SciMedCentral

JSM Computer Science & Engineering

Editorial

Functional Representations of Scientific Workflows

Noe Lopez-Benitez*

Department of Computer Science, Texas Tech University, USA

***Corresponding author**
Noe Lopez-Benitez, Department of Computer Science, Texas Tech University, Lubbock, Texas, 79409-3104, USA, Email: noe.lopez-benitez@ttu.edu

Submitted: 16 December 2013

Accepted: 17 December 2013

Published: 18 December 2013

Copyright
© 2014 Lopez-Benitez

OPEN ACCESS

EDITORIAL

Workflows describe not only a collection of component functions, but also their dependencies, which predefine a constrained order of execution. Scientific workflows are used to describe not only computational and service requirements but also the location of such services or computational units. Instruments in scientific laboratories, robots in remote inaccessible areas, a satellite unit in outer space, a set of databases, storage units as well as computational units, all provide services that must be orchestrated to satisfy an overall scientific objective. In this paper functional representations of workflows are discussed as a convenient alternative abstractions that can provide the basis for dynamic management of service requests; furthermore they can be regarded as a paradigm to organize and easily develop entire applications via the use of functional languages to explore not only fine-grained parallelisms, but also functional dynamic parallelisms suitable for grid and cloud execution environments.

Functional representations are proposed in the context of functional languages such as Haskell [1], Parallel Haskell [2], SequenceL [3], and others. Functional languages provide users the ability to specify and/or generate possible parallel operations for fine-grained computational platforms. Haskell has been extended to Cloud Haskell to provide message-passing support [4]. Translation of functional code may lead to fine-grain parallelism; sequenceL, for example, generates n-tuples of independent computations that can be mapped into multiple independent threads of execution; consequently, fine-grain parallelisms can be mapped into high-level languages such as MCUDA suitable for multi-core execution platforms.

Background Work

Abstract-to-concrete workflow is a transformation that prevails in a cloud environment [5], where an orchestration process completes the mapping into a concrete web-based executable workflow. In [6] a series of workflow transformations referred to as '*sequence', 'and split*' and '*and join'* patterns lead to single node reductions. These transformations provide the basis for the reduction schemes discussed in this paper as applied to functional representations. The grouping of tasks reported in GridSolve [7] is intended to minimize transfer delays by either having a multiple-resource site execute all tasks or supporting the transfer of data between different parallel tasks executing in different service sites. Furthermore, fundamental work has been reported on workflow optimization for grid environments, on scheduling parallel clusters through Condor in [8], on schedule-based workflow balancing [9], on performance and overhead of high-performance applications [10], on task clustering for balanced workflows in [11] task clustering is of interest because the functional representation models described in this paper are based on partitioning an entire workflow into sub-workflows, similar to the heuristics reported in [12] and complemented with the integration of resource provisioning as reported in [13].

Functional Representation of Workflows

A workflow, used as a typical directed acyclic graph (DAG) is described by a set of vertices representing individual tasks and a set of edges representing data dependencies. It is possible to regard each task in a workflow as a functional unit depending on the execution of its predecessors. For example, if a task *A* is followed by independent tasks *B* and *C* in the workflow, then a notation *[B,C]A* will model not only such dependencies but also indicates that tasks B and C can be executed in parallel. Using this representational approach, a functional description of a workflow *W* can be described as:

$$W = [T_{F1}[..], T_{F2}[...], \ldots, T_{Fn}[...]] \quad (1)$$

Where $T_{Fi}[...]$ represents a collection of nodes describing a path of dependent functions. If we let T_{Fi} represent a terminal node (function) in the workflow, then each $T_{Fi}[...]$ in equation (1) can be described as:

$$T_{Fi}[T_x[T_y[\ \ldots\ ]\ldots]],\ i=1,\ldots,n$$

Where T_x and T_y are nodes in one of the execution paths leading to terminal node T_{Fi}. Thus, each node in the workflow is expressed as a function of all previous nodes in its execution path. The collection of functions forms a queue of dependent functions that must be executed in sequence. Each queue structure includes an initial task i.e., with no incoming arcs, identified as a root task. A set of root tasks corresponds to independent tasks that can be dynamically scheduled for a parallel execution [14]. Consider for example the workflow shown in [Figure. 1a]. A functional representation can be derived from the workflow shown in Fig. 1b, where each node is expressed as a function of all previous computing nodes in its path. To maintain the order of execution each terminal node tails each queue derived from [Figure.1b] which, following a functional representation, can be expressed as follows:

W = [[F[D[B[A]]], G[D[B[A]],E[B[A]], C[A]]] (2)

Removing the square brackets from equation (2) then a set of queues given by all paths in the workflow are shown:

{FDBA, GDBA, GEBA, GCA}

The right most function in each queue corresponds to a "root" function and the tail function corresponds to a terminal node in the workflow. These structures can be easily obtained applying well-known depth-first search algorithms.

