The Journal of Systems & Software 168 (2020) 110665

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Kulla, a container-centric construction model for building infrastructure-agnostic distributed and parallel applications

Hugo G. Reyes-Anastacio [a], J.L Gonzalez-Compean [a,*], Victor J. Sosa-Sosa [a,b], Jesus Carretero [b], Javier Garcia-Blas [b]

[a] CINVESTAV Unidad Tamaulipas, Km. 5.5 Carr. a Soto la Marina, Victoria, Tamaulipas, 87130, Mexico
[b] Computer Science and Engineering Department, University Carlos III de Madrid Av. Universidad, 30, 28911 Leganés, Madrid, Spain

ARTICLE INFO

Article history:
Received 9 September 2019
Received in revised form 10 May 2020
Accepted 25 May 2020
Available online 28 May 2020

Keywords:
Infrastructure-agnostic applications
Construction model
Virtual containers
Parallel patterns
Pipelines

ABSTRACT

This paper presents the design, development, and implementation of *Kulla*, a virtual container-centric construction model that mixes loosely coupled structures with a parallel programming model for building infrastructure-agnostic distributed and parallel applications. In Kulla, applications, dependencies and environment settings, are mapped with construction units called *Kulla-Blocks*. A parallel programming model enables developers to couple those interoperable structures for creating constructive structures named *Kulla-Bricks*. In these structures, continuous dataflow and parallel patterns can be created without modifying the code of applications. Methods such as *Divide&Containerize* (data parallelism), *Pipe&Blocks* (streaming), and *Manager/Block* (task parallelism) were developed to create *Kulla-Bricks*. Recursive combinations of Kulla instances can be grouped in deployment structures called *Kulla-Boxes*, which are encapsulated into VCs to create infrastructure-agnostic parallel and/or distributed applications. Deployment strategies were created for Kulla-Boxes to improve the IT resource profitability. To show the feasibility and flexibility of this model, solutions combining real-world applications were implemented by using Kulla instances to compose parallel and/or distributed system deployed on different IT infrastructures. An experimental evaluation based on use cases solving satellite and medical image processing problems revealed the efficiency of Kulla model in comparison with some traditional state-of-the-art solutions.

1. Introduction

Vendor lock-in is a critical issue for organizations and end-users that deploy applications on the cloud (Opara-Martins et al., 2014). To avoid this vendor lock-in, the preparation of large volume of data before sending them to data centers and/or cloud-based services is becoming a practice commonly carried out by organizations (Celesti et al., 2019; Zhang et al., 2015). In these preparation processes, organizations add properties to their contents for managing, and producing massive content collections. For example, the contents are preprocessed to reduce data volume (Gonzalez-Compean et al., 2017), to save storage space and to reduce costs, which adds a storage cost-efficiency property to contents. In other cases, the contents are encrypted to ensure privacy, signed to ensure authenticity, and commonly processed to ensure access control as well as fault tolerance for adding confidentiality and reliability respectively (Gonzalez-Compean et al., 2018). Hospitals (Abushab et al., 2018; Marcelín-Jiménez and Rajsbaum, 2003) and space agencies (Gonzalez-Compean et al., 2017) are some examples of organizations that commonly process large volumes of data (images and data/metadata) for adding properties to the health contents (Abushab et al., 2018; Marcelín-Jiménez and Rajsbaum, 2003) and satellite data (Gonzalez-Compean et al., 2017). Those contents are commonly accessed through services by partners, external organizations, and end-users.

In practice, developers are moving the building of their applications from monolithic to loosely coupled solutions to achieve more efficient and easily maintainable applications. For instance, currently there are solutions available that provide processing structures such as pipelines, workflows, and processing patterns to integrate sets of applications into a single solution (Babuji et al., 2019; Montella et al., 2018a; Ferguson, 2011; Taylor et al., 2007; Montella et al., 2015; Skluzacek et al., 2016). In this type of solutions, the outputs of some applications represent the inputs of other, which creates software patterns producing *continuous delivery* of data/metadata from a data source (e.g. a folder or

* Corresponding author.
E-mail addresses: hugo.reyes@cinvestav.mx (H.G. Reyes-Anastacio), joseluis.gonzalez@cinvestav.mx (J.L. Gonzalez-Compean), vsosa@inf.uc3m.es (V.J. Sosa-Sosa), jcarrete@inf.uc3m.es (J. Carretero), fjblas@inf.uc3m.es (J. Garcia-Blas).

https://doi.org/10.1016/j.jss.2020.110665
0164-1212/

hard disk partition) to the processing stages and to a data sink (e.g. cloud storage location or remote/shared folder).

In complex scenarios, management strategies are required for applications to exchange data (del Rio Astorga et al., 2018; Badia et al., 2015). As a result, frameworks and engines (Babuji et al., 2019; Montella et al., 2015; Taylor et al., 2007; Wilde et al., 2011) are becoming popular solutions for developers to build content preparation solutions based on this type of processing model. Nevertheless, troubleshooting IT processes and portability of applications are challenges that arise in real-world scenarios when organizations deploying this type of solution on IT infrastructures such as cloud, clusters and stand-alone servers. Troubleshooting IT processes are quite common in organizational environments where the applications are migrated or installed over different types of IT infrastructures (any of private, public or hybrid). In this type of scenario, the portability of applications is not commonly granted by developers, which produces in some cases a vendor or platform lock-in problem (Tsidulko, 2015; Miranda et al., 2012). In this situation, issues related to failed installations, missed dependencies or misplaced environment settings must be solved by IT staff using debugging procedures, which could take valuable time causing either downtime or disturbing business continuity (Hayden and Carbone, 2015; Souppaya et al., 2017). To overcome those problems, virtual containers (VCs) have become a popular solution for organizations to deploy applications on different infrastructures in an immutable manner (Karmel et al.).[1] Moreover, some studies have shown that VCs are lightweight in comparison with traditional virtual machines (Sharma et al., 2016; de Alfonso et al., 2017).

In real-world scenarios, the solutions for data preparation not only should be portable, but also be built in a dynamic and flexible way. Moreover, improving the efficiency of the solutions results crucial, as the service experience of end-users is quite important for decision-making processes. In this context, we consider that there is a need for tools, not only enabling organizations and end-users to create solutions with added value to their contents, but also including features required in real-world scenarios such as portability, flexibility, and efficiency. *Portability* is required for avoiding/reducing the troubleshooting IT procedures. *Flexibility* is demanded for creating solutions including as many applications as value properties to be added to the contents, which should be created by solving the interoperability among the applications without making major and complex adjustments in the solutions. *Efficiency* is required to avoid/reduce the side effects of performing preprocessing and preparation tasks on the service experience of the end-users by profiting as many resources as available in IT infrastructure. In addition, the profitability of IT resources is crucial for organizations to achieve efficiency in the processing and preparation of their contents. By *profitability* of resources, we refer to the way in which a solution, integrating different applications, maximizes the usage of existing resources when processing each content in a given infrastructure where that solution is deployed on (e.g. using the available servers, virtual containers or machines as well as number of cores and memory per each computing resource).

In this paper, we present the design, development, and implementation of *Kulla*, a construction model for building infrastructure-agnostic parallel and distributed applications. In this construction model, loosely coupled structures are mixed with parallel programming model to provide the applications with portability and efficiency properties in flexible and dynamic manners.

In Kulla model, each application is mapped with its dependencies, environments variables and operating system settings by using logical construction units called *Kulla-Blocks*. This structure represents an image of a given application and is used as a deployment guide for the correct functionality of each application. These units also include I/O interface maps that convert a *Kulla-Block* into an interoperable software piece, which enables developers to chain *Kulla-Blocks* in the form of directed graphs to create constructive structure called *Kulla-Bricks*. In addition, a parallel programming model enables this type of structure to produce continuous data delivery and parallel patterns, which can be built without altering/modifying the code of applications. These structures represent an image of a parallel solution and are used as a deployment guide to execute them. The programming model is implemented by using software instances such as in-memory data storage mechanisms, control messages, load balancing and synchronization of workload as well as I/O network/file call management systems. When these instances are added to the *Kulla-Brick* images, continuous delivery and parallel patterns are created in execution time.

Methods such as *Pipe&Blocks*, *Divide&Containerize*, and *Manager/Block* were defined for developers to create *Kulla-Bricks* producing implicit streaming, data parallelism and task parallelism respectively. In addition, groups of recursive combinations of *Kulla-Blocks* and/or *Kulla-Bricks* images are managed by deployment structures called *Kulla-Boxes*. Each Kulla-Box is encapsulated into a virtual container, which is deployed on an IT infrastructure.

In *Kulla* model, all software instances are self-similar.[2] to the smallest construction unit, the *Kulla-Block* The rest of components (bricks and boxes) are similar to the smallest constructive component in the way of this structure processes data: by following an ETL model. This means the data Extraction is performed through the input interfaces, the data Transformation is performed by the filters and the Load of results is performed through the output interfaces. All the components (any of bricks, boxes and bricks of boxes) process data by following this very model. For instance, a Kulla-Box (a virtual container) used to deploy a Kulla-Brick (pattern) including a set of Kulla-Blocks (application) is similar to the Kulla-Brick, which is similar to the Kulla-Blocks because the VC of the Kulla-Box extracts data from a source, the inside Kulla-Brick patterns transforms these processes data by following an ETL model and loads the results in a data sink of the Kulla-Box's virtual container. In this example, the pattern of the inside Kulla-Brick is thus similar to the Kulla-Box as both process data by following the very same ETL model. In this context, the Kulla-Blocks are similar to the Kulla-Bricks and the Kulla-Box as the application processes data by following again an ETL model. This property enables developers to couple *Kulla-Boxes* to create Bricks of *Kulla-Boxes*. These groups of *Kulla-Boxes* are managed as a single infrastructure-agnostic processing structure.

A set of deployment strategies such as *Scale-In*, *Scale-Out*, and a combination of both are proposed in Kulla model for deploying distributed and/or parallel applications by using *Kulla-Boxes*. These strategies determine the way in which a *Kulla* solution is deployed on a given infrastructure and allow organizations to improve the profitability of computing resources. This deployment scheme avoids organizations to perform troubleshooting processes when having container platforms installed.

The main contributions of this paper are as follows:

[1] Market studies predict that by the year 2022, more than 75% of companies will use container technology which is a significant increase from fewer than 30% today (Gartner, Inc. and/or its affiliates, 2017). 82% of the companies expect to have more than 100 containers deployed within the next two years, according to "Containers: Real Adoption and Use Cases in 2017" (Brozek, 2019).

[2] Something that is exactly or approximately similar to a part of itself.

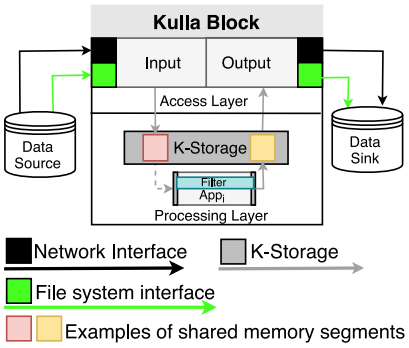


Fig. 1. A conceptual representation of Kulla-Block.

- A container-centric construction model for building distributed and/or parallel applications by using interoperable self-similar construction structures.
- A parallel programming model to build continuous data delivery and parallel patterns. Different patterns or a combination of them solving efficiency issues can be created by using methods such as *Pipe&Blocks*, *Divide&Containerize* and *Manager/Blocks*, as well as combinations of these patterns.
- A strategy for deploying *Kulla* solutions on different types of computing resources in flexible and dynamic manners. This adds the agnosticism property to solutions and improves the profitability of resources.
- An experimental evaluation with distributed and parallel *Kulla* solutions to solve use cases of satellite and medical imagery processing.

The rest of the paper is organized as follows. Design principles of *Kulla* are described in Section 2. Section 3 presents the patterns developed by composition of *Kulla-Bricks*. The evaluation methodology and experimental results from experiments and case studies are described in Section 4, where performance results are described. In Section 5 the related work is described. Finally, conclusions and future research lines are described in Section 6.

2. Design principles of the *Kulla* construction model

The design principles of the *Kulla* construction model rely on the following construction structures: (i) Kulla-Blocks and Kulla-bricks, which are abstract constructive structures to create single and parallel application images respectively. (ii) Kulla-Boxes, constructive and deployment structures for building infrastructure-agnostic solutions. (iii) Bricks of Kulla-Boxes, which are management structures for building infrastructure-agnostic distributed solutions.

2.1. Kulla-Block: A building block structure

As previously mentioned, a *Kulla-Block* instance is the smallest logical construction structure and the basic building block in the *Kulla* model.

A *Kulla-Block* is an image that maps an application with its dependencies and environment settings, which ensures the correct operation of that application in production.

In addition, the *Kulla-Block* structure also has been designed to convert applications into independent, interoperable, and reusable pieces of software as depicted in Fig. 1.

As it can be seen, a *Kulla-Block* (*KB*) is conformed by access and processing layers. The access layer includes *Input* and *Output* interfaces to manage the data coming from a *Data Source* (DSr) and the outgoing data produced by the applications to a *Data Sink* (DSk): $KB[in] = Input$ and $KB[out] = Output$. The processing layer includes maps to either the code or binary of an application and to a set of reserved *Kulla* software instances called *Adapters* as well as to I/O interfaces libraries and services. An *Adapter* is a Kulla software instance that represents an intermediary between the application and the access layer of a *Kulla-Block*. This means that an *Adapter* intercepts the data arriving at the *Kulla-Block*, writes them in memory, ingests them to the applications and retrieves the results from the applications. As may be seen, a *Kulla-Block* instance follows the traditional ETL model (Extract, Transform and Load). In the extract phase, a $KB[in]$ receives data from a data source (*DSr*). The *DSr* could be any of a hard disk partition, cloud location, or another *KB* through its output interface $KB[Out]$.

