Cuneiform

A Functional Language for Large Scale Scientific Data Analysis

Jörgen Brandt Marc Bux Ulf Leser

Humboldt-Universität zu Berlin
Unter den Linden 6, D-10099 Berlin, Germany
{brandjoe, bux, leser}@informatik.hu-berlin.de

ABSTRACT

The need to analyze massive scientific data sets on the one hand and the availability of distributed compute resources with an increasing number of CPU cores on the other hand have promoted the development of a variety of languages and systems for parallel, distributed data analysis. Among them are data-parallel query languages such as Pig Latin or Spark as well as scientific workflow languages such as Swift or Pegasus DAX. While data-parallel query languages focus on the exploitation of data parallelism, scientific workflow languages focus on the integration of external tools and libraries. However, a language that combines easy integration of arbitrary tools, treated as black boxes, with the ability to fully exploit data parallelism does not exist yet. Here, we present Cuneiform, a novel language for large-scale scientific data analysis. We highlight its functionality with respect to a set of desirable features for such languages, introduce its syntax and semantics by example, and show its flexibility and conciseness with use cases, including a complex real-life workflow from the area of genome research. Cuneiform scripts are executed dynamically on the workflow execution platform Hi-WAY which is based on Hadoop YARN. The language Cuneiform, including tool support for programming, workflow visualization, debugging, logging, and provenance-tracing, and the parallel execution engine Hi-WAY are fully implemented.

1. INTRODUCTION

Over the recent years, data sets in typical scientific (and commercial) areas have grown tremendeously. Also, the complexity of analysis procedures has increased at essentially the same pace. For instance, in the field of bioinformatics the cost and speed at which data can be produced is improving steadily [3, 18, 35]. This makes possible entirely novel forms of scientific discoveries which require ever more complex analysis pipelines (e.g., personalized medicine, meta-genomics, genetics at population scale) developed by tens of thousands of scientists around the world. Analysis infrastructures have to keep up with this development. In particular, they must be able to scale to very large data sets, and they must be extremely flexible in terms of integrating and combining existing tools. Scientific workflow management systems have been proposed to deal with the latter problem [12]. Scientific workflows are descriptions of data processing steps and the flow of information between them [10]. Most scientific workflow languages like Galaxy [17] or Taverna [21] allow the user to create light-weight wrappers around existing tools and libraries. In doing so, they avoid the necessity of reimplementing these tools to conform with the API of the execution environment. However, many scientific workflow systems do not take advantage of splitting input data into partitions to exploit data parallelism which limits their scalability. In contrast, partitioning input data to achieve data parallelism is a main advantage of the execution environments underlying data-parallel query languages like Pig Latin [16, 33] or Spark [45], thus, addressing the former issue. However, these are not designed to make native integration of external tools easy. Instead, they require developers to create heavy wrappers around existing tools as well as to perform costly conversion of data from their native formats to the system-specific data model, or they expect developers to reimplement all algorithms in their specific data-parallel languages. While creating wrappers is laborious, error-prone, and incurs runtime penalties, reimplementation is infeasible in areas like genomics where new algorithms, new data types, and new applications emerge essentially every day.

To the best of our knowledge, a language for large-scale scientific computing that offers both light-weight wrapping of foreign tools and high-level data-parallel structures currently does not exist. Here, we present Cuneiform, a language that aims at filling this gap. Cuneiform is a universal functional workflow language offering all important features currently required in scientific analysis like abstractions and a dynamic execution model in a language with implicit state. Its main focus, however, is (i) the ease with which external tools written in any language can be integrated, and (ii) the support for a rich set of algorithmic skeletons (or second-order functions) enabling automatic parallelization of execution. Cuneiform is fully implemented and comes with decent tool support, including executable provenance traces, visualization of conceptual and physical workflow plans, and debugging facilities. Cuneiform workflows can be executed on the workflow execution engine Hi-WAY which builds on Hadoop YARN. Together, Cuneiform and Hi-WAY form a fully functional, scalable, and easily extensible scientific workflow system.

The remaining parts of this paper are structured in the following way: Section 2 introduces characteristics of workflow languages important for scientific analysis that drove the design of Cuneiform. We introduce Cuneiform by example in Section 3. In Section 4 we explain the implementation of two exemplary workflows in Cuneiform to highlight its versatility and expressive power. Hi-WAY is briefly introduced in Section 5.[1] Related work is presented in Section 6, and Section 7 concludes the paper.

2. LANGUAGE CHARACTERIZATION

In this section we outline the language properties that we consider important for parallel scientific analysis and that drove the design of Cuneiform. We categorize different types of languages and discuss important language features.

First, we can distinguish languages with *implicit state* and *explicit state* [36]. Implicit state languages do not have mutable memory. Purely functional languages like Haskell or Miranda have implicit state. In these languages variables are only place-holders for immutable expressions. In contrast, imperative languages like C or Java and multi-paradigm languages like Scala or Lisp have explicit state. In these languages variables can change their values throughout the execution of a program.

