

The Flowbster Cloud-Oriented Workflow System to Process Large Scientific Data Sets

Peter Kacsuk  · József Kovács · Zoltán Farkas

Received: 13 April 2017 / Accepted: 31 October 2017
© Springer Science+Business Media B.V., part of Springer Nature 2018

Abstract The paper describes a new cloud-oriented workflow system called Flowbster. It was designed to create efficient data pipelines in clouds by which large compute-intensive data sets can efficiently be processed. The Flowbster workflow can be deployed in the target cloud as a virtual infrastructure through which the data to be processed can flow and meanwhile it flows through the workflow it is transformed as the business logic of the workflow defines it. Instead of using the enactor based workflow concept Flowbster applies the service choreography concept where the workflow nodes directly communicate with each other. Workflow nodes are able to recognize if they can be activated with a certain data set without the interaction of central control service like the enactor in service orchestration workflows. As a result Flowbster workflows implement a much more efficient data

path through the workflow than service orchestration workflows. A Flowbster workflow works as a data pipeline enabling the exploitation of pipeline parallelism, workflow parallel branch parallelism and node scalability parallelism. The Flowbster workflow can be deployed in the target cloud on-demand based on the underlying Occopus cloud deployment and orchestrator tool. Occopus guarantees that the workflow can be deployed in several major types of IaaS clouds (OpenStack, OpenNebula, Amazon, CloudSigma). It takes care of not only deploying the nodes of the workflow but also to maintain their health by using various health-checking options. Flowbster also provides an intuitive graphical user interface for end-user scientists. This interface hides the low level cloud-oriented layers and hence users can concentrate on the business logic of their data processing applications without having detailed knowledge on the underlying cloud infrastructure.

P. Kacsuk (✉) · J. Kovács · Z. Farkas
Institute for Computer Science and Control,
Hungarian Academy of Sciences, 1111 Budapest,
Kende u. 13-17, Hungary
e-mail: peter.kacsuk@sztaki.mta.hu

J. Kovács
e-mail: jozsef.kovacs@sztaki.mta.hu

Z. Farkas
e-mail: zoltan.farkas@sztaki.mta.hu

P. Kacsuk
Centre for Parallel Computing, University of Westminster,
115 New Cavendish Street, London W1W 6UW, UK

Keywords Workflow · Data stream · Scientific data · Cloud orchestration · Multi-cloud

1 Introduction

Workflow systems are popular when complex processing on large scientific data set is required. This typically means that the processing of data consists of several steps and in each step different kind of processing activities are executed. These steps are formulated

as the nodes (tasks) of the workflow and the transfer of data between these tasks are represented by the arcs of the workflow. Many scientific processing of data can be represented by this kind of workflows even if the scientists are not aware that the scientific process they are actually doing is in fact a workflow. As long as they do not formalize these activities as scientific workflows they have to do manually all the required data preparation and also they have to manually initiate the next required processing step. (Of course, it is possible to write scripts that execute the required steps in an automatic way but in fact this is also a workflow but not generic. It specifically executes only the given tasks.)

The scientific workflows take over the responsibility of preparing the required data and launching the next steps of activities with the prepared data in an automatic way based on the formal definition of the workflows. As a result, workflows can significantly accelerate the work of scientists. Already many scientific communities recognized these advantages of using workflows and hence the use of workflows are getting more and more popular among scientists. The increasing popularity of workflows has led to the creation of many different workflow systems [1–3]. These workflow systems can be divided into two major categories:

1. Service orchestration (enactor) based workflows [4–8]
2. Service choreography based workflows [9, 10]

The service orchestration based workflow systems work with a centralized control mechanism called the enactor. The enactor recognizes that certain tasks (nodes) of the workflow are completed and hence new set of tasks can be started. Then the enactor typically submits the executable tasks as jobs to the underlying computing infrastructure. If the data to be processed by a new task is small, then it can be transferred from the workflow system data storage. If the data is large (e.g. a large file) the submitted task contains the address of the data storage where this large file is stored and the first action of the submitted job is to download the data from the storage. Similarly, after completing the submitted workflow task if the result data is small it is transferred back to the workflow system data storage otherwise it is placed to a target data storage whose address is given as part of the submitted job. In fact, the enactor simulates the scientist and does

exactly the same tasks the scientist would manually have to do. The advantages of the service orchestration concept are:

- Complex control structures like loops can be applied in the workflow.
- Dynamic launching of the workflow tasks can easily be organized and implemented.

The drawbacks of this concept are:

- The management of the workflow execution is a complex process.
- The control and data planes are separate. The data path among the tasks is usually complex and not efficient. Data do not move directly between workflow tasks rather they are moved back and forth between the submitted tasks and the workflow system data storage (or a remote storage).

When web services and generally service technology became popular it was a natural idea that the nodes of the workflow could be realized as communicating services and the data to be processed just could flow through the set of services and transformed as defined by the services (tasks) of the workflow. The workflow works as a virtually hard-wired set of services through which the data should flow and meanwhile it flows it is transformed. The advantages of this service choreography concept are:

- No enactor is needed and hence workflow management is very simple.
- The control and data planes are the same. The data is directly passed among the tasks of the workflow and hence the separation of the data and control paths is not needed. Exactly the availability of the required data enables the execution of workflow tasks without any special control path activity.
- Data is typically passed through the workflow in the form of a data stream. This enables that the workflow can be used as a pipeline: as the first data element of the input data stream is processed by the first task of the workflow the second data element of the input data stream can enter to the workflow pipeline. At this time the workflow processes in parallel the first and second data elements. This type of parallelism is often called as pipeline or vertical parallelism. Workflows based on service choreography can realize vertical parallelism in a very natural way.

The drawback of this concept is:

- Complex control structures like loops cannot easily be applied in the workflow.

Having advantages and disadvantages both types of workflow systems are used by many projects. However, recently processing very large scientific data sets becomes more and more important and hence the service choreography workflow concept might get momentum since it provides more efficient data paths than the enactor based concept.

The research described in this paper aims at providing a convenient and easy to use workflow system based on the service choreography concept in order to support the processing of large scientific data sets. This concept also fits to the IoT data processing where large set of sensors collects data that should be processed in real-time. The new workflow system is called Flowbster and will be described in detail in Section 2. Flowbster is based on the idea of deploying the required data pipeline in a cloud and then use it as a pipeline infrastructure until the whole data set is processed. Then the data pipeline can be removed. Such a deployment of the data pipeline requires the usage of a cloud deployment tool which is in our case Occopus. Flowbster and Occopus together represent a cloud tool family by which complex data processing applications can be built on demand in various cloud systems. They represent different layers of the required software stack as it will be explained in Section 2. The main features of Occopus will be summarized in Section 3 while Section 4 describes the main features and usage scenarios of Flowbster. The current version of Flowbster supports static scalability as explained in Section 4. In order to support dynamic scalability for Flowbster workflows several possible options are described in Section 5. In Section 6 we will show how the Flowbster workflow system can be applied for defining data processing applications on top of Flowbster. Section 7 shows performance measurements on Flowbster workflows. Finally, Section 8 compares Flowbster and Occopus with related work.

2 Concept and Overview

The initial assumption for the use of Flowbster is that a scientist would like to process a large set of scientific data (for example, images of the Via Lactea

star system in order to find star formulation patterns) stored in a large data center. The processing pipeline (workflow) is either stored in a workflow repository like the SHIWA Workflow Repository [11] from where the scientist can initiate the deployment and usage of the workflow or should be developed as a new workflow.

The toolset we offer should support the easy development of the workflow as well as its easy and efficient on-demand usage. A previous heavy weight solution was the WS-PGRADE/gUSE science gateway framework that consisted of the graphical WS-PGRADE workflow editor layer and the gUSE workflow management layer [12]. Using the DCI Bridge [12] and Data Avenue services [13] WS-PGRADE workflows can be executed in various cluster, grid and cloud infrastructures meanwhile accessing various types of storages. This is a popular toolset of defining and executing scientific workflows [14] but it needs a large and heavy service stack that requires expertise to set up and operate.

With the new Flowbster/Occopus concept our objective is to simplify the data pipeline (workflow) creation and operation activities. For example, instead of maintaining a science gateway the scientist herself can initiate the execution of an already developed workflow (data processing service pipeline) in a target cloud by some simple clicks and by giving the URL of the data location where the data set to be processed is stored and where the processed data should be written back. As a result, the workflow will be deployed in the target cloud and will enable the flow of data from the source data storage to the target data storage. The deployed workflow can be considered as a virtual infrastructure that processes and transforms the data meanwhile it flows through this virtual infrastructure. When the data is processed the virtual infrastructure can be removed (by the user) from the cloud and hence long term maintenance like in the case of WS-PGRADE/gUSE based science gateways is not needed.

The data to be processed by a Flowbster workflow represents a data stream. The elements of the stream are the data elements and Flowbster works as a stream processing workflow system that can exploit both vertical (see in Section 1) and horizontal parallelism. Horizontal parallelism is exploited when parallel branches are in the workflow. This is the most natural way of exploiting parallelism both in orchestration and

choreography based workflow systems. There is a third type of parallelism that can also be exploited in both types of workflows. This is what we call node scalability parallelism. It means that if the processing of a node (workflow task) becomes too slow then several instances of the node can be applied in parallel in order to accelerate its work. Typical example of the usage of node scalability parallelism appears in the parameter sweep workflow pattern (see later).

The Flowbster/Occopus concept is also a layered concept. Here we distinguish four layers from bottom to top:

1. Occopus cloud deployment and orchestration layer
2. Flowbster workflow system layer
3. Flowbster application description layer
4. Flowbster graphical design layer

Occopus serves to automatically deploy and manage the data pipeline in the target cloud. It is described in detail in Section 3.

The Flowbster workflow system layer defines its uniform building block (that implements the workflow tasks and will be explained in detail later in Section 4.1) and execution framework by which complex data pipelines can be built. Flowbster is based on the service choreography concept where the workflow tasks as services autonomously work meanwhile communicating with other such services. The beauty of Flowbster is that it uses a uniform building block concept for defining these workflow task services. This uniform building block can be customized according to the applied workflow execution pattern and the required task functionalities.

Here we mention as examples only two typical workflow execution patterns: the parameter sweep pattern and the if-then-else pattern.

The simplest parameter sweep processing pattern consists of three service (task) types:

1. Generator task
2. Worker task
3. Collector task

Figure 1 shows the basic architecture of a Generator-Worker-Collector parameter sweep processing pattern. The role of the generator is to split a large incoming data element into N smaller ones and output these generated small data elements as an output stream $(0, 1, 2, \dots, N)$ in Fig. 1). This output stream is

taken as an input stream and processed by the worker node. Meanwhile the input stream is processed the worker node produces an output stream and passes it to the collector node. The collector collects all the elements of this output stream processes them and merges them into a single data element. This single data element can be placed on its output and can be considered as the result for the incoming data element that originally arrived to the input of the generator node.

The worker node typically represents a bottleneck in the processing of the parameter sweep pattern since meanwhile the generator and collector are executed only once the worker should be executed on a stream as many times as many data elements are in the stream. In order to accelerate the processing of this data stream the node scalability parallelism should be applied for the worker node. It means that several instances of the worker node are created and the input stream of the worker is evenly distributed among its instances. Figure 2 shows the case when three instances of the worker node are created (W , W' and W'') and it also shows an example how the data elements of the incoming data stream of W is distributed among the instances of W .

If the instances are created statically at deployment time based on the scalability parameters given by the user the so-called static node scalability is implemented. Otherwise, if the instances are generated at run time dynamically based on the actual load of the worker node dynamic scalability is realized. Flowbster currently supports only static node scalability as described in Section 4.6. Supporting dynamic node scalability is under investigation and the possible options are described in Section 5. Notice that Flowbster can support both vertical, horizontal and static node scalability parallelism. This makes Flowbster an efficient system to implement large and complex data processing pipelines.

The second workflow execution pattern we show as an example is the if-then-else pattern. It requires two special node types:

- Fork
- Join

Fork enables to direct elements of an incoming data stream to different target nodes in the workflow depending on the value of the incoming data element or on the status of the node. The join node enables to merge incoming data streams into a single output

Fig. 1 Generator-Worker-Collector parameter sweep processing pattern



stream. Notice that fork and join nodes can be easily formed from the uniform Flowbster nodes since their input numbers, output numbers and functions to be processed can be defined without any restrictions.