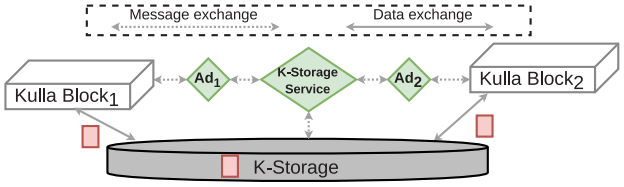


Fig. 2. Sharing results through the K-Storage.

In the transformation phase, the adapters store the data (d_x) retrieved from *DSr* in a data exchange area created by an in-memory service called *K-Storage*. The d_x are retrieved by the application (App_j), launched within the *Kulla-Block* by using $KB[In]$. The d_x are processed to produce results r_x, which are also stored in the *K-Storage*. In the load phase, r_x are sent to either a *DSk* or another *Kulla-Block* instance through the $KB[Out]$ by using the following notation:

$(DSr \bigvee (i > 1 \Rightarrow KB_{i-1}[Out])) \xrightarrow{d_x} KB_i[In] \xrightarrow{d_x} Ad \xrightarrow{d_x} App \xrightarrow{r_x} Ad \xrightarrow{r_x} KB_i[Out] \xrightarrow{r_x} (DSk \bigvee KB_{i+1}[In])$. Where $i \geq 1$

The *K-Storage* service provides exchange data areas for Kulla instances (blocks or bricks) by using a *shared resource pattern*. Fig. 2 depicts the shared resource pattern used to build an in-memory service as *K-Storage*. This service offers the traditional Put and Get operations, which are performed by using memory positions that are accessed through pointers. Fig. 2 depicts an example of a pipeline of two Kulla-boxes where two adapters (Ad_1 and Ad_2) manage the delivery and retrieval of data between the two Kulla-blocks as well as the K-storage service and the in-memory space created by this service (K-storage). As it can be seen, the Kulla-blocks can create an in-memory space in K-Storage. The adapters invokes the Put (Ad_1) and Get (Ad_2) for establishing the Input/output data management between Kulla-Blocks ($Kulla - Block_1$ and $Kulla - Block_2$) in an implicit manner. The Put and Get operations return pointers to the K-Storage that are used by Kulla-Blocks for delivering/retrieval data to/from in-memory storage. In this pattern, the management of shared memory is transparent for the Kulla-Blocks. Moreover, this pattern enables the coupling and decoupling of Kulla-Blocks in a flexible manner as the coupling is not a direct one but through a third party (K-Storage); as a result, a Kulla-Block can be decoupled from a given Kulla-Block and to be coupled to another one by changing the configuration file.

We can define a *Kulla-Block* as a *ETL* map built by using the following notation: $KB = \{In, Ad, App, \text{K-Storage}, Out\}$. Where *In* is the input interface configuration, the adapter *Ad* $\in$ {libraries, I/O_{ad}, dependencies, controls}, where the *controls* are instructions to control the execution of the Kulla-Blocks, *App* is the application to be executed, *K-Storage* represents the in-memory data exchange service and *Out* is the output interface configuration.

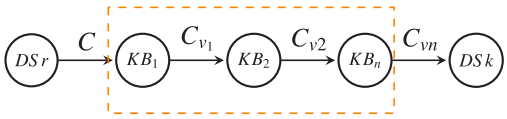


Fig. 3. A DAG describing a *Kulla-Brick* implementing pipelines by using Pipe&Block method.

2.2. Kulla-Bricks: *Structures to build parallel patterns*

A *Kulla-Brick* is a constructive unit pattern that was designed for coupling *n Kulla-Blocks* through the I/O interfaces in the form of a directed acyclic graph (DAG) of *ETL* maps, which also include maps to an input channel (Source, DSr) and to output channel (Sink, DSk). The DAG map is specified as a topology in a file. The model is fully hierarchical with potentially *n* levels of composition, as blocks can include inside other blocks and bricks without restriction.

A *Kulla-Brick* can be defined by using the following notation: $KBr = (DSr, (KB_{i=1}^{n}, DAG), DSk)$. Where $n \geq 1$.

To enable developers to build Kulla-Bricks, methods such as *Pipe&Blocks*, *Divide&Containerize* and *Manager/Block* were considered in Kulla model. Methods for coupling different combinations of these patterns into a single solution are also described in this section.

These methods were considered in the parallel programming model of Kulla because they are useful in many problems decomposition and production processes such as streaming/dataflow, task, and data parallelism respectively.

2.2.1. *Pipe&Block: A* Kulla-Brick *to create software pipelines*

We designed a method named *Pipe&Block* to create the traditional *Pipe&Filter* pattern, which is used to chain applications (Filters: $\{F_1, \ldots, F_n\}$) in adjacent manner through I/O paths (Pipes) where $n \geq 1$ is required (Grawinkel et al., 2015; Buschmann et al., 2007).

Fig. 3 shows a directed graph depicting a pattern created by using *Pipe&Block* method. This pattern, shown inside of dashed line rectangle, creates a continuous data delivery for each *Kulla-Block* (KB_i) in the pipeline, which creates a sequence of processing *Kulla-Blocks* adjacently connected through the I/O interfaces (edges). The notation of the DAG map for this Kulla-Brick is:

$$KBr_{pb} = (DSr, (KB_{i=1}^{n-1}, Pattern), DSk).$$

where $n \geq 1 \wedge DSr \rightarrow KB_1[in] \wedge KB_n[Out] \rightarrow DSk \wedge Pattern = \{KB_i[Out] \rightarrow KB_{i+1}[In]\}_{i=1}^{n-1}$.

In execution time, this arrangement will create a dataflow from a data source to a data sink. In this dataflow each *KB* transforms the received contents from input data (see *C* input in KB_1 in Fig. 3) into a new version (see C_{v1} input in KB_1 in Fig. 3) and forwards it to either a data sink or another Kulla-Block (KB_2 in Fig. 3) through a *pipe* represented by an edge.

To implement the DAG of ETLs of a *Kulla-Brick*, we developed control *adapters* such as *engine*, *orchestrator*, and *launcher* that are used in configuration, deployment and execution times.

- *Orchestrator* is an adapter that is executed in configuration time. It is in charge of converting a DAG notation into a configuration file that will use in deployment time. This file includes information required to identify the *Kulla-Blocks* included in a *Kulla-Brick* and the labels indicating the place of each *Kulla-Block* in a software pipeline (Labels such as *initiator*, *intermediary* and *ending*). The *orchestrator* also creates a file configuration per *Kulla-Block* and delivers them to an adapter called *launcher*.
- *Launcher* is an adapter that, in development time, creates a coupling configuration file, which describes the way in which the chaining of *Kulla-Blocks* should be performed when the Kulla instances have been deployed on a given infrastructure. This chaining will build a software pipeline by executing a procedure performed by the launcher (see Algorithm 1).
- *Engine* is an adapter that supervises the execution of all the components of each *Kulla-Block* included in a *Kulla-Brick* (e.g. applications, adapters, K-Storage management, etc.). In execution time, the *engine* performs the following tasks: (i) Injects workload to the *beginning Kulla-Block* in the pipeline for starting the continuous delivery and to process contents (injects a new content when the pipeline finished processing the previous one); (ii) Establishes controls over the execution of the *Kulla-Blocks* and the exchange of data through K-Storage; (iii) Manages the notifications of the ending of each processing task performed by each *Kulla-Block* to establish controls over the workload injection.

Algorithm 1 Pipe&Filter Pattern Creation.

```
Require: NBinBrick ∨ Brick − BlockIDMaps[] ∨ Paths[]
  DSr ← Paths[0]
  DSk ← Paths[1]
  Initiator ← BlockIDMaps[0]
  Kulla-Brick[0] ← Launch(Initiator)
  Previous ← Initiator
  NextBlock ← BlockIDMaps[1]
  Kulla-Brick[1] ← Launch(NextBlock)
  Kulla-Brick[0] ← Map(Initiator.InPar, In = DSr, Out = NextBlock.In)
  i = 2
  while NextBlock ≠ Ending do
    Previous ← NextBlock
    NextBlock ← BlockIDMaps[i]
    Kulla-Brick[i] = Launch(NextBlock)
    Kulla-Brick[i-1] ← Map(Previous.In, In = Previos.out, Out = NextBlock[i].In
    i++
  end while
  if NextBlock[i] == Ending then
    Kulla-Brick[i] ← Map(Previous.In, In, Out = DSk
    EXIT
  else
    ERROR(EndingPipeline)
  end if
```

As it can be seen in Algorithm 1, the *Input* interface of *Kulla-Block*, labeled as *beginning*, is linked to the Data Source (DSr), whereas the *Output* interface is linked to the *Input* of the first *intermediary*. This means the *Input* and *Output* interfaces of *intermediary Kulla-Blocks* are linked to the interfaces of previous and next *Kulla-Blocks* in the form of a chain, which enables each *Kulla-Block* to retrieve data and deliver processed results respectively. The Output interface of the *Kulla-Block* labeled as *ending* is linked to a Data Sink (DSk) path where the final results will be stored. When the *launcher* and *orchestrator* return the control to the *engine*, the global configuration file of the *Kulla-Brick* is ready to be added to a *Kulla-Box* where it will be executed to process data.

2.2.2. *Divide&Containerize: A segmentation Kulla-Brick for producing data parallelism*

Divide&Containerize (*D&C*) is a method implementing the traditional divide and conquer algorithm (Posner et al., 2010), which was developed in *Kulla* to build *Kulla-Bricks* useful for processing large size contents.

In this method, only one application is mapped to a *Kulla-Block* (*worker*). The goal of this type of *Kulla-Brick* is to clone this *Kulla-Block* and to execute all the replicas of *Kulla-Blocks* in a concurrent manner.

The D&C method includes *adapters* such as *Divide*, *Containerize* and *Conquer*. Fig. 4 depicts, inside of dashed line rectangle, the DAG of a *D&C* Kulla-Brick.

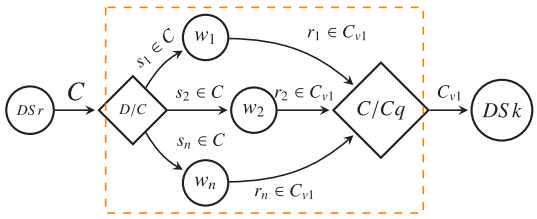


Fig. 4. Divide&Containerize (D&C) pattern represented as directed graph.

As it can be seen, the *Divide* adapter (see D/C in Fig. 4) read contents (C) from data source (DSr), splits each content into n data segments (see $\{s_1, \ldots, s_n\} \in C$ Fig. 4). *Containerize* sends these segments to a set of *Kulla-Blocks* called $workers$ (see $\{w_1, \ldots, w_n\}$ in Fig. 4). The workers end up processing segments created by Divide in concurrent manner.

In a reduction phase, the workers send the processed data segments (see $\{r_1, \ldots, r_n\} \in C_{v1}$ in Fig. 4) to the *Containerize*, which makes them available for *Conquer* adapter (Cq) to consolidate these results into a new version of the processed content (C_v1). This new version is delivered to a *Data Sink* (DSk).

Notice that the management of the control functions and synchronization of data exchange between *Divide* and *Workers* as well as Workers and *Conquer* are implicit procedure performed by *Containerize* adapter (See D/C and C/Cq).

As it can be seen, this *Kulla-Brick* is self-similar to the worker (*Kulla-Block*) as it uses the very ETL model used by *Kulla-Blocks*: The content C is extracted from the data source (DSr), transformed by the *Kulla-Brick* into a new version ($Cv1$), that is loaded to the Data Sink (DSk).

The map of this *D&C Kulla-Brick* can be represented by the following the DAG notation.

$KBr_{dc} = (DSr, (D\&C, (w_{i=1}^{n}, Pattern), Cq), DSk)$. Where $n > 1 \wedge$ $Pattern = \{D\&C \xrightarrow{s_i} w_i \xrightarrow{r_i} Cq\}_{i=1}^{n}$

Algorithm 2 describes the segmentation process performed by *Divide* adapter and the way the workers use *Containerize* adapter to get access to the *K-Storage* pointers for applications encapsulated into the *Kulla-Blocks* to process the segments created by Divide as well as for putting the data results in the *K-Storage*.

Algorithm 2 Divide Segmentation Process.

Require: Content name *name*, Data Content $|C|$, number of segments s, output list of K-Storage pointers $outs = \{out_1, \ldots, out_s\}$
1: $CurrentSegment = 1$, $SegmentSize = 0$, $WritedSegments = 0$
2: **while** $CurrentSegment \leq s$ **do**
3: *CreateThread*()
4: $SegmentSize = getSegmentSize(sizeof(|C|))$
5: **if** $(CurrentSegment > 1)$ **then**
6: $initPosition = SegmentSize * (CurrentSegment - 1)$
7: **else**
8: $initPosition = 0$
9: **end if**
10: $SegmentedContent = ReadSegment(|C|, InitPosition, SegmentSize)$
11: **if** $(WriteOnKStorage(SegmentedContent, out_{CurrentSegment}))$ **then**
12: $WritedSegments+ = 1$
13: **end if**
14: *KillThread*()
15: **end while**
16: **if** $(WritedSegments == s)$ **then**
17: **return** *GenerateContentId*(*name*)
18: **else**
19: **return** *ERROR*(*WritingSegments*)
20: **end if**

As previously established, the *Conquer* adapter receives the *K-Storage* pointers of K-Storage from Containerize, gets the partial results and consolidates these results into one single output (depending on the action suggested in each case). The integration process used by Conquer is described in Algorithm 3.

