CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2015; **00**:1–16

Towards an Autonomous Decentralised Orchestration System

Ward Jaradat[†], Alan Dearle, and Adam Barker

School of Computer Science, University of St Andrews, North Haugh, St Andrews, Fife, KY16 9SX, United Kingdom

SUMMARY

Orchestrating workflows needed for modern scientific data analysis presents a significant research challenge: they are typically executed in a centralised manner such that all data pass through a single compute server known as the engine, which causes unnecessary network traffic that leads to a performance bottleneck. This paper presents a scalable decentralised orchestration system that relies on a functional, high-level data coordination language for executing workflows. This system consists of distributed execution engines, each of which is responsible for executing part of the overall workflow. It exploits parallelism in the workflow by partitioning it into smaller sub workflows, and determines the most appropriate engines to execute them using network resource monitoring and placement analysis. This permits the computation logic of the workflow to be moved towards the services providing the data, which improves the overall execution time. The system supports data-driven execution that allows each sub workflow to be executed as soon as the data needed for its execution becomes available from other sources. Therefore, a scheduling mechanism is not required to manage the order in which the sub workflows are orchestrated. This paper provides an evaluation of the proposed system, which demonstrate that decentralised orchestration provides scalability over centralised orchestration. Copyright © 2015 John Wiley & Sons, Ltd.

Received 28 February 2015; Revised 8 July 2015

KEY WORDS: Service-oriented architecture; decentralised orchestration; data-centric workflows; partitioning; network resource monitoring; placement analysis

1. INTRODUCTION

Service computing encompasses techniques for designing and implementing web services, based on a suite of standards that enables communication between web services and facilitates their interoperability. These standards include the *eXtensible Markup Language* (XML) [1], *Simple Object Access Protocol* (SOAP) [2], *Web Services Description Language* (WSDL) [3], and *Representational State Transfer* (REST) [4] technology. However, these standards are insufficient to compose services where each service has a role in a larger collaboration known as a workflow.

Service-oriented workflows can be defined as the automation of services during which data is passed between the services for processing according to a set of procedural rules, whereas a generic definition of a workflow can be found in [5]. Many researchers have studied the design and orchestration of workflows [6, 7, 8]. Typically, workflows are designed by composing services together using a workflow language such as *Business Process Execution Language* (BPEL) [9], and orchestrated using a centralised compute server known as the execution engine. Centralised orchestration provides total control over the workflow, supports process automation, and allows the workflow logic to be encapsulated, modified or extended as necessary at a single location [10]. The *Simple Conceptual Unified Flow Language* (SCUFL) is another example of a language for specifying data-centric workflows such as those seen in scientific applications [11]. It is supported

[†]E-mail: ward.jaradat@st-andrews.ac.uk

by the *Taverna* workbench and is typically executed using a workflow engine known as *Freefluo* [12]. However, the service interactions result in inter-process communication in which the output of a service process is passed to another as an input through the centralised engine. This causes the engine to become a performance bottleneck as all data and control flows are passed through it.

Many modern scientific challenges require the integration of data-centric services, which may be globally distributed and provided by different institutions [13]. Scientists require the ability to access, compose, and orchestrate these distributed services to conduct data analysis and produce useful results. For instance, modern astronomers use computational workflows [14, 15] that construct sky mosaic images to study the structure of distant galaxies. These workflows involve services that may be geographically distributed, and provide functionality for processing, analysing, and producing new imagery, the discussion of which is beyond the scope of this paper. There are many data-centric workflows used in different scientific domains which are characterised in [11]. However, determining the most appropriate location at which to execute the workflow logic becomes difficult as the number of geographically distributed services increases.

Most existing workflow orchestration approaches are based on data placement that employs staging, replication, management and resource allocation techniques which are discussed in [16, 17, 18]. However, the distribution of large portions of data between the services and across long distances through the centralised engine can affect the data transfer rate, increase the execution time, risk overwhelming the storage resources at execution sites, and degrade the overall workflow performance. Recent research efforts show interest in using Infrastructure as a Service (IaaS) clouds that provide on-demand computational services to support cost-efficient deployment and management of scientific workflows [19, 20], but do not examine how the location of distributed services can affect the workflow performance as demonstrated in this paper.

This paper presents a decentralised orchestration system that decomposes a workflow into smaller sub workflows. It determines the most appropriate locations to which these sub workflows are transmitted and subsequently executed. Unlike existing approaches that depend on data placement techniques, our approach migrates the workflow computation towards the services providing the data. Through adopting this approach, autonomous distributed engines can collaborate together to execute the workflow. Each engine is carefully selected to execute a particular sub workflow based on placement analysis that relies on network resource monitoring. Quality of Service (QoS) information such as the network latency and bandwidth are collected and analysed using heuristic algorithms to select the nearest engines to the services. For instance, an engine may be selected to execute a sub workflow composed of services hosted in the same region where the engine is hosted.

1.1. Research Contributions

Decentralised orchestration can bring scalability and performance benefits due to the lack of a single execution engine that may become a performance bottleneck, and the distribution of both the computation and the intermediate data in the workflow. Executing the workflow computation "closer" to the services providing the data improves the overall execution time, whereas distributing the intermediate data across several workflow engines reduces the overall network traffic and improves data transfer time. This is because the data is forwarded directly to where it is required in the workflow without passing through a centralised engine. However, decentralised orchestration is inherently more complicated to implement than centralised orchestration due to the interactions between distributed, asynchronous, and concurrent service processes in the workflow. This paper provides the following research contributions:

- **Decentralised orchestration system**: Previously we created an experimental architecture for executing service workflows [21], which relies on a high-level data coordination language [22] for the specification of workflows. This language is discussed in Section 3. This paper builds on our previous works and presents the design of a novel decentralised orchestration system that supports an autonomous approach to addressing the shortcomings of existing workflow management systems. The principal goal of this system is to support scalability and performance when executing data-centric workflows composed of distributed services across distant network locations. This paper provides an overview of the system's architectural

components and their interactions, and discusses the components' internal modules. It presents a model that explains our decentralised orchestration approach, which provides a mathematical representation of the total cost for executing multiple workflow partitions by distributed engines, and the total cost of transferring data between the engines.

- **Orchestration language**: This paper presents a high-level data coordination language that is called *Orchestra*. This language permits a workflow to be specified without the knowledge of how it is executed, and provides succinct abstractions for defining a set of services, composing them together, and regulating the data movement between them.
- **Novel workflow partitioning and placement analysis approach**: This paper presents a partitioning approach that permits a workflow to be decomposed into smaller sub workflows for parallel execution. This approach relies on network resource monitoring, and placement analysis to determine the most appropriate locations to which these sub workflows are transmitted and subsequently executed. It is hypothesised that this approach improves the workflow performance by reducing the overall data transfer among the services.
- **Experimental evaluation**: In order to investigate the benefits of our approach, we use a set of experimental workflows that are orchestrated over Amazon EC2 and across several geographic network regions. These workflows are based on dataflow patterns that are commonly used to compose large-scale scientific workflows [23].