As we have seen the main concept of Flowbster programming is that Flowbster provides the uniform node building blocks and users should customize them in order to create the concrete data processing pipeline. The Flowbster application description layer has got exactly this role. It enables for the user to

- define graph topology of the data processing workflow by defining the required number of input and out data arcs of the individual nodes and their connections
- define the functionality of the individual nodes

Since the workflow will be deployed and managed in the target cloud by Occopus this definition should be done according to the Occopus node description language. This would require the knowledge of this language that could be too complicated for end-user scientists. Therefore, we have defined a new graphical layer on top of this layer. This is called Flowbster graphical design layer. It provides an easy and intuitive graphical user interface to design the workflow layout and to define the functions for the various workflow nodes. Scientists can use this graphical layer to draw the graph view of the required workflow and then this layer will automatically create the Occopus node descriptors by which Occopus can deploy the workflow in the target cloud.

The paper will describe all these four layers in detail in the next four sections and will also explain their potential usage.

3 Occopus Cloud Deployment and Orchestration Layer

3.1 Concept

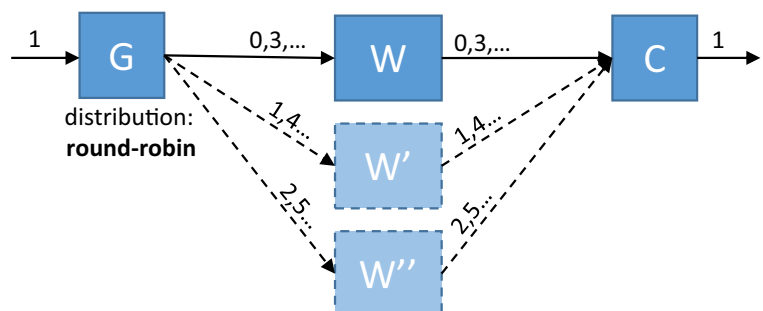
Occopus is an orchestrator tool to build network of nodes containing interconnected services forming a virtual infrastructure. Each node is built up by virtual machines executing services or applications. The configuration and initial setup of nodes can be performed by contextualization and configuration management tools. The built-up infrastructure can then be continuously maintained and the health of nodes can be monitored to detect and recover faulty nodes.

Occopus has a pluggable architecture to cooperate with different types of resources, with different types of configuration management (CM) tools, to apply different contextualization methods and procedures. As a result Occopus can be utilized in a broad range of environments by applying any combination of its plugins.

3.2 Describing an Infrastructure

In order to specify a particular virtual infrastructure for Occopus, one needs to provide an infrastructure description and node definition(s). Notice that the infrastructure node mentioned here is not the same as the workflow node. Unfortunately, the term node is used both for infrastructures and for workflows. In the case of workflows, nodes represent tasks to be executed by the workflow while in virtual infrastructures nodes represent virtual machines or containers that provide the execution infrastructure for the workflow tasks.

Fig. 2 Scaling of worker nodes



The *infrastructure description* is the most abstract form of defining the infrastructure, it does not contain any implementation-dependent details. It contains two big sections: list of nodes and dependencies.

List of nodes contains name, type, implementation selector, scaling information and variables for each node (virtual machine or container) of the virtual infrastructure. Notice that only name and type are obligatory for the nodes. Type is a reference to a node definition (explained later). Node definition selector is optional and may perform selection among multiple implementations (definitions) of a certain node. For example, in Fig. 3 Node B uses selector to select definition B₂ while Node A does not. It means that for Node A Occopus can select any definition (current selection is random).

Scaling-related information covers the minimum and maximum number of instances a node may have during the lifetime of the infrastructure. Later this section may include rules and policies how the actual number of instances should be calculated in case of auto-scaling. The variables section contains user defined key-value pairs, these variables can be referred later in other parts of the infrastructure description or in node definitions. This is mainly used to make certain parameters in node definitions tunable from the infrastructure description.

Dependencies in the infrastructure description define if a certain node requires the existence of another node before coming to live. Dependencies are defined by dependency pairs like B depends on A, which means node A must be deployed before node B (as in Fig. 3). The entire dependency graph must be described by dependency pairs among nodes.

Node definition contains details how a certain node must be implemented. A node might have more than

one implementations. For example, one of the implementations can describe how the node can be deployed on the Amazon cloud, while another one defines the same for a particular Openstack cloud. The selection among implementations can be delegated to Occopus (i.e., random as mentioned above) or can be performed by using a selector in the infrastructure description as depicted in Fig. 3.

Focusing on the implementation of a node there are one obligatory ('resource') and three optional ('contextualization', 'config_management', 'health_check') sections to be specified (see Fig. 4).

3.2.1 Resource

The resource section of a node definition specifies the most important and obligatory parameters to inform Occopus where and how to create a virtual machine (or container) for that particular node. In this section, there are two obligatory parameters: type and endpoint. Type informs Occopus which protocol the interface located at endpoint applies. The example in Fig. 5 defines an 'EC2' type of interface. The rest of the parameters in this section are protocol specific parameters. As it can be seen, EC2 interface requires name of region, id of image and instance type to instantiate a particular virtual machine to implement a node.

The credential information is stored in a separate authentication file for all possible interfaces and particular endpoints. Currently, ec2, nova, docker, occi

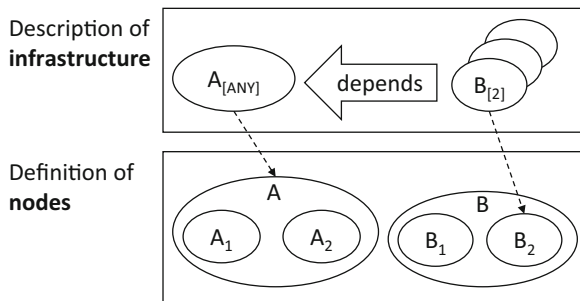


Fig. 3 Relation among infrastructure description and node definition

Node definition

```

MyNode:
-
  resource:
    ...
  contextualisation:
    ...
  config_management:
    ...
  health_check:
    ...
....

```

Fig. 4 Sections of a node definition

Node definition

```
....
resource:
  type: ec2
  endpoint: http://ec2.endpoint.com:4567
  regionname: ROOT
  image_id: ami-00001234
  instance_type: m1.small
....
```

Fig. 5 Resource section of a node definition

and cloudbroker type resources can be used. Each of them has their own list of parameters to be defined in the resource section.

3.2.2 Contextualization

Contextualization provides the possibility to apply user defined configuration and initial settings for the newly created virtual machine at startup by executing scripts, creating config files, adding authentication keys, etc. Currently, contextualization is supported through the cloud-init tool, which is a de facto standard. In a node definition one may add a complete cloud-init configuration file which is then passed as user-data when launching a new virtual machine. However, an additional extension is added for handling the contextualization information.

3.2.3 Config Management

Occopus is able to utilize external config manager tools, like Chef. In cases when nodes are built by config managers, Occopus is able to cooperate with them to organize nodes into an infrastructure. For example, in case of Chef the endpoint of Chef server and the list of recipes must be defined (see Fig. 6). Occopus also supports the use of Puppet and since this is a pluggable system other config management tools can easily be added.

3.2.4 Health Check

In this section, one can define how Occopus should check if a particular node is still alive on that node. There are built-in methods like ping, port checking, URL checking and mysql database connectivity checking. Occopus assumes healthy services on the node until all checks are executed successfully. A

Node definition

```
....
config_management:
  type: chef
  endpoint: https://chef.server.com
  run_list:
    - recipe[database-setup::db]
....
```

Fig. 6 Example for config_management section in node definition

timer is started once at least one of the checks fails and node is restarted after a timeout period—defined in the node definition – has been reached. In the example (see Fig. 7), all possible checks are represented.

3.3 Overview of Functionalities

Once all the descriptions are finalized, the infrastructure is defined, it is ready to be sent to Occopus for deployment. There are three variants of deploying an infrastructure by Occopus regarding the interface: by using command-line utilities, by sending commands to the Occopus service through REST interface and finally, Occopus can be built into an application as a library providing an API through which the application can utilize Occopus functionalities.

The four basic functionalities provided by Occopus for Flowbster are as follows:

Node definition: health_check

```
...
MyNode:
...
health_check:
  ping: True
  urls:
    - http://{ip}/myapp
  mysqldb:
    - { name: 'dbname', user: 'dbuser', pass: 'dbpass' }
  ports:
    - 22
  timeout: 300
...
```

Fig. 7 Example for health_check section in node definition

- Building workflow infrastructure: based on the descriptors Occopus launches all virtual machines and configures workflow tasks on these virtual machines.
- Maintaining the workflow infrastructure: Occopus periodically performs scanning the health of services on nodes and reports errors.
- Scaling up or down the nodes of a workflow infrastructure: the target number of instances for a given node can be modified by adding or removing instances in order to realize node scalability parallelism.
- Destroying a workflow infrastructure: Occopus performs a graceful shutdown for each of the node in the infrastructure by explicit user request.

The set of functionalities—build, maintain, scale and destroy—covers the most important functionalities for managing the life-cycle of a flowbster virtual infrastructure.

4 Flowbster Workflow System Layer

A Flowbster workflow is a directed acyclic graph where the nodes represent the data processing tasks and the arcs represent the data transfer between the tasks. The graph topology is designed by the Flowbster application description layer (textual definition layer) or by the Flowbster graphical design layer. The Flowbster workflow system layer defines how to create the workflow tasks in the cloud and how to interconnect them to realize the workflow graph topology.

The main motivation behind this layer was to create a very simple, easy-to-use, basic building block for workflows to provide low-level service for receiving, executing and forwarding a piece of data item belonging to a data stream among the tasks of the workflow. The generic Flowbster node was designed to be a simple universal building block based on which complex network of processing nodes, i.e., workflows can be built by customizing and interconnecting these universal building blocks.

4.1 Structure

In order to implement the basic building block of Flowbster the focus was to design a very simple node

structure. The Flowbster node is constructed by three basic components (Fig. 8):

- Receiver
- Executor
- Forwarder

The *Receiver component* is a service continuously waiting for input data items to stage down on the node for execution. Whenever a new input data item arrives, the receiver decides whether a data set can be passed for execution. The Receiver continuously registers the pieces of data items and monitors if all the required inputs, i.e., all inputs of the preconfigured executable are available. The pieces of input data items for a particular execution can arrive in any order. Once all the inputs are available for an execution, the inputs are marked as ready-to-run and passed for the Executor component.

The *Executor component* is the most simplified part of the Flowbster node. It continuously monitors the ready-to-process input data items. When one is found, the execution is prepared, i.e., separate working directory with properly named inputs and all configuration files required by the application are staged. The execution of the predefined application is done as a next step and all the outputs are finally created. Once the execution is finished and the outputs are available the

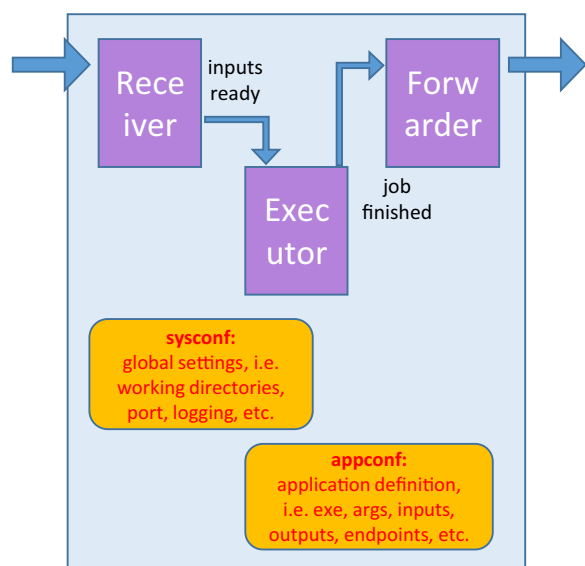


Fig. 8 Basic components of a Flowbster node

output data package is marked as ready-to-forward and passed to the Forwarder component.

The *Forwarder component* is the link between a successful run and the Receiver of a connected node. It means it has the task to properly index and forward the output data items to one or more receiver node(s). During indexing of the outgoing data item the indices are calculated based on the indices of data items of the run and on the sequential number of the execution (see details in Section 4.4). For each particular output the Forwarder has the details of the target endpoint of the next Receiver specified.

4.2 Configuration

The configuration consists of two main parts: system and application. The system wide configuration contains settings independent of the target application or executable. It contains for example port definitions, working directories, settings for logging, etc. The parameters defined here are not affecting the execution of the application only the general behavior of the three Flowbster components.

Application configuration customizes the node for a particular workflow task by describing the details of the application to be executed by the workflow task and the interconnection details for the connected workflow tasks. The application-related details are for example executable path, arguments, list of input and output files necessary for the application. The interconnection related details are URL and name of input for the target Flowbster node to which the output data must be forwarded. Notice that the application-related details customize the Executor service while the interconnection details are used by the Receiver and Forwarder.