Algorithm 3 Conquer Integration process.

Require: Content Id F_{id}, File Size $|F_{size}|$, number of segments s, input list of K-Storage pointers $input = \{input_1, \ldots, inputs\}$, output path out
1: $i = 0$, $j = 0$, $recoveredSegments = [s]$
2: **while** $i \leq s$ **do**
3: *CreateThread*()
4: $recoveredSegments[i] = RecoverSegment(i, F_{id})$
5: *KillThread*()
6: $i+ = 1$
7: **end while**
8: $recoveredData = ReserveMemory(F_{size})$
9: $recoveredData = IntegrateContent(recoveredSegments[])$
10: **if** $(sendContent(recoveredData, F_{id}, out))$ **then**
11: **return** *msj*(*OK*)
12: **else**
13: **return** *ERROR*(*SendContent*)
14: **end if**

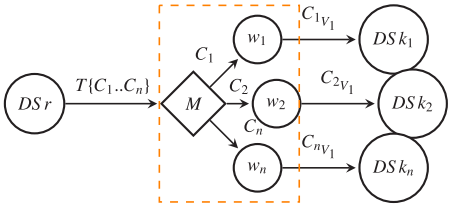


Fig. 5. *Manager/Block* (M/B) example.

To create a $D/\&C$ Kulla-Brick image, in deployment time we reuse the *adapters* previously defined in the Filter&Blocks (*Launcher*, *Engine*, and *Orchestrator*).

In a preparation phase, *Orchestrator* creates two configuration files: The first one is for the Kulla-Block (*Worker*), which enables the method to prepare the application to be executed in parallel; The second one is for the reserved Kulla instances *(Divide & Containerize and Containerize & Conquer)* (See *D&C* and *C&Cq* in Fig. 4).

In a configuration phase, *Launcher* adapter creates as many Kulla-Blocks (n) as described in the DAG. This adapter creates a deployment configuration file per each Kulla-Block worker in the Kulla-Brick.

In a chaining phase, *Orchestrator* creates (n) configuration files by using the *worker basis* images and configures the input and output interfaces of DS_r, *D&C*, *Workers*, *C&Cq* and DS_k to implement the DAG of this pattern. When the chaining of Kulla-Blocks has been finished, the Kulla-Brick is ready to be encapsulated into a Kulla-Box to process data.

In execution time, *Containerize* creates a data exchange area by using the *K-Storage* service for all the Kulla-Blocks to exchange data. At this point, *Engine* establishes the controls over the synchronization of the process execution of the Kulla-Brick components.

2.2.3. Manager/Block: A Kulla-Brick producing task parallelism

We also developed a method named *Manager/Blocks* (M/B), which creates manager/worker patterns inside a Kulla-Brick for producing task parallelism. This method is quite useful when processing large sets of contents.

Fig. 5 shows, inside of dashed line rectangle, a DAG of a Kulla-Brick built by using M/B method.

The basic idea of this method is having a manager (See M in Fig. 5) to create tasks for processing sets of contents ($T = \{C_1, C_2, \ldots, C_n\}$) extracted from a Data Source (DSr). Each content is sent to one $worker$ ($\{w_1, w_2, \ldots, w_n\}$ to process these tasks in parallel.

The manager also creates a data exchange area by using *K-Storage* for the workers to retrieve data from. This adapter

also retrieves data by using input interfaces and sends results to *K-Storage*. The manager includes a load balancing algorithm (Morales-Ferreira et al., 2018) that assigns tasks to the workers through an I/O interface in a fair manner as well as a module to keep control over the execution of the collection of tasks and to assign new tasks to the workers.

As previously established, the *Workers* follow the ETL model (Extract, Transform and Load). This means that the worker receives a *task* as input parameter, *extracts* the content C_x associated to received tasks by using the exchange area (by using K-Storage), and executes a given application to *transform* that content into a new version (see, in Fig. 5, for instance $C_{1_{V_1}}$, $C_{2_{V_1}}$ *and* $C_{n_{V_1}}$ produced by w_1, w_2 and w_n respectively). The worker also *Loads* the results to either shared memory by using *K-Storage* or puts them either in an output interface or a Data Sink. This means the worker can either start another processing procedure or finish a current process by sending results to a Data Sink (DSk_n). A worker therefore can invoke any of a new pattern, other Kulla-Block or even its Manager for retrieving data.

As it can be seen, the *Worker* adapters do not return control to another adapter (e.g. *Manager* in this pattern, or *Conquer* in *D&C* method) as there is no previous data segmentation and a consolidation of results is not required. This enables developers to combine this pattern with other patterns. In Kulla model, patterns based on complex graph systems can be built by using recursively patterns and/or by combining different types/numbers of *Kulla-Bricks* in a single solution. For instance, each worker of the M/B pattern can launch another M/B, and so on.

This Kulla-Brick can be defined by the following notation:

$KBr_{mb} = (DSr, (M, w_{i=1}^{n}, DSk_{i=1}^{n}, Pattern))$. Where $n > 1 \wedge Pattern = \{M \xrightarrow{C_i} w_i \xrightarrow{C_{i_{V_1}}} DSk_i\}_{i=1}^{n}$

Algorithm 4 describes the construction of the dataflow produced by this pattern.

Algorithm 4 *Manager/Block* algorithm.

Require: Number of workers w, instance name *name*, output list *output* $= \{o_1, o_2, \ldots, o_w\}$, data source path *in*, complete file paths of files list with tasks ids *collection*[], set of clones id *clonesID*[]

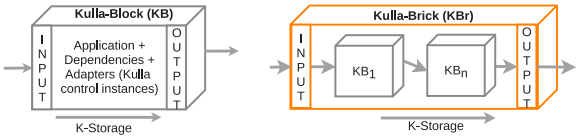
```
1: i = 1, j = 1
2: collection = readDataSource(in)
3: while i ≤ w do
4:     clonesID = launchClone(name, i)
5:     dataPerClone[i] = doLoadBalancing(collection, i)
6:     i++
7: end while
8: while j ≤ w do
9:     readDataInvokesNextKullaElementInThread(
           clonesID[j], dataPerClone[j], output[j])
10: end while
11: waitThreads()
```

To implement this pattern, we reused adapters such as *engine and orchestrator* previously described in *D&C* to produce development files.

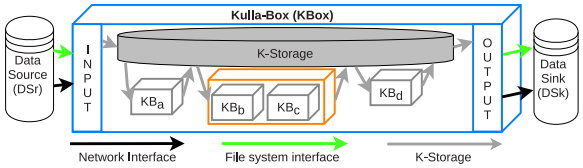
3. Kulla-Boxes:Deployment structures for building Infrastructure-agnostic applications

Kulla-Box is the only deployment structure used in this model. *Kulla-Box* receives the deployment files created by *Kulla-Blocks*, *Kulla-Bricks* images or any combination of them and implements them in the form of *Kulla* software instances. These instances are encapsulated into a virtual container. A Kulla-Box therefore is ready to be deployed by the end-users on a given infrastructure having installed a container platform (e.g. Docker or Linux containers). A *Kulla-Box* could be represented as:

$$KBox_i = DSr \xrightarrow{d_x} KBox_i[In] \xrightarrow{d_x} KullaImages \xrightarrow{r_x} KBox_i[Out] \xrightarrow{r_x} DSk$$



(a) Example of Kulla-Block (b) Example of Kulla-Brick



(c) Example of Kulla-Box: The result of the encapsulation of images into a virtual container

Fig. 6. Example of an Infrastructure-agnostic application created by using Kulla-Blocks (a), Kulla-Bricks (b) and the Kulla-Box abstraction (c).

where:

$$DSr = DataSource \bigvee (i > 0 \Rightarrow KBox_{i-1}[out]) \wedge$$

$$DSk = DataSink \bigvee KBox_{i+1}[In] \wedge$$

$$Pattern \exists \bigvee KBs \exists \rightarrow KullaImages = \{KBsImages\} \bigvee \{KBrImages\}$$

Where:

$$KBsImages = \{KB_{i=1}^{j} \in KBs\} \wedge j \geq 1 \wedge$$

$$KBrImages = \{KBr_{i=1}^{k} \in KBrs\} \wedge k \geq 1 \wedge$$

$$KBs = \{KB_{ud_1}, \ldots, KB_{ud_n}\} \wedge n \geq 1 \wedge$$

$$KBrs = \{KBr_{pb}, KBr_{dc}, KBr_{mb}, KBr_{ud}\}$$

Fig. 6 shows an example of an infrastructure-agnostic application created by using Kulla-Blocks and Kulla-Bricks images implemented in a Kulla-Box. First, the applications, dependencies and Kulla adapters are grouped in the form of a Kulla-Block image (see Fig. 6(a)). Then, if any, sets of Kulla-Block images are coupled by using DAG adapters to create a Kulla-Brick image (see Fig. 6(b)). The images are converted into Kulla software instances into the virtual container of the Kulla-Box. The result of the encapsulation performed in the example depicted in Fig. 6 is an Infrastructure-agnostic software pipeline application (see Fig. 6(c)). This application is ready for a developer or end-user to deploy it on a given IT infrastructure to process contents of a data source and to deliver the results in a data Sink.

All the Kulla software instances encapsulated into the virtual container of a Kulla-Box (any of adapters, applications of Kulla-Blocks or Kulla-Bricks) share the three I/O interfaces included in the Kulla-Blocks (Network, File system and shared memory). This means that the Kulla instances in a Kulla-Box can build a data exchange space on *shared memory* by using the *K-Storage* service and to retrieve/deliver data from/to source/sink by using either file systems or the network. We reused adapters such as engine and launcher to convert the Kulla-Blocks and/or Kulla-Bricks into software instances and implement them into the virtual container of a Kulla-Box.

The orchestrator first creates a configuration file of a virtual container image (e.g. Dockerfile and Docker compose file) by using the configuration files of all Kulla images (Kulla-Blocks and/or Kulla-Bricks). This file configures a virtual container by using information such as paths of adapters, dependencies, the input/output interfaces for each Kulla-Block and the application to be executed by each Kulla-Block as well as the adapters of each Kulla-Brick included in a Kulla-Box. In this file are exposed the

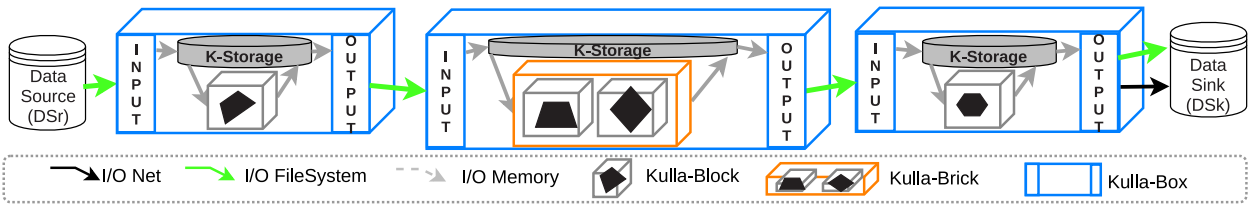


Fig. 7. An example of a Brick of Kulla-Boxes.

ports and/or domains (if required) of Input and Output interfaces, the paths of the volumes that will be used as both data sources and data sinks for each Kulla-Box and the resources assigned to each Kulla-Box (number of cores, RAM). The engine invokes the launcher, which executes the building of a VC image by using the configuration file. At this point, the VC image can be transported, stored or executed by end-users.

This Kulla-Box image can be cloned to launch as many VC instances of a Kulla-Box as required by a given solution. Please notice that a VC image of a Kulla-Box is just a configuration basis, whereas a VC instance is a Kulla-Box being executed in an infrastructure and running all its Kulla instances.

3.1. Bricks of Kulla-Boxes: building infrastructure-agnostic distributed applications

The *Kulla-Box* structures can be converted into reusable and interoperable software pieces because these structures follow the ETL model used by Kulla-Blocks and Kulla-Bricks to Extract input data, Transform contents into new versions and Load output data to a data sink or to another Kulla-Box. This feature provides solutions with interoperability, which enables organizations to chain Kulla-Boxes to other Kulla-Boxes to create *Bricks of Kulla-Boxes*.

Fig. 7 shows an example of a Brick of Kulla-Boxes created in the form of a pipeline of four types of applications (represented by geometric figures encapsulated into Kulla-Blocks) by using the Pipe&Blocks method previously described in this paper. In this example, the first Kulla-Box only encapsulates one Kulla-Block into a virtual container, which could be used as a single application or as part of a system. This is the case where the first Kulla-Box (see Fig. 7) is coupled to the second Kulla-Box, which includes a Kulla-Brick implementing a pipeline of two Kulla-Blocks executing applications. Finally, we can see how the second Kulla-Box is coupled to the last Kulla-Block in this software pipeline, which is similar to the first Kulla-Box.

