DFScala: High Level Dataflow Support for Scala

Daniel Goodman* Salman Khan Chris Seaton Yegor Guskov Behram Khan Mikel Luján Ian Watson
University of Manchester
Oxford Road
Manchester, UK
{*goodmand, salman.khan, seatonc, guskovy9, khanb, mlujan, watson*}*@cs.man.ac.uk*

***Abstract*—In this paper we present DFScala, a library for constructing and executing dataflow graphs in the Scala language. Through the use of Scala this library allows the programmer to construct coarse grained dataflow graphs that take advantage of functional semantics for the dataflow graph and both functional and imperative semantics within the dataflow nodes. This combination allows for very clean code which exhibits the properties of dataflow programs, but we believe is more accessible to imperative programmers. We first describe DFScala in detail, before using a number of benchmarks to evaluate both its scalability and its absolute performance relative to existing codes. DFScala has been constructed as part of the Teraflux project and is being used extensively as a basis for further research into dataflow programming.**

***Keywords*-Dataflow; Scala; Coarse Grained; Parallel Programming Model**

I. Introduction

In this paper we introduce and evaluate DFScala, a dataflow programming library to allow the construction of coarse grained Dataflow programs [20] in the Scala [14] programming language.

The recent trend towards ever higher levels of parallelism through rising core counts in processors has reinvigorated the search for a solution to the problems of constructing correct parallel programs. This in turn has revived interest in Dataflow programming [20] and associated functional programming approaches [2]. With these the computation is side-effect free and execution is triggered by the presence of data instead of the explicit flow of control. These constraints simplify the task of constructing and executing parallel programs as they guarantee the absence of both deadlocks and race conditions.

The Teraflux [19] project is investigating highly extensible multi-core systems including both hardware architectures and software systems built around the dataflow approach. One part of this project is the construction of a high level dataflow implementation which serves two purposes:

1) To provide a high productivity language in which to construct dataflow programs.
2) To provide a high level platform for experimenting with new ideas such as using the type system to enforce different properties of the dataflow graph and different memory models [16].

DFScala provides a key foundation and implements the base functionality of this research platform. One distinguishing feature of DFScala is the static checking of the dynamically constructed dataflow graph. This static checking ensures that at runtime there will be no mismatch of the arguments to functions. DFScala does not require the usage of special types and thus a node can be generated from any existing Scala function without complex refactoring of code. Each node in the dataflow graph is a function which cannot be subdivided; a function is sequential. To support nested parallelism within a function, subgraphs can be created which are wholly contained within a function, returning a value to the node upon completion.

In the remainder of this paper we first describe the dataflow programming model in more detail, we then introduce Scala before describing why we feel that a new dataflow library in Scala can improve on the existing libraries available in other languages. Next we describe the implementation and API of DFScala before evaluating its scalability and performance, considering alternative Scala dataflow libraries and finally making our concluding remarks.

II. Dataflow Programs

In a dataflow program the computation is split into sections. Depending on the granularity of the program these vary from a single instruction to whole functions which can include calls to other functions, allowing arbitrarily large computation units. All of these sections are deterministic based on their input and side-effect free. To enforce this, once a piece of data has been constructed it remains immutable for the lifetime of the program. The execution of the program is then orchestrated through the construction of a directed acyclic graph where the nodes are the sections of computation and the vertices are the data dependencies between these. An example of this can be seen in Figure 1. Once all the inputs of a node in the graph have been computed the node can be scheduled for execution. As such the execution of the program is controlled by the flow of

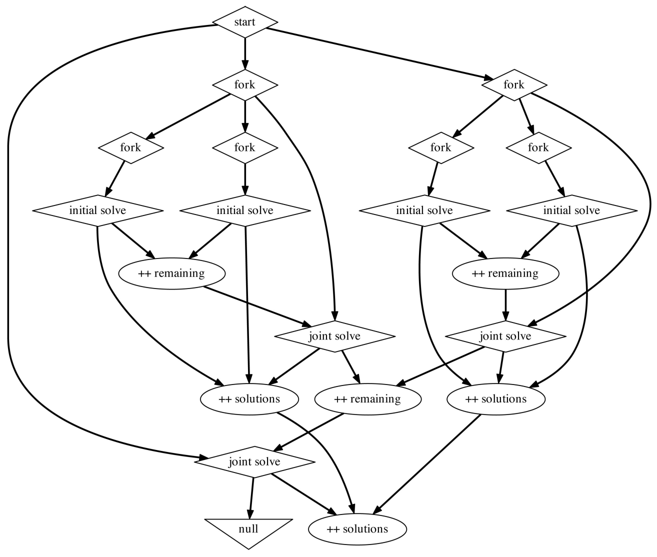

Figure 1. An instance of a dataflow graph for a circuit routing algorithm.

```
apply(life _, (x:Int) => x*x)

......

def life():Int = { 42 }

......

def apply( f1:() => Int, f2:Int => Int) =
{
  println(f2(f1()))
}
```