1.2. Paper Organisation

The rest of this paper is organised as follows: Section 2 presents an overview of our decentralised orchestration system, its components and their interactions, and presents our decentralised orchestration cost model. Section 3 presents the syntax of the *Orchestra* language and discusses its general properties. Section 4 discusses the stages of our decentralised orchestration approach. Section 5 evaluates our approach implementation by orchestrating experimental workflows in both centralised and decentralised settings. Section 6 reviews related literature and highlights the differences between our approach and existing works. Finally, Section 7 summarises our work achievements and states future research directions.

2. DECENTRALISED ORCHESTRATION SYSTEM

This paper proposes a decentralised orchestration system to address the shortcomings of existing workflow management systems. The system's principal goal is to support scalability and performance when executing data-centric workflows composed of services distributed across distant network locations. It is often possible for users to manage a workflow application effectively up to a certain size and complexity, beyond which manual management using software tools becomes unrealistic. Hence, it becomes necessary to employ a system that can make decisions without human guidance to manage workflows. The following sections describe the architectural design of the system, its components and their interactions.

2.1. Architecture Overview

The system's architecture consists of multiple components that represent service-oriented endpoints including the distributed execution engines, and the services participating in the workflow. These engines are lightweight components that can be hosted on physical or virtual servers at different locations, each of which can be exposed to the network by a standard web service interface to enable communication with other components. Unlike existing service orchestration approaches where the locus of control is represented by a centralised engine that holds the decision logic of the workflow, the notion of a single locus of control does not exist in this architecture. During the execution of a workflow, its decision logic can be found at one or more engines as each engine executes part of the overall workflow. This is achieved by initially analysing and partitioning a workflow

specification into smaller partitions, which represent independent sub workflow specifications that can be executed in parallel. Following partitioning, the architecture employs network resource monitoring and placement analysis to determine the most appropriate engines onto which the partitions may be deployed and executed. The system's architecture relies on a high-level data coordination language that we refer to as *Orchestra*, which is used for the specification and execution workflows [22]. Figure 1 provides an architectural overview of the decentralised system that shows the interactions amongst multiple engines and services. The directed edges labeled (E-E) represent engines' interactions, whereas those labeled (E-S) represent engine and service interactions. These interactions are discussed in Section 2.2.

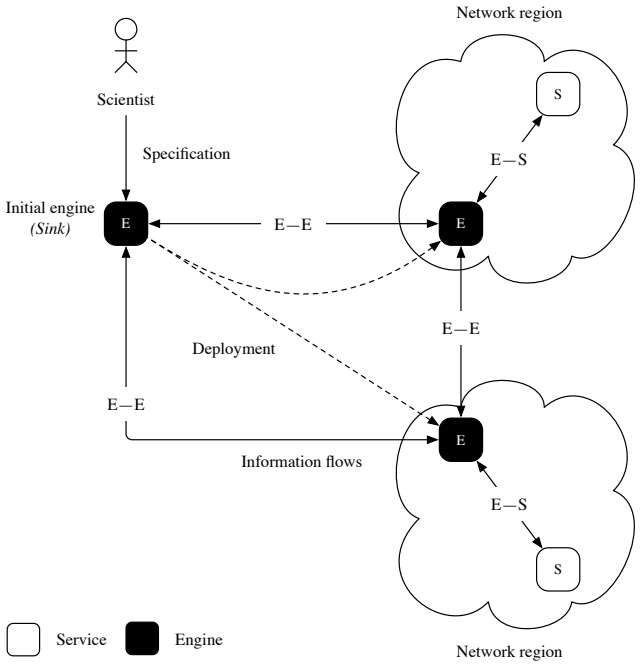

Figure 1. Decentralised orchestration system architecture.

2.2. Components' Interactions and Configurations

Each engine may interact with one or more services hosted in the same network region, and may interact with multiple engines hosted in remote network regions as shown in Figure 1. These interactions are classified as follows:

- **Engine and service interactions**: Centralised orchestration architectures rely on a single execution engine to interact directly with all services. Using our architecture, several engines collaborate together to execute the overall workflow. Each engine may be employed as a *proxy* component to invoke a set of services which are located in the same network region where it is hosted. Service invocations take place as soon as the input data that is required for their invocation becomes available from other sources (e.g. services, or remote engines). The service invocations's results are collected by the engine and stored to be used in the future when needed during the workflow execution.
- **Engines' interactions**: Engines collaborate with each other by automatically transferring data segments relating to the workflow to remote locations where they are required. For instance, the a service invocation result may be forwarded by an engine that collected it to a remote engine that requires it in order to execute a particular sub workflow, which allows the overall execution of the workflow to progress.

Execution engines are preferably installed and configured "closer" to the services in terms of network distance. This minimises the communication overhead between an engine and a particular service. Depending on our workflow partitioning and placement analysis approach, an engine can be responsible for invoking a single service or multiple services. It may not be possible to install an engine as closely as possible to a particular service (e.g. on the same server or within the same network domain) due to restrictions imposed by the service provider or administrator. However, performance benefit can still be gained by harnessing the connectivity of available engines installed across different network regions as demonstrated in the evaluation section of this paper.

2.3. Execution Engine's Internal Modules

Execution engines in this architecture are identical in design, and provide the same functionality. Each engine consists of internal modules that provide self-management capabilities. These modules cooperate together to effectively parse and analyse a workflow, partition it into smaller sub workflows and deploy them for execution in a monitored service-oriented environment. Figure 2 shows the internal modules of the execution engine. The implementation of the engine is based on Java, and it can be deployed as a web service on any physical or virtual server that supports the *Java Runtime Environment* (JRE) and *Apache Tomcat* server technology.