Scientific workflow languages are typically implicit state languages while Spark [44, 45], FlumeJava [7], or DryadLINQ [14, 43] inherit explicit state from their respective host languages. There are arguments promoting either of both approaches: Implicit state languages profit from their ability to fully exploit task parallelism because the order of tasks is constrained only by data dependencies [6].[2] Explicit state is preferable if functions need to learn from the past and change their behavior [36]. However, the introduction of explicit state incurs additional constraints on the task execution order, thereby, limiting the ability to automatically infer parallelism.

[1]Note that the focus of this paper is on Cuneiform, while a complete description of Hi-WAY will be published elsewhere.
[2]Taverna is an exception as it introduces control links to explicitly constrain the task execution order in addition to data dependencies.

In the following we outline the requirements towards a scalable workflow specification language. By focusing on this specific set of requirements, we define the scope for the discussion of Cuneiform and for the comparison of different languages. This list of requirements is, however, not comprehensive.

Abstractions In the Functional Programming (FP) paradigm the term abstraction refers to an expression that binds one (or more) free variables in its body. When a value is applied to the expression, the first bound variable is replaced with the applied value. In scientific workflows, abstractions are referred to as subworkflows where the subworkflow's input ports represent the bound variables. While abstractions are common in functional languages for distributed computation (like Eden [5, 26]) or distributed multi-paradigm languages (like Spark), some scientific workflow languages do not allow for the definition of abstractions in the form of subworkflows, e.g., Galaxy [17]. Other scientific workflow languages like Pegasus DAX [13], KNIME [4], Swift, or Taverna do provide subworkflows. Pig Latin [16, 33] introduces abstractions through its macro definition feature. Since abstractions facilitate the reuse of reoccurring patterns, they are an important feature of any high-level programming model.

Conditionals A conditional is a control structure that evaluates to its then-branch only if a condition is true and otherwise evaluates to its else-branch. Like abstractions, conditionals are common in functional and multi-paradigm languages. Many scientific workflow languages provide conditionals as top-level language elements [2], e.g., KNIME, Taverna, or Swift. However, in some other scientific workflow languages they are omitted, e.g., Galaxy or Pegasus DAX. Also, Pig Latin comes without conditionals that would allow for alternate execution plans depending on a boolean condition. Spark, on the other hand, inherits its conditionals from Scala. Conditionals are important when a workflow has to follow a different execution path depending on a computational result that cannot be anticipated a priori. For instance, consider a scenario where two algorithms can be employed to solve a problem. One algorithm performs comparably better on noisy input data, while the other performs better on clean data. If assessing the quality of the data is part of the workflow then the decision what algorithm to use has to be made at execution time. Another example is the application of an iterative learning algorithm. If the exit condition of the algorithm is determined by some convergence criterion, the number of iterations cannot be anticipated a priori. This way, conditionals introduce uncertainty in the concrete workflow structure making it impossible to infer a workflow's invocation graph prior to execution. Nevertheless, conditionals are an important language feature.

Composite data types Composite data types are data structures composed of atomic data items. Lists, sets, or bags are composite data types. In many cases, languages with support for composite data types also provide algorithmic skeletons (see below) to process them, e.g., map, reduce, or cross product. Swift, KNIME, and Taverna are scientific workflow languages with support for composite data types. Other scientific workflow languages, like Galaxy or Pegasus DAX, support only atomic data types. Data-parallel query languages, like Spark or Pig Latin, however, provide extensive support for composite data types. Note that composite data types, like conditionals, introduce uncertainty in the concrete workflow structure before its actual execution. For instance, if a task outputs a list with an unknown size and each list item is consumed by a proper subsequent task, the number of such tasks is unknown prior to execution. This calls for a dynamic, adaptive approach to task scheduling. Using composite data types is a powerful and elegant way to specify data-parallel programs.

Algorithmic skeletons Algorithmic skeletons are second order functions that represent common programming patterns. From the perspective of imperative languages, they can be seen as templates that outline the coarse structure of a computation [8]. To exploit the capabilities of parallel, distributed execution environments, a language can emphasize parallelizable algorithmic skeletons and de-emphasize structures that could impose unnecessary constraints on the task execution order. For instance, expressing the application of a function to each element of a list as a for-loop with an exit condition dismisses the parallel character of the operation. Expressing the exact same operation as a map, on the other hand, retains the parallelism of the operation in its language representation. Some scientific workflow languages, like Pegasus DAX or Galaxy, do not provide any algorithmic skeletons. In contrast, Taverna, Swift, or KNIME provide algorithmic skeletons in various forms. For instance, Taverna implicitly iterates lists if an unary task is applied to a list. Moreover, Taverna provides cross- and dot product skeletons. Swift provides the foreach and iterate-until skeletons. Algorithmic skeletons are particularly important in Scala. Thus, Spark exposes a number of algorithmic skeletons to control distributed computation [44]. Pig Latin uses algorithmic skeletons based on the SQL model of execution. Like general abstractions, algorithmic skeletons facilitate the reuse of reoccurring algorithmic patterns. Such patterns commonly appear in scientific data analysis applications.