Figure 2. A snippet of Scala code demonstrating the construction of conventional and anonymous functions and their use as arguments in other function calls. The first line calls `apply` with the function `life` and an anonymous function as inputs. `life` returns the value 42 and the anonymous function takes an integer as an input and returns this value squared. `apply` takes these two functions as arguments and applies the second function to the output of the first. It then prints the resulting value, in this case 1764.

data through the graph unlike imperative programs where control is explicitly passed to threads.

Dataflow programming has been shown to be very effective at exposing parallelism [7] as the side-effect free nature means segments of code forming nodes whose inputs have been generated can be executed in parallel regardless of the actions of other segments in the graph and without effect on the deterministic result. Dataflow programs are by definition deadlock and race condition free. These properties make it easier to construct correct parallel programs.

In addition to having properties that reduce the complexity of constructing parallel programs, dataflow programs also reduce the required complexity of the supporting hardware. The explicit description within dataflow programs of when results must be passed allows for a relaxation of the cache coherency requirements. Instead of maintaining coherence at the instruction level it is now only necessary to ensure that results from one segment are written back to main memory before the segments that depend on these results begin. While it is argued that cache coherency does not limit the core count on processors [12] it is observable that well blocked programs that do not require cache coherency outperform those that do, and processors such as GPGPU's which are closer to the dataflow model achieve better performance per watt and higher peak performance.

III. The Choice of Scala

Scala [14] is a general purpose programming language designed to smoothly integrate features of object-oriented [21] and functional languages [2]. By design it supports seamless integration with Java, including existing compiled Java code and libraries. The compiler produces Java byte-code [11], meaning that Scala can be called from Java and visa versa. Scala is a pure object-oriented language in the sense that every value is an object. Types and behaviour of objects are described by classes and traits, and classes are extended by sub-classing and a flexible mixin-based composition mechanism as a replacement for multiple inheritance. However, Scala is also a functional language in the sense that every function is a value. This power is furthered through the provision of a lightweight syntax for defining anonymous functions as shown in Figure 2, support for higher-order functions, also seen in Figure 2, the nesting of functions, and for currying. Scala's case classes and its built-in support for pattern matching and algebraic types is equivalent to those used in many functional programming languages.

Scala is statically typed and equipped with a type system that enforces that abstractions are used in a safe and coherent manner. A local type inference mechanism means that the user is not required to annotate the program with redundant type information.

Finally, Scala provides a combination of language mechanisms that make it easy to smoothly add new language constructs in the form of libraries. Specifically, any method may be used as an infix or postfix operator, and closures are constructed automatically depending on the expected type (target typing). A joint use of both features facilitates the definition of new statements without extending the syntax and without using macro-like meta-programming facilities.

A. Why Create a New Dataflow Library

Having introduced Scala we will now describe how a Scala based dataflow framework extends the functionality provided by existing dataflow frameworks.

Current dataflow frameworks overwhelmingly fall into two categories: those implemented in pure functional programming languages [4], [5] and those implementations based on low level languages such as C [1], [18]. In addition to these there are a small number implemented in managed environments such as Java [10]. Scala's combination of both functional and imperative elements in a high level language allows it to add to these by introducing a dataflow programming library that maintains the accessibility to programmers of high level imperative languages by allowing imperative code in the dataflow nodes while adding the strengths and functionality of pure functional programming. In doing so it creates a bridge between these two styles of dataflow language.

Purely Functional Implementations: While functional programming is extremely powerful, the lack of mutable state prevents the construction of many intuitive programming constructs such as loops. This in turn makes these implementations challenging for many imperative programmers to use, limiting their uptake. While the functional semantics between nodes in a dataflow graph is the key to their power, there is no need to restrict the programmer to using purely functional semantics within the single threaded execution of the dataflow nodes. Scala provides the semantics and a clean syntax with which to add this freedom.