Systematic Partitioning of Workflows

Extracting sets of root functions leads to possible partitions of the original workflow. Consider for example the following partitions from the set of queues generated for the workflow in (Figure 1a)

1. { FDB, GDB, GEB, GC\} [A]
2. { FD, GD, GE \} [B,C][A]
3. [F, G][D, E][B, C][A]

Partition 1 is formed by extracting root *A* to the right. The set of roots *{B, C}* are shown as heads of the remaining set of queues and are extracted to the right as two parallel functions as shown in partition 2. Likewise the sequence shown in partition 3 shows sets of functions that are unique, i.e., no function is contained in any other set. Identifying partitions of a workflow composed of sequential and/or parallel patterns in a workflow may lead to a reduced representation without altering the functionality of the original workflow.

The notation used this far describes parallel functions separated by commas; otherwise, a sequential execution is indicated. Using square brackets enforces serialization. As illustrated, to preserve the dependencies explicit in each queue, root functions are always extracted to the right. Also, if different roots are extracted from different queues they are indicated as a parallel structure.

The following rules are intended to formalize the manipulation of workflows and obtain alternate representations whenever possible. The general description of sequential and parallel patterns assumes that x_i , $i = 1, ..., n$, identifies the ith node in a pattern with *n* number of nodes (functions).

Extraction Rules

Extracting common functions leads to the generation of parallel structures amenable to the reduction rules discussed this far.

1. Right Extraction Rule: This rule extracts a common function or a common composition to the right. Extracting a common function *y* results in a parallel composition in which *y* becomes a root node that precedes the parallel execution of all nodes x_i:

$$\{x_n y, ..., x_2 y, x_1 y\} = \{ [x_n, ..., x_2, x_1] y\}$$

Extracting an entire composition to the right leads to a sequence of two parallel compositions.

2. Left Extraction Rule: Extracting a node *y* to the left results in a parallel composition in which *y* is a terminal node following the parallel execution of all x_i nodes:

$$\{ yx_n , ... , yx_2, yx_1\} = \{ y[x_n, ..., x_2, x_1]\}$$

Extracting an entire composition to the left also leads to a sequence of two parallel compositions.

Sequential Composition Rules

This composition describes a functional description of a workflow in which all nodes x_1 to x_n must be executed in sequence $x_n[x_{n-1} ... x_2 x_1]$; in this pattern x_n is the terminal function and it is dependent on the sequential execution of all the functions in the set $\{x_{n-1}, ..., x_2 , x_1\}$. The following rules apply to sequential embedded patterns:

3. Sequential Reduction Rule 1: A sequential composition of functions in the set $\{x_n, ..., x_2, x_1\}$ can be embedded in a functional representation as follows:

$$x_n[x_{n-1} ... x_2 x_1], x_n[y]$$

where $y \notin \{x_n, ..., x_2, x_1\}$. Then a reduction functionally equivalent is of the form:

$$x_n[x_{n-1} ... x_2 x_1, y]$$

4. Sequential Reduction Rule 2: This rule applies if the sequential composition of the set $\{x_n, ..., x_2, x_1\}$ is embedded in a functional representation of the form:

$$x_n[x_{n-1} ... x_2 x_1], y[x_{n-1} ... x_2 x_1]$$

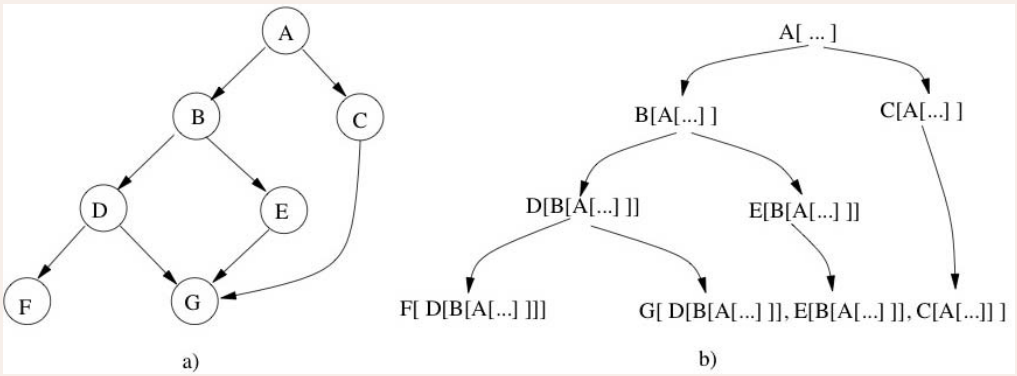

Figure 1 A workflow and its functional representation derivation.