A brick of Kulla-Boxes can be built by using the D/C, M/B and *Pipe&Block* methods or a combination of them in a recursive manner as a complete solution or as part of another solution. For example, the developer could change all the stages described in Fig. 7 for a set of Kulla-Boxes with the M/B, D/C and *Pipe&Blocks* methods, or could chain multiple methods inside a Kulla-Box and chain it as a stage of this brick of Kulla-Boxes. This type of structure is also built by using DAG map of ETLs of Kulla-Boxes (models previously described in this paper). End-users/developers can create infrastructure-agnostic distributed applications in a flexible and dynamic manner by deploying *Bricks of Kulla-Boxes* on a distributed infrastructure (clusters, clouds, etc.).

To build this type of solution, a DAG notation describing the *Brick of Kulla-Boxes* is converted into a Global deployment configuration file. In this paper, *Kulla* performs this deployment by executing a set scripts for the Docker platform or executing just one file called docker-compose file. This file includes information about all the *Kulla-Box* images considered in a distributed application, the type of pattern considered for that *Brick of Kulla-Boxes* as well as the number of workers in a pattern and the role of each *Kulla-Box* in each pattern. When all the *Kulla-Boxes* have been executed by using the very previously described procedure for a single *Kulla-Box*, the engine invokes the *orchestrator* to create the chaining of the *Kulla-Boxes* as well as the coupling of components within the VC instance of each *Kulla-Box* in a Brick.

3.1.1. Deployment and resource profitability strategies

In *Kulla* model, the end-users and/or developers can determine the way in which the VC of the *Kulla-Boxes* will be deploy on a given infrastructure (e.g. Cluster or the Cloud) by using a deployment and profitability strategy.

Deployment strategies such as scale-in, scale-out and both (mixing scale-in and scale-out) are available in *Kulla*. *Scale-In* is a strategy that enables the developers to create infrastructure-agnostic parallel applications in a *Kulla-Box*, where the *Kulla* instances will use all the cores in either a single server or a virtual machine. This is feasible because of developers can assign a set of resources (e.g. cores, RAM and data volumes) to the VC of a *Kulla-Box*. In this deployment strategy, an in-memory data exchange area is created by *K-Storage* service and used as a communication channel by all software instances in a given *Kulla-Box*. *Scale-Out* strategy enables the developers to build infrastructure-agnostic distributed applications by deploying *Bricks of Kulla-Boxes* on clusters of either servers or virtual machines in the cloud. In this strategy, *K-Storage* service is used by each *Kulla-Box* and the distributed data exchange area is built by using either the network interfaces (Sockets for clusters and Curl for Cloud) or file system (i.e. by using distributed file systems). *Scale-Out/In* is a greedy deployment strategy that enables developers to create workflows of distributed and parallel applications by mixing the above policies. In this strategy, *Bricks of Kulla-Boxes* are deployed on different types of infrastructures by using first, for instance, *scale-out* to use all the servers and then to use *scale-in* to use as many resources as available in each server (i.e. cores and RAM).

In the current Kulla implementation, the terms Scale-in/out refer only to the deployment methods available to create distributed and parallel solutions. The scale-in/out in execution time for elasticity purpose is not considered in the current implementation of Kulla.

3.2. Implementation details of Kulla model

In the *Kulla model*, the VC images and instances were created and launched by using the Docker platform. The Kulla control adapters such as Divide, Containerize, Conquer, Manager, Launcher, Orchestrator, Worker, etc. were written in C programming language. Versions of some of these components (e.g. Manager and Workers) are available in Java, Python and C++. Scale and deployment schemes were developed by using C scripting and structured data (I.e. Json, Dockerfiles and Docker compose files).

3.2.1. Kulla-Silo: Repository of Kulla-Boxes and Kulla software instances

We develop a service called *Kulla-Silo* to concentrate all the *Kulla-Boxes* images in a single repository to simplify the management of Kulla solutions to developers and end-users.

In this repository, the *Kulla* images (Blocks, Bricks and Boxes) are classified as either *reserved* or *user-defined* instances. The adapters such as Managers, Orchestrators, Launchers, Divide, Containerize, Conquer and Workers as well as the implementation of I/O libraries and services such as *K-Storage* and Input/Output interfaces management services are examples of reserved instances. The *Kulla-Boxes* created by end-users/developers are considered as user-defined, which can be used in either public or private manners.

The *Kulla-Silo* service includes functions to *Put*, *Get*, *Update*, *List* and *Delete* virtual container images of *Kulla-Boxes*, as well as functions to *ADD*, *REMOVE*, *CHANGE* reserved *Kulla* software instances to/from a given *Kulla-Box* image (which only can be used by the *Kulla* Adapters). This means that the end-users can choose from existent and available *Kulla-Blocks* and/or *Kulla-Bricks* in the *Kulla-Silo* by using above functions to create user-defined *Kulla-Boxes* in flexible and easy manners.

The *Kulla-Silo* service is implemented as a pair of virtual containers: the first one including a PostgreSQL database for the indexing of *Kulla* images and another one includes an instance of a cloud storage service called SkyCDS (Gonzalez et al., 2015) for storing the images indexed in the *Kulla-Silo*.

3.2.2. Kulla I/O libraries

In the case of end-users being also developers, *Kulla-Silo* includes templates based on functions to get I/O libraries and control structure functions for programming code. The following libraries are available in templates:

- The *management* library includes functions to create, launch, list and delete adapters at the processing layer, which were developed in C programming. It also includes the control software instances as a library (e.g. manager or worker). Some of these functions also available for Java, Python and C++ templates.
- The shared memory management library function calls to PUT/GET data to/from K-Storage service and to keep monitoring the in-memory areas. This library was developed in C with the IPC library (Garrido, 2002).
- The I/O library includes functions to access Input and Output interfaces management system. This library includes sockets and curl I/O functions.

4. Experimental evaluation and results

The assessment and evaluation of the *Kulla* model was conducted through an experimental evaluation in the form of case studies based on real-world application built by using *Kulla* model.

4.1. Prototyping

We developed *Kulla-Boxes* images including *Kulla-Bricks* built by using Divide&Containerize, Pipe&Blocks and *Manager/Blocks* methods by using the Docker platform. The images of these *Kulla-Boxes* were added and indexed in the *Kulla-Silo* and were deployed on the infrastructure described in Table 1 where a distributed cluster of virtual containers was created in a private cloud.

Different solutions were built by using these *Kulla-Boxes* to conduct the experimental evaluation.

We also developed a *Kulla-Box* called *Client* that includes a *Kulla-Block* executing a workload producer bot for sending requests to the *Kulla-Box* instances. In execution time, the Kulla-Boxes assume that the workload generated by this bot is valid, as far as valid credentials of real end-users are provided. This *Client* also includes an adapter for capturing a set of metrics defined to perform this evaluation.

Table 1
IT Infrastructure.

Name	PCs	Cores	RAM	Space	Scenario
PC	1	4	6 GB	240 GB	Scale-In
Server16	1	16	64 GB	2 TB	Scale-In
Server12	1	12	64 GB	2 TB	Scale-In
Cluster	4	12	64 GB	500 GB	Scale-Out/In

4.2. Metrics

The following metrics were extracted from the studied cases:

- *Service Time (ST)* metric represents the elapsed time in which a content is processed by each application encapsulated into a *Kulla-Box*.
- *Response Time (RT)* metric represents the spent time by a *Kulla-Box* solution to successfully dispatch requests sent by the client bot. This is the sum of the *ST* produced by each *Kulla* instance considered in a solution plus the time spent to retrieve/deliver data from/to a data Source/Sink.
- *Percentage of performance gain* metric represents the performance increase in percentage when comparing response times produced by Kulla solutions with traditional solutions such as serial IDA, parallel IDA implementations, Parsl, Makeflow, and Jenkins (described in Section 4.3.1 and Section 4.5.1).

Table 1 shows the features of the infrastructure for each deployment scenario in the evaluation. The configurations, the experiments and the results captured by the defined metrics are described in each case study.

In order to test the behavior of our solution, we run experiments in two scenarios. In the first one, the solutions were deployed on a single server by using *Scale-In* deployment strategy, whereas in the second one, *Scale-In/Out* was used to deploy solutions on a cluster of virtual containers deployed on a set of servers.

4.3. Scale-In deployment scenario: Experiments analysis

In this evaluation scenario, *Kulla-Boxes* were used to create infrastructure-agnostic parallel solutions by using real-world applications. Three solutions based on Kulla-Boxes were implemented by using Divide&Containerize ($D\&C$), Pipe&Blocks ($(P\&B)$) and Manager/Blocks (M/B) methods. These solutions were created and deployed on both a PC and single servers (see details in Table 1) by using the *Scale-In* deployment strategy.

Each experiment was performed 31 times for each studied solution and the median of the metrics was captured for each corresponding evaluation.

4.3.1. Divide&Containerize Kulla-Box study case

In this case, we evaluated a *Kulla-Box* including a *Kulla-Brick* (built by using D&C method) that adds the reliability property to the contents before sending them to the cloud or sharing them with other users/partners. Fig. 8 shows the DAG map of this solution.

The application executed by the *workers* included the implementation of an information dispersal algorithm (IDA) (Rabin, 1989; Spillner et al., 2011; Marcelin-Jimenez et al., 2006; Quezada Naquid et al., 2010; Gonzalez and Marcelín-Jiménez, 2011). This algorithm adds reliability features to the contents for storage system to withstand service failures (data missing, data bit errors, unavailability of servers, etc.). This fault-tolerant technique splits each content $|C|$ of length L into n pieces called

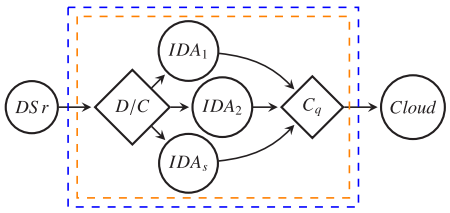


Fig. 8. DAG of the first Kulla-Box evaluated.

dispersal files (*dfs*) each of length $L_{dfs} = L_C/m$. Where m represents the number of *dfs* sufficient for reconstructing $|C|$; as a result, this algorithm can withstand the unavailability of $(n - m)$ *dfs*. The capacity used by this algorithm is $Cap = n*(L_C/m)$, which means the overhead is $Ov = Cap - L_C$.

In practice, the implementation of this algorithm represents a suitable cost-effective solution for the preservation of sensitive contents such as satellite and medical images because of the trade-off between failure tolerance and storage consumption achieved by this technique. However, this algorithm produces expensive computing costs because of the processing of the segments (*dfs*) to add redundancy, which reduces its utilization in real-world scenarios. This implementation becomes a good candidate to use parallelism patterns in the processing of its stages.

The following configurations of the IDA (*Kulla-Box* and other solutions) were tested:

- Serial IDA (*IDA-S*): This configuration executes a serial development of IDA (implemented in C) (Quezada Naquid et al., 2010).
- Parallel IDA (*TBB-IDA*): This configuration represents a parallel version of IDA algorithm (Gonzalez et al., 2015) developed by using Intel TBB framework (Reinders, 2007), where the matrix multiplication routine performed by IDA algorithm was identified to be processed in parallel. The default configuration of this solution considers using all the available cores in a computer (having previously installed and configured the TBB platform, CURL libraries and environment variables).
- *Kulla-IDA*: This configuration represents a Kulla-Box including a Kulla-Brick that executes IDA serial (IDA-S) in a pattern created by *D&C* method without modifying the routines of the original IDA code. Different configurations of this solution were defined by varying the number of workers launched by the Divide module of this solution (from 1 to 5).

4.3.2. Analysis of Divide&Containerize Kulla-Box performance

In this section, we present a performance analysis from the experiments carried out with the configurations and solutions tested in this case study.

Fig. 9(b) shows the response times produced by the solutions evaluated for different file sizes.

As expected, the parallel implementation of the IDA algorithm by using Intel TBB (*TBB-IDA*) produced better response times than the serial implementation of the original algorithm (*IDA-S*). As it can be seen, the performance of TBB-IDA solution is reduced in proportion to the file size: the larger the file size, the lower the performance improvement.

Fig. 9(b) also shows the performance of configurations of *Kulla-Box* when launching different number of workers (from 1 to 5 workers) with the IDA-S process. As it can be seen, Kulla-D/C(2) configuration (two workers in parallel) is not competitive in comparison with TBB-IDA configuration for small files, which is an expected behavior considering that Kulla-DC configurations is not quite efficient with few workers and small tasks. This premise is evident when Kulla-DC(2) becomes competitive for large files in comparison with TBB-IDA configuration. This is also evident when the number of workers of the Kulla-Brick was increased (2, 3, 4 and 5), which producing better performance than TBB-IDA. We identified two causes of this improvement effect in the response times produced by Kulla-DC configurations: the first one is the size of the tasks managed by each worker,[3] which results in the more workers, the less task size is processed by each worker; the second one is the in-memory exchange of information that only produces two I/O operations sent to the file system (one read from the data source and one write in the data sink) and the rest of I/O operations are performed in-memory, which also reduces the service times of *Kulla* solution.