Other attempts have been made to address this by either adding specific imperative constructs to functional languages such as OCamel [17], which maintains a functional style of syntax and semantics while allowing a set of imperative constructs, or alternatively by the addition of functional constructs into languages such as C# [9] for specific tasks. However to date we are unaware of a language that would be familiar to imperative programmers both supporting dataflow computations and functional programming is such a way that these programs can be both clean and high level.

Pure Imperative Languages: Imperative libraries often allow for coarse grained dataflow nodes where the code within the node is imperative. However, the libraries that do this can be split into those supporting high level languages and those supporting low level languages.

Libraries based on low level languages have weaker type systems and do not support features such as garbage collection. For example StarSS [1] and TFlux [18] are pragma based systems, where sequential C code is annotated with task information including inputs and outputs or specific tasks and their dependences respectively.

High level languages are able to maintain a strong type system, and a managed environment with features such as garbage collection, however, the code can become restrictive or verbose as functions are explicitly wrapped to be passed to the library. For example the absence of functional semantics in Out of Order Java [10] requires the user to manually mark up tasks in the code which can then be examined by the compiler and runtime to determine if they can be executed in parallel. Scala's inbuilt ability to pass functions as first class variables removes this complexity allowing for cleaner code while maintaining the properties of a high level language. This is furthered by Scala's type inference system which prevents users from having to appreciate the subtleties of the types used by the underlying library. We are aware that there are other dataflow libraries for Scala available and we will consider these in section VI after we have described DFScala.

IV. DFScala Library

The DFScala library provides the functionality to construct and execute dataflow graphs in Scala. The nodes in the graph are dynamically constructed over the course of a program and each node executes a function which is passed as an argument. The arcs between nodes are all statically typed. An example of a function using the library can be seen in Figure 3.

The library is constructed from two principal components: a Dataflow Manager that controls the execution of the graph and a hierarchy of thread classes which represent the nodes in the graph. For any given dataflow graph there will be a single manager object; however, as we discuss in more detail later, within a given program there may be many independent dataflow graphs calculating different results for the overall program. These can exist either because dataflow functions have been inserted into legacy code, or because dataflow graphs are being nested within other dataflow graphs to calculate sub-results. We will now look at these components, starting with the Dataflow Manager.

A. *Dataflow Manager*

The Dataflow Manager performs two functions.

1) Scheduling dataflow threads to execute when all their inputs become available.
2) Providing a factory which takes a function and constructs the appropriate node for the graph (DFThread).

To achieve this, the Dataflow Manager is constructed from two parts: a static singleton object that contains the factory methods to construct new threads and a dynamic object that manages the scheduling of threads for a given dataflow graph. This separation allows the factory methods to be accessed directly from anywhere without having to explicitly pass a manager around, while at the same time there can be many managers each managing a separate dataflow graph.

B. *Dataflow Threads*

Threads are implemented through a hierarchy of classes. This hierarchy allows threads to be passed between functions without forcing the receiving functions to be overly specific about the type of thread they are expecting, instead merely specifying the required properties of the thread within the function.

Passed arguments are called tokens. The threads support static type checking of the tokens they pass between each other. This checking allows the compiler to guarantee the

Expanded Version

```
def fib(n :Int , out:Token[Int]){
   if(n <= 2)
      out(1)
   else {
     var t1 = createThread(
         (x:Int, y:Int,
          out:Token[Int]) => {out(x + y)}
     )
     var t2 = createThread(fib _)
     var t3 = createThread(fib _)

     t2.arg1 = n - 1
     t2.arg2 = t1.token1

     t3.arg1 = n - 2
     t3.arg2 = t1.token2

     t1.arg3 = out
  }
}
```

Concise Version

```
def fib(n :Int):Int = {
  if(n <= 2)
    1
  else
    fib(n-1) + fib(n-2)
}
```

Figure 3. An example of a dataflow program to compute Fibonacci numbers. This demonstrates the API, but the functional nature of the language means this function could be automatically constructed from the more concise function appearing below.

type correctness of a dataflow graph. To improve programmability threads will not release any threads or tokens they create or pass until the thread completes its execution. This restriction simplifies the task of keeping track of the order in which events can occur when debugging.

All dataflow threads are constructed from a base class DFThread which contains all the logic required to interact with the Dataflow Manager, and manage the storing of tokens and newly constructed threads until the thread completes. This class is then extended by a chain of abstract classes, each of which represents a thread that takes at least n arguments for increasing values of n. Finally each of these classes is extended with a concrete class that contains the functionality to execute a single method.