which can be reduced to the functionally equivalent form:

$[x_n, y][x_{n-1} \ldots x_2 x_1]$

Parallel Composition Rules

A parallel-to-series pattern transformation to transform and reduce embedded parallel patterns. Using a parallel functional expression can be transformed into a series expression indicated as follows:

$[x_n, \ldots, x_2, x_1] = x_n \ldots x_2 x_1$

A parallel pattern can appear in two forms. A *join* or a *fork* structure. A *join* structure is identified in the following form:

$y\,[x_n, \ldots, x_2, x_1]$

In this composition y is a terminal node that depends on the parallel execution of all functions in the set $\{x_n, \ldots, x_2, x_1\}$. Extracting to the left a common function results in a common n-degree node y which defines an embedded *join* structure.

The following pattern identifies a *fork* composition:

$[x_n, \ldots, x_2, x_1]\,y$

This composition shows that the set of parallel functions $\{x_n, \ldots, x_2, x_1\}$ can be executed but only after node y (which is not in the parallel set) executes successfully.

5. Parallel Fork Composition Rule: A functional representation given as follows:

$[x_n, \ldots, x_2, x_1]x_i, x_k y$

contains and embedded fork composition $x'_i = [x_n, \ldots, x_2, x_1]\,x_i$. If $x_k \in \{x_n, \ldots, x_2, x_1\}$, and $y \notin \{x_n, \ldots, x_2, x_1\}$ then the following reduced sequential compositions are functionally equivalent:

$[x_n, \ldots, x_2, x_1]\,[x_i, y] = [x_n, \ldots, x_2, x_1]\,x_i y$

6. Parallel Join Composition Rule: If a functional description of a workflow is given as follows:

$x_i\,[x_n, \ldots, x_2, x_1], [y\,x_k]$

Then for any pair $y \notin \{x_n, \ldots, x_2, x_1\}$ and $x_k \in \{x_n, \ldots, x_2, x_1\}$, the following sequential compositions are functionally equivalent:

$[y, x_i][x_n, \ldots, x_2, x_1] = y x_i][x_n, \ldots, x_2, x_1]$

To illustrate the application of these rules consider the workflow shown in (Figure 2a). The set of paths for this workflow are given as follows:

{ FCA, FDA, FDB, HDA, HDB, HEB }

Extracting to the right the initial two roots results in the following partition:

{ FC, FD, FD, HD, HD, HE } [A, B]

Extracting each time the root functions from the remaining queues, the following sequence is reached:

[F. H] [C,D,E][A,B]

This partition set corresponds to three parallel compositions shown in (Figure 2a). Alternatively, the first two parallel compositions can be merged into a single partition as shown

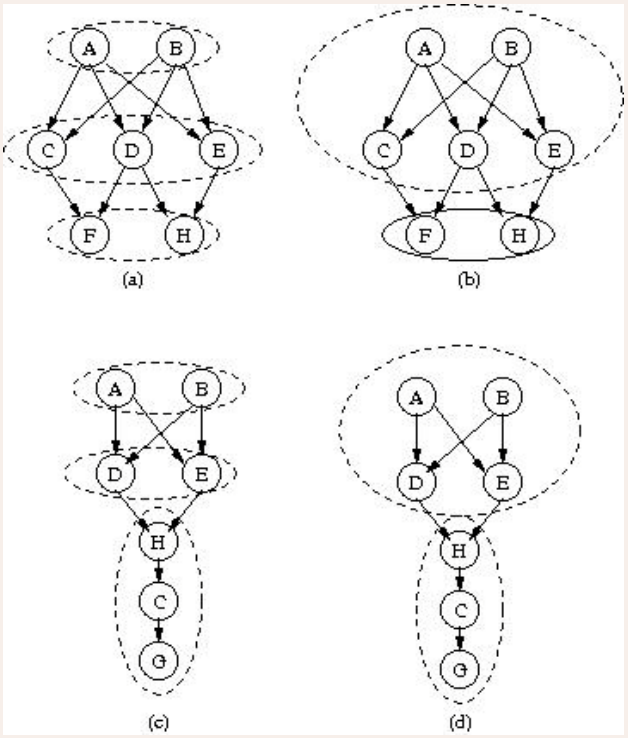

Figure 2 Illustration of workflow partition rules. a) A workflow with a sequence of three parallel compositions, b) with a sequence of two parallel compositions, c) with a sequence of three compositions, two are parallel compositions, the third one is a series composition, and d) with a sequence of two compositions with reduced data transfers.