Once the system and application config files are deployed and the components are launched on the Flowbster node, the node is ready to process incoming data sets. The example app config file (see Fig. 9) on the Flowbster node will execute ‘myexe.sh’ command (after downloaded and unpacked the file from tgzURL with ‘-i in -o out’ command line parameters, and input data stored in file called ‘in’ when a new data arrives marked as ‘in’ input. When the execution is finished, the file ‘out’ is forwarded to the next Flowbster node located at targetURL as ‘in’ input file. This very simple example shows the simplicity of the basic building block of the Flowbster node.

Flowbster app config

```
executable:
  filename: myexe.sh
  tgzURL: http://somewhere.com/myexe.tgz
inputs:
  -
    name: in
outputs:
  -
    name: out
    targetname: in
    targeturl: http://flowbster.node.com:5000/flowbster
arguments: -i in -o out
```

Fig. 9 Example application config file of a Flowbster node

4.3 Feeding, Gathering Data Items

Flowbster nodes can have either internal input/output ports or external input/output ports. Internal input/output ports serve to transfer data between the tasks of the Flowbster workflow. External input ports serve to get the data to be processed from an external file system or database. In order to keep the Flowbster workflow nodes uniform we have to create a new component that can access the external file systems or databases and transfer the fetched data according to the input port protocol of Flowbster nodes to the target external input ports of the Flowbster workflow. This new component is called Feeder. This is not part of Flowbster since there are many different kind of file systems and databases. Usually the Feeder is a very simple service that should be written by the user. However, as an example how to write such a Feeder service we have created two types of Feeders. The first one can read data from a local file system and the other from an S3 type cloud storage. The source code of these Feeders just like the code of the whole Flowbster/Occopus system is open source and downloadable from GitHub [15, 16]. Based on this downloadable Feeder code, users can write code to access other types of storages and databases. In the case of storages the usage of the Data Avenue service makes even simpler to write Feeder code for a large set of different storage types [13]. Similarly, to the Feeder component we need a Gather component that writes the result data of the Flowbster workflow to the target file systems or databases. The Gather component takes the workflow results from the external output ports of the workflow using the Flowbster data communication protocol. In order to give example how to write Gather components with access to different file systems and storages

we have created two types of Gatherers. The first one can write data to a local file system and the other to an S3 type cloud storage. The source code of these Gather variants are downloadable from GitHub, too [16]. Using the Feeder and Gather components the entire data processing system looks like as shown in Fig. 10.

4.4 Parameter Sweep Pattern and Its Metadata Support

One of the most frequently used workflow pattern was the parameter sweep one in the SCI-BUS project where the vast majority of the more than 30 user communities exploiting the WS-PGRADE workflow system were actually using this pattern [14]. Therefore, we selected it as the example pattern through which we show the power of the Flowbster/Occopus concept. Flowbster supports the parameter sweep execution pattern as described in Section 2 and illustrated in Figs. 1 and 2. A generator always adds 2 new components to the metadata of the incoming data elements in order to produce the metadata of the data elements of the output data stream it generates. These two new components of the metadata are:

- Index of the newly created data element inside the output data stream (for each output data element).
- The number of data elements belonging to the output data stream (common for all the output data elements).

When a generator node (G in Fig. 1) emits a data stream with multiple data elements, two new metadata components are automatically assigned to each of them. Flowbster forwards each data elements with its

index and the number of generated elements as metadata, i.e., $\{0,N\}$, $\{1,N\}$, $\dots\{N-1,N\}$ index pairs will be generated as metadata. Worker node (W in Fig. 1) keeps the indices untouched, i.e., node W will copy the metadata of an input data element to its corresponding output element as metadata. In Flowbster, a Collector node (C in Fig. 1) will know the overall number of data items in a stream based on the second metadata component in the message. Based on this information collector waits until all the data elements belonging to a certain data stream arrives, i.e., it collects all the data elements of the incoming data stream. At this point the collector function is activated and it processes the data elements of the stream. The result will be placed on its output as a single data element. This is achieved by removing the two metadata components that were originally added to the data stream by the generator node.

As mentioned in Section 2 generators can be created by customizing a generic Flowbster node as generator. Here customization simply means that at least one output port of the node should be defined as generator output port. The *mask* property serves to define which result files should be selected to be considered as the output of the generator. Similarly, collectors can be created by customizing a generic Flowbster node as collector by defining at least one input port as a collector port. The presence of the *collector* and *format* properties will result in making the given input port work as a collector port. More details will be given in Section 6 where an AutoDock example will be used to explain the actual description of a simple parameter sweep workflow.

So far we showed only the simplest version of the parameter sweep pattern. However, there are many

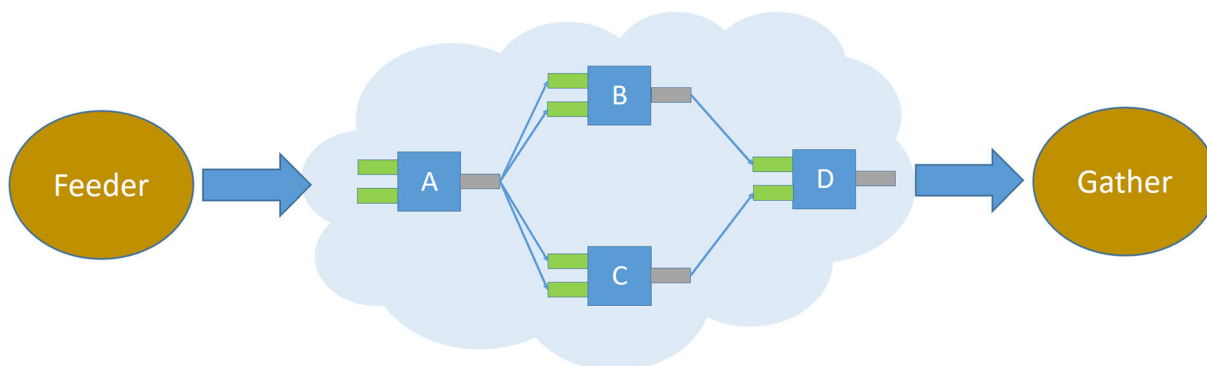


Fig. 10 Handling external inputs and outputs of Flowbster nodes

possibilities to build much more complex parameter sweep workflow pipelines where the number of generator, worker and collector nodes are not restricted to one. A typical example for a more complex, but still basic parameter sweep workflow is the application of embedded parameter sweep patterns as shown in Fig. 11 by the $G1 \rightarrow G2 \rightarrow W \rightarrow C2 \rightarrow C1$ chain of nodes. Here $G2 \rightarrow W \rightarrow C2$ is embedded into the $G1 \rightarrow W' \rightarrow C1$ pattern by simply substituting W' with the embedded chain of $G2 \rightarrow W \rightarrow C2$.

In Fig. 11 the number of data items per execution, number of executions and number of data items altogether are shown to follow how the execution is performed by Flowbster in this workflow. Focusing on the embedded pattern we can see that for each incoming item node W' emits one result. However, inside node W' five items generated by node $G2$, each of them processed by node W and finally grouped by node $C2$ for one execution providing one final result. Looking at the overall workflow, $G1$ emits a data stream with three data elements. This data stream is processed by W' and sends the output data stream to $C1$. When the whole data stream arrived to $C1$ it processes the data stream and places a single data on its output. Inside W' , $G2$ generates 3 output data streams from the three data elements of its input data stream. Each of these stream will contain five data elements with the following metadata: $(i1, N1, i2, N2)$ where $N1 = 3$, $N2 = 5$, $i1$ can be any of $\{0, 1, 2\}$ and $i2$ can be any of $\{0, 1, 2, 3, 4\}$. $C2$ collects the data elements of the three incoming data streams. $N1$ specifies the number of arriving data streams and $i1$ identifies the actual data streams. $N2$ specifies how many data element should be collected for each incoming data streams and $i2$ identifies the actual data elements in the input data streams identified by $i1$. Once a data stream completely arrived $C2$ processes it and places the output data element on its output. This data element will belong to the data stream emitted by $G1$

and its index inside that stream is identified by $i1$. Notice that $C2$ removes the last two components of the metadata when places the result data on its output. In general, a generator creates new data streams by adding an (I, N) tuple to the metadata and a collector transforms an incoming complete data stream into a single result data element by removing the last (I, N) tuple.

This rule ensures the proper handling of embedded workflow patterns.

Notice that generally Flowbster workflows can be embedded into other Flowbster workflows provided that the workflow node substituted with the embedded workflow has the same number of input and output ports with the same data semantics as the embedded workflow has. That's why Feeder and Gather must not be part of Flowbster. If they were part of Flowbster they would prevent the embedding of workflows.

The next example (in Fig. 12) shows two paths on the way toward node $C2$. Data items on the different input ports of node W are combined as cross-product, so the number of execution of a particular application is the multiplication of the number of data items on all its input ports. $C2$ groups the combination of all data elements generated by $G2$ and $G3$ (the nearest generators on the different paths backwards), therefore 25 data elements are processed in one run. Looking inside node W' , node $C2$ can only consider and group elements generated by $G2$ and $G3$, otherwise breaking the border of the embedding pattern.

The example in Fig. 12 shows several combinations of generator and collector nodes to demonstrate a workflow where behavior of the Flowbster nodes can be explained. The figure shows the number of executions, the number of generated data items per execution and the number of data items generated altogether. $MG1$ and $MG2$ nodes are two consecutive Generator nodes with 3 and 5 multipliers. W node applies cross-product on its inputs ($MG2:15$, $G1:5$,

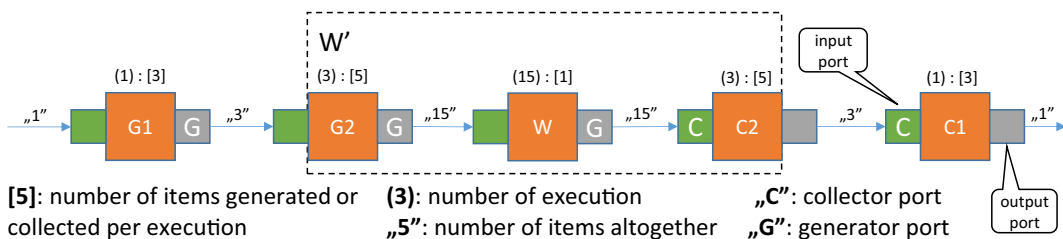


Fig. 11 Generator and Collector nodes in a Flowbster workflow

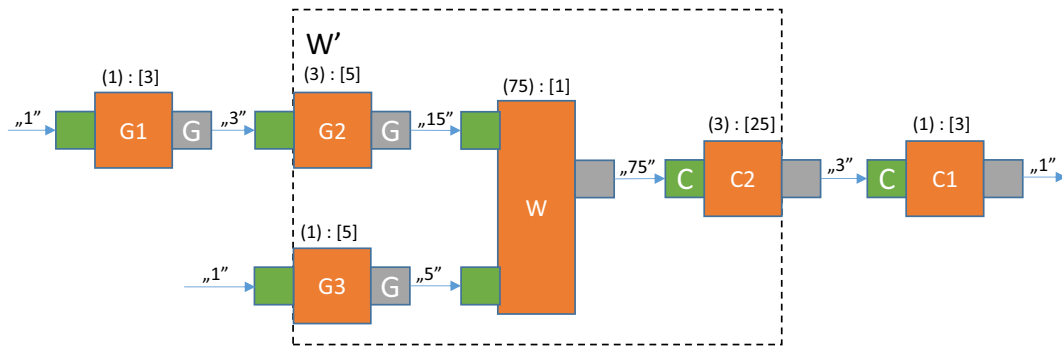


Fig. 12 Generator and Collector nodes in a Flowbster workflow

N1:1) which results 75 pieces of execution where each execution processes 1 combination of data inputs. The node MC1 applies collector port to group the outputs of the nodes belonging to the generators (MG2, G1) belonging to the nearest level (same distance on the way back on the network). MC1 therefore groups 5*5 pieces of items for one execution providing 3 pieces of outputs altogether. MC2 similarly groups the number of items generated on the next level of generators (MG1) to provide 1 element at the end. The rest two paths on Fig. 13 shows simpler cases, with one Generator and Collector (G1, C2) and two Normal nodes (N1,N2).

4.5 Set Up Interconnections by Occopus

To build a network of Flowbster nodes links between outputs and inputs must be harmonized. In order to have a properly configured network, each application configuration's output definition must point to the input of the next nodes. Creating application configuration files for each node by hand is a hard task, so it is

a natural step to integrate Flowbster with a tool providing automatic deployment and configuration of nodes. For this purpose we utilized the formerly introduced Occopus tool.