This separation of methods and the collection of arguments not only saves re-implementing code for each of the concrete classes, but also allows the casting down of DFThreads to threads that expect less arguments, and

```
def join (arg1 :Int, arg2 :Int,
     out: DFThread4[Int, Int, _, Pos, _],
     position :Int)
{
  if(position == 1)
    out.arg1 = arg1 + arg2
  else
    out.arg2 = arg1 + arg2
}
```

Figure 4. An example of a function for reducing multiple values through a binary tree. Note the additional complexity determining which value to set and the fixed type of the thread argument. Compare this with the reduction using tokens in Figure 5. Figure 3 shows a function using tokens to pass data between threads when reducing a binary tree.

```
def join (arg1 :Int, arg2 :Int,
          out: Token[Int])
{
  out(arg1 + arg2)
}
```

Figure 5. Using the indirection provided by tokens to simplifiy the code in Figure 4

the construction of special classes of thread. Such special threads include those used to collect a number of elements and either return them as a collection or reduce them to a single value using an appropriate function. This is achieved by allowing the thread to have multiple tokens passed to one argument. The number of times tokens can be passed is specified at runtime when the thread is created. This class of threads is required because, while the number of threads spawned can be defined at runtime, for example by iterating round a loop and spawning a thread on each iteration, the number of tokens received by a regular thread is set by the thread type at compile time.

C. Passing Tokens

Tokens can either be pushed by the producing thread, or they can be pulled by the consuming thread.

Pushing Tokens: If a thread is available in the code that generates its token, the token can be directly assigned, for example: `t.arg2 = 10`. This invokes an underlying setter method which handles the control logic. The type of the thread specifies the number and types of arguments that can be passed. As threads with higher numbers of arguments extend threads with lower numbers of arguments any thread which has n or more arguments of the appropriate type can be used.

Setting thread arguments directly can be convenient, however, as the signature of the thread is fixed, this can also be restrictive. For example to reduce a binary tree to a single value a thread can be constructed for each node, these threads take two arguments, a thread and a parameter to

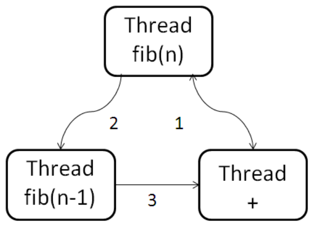

Figure 6. A diagram of the events when a token is used in Figure 3. 1. The thread calculating fib(n) requests the token for the first argument of the function that will perform the addition. 2. This token is passed to the thread calculating fib(n-1). 3. This thread sets the token to the correct value once it is known, in doing so it passes this value

indicate if the result is output to the left or right argument of the next thread. See Figure 4. This approach has two problems:

1) First the parameter indicating where to assign the argument adds complexity;
2) Second the signature of the thread that the final result is to be passed to, is now required to match that of the intermediate threads. This will clearly not normally be the case.

Token objects address this by acting as a proxy for the argument setter method, so allowing the consumer and producer to be decoupled. Tokens can be retrieved from threads for each argument. As the token is a proxy for the setter method it handles all the required control logic to manage the release of the input once the thread has finished executing. An example of this technique also being used to merge two values in a binary tree can be seen in the Fibonacci program in Figure 3, and the resultant sequence of events can be seen in Figure 6. The code that would be used to replace that presented in Figure 4 can be seen in Figure 5.

Pulling Tokens: Pushing values into threads directly from within the code allows for very descriptive code, but it has two key weaknesses:

1) The executing thread has to be aware of how many other threads need to receive a given value as a token.
2) Legacy functions cannot be used as they do not include the required code to push tokens to other threads.

To address this all DFThreads have the ability to take Tokens as listeners. Then, once the thread has completed, these Tokens will be assigned the value returned by the thread's function. This allows the thread's result to be distributed to an arbitrary number of threads, and allows legacy functions to be included within a dataflow graph.

D. Constructing a Dataflow Graph

Two techniques are provided to start the initial thread of a dataflow graph depending on the situation in which the thread is to be started.

1) Dataflow Applications: The simplest way to start a dataflow graph in DFScala is, instead of constructing a main method to start the program, to extend the class `DFApp` implementing the method `dfMain`. When this extended class has its main method called to start the program, it will perform all the required initialisation and start a thread which will call the function `dfMain` with the arguments provided to main.