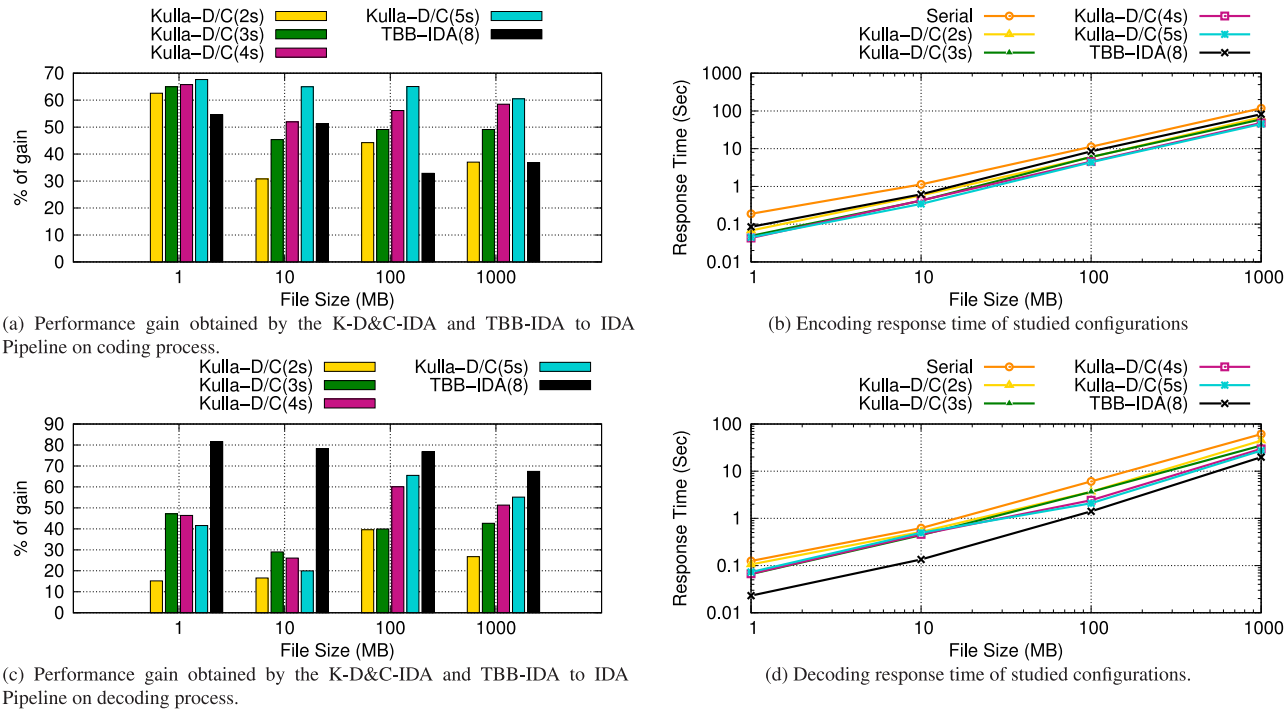
The impact of the evaluated solutions on the performance of IDA algorithm, as well as the performance increase obtained by the solutions evaluated for different file sizes are presented in Fig. 9(a). The improvement of TBB-IDA decreases from 54% (for 1MB files) to 33% in comparison with the serial version (IDA-S), whereas Kulla-DC(5) decreases from 68% to 60% in the same comparison. Moreover, Kulla-DC(5) can even produce a better performance than $TBB - IDA$ depending on the file size to be processed (between 14% and 27%).

In order to understand the effects, not only of the data parallelism, but also of the in-memory management on the performance of the evaluated solutions, we added in-memory management to the TBB-IDA solution by using *Kulla*. Fig. 9(d) shows the response time obtained by the *Kulla-Box* configurations (Kulla-DC(2-5)) and TBB-IDA in the decoding process for contents of different sizes. Fig. 9(c) shows the percentage of gain for all configurations in relation to the serial version (IDA-S). As it can be seen, in-Memory TBB-IDA solution improved its performance, in mean, 40% for small files and 12% for large files in comparison with Kulla-DC. The behavior shown in Fig. 9(c) is produced by the in-memory management of the I/O calls added to TBB-IDA solution, which in this version now only performs two I/O calls to the file system to process data faster than in the original version.

The evaluation revealed that the developers can create solutions producing a better performance than that produced by *Kulla-DC* when they have enough experience to perform tasks such as identifying the routines suitable to be executed in parallel, developing shared memory functions for those routines and preparing the environment (declaring environment variables, adding libraries, etc.) for the applications can successfully be deployed on a given infrastructure. However, when this is not the case, *Kulla* represents a good deal for developers requiring dynamic, rapid and efficient solutions, as the management of parallel data processing and in-memory storage is transparently performed by *Kulla*. Moreover, the construction of the solutions is almost immediate as the parallel pattern is built in advance by *Kulla* and the developer/user only requires to incorporate an application to a worker and choosing the number of workers to be launched in the pattern for obtaining a parallel solution that not only will be portable but also will deliver a competitive performance.

A third option is encapsulating a parallel application into a *Kulla-Box* to add, in a rapid manner, an in-memory processing/storing capacity to create an efficient infrastructure-agnostic application. We performed this in TBB-IDA configuration shown in Fig. 9(d). In this case, the in-memory processing is transparent for developers/end-users as this process is managed inside of the *Kulla-Box*.

[3] *Task Size = FileSize/Number of workers.*



(a) Performance gain obtained by the K-D&C-IDA and TBB-IDA to IDA Pipeline on coding process.

(b) Encoding response time of studied configurations

(c) Performance gain obtained by the K-D&C-IDA and TBB-IDA to IDA Pipeline on decoding process.

(d) Decoding response time of studied configurations.

Fig. 9. Response time and gain obtained by the configurations when varying the input size, number of filters, cores and slaves.

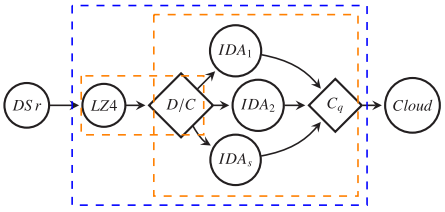


Fig. 10. Second solution evaluated: a Kulla-Brick created by using Pipe&Blocks method.

4.4. Pipe&Block Kulla-Box: a study case for processing satellite imagery

In this case study, we evaluated a *Kulla-Box* including a *Kulla-Brick* of two stages, which was developed by using the Pipe&Block method. Fig. 10 shows the DAG representation of the solution encapsulated in this *Kulla-Box*.

As it can be seen in Fig. 10, the first stage of the pipeline includes an implementation of the *LZ4 Lossless compression algorithm (lZ4)*[4] was encapsulated into a *Kulla-Block*. In this stage, the *Kulla-Block* reads a given file as input and produces a compressed/decompressed version of original file as output. This new version is sent forward to the next stage in the pipeline. The second stage of this pattern is the very Divide&Containerize *Kulla-Brick* evaluated in the previous section.

This *Kulla-Box* allowed us to show an interesting feature of the *Kulla* construction model, which enables developers to create complex solutions by chaining different types of *Kulla* instances. This feature is quite useful in real-world scenarios where the combination of different quality features should be required to be added to the contents. This *Kulla-Box* is represented by the *Kulla-Lz4+IDA* configuration. We reused the parallel configuration (TBB-IDA) and a software pipeline (Serial LZ4-IDA) created by using the serial configuration previously evaluated (IDA-S).

We conducted a case study based on the processing of satellite imagery repository by using these solutions. This repository includes the catalogs of sensors such as Landsat, AQUA and Terra, which were captured by a ground station placed at Chetumal, Mexico. The number of images in these catalogs grows in a constant manner, are large (between 252MB to 1.6GB for images) and must be preserved as information assets for long periods of time as these images are used in the creation of earth observation products.

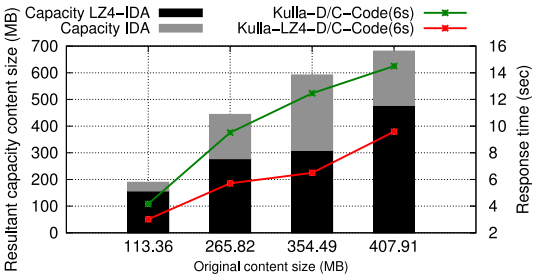
The *Kulla-Box* (*Kulla-Lz4+IDA*) adds cost-efficiency utilization and storage reliability properties to the satellite images in a combined manner. For instance, Kulla-IDA configuration produces 66% (667MB) of redundancy overhead when processing a satellite image of 1GB size to withstand 2 failure of servers/virtual machines, whereas *Kulla-Lz4+IDA* only produces in average 6% (39.9MB) of extra capacity when performing the same operation; as a result, the cost of adding reliability to the products is almost for free in the case of *Kulla-Lz4+IDA*. The goal of this combination of properties is not only to improve storage utilization but also to reduce the service time of reliability techniques. This is a quite interesting feature for earth observation missions.

4.4.1. Pipe&Block Kulla-Box case study: a performance analysis

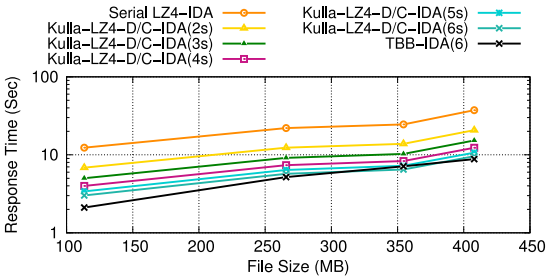
Fig. 11(a) shows, in left vertical axis, the capacity produced when applying the IDA fault-tolerant technique to the satellite images with (See *Capacity LZ4-IDA* bars) and without (See *Capacity IDA* bars) the compression process when coding satellite images of different size (horizontal axis). The costs of the redundancy can be easily observed by comparing the capacity produced by each studied configuration. The evaluation revealed that the improvement of the storage utilization produced in the first stage of the *Kulla-Lz4+IDA* configuration also produced an improvement of the service time on the second stage because of the encoding/decoding workers received less data, which reducing the response time of this solution.

The reliability costs can be significantly reduced when combining encoding with compression depending obviously on the compression degree achieved by the first stage of this pattern. For

[4] Implemented by using the libraries described in Collet (2017).



(a) Resultant capacity space and response time produced by the evaluated solutions



(b) Response time produced by Serial, TBB-IDA and different Kulla-Box configurations.

Fig. 11. Pipe&Filter study case results when using *Server*12 in Table 1(6 physical cores, 6 virtual).

instance, the 40% of reduction was obtained for the first images. This reduced the costs of reliability to zero in terms of storage utilization (saving more than 67% of storage capacity). In turn, the encoding and decoding response times for last image where the compression only could reduce the size of the file in 15% affected the next stage producing an increasing of redundancy overhead of 18%.

Fig. 11(a) also shows, in right vertical axis, the response time produced by Kulla configurations when processing satellite images by using six workers in the Kulla-Brick D/C (see lines of Kulla-*Lz4-D/C-Code(6s)* and Kulla-*D/C-Code(6s)*). The experimental evaluation revealed that the combination of compression with encoding/decoding not only reduces the capacity to be processed but also the response time of coding/encoding processes, which was reduced in 20%. This evidence encouraged us to perform a comparison between the performance of *Kulla-Lz4-D/C-IDA* and *TBB-IDA* (in-memory version offering the best performance in previous experiments). The idea was to establish how much a combination of features enables Kulla-Bricks to be close to reach the performance of a routine paralleled and in-memory solution (TBB-IDA-Mem).

Fig. 11(b) shows, in vertical axis, the response times produced by *Kulla-Lz4-D/C-IDA* and *TBB-IDA* when encoding satellite images by using the studied configurations. As it can be seen, the more the workers, the higher improvement of the response times of these configurations. When *Kulla-Lz4-D/C-IDA* processing large satellite images, it is possible for this configuration to reach the performance of the paralleled *TBB-IDA-Mem* using in-memory storage.

4.5. A study case based on processing medical images: Combining all patterns in a Kulla-Box

At this point, the results showed how the self-similar and modular properties of *Kulla* enables developers to combine patterns to improve the value added to the contents for reducing storage utilization and even improving the performance until to be competitive with routine-based parallel solutions.

We added a new pattern to the previously evaluated *Kulla* solution by converting the LZ4 *Kulla-Block* into a *Kulla-Brick* implementing a Manager/Block pattern (M/B). We encapsulated this solution into a *Kulla-Box*, which represents the third and last solution to be evaluated in the scale-in deployment scenario.

Fig. 12 shows the DAG used to create the third *Kulla-Box* evaluated in this scenario.

The master adapter of the *Kulla-Brick* reads files from the data source and sends one file to each worker, which compress the input file and sent the compressed file to one worker. This worker receives a compressed content and launches a D&C pattern, this pattern splits the received content (Cc) into n segments $\{S_1, S_2, \ldots, S_n\} \in Cc$. In the divide phase, the segments are sent

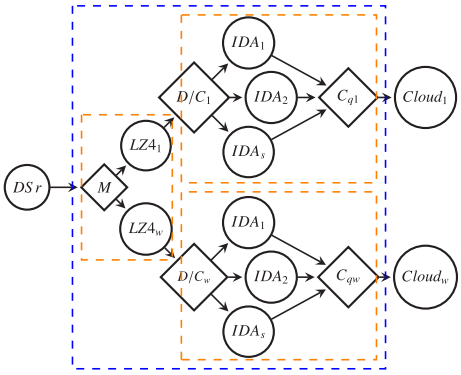


Fig. 12. Third Kulla solution evaluated represented as a directed graph.

to n workers. Each worker adds redundancy to its segment $S \in Cc$ and produces $\{Df_1, Df_2 \ldots, Df_5\} \in S_i \in Cc$. In order to reduce the number of dispersals (Dfs), Conquer consolidates all Dfs produced by the workers to preserve the efficacy of the original algorithm (in reduction phase). We conducted a case study based on computed tomography (CT) imaging repository where both solutions processed 55 images with quality (512×512), of 512 MB each. These images were captured from a crocodile by a tomograph. The medical imagery, with a volume of 27.5 GB, was processed by the evaluated solutions.