Foreign Function Interface (FFI) An FFI allows a program to call routines written in a language other than the host language. Many programming languages provide an FFI with the goal of accelerating common subroutines by interfacing with machine-oriented languages like C. Scientific workflow languages provide FFIs in the form of simple wrappers for external tools. For instance, Swift and Pegasus DAX allow language users to integrate Bash scripts. Taverna provides Beanshell and R services, and KNIME provides snippet-nodes for Java, R, Perl, Python, Groovy, and Matlab. These FFIs do not have the purpose to accelerate routines but to integrate existing tools and libraries with minimum effort. In Pig Latin or Meteor [19], User Defined Functions (UDFs) are provided in the form of Java libraries which need to be wrapped by an extra layer of code providing particular data transformations from the tools native file formats to the system's data model and back. Similar wrappers have to be implemented to use foreign tools in Spark. The FFI is the language feature that makes integration of external tools and libraries easy. It is the entry point for any piece of software that has not been written in the host language itself. A general and light-weight FFI enables researchers to reuse their tools in a data-parallel fashion without further adaptation or the additional layer of complexity of a custom wrapper.

Universality Universal languages can express any computable function. Most general purpose programming languages are universal. Scientific workflow languages including Swift, Galaxy, Taverna, and Pegasus DAX are not universal. Additionally, some data-parallel query languages like Pig Latin are not universal. In contrast, Skywriting is an example for a universal language for distributed computation [28]. Spark inherits the universality property from Scala. Similarly, FlumeJava and DryadLINQ inherit the universality property from their respective host languages Java and C#. We do not consider universality a requirement for a workflow specification language. Nonetheless, it is a language property worth investigating.

3. CUNEIFORM

In this section we present Cuneiform. We show that it is simple to express data-parallel structures and to integrate external tools in Cuneiform. Furthermore, we demonstrate fundamental language features by example and discuss how Cuneiform workflows are evaluated and mapped to distributed compute resources for scheduling and execution.

Cuneiform is a Functional Programming (FP) language with implicit state. Cuneiform has in common with scientific workflow languages its light-weight, versatile FFI allowing users to directly use external tools or libraries from scripting languages including Lisp, Matlab, Octave, Perl, Python, and R. In principle, Cuneiform can interface with any programming language that has support a string and list data type. Cuneiform has in common with data-parallel query languages that it provides facilities to exploit data parallelism in the form of composite data types and algorithmic skeletons to process them. Cuneiform comes in the form of a universal FP language providing abstractions and conditionals.

In the following, we introduce important concepts of Cuneiform by example. We highlight the interplay of Cuneiform's features using more complex workflows in Section 4, while Section 5 briefly sketches the Hi-WAY execution environment.

3.1 Task definition and Foreign Function Interface

The *deftask* statement lets users define Cuneiform tasks, which are the same as functions in FP languages. It expects a task name and a prototype declaring the input/output variables a task invocation consumes/produces. A task definition can be either in Cuneiform or in any of the supported foreign scripting languages. In the following example we define a task *greet* in Bash which consumes an input variable *person* and produces an output variable *out*.

```
deftask greet( out : person )in bash *{
  out="Hello $person"
}*
```

The task defined in this listing can be applied by binding the parameter *person* to a value. In this example we bind it to the string "Peter".

```
greet( person: 'Peter' );
```

The value of this expression is the string "Hello Peter". Cuneiform assumes foreign tasks to be side effect-free. I.e., the result of a task should be deterministic and depend only on the value of its arguments. However Cuneiform has no way of enforcing this behavior.

3.2 Lists

Cuneiform has one built-in composite data type: the list. There is no atomic data type. In the following example, we define a variable *friends* to be a list of two strings being "Jutta" and "Peter".

```
friends = 'Jutta' 'Peter';
greet( person: friends );
```

Applying the function *greet* to this list, evaluates to a list with two string elements: "Hello Jutta" and "Hello Peter". Thus, the standard way of applying a task to a single parameter, is to map this task to all elements in the list.

To consume a list as a whole, we have to aggregate the list. We can mark a parameter as aggregate by surrounding it with angle brackets. The following listing defines the task *cat* that takes a list of files and concatenates them.

```
deftask cat
  ( out( File ) : <inp( File )> )in bash *{
  cat ${inp[@]} > $out
}*
```

When a list is aggregated in a foreign task call, Cuneiform has to hand over this list as a whole. Thus, Cuneiform loses control over the way data parallelism is exploited in processing this list. Furthermore, the interpreter has to defer the aggregating task unless the whole list has been computed.

3.3 Parallel algorithmic skeletons

Cuneiform provides three basic algorithmic skeletons: aggregate, n-ary cross product, and n-ary dot product. A map can be viewed as any unary product. These basic skeletons are the building blocks to form arbitrarily complex skeletons. If a task has multiple parameters, the standard behaviour is to apply the function to the cross product of all parameters.