2) Nested Dataflow Graphs: Extending a class and calling its main method is not always an appropriate way to start a dataflow graph, either because another library such as QT [13] that uses this technique is being used, or because it is desirable to insert independent dataflow graphs into either a legacy program, or into another dataflow graph. We will now look at this final scenario in more detail.

As this is a coarse grain dataflow model, dataflow threads may call other functions. These functions may contain code that it is advantageous to run in parallel. However, to ensure the absence of deadlocks in the dataflow graph, once a thread has started it cannot receive input from any other thread within the graph. This prevents the creation of additional worker threads within called functions. This could be addressed by splitting the thread into two threads at the function call. This would then allow the first thread to create the worker threads and then for the worker threads to pass their results on to the second thread. However, this has several disadvantages:

- It requires the tokenizing of any thread local data that spans the split threads.
- The separation either has to be done at runtime which could be expensive, or the compiled code has to know if the function will be executed in parallel, which prevents the decision from being made at runtime based on data set size etc.

To overcome this a dataflow thread can create a NestedGraph object representing an independent dataflow graph. This will run to completion and return a result. The independent execution allows the function to continue to execute as pure dataflow while spawning helper threads.

The NestedGraph provides two elements: An object of type Token that can be passed into the dataflow graph to return a result; and a function that takes a function and a set of arguments for the function. The passed function is then executed as the root of a separate dataflow graph. This separate execution allows it to spawn additional threads to handle the computation. An example of this can be seen in Figure 7.

To implement the nested dataflow graphs, the NestedGraph object creates a new manager object to manage the graph. It then runs the passed function using a new thread obtained from this manager. This ensures that all threads created by this function will also be run using this new manager so ensuring their independence of the existing

```
def foo(argument: Int): Int = {
    val subgraph = new NestedGraph[Int]
    subgraph.createThread(
                              bar _,
                              argument,
                              subgraph.token1
                             )
}
```

Figure 7. An example of using a `NestedGraph` object to nest a separate dataflow graph in a function. The nested graph is spawned by executing the function passed to `createThread` (in this case `bar`) in a new dataflow thread. `createThread` also takes the arguments for `bar` as its remaining arguments.

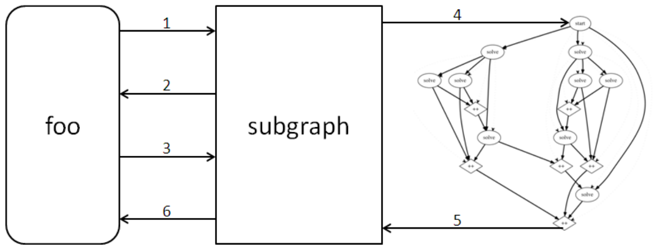

Figure 8. The chain of events involved in the construction and execution of an independent dataflow graph within an existing program. 1. A NestedGraph object is created by the function foo (subgraph). 2. A result token is retrieved from this object. 3. The root thread of the independent graph is constructed by foo providing a function (bar) and the required arguments. This call blocks while the graph is executed. 4. subgraph constructs all the required framework and executes bar and any resulting child threads. 5. The dataflow graph sets the value of the passed token, in doing so this value is passed to the object subgraph. 6. The create thread method returns the resultant value passed from the dataflow graph.

dataflow graph. A diagram detailing this chain of events can be seen in Figure 8

E. Debugging and Performance Analysis

It is important to be able to extract from an execution the dataflow graph that was executed. This information is needed for the detection of errors, performance measuring and optimisation. In terms of error detection, there is potential for both deadlocks and livelocks resulting from the failure to pass tokens. This can result from an error that occurred in a thread that has long since finished executing, and being able to trace where a thread was expecting an argument from is important. To address this the library makes calls out to a DFLogging object which, depending on the way the application was started, will record the information passed to it in such a way that debugging tools can read this information while the program is executing, or it can be saved onto disk or into a database for later analysis.

V. Evaluation

To evaluate the scalability of DFScala we implemented four benchmarks and ran them on both Intel and AMD based systems. The benchmarks we used were: KMeans, 0-1 Knapsack, Monte-Carlo Tree Search, and Matrix Matrix Multiplication. To evaluate absolute performance we implemented a version of Scala Parallel Collections based on DFScala. All tests used Scala 2.9.