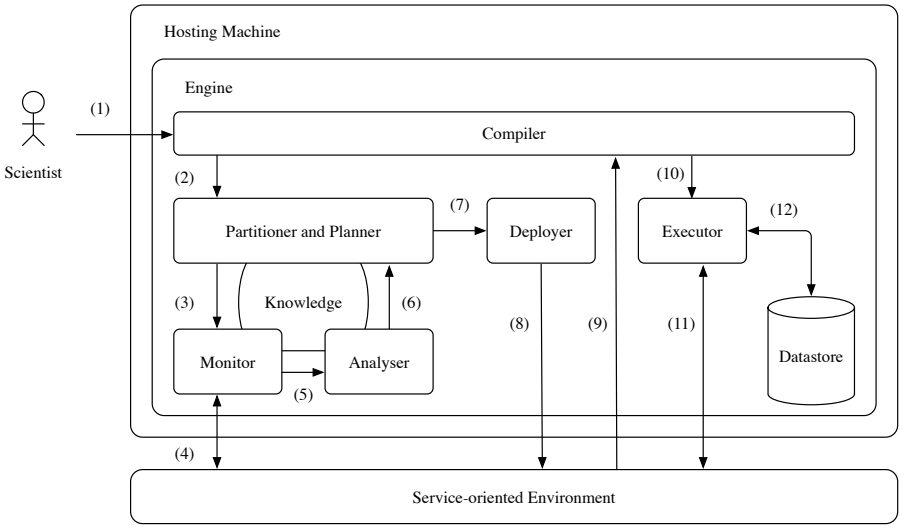

Figure 2. The execution engine's internal modules and their interactions: The compiler accepts a workflow specification from the user (e.g. scientist) (1), and generates a dataflow graph which is passed to the partitioner for decomposition (2). The partitioner cooperates with the monitor (3) to gather QoS information from the environment (4). This information is passed to the analyser (5). The analysis outcome is passed to the planner (6), which generates a deployment plan (7). The deployer transmits the partitions to remote engines (8). The compiler parses a partition's specification (9) and passes its dataflow graph to the executor (10), which invokes the services and communicates with remote engines (11). The executor uses a persistent datastore to store and retrieve data relevant to the workflow execution (12).

- **Compiler**: The compiler module is built from a set of procedures matching the production rules of the *Orchestra* language [22]. It ensures the correctness of the workflow specification, and constructs an executable data structure that represents the workflow.
- **Partitioner and planner**: This module is responsible for partitioning the workflow, and generating a deployment plan for the partitions.
- **Monitor**: This module monitors network resources including remote engines and services, and collects QoS metrics such as the network latency and bandwidth which are stored in

a *knowledge base* module, and can be used in the future to inform placement analysis and deployment planning decisions in the system.

- **Analyser**: This module performs placement analysis using information collected from the environment to determine candidate engines for executing the workflow partitions. It may cooperate with the partitioner module to restructure the workflow partitions as necessary.

- **Deployer**: Based on the placement analysis outcome, this module generates a deployment plan for transmitting the workflow partitions to remote engines and triggering their execution.

- **Executor**: This module is responsible for invoking services, collecting the invocations' results, and forwarding these results to remote engines as necessary.

- **Datastore**: The engine maintains information about the workflows it is executing or have executed, and pertains data relevant to these workflows using a persistent datastore module.

2.4. Orchestration Cost Modelling

The presented decentralised service orchestration approach permits a workflow to be decomposed into sub workflows (e.g. partitions) that can be executed concurrently using distributed engines. This paper explains this approach using a mathematical model that describes the overall cost of transferring data amongst the entities participating in a workflow. Before presenting this model, it is important to understand how centralised service orchestration works. Figure 3 provides a mathematical model that expresses the cost of transferring data in centralised service orchestration using a set of equations. Equation 1 which expresses the cost of executing a service invocation, in which $\vec{I}$ is the cost of transferring the input data from engine E to service S, and $\vec{O}$ is the cost of returning the service invocation output to the same engine. Equation 2 which expresses a *one-way* invocation in which the engine transmits input data to a service without expecting a result from the service. Equation 3 which expresses the execution cost of the entire workflow, where E_σ is the centralised engine that invokes each service S_i and S_k.

$$\overleftrightarrows{C}_{invocation}(E,S) = \vec{I}_{E,S} + \vec{O}_{S,E} \tag{1}$$

$$\vec{C}_{invocation}(E,S) = \vec{I}_{E,S} \tag{2}$$

$$C_{workflow} = \sum_{S_i=1}^{n} \overleftrightarrows{C}_{invocation}(E_\sigma, S_i) + \sum_{S_k=1}^{k} \vec{C}_{invocation}(E_\sigma, S_k) \tag{3}$$

Figure 3. Centralised orchestration cost model.

Decentralised service orchestration is modelled mathematically using a number of equations provided in Figure 4. Equation 4 expresses the cost of executing a single workflow partition P using the engine $E(P)$. Typically, the specification of this partition defines a group of services (e.g. S_i and S_k) to be invoked using the engine $E(P)$. Equation 5 expresses the total cost of transferring intermediate data in the workflow from engine $E(P)$ to each remote engine E_k that requires the data, which is denoted by $\vec{D}$ in addition to transferring the partition's execution outputs from engine $E(P)$ to an engine that acts as an ultimate data sink E_δ. Equation 6 expresses the overall cost of executing a workflow which consists of the total cost of executing each workflow partition P_i, and the total cost of routing data between the distributed engines.

$$C_{partition}(P) = \sum_{S_i=1}^{n} \overleftrightarrows{C}_{invocation}(E(P), S_i) + \sum_{S_k=1}^{k} \overrightarrow{C}_{invocation}(E(P), S_k) \quad (4)$$

$$C_{routing}(P) = \sum_{E_k=1}^{m} (\overrightarrow{D}_{E(P),E_k}) + \overrightarrow{O}_{E(P),E_\delta} \quad (5)$$

$$C_{workflow} = \sum_{P_i=1}^{n} (C_{partition}(P_i) + C_{routing}(P_i)) \quad (6)$$

Figure 4. Decentralised orchestration cost model.

3. THE WEB SERVICE ORCHESTRA LANGUAGE

Orchestra is a high-level functional data coordination language for the specification and execution of web service workflows. It permits a workflow to be expressed in a simple and intuitive manner such that the compiler can effortlessly produce highly scalable, distributed form of the workflow. *Orchestra*'s logic can be partitioned into smaller computational units that can roam the network and may be executed at different sites. It is uniquely characterised by its deterministic nature, data-driven execution support, and strong type system [22].

3.1. General Syntax

Orchestra consists of several abstractions that define the workflow namespace, service endpoints, I/O interface, computation components and data coordination as shown in listing 1.

```
01 workflow example
02 description d1 is http://ec2-54-80-6-125.compute-1.amazonaws.com/
   services/Service1?wsdl
03 service s1 is d1.Service1
04 port p1 is s1.Port1
05 input:
06    int a
07 output:
08    int b
09 a -> p1.Op1
10 p1.Op1 -> b
```

Listing 1. Specification of a simple service invocation using *Orchestra*.