The performance of the *Kulla-Box* described in previous section was compared with the performance produced by a set of solutions from the state-of-the-art.

The implemented solutions in this case study were tested in two environments: the first one is a *Scale-in deployment on a Standalone server* and the last one is a scale-in/out deployment on a *Distributed cluster of virtual containers*.

4.5.1. Comparison with state-of-the-art solutions

To compare the performance of Kulla with the performance produced by the following traditional and popular state-of-the-art solutions:

- *TBB-IDA(t)* is the implementation of TBB-IDA described in Section 4.3.1. A virtual container for both coding/encoding tasks as well as storage nodes (as many virtual containers as n, m). These containers are deployed on one *Server 12* for scale-in scenario. For the *scale-in/out scenario* the containers are deployed on the *Cluster* (see Table 1). The t represents the number of threads used by IDA application to process each content stored in the data source of the one PC on the *Cluster*. The data are processed by adding redundancy to the contents, which produces redundant portions that are distributed to the storage nodes. For the scale-in scenario, n portions are dispersed to n local virtual containers.

For the distributed scale-in/out scenario, the *n* portions are distributed to *n* partitions of virtual containers placed at three servers. The data dispersion is performed by using a CURL (Stenberg et al., 2012) application.

- *Parsl(t)* implements the IDA algorithm (described in Section 4.3.1) and the LZ4 implementation (described in Section 4.4) by using a workflow engine called Parsl (Babuji et al., 2019). This engine creates a pipeline including two stages, such as LZ4 and IDA, which were encapsulated into a virtual container by using the local file system (for scale-in scenario) or distributing the results through the network to three VC storage nodes by using a CURL (Stenberg et al., 2012) application (for scale-in/out scenario). The deployment method (either scale-in or scale-in/out) determines the way in which the virtual containers (parsl and storage) are deployed on the infrastructure. The *t* represents the number of threads used by Parsl to run each stage of this solution within each virtual container, which produces task parallelism.
- *Makeflow(t)* provides a pipeline similar to the pipeline built by Parsl, but using another workflow engine called Makeflow (Albrecht et al., 2012). This engine creates a list of tasks that will be executed, including their data dependencies, and creates *t* threads to execute these tasks. In the Scale-in scenario, data are stored in the local file system, whereas in the Scale-in/out, data are dispersed to other servers by using a CURL (Stenberg et al., 2012) application.
- *Jenkins(j)* is a solution created by using popular software pipeline builder called Jenkins (Armenise, 2015). For the scale-in scenario, the processing stages (LZ4 and IDA) are executed in *Server12* by using *j* threads, and the data are stored locally. In the scale-in/out scenario, the pipeline is deployed in four servers (one manager and three workers). The manager runs the LZ4 application in *j* threads, which send the compressed contents to the three workers. Each worker receives their corresponded data, computes the IDA application by using *j* threads, and storage the results in their local file system in the case of scale-in, whereas the servers are used to store data in the Scale-in/out deployment scenario. The *j* represents the number of parallel stages used for run applications of the pipeline, producing task parallelism.
- *K-M/B(w)-D/C(s)* represents the Kulla-Box described in Fig. 12. The scale in/out scenario is created by building a *Brick-Of-Kulla-Boxes*, which was deployed on four servers. This deployment results in a distributed agnostic application.

4.5.2. Prototype details and specifications of studied solutions

In the case of Parsl solution, a Python script was used to execute the applications as stages of a pipeline. Each stage is executed in parallel. In the case of Makeflow solution, a file with extension *jx* was used to define the description of the pipeline: the inputs and outputs of each stage as well as the data dependencies that exist between them (if they exist). This *jx* file is read by Makeflow, which makes a schedule with all the tasks discovered in the input file. At this point, the execution of tasks in parallel starts and the pipeline is online. In the case of Jenkins, a *Jenkinsfile* was used with the description of the studied pipeline. This file describes: (i) the stages that will be executed (defining which will be executed in parallel and the number of clones to be launched for this stage); (ii) the list of files to be processed (must be included in the workspace of the project manually, otherwise Jenkins will detect that the files do not exist or by using a Content Delivery Service or CDS); (iii) the way in which the files will be distributed (does not include a load balancer for the parallel stages; (iv) the container image that will be used to run the pipeline. This framework manages the continuous delivery process required by the pipeline and partially from the parallel execution of the stages, the definition of the number of clones for each parallel stage and the distribution of the load are under the responsibility of the developer who must create an application for this purpose or do it manually. In the configurations Parsl (with LZ4+IDA-S) and Makeflow (with LZ4+IDA-S), the engines of these solutions are in charge of managing both the continuous delivery process required by the pipeline and the task parallelism in such a process. This is similar to the processing performed by *Kulla* configurations.

The experiments for *Kulla* configurations were performed by varying the number of workers launched by the first *Kulla-Brick* (*M*/*B*) as well as varying the number of workers launched by the second *Kulla-Brick* (*D*&*C*). k-M/B(x)-D&C(x), Parsl, Makeflow and Jenkins reduce the content size sent to IDA stage (by using LZ4). This is not the case of TBB-IDA, which instead produces routine parallelism and in-memory storage. To make a fair comparison, TBB-IDA, Parsl, Makeflow and Jenkins were executed on all the 12 and 16 cores of the servers used in these experiments.

4.6. Usability comparison between Kulla and studied solutions

The first assessment is focused on the usability of the studied solutions.

Fig. 13 shows the steps for the implementation of the studied solutions. It shows, in grayscale, the three phases required for a developer/designer to do for getting an online solution: from configuration (white), to the development (light gray) and execution (gray). A developer/designer can reproduce the experiments performed with the evaluated solutions by following this step-by-step timeline description. The manual tasks are represented by large rectangles and automatic ones by small rectangles. White stars represents the actions to change form scale-in to scale-out development, whereas black stars represents third party frameworks that are not included in the solutions and are required for them to enable Scale-out deployment.

As it can be seen, the developers/designers perform less operations (manual) and scripting tasks with Kulla and Jenkins than the rest of solutions to get an online parallel pipeline solution. Nevertheless, the Kulla is the only solution that does not impose requirements (third party frameworks) to create a distributed solution by a Scale-out deployment. It is important to note that the configuration and deployment tasks only can be performed when downloading the virtual containers of each solution and this action is performed just one time. The execution tasks can be performed as many times as solutions be created by the developers.

4.6.1. Scale-in deployment results: A case study based on the processing of medical images

Fig. 14 shows, in the left vertical axis, the response times produced by each task performed by the evaluated solutions, such as I/O management and service times, produced by each stage (LZ4 and IDA). The *Kulla-Box* configuration is represented by *K-M/B(x)-D&C(x)*, whereas the rest are represented by *Parsl*, *TBB-IDA*,[5] Makeflow.[6] and Jenkins[6] when processing sets of medical images (see columns) by using the 12 cores available in *Server12*

Fig. 14 also shows, in the right vertical axis, the throughput produced by the solutions in the form of a line.

[5] This information is not available for TBB as the I/O is coupled with IDA processing and making the timers resulted in degraded performance for TBB-IDA.

[6] Due to the way of execute the tasks in parallel, it was not possible to break down the response times of each application or the I/O operations and they are shown as a single response time.

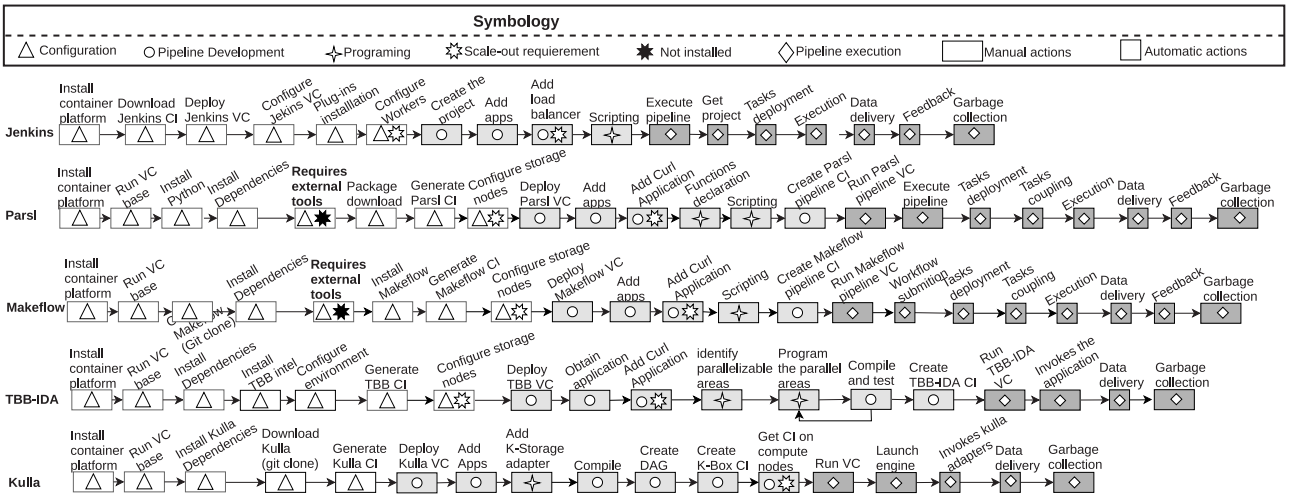


Fig. 13. A step-by-step of the implementation of the studied solutions.

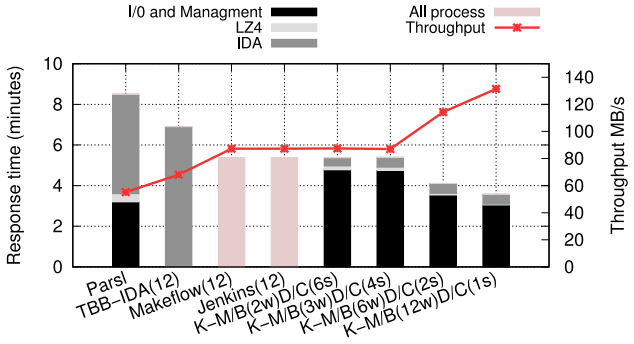


Fig. 14. Breakdown of response time per processing stage and throughput.

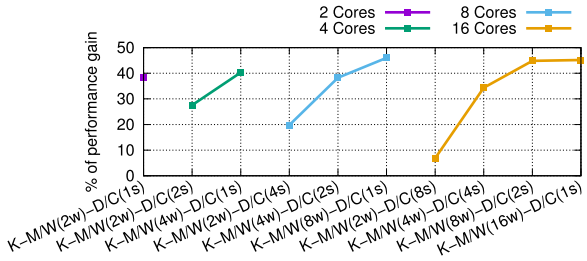


Fig. 15. Performance gain obtained by the Kulla Boxes to *TBB-IDA*.

As it can be seen, all configurations of *K-Box(w)* produced better response time than Parsl, TBB-IDA, Makeflow and Jenkins when using all available cores in the server. *Parsl* processed 27.5 GB in 8,3 Min for a Throughput of 56,08 MB/s. In turn, TBB-IDA performs this very task in 6,9 Min for a 68.01 MB/s. The two best configurations of Kulla (when using as workers in the *M/B* pattern as cores in the server) performed this very task in 3,54 min (114 MB/s) and 3,05 min (131 MB/s). *Makeflow* and *Jenkins* processed the 27.5 GB in 5,37 Min for a Throughput of 87.27 MB/s. Makeflow performance is better than Parsl and TBB-IDA by 36.83% and 22.17% respectively. Nevertheless, the Makeflow performance is similar to slowest Kulla configurations K-M/B(2w)D/C(6w) and K-M/B(3w)D/C(4w), but it is not enough to reach the performance of Kulla K-M/B(6w)D/C(2w) and K-M/B(12w)D/C(1w). The percentage of performance gain of these two Kulla configurations in comparison with Makeflow was 23.53% and 33.48% respectively.

In the case of Kulla configurations, we observed that the more workers in the *M/B* pattern, the less workload is delivered to the workers of the next pattern (*D&C*) and the less time required by this pattern to process contents; as a result, the performance of Kulla solutions is better than TBB-IDA. The improvement of Kulla's performance is increased from 6.88% (the lowest configuration) when using two workers in the *M/B* pattern to 45.99%.

When analyzing these results, we identified three main performance improvement causes: (i) In the first stage, the contents are processed by using a task parallel strategy (see LZ4 service time in Fig. 14). This reduces the idle time in the next stage (IDA); (ii) LZ4 reduces the size of the data delivered to IDA stage, which reduces the service time required by IDA in this stage to process the images (see IDA service time in Fig. 14); (iii) Although the increment of workers in D/C patterns results in an increment of data parallelism, it also increases the cost of I/O management (See I/O response time produced by Kulla configurations in Fig. 14). When Kulla configurations uses more than one D/C pattern, an increment of the I/O operations is required to store the processed data in the Sinks, as each D/C pattern launches five workers as default configuration and each one produces an output write operation. This increment in I/O operations for the data sinks reduces the effectiveness of data parallelism.

Fig. 15 shows the percentage of performance gain of Kulla Boxes in comparison with *TBB-IDA*, which were developed on a server of 16 cores (Server16). As it can be seen, the performance of best *K-Box* configurations (varying the number of workers of M/B pattern, which only launching a D&C pattern per worker) grows from 38.45% to be stable in 45% (e.g. 45.99 for W/B(8) and 45.69% for W/B(16)), which was similar to the experiments performed when using the server of 12 cores.