In order to create a Flowbster node, three executables and two configuration files must be deployed. Deployment of the three executables is a straightforward task, however deployment of the configuration files with proper content is not trivial. With the contextualization and the jinja2 template language support in Occopus, complex file contents can be generated and deployed on the target node. The application configuration related information must be defined for each Flowbster node, however the information on the interconnection of nodes is retrieved from Occopus.

Application related information can be easily stored under the variables section in infrastructure description of Occopus under each node. Figure 14 shows the structure of application information with keywords needed to properly generate the application configuration file for a Flowbster node.

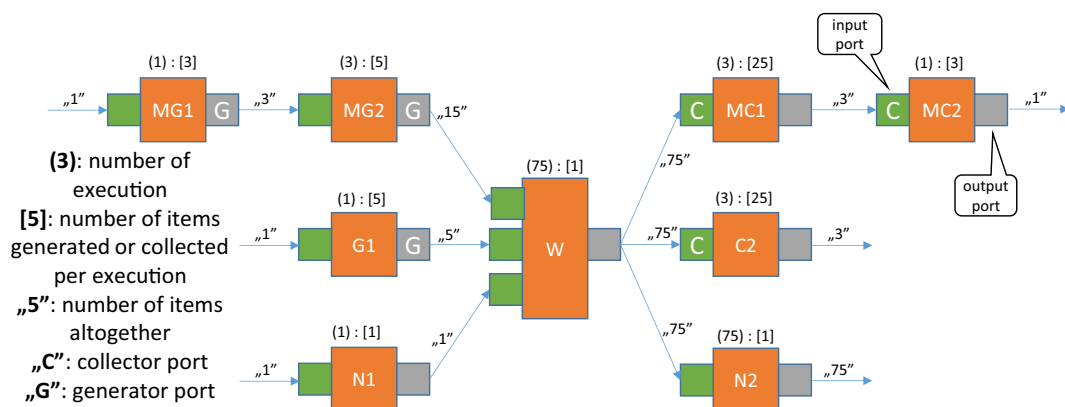


Fig. 13 Example with Generator and Collector nodes in a Flowbster workflow

Flowbster app config in Occopus

```
variables:
  flowbster:
    app:
      exe:
        filename:
        tgzurl:
      args:
        in:
          -
            name:
        out:
          -
            name:
            targetinput:
            targetnode:
```

Fig. 14 Flowbster application configuration in Occopus

The structure of application config in Occopus is similar to the one shown in Fig. 9. The only part that needs explanation is the `targetinput` and `targetnode` attributes of the output section. `Targetnode` must be the name of the node in Occopus to which the output data must be forwarded. During the resolution the contextualization template will replace the name of the `targetnode` with the ip address. Similarly to `targetnode`, `targetinput` means the name of input to which the data input must be forwarded. With these definitions the proper application configuration file can be generated and the interconnections are built.

During the deployment of the Flowbster network, Occopus creates each virtual machine, deploys the executables and creates the configuration files. During node deployment the network topology can be specified by dependency pairs in the Occopus infrastructure description.

4.6 Static Node Scalability

Node scalability is most often used when a workflow node must be multiplied in order to provide enough processing capacity to meet response time requirements as it was explained in Section 2. For example, in a workflow where the structure follows the Generator-Worker-Collector pattern node scalability can be used for the worker node as shown in Fig. 2. In this case multiple data elements emitted by the Generator (G) must be distributed (e.g. with round-robin) among the Workers (W,W',W'') in a way that the Collector (C) receives the union of the outputs of the workers. Scalability can have two versions as described in Section 2:

- **Static scalability** when the user specifies the required number of node multiplications at workflow deployment time
- **Dynamic scalability** when the workflow execution is monitored and if a certain node is overloaded new multiplied copy of the overloaded workflow node is automatically deployed and connected to the workflow at run time without user interaction.

Notice that in both cases the workflow graph contains only a single Worker node as shown in Fig. 2. It is only Occopus that can create (and “see”) the multiple instances of the Worker node either based on the user definition or the monitoring information. Currently Flowbster supports static scalability. In order to scale up a node at startup in Occopus extra scaling section must be provided in the infrastructure description for the node as shown in Fig. 15. In this example the worker node will have exactly 3 instances after building up the infrastructure due to the minimum specified. Optionally, maximum can also be specified for Occopus to limit the number of instances in case dynamic scaling is utilized. In this example the maximum will be equal to minimum by default due to unspecified maximum value.

When Occopus builds the nodes of the workflow in Fig. 2, the dependency in the infrastructure description will force Occopus to keep the following order in node deployment: Collector, all Workers, Generator. The Generator will therefore receive the endpoints of all the worker nodes and hence the Generator can perform the distribution of data elements among the multiple worker nodes as shown in Fig. 16.

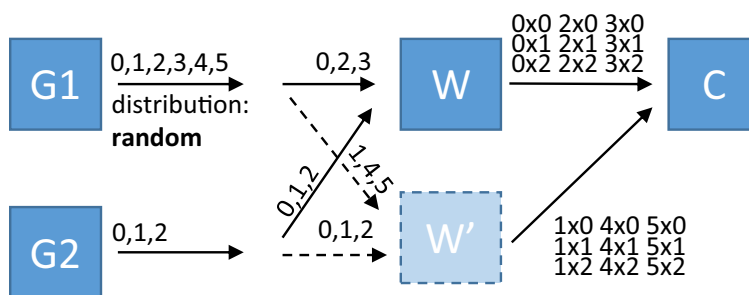
In Flowbster output data elements can be transferred to the target input ports according to two options:

- **Copy (default):** Without any special setting on the output port with multiple endpoints,

```
nodes:
  -
    name: worker
    type: my_worker_node
    scaling:                                     <= scaling
      min: 3                                    <= up to 3 instances
    variables:
      flowbster:
        ...
```

Fig. 15 Scaling up section in Occopus

Fig. 16 Scaling up workers feed by multiple generators in Flowbster



Flowbster will copy each outgoing data element to all the attached endpoints (i.e. to multiple instances) of the target node.

- **Distribution:** When the output port is marked with the keyword “distribution” in Flowbster, the outgoing data elements will be exclusively assigned to one of the attached target nodes. With this option the output data is distributed to multiple instances of the target nodes supporting the scalability concept (for example, the outputs of the Generator are distributed for the three worker node instances W, W', W'' in Fig. 2). Currently two ways of output data distribution is supported: random and round-robin.

Let us take the previous workflow (G-W-C) as an example and attach an additional generator to the worker node. Now, the worker node must process each combination of data elements emitted by the two generators. Figure 16 introduces this situation, where G1 and G2 are the generators, W is a worker scaled up to two instances and C is a collector. In case both of the generators would apply distribution on their outputs, certain combination of the data elements to be processed by the workers would be lost. In order to avoid this situation, one of the generator outputs (G1 with 6 elements) must be configured to apply distribution (e.g. “random” as shown in Fig. 16), while all the others (in the current example only G2 with 3 elements) must simply copy all the output elements to all the attached worker nodes (see Fig. 16). With this technique we can split the combination of data elements (altogether 18) by one of the dimensions. This ensures that all the combination of data elements in this situation are covered by the worker instances (see list of combinations traveling to node C in Fig. 16).

Returning to the previous example on Fig. 13, we can easily come to the conclusion that the node named

“W” can be scaled up if one of the nodes named “MG2” and “G1” is set to apply distribution while the other applies the default copying mechanism.

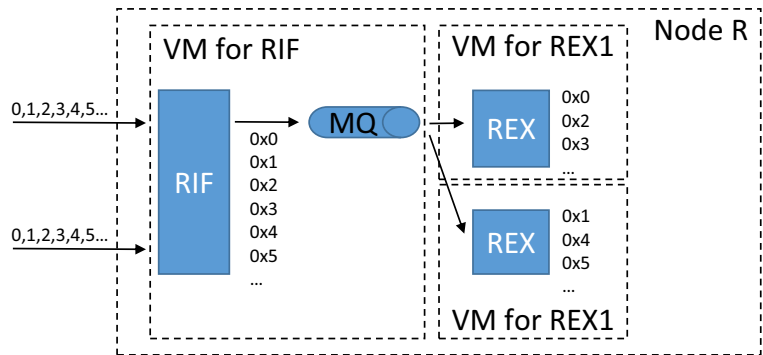
5 Dynamic Scalability Alternatives

In this section we discuss alternatives for supporting dynamic scalability in Flowbster. Occopus is able to dynamically scale up and down the nodes in the infrastructure. The current solution in Flowbster introduced above works only when nodes scales up at startup, no dynamic scaling is supported yet. Dynamic scaling is needed when further instances of a node are required due to e.g. increased load during the execution of the jobs by a node. The next alternatives detail how Flowbster and its architecture could be modified to support scaling at runtime.

5.1 Component-Level Scaling

In the first alternative, dynamic scaling is supported by applying message queues. Message queues are useful to automatically distribute items among the consumers, i.e., to distribute the load. The Flowbster nodes implement a component called “receiver” as input interface (“RIF” in Fig. 17) that performs the receiving of input data elements, the combination of the data elements (cross-product among the inputs) and generation of jobs to be executed by the “executor” components (“REX” on Fig. 17).

Filling up the message queue by jobs, multiple executors can be attached. This automatically balances the load and exclusive job distribution and fault handling, i.e., reschedule jobs assigned to failed executors. This solution provides the possibility to dynamically scale the executor services, however, unfortunately doubles the number of virtual machines required to implement a workflow node.

Fig. 17 Flowbster node with external executors

5.2 Component-Level Scaling with Executor on Master

The previous scalability solution can further be improved by adding an executor component on the master virtual machine of the node. The new layout can be seen in Fig. 18 where additional executor components are only required when unexpectedly high load appears on the first virtual machine of the node.

The advantage is straightforward: one virtual machine is enough to implement a node. However, the additional instances during scale-up is not straightforward, special preparation and support is needed to ensure that the additional instances does not deploy receiver interface (“RIF” on Fig. 18). In both versions, a Forwarder component is attached to each Executor component.

5.3 Workflow-Level Scaling

The most trivial scaling can be performed at workflow-level. This requires a complex solution where each data processing workflow is maintained by a central service. The load of the workflows are measured and when further processing capacity is required, new instance of the overall workflow can be created as depicted on Fig. 19.

This solution may benefit from a central data feeder component which automatically balances of the load generated on the workflows by the data elements. On the other end a data collector component can insert the results to the target storage e.g., a database.

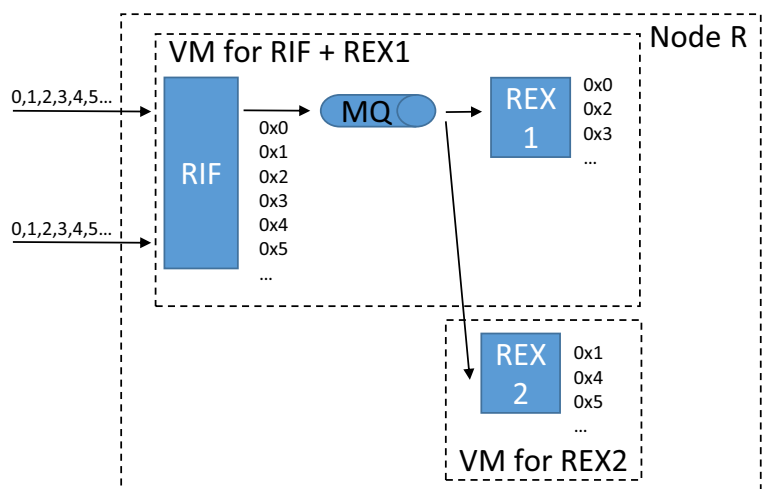
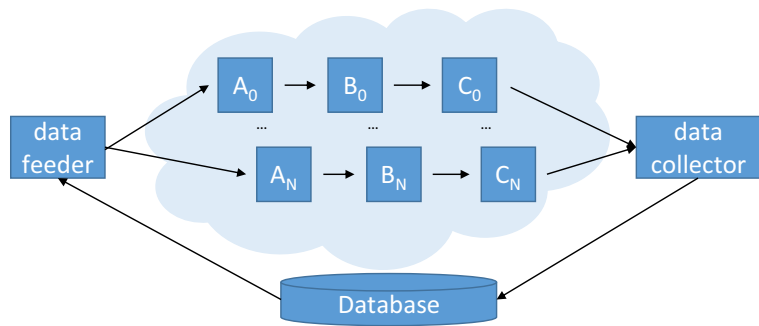
Fig. 18 Flowbster node with external and one internal executors

Fig. 19 Workflow level scaling



6 Flowbster Application Layers

In this section we present the two Flowbster application layers: the description and the graphical design ones. These two layers, building on top of Occopus and the Flowbster framework, enables users (both professional IT developers and end-user scientists) to develop workflow applications on top of managed cloud infrastructures.