The workflow namespace ⌜example⌝ is defined in Line 1 using the ⌜**workflow**⌝ keyword. Service description identifiers such as ⌜d1⌝ can be declared using the ⌜**description**⌝ keyword in Line 2. This permits *Orchestra's* compiler to locate a service description document based on WSDL by URL, and analyse it to obtain information about the service such as its port, operations and associated data types. The service identifier ⌜s1⌝ is declared in Line 3 using the ⌜**service**⌝ keyword. It represents a service that is specified as ⌜Service1⌝ in the service description document which is identified by ⌜d1⌝. Similarly, the port identifier ⌜p1⌝ is declared in Line 4 using the ⌜**port**⌝ keyword. It represents a port of service ⌜Service1⌝ that is specified as ⌜Port1⌝ in the service description document. The ⌜**input**⌝ and ⌜**output**⌝ keywords define the workflow's input and output interfaces through Lines 5-8, which provide an input ⌜a⌝ and an output ⌜b⌝. The input ⌜a⌝ of integer type is used to invoke operation ⌜Op1⌝ that is provided by port ⌜p1⌝ , which produces the output ⌜b⌝ of integer type through Lines 9-10. The arrow symbol ⌜->⌝ indicates the dataflow in and out of the service and acts as a data type constructor when assigning values of service invocation outputs to variables.

3.2. Service Composition

Services can be composed together using *Orchestra* by simply directing the output of a service operation to another. Service composition abstractions can be used in *Orchestra* to express simple dataflow patterns including the pipeline, data distribution and aggregation patterns. The process pattern represents a simple service invocation whose specification has been discussed in Section 3.1. The pipeline pattern is used for chaining several services, where the output of particular service is passed as an input to another service in a sequential manner. The data distribution pattern represents the passing of a service output as an input to multiple services. The data aggregation pattern represents the collection of multiple outputs from different services which are all used as input parameters to a single service. These patterns are commonly used to construct large-scale workflows [11], and their complete specification using *Orchestra* can be found in [22].

3.3. Distributed Computation Support

Orchestra's computation logic can flow freely from site to site along communication channels between multiple compute servers called *execution engines*. Figure 5 shows an arbitrary workflow specification that composes services together, and illustrates its orchestration. It shows the partitioning of the overall specification into smaller fragments of code, which are transported to remote engines for execution. Execution engines are web services themselves and hence are treated

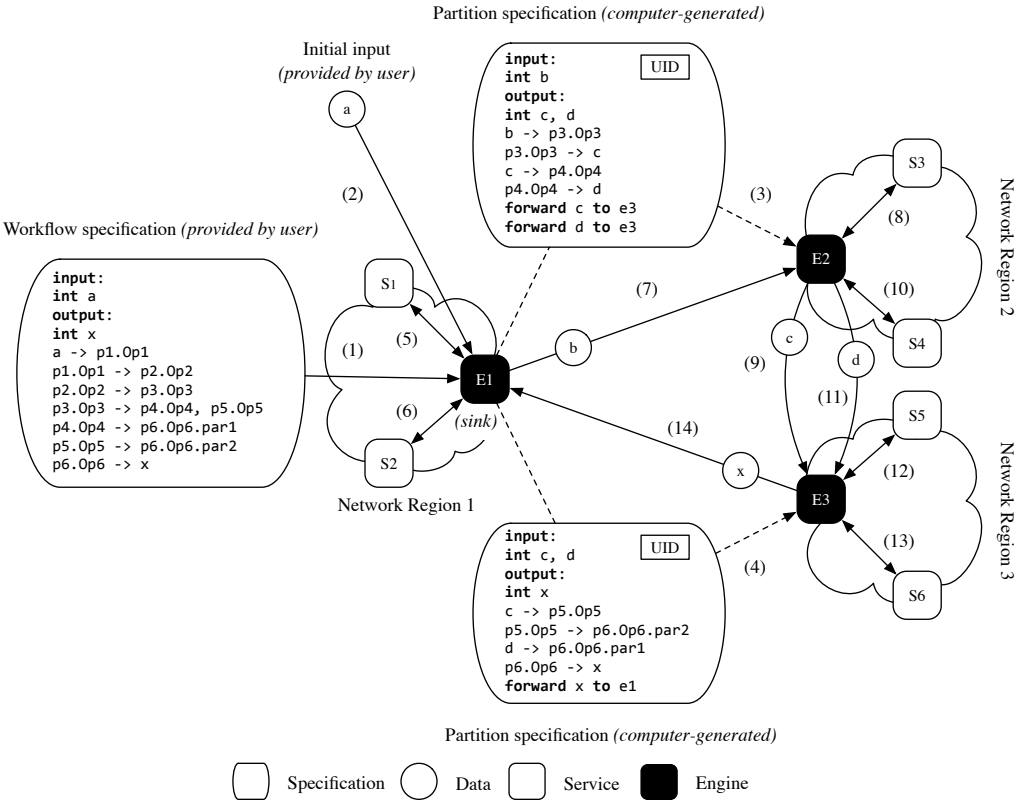

Figure 5. Orchestration of an arbitrary workflow where an initial engine (E_1) accepts a workflow specification and input from the user in (1, 2). Engine (E_1) decomposes the workflow and deploys one or more workflow partitions onto remote engines (E_2 and $E3$) in (3, 4). Each engine invokes a group of services in the same network region where it is hosted in (5, 6, 8, 10, 12, and 13). The engines communicate with each other by forwarding data entities to locations where they are required in (7, 9, 11, and 14). In this example, engine (E_1) executes a partition that generates the output (b) which is transmitted to engine (E_2) to perform further computation. Similarly, engine (E_2) produces and forwards (c and d) to engine (E_3).

as such by *Orchestra*, and these engines can be uniquely identified by their web address location. Each engine maintains a table that consists of information about the workflows it is executing or has executed, and pertains data relevant to these workflows using a persistent datastore. *Orchestra* provides abstractions to define references to execution engines which are discussed in [24], but these engines are used exclusively by the workflow execution architecture. For instance, information about the execution engines may be obtained by the compiler and used during workflow partitioning and placement to determine the appropriate engines to execute the workflow partitions. Hence, the user does not need to indicate which parts of the workflow that need to be partitioned and on which engines should they be executed, or how the engines communicate with each other. In order to support distributed computation, *Orchestra* must provide abstractions to support collaboration between execution engines. Such abstractions can be used to form statements in the specifications of workflow partitions, which may indicate the following:

- **Execution engines**: the execution engines may be defined in the original workflow specification, and in the specification of workflow partitions.
- **Data forwarding**: the workflow partitions may provide statements that indicate the dataflow between the execution engines.
- **Unique identification information**: *Orchestra*'s compiler generates a *Unique Identifier* (UID) for each workflow partition that enables an engine to distinguish it from other partitions being executed on the same machine.