Suppose there is a command line tool *sim* that takes a temperature in $°C$ and a pH value, performs some simulation, and outputs the result in the specified file. We could wrap and call this tool in the following way:

```
deftask simulate
  ( out( File ) : temp ph )in bash *{
  sim -o $out -t $temp -p $ph
}*

temp = -5 0 5 10 15 20 25 30;
ph = 5 6 7 8 9;

simulate( temp: temp ph: ph );
```

This script performs a parameter sweep from -5 to $30°C$ and from pH value 5 to 9. Herein, each of the 8 temperature values is paired with each of the 5 pH values resulting in 40 invocations of the *sim* tool. How multiple lists are combined is generally determined by the prototype of a task. The cross product is the default algorithmic skeleton to combine task parameters.

Lastly, suppose we are given two equally long lists of strings. We want to concatenate each string from the first list with each string from the second list separated by a white space character. A dot product between two or more parameters is denoted in the task prototype by surrounding them with square brackets. We choose to perform the string concatenation in Python.

```
deftask join( c : [a b] )in python *{
c = a+' '+b
}*
```

In the following listing we define the variables *a* and *b* to be a pair of two-element lists and call the previously defined task *join* on them. The result of this operation is a two-element list with the members "Hello world" and "Goodnight moon".

```
a = 'Hello' 'Goodnight';
b = 'world' 'moon';

join( a: a b: b );
```

3.4 Execution semantics

Cuneiform workflow scripts are parsed and transformed into a graph representation prior to interpretation. Variables are associated not with concrete values but with uninterpreted expressions thereby constituting a call-by-name evaluation strategy. Consequently, an expression is evaluated only if that expression is actually used (lazy evaluation). This ensures, not only, that all computation actually contributes to the result, but also, since evaluation is deferred to the latest possible moment, that parallelization is performed on the level of the whole workflow rather than the level of only subexpressions. Furthermore, instead of traversing the workflow graph during execution, Cuneiform performs workflow graph reduction. This means that subexpressions in the workflow graph are continuously replaced with what they evaluate until the result of the computation remains. Accordingly, workflow execution is dynamic, i.e., the order in which which tasks are evaluated is determined only at runtime, a model which naturally supports data dependent loops and conditions. This aspect discerns Cuneiform from many other systems that require a fixed execution graph to be compiled from the workflow specification. Herein, Cuneiform resembles Functional Programming language interpretation.

When an expression involves external software, a ticket is created and passed to the execution environment. Expressions depending on that ticket are deferred while other expressions continue to be evaluated. A ticket, encapsulating a concrete task on a concrete input, thus, is the basic computational unit in Cuneiform. For any given point in time, the set of available tickets may be evaluated in any order. This order has to be detemined by the scheduler of the runtime environment, taking into account the currently available resources. Each time the execution environment finishes evaluation of a ticket, the result is reported back to the Cuneiform interpreter which then continues reduction of the respective expression. When there are no more tickets to evaluate and the expression cannot be further reduced, execution stops and the workflow result is returned.

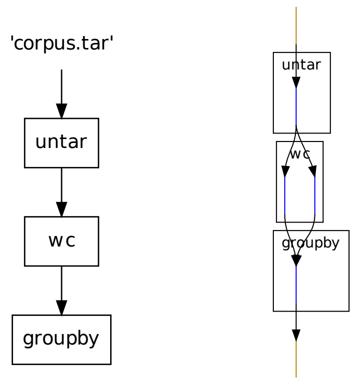

Figure 1: Static call graph (left) and invocation graph (right) for canonical word count with a corpus of 2 text files.

4. WORKFLOW EXAMPLES

In this section we present two example workflows in Cuneiform. The first, a canonical word count example, is chosen for its simplicity and comparability with other programming models (e.g., MapReduce [11]). The second workflow performs variant calling on Next-Generation Sequencing data [34].

4.1 Canonical word count

The canoncical word count workflow consumes a corpus of text files and, for each file, counts the occurrences of words. It outputs a table that sums up the occurrences of words in all files. The workflow consists of two steps. In the first step, words are counted individually in each file. In a second step, the occurrence tables are aggregated by summing up the corresponding occurrence counts. Figure 1 displays (a) the static call graph automatically derived from the workflow script and (b) the invocation graph that unfolds during workflow execution. Herein, the static call graph is a visualization that takes into account only the unevaluated workflow script. In contrast, the invocation graph is derived from the workflow execution trace. Each yellow line in the invocation graph stands for a single data item. Each blue line stands for a task invocation. A task invocation depends only on its predecessors connected to it via directed edges.

