Reyes-Anastacio, H. G., Gonzalez-Compean, J., Sosa-Sosa, V. J., Carretero, J. & Garcia-Blas, J. (2020). Kulla, a container-centric construction model for building infrastructure-agnostic distributed and parallel applications. *Journal of Systems and Software, 168*, 110665.

DOI: 10.1016/j.jss.2020.110665

Kulla: A construction model based on virtual containers for parallel and distributed agnostic application composition

Hugo G. Reyes-Anastacio, Jose L. Gonzalez-Compean, Victor J. Sosa-Sosa, Jesus Carretero, *Senior Member, IEEE* and Javier Garcia-Blas, *Member, IEEE*

Abstract—This paper presents the design, development, and implementation of Kulla, a construction model that takes advance of lightweight and immutability features of virtual containers (VCs) in the composition of parallel and distributed agnostic applications. In this model, the applications as well as dependencies and environment settings are encapsulated, with in-memory data storage mechanisms and I/O network/file call management systems into interoperable construction units called *Kulla-Blocks*. These units can be chained in the form of graphs and parallel patterns called *Kulla-Bricks*, which can be created without altering/modifying the code of applications. Agnostic applications can be built, on-the-fly and on-demand, by using recursive combinations of Kulla instances by grouping them in deployment structures called *Kulla-Boxes*, which are encapsulated into VCs. Agnostic distributed applications can be built in the form of Brick of Kulla-Boxes by using deployment strategies to improve the resource profitability. To show the feasibility and flexibility of this model, parallel patterns such as *Divide-and-Containerize* (data parallelism), *Pipe&Blocks* (streaming) and *Manager/Blocks* (task parallelism) were developed by using Kulla instances. Different parallel and/or distributed Kulla solutions were created for processing satellite and medical imagery by combining real-world applications resulting in different case studies. The experimental evaluation performed in different IT infrastructures revealed the efficiency of Kulla model in comparison with serial and parallel applications created by using IntelTBB.

1 Introduction

Troubleshooting IT issues, interoperability of applications, as well as the profitability of compute resources are challenging tasks that arise when organizations deploy software solutions on IT infrastructures (any of cloud, clusters or servers).

Troubleshooting IT issues are quite common in organizational environments where the applications are commonly migrated/installed to/in different departments over different types of installed IT infrastructures. A troubleshooting process is required when the immutability and portability of applications is not granted by developers and issues related to failed installations, missed dependencies or misplaced environment settings must be solved by IT staff. In this type of situation, IT staff commonly considers debugging procedures that, depending on the codification of a given application, could take valuable time causing either downtime or disturbing business continuity [1], [2]. To overcome those problems, virtual containers (VC) have become a popular solution for organizations to deploy applications on different infrastructures in an immutable manner [3] [1]. Moreover, studies [6] have shown that VCs are lightweight in comparison with the virtual machines.

Nevertheless, this virtualization technology does not solve application interoperability issues that arise when multiple applications must be integrated into a single system/solution. In this type of solution, several preprocessing tasks are concatenated to create solutions of chained applications [7], [8], [9], [10], [11]. The outputs of some applications represent the inputs of others; as a result, construction models and frameworks are required by developers to build these solutions [9], [10], [12]. Data management strategies are also required for delivering data among the applications [13] [14] when are grouped into a single solution. For instance, organizations focused on the management of health [15], [16] and satellite [17], [18] data commonly process large volumes of images (datasets) to add properties to contents that are sent to data centers and/or cloud services. Different properties are desirable to be added to those datasets before sending them to public/private cloud storage services. For instance, the contents are preprocessed to reduce data volume [18] to save storage space and to reduce costs, which adds frugality 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 to them even when organizations store them in private clusters/servers.

In practice however, the organizations end up deal-

Hugo G. Reyes-Anastacio, Jose L. Gonzalez-Compean are with CINVESTAV-Tamaulipas (Mexico).
Victor J. Sosa-Sosa is with CINVESTAV-Tamaulipas (Mexico) and University Carlos III of Madrid (Spain).
Jesus Carretero, Javier Garcia-Blas are with the University Carlos III of Madrid (Spain).
Contact Email: jgonzalez@tamps.cinvestav.mx

1. Market studies predicting that "By the year 2020, more than 50% of companies will use container technology, up from less than 20% in 201" [4], 63% have more than 100 instances deployed, and 82% expect to have more than 100 deployed within the next two years, according to "Containers: Real Adoption and Use Cases in 2017," [5]

ing with systems integrating several applications and IT commonly struggle to solve interoperability issues as this integration should be performed in dynamic and flexible manners.

In addition, the efficiency of aforementioned processes is key for organizations such as hospitals [15], [16] and space agencies [17] because of the rate of the content production of these organizations is massive. In this context, the $profitability$ of IT resources becomes crucial because of the service experience of end-users is important for decision-making processes in these scenarios. By profitability of resources we refer to the way in which a solution integrating different applications can be deployed on a given infrastructure by using as many resources as possible for processing each content (e.g. using the servers available as well as number cores and memory per server).

An example of solutions created to reduce service times and to improve the resource profitability are the parallelism patterns [19], [20].

Nevertheless, 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