar and V. Sarkar. Data-Driven Tasks and their Implementation. In *Proceedings of the International Conference on Parallel Processing (ICPP) 2011*, September 2011.