(Figure 2b). Using square brackets to separate different partitions then:

[F. H] [C,D,E][A,B] = [F. H] [[C,D,E][A,B]]

Applying the reduction rules described previously, alternate partitions can be derived as follows:

{ FC, FD, FD, HD, HD, HE } = {FC, FD, H[D, E]}

= { F[C,D], H[D, E] }

= { FCD, H[D,E] }

= { FCH [D,E] }

The overall expressions are shown in (Figure 2c) and [Figure 2d] which correspond to the following functionally equivalent representations:

FCH [D,E] [A,B] = FCH [[D,E] [A,B]]

CONCLUSIONS

This paper illustrates the feasibility of generating alternate representations of workflows. Partitions are derived based on a functional representation of the original workflow to which a set of reduction rules are systematically applied. Each partition corresponds to a sub-workflow represented by a composition in the functional representation. Each composition therefore can be orchestrated for submission and execution in a grid or a cloud environment. Resource provisioning can be functionally integrated for each composition. As each sub-workflow demands

fewer resources for a shorter amount of time, a particular set of submissions can be optimized to require less data transfers, or to seek a balanced computational and data exchange requirements. The workflow partitioning heuristics reported in [12] and the integration of resource provisioning reported in [13] provide a context in which the partitioning rules discussed in this paper can be useful. In addition, a functional representation addresses coarse levels of granularity present in small (desktop) applications written using a functional language or any modern language with functional expressiveness.

REFERENCES

1. Hudak P, Hughes J, Jones S P, Wadler P. A History of Haskell: Being Lazy with Class, The Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III). 2007.
2. Marlow S, Parallel. Concurrent Programming in Haskell. version 1.2, Mircrosoft Research Ltd., Cambridge, U.K.. 2012.
3. Cooke D E, Rushton N J, Nemanich B, Watson R G, Andersen P. Normalize, Transpose, and Distribute: An Automatic Approach for Handling Nonscalars. ACM Transactions on Programming Language Systems. 2008.
4. Epstein J, Black A P, Peyton-Jones S. Towards Haskell in the Cloud}, ACM Haskell'11. 2011.
5. Juve G, Deelman E, Cafaro M, Alisio G. Scientific Workflows in the Cloud, in Grids, Clouds and Virtualization, Compute Communications and Networks. Springer-Verlag. 2011; 71-91.
6. Jaeger, M.C. Rojec-Goldmann, G. Muhl, G. QoS Aggregation in Web Service Compositions, e-Technology, e-Commerce and e-Service. 2005; 181:185.
7. Li Yinan, YarKhan Asim, Dongarra Jack, Seymour Keith, and Hurault Aurèlie, Enabling workflows in GridSolve: request sequencing and service trading, the Journal of Supercomputing, 2013; 3:1133-1152.
8. Singh G, Kesselman C, Deelman E, Optimizing Grid-Based Workflow Execution, Journal of Grid Computing 2006; 3:201-219.
9. Rajakumar S, Arunachalam V P, Selladurai V. Workflow Balancing Strategies in Parallel Machine Scheduling, International Journal for Advanced Manufacturing Technology. 2004; 366-374.
10. Mehrotra P, Djomehri J, Heistand S, Hood R, Jin H, Lazanoff A. Performance Evaluation of Amazon Elastic Compute Cloud for NASA High-performance Computing Applications, Concurrency and Computation Practice and Experience. 2013.
11. Chen W, Ferreira da Silva R, Deelman E, Sakellariou R. Balanced Task Clustering in Scientific Workflows, 9th International Conference on eScience, Beijin. China. 2013.
12. Chen W, Delman E. Partitioning and Scheduling Workflows across Multiple Sites with Storage Constraints, Workshop on Scheduling for Parallel Computing, 9th Intl. Conf. on Parallel Processing and Applied Mathematics. 2011.
13. Chen W, Delman E, Integration of Workflow Partitioning and Resource Provisioning, 2012 12th IEEE/ACM International Symposium on Cluster. Cloud and Grid Computing. 764-768.
14. Lopez-Benitez N, Andersen P, Dynamic Structures for the Management of Complex Applications in Grid Environments, Proceedings of the 2009 International Conference on Grid Computing and Applications. 2009; 80-85.

Cite this article

Lopez-Benitez N (2014) Functional Representations of Scientific Workflows. Comput Sci Eng 1(1): 1001.