4. STAGES OF DECENTRALISED ORCHESTRATION

This section explains the stages of orchestration in our system which relate to the compilation, partitioning, placement, deployment and execution of workflows, and network resource monitoring.

4.1. Compilation

The workflow engine uses a recursive descent compiler that is built from a set of procedures matching the production rules of the *Orchestra* language grammar [22]. This compiler parses a given workflow specification and analyses it to ensure its correctness. It performs semantic matches against distributed system components including services and execution engines specified in the workflow. It does not generate machine code representation from the workflow specification, instead it constructs a data structure that represents a *dataflow graph* in which the vertices are service operations to be invoked with edges between them as data dependencies. Typically, dataflow graphs are used to guide the execution process where the dataflow tokens represent inputs for service operations, which may be activated when all the required input parameters become available. This data-driven workflow representation permits its data structure to be decomposed into independent parts that can be distributed across multiple machines and executed in parallel. Furthermore, it allows the workflow to be maintained upon its distribution such that it can be restructured for re-deployment to deal with emergent run-time issues.

4.2. Partitioning

Following compilation, the workflow is analysed to gain insight about its complexity and to detect its intricate parallel parts. Hence, a traverser is used to explore the data structure of the dataflow graph generated by the compiler. This traverser obtains useful information relating to the workflow inputs, outputs, services, service operations and the type representations associated with their input parameters and outputs. This information may be used to introduce modifications to the dataflow graph structure as necessary during partitioning. Partitioning relies on network resource monitoring and placement analysis, and consists of the following steps:

1. Firstly, the dataflow graph is decomposed into the maximum number of smallest sub workflows that can be isolated for parallel execution. The result of this decomposition is a set of sub workflows each of which represents a single service invocation. These sub workflows may require new inputs and provide outputs that may be required by other sub workflows.

2. Secondly, network resource monitoring is used to determine available engines that can be used to execute the sub workflows, and to detect the network condition between the engines and the services involved in the workflow. Section 4.4 discusses network resource monitoring.

3. Thirdly, placement analysis is used to determine the most appropriate execution engine for each sub workflow. It relies on the knowledge about the network condition and uses a heuristic algorithm to select a candidate engine. Section 4.3 discusses placement analysis in detail.

4. Finally, based on placement analysis the sub workflows may be combined together if the same engine is selected to execute them. This involves introducing directed edges between the sub workflows wherever a data dependency exists. Consequently, the composite workflows are encoded using the same language as used to specify the whole workflow. During the recoding, relevant information such as the workflow inputs, outputs, services, service operations, data dependencies and type representations are all captured, and associated with the composite workflows to make each a self-contained stand-alone workflow specification called a partition.

4.3. Placement Analysis

Placement analysis uses the knowledge about the network with a combination of heuristics for selecting the most appropriate engines onto which the workflow partitions are deployed. Figure 6 illustrates the placement analysis steps which include the following:

1. Firstly, all the available execution engines are organised into groups using a clustering algorithm such as *k-means* that identifies similar engines based on their QoS metrics.

2. Secondly, the groups containing inappropriate engines are eliminated from further analysis by recursively comparing samples obtained from different groups. For instance, the QoS metrics of an engine sample that is selected randomly from one group is compared with the metrics of a sample that is selected from another group. This helps in identifying the engines with high latency and low bandwidth metrics, which are worse than those of engines in other groups. Consequently, a single group remains which represents a collective of candidate engines to execute the workflow partition.

3. Finally, each candidate engine is ranked by predicting the transmission time between the engine and the service to be invoked in the sub workflow using the following simple equation:

$$T = L_{e-s} + S_{input}/B_{e-s} \tag{7}$$

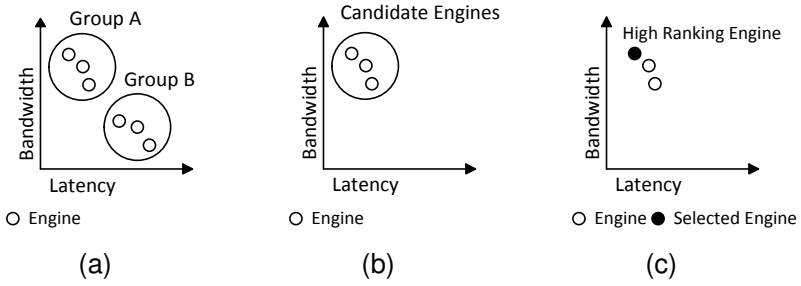

Figure 6. Placement analysis: (a) clustering engines into groups based on their QoS metrics, (b) eliminating inappropriate engines, and (c) selecting a candidate execution engine.

where T is the transmission time, L_{e-s} and B_{e-s} are the latency and bandwidth between the engine and the service respectively, and S_{input} is the size of the input that is used to invoke the service. Consequently, the engine with the shortest transmission time is selected.

4.4. Network Resource Monitoring

Network resource monitoring constructs a logical network topology that represents an indirect graph in which the vertices are network resources (e.g. services, and available engines), where the edge between any pair of vertices represents the probable network distance between them. Constructing this topological view involves reachability analysis which determines if the engines can communicate with the services and with themselves, after which Quality of Service (QoS) metrics are collected from probing the network resources to calculate the probable network distance between them. The engines can probe other network resources by emulating the *ping* protocol from the application layer using HTTP HEAD requests. QoS metrics include the network latency and bandwidth, which are described as follows:

- **Network latency:** This metric represents the length of time that takes a request message sent by an engine to reach a service, and the time required to acknowledge the engine's request by transmitting a response message.
- **Network bandwidth:** This metric represents the total amount of information that can be transmitted between a particular engine and a service in a given time.

QoS metrics can be useful in placement analysis for selecting the nearest engine to a service to execute a particular sub workflow, where the engine has short latency and high bandwidth with the service. Such metrics indicate faster time for transferring data between the engine and the service.

4.5. Deployment and Execution

Following workflow partitioning, the specification of each workflow partition is dispatched to a designated remote engine that may be located in a different network region. This specification is compiled and analysed accordingly by the receiving engine, and may be executed immediately after generating a corresponding dataflow graph of the workflow partition. Executing this dataflow graph involves invoking a set of services as specified in the workflow partition, collecting the invocation results, and forwarding data segments relating to these results to remote engines as necessary. Since this execution process is data-driven, service operations are only invoked as soon as the input parameters that are required for their execution become available. Data-driven execution is useful because it permits the inputs to be obtained from different sources including remote engines, which automatically forward the values of these inputs to locations at which they are required. It incurs no communication overhead to request inputs from remote sources. Furthermore, there is no need for a centralised scheduling mechanism to manage the order in which the partitions are executed.