KMeans is an algorithm that groups points into clusters through a process of iterative refinement. In this instance there are 16 thousand 24 dimensional points which form 16 clusters.

0-1 Knapsack is an optimisation problem requiring items to be picked from a set such that their weight does not exceed a given amount while maximising the overall value of the items. This benchmark solves the problem through the use of dynamic programming. In this instance there are 1000 items and a maximum weight of 10,000.

Monte-Carlo Tree Search is a randomised technique used to search a state space for a best guess at an optimum solution. It is used in situations where searching exhaustively is too costly. In this instance it is used by a computer player to look for moves in a game of Go [6].

Matrix Matrix Multiplication For this benchmark we performed Matrix Matrix multiplication on matrices of size varying from 256x256 to 2048x2048.

A. Results

For each of these we used a constant problem size and varied the number of system threads. For KMeans we recorded the time for each iteration, for the other benchmarks we iterated the algorithm numerous times. We then took the median time and used this to determine the speedup relative to the single threaded solution. The result of this analysis can be seen in Figures 9 and 10 for these run on a 1.6GHZ 4 core Intel Xeon machine and a dual socket 2.2GHz 12 core AMD Opteron machine. These show that for both test platforms DFScala shows very good scaling properties for all four benchmarks. The inflection point on the Xeon processor is a well documented artefact of Hyper Threading, not a property of the library.

B. Performance

To demonstrate that this scalability was not gained at the expense of performance when compared with existing solutions we also implemented a version of Scala Parallel Collections based on DFScala. Both this version and the current Scala Parallel Collections shipped with the Scala runtime had their performance compared relative to the standard single threaded Scala Collections libraries. This means that the speed improvements shown in the graphs in Figure 11 and 12 is the improvement that would be seen by replacing the currently shipping single threaded libraries with these implementations, not just the performance gain by adding more threads to our own implementation. The results of these tests on a 4 socket 48 core 2.2 GHz AMD Opteron based system can be seen in Figures 11 and 12. The specific test used here was mapping a function to count the prime

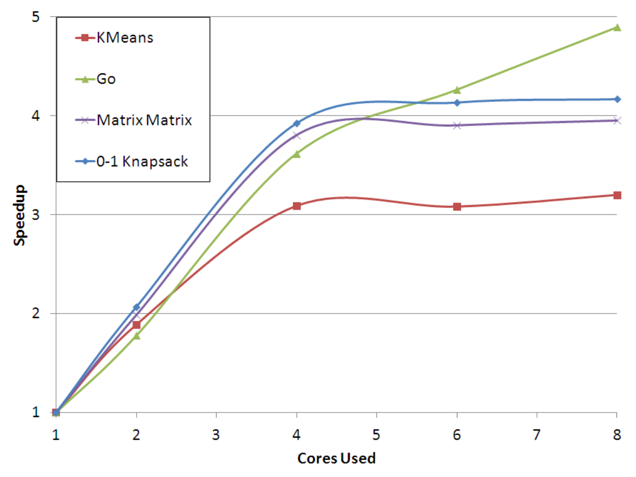

Figure 9. Benchmark performance on a 4 core Intel Xeon processor with Hyper Threading.

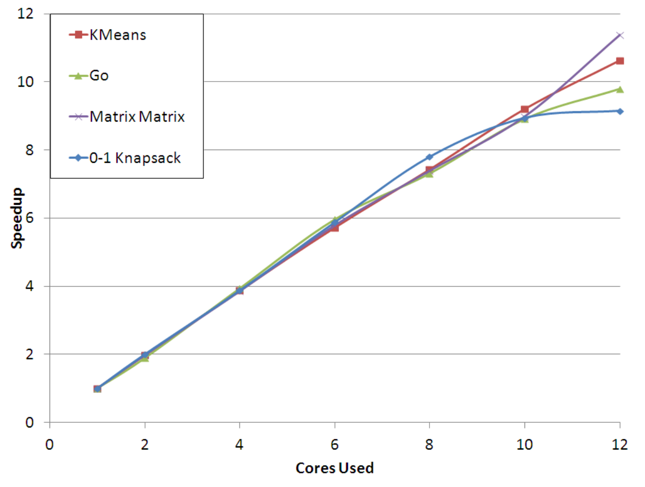

Figure 10. Benchmark performance on a machine with two 6 core AMD Opteron processors.

factors of an integer across all the elements in a collection. These show that the scalability of the library based on DFScala is maintained and the ultimate performance exceeds that of the current implementations of Parallel Collections as the size of the datasets increases. We attribute the better performance to the dataflow framework resulting in better separation of the data used by the different threads and reducing the need for inter-thread communication. Unfortunately due to limitations of the existing parallel collections library it was not possible to compare the performance of the two with differing numbers of threads.