As shown in previous experiments, besides of performance improvement, *Kulla-Box* configurations also reduce the cost of withstanding failures of services as well as the storage utilization, which was achieved without analyzing/modifying the code of the applications to find routines to be executed in parallel.

4.7. Scale Out/In deployment results: The deployment of infrastructure agnostic distributed applications

This section shows the results of the experiments performed to conduct a performance comparison of Kulla with the traditional and popular state-of-the-art solutions described in previous section.

As it can be seen in Fig. 13, Makeflow and Parsl solutions requires additional software and tools to distribute data to the workers. These requirements were not installed in the cluster for the evaluation because Jenkins and Kulla do not require additional software (it is installed in the virtual containers).

In the *Brick-Of-Kulla-Boxes* solution, the first Kulla-Box implements a M/B pattern, where the manager launches workers that execute a Pipe&Block Kulla-Brick. The first stage of this Kulla-Brick included a LZ4 and a launcher of an image of Kulla-Box including a D&C Kulla-Brick. This image of Kulla-Box was used to create three Kulla-Boxes that were launched and deployed on different servers. Each D&C pattern included one master and as many workers as cores available in the servers. The exchange of data between M/B and D&C Kulla-Boxes was performed through the network I/O interface (sockets chosen as all servers are placed at the same site). The deployment of *Brick of Kulla-Boxes* solution resulted in a M/B(3w), D&C(12w) configuration.[7]

The performance of this solution was compared with the performance of *TBB-IDA*, *Parsl*, *Makeflow* and *Jenkins* solutions, which were configured to execute a pipeline for the encoding medical images stored in a server by using all its cores and then sending the encoded segments to the rest of servers (three servers). We performed the experiment described in previous section, but now the 55 medical images were processed by the solutions built with *Brick of Kulla-Boxes*, *Parsl*, *TBB-IDA Makeflow*, and *Jenkins* configurations. The response time produced by these solutions was captured to perform a performance assessment.

4.7.1. Analyzing results of the case study based on bricks of Kulla-Boxes

Fig. 16 shows, in the left vertical axis, the response times produced by the *Brick of Kulla-Boxes* configuration *K-M/B(6w)-D&C(6s)* as well as *Parsl(12)*, *Makeflow(12)*, *TBB-IDA(12)*, and Jenkins(12). In right vertical axis, Fig. 16 also shows the throughput produced by the solutions in the form of a line.

The *Brick of Kulla-Box* solution processed the image repository (27.5GB) in 6.18 min for a throughput of 4.44 GB/min. *TBB-IDA* spent 70,28 min (0,40 GB/min), *Parsl* spent 18,73 min (1.46 GB/min), *Jenkins* spent 12,45 min (2.2 GB/min) and *Makeflow* spent 10.89 min (2.52 GB/min) when executing their pipelines. *TBB-IDA* produced 18.34GB of extra capacity that is the cause of the delays observed by this type of solution, whereas the fault tolerance provided by *Brick of Kulla-Boxes* to withstand the same number of failures than *TBB-IDA* configuration was achieved for free as the compression degree in this type of digital products was high (in a range of 30%–40%).

4.7.2. Qualitative comparison

A qualitative comparison of the *Kulla* model with the evaluated solutions, as well as with other solutions available in literature, was also conducted by taking into account the results of the experimental evaluation and the usability considerations in this study. This comparison includes the feasibility of the solutions to build storage for data exchange (in distributed, in-memory and file system manners), the type of parallelism supported (task, data and routine), as well as the architecture used (engines to launch applications and Building Blocks for multiple coupling of solutions). As shown in Table 2, *Kulla* model provides developers and end-users with a comprehensive portfolio of features to be used when they build their data flow solutions.

[7] The end-user can design a different solution configuration by setting up the parameters of both patterns in the manager of the Brick of Kulla-Boxes; one parameter for servers and another for cores in each server.

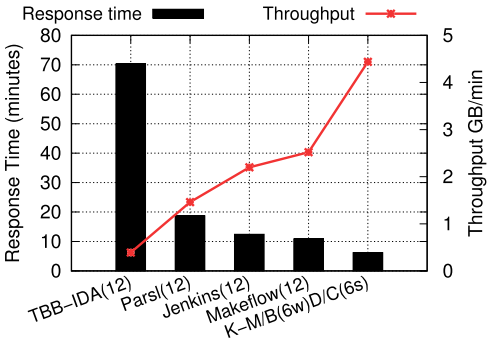


Fig. 16. Response time and throughput obtained by the evaluated solutions.

5. Related work

The encapsulation of applications into virtual containers has been explored in recent years (Boettiger, 2015) in areas such as bioinformatics (Belmann et al., 2015), archaeology (Marwick, 2017), software and web engineering (Cito et al., 2016), storage systems (Morales-Ferreira et al., 2018; Reyes-Anastacio et al., 2018), etc. This type of solution is only focused on the improvement of the application deployment for avoiding the troubleshooting issues in real-world scenarios. However, in scientific environments, workflows are also required to interconnect different applications for processing models about environment, climate (Skluzacek et al., 2016), etc.

In a previous work we proposed Sacbe (Gonzalez-Compean et al., 2018), a solution based on building blocks (BB) to manage multiple applications as black boxes and in-memory software patterns. Sacbe, implemented in Java language, builds end-to-end applications, based on in-memory storage, that process data before sending them to the cloud. Nevertheless, this model is only suitable for Java Virtual Machines and only the Pipe&Filter pattern was studied. Moreover, this was a model designed specifically for end-to-end cloud storage solutions, whereas *Kulla* not only exploits shared memory management, but also creates different types of parallelism patterns. *Kulla* model can be also used in a comprehensive manner for different types of applications, not only in storage systems.

Workflow engines (Montella et al., 2018b; Deelman et al., 2015; Liu et al., 2015; Albrecht et al., 2012) are similar to *Kulla* model as they enable organizations to create solutions for preparation, preprocessing and processing stages, which commonly can be executed in parallel (by using task parallelism models). Those workflows are quite useful for organizations to face up effects of the information accumulation (Gantz and Reinsel, 2012) produced by the constant information archival. In practice, those workflow frameworks are mainly focused on the application interconnections and are mainly available for virtual and physical machines. Although these workflows engines include parallel patterns mainly based on task parallelism (Babuji et al., 2019; Badia et al., 2015), there is still an opportunity window for engines to improve the resource profitability, which also should be achieved without affecting the feasibility and flexibility of the solutions/systems. Moreover, there is an opportunity window for this type of solution to include methods to improve the resource profitability, the portability of workflows and the flexibility to build solutions not only based on task parallelism. In this context, we also performed a direct comparison with solutions built by using one of this type of solution (Babuji et al., 2019) with solutions built by using *Kulla* model.

The former examples represent an important issue due to the volume of data to be processed and the response time experimented by end-users depending on data allocation (Perez et al.,

Table 2
A qualitative comparison.

	Storage			Parallelism			Architecture		
	Distributed	In-memory	File system	Data	Task	Routine	Engine	Building	Block
GrPPI Blas and Garcıa (2016)		✓	✓	✓	✓	✓	✓		
Triana Taylor et al. (2007)			✓		✓		✓		
Sacbe Gonzalez-Compean et al. (2018)	✓	✓	✓					✓	
Parsl Babuji et al. (2019)	✓		✓		✓		✓		
Makeflow Albrecht et al. (2012)	✓[a]		✓		✓		✓		
Jenkins Armenise (2015)	✓[a]		✓		✓		✓		
Kulla	✓	✓	✓	✓	✓		✓	✓	

[a]By using external tools.

2003). Software processing pipelines were proposed to reduce those response times (Gonzalez-Compean et al., 2017), while the use of data parallel mechanisms have been studied (Barney et al., 2010) for Big Data scenarios (Dean and Ghemawat, 2008). The MapReduce processing model (Dean and Ghemawat, 2008) is based on the Single Program Multiple Data (SPMD) paradigm (Darema, 2001), which is inspired by divide and conquer paradigm to reduce the incoming data for meaning information. This traditional method enables developers to execute a replicated application in concurrent manner for delivering results to consolidation instances, which improves the performance of analytic solutions (Pieterse and Black, 2004). In *Kulla*, these patterns are available (D&C method), but its instances are encapsulated within virtual containers including its software dependencies such as libraries and environment variables. This not only reduces the need for troubleshooting, but also it improves the feasibility for SPMD solutions to combine different patterns in immutable and agnostic solutions.

Solutions based on parallel patterns are also available to improve the application efficiency (del Rio Astorga et al., 2018; Badia et al., 2015). However, the implementation of applications for processing data in parallel is not a trivial task, as parts of the application (routines or cycles) must be identified to be executed in parallel and not all IT staffers are familiar with this type of processing. Moreover, frameworks for paralleling applications (Reinders, 2007; Gropp et al., 1999; White, 2012; Mavridis and Karatza, 2017) usually involve issues related to dependencies, environment variables, libraries, and infrastructure that are complex for IT staff to fix. This avoids developers to grant application immutability/portability, reducing interoperability and/or requiring for end-users to perform troubleshooting IT issues.

Parallel patterns have been proposed for developers to avoid dealing with complexity of parallelism frameworks such as MPI (Gropp et al., 1999), OpenMP (Chandra et al., 2001) and TBB (Reinders, 2007). A survey of the different parallel programming models and tools are available today, with special consideration to their suitability for high-performance computing, was presented in Diaz et al. (2012). More sophisticated patterns have been proposed in del Rio Astorga et al. (2017). However, all those patterns must be added to the application code and the templates must be used in those code sections where parallelism is feasible (Blas and Garcıa, 2016; Sotomayor et al., 2017). In turn, *Kulla* is not focused on routine parallelism, but focused on data and tasks parallelism patterns as well as pipelines, which enable developers to create parallel patterns without analyzing or altering routines of application code. This is a quite interesting feature for the scientific community that requires to deploy solutions, but either is not familiar with the analysis of application codes to identify sections suitable for being improved by parallelism, or they simply have no time to do this task.

6. Conclusions and future work

This paper presented the design, development, and implementation of *Kulla*, a virtual container-centric construction model that mixes loosely coupled services with parallel programming model for building infrastructure-agnostic distributed and parallel applications.

Kulla construction model relies on the interconnectedness of the following software structures: (i) Kulla-Blocks that create Maps of a given application with its dependencies and environment requirements; (ii) Kulla-Bricks that produce implicit parallelism by coupling a set of Kulla-Blocks in the form of parallel patterns, which are built in transparent manner and without modifying the application code by using methods such as Divide&Containerize, Pipe&Blocks and Manager/Blocks; (iii) Kulla-Boxes that convert Kulla-Blocks, Kulla-Bricks or a combination of these structures into an infrastructure-agnostic application by taking advance of lightweight and immutability features of virtual containers (VCs); (iv) Brick-of-Kulla-Boxes that interconnects Kulla-Boxes to build infrastructure-agnostic distributed and/or parallel applications. Those structures enable developers/end-users to create continuous delivery and to build parallel patterns without altering/modifying the applications code to improve the efficiency of these applications. All these structures are operable by using an ETL model (Extract-Transform-Load), where Extraction and load are mapped to I/O interfaces. This model provides Kulla structures with self-similarity property, which means complex systems can be built by the coupling multiple of these structures even in recursive manner.

All these combinations are presented to the end-users as a *Kulla-Box*: a single application where an implicit process interaction and parallelism; as a result, the management of resources such as Network, cores, RAM, storage locations, parallelism, intercommunication is performed inside of the *Kulla-Boxes* by the Kulla control software instances. Multiple solutions could be built by reusing applications that also could be deployed on different types of IT infrastructures, which shows the benefits of agnostic and portability properties provided to the applications by this model.

To show the feasibility of the *Kulla* model, real-world applications were created by using different distributed and parallel *Kulla* solutions to solve use cases of satellite and medical imagery processing. An experimental evaluation was conducted by testing different experimental scenarios where *Kulla* instances were deployed on servers and cluster of VCs. In this evaluation, we performed a direct comparison of *Kulla-Boxes* with solutions available in the state of the art such as serial and parallel applications created by using Intel TBB and the workflow engines Parsl, Makeflow and Jenkins. This evaluation revealed the feasibility and efficacy of *Kulla's* self-similarity property to build infrastructure-agnostic solutions. The experimental evaluation also revealed

the efficiency, in terms of performance, of the *Kulla* model in comparison to studied solutions.

Kulla implements Scale-in/out for improving resource profitability but only in deployment time. Currently, we are working on models for improving the profitability of resources in an automatic manner and in execution time. We are also developing a framework for building Kulla solutions defined by code, by using programmable scripts, instead of a GUI or the Kulla-Silo service.

CRediT authorship contribution statement

Hugo G. Reyes-Anastacio: Methodology, Conceptualization, Software, Data curation. **J.L Gonzalez-Compean:** Supervision, Writing - original draft, Validation, Conceptualization, Formal analysis, Project administration. **Victor J. Sosa-Sosa:** Conceptualization, Writing - review & editing, Validation. **Jesus Carretero:** Writing - review & editing, Conceptualization. **Javier Garcia-Blas:** Writing - review & editing, Validation, Resources.

Acknowledgments

This work has been partially supported by the EU project "ASPIDE: Exascale Programing Models for Extreme Data Processing" under grant 801091 and the project "CABAHLA-CM: Convergencia Big data-Hpc: de los sensores a las Aplicaciones" S2018/TCS-4423 from Madrid Regional Government.