To specify the word count workflow we express both tasks separately as R scripts. First, we use R's *table* function to extract word counts from a string:

```
deftask wc( csv( File ) : txt( File ) )in r *{
  dtm <- table( scan( txt, what='character' ) )
  df  <- as.data.frame( dtm )
  write.table( df, csv, col.names=FALSE,
    row.names=FALSE )
}*
```

Next, we use the function *rbind* to concatenate the list of tables, generated in the previous step and aggregate the resulting table using *ddply* which is part of the R library *plyer*.

```
deftask groupby
  ( result( File ) : <csv( File )> )in r *{

  library( plyr )
  all <- NULL
  for( i in csv )
    all <- rbind( all,
             read.table( i, header=FALSE ) )
  x <- ddply( all, .( V1 ), summarize,
         count=sum( V2 ) )
  write.table( x, result, col.names=FALSE,
    row.names=FALSE )
}*
```

To extract all files in an archive holding the text corpus to be analyzed we use the following task definition:

```
deftask untar
  ( <list( File )> : tar( File ) )in bash *{
  tar xf $tar
  list=`tar tf $tar`
}*
```

The workflow definition calls the tasks *untar*, *wc*, and *groupby* in order. Finally, we query the workflow result:

```
txt = untar( tar: 'corpus.tar' );
csv = wc( txt: txt );
result = groupby( csv: csv );
result;
```

Called this way, *wc* is invoked once for each file. Each invocation is processed in parallel by Hi-WAY. In contrast, the tasks *groupby* and *untar* each have a single invocation.

Note that the two tasks, *wc* and *groupby*, implement a complete word count, including file I/O, parsing, dictionary management, and two-phase counting. No other tools are needed. Furthermore, we are free to choose the programming language. For instance, in a different implementation we might use Perl libraries or the command line tool *awk*.

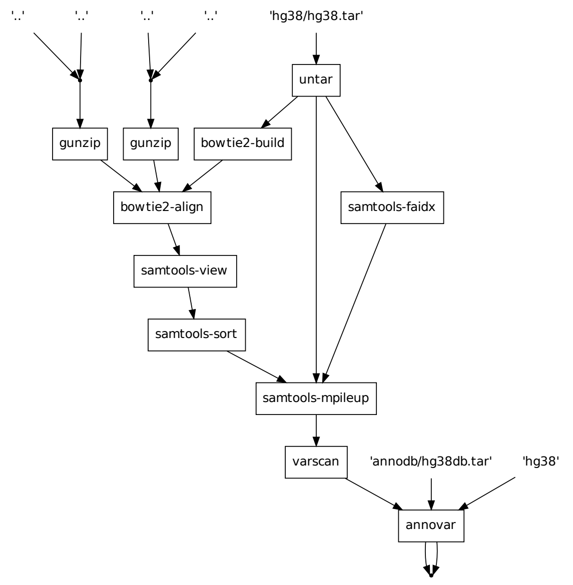

Figure 2: Static call graph for variant calling workflow

4.2 NGS variant calling

The second workflow demonstrates how variant calling in the application domain of Next-Generation Sequencing (NGS) can be performed in Cuneiform. In this workflow, a set of DNA sequence read files in FastQ format is mapped against a reference genome. Subsequently, the alignments are sorted, a multiple pileup is performed, and variants are called and annotated. As typical for scientific analysis pipelines, all steps are performed by external command line tools [34]. Figure 2 shows the static call graph and Figure 3 shows the invocation graph for this workflow. In the following discussion we omit all foreign task definitions.[3]

The input to the workflow is a reference genome, a set of sample files, as well as an annotation database. The workflow calls two nested subworkflows *per-sample* and *per-chromosome* which reflect the data parallelization scheme. Up to this point, we defined only variable assignments which would not trigger any computation. Thus, we need to query the variables *fun* and *exonicfun* to define the workflow output.

[3]The full workflow can be downloaded from https://github.com/joergen7/cuneiform/blob/master/cuneiform-dist/src/main/cuneiform/variant-call11.cf

```
hg38-tar = 'hg38/hg38.tar';

fastq1-gz =
  '1000genomes/SRR062634_1.filt.fastq.gz'
  '1000genomes/SRR062635_1.filt.fastq.gz';
fastq2-gz =
  '1000genomes/SRR062634_2.filt.fastq.gz'
  '1000genomes/SRR062635_2.filt.fastq.gz';

db = 'annodb/hg38db.tar';

deftask per-chromosome(
    vcf( File )
  : fa( File )
    [fastq1( File ) fastq2( File )] ) {

  bt2idx = bowtie2-build( fa: fa );
  fai = samtools-faidx( fa: fa );

  sam = bowtie2-align(
    idx:    bt2idx
    fastq1: fastq1
    fastq2: fastq2 );

  bam = samtools-view( sam: sam );

  sortedbam = samtools-sort( bam: bam );

  mpileup = samtools-mpileup(
    sortedbam: sortedbam
    fa:        fa
    fai:       fai );

  vcf = varscan( mpileup: mpileup );
}

deftask per-sample(
    fun exonicfun
  : <fa( File )> db( File )
    [fastq1( File ) fastq2( File )] ) {

  vcf = per-chromosome(
    fa:     fa
    fastq1: fastq1
    fastq2: fastq2 );

  fun exonicfun = annovar(
    vcf:      vcf
    db:       db
    buildver: 'hg38' );
}

fa = untar( tar: hg38-tar );
fastq1 = gunzip( gz: fastq1-gz );
fastq2 = gunzip( gz: fastq2-gz );

fun exonicfun = per-sample(
  fa:     fa
  fastq1: fastq1
  fastq2: fastq2
  db:     db );
fun exonicfun;
```