The application description layer is responsible for representing the workflow application in a textual form, building on the features provided by the Flowbster/Occopus framework. This layer requires the knowledge of clouds and the Occopus descriptor language and hence this layer is offered for professional IT developers. They can develop optimized Flowbster workflows that could be used by end-user scientists. The graphical design layer offers a convenient graphical tool for designing Flowbster workflows without the need to have knowledge on clouds and Occopus description language. Therefore, this user interface is offered for end-user scientists in order to quickly and conveniently develop prototype Flowbster workflows without waiting for the IT people to develop the optimized workflows.

6.1 Flowbster Application Description Layer

As discussed earlier in this paper, the Flowbster framework offers convenient building blocks to create complex workflow applications. The description layer relies on this framework for composing a workflow application. The task of the user exploiting this layer is to create an Occopus infrastructure description by customizing the Flowbster nodes properly, i.e., by defining all the required properties.

6.1.1 Single-Node Workflow

The very basic Flowbster workflow is a single-node workflow. Such workflow has only one single node running one executable, accepting a set of input files, and producing a set of output files. The example in Fig. 20 shows the descriptor of such a single-node workflow where the single node runs a “copy” service.

In the above example, we have only one single node in the infrastructure called *Copy*. The type of the node is *flowbster_node*, which refers to the uniform node provided by Flowbster. The task of this descriptor is to instruct Flowbster how to customize the uniform node to work as a “copy” service. In order to enable node scaling the user can specify the minimum and maximum number of instances that can be created from this node. In the current example the enabled number of instances is 1 for simplicity.

The variables inside the node descriptor define properties of the executable to be run inside the given node. First, the name (*filename*) and the download location (*tgzurl*) of the executable are defined. The Flowbster framework will fetch the executable TGZ archive from the given URL, expand it into a temporary directory, and invoke the file set inside the *filename* property when executing the application. Next, the command line arguments (*args*) are defined for the application. Afterwards, the input and output files’ properties are set. The common property for both input and output files is the *name* property, specifying the name as the application will open the files. In the output file section, three additional properties specifying the target of the produced output files is set: the *target-name* specifying how the target file will be called, the *targetip* and *targetport*, specifying the location of the target data service where the output file will be written.

Fig. 20 Descriptor of a single-node workflow

```

infra_id: copy_workflow
user_id: foo@bar.com
name: copy_workflow

variables:
  flowbster_global:
    gather_ip: &gatherip 192.168.1.1
    gather_port: &gatherport 5001
    receiver_port: &receiverport 5000

nodes:
- &Copy
  name: Copy
  type: flowbster_node
  scaling:
    min: 1
    max: 1
  variables:
    flowbster:
      app:
        exe:
          filename: copy.sh
          tgzurl: http://foo.bar/copy.tgz
          args: -i in_file -o out_file
        in:
          -
            name: in_file
        out:
          -
            name: out_file
            targetname: copy_result
            targetip: *gatherip
            targetport: *gatherport

```

Besides the local (node-level) variables there are also global ones. The global variables define those properties each node inside the workflow will receive. These are the IP address and listening port of the Gather component where the result data should be transferred, and the listening port of the receiver components inside each Flowbster node.

6.1.2 Multi-node Workflow

Multi-node workflows are capable of exploiting the true power of the Flowbster framework. Beside the basic executable and input/output file properties, one can define data dependency among the workflow nodes.

In this section we use as an example the well-known AutoDock Vina [17] application and its workflow that was originally developed for WSPGRADE/gUSE based science gateways [18]. The AutoDock Vina tool can be used to perform molecular docking simulations. For this, it needs a configuration file, a receptor molecule and a set of ligands to be docked against the receptor molecule. The different dockings are completely independent from each other thus the different dockings can be run in parallel. In fact, the AutoDock Vina application is a good example for the parameter sweep processing pattern (Generator -> Worker -> Collector) described in Section 2.

Figure 21 shows the AutoDock workflow descriptor. This workflow contains three nodes implementing a parameter sweep (PS) version of the AutoDock Vina molecular docking tool: a *GENERATOR* node, a *Vina* node (with 16 node instances) and a *COLLECTOR* node. The Vina node as the worker of the PS workflow pattern can be multiplied to support node scalability parallelism. In order to achieve it the *scaling* parameters *min* and *max* are set to 16. As a result Occopus will deploy 16 Vina nodes in 16 virtual machines that are connected to the rest of the workflow nodes as explained in Section 2.

The task of the *GENERATOR* node, after receiving the Vina configuration file, a receptor molecule and a set of ligands to be docked against the receptor molecule, is to split the input ligand set into as many parts as many has been specified in the command line arguments (1024 in our example). The split ligand set is distributed in a random manner among the 16 instances of the *Vina* node. The receptor molecule and the Vina configuration file are sent to all the instances of the *Vina* node to perform the docking simulations. Finally, the results of the Vina docking simulations from the *Vina* node instances are sent to the *COLLECTOR* node, which selects the five best docking results and passes to the Gather component as the results of the AutoDock workflow.

```

infra_id: autodock-3node
user_id: foo@bar.com
name: autodock-3node

variables:
  flowbster_global:
    gather_ip: &gatherip 192.168.1.1
    gather_port: &gatherport 5000
    receiver_port: &receiverport 5000

nodes:
  - &GENERATOR
    name: GENERATOR
    type: flowbster_node
    scaling:
      min: 1
      max: 1
    variables:
      jobflow:
        app:
          exe:
            filename: execute.bin
            tgzurl: http://foo.bar/gen.tgz
            args: '1024'
          in:
            -
              name: input-ligands.zip
            -
              name: vina-config.txt
            -
              name: input-receptor.pdbqt
          out:
            -
              name: output.zip
              mask: "output.zip*"
              distribution: random
              targetname: ligands.zip
              targetnode: Vina
            -
              name: config.txt
              targetname: config.txt
              targetnode: Vina
            -
              name: receptor.pdbqt
              targetname: receptor.pdbqt
              targetnode: Vina
  - &Vina
    name: Vina
    type: flowbster_node
    scaling:
      min: 16
      max: 16

```

```

variables:
  jobflow:
    app:
      exe:
        filename: vina.run
        tgzurl: http://foo.bar/vina.tgz
        args: "
        in:
          -
            name: ligands.zip
          -
            name: config.txt
          -
            name: receptor.pdbqt
        out:
          -
            name: output.tar
            targetname: output.tar
            targetnode: COLLECTOR
  - &COLLECTOR
    name: COLLECTOR
    type: jobflow_node
    scaling:
      min: 1
      max: 1
    variables:
      jobflow:
        app:
          exe:
            filename: execute.bin
            tgzurl: http://foo.bar/coll.tgz
            args: '5'
          in:
            -
              name: output.tar
              collector: true
              format: "output.tar_%i"
          out:
            -
              name: best.pdbqt
              targetname: COLLECTOR_result
              targetip: *gatherip
              targetport: *gatherport

dependencies:
  -
    connection: [ *GENERATOR, *Vina ]
  -
    connection: [ *Vina, *COLLECTOR ]

```

Fig. 21 AutoDock workflow descriptor

The following differences can be identified regarding input and output files when comparing with the single node workflow:

1. Connecting ordinary output ports to ordinary input ports: The *targetnode* property in the “out” part of a node descriptor specifies the name of the target node to which the produced output file (or files) should be sent to. In this way node outputs can be connected to inputs of target nodes.
2. Defining generator output port: The *mask* property serves to select files to be considered as the generator outputs. It instructs the Flowbster framework to create output file names according to a regular expression. For example, `output.zip*` is defined

in the case of the first output of the GENERATOR. Any file matching the regular expression will be considered as a generated instance of the given output file. The optional *distribution* property defines the way of distributing the generated output file instances. Two distribution methods are supported: *random* (choosing randomly from the set of target node instances) and *round-robin* (distributing file 1 to node instance 1, file 2 to node instance 2, etc.). If the *distribution* property is omitted, the generated output file set is sent to all instances of the target node.

3. Defining collector input port: the presence of the *collector* and *format* properties will result in

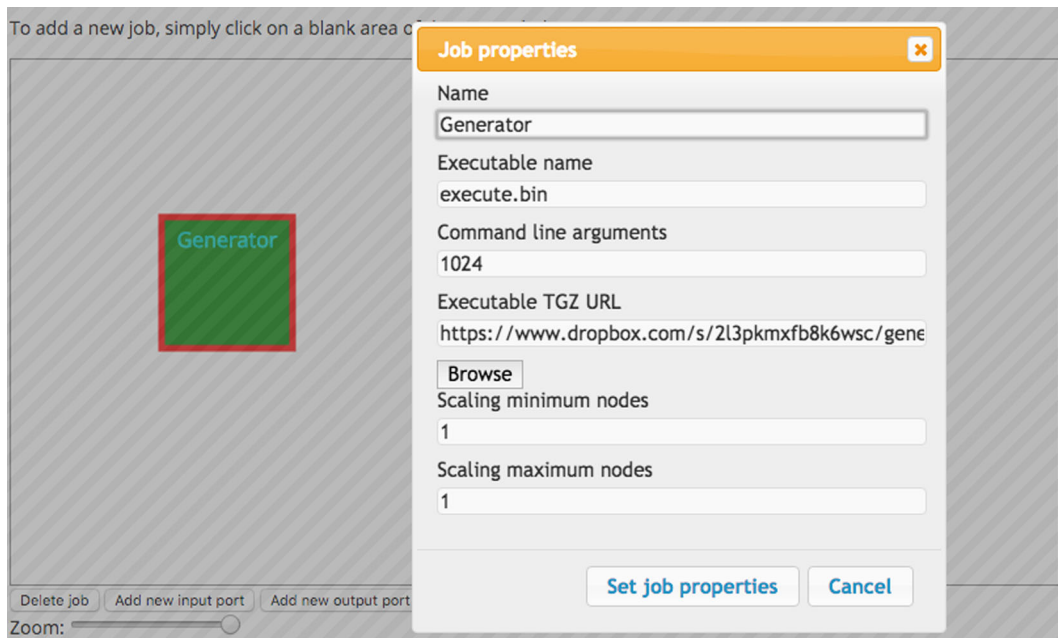


Fig. 22 Job property dialog and graph workflow interface

making the given input port work as a collector port. It means that the COLLECTOR node can only be executed when all the files generated by the Vina instances have arrived to the COLLECTOR node as explained in Section 2.

To sum up, the above workflow will be deployed in a target cloud as a set of 18 virtual machines that are connected to each according to the Flowbster workflow topology.

6.2 Flowbster Workflow Design Layer

Although the workflow description layer is relatively simple, it is still not a convenient interface for end-user scientists. A straightforward step toward providing a user-friendly interface for creating Flowbster-based workflows is a graphical user interface, which enables creating the workflow layout structure, and setting executable and input/output file properties for the different nodes in the workflow.

Instead of creating a user interface from scratch, we have decided to evaluate existing user interface frameworks for defining graphs. A number of graph and workflow definition frameworks exists, and, we have selected to create the Flowbster graph editor based

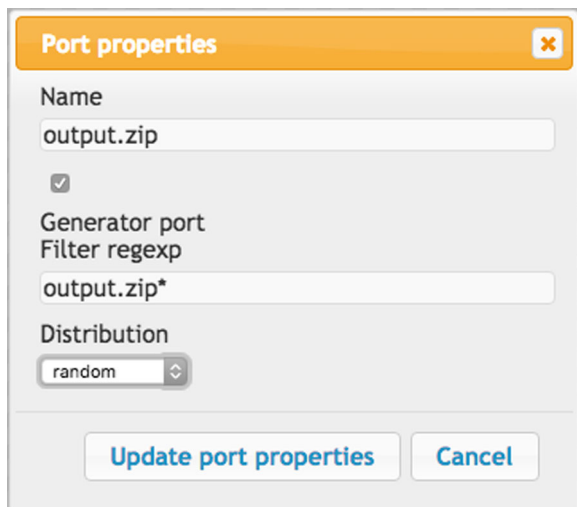
on JointJS,¹ with the additional discrete event system specification (DEVS) extension.² The advantage of using JointJS is that it can be extended in many different ways, is based on pure Javascript, HTML and CSS, and finally, is able to serialize and deserialize the graph into and from JSON. These advantages did allow us to produce a usable Flowbster editor very quickly, which can be deployed onto a big variety of website hosting environments.