References

Abushab, K., Suleiman, M., Alajerami, Y., Alagha, S., ALnahal, M., Najim, A., Naser, M., 2018. Evaluation of advanced medical imaging services at Governmental Hospitals-Gaza Governorates, Palestine. J. Radiat. Res. Appl. Sci. 11 (1), 43–48.

Albrecht, M., Donnelly, P., Bui, P., Thain, D., 2012. Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids. In: Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies, pp. 1–13.

Armenise, V., 2015. Continuous delivery with jenkins: Jenkins solutions to implement continuous delivery. In: 2015 IEEE/ACM 3rd International Workshop on Release Engineering. IEEE, pp. 24–27.

Babuji, Y., Woodard, A., Li, Z., Katz, D.S., Clifford, B., Kumar, R., Lacinski, L., Chard, R., Wozniak, J.M., Foster, I., et al., 2019. Parsl: Pervasive parallel programming in python. In: Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing, pp. 25–36.

Badia, R.M., Conejero, J., Diaz, C., Ejarque, J., Lezzi, D., Lordan, F., Ramon-Cortes, C., Sirvent, R., 2015. Comp superscalar, an interoperable programming framework. SoftwareX 3, 32–36.

Barney, B., et al., 2010. Introduction to Parallel Computing, Vol. 6, No. 13. Lawrence Livermore National Laboratory, p. 10.

Belmann, P., Dröge, J., Bremges, A., McHardy, A.C., Sczyrba, A., Barton, M.D., 2015. Bioboxes: standardised containers for interchangeable bioinformatics software. Gigascience 4 (1), 47.

Blas, J.G., Garcıa, J.D., 2016. A C++ generic parallel pattern interface for stream processing. In: Algorithms and Architectures for Parallel Processing: 16th International Conference, ICA3PP 2016, Granada, Spain, December 14–16, 2016, Proceedings, Vol. 10048. Springer, pp. 74–87.

Boettiger, C., 2015. An introduction to docker for reproducible research. Oper. Syst. Rev. 49 (1), 71–79.

Brozek, M., 2019. A Forrester Consulting Thought Leadership Paper Commissioned by Dell EMC, Intel, and Red hat, Containers: Real adoption and use cases in 2017. 2007, https://i.dell.com/sites/doccontent/business/solutions/whitepapers/en/Documents/Containers_Real_Adoption_2017_Dell_EMC_Forrester_Paper.pdf. (Last Accessed 4 September 2019).

Buschmann, F., Henney, K., Schmidt, D.C., 2007. Pattern-oriented software architecture, on patterns and pattern languages, Vol. 5. John wiley & sons.

Celesti, A., Galletta, A., Fazio, M., Villari, M., 2019. Towards hybrid multi-cloud storage systems: Understanding how to perform data transfer. Big Data Res. 16, 1–17.

Chandra, R., Dagum, L., Kohr, D., Menon, R., Maydan, D., McDonald, J., 2001. Parallel Programming in OpenMP. Morgan Kaufmann.

Cito, J., Ferme, V., Gall, H.C., 2016. Using docker containers to improve reproducibility in software and web engineering research. In: International Conference on Web Engineering. Springer, pp. 609–612.

Collet, Y., 2017. LZ4 - Extremely fast compression. https://github.com/lz4/lz4,

Darema, F., 2001. The spmd model: Past, present and future. In: European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting. Springer, p. 1.

de Alfonso, C., Calatrava, A., Moltó, G., 2017. Container-based virtual elastic clusters. J. Syst. Softw. 127, 1–11.

Dean, J., Ghemawat, S., 2008. Mapreduce: simplified data processing on large clusters. Commun. ACM 51 (1), 107–113.

Deelman, E., Vahi, K., Juve, G., Rynge, M., Callaghan, S., Maechling, P.J., Mayani, R., Chen, W., da Silva, R.F., Livny, M., et al., 2015. Pegasus, a workflow management system for science automation. Future Gener. Comput. Syst. 46, 17–35.

del Rio Astorga, D., Dolz, M.F., Fernández, J., García, J.D., 2017. Supporting advanced patterns in G r PPI, a generic parallel pattern interface. In: European Conference on Parallel Processing. Springer, pp. 55–67.

del Rio Astorga, D., Dolz, M.F., Fernández, J., García, J.D., 2018. Paving the way towards high-level parallel pattern interfaces for data stream processing. Future Gener. Comput. Syst. 87, 228–241.

Diaz, J., Munoz-Caro, C., Nino, A., 2012. A survey of parallel programming models and tools in the multi and many-core era. IEEE Trans. Parallel Distrib. Syst. 23 (8), 1369–1386.

Ferguson, J., 2011. Jenkins: The Definitive Guide. USA. O'Reilly Media.

Gantz, J., Reinsel, D., 2012. The Digital Universe in 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East, Vol. 2007. IDC iView: IDC Analyze the Future, pp. 1–16.

Garrido, J.M., 2002. Performance Modeling of Operating Systems Using Object-Oriented Simulation: A Practical Introduction, first ed. Series in Computer Science, Springer US.

Gartner, Inc. and/or its affiliates, 2017. 6 best practices for creating a container platform strategy. https://www.gartner.com/smarterwithgartner/6-best-practices-for-creating-a-container-platform-strategy/. (Last Accessed 4 September 2019).

Gonzalez, J.L., Carretero, J., Sosa-Sosa, V.J., Sanchez, L.M., Bergua, B., 2015. SkyCDS: A resilient content delivery service based on diversified cloud storage. Simul. Model. Pract. Theory 54, 64–85.

Gonzalez, J.L., Marcelín-Jiménez, R., 2011. Phoenix: A fault-tolerant distributed web storage based on URLs. In: 2011 IEEE 9th International Symposium on Parallel and Distributed Processing with Applications. ISPA, IEEE, pp. 282–287.

Gonzalez-Compean, J., Sosa-Sosa, V.J., Diaz-Perez, A., Carretero, J., Marcelin-Jimenez, R., 2017. FedIDS: a federated cloud storage architecture and satellite image delivery service for building dependable geospatial platforms. Int. J. Digit. Earth 1–22.

Gonzalez-Compean, J., Sosa-Sosa, V., Diaz-Perez, A., Carretero, J., Yanez-Sierra, J., 2018. Sacbe: A building block approach for constructing efficient and flexible end-to-end cloud storage. J. Syst. Softw. 135, 143–156.

Grawinkel, M., Mardaus, M., Süß, T., Brinkmann, A., 2015. Evaluation of a hash-compress-encrypt pipeline for storage system applications. In: Networking, Architecture and Storage (NAS), 2015 IEEE International Conference on. IEEE, pp. 355–356.

Gropp, W., Thakur, R., Lusk, E., 1999. Using MPI-2: Advanced Features of the Message Passing Interface. MIT Press.

Hayden, M., Carbone, R., 2015. Securing linux containers. GIAC (GCUX) Gold Certification, Creative Commons Attribution-ShareAlike 4.0 International License 19.

Karmel, A., Chandramouli, R., Iorga, M., Nist definition of microservices, application containers and System Virtual Machines, National Institute of Standards and Technology (NIST) Special Publication, pp. 1-5.

Liu, J., Pacitti, E., Valduriez, P., Mattoso, M., 2015. A survey of data-intensive scientific workflow management. J. Grid Comput. 13 (4), 457–493.

Marcelín-Jiménez, R., Rajsbaum, S., 2003. Cyclic strategies for balanced and fault-tolerant distributed storage. In: Latin-American Symposium on Dependable Computing. Springer, pp. 214–233.

Marcelin-Jimenez, R., Rajsbaum, S., Stevens, B., 2006. Cyclic storage for fault-tolerant distributed executions. IEEE Trans. Parallel Distrib. Syst. 17 (9), 1028–1036.

Marwick, B., 2017. Computational reproducibility in archaeological research: basic principles and a case study of their implementation. J. Archaeol. Method Theory 24 (2), 424–450.

Mavridis, I., Karatza, H., 2017. Performance evaluation of cloud-based log file analysis with Apache Hadoop and Apache Spark. J. Syst. Softw. 125, 133–151.

Miranda, J., Murillo, J.M., Guillén, J., Canal, C., 2012. Identifying adaptation needs to avoid the vendor lock-in effect in the deployment of cloud SBAs. In: Proceedings of the 2nd International Workshop on Adaptive Services for the Future Internet and 6th International Workshop on Web APIs and Service Mashups. ACM, pp. 12–19.

Montella, R., Brizius, A., Di Luccio, D., Porter, C., Elliot, J., Madduri, R., Kelly, D., Riccio, A., Foster, I., 2018a. Using the FACE-IT portal and workflow engine for operational food quality prediction and assessment: An application to mussel farms monitoring in the Bay of Napoli, Italy. Future Gener. Comput. Syst..

Montella, R., Kelly, D., Xiong, W., Brizius, A., Elliott, J., Madduri, R., Maheshwari, K., Porter, C., Vilter, P., Wilde, M., et al., 2015. FACE-IT: A science gateway for food security research. Concurr. Comput.: Pract. Exper. 27 (16), 4423–4436.

Montella, R., Kosta, S., Foster, I., 2018b. DYNAMO: Distributed leisure yacht-carried sensor-network for atmosphere and marine data crowdsourcing applications. In: Cloud Engineering (IC2E), 2018 IEEE International Conference on. IEEE, pp. 333–339.

Morales-Ferreira, P., Santiago-Duran, M., Gaytan-Diaz, C., Gonzalez-Compean, J., Sosa-Sosa, V.J., Lopez-Arevalo, I., 2018. A data distribution service for cloud and containerized storage based on information dispersal. In: Service-Oriented System Engineering (SOSE), 2018 IEEE Symposium on. IEEE, pp. 86–95.

Opara-Martins, J., Sahandi, R., Tian, F., 2014. Critical review of vendor lock-in and its impact on adoption of cloud computing. In: International Conference on Information Society. I-Society 2014, IEEE, pp. 92–97.

Perez, J.M., Garcia, F., Carretero, J., Calderon, A., Sanchez, L.M., 2003. Data allocation and load balancing for heterogeneous cluster storage systems. In: CCGrid 2003. 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid, 2003. Proceedings. pp. 718–723. http://dx.doi.org/10.1109/CCGRID.2003.1199438.

Pieterse, V., Black, P.E., 2004. single program multiple data. In: Pieterse, V., Black, P.E. (Eds.), *Dictionary of Algorithms and Data Structures* [online]. Available from: https://xlinux.nist.gov/dads/HTML/singleprogrm.html. (Accessed 14 March 2018).

Posner, E.A., Spier, K.E., Vermeule, A., 2010. Divide and conquer. J. Legal Anal. 2 (2), 417–471.

Quezada Naquid, M., Marcelín Jiménez, R., López Guerrero, M., 2010. Fault-tolerance and load-balance tradeoff in a distributed storage system. Comput. Sist. 14 (2), 151–163.

Rabin, M.O., 1989. Efficient dispersal of information for security, load balancing, and fault tolerance. J. ACM 36 (2), 335–348. http://dx.doi.org/10.1145/62044.62050, URL http://doi.acm.org/10.1145/62044.62050.

Reinders, J., 2007. Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism. O'Reilly Media, Inc.

Reyes-Anastacio, H.G., Gonzalez-Compean, J., Morales-Sandoval, M., Carretero, J., 2018. A data integrity verification service for cloud storage based on building blocks. In: 2018 8th International Conference on Computer Science and Information Technology. CSIT, IEEE, pp. 201–206.

Sharma, P., Chaufournier, L., Shenoy, P., Tay, Y., 2016. Containers and virtual machines at scale: A comparative study. In: Proceedings of the 17th International Middleware Conference. ACM, p. 1.

Skluzacek, T.J., Chard, K., Foster, I., 2016. Klimatic: a virtual data lake for harvesting and distribution of geospatial data. In: Parallel Data Storage and Data Intensive Scalable Computing Systems (PDSW-DISCS), 2016 1st Joint International Workshop on. IEEE, pp. 31–36.

Sotomayor, R., Sanchez, L.M., Blas, J.G., Fernandez, J., Garcia, J.D., 2017. Automatic CPU/GPU generation of multi-versioned OpenCL kernels for C++ scientific applications. Int. J. Parallel Program. 45 (2), 262–282.

Souppaya, M., Morello, J., Scarfone, K., 2017. Application Container Security Guide (2nd Draft), Vol. 800. NIST Special Publication.

Spillner, J., Bombach, G., Matthischke, S., Muller, J., Tzschichholz, R., Schill, A., 2011. Information dispersion over redundant arrays of optimal cloud storage for desktop users. In: 2011 Fourth IEEE International Conference on Utility and Cloud Computing. IEEE, pp. 1–8.

Stenberg, D., Fandrich, D., Tse, Y., 2012. libcurl: The multiprotocol file transfer library. online at: http://curl.haxx.se/libcurl.

Taylor, I., Shields, M., Wang, I., Harrison, A., 2007. The triana workflow environment: Architecture and applications. In: Workflows for e-Science. Springer, pp. 320–339.

Tsidulko, J., 2015. Overnight AWS Outage Reminds World How Important AWS Stability Really Is. CRN, Online; (Accessed 1 June 2016).

White, T., 2012. Hadoop: The Definitive Guide. O'Reilly Media, Inc.

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

Zhang, Q., Li, S., Li, Z., Xing, Y., Yang, Z., Dai, Y., 2015. CHARM: A cost-efficient multi-cloud data hosting scheme with high availability. IEEE Trans. Cloud Comput. 3 (3), 372–386.