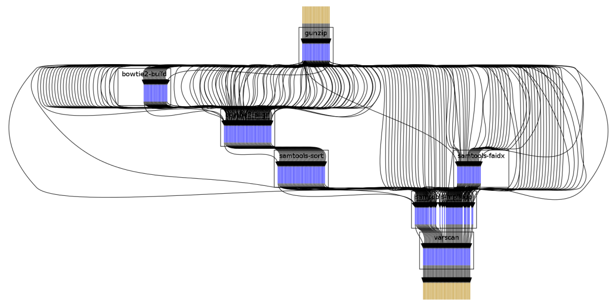

Figure 3: Invocation graph for variant calling workflow

In this workflow parallelism is exploited along two dimensions: (i) Each self-contained region in the reference genome can be processed individually and (ii) each sample can be processed individually. Consequently, the workflow interpreter performs a cross-product of reference regions and samples. This leads to a high degree of parallelism not only for read alignment, which is the computationally most expensive task, but also for all subsequent tasks. The cross product behavior needs no extra denotation in the task prototypes since it is the default behavior. Thus, we can exploit data parallelism in variant calling to execute the workflow in a parallel, distributed compute environment while reusing established tools.

5. EXECUTION PLATFORM

In this section we describe Hi-WAY, an execution environment for Cuneiform workflows. Cuneiform, as a workflow specification language, depends on an execution environment that executes tasks in parallel. To this end, the Cuneiform interpreter can either execute a script on a single, multi-threaded machine (using a simple built-in greedy task scheduler) or feed a distributed workflow engine. Currently, it interfaces only with Hi-WAY, a novel scientific workflow management system running on top of Apache Hadoop. Hi-WAY offers features like adaptive scheduling and, this way, embraces the dynamic nature of Cuneiform workflows. By using Hi-WAY as its distributed execution environment, Cuneiform takes advantage of the Hadoop ecosystem, including the distributed file system HDFS, multi-user resource management, job monitoring, and failure recovery. Details on Hi-WAY will be published in a separate publication.

As a proof-of-concept, the variant calling workflow described in Section 4.2 has been executed using Cuneiform and Hi-WAY on a Hadoop YARN cluster comprising 24 Xeon E52620 2GHz nodes each representing one Hadoop YARN container with 24GB main memory and 24 logical cores at its disposal (as well as 2 additional master nodes). 12 Samples from the 1000

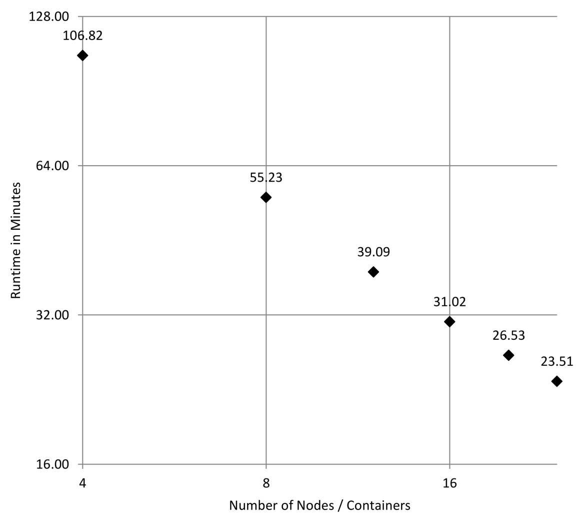

Figure 4: Workflow runtime with increasing number of containers

genomes project [38] amounting to 10GB of compressed input data have been processed. Figure 4 shows the runtime behaviour for the variant calling workflow for different cluster sizes. Within the limits of this experiment the workflow shows a linear scaling behaviour with an increasing number of available containers.

6. RELATED WORK

A number of scientific workflow systems have emerged, some with a particular focus on large scale data analysis. The exponential growth of data sets in many scientific areas, such as Next-Generation Sequencing (NGS), promotes scientific workflow systems that run in parallel and distributed environments, like e-Science Central [20], Pegasus [13], or Swift [42]. Some scientific workflow systems have been extended to this end, e.g., Kepler [41] or Galaxy [17]. These systems, however, either do not take full advantage of partitioning input data to exploit data parallelism or their integration with data-parallel compute platforms is only partial.

The advent of data-parallel query languages enabled researchers to exploit parallel, distributed compute infrastructures to analyze large-scale data sets. A number of dataflow systems with their according query languages have been proposed, most notably Pig [16, 33], FlumeJava [7], Flink [1], DryadLINQ [14, 43], and Spark [44, 45]. Their aptitude for NGS problems has been assessed [46] and they are the underlying execution environments for a number of emerging workflow systems like Nova [32] or Oozie [22]. However, the integration of external tools in these systems can be achieved only through wrapping or reimplementing the external tools. The speed-up potential from data parallelism has been exploited in many scientific application domains and particularly in NGS: CloudBurst [37] is a read alignment implementation for Hadoop. Crossbow [25] wraps the read aligner Bowtie [24] to run on Hadoop. Both Adam [27], an alignment processor, and Avocado [30], a variant caller, are algorithm reimplementations for Spark. The BioPig [29] project extends Pig Latin by providing User Defined Functions (UDFs) that wrap tools commonly used in NGS data analysis. These approaches show that it is feasible to integrate diverse scientific algorithms in data-parallel programming models either through wrapping or reimplementing. However, in use cases in which the cost for tool reimplementation is prohibitive (and in which the scientific community is very reluctant to accept algorithm reimplementations from outside their domain), the optimal programming model is one that minimizes the effort to create wrappers for existing tools.