Figure 22 shows the job property dialog and the Flowbster workflow editor's main view in the background. New jobs can be created by simply clicking onto a blank area of the main view, which will result in bringing up the Job properties dialog, where the user can enter properties of the job as shown in Fig. 22.

Once a job has been selected (by clicking on it), the user can add new input and output files by clicking the *Add new input port* and *Add new output port* buttons (see in the bottom of Fig. 22). The files' properties can be set by double-clicking on the circles attached to the job. Figures 23 and 24 show the output and input port definition dialogs, respectively.

¹<http://www.jointjs.com/>

²<http://www.jointjs.com/demos/devs>



Port properties

Name
output.zip

☒

Generator port
Filter regexp
output.zip*

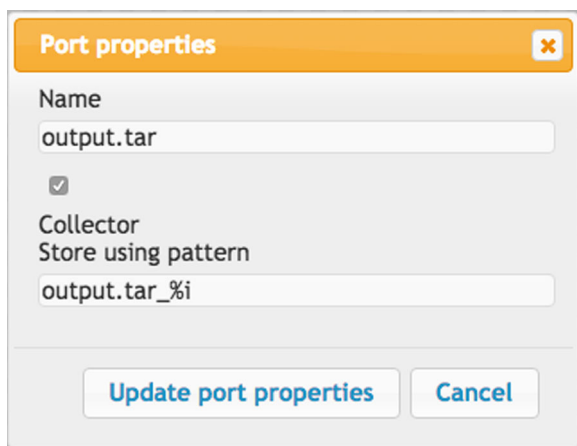
Distribution
random

Update port properties Cancel

Fig. 23 Output port property dialog

Finally, connections between nodes of the workflow can be defined by dragging an output file of the job, and connecting it to the input port of another job. The graphical representation of the AutoDock workflow described earlier is shown in Fig. 25.

The final task for supporting Flowbster workflows using the graphical user interface is the conversion between the internal representation of the graphical workflow editor and Occopus. JointJS can dump and parse the graph along with its properties into a JSON representation. This JSON representation contains two main properties: the list of cells and the list of links. The cells property enumerates the nodes of the workflow along with their properties, and links



Port properties

Name
output.tar

☒

Collector
Store using pattern
output.tar_%i

Update port properties Cancel

Fig. 24 Input port property dialog

property enumerates the connection between the different nodes' output and input files. As a result transforming the JSON representation into the Flowbster/Occopus descriptor like the one in Fig. 21 is straightforward. As a result the users do not have to learn the Flowbster/Occopus descriptor language, only the very intuitive graphical user interface.

7 Performance Measurements

In this section we examine the performance of Flowbster with respect to three design aspects. First, we measured how the node scalability parallelism performs in Flowbster in a single cloud environment. Second, we measured the impact on performance of using a hybrid cloud environment. In the third type of experiment we compared the performance of a well-known orchestration based workflow system (WS-PGRADE) and Flowbster to justify our claim that the choreography based concept of Flowbster with its direct data messaging and static cloud deployment indeed performs better than the orchestration and job submission based workflow system.

For all the measurements we have selected the AutoDock Vina application that we introduced and defined as Flowbster workflow in Section 6. The original workflow implementation of the AutoDock Vina application was published for WS-PGRADE/gUSE in [18] and hence this application is a good candidate to compare the performance of the two execution mechanisms used in WS-PGRADE/gUSE and Flowbster. The AutoDock Vina application is a real scientific application that has been used for many scientific projects. Similarly, WS-PGRADE/gUSE is a production level workflow system based on which more than 30 science gateways have been installed and used in Europe for large scientific projects like VERCE [19], DRiHM [20] and MoSGrid [21] therefore, their use in the performance measurements provide real-life performance results.

The AutoDock Vina application workflow consists of three types of nodes: a generator, a set of Vina processing nodes, and a collector (as was described in Section 6). The input of the workflow includes the followings: a receptor molecule, a Vina configuration file, and a set of molecules to dock against the receptor molecule. The task of the generator node is to split the set of molecules to dock within a number of parts.

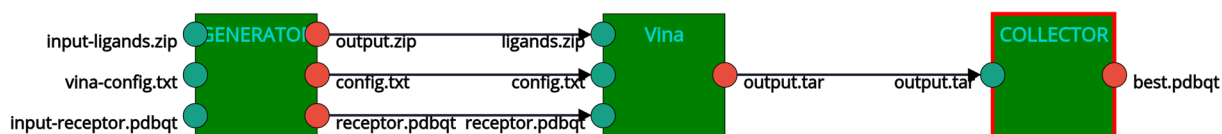


Fig. 25 AutoDock Vina workflow constructed

The task of the Vina nodes is to process these parts iterating through each molecule in the given part, by performing the docking simulation. The result of each docking includes an energy level. At the end the user is interested in the docking with the lowest energy level. The task of the collector node is to get the processing result of each molecule part from the Vina nodes and select those dockings that result in the best 5 energy levels.

For running the performance measurement experiments, we selected a molecule set of 3840 molecules. This set was split into 240 parts, so each part included 16 molecules to dock against the receptor molecule. The input data for the simulation (including the receptor molecule, the Vina configuration file and the 3840 molecules) is available for download [22].

7.1 Running AutoDock Vina Workflow by Flowbster in MTA Cloud

In the first experiment type we have run the AutoDock Vina workflow by Flowbster in the MTA cloud

infrastructure. MTA Cloud is the cloud of the Hungarian Academy of Sciences [23]. It is based on OpenStack and currently supports 40 different scientific projects. In this type of experiment we wanted to compare the execution times of running the Flowbster Vina workflow with 1, 5 and 10 Vina worker nodes in order to investigate the performance of node scalability in Flowbster. Accordingly, we applied three experiment types using 1, 5 and 10 Vina worker nodes, respectively. All the experiment types were executed 5 times (5 experiments) in MTA Cloud using the same instance type m1.small (2 GB of RAM, 1 vCPU).

The results of the experiments are shown in Table 1. The three experiment types are represented by columns labeled with MTA (1), MTA (5) and MTA (10). Each column has three parts. The first one shows the start-up (deployment) time of the Flowbster workflow infrastructure. In case of experiment type MTA (1) we had 3 VMs (Generator, Worker and Collector) and the average start-up time is 4 min and 7 s. For MTA (5) we had 7 VMs (Generator, 5 Workers and Collector) and the deployment time is 5 min and

Table 1 AutoDocking execution times with Flowbster on MTA Cloud

Experiment	Flowbster MTA (1)			Flowbster MTA (5)			Flowbster MTA (10)		
	Infra startup (min)	Period	Duration (min)	Infra startup (min)	Period	Duration (min)	Infra startup (min)	Period	Duration (min)
1	04:16,5	14:06–18:07	241	05:42,0	08:26–09:28	62	07:20,4	09:03–09:44	41,25
2	04:02,6	8:42–13:06	263	05:01,0	09:36–10:38	62	07:10,7	11:12–11:57	45
3	04:17,1	13:46–17:47	241	06:08,0	10:40–11:42	62	07:13,3	12:18–12:58	40
4	03:53,9	06:53–10:57	243	05:09,0	13:24–14:27	63	07:39,0	13:15–13:54	39
5	04:08,1	11:44–15:48	244	05:21,0	09:19–10:21	62	06:51,7	07:49–08:29	40
Average time:	04:07,6		246,4	5:28		62,2	07:15,0		41,05
Average: deviation time	0:00:08		6,64	0:00:21		0,32	0:00:12		1,66

28 s. Meanwhile the number of VMs were increased by $7/3 = 2.333$ the deployment time was increased only by $328/247 = 1.328$. Similarly, in case of MTA (10) where 12 VMs were used (4 times more than in MTA (1)) the average start-up time was only $435/247 = 1.761$ times longer than in MTA (1). This slow increase of the deployment time is due to the capability of Occopus to deploy the independent Wina Worker node VMs in parallel. Notice that the average deviation time for deployment is also very small for every experiment type: 8, 21 and 12 s, respectively.

The second column in Table 1 shows the exact execution periods for every experiment from which the data of the third column, i.e., the execution time of every experiment (including the start-up time, too) can be computed. With 5 worker nodes the speedup is $246/62 = 3.968$ while with 10 worker nodes is $246/41 = 6$.

The speed-up values for all these three experiment types are shown in Fig. 26. The light gray line represents the optimal speed-up while the dark line shows the measured speed-up. The figure shows that speed-up significantly grows as we increase the number of Vina worker nodes but cannot reach the optimal values due to the communication cost among the virtual machines realizing the workflow nodes.

7.2 Flowbster Execution Performance in a Hybrid-Cloud System

In the fourth type of experiments we have executed the Flowbster Vina workflow in a hybrid-cloud environment consisting of MTA Cloud and Amazon Cloud. For both clouds we have allocated 5 Vina worker nodes (5 + 5 worker nodes). The Collector workflow node was deployed in MTA Cloud and the Generator node in Amazon. These experiments also used the instance type m1.small (2 GB of RAM, 1 vCPU). The deployment and execution times of these experiments are shown in Table 2 using the same format as in Table 3. Notice that both the deployment and execution time was slightly shorter than in the MTA (10) experiment type. The deployment time was shorter because Occopus can deploy virtual machines implementing the Vina worker nodes in parallel in the two clouds. The execution time was better because Amazon resources were slightly faster than the MTA Cloud resources. The cost of running one docking experiment partially in Amazon and partially in MTA Cloud costed \$0.115 on Amazon and was free on MTA Cloud (since this cloud is free for the Hungarian scientists).

The deployment and execution time results of the four experiment types described above are shown together in Fig. 27 represented by columns labeled

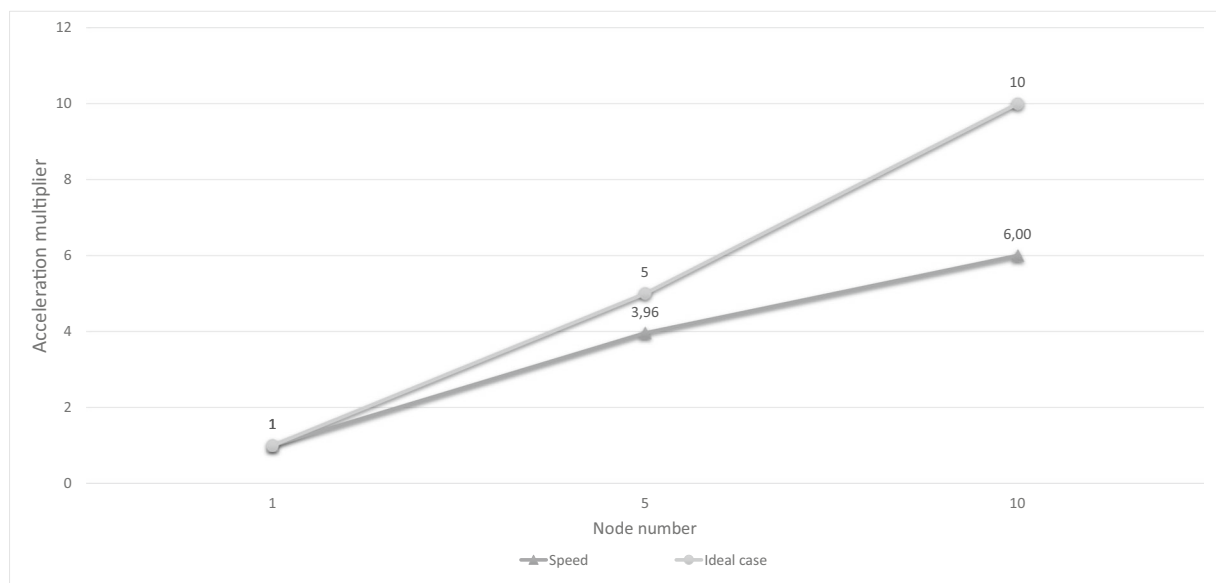


Fig. 26 Node scalability speed-up diagram of Flowbster in MTA Cloud

Table 2 AutoDocking execution times with Flowbster on MTA Cloud and Amazon

Experiment	Flowbster MTA Cloud (5) + Flowbster AWS (5)		
	Infra startup (min)	Period	Duration (min)
1	05:09,0	12:12:48–12:48:48	36
2	04:32,0	12:49:52–13:34:40	35
3	04:27,0	19:15:25–19:52:16	37
4	04:04,0	11:37:54–12:14:39	37
5	05:02,0	12:36:43–13:13:29	37
Average time:	4:34		36,4
Average deviation time:	0:00:21		0,72

with Node number 1, 5, 10 and 5 + 5. Each column has two parts. The shorter one shows the deployment time and the higher one represents the execution time (including the deployment time). The values presented in the figure are the average execution times of the four types of experiments. The deviation values were negligible in all four cases showing the strength of the Flowbster workflow approach: once the workflow infrastructure is created in the cloud it works in a stable and predictable way as the data flows through the workflow structure. Figure 27 also shows that the hybrid-cloud execution produced slightly better speed-up than the single cloud one due to the reasons mentioned above.