5. EVALUATION

This section provides an evaluation of the presented orchestration system. It presents a set of experiments that aim to compare between the performance of centralised and decentralised service orchestration. These experiments involve the orchestration of a workflow based on common dataflow patterns discussed in Section 3.2. Each workflow consists of a number of services that are hosted in different network regions across *Amazon* EC2. The configuration of the services used in the experimental workflows consists of regional and inter-regional configurations. This paper defines a regional configuration as the organisation and placement of services in a workflow, where all the services are hosted in the same network region. Services configured in the same region are orchestrated in both centralised and decentralised manner. For example, centralised orchestration is carried out using a single engine that is hosted in the same network region (e.g. locally) where the services are hosted, whereas decentralised orchestration is carried out using multiple engines

that are hosted locally in the same region. This paper defines an inter-regional configuration as the organisation and placement of services in a workflow, where the services are hosted in different network regions. Services configured in multiple regions are orchestrated in both centralised and decentralised manner. For example, centralised orchestration is carried out using a single engine that is hosted in an arbitrary network region, whereas decentralised orchestration is carried out using multiple engines that are hosted in different network regions where the services are hosted. The completion time for each workflow is recorded in seconds, and the size of total communicated data in MB. The mean speedup is computed using: $S = T_c/T_d$ where T_c and T_d are the average workflow completion times using centralised and decentralised orchestration.

5.1. Pipeline Pattern Workflows

These experiments aim to evaluate the execution of workflows based on the pipeline pattern using centralised and decentralised orchestration approaches. They are used to compare the performance between centralised and decentralised orchestration approaches as both the number of services and the total size communicated data in the workflow increase. In this pattern, a particular engine would invoke a service operation and obtain its result which may then be forwarded to another engine for a subsequent service invocation. Since the output of one operation is to be used as an input for another, then decentralised orchestration may not provide significant improvement over centralised orchestration. Figure 7 shows the experimental results of these workflows, where workflows (1, 2, 3, and 4) consist of 10, 20, 30, and 20 services respectively. The services of workflow (1) are all hosted in Oregon, whereas the services of workflow (2) are all hosted in N. California. The services of workflow (3) are dispersed evenly across Oregon, N. California, and Ireland, whereas the services of workflow (4) are dispersed randomly across N. Virginia, Ireland, and Oregon. The average speedup for workflows (1, 2, 3, and 4) are 1.31, 1.23, 1.74, and 2.07 respectively.

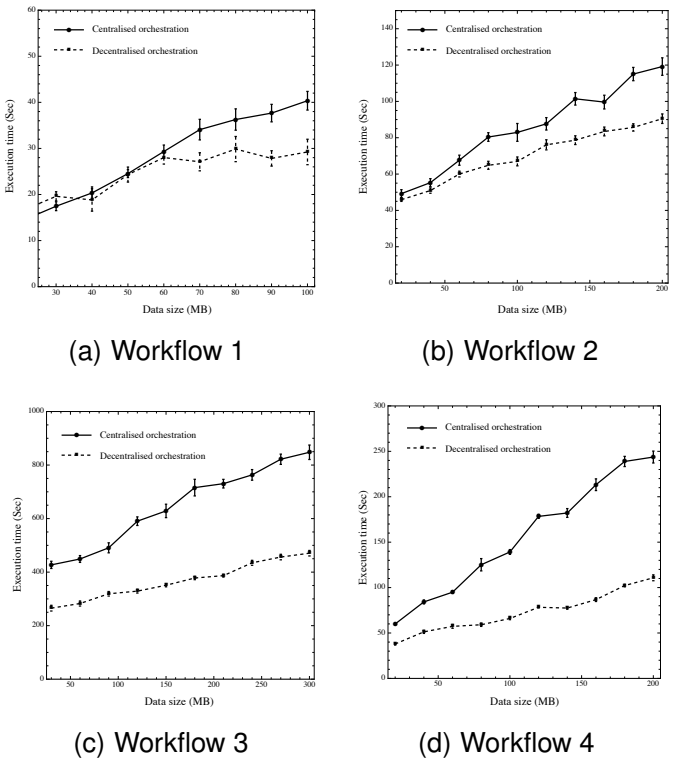

Figure 7. Experimental results of orchestrating workflows based on the pipeline dataflow pattern.

5.2. Data Aggregation Pattern Workflows

These experiments aim to evaluate the orchestration of workflows structured as reductive dataflow graphs, in which the data is consumed gradually as the graph grows inward towards a single sink. Similar to the experimental workflows based on the pipeline pattern, these workflows are used to compare the performance of orchestration using centralised and decentralised approaches as the number of services and the total size of communicated data in the workflow increase. In a centralised orchestration model, the data aggregation pattern would be treated similar to the pipeline dataflow pattern. This is because although multiple data providing services could send the data directly to a single service that acts as a data sink, they will instead send the data to a centralised engine. The centralised engine will wait for the data from all the services to be received before forwarding it to the final service, and therefore becomes a synchronisation point. In this experiment, as the data becomes available from other services, distributed engines acting as proxies to these services may forward such data to the engine that is responsible for invoking the final service concurrently. This helps in reducing the overall time of executing the workflow. Figure 8 shows the experimental results of these workflows, where workflows (5, 6, 7, and 8) consist of 10, 20, 30, and 20 services respectively. The services of workflow (5) are all hosted in Oregon, whereas the services of workflow (6) are all hosted in N. California. The services of workflow (7) are dispersed evenly across N. Virginia, N. California, and Frankfurt, whereas the services of workflow (8) are dispersed randomly across N. Virginia, Ireland, and Oregon. The average speedups for workflows (5, 6, 7, and 8) are 1.09, 1.20, 1.94, and 2.00 respectively.