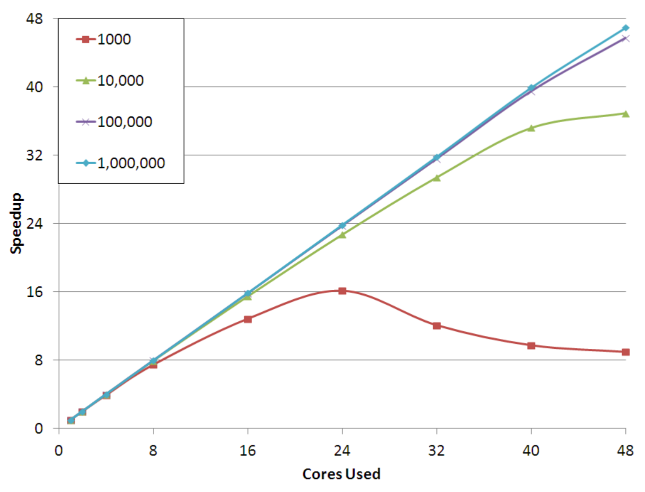

Figure 11. Graph showing the speedup provided by our Dataflow collections with differing numbers of elements and threads. All speedups are normalised against the performance of the standard Scala single threaded collections.

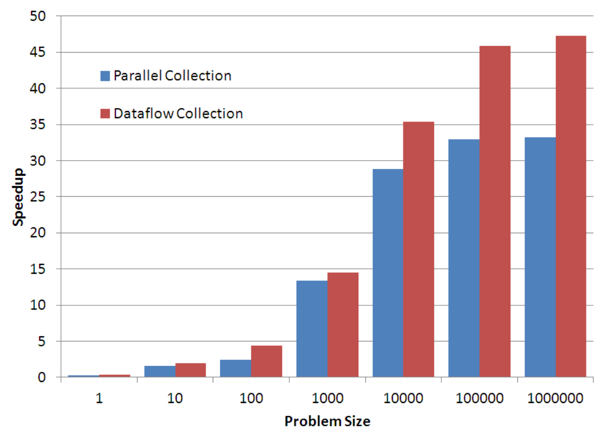

Figure 12. Graph showing the performance of Scala Parallel collections and our Dataflow collections library on a 48 core AMD based machine for differeing sizes of collection. Again all speedups are normalised against the performance of the standard Scala single threaded collections.

VI. Alternative Libraries

We are aware of two other implementations of dataflow for the Scala programming language, Akka [15] and CNC-Scala [8]. Akka is a Scala framework for highly concurrent, distributed, event-driven applications. It provides a set of tools for concurrency in a distributed environment including dataflow. CnC-Scala is a port of the Intel Concurrent Collections programming model [3] from C++ to Scala. Like DFScala, both the Akka and CnC-Scala models are coarse-grained, deterministic dataflow. All three models require that the programmer either writes side-effect free code, or carefully implements synchronisation to maintain the dataflow properties.

CnC-Scala is aimed at the “parallelism oblivious developers” [8] and like DFScala places an emphasis on high productivity. However, the CnC-Scala implementation uses a pre-processor to translate a DSL into pure Scala. This has implications for tooling such as IDEs and debuggers that are not aware of the DSL. The programming model of CnC- Scala is closer to that of the C++ original, requiring each node to be a method in a separate class. We think that our model of one function per node, without any extra infrastructure, is more likely to encourage the kind of parallel- programming-by-default that may be needed for many-core architectures. Like Akka, by default the user's kernel function is run before all input is ready and then explicitly waits.

VII. Conclusion

In this paper we have presented a library for constructing and executing coarse grained dataflow tasks in Scala. In doing so we have built on Scala's merging of Object Oriented programming and Functional programming to provide a means of constructing dataflow graphs that cleanly takes advantage of functional semantics while allowing the user to use more familiar imperative programming within the node. We believe that this will allow the construction of cleaner dataflow codes that are more accessible to the majority of programmers.