Scientific workflow systems and data-parallel query languages are linked to Functional Programming (FP). Pig Latin maps execution plans to MapReduce, a programming model inspired by the algorithmic skeletons map and reduce which originate from FP [11, 15]. Spark extends Scala, a multi-paradigm language that combines concepts from Object Orientation and FP [31]. Furthermore, Scala provides a large number of algorithmic skeletons, of which Spark uses a subset including map, reduceByKey, and crossProduct to derive parallelism and distribute computation [44]. The scientific workflow language Taverna has its semantics defined in functional terms [40] and Kelly et al. [23] showed that scientific workflow languages can be considered a subset of FP. A number of FP languages are designed for parallel, distributed environments. For instance, Skywriting is a universal functional scripting language for distributed computation [28]. GUM [39] is a parallel implementation of Haskell and Eden [5, 26] extends Haskell with parallel algorithmic skeletons. However, in many parallel, distributed FP languages the user has to take control over the way parallelism is exploited, how processes are created, or how computation is distributed.

7. CONCLUSION

We presented Cuneiform[4], a functional workflow language for parallel and distributed execution that facilitates the reuse of existing tools and libraries. Cuneiform can process large-scale data sets by providing data parallel algorithmic skeletons operating on lists. Furthermore, it can integrate foreign tools in a straightforward way by providing a versatile Foreign Function Interface and offers many of the high-level language features commonly encountered in Functional Programming languages. We have contrasted the advantages and disadvantages of current scientific workflow languages and data-parallel query languages and discussed their relation to

[4]https://github.com/joergen7/cuneiform

Functional Programming. We demonstrated the versatility and power of Cuneiform using two exemplary workflows. In its current implementation Cuneiform can be executed locally on a single machine or using Hi-WAY[5], a scientific workflow execution environment running on Hadoop YARN. In future work, we intend to integrate Cuneiform with scientific computing platforms other than Hadoop like, e.g, HTCondor [9] which enjoys wide adoption. Furthermore, we intend to create compilers that consume Pegasus or Galaxy workflows and generate Cuneiform scripts. This way, researchers may run their existing Pegasus and Galaxy workflows in any data-parallel execution environment supporting Cuneiform without extra effort.

8. ACKNOWLEDGEMENTS

Jörgen Brandt and Marc Bux are funded by the European Commission's 7th Framework Programme (FP7) through the BiobankCloud project (project no. 317871).

9. REFERENCES

[1] http://flink.incubator.apache.org/.

[2] E. M. Bahsi, E. Ceyhan, and T. Kosar. Conditional workflow management: A survey and analysis. *Scientific Programming*, 15(4):283–297, 2007.

[3] M. Baker. Next-generation sequencing: adjusting to data overload. *Nature Methods*, 7(7):495–499, 2010.

[4] M. R. Berthold, N. Cebron, F. Dill, T. R. Gabriel, T. Kötter, T. Meinl, P. Ohl, C. Sieb, K. Thiel, and B. Wiswedel. Knime: The konstanz information miner. In *Studies in Classification, Data Analysis, and Knowledge Organization*, 2007.

[5] S. Breitinger, U. Klusik, and R. Loogen. From (sequential) haskell to (parallel) eden: An implementation point of view. In *Principles of Declarative Programming*, pages 318–334. Springer, 1998.

[6] M. Bux and U. Leser. Parallelization in scientific workflow management systems. *Computing Research Repository (CoRR)*, 2013.

[7] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. Flumejava: Easy, efficient data-parallel pipelines. *SIGPLAN Not.*, 45(6):363–375, 2010.

[8] M. I. Cole. *Algorithmic skeletons: structured management of parallel computation.* Pitman London, 1989.

[9] P. Couvares, T. Kosar, A. Roy, J. Weber, and K. Wenger. Workflow management in condor, 2007.

[10] V. Curcin and M. Ghanem. Scientific workflow systems-can one size fit all? In *Cairo International Biomedical Engineering Conference (CIBEC)*, pages 1–9, 2008.

[11] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[12] E. Deelman, D. Gannon, M. Shields, and I. Taylor. Workflows and e-science: An overview of workflow system features and capabilities. In *Future Generation Computer Systems*, 2008.

[13] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. Berriman, J. Good, A. Laity, J. Jacob, and D. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13:219–237, 2005.

[14] J. Ekanayake, T. Gunarathne, G. Fox, A. S. Balkir, C. Poulain, N. Araujo, and R. Barga. Dryadlinq for scientific analyses, 2009.

[15] J. Ekanayake, S. Pallickara, and G. Fox. Mapreduce for data intensive scientific analyses. In *IEEE Fourth International Conference on eScience*, pages 277–284, 2008.