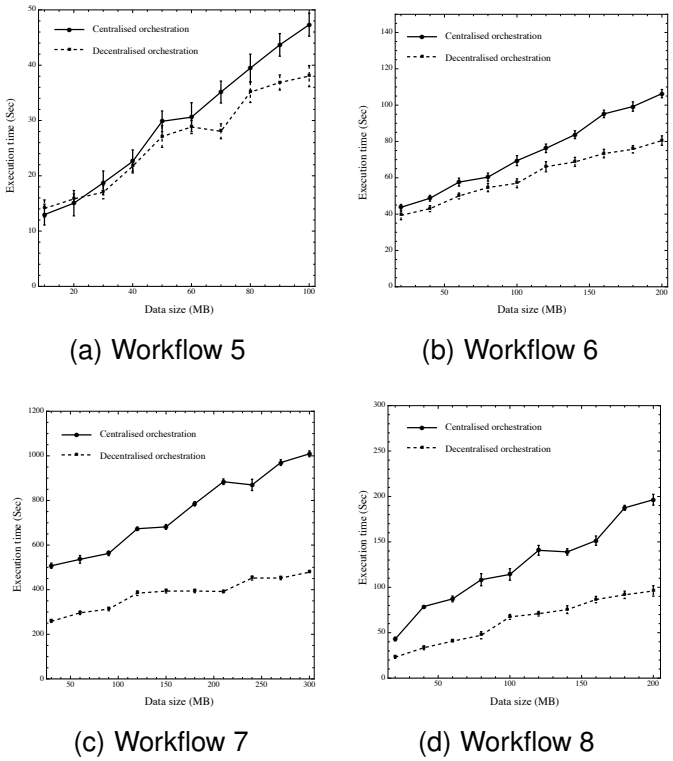

Figure 8. Experimental results of orchestrating workflows based on the data aggregation pattern.

5.3. Data Distribution Pattern Workflows

These experiments aim to evaluate the orchestration of workflows structures as expansive dataflow graphs, in which more data is produced gradually as the graph grows outward towards multiple sinks. Similar to the experimental workflows based on both the pipeline and data aggregation

patterns, these workflows are used to compare the performance of orchestration using centralised and decentralised approaches as the number of services and the total size of communicated data in the workflow increase. Each service provides a set of data to be forwarded to one or more services, but unlike the data aggregation pattern there is no single service that is acts as a data sink for multiple outputs produced by other services. Hence workflows of this kind can be parallelised by using distributed engines, where each executes part of the workflow concurrently. Figure 9 shows the experimental results of these workflows, where workflows (9, 10, 11, and 12) consist of 10, 20, 30, and 20 services respectively which are configured similarly to the experimental workflows based on the data aggregation pattern. The average speedups for workflows (9, 10, 11, and 12) are 1.12, 1.26, 1.90, and 1.95 respectively.

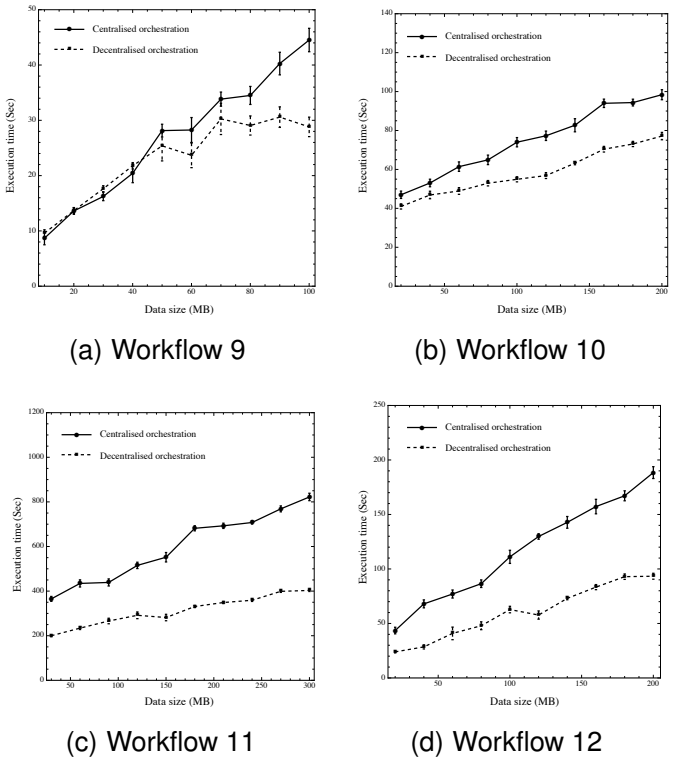

Figure 9. Experimental results of orchestrating workflows based on the data distribution pattern.

5.4. Discussion

Decentralised orchestration provides performance improvement for both regional and inter-regional workflows, and based on the presented results we make the following general observations:

1. Centralised orchestration may take considerable amount of time to execute a workflow often due to passing multiple copies of intermediate data between the engine and the services. Furthermore, the high latency and low bandwidth between the centralised engine and a group of services may affect the overall workflow performance negatively.

2. Executing a regional workflow that consists of a small number of services using decentralised orchestration provides some performance improvement over centralised orchestration but may not be useful at all times. This is because introducing more engines involves the communication of additional intermediate copies of data between them, which may increase the workflow execution time.

3. Decentralised orchestration provides some performance improvement over centralised orchestration when executing workflows. It provides significant performance improvement over centralised orchestration as both the number of services and the data communicated in the workflow increase, which can be attributed to:

   (a) The distribution of the workflow logic. Executing parts of the workflow "closer" to the services providing the data reduces the round-trip times for transferring the data between the engines and the services. Consequently, this improves the overall execution time.

   (b) The distribution of the intermediate data. The execution engines permit the intermediate data produced during the workflow execution to be transferred directly to network locations where the data is required without a centralised point of coordination.

6. RELATED WORKS

There are a small number of research articles that have investigated the scalability of centralised orchestration. Chafle et al. [25] proposed a decentralised orchestration approach to support the execution business-oriented workflows, which decomposes a workflow into smaller parts that can be executed by multiple engines, but the authors do not explain how the decomposition process is performed. Binder et al. [26] proposed an approach that relies on proxies to trigger the invocation of services, collect their outputs and forward them directly to locations where they are required. Hence, each proxy executes a service invocation upon the availability of the input data that is required for its execution. Similarly, Liu et al. [27] and Balasooriya et al. [28] provide decentralised approaches in which self-managing proxies embed the coordination logic of a workflow. The main drawback of these works is that their implementations are platform dependent and require the modification of the web services being coordinated. Barker et al. [29] proposed a dataflow optimisation architecture that supports the distribution of data in a workflow using multiple proxies that can be deployed "closer" to the services. However, this architecture relies on a centralised mechanism to coordinate the proxies, and there seems to be no automated workflow partitioning mechanism.

7. CONCLUSION AND FUTURE RESEARCH DIRECTIONS

This paper has focused particularly on centralised service orchestration scalability challenges including the unnecessary consumption of the network bandwidth, high latency in transmitting data between the services, and performance bottlenecks. It has presented and evaluated a novel decentralised orchestration system that permits a workflow to be partitioned into smaller sub workflows, which may then be transmitted to appropriate locations at which their execution takes place. These locations are carefully determined using a heuristic technique that relies on the knowledge of the network condition. This allows the workflow logic to be executed within short distance to the services, which improves the overall performance.