This library forms the basis of a wide range of work within the Teraflux project and has been used to implement a sizable number of applications of varying sizes from small benchmarks to entire parts of the Scala runtime. A subset of these applications is described here, along with measurements of their scalability and performance on a range of systems. These results show that DFScala provides effective scaling, and good performance. There is always room for improvement and targets for such improvement include the addition of a more advanced scheduler and reducing the overhead of creating DFThread objects. However, as described in the introduction the Teraflux project aims to develop a complete system including processors, compilers, tools and languages for dataflow computing, and as a result many of the overheads in this first software implementation will ultimately be able to be off loaded onto dedicated hardware. Applications written using this library can then be used to assess the effectiveness of any such hardware. In addition this library is also being used as a platform to experiment with different memory models, the automatic insertion of dataflow constructs into programs and the enforcement of specific properties of dataflow graphs.

Acknowledgment

The authors would like to thank the European Community's Seventh Framework Programme (FP7/2007-2013) for funding this work under grant agreement no 249013 (Teraflux-project). Chris Seaton is an EPSRC funded student. Dr. Luján is supported by a Royal Society University Research Fellowship.

References

[1] Ayguadé, E., Badia, R. M., Igual, F. D., Labarta, J., Mayo, R., and Quintana-Ortí, E. S. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing* (Berlin, Heidelberg, 2009), Euro-Par '09, Springer-Verlag, pp. 851–862.

[2] Bird, R. *Introduction to Functional Programming using Haskell*, second ed. Prentice Hall, 1998.

[3] Burke, M. G., Knobe, K., Newton, R., and Sarkar, V. The concurrent collections programming model. Tech. Rep. TR 10-12, Rice University, 2010.

[4] Cann, D. Retire Fortran?: a debate rekindled. *Commun. ACM 35* (August 1992), 81–89.

[5] C.A.R. Hoare, Ed. *OCCAM Programming Manual*. Orentice Hall International Series in Computer Science. Prentice Hall International, 1984.

[6] Chaslot, G., Winands, M. H. M., and van den Herik, H. J. Parallel monte-carlo tree search. In *Computers and Games* (2008), pp. 60–71.

[7] Gurd, J. R., Kirkham, C. C., and Watson, I. The Manchester prototype dataflow computer. *Commun. ACM 28* (January 1985), 34–52.

[8] Imam, S., and Sarkar, V. CnC-Scala: a declarative approach to multicore parallelism. In *Scala Days* (2012).

[9] International, E. *Standard ECMA-334 - C# Language Specification*, 4 ed. June 2006.

[10] Jenista, J. C., Eom, Y. h., and Demsky, B. C. Ooojava: software out-of-order execution. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming* (New York, NY, USA, 2011), PPoPP '11, ACM, pp. 57–68.

[11] Lindholm, T., and Yellin, F. *Java Virtual Machine Specification*, 2nd ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[12] Martin, M. M. K., Hill, M. D., and Sorin, D. J. Why on-chip cache coherence is here to stay. *Communications of the ACM* (2012).

[13] Nokia. *QT, http://qt.nokia.com/*, 2012.

[14] Odersky, M., Spoon, L., and Venners, B. *Programming in Scala: [a comprehensive step-by-step guide]*, 1st ed. Artima Incorporation, USA, 2008.

[15] Scalable Solutions AB. Akka project, February 2011.

[16] Seaton, C., Goodman, D., Luján, M., and Watson, I. Applying dataflow and transactions to Lee routing. In *Workshop on Programmability Issues for Heterogeneous Multicores* (2012).

[17] Smith, J. B. *Practical OCamel (Practical)*. Apress, Berkely, CA, USA, 2006.

[18] Stavrou, K., Nikolaides, M., Pavlou, D., Arandi, S., Evripidou, P., and Trancoso, P. Tflux: A portable platform for data-driven multithreading on commodity multicore systems. In *Proceedings of the 2008 37th International Conference on Parallel Processing* (Washington, DC, USA, 2008), ICPP '08, IEEE Computer Society, pp. 25–34.

[19] Teraflux. *The TERAFLUX project, http://www.teraflux.org*, 2010.

[20] Watson, I., Woods, V., Watson, P., Banach, R., Greenberg, M., and Sargeant, J. Flagship: A parallel architecture for declarative programming. In *ISCA* (1988), pp. 124–130.

[21] Wu, C. T. *An Introduction to Object-Oriented Programming with Java 2nd Edition*, 2nd ed. McGraw-Hill, Inc., New York, NY, USA, 2000.