All these experiment types are reproducible for the interested readers. A tutorial on the experiment type MTA (5) can be found among the Flowbster tutorials at the Occopus tutorial webpage [24].

This experiment type can easily be modified to run the other three types of experiments, too as explained

at the end of the tutorial. All these experiment types can be executed in Amazon, OpenStack, OpenNebula and CloudSigma depending on where the reader has got access to.

7.3 Running AutoDock Vina Workflow by WS-PGRADE/gUSE

In the fifth experiment type we have run the above mentioned AutoDock Vina workflow by a WS-PGRADE workflow system that was running in a WS-PGRADE/gUSE portal version 3.7.5. The portal was set up for the time of the experiments in MTA Cloud and executed the workflow nodes also in MTA Cloud. The VM (virtual machine) where the WS-PGRADE/gUSE portal was running had 4GB of RAM and 2 vCPUs. The portal was set up in a way that it could use 5 Vina node VMs in parallel with instance type m1.small (2 GB of RAM, 1 vCPU).

We have run the experiment 5 times and the results are in Table 3 showing the starting time, completion time and duration of each experiment. The table also shows the average execution time and the average deviation time. As it can be seen the 2nd–5th experiments had very similar execution time close to 2 h. During the 1st experiment the cloud was overloaded and hence the execution time was about 60% longer than in the other experiments resulted in a significant average deviation time.

In order to make the experiment reproducible for the readers we made the AutoDock Vina workflow used in the experiment (with the inputs described previously, prepared inside) publicly accessible and downloadable [25].

Table 3 AutoDocking execution times with gUSE on MTA Cloud

Experiment	gUSE MTA (5)	
	Period	Duration (min)
1	06:18:40–09:26:34	188
2	04:53:34–06:45:14	112
3	07:52:32–09:44:47	112
4	10:29:39–12:37:57	128
5	12:51:43–14:44:16	113
Average time:		130,6
Average deviation time:		22,96

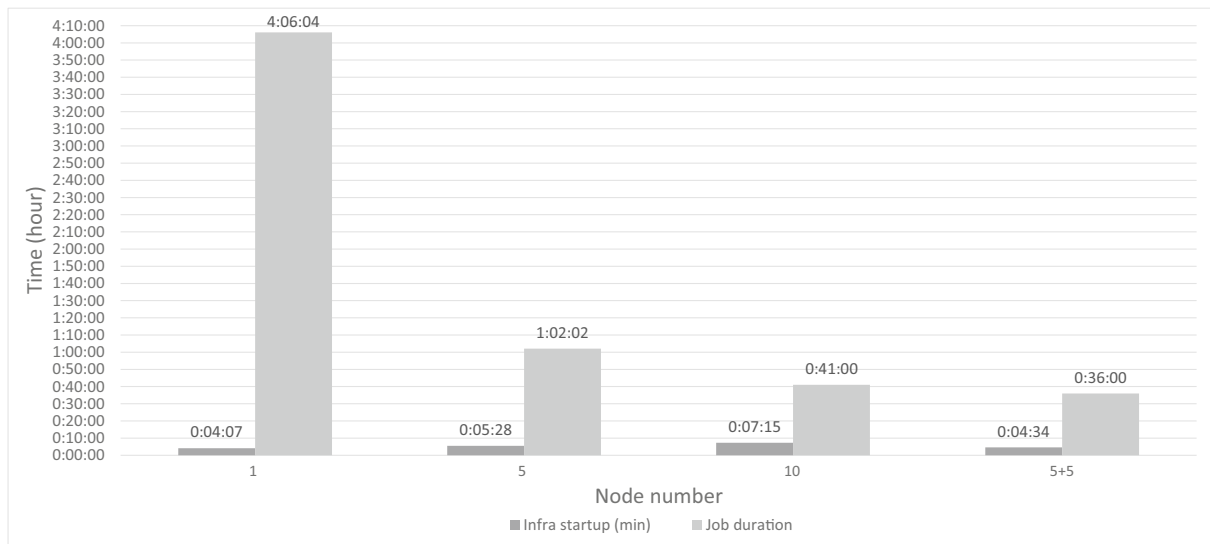


Fig. 27 Flowbster infrastructure startup and job duration time

The workflow can be tried for example through the public DARIAH Gateway (<https://dariah-gateway.lpds.sztaki.hu/>) after eduGAIN-based login. This gateway is connected to the EGI cloud infrastructure (e.g. RECAS-BARI or INFN-CATANIA-STACK sites).

The goal with this type of experiments was to compare the performance of a well-known orchestration based workflow system (WS-PGRADE) and Flowbster to justify our claim that the choreography based execution mechanism of Flowbster is significantly faster than the orchestration based execution mechanism for DAG workflows. The comparison of the execution times in Table 3 and the MTA (5) part of Table 1 justifies our claim. The average Flowbster execution time (62,2 min) is about half of the WS-PGRADE/gUSE average execution time (130,6 min). Moreover, the average deviation time is much less in case of Flowbster than in the case of WS-PGRADE/gUSE.

8 Related Work

Scientific workflow languages have been going through a long history of development. They became really popular when grid systems provided a large set of distributed resources and running complex applications on grid systems required special skill. The

complexity of running grid applications could significantly be reduced by applying workflows that made more or less automatic the access and usage of grid resources hiding many tedious details of grids from the scientists. In the case of grid workflows the main issue was how to schedule the jobs to the different grid resources in the most efficient way [1–3]. These workflow systems were typically based on the service orchestration concept with a central enactor that was running either on the user local computer [4–6] or on a science gateway [12]. The transfer of data among the nodes of the workflow was a difficult issue especially if the workflow was designed to be portable among different grid systems [13, 26]. For example, [26] introduced the so-called data staging site that stored temporary files (that are generated by the workflow but do not need to be saved after the workflow completes) in order to improve the performance of data staging. Although, such a data staging storage could be placed in an object store of a cloud where the workflow is executed the moving of data in and out as the workflow nodes generate and consume these temporary files is still needed. In case of cloud choreography (like Flowbster) such copying of files between the workflow tasks and a data staging storage is not needed at all since workflow tasks directly pass the data from the generator to the consumer.

As cloud appeared the first natural progress in these languages was to enable the access of cloud resources but in principle they used clouds in the same way as grid resources earlier [27–29]. Recently new directions can be noticed to better organize workflows in cloud environments. One such step was to deploy the workflow enactor together with the workflow into the target cloud system [30]. This resulted in significant improvement in the execution performance of workflows in the cloud. Another example to find new, innovative ways of executing workflows in clouds is described in [31] and [32] by Qasha et al. This approach also works based on service orchestration but the workflow tasks are submitted in a novel way to the cloud. If a workflow task becomes executable a cloud orchestrator (Cloudify) dynamically deploys the workflow task node in a docker container in the cloud and initiates the associated executable to run inside the container. When the task execution is finished its container is removed from the cloud (except for container optimization cases when several workflow tasks are deployed in the same container). Although this approach is based on service orchestration and Flowbster is based on service choreography their implementation method has many similarities. In both concepts the workflow is described by the descriptor language of a cloud orchestrator (TOSCA in case of [32] and Occopus in case of Flowbster) which is a new cloud-oriented approach compared to “traditional” workflow specifications. Then the workflow nodes are deployed by a cloud orchestrator: Cloudify in case of [32] and Occopus in case of Flowbster. However, the deployment in [32] is dynamic and for the different workflow tasks it requires a separate action to deploy the required container while in Flowbster all the workflow tasks are deployed within the same deployment phase (and whenever nodes are independent they are deployed in parallel which makes the deployment phase quite efficient). Notice that [32] deploys the workflow nodes as docker containers while Flowbster as virtual machines. Deploying workflow nodes as docker containers is important in case of small workflow tasks or “glue” codes required to match input and output interfaces of subsequent workflow blocks. From the point of view of Flowbster deploying workflow tasks as containers or virtual machines is not a fundamental issue rather a technical difference. It is a work in progress in Flowbster to enable

to deploy the workflow nodes as docker containers, as well.

Another novel approach to take into consideration the new possibilities offered by clouds was the introduction of infrastructure-aware workflows [33]. This extended the workflows with a new type of node that enabled the deployment of virtual infrastructures in the target cloud. Once this deployment node is executed the next computational nodes in the same workflow can run on this newly created virtual infrastructure. This concept significantly improves the portability of workflows among different kind of infrastructures. In fact, this concept can be considered as a generalization of the dynamic workflow node deployment concept of [32].

The work described in this paper significantly defers from the previous scientific workflow concepts. Recognizing that in processing very large data sets the data path is the most crucial aspect of the performance of workflow execution it unifies the data and the control paths of the workflow execution. Instead of a central control element like the enactor the data itself controls the execution of the workflow tasks. A task can be executed when all the required data is available and this is recognized by the tasks themselves and not by the enactor. The workflow tasks are realized as virtual machines which communicate with each other. The most critical aspect in the organization of such workflows is the communication interface that is provided among the workflow tasks. The significance of Flowbster is that it defines and implements this communication interface in a generic way separated from the user level where the workflow business logic is defined by the user. Inside the service assembly layer (Flowbster), the requested interfaces of one service is wired to provide interfaces to other services. The services assembled in this way can again be recursively exposed as a service which can be wired and invoked as it was shown in Section 4.4. The result of Flowbster is a service assembly layer, a deployable artifact, which can be deployed in one or more target cloud systems.

In order to deploy the Flowbster defined service assembly layer a cloud deployment and orchestration tool is needed. There are quite a few orchestration tools which aim to ease the creation, deployment and management of virtual infrastructures consisting of virtual machines that execute services. We overview

some of the most relevant and successful cloud orchestration tools in the following paragraphs.

The Ubuntu Juju [34] orchestration system allows quick deployment, configuration, management and maintenance of cloud services. It can orchestrate various public and private clouds, physical servers, OpenStack and containers with multiple cloud backends including AWS EC2, Microsoft Azure and Openstack. Juju provides an attractive graphical user interface for building complex infrastructures with a drag-and-drop style editing. Beyond its strengths, unfortunately Juju implements vendor and OS lock-in (supports only Ubuntu on VMs). Moreover, currently it does not support advanced features like auto-scaling or node auto-healing.

HashiCorp developed a tool called Terraform [35] for building and changing infrastructures. Terraform can manage existing and popular service providers as well as custom in-house solutions. Terraform is integrated with HashiCorp's other tools, however, on its own it lacks advanced features, and is only capable of deployment of an infrastructure without lifecycle-management, scaling, error-handling. It only has a command-line interface, however, it can integrate with Chef to provide configuration management and is open-source.

CloudFormation [36] is one of the most mature and heavyweight contenders between orchestrators developed by Amazon. Infrastructures are defined as JSON templates which can be submitted through CLI, API or the AWS management console to the AWS EC2 Cloud. Unfortunately, it does not support any other cloud providers or backends. The vendor lock-in makes it hard to use for academic purposes and it is not an open source software.

Openstack's Heat [37] is template-based, provides auto-scaling features through integration with Telemetry, and has Chef/Puppet integration. Heat can be used with a CLI, API, or the Horizon Dashboard. Heat has its own template format, HOT (Heat Orchestration Template), but can process CloudFormation templates. Heat is open source, however, it only supports OpenStack clouds.

Cloudify [38] is one of the latest orchestrators, with its first release being in 2014. It is an open source cloud orchestration framework, allowing the user to model applications and services and automate their entire life cycle, including deployment on any cloud or data center environment, monitoring all aspects of

the deployed application, detecting issues and failure, manually or automatically remediating them and handle ongoing maintenance tasks. However, some advanced features including the Web UI are only available in the commercial (premium) edition.

Slipstream [39] is a quite versatile orchestration tool which provides an application for managing multi-cloud environments, with almost every major cloud providers supported. It also has an easy-to-use graphical web UI. Its source code is partially open source, with connectors for proprietary solutions remaining closed source. Unfortunately, its orchestration method is strictly image-based so it does not make use of modern configuration management tools, making the configuration of the instances more complicated. It also does not support advanced features like node auto-healing or automatic scaling.