Future research work will focus on dynamic optimisation of workflows. Dynamic optimisation is required to re-configure the deployment of executing workflow partitions. From the deployment process onwards in the system implemented, the analyser does not interfere with the execution by producing a new deployment plan in response to dynamic changes in the network environment. Such changes are often unpredictable and may include unexpected service response delays, external load on the engine as it may be responsible for executing multiple workflows, changes in the network latency or bandwidth. Dynamic optimisation steers the execution process and aims to minimise performance losses due to unpredictable changes that disrupt the static deployment plan computed by the analyser. Given the ability to control the state of the executing workflow partitions, a dynamic optimisation mechanism can be used to pause the overall execution, after which the workflow partitions are refactored to be redeployed in new network locations at which their execution can resume and progress efficiently. Dynamic optimisation requires a real-time distributed monitoring approach to track the execution progress of the workflow partitions, and to collect information about

the network condition that can be used to devise new deployment plans. Furthermore, dynamic optimisation may change the overall workflow structure by combining workflow partitions or decomposing them further into smaller partitions, and modifying information relating to the engines and instructions to route the data between the engines in the workflow partitions.

REFERENCES

1. Bray T, Paoli J, Sperberg-McQueen CM, Maler E, Yergeau F. eXtensible Markup Language (XML) 1998.
2. Box D, Ehnebuske D, Kakivaya G, Layman A, Mendelsohn N, Nielsen HF, Thatte S, Winer D. Simple Object Access Protocol (SOAP) 1.1 2000.
3. Christensen E, Curbera F, Meredith G, Weerawarana S, *et al.*. Web Services Description Language (WSDL) 1.1 2001.
4. Fielding RT. Architectural styles and the design of network-based software architectures. PhD Thesis, University of California, Irvine 2000.
5. Hollingsworth D, Hampshire U. Workflow management coalition the workflow reference model. *Workflow Management Coalition* 1993; **68**.
6. Deelman E, Gannon D, Shields M, Taylor I. Workflows and e-science: An overview of workflow system features and capabilities. *Future Generation Computer Systems* 2009; **25**(5):528–540.
7. Görlach K, Sonntag M, Karastoyanova D, Leymann F, Reiter M. Conventional workflow technology for scientific simulation. *Guide to e-Science*. Springer, 2011; 323–352.
8. Ludäscher B, Weske M, McPhillips T, Bowers S. Scientific workflows: Business as usual? *Business Process Management*. Springer, 2009; 31–47.
9. Andrews T, Curbera F, Dholakia H, Goland Y, Klein J, Leymann F, Liu K, Roller D, Smith D, Thatte S, *et al.*. Business process execution language for web services 2003.
10. Erl T. *Service-oriented architecture (SOA): concepts, technology, and design*. Prentice Hall Englewood Cliffs, 2005.
11. Juve G, Chervenak AL, Deelman E, Bharathi S, Mehta G, Vahi K. Characterizing and profiling scientific workflows. *Future Generation Computer Systems* 2013; **29**(3):682–692.
12. Oinn TM, Addis M, Ferris J, Marvin D, Senger M, Greenwood RM, Carver T, Glover K, Pocock MR, Wipat A, *et al.*. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics* 2004; **20**(17):3045–3054, doi:10.1093/bioinformatics/bth361.
13. Hey AJ, Trefethen AE. The data deluge: An e-science perspective 2003; .
14. Jacob JC, Katz DS, Prince T, Berriman BG, Good JC, Laity AC, Deelman E, Singh, *et al.*. The montage architecture for grid-enabled science processing of large, distributed datasets 2004; .
15. Rynge M, Juve G, Kinney J, Good J, Berriman GB, Merrihew A, Deelman E. Producing an infrared multiwavelength galactic plane atlas using montage, pegasus and amazon web services. *23rd Annual Astronomical Data Analysis Software and Systems (ADASS) Conference*, 2013.
16. Bharathi S, Chervenak A. Data staging strategies and their impact on the execution of scientific workflows. *Proceedings of the second international workshop on Data-aware distributed computing*, ACM, 2009.
17. Ranganathan K, Foster I. Simulation studies of computation and data scheduling algorithms for data grids. *Journal of Grid Computing* 2003; .
18. Kosar T, Livny M. A framework for reliable and efficient data placement in distributed computing systems. *Journal of Parallel and Distributed Computing* 2005; **65**(10):1146–1157.
19. Deelman E, Singh G, Livny M, Berriman GB, Good J. The cost of doing science on the cloud: the montage example. *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2008, November 15-21, 2008, Austin, Texas, USA*, 2008; 50, doi:10.1145/1413370.1413421.
20. Hoffa C, Mehta G, Freeman T, Deelman E, Keahey K, Berriman GB, Good J. On the use of cloud computing for scientific workflows. *Fourth International Conference on e-Science, e-Science 2008, 7-12 December 2008, Indianapolis, IN, USA*, 2008; 640–645, doi:10.1109/eScience.2008.167.
21. Jaradat W, Dearle A, Barker A. An architecture for decentralised orchestration of web service workflows. *Proceedings of the 20th IEEE International Conference on Web Services*, 2013; 603–604.
22. Jaradat W, Dearle A, Barker A. A dataflow language for decentralised orchestration of web service workflows. *Proceedings of the Ninth IEEE World Congress on Services*, 2013; 13–20.
23. Barker A, Van Hemert J. Scientific workflow: a survey and research directions. *Proceedings of the 7th international conference on Parallel processing and applied mathematics*, Springer-Verlag, 2007; 746–753.
24. Jaradat W, Dearle A, Barker A. Workflow partitioning and deployment on the cloud using orchestra. *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, IEEE Computer Society, 2014; 251–260.
25. Chafle GB, Chandra S, Mann V, Nanda MG. Decentralized orchestration of composite web services. *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, ACM, 2004; 134–143.
26. Binder W, Constantinescu I, Faltings B. Service invocation triggers: a lightweight routing infrastructure for decentralised workflow orchestration. *International journal of high performance computing and networking* 2009; **6**(1):81–90.
27. Liu D, Law KH, Wiederhold G. Data-flow distribution in ficas service composition infrastructure. *Proceedings of the 15th International Conference on Parallel and Distributed Computing Systems*, Citeseer, 2002.
28. Balasooriya J, Padhye M, Prasad SK, Navathe SB. Bondflow: A system for distributed coordination of workflows over web services. *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, IEEE, 2005.
29. Barker A, Weissman JB, van Hemert JI. Reducing data transfer in service-oriented architectures: The circulate approach. *Services Computing, IEEE Transactions on* ; **5**(3):437–449.