[16] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a high-level dataflow system on top of map-reduce: the pig experience. *Proc. VLDB Endow.*, 2(2):1414–1425, 2009.

[17] J. Goecks, A. Nekrutenko, J. Taylor, and T. G. Team. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome Biology*, 11(8), 2010.

[18] R. R. Gullapalli, K. V. Desai, L. Santana-Santos, J. A. Kant, and M. J. Becich. Next generation sequencing in clinical medicine: Challenges and lessons for pathology and biomedical informatics. *Journal of pathology informatics*, 3, 2012.

[19] A. Heise, A. Rheinländer, M. Leich, U. Leser, and F. Naumann. Meteor/sopremo: An extensible query language and operator model. In *International Workshop on End-to-end Management of Big Data*, 2012.

[20] H. Hiden, P. Watson, S. Woodman, and D. Leahy. *e-Science Central: Cloud-based e-Science and its application to chemical property modelling.* Newcastle University, Computing Science, 2010.

[21] D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. Pocock, P. Li, and T. Oinn. Taverna: a tool for building and running workflows of services. *Nucleic Acids Research*, 34:729–732, 2006.

[22] M. Islam, A. K. Huang, M. Battisha, M. Chiang,

[5]https://github.com/marcbux/Hi-WAY

S. Srinivasan, C. Peters, A. Neumann, and A. Abdelnur. Oozie: towards a scalable workflow management system for hadoop. In *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, 2012.

[23] P. M. Kelly, P. D. Coddington, and A. L. Wendelborn. Lambda calculus as a workflow model. *Concurrency and Computation: Practice and Experience*, 21(16):1999–2017, 2009.

[24] B. Langmead, M. C. Schatz, J. Lin, M. Pop, and S. L. Salzberg. Searching for snps with cloud computing. *Genome Biology*, 10(11), 2009.

[25] B. Langmead, C. Trapnell, M. Pop, S. L. Salzberg, et al. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biology*, 10(3), 2009.

[26] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel functional programming in eden. *Journal of Functional Programming*, 15(03):431–475, 2005.

[27] M. Massie, F. Nothaft, C. Hartl, C. Kozanitis, A. Schumacher, A. D. Joseph, and D. A. Patterson. Adam: Genomics formats and processing patterns for cloud scale computing. Technical report, EECS Department, University of California, Berkeley, 2013.

[28] D. G. Murray and S. Hand. Scripting the cloud with skywriting. *Proceedings of HotCloud*, (3), 2010.

[29] H. Nordberg, K. Bhatia, K. Wang, and Z. Wang. Biopig a hadoop-based analytic toolkit for large scale sequence data. *Bioinformatics*, 29(23):3014–3019, 2014.

[30] F. A. Nothaft, P. Jin, and B. Brown. avocado: A variant caller, distributed. Technical report, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 2013.

[31] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the scala programming language. Technical report, École Polytechnique Fédérale de Lausanne, 2004.

[32] C. Olston, G. Chiou, L. Chitnis, F. Liu, Y. Han, M. Larsson, A. Neumann, V. B. N. Rao, S. Seth, C. Tian, T. Zicornell, and X. Wang. Nova: Continuous pig/hadoop workflows. In *SIGMOD*, 2011.

[33] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, 2008.

[34] S. Pabinger, A. Dander, M. Fischer, R. Snajder, M. Sperk, M. Efremova, B. Krabichler, M. R. Speicher, J. Zschocke, and Z. Trajanoski. A survey of tools for variant analysis of next-generation genome sequencing data. *Briefings in bioinformatics*, 15(2):256–278, 2014.

[35] E. Pennisi. Will computers crash genomics? *Science*, 331(6018):666–668, 2011.

[36] P. V. Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming.* MIT Press, 2004.

[37] M. C. Schatz. Cloudburst: highly sensitive read mapping with mapreduce. *Bioinformatics*, 25(11):1363–1369, 2009.

[38] N. Siva. 1000 genomes project. *Nature biotechnology*, 26(3):256–256, 2008.

[39] P. W. Trinder, K. Hammond, J. S. Mattson Jr, A. S. Partridge, and S. Peyton Jones. Gum: a portable parallel implementation of haskell. In *ACM SIGPLAN Notices*, volume 31, pages 79–88, 1996.

[40] D. Turi, P. Missier, C. Goble, D. De Roure, and T. Oinn. Taverna workflows: Syntax and semantics. In *IEEE International Conference on e-Science and Grid Computing*, pages 441–448, 2007.

[41] J. Wang, D. Crawl, and I. Altintas. Kepler + hadoop: A general architecture facilitating data-intensive applications in scientific workflow systems. In *Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*, 2009.

[42] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9):633–652, 2011.

[43] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, pages 1–14, 2008.

[44] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, 2012.

[45] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. *HotCloud*, 2010.

[46] Q. Zou, X.-B. Li, W.-R. Jiang, Z.-Y. Lin, G.-L. Li, and K. Chen. Survey of mapreduce frame operation in bioinformatics. *Briefings in bioinformatics*, 2013.