OneFlow [40] allows users and administrators to define, execute and manage multi-tiered applications, or services composed of interconnected Virtual Machines with deployment dependencies between them. Each group of Virtual Machines is deployed and managed as a single entity, and is completely integrated with the advanced OpenNebula user and group management. While providing advanced features like scaling and auto-healing, and being relatively easy-to-learn with a simple JSON format for templates, it is locked to general EC2 or OpenNebula clouds. Similarly to SlipStream, it is purely image-based, with no support of modern configuration management tools. Finally, it does not provide multi-cloud or hybrid cloud support.

Looking over the list of orchestrators introduced above and their features, we can conclude that an easy-to-use, freely available, open-source orchestrator with multi-cloud and scaling support and without vendor, cloud, or operating-system, etc. lock-in can be hardly found. Occopus through its several recent releases is very close to provide all these features. Therefore, we selected Occopus as the cloud deployment and orchestration tool and integrated it with Flowbster resulting in the Flowbster/Occopus framework to build data pipeline workflow as explained earlier. This choice has the further advantage that both Flowbster and Occopus are open source and easily modifiable so researchers inventing new features for the framework can easily develop these required extensions.

9 Conclusions and Further Development

The appearance of cloud systems and the need for workflows by which large scientific data sets can be efficiently processed requires to find new directions in organizing scientific workflows. Such a new concept is the Flowbster/Occopus framework by which users can virtually hardwire their workflows as virtual infrastructures into the target clouds. Occopus guarantees that the workflow can be deployed in any major type of IaaS clouds (OpenStack, OpenNebula, Amazon, EC2, CloudSigma). It also enables if needed to deploy the workflow in a multi- or even hybrid-cloud environment as was shown in Section 7.2. Occopus takes care of not only deploying the nodes of the workflow but also to maintain their health by using various health-checking options.

Occopus also enables the implementation of the node scalability parallelism which is an important feature to accelerate data processing inside the workflow. Performance measurements show that the exploitation of node scalability parallelism indeed significantly accelerates the work of the data pipeline in case of parameter sweep applications. Currently Flowbster/Occopus supports the static version of node scalability parallelism but soon the dynamic version will be developed based on the principles described in Section 5.

Flowbster defines and implements the service assembly layer of the workflows by defining and implementing the required Receiver, Executor and Forwarder micro-services and their communication protocols. Based on these micro-services it creates the uniform building blocks of Flowbster workflows that can be customized by the workflow developers to create specialized workflow nodes like generator, worker, collector, join, fork and others from which complex, high-level workflow patterns like parameter sweep, if-then-else and others can be easily created. Due to space restrictions in the paper we focused on the description of how the very popular parameter sweep pattern can be realized in Flowbster. This pattern also well demonstrates how node scalability parallelism can be exploited in Flowbster.

It was also shown how the Occopus descriptors should be created for Flowbster workflows to make them deployable by Occopus. An important feature of the Flowbster/Occopus framework is that it provides an intuitive graphical user interface that

completely hides and automatically generates the required Occopus descriptors. As a result developing Flowbster workflows and deploying them in target clouds is easy and does not require thorough cloud knowledge. The user can concentrate on the business logic of her workflow while the Flowbster/Occopus framework guarantees an efficient cloud based execution.

In order to further improve the performance we will exploit the feature of Occopus to be able to start the workflow tasks not only as VMs but also as docker containers. It is a work in progress to enable the user to specify which workflow tasks should be executed by VMs and which tasks by docker containers. In this case, a further optimization will be that small granularity workflow tasks implemented by docker containers could be placed together in a shared VM.

For Flowbster we wanted to use an open source, general, multi-purpose, cloud-, operating system- and programming language agnostic orchestrator with very simple, easy-to-understand, portable descriptors, with flexible, extendable architecture and with health-monitoring, life-cycle management, scaling and error handling. Detailed analysis showed that among the tools mentioned in Section 8 Occopus is the one that satisfies all these requirements so we selected Occopus as the underlying cloud orchestrator layer of Flowbster. However, Occopus can easily be replaced with any of the above mentioned orchestrators if due to certain reasons a user community prefers to use another one.

The Flowbster/Occopus framework is an open source product licensed under the Apache License, Version 2.0 (the “License”). The complete source code can be downloaded from github [15].

Acknowledgements This work is partially funded by the European CloudiFacturing - Cloudification of Production Engineering for Predictive Digital Manufacturing project under grant No. 768892 (H2020-FoF-2017), and by the International Science & Technology Cooperation Program of China under grant No. 2015DFE12860. On behalf of the Flowbster project we thank for the usage of MTA Cloud (<https://cloud.mta.hu/>) that significantly helped us achieving the results published in this paper.

References

1. Liu, J., Pacitti, E., Valduriez, P., Mattoso, M.: A survey of data-intensive scientific workflow management. *J. Grid Comput.* **13**(4), 457–494 (2015)

2. Deelman, E., Gannon, D., Shields, M., Taylor, I.: Workflows and e-science: an overview of workflow system features and capabilities. *Futur. Gener. Comput. Syst.* **25**(5), 528–540 (2009)
3. Yu, J., Buyya, R.: A taxonomy of workflow management systems for grid computing. *J. Grid Comput.* **3**(3–4), 171–200 (2005)
4. Deelman, E., Vahi, K., Juve, G., Rynge, M., Callaghan, S., Maechling, P.J., Mayani, R., Chen, W., Silva, R.F.d., Livny, M., Wenger, K.: Pegasus: a workflow management system for science automation. *Futur. Gener. Comput. Syst.* (2014)
5. Fahringer, T., Prodan, R., Duan, R., Hofer, J., Nadeem, F., Nerieri, F., Podlipnig, S., Qin, J., Siddiqui, M., Truong, H.-L., Villazon, A., Wiecek, M.: Askalon: a development and grid computing environment for scientific workflows. In: Taylor, I.J., Deelman, E., Gannon, D.B., Shields, M. (eds.) *Workflows for E-Science*, pp. 450–471. Springer, London (2007)
6. Altintas, I., Berkley, C., Jaeger, E., Jones, M., Ludascher, B., Mock, S.: Kepler: an extensible system for design and execution of scientific workflows. In: 16th International Conference on Scientific and Statistical Database Management (SSDBM), pp. 423–424 (2004)
7. Goecks, J., Nekrutenko, A., Taylor, J.: Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome Biol.* **11**(8), 1–13 (2010)
8. Oinn, T.M., Addis, M., Ferris, J., Marvin, D., Senger, M., Greenwood, R.M., Carver, T., Glover, K., Pocock, M.R., Wipat, A., Li, P.: Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics* **20**(17), 3045–3054 (2004)
9. Zaha, J.M., Barros, A., Dumas, M., ter Hofstede, A.: A language for service behavior modeling. In: *CoopIS*, Montpellier, France (2006)
10. Kavantzaz, N., Burdett, D., Ritzinger, G., Lafon, Y.: Web services choreography description language version 1.0, W3C Candidate Recommendation. Tech. Rep. (2005)
11. Terstyanszky, G., Kukla, T., Kiss, T., Kacsuk, P., Balasko, A., Farkas, Z.: Enabling scientific workflow sharing through coarse-grained interoperability. *Futur. Gener. Comput. Syst.* **37**, 46–59 (2014)
12. Kacsuk, P., Farkas, Z., Kozlovsky, M., Herman, G., Balasko, A., Karóczkai, K., Marton, I.: WS-PGRADE/GUSE generic DCI gateway framework for a large variety of user communities. *J. Grid Comput.* **10**(4), 601–630 (2012)
13. Hajnal, Á., Márton, I., Farkas, Z., Kacsuk, P.: Remote storage management in science gateways via data bridging. *Concurr. Comput.: Pract. Exp.* **27**(16), 4398–4411 (2015)
14. Kacsuk, P. (ed.): *Science gateways for distributed computing infrastructures: development framework and exploitation by scientific user communities*. Springer International Publishing. ISBN: 978-3-319-11267-1 (2014)
15. Occopus github repository: <https://github.com/occopus>
16. Flowbster github repository: <https://github.com/occopus/flowbster>
17. Trott, O., Olson, A.J.: Autodock Vina: improving the speed and accuracy of docking with a new scoring function, efficient optimization and multithreading. *J. Comput. Chem.* **31**, 455–461 (2010)
18. Farkas, Z., Kacsuk, P., Kiss, T., Borsody, P., Hajnal, Á., Balaskó, Á., Karóczkai, K.: Autodock gateway for molecular docking simulations in cloud systems. In: Terzo, O., Mossucca, L. (eds.) *Cloud Computing with E-Science Applications*. p. 300, pp. 217–235. CRC Press - Taylor and Francis Group, Boca Raton (2015). ISBN:978-1-4665-9115-8
19. Kiss, T., Kacsuk, P., Lovas, R., et al.: WS-PGRADE/GUSE in European Projects. In: Kacsuk, P. (ed.) *Science Gateways for Distributed Computing Infrastructures: Development Framework and Exploitation by Scientific User Communities*, pp. 235–254. Springer, Berlin (2014)
20. D’Agostino, D., Danovaro, E., Clematis, A., Roverelli, L., Zereik, G., Galizia, A.: From lesson learned to the refactoring of the DRIHM science gateway for hydro-meteorological research. *J. Grid Comput.* **14**(4), 575–588 (2016)
21. Gesing, S., Kruger, J., Grunzke, R., Herres-Pawlis, S., Hoffmann, A.: Using science gateways for bridging the differences between research infrastructures. *J. Grid Comput.* **14**(4), 545–557 (2016)
22. Vina input files: https://sourceforge.net/p/guse/git/ci/master/tree/vina/vina_inputs.tar.gz?format=raw
23. MTA Cloud: <https://cloud.mta.hu/>
24. Occopus tutorial webpage: <http://occopus.lpd.sztaki.hu/tutorials>
25. Vina application files: https://sourceforge.net/p/guse/git/ci/master/tree/vina/AutoDock-Vina_2017-08-17-060932.all.zip?format=raw
26. Vahi, K., Rynge, M., Juve, G., Mayani, R., Deelman, E.: Rethinking data management for big data scientific workflows. In: 2013 IEEE International Conference on Big Data. Silicon Valley. <https://doi.org/10.1109/BigData.2013.6691724> (2013)
27. Farkas, Z., Kacsuk, P., Hajnal, Á.: Enabling workflow-oriented science gateways to access multi-cloud systems. *J. Grid Comput.* **14**(4), 619–640 (2016)
28. Flanagan, K., et al.: Microbase2.0: a generic framework for computationally intensive bioinformatics workflows in the cloud. *J. Integr. Bioinform. (JIB)*. <https://doi.org/10.2390/biecoll-jib-2012-212> (2012)
29. Emeakaroha, V.C., Maurer, M., Stern, P., Labaj, P.P., Brandic, I., Kreil, D.P.: Managing and optimizing bioinformatics workflows for data analysis in clouds. *J. Grid Comput.* **11**(3), 407–428 (2013)
30. Balis, B., Figliola, K., Malawski, M., Pawlik, M., Bubak, M.: A lightweight approach for deployment of scientific workflows in cloud infrastructures. In: *Parallel Processing and Applied Mathematics*, Volume 9573 of the series *Lecture Notes in Computer Science*, pp. 281–290 (2016)
31. Qasha, R., et al.: A framework for scientific workflow reproducibility in the cloud. In: 2016 IEEE 12th International Conference on e-Science (e-Science), pp. 81–90. IEEE. <https://doi.org/10.1109/eScience.2016.7870888> (2016)
32. Qasha, R., et al.: Dynamic deployment of scientific workflows in the cloud using container virtualization. In: 2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom). IEEE, pp. 269–276. <https://doi.org/10.1109/CloudCom.2016.0052> (2016)

33. Kacsuk, P., Kecskemeti, G., Kertesz, A., Nemeth, Z., Visegradi, A., Gergely, M.: Infrastructure aware scientific workflows and their support by a Science Gateway. In: Proceedings of the 7th International Workshop on Science Gateways (IWSG 2015), pp. 22–27. Budapest (2015)
34. Ubuntu Juju: <http://juju.ubuntu.com>
35. Terraform: <https://www.terraform.io/>
36. Cloudformation: <https://aws.amazon.com/cloudformation/>
37. Heat: <https://wiki.openstack.org/wiki/Heat>
38. Cloudify: <http://getcloudify.org/>
39. Slipstream: <http://sixsq.com/products/slipstream/index.html>
40. Oneflow: http://docs.opennebula.org/4.12/advanced_administration/application_flow_and_auto-scaling/appflow_use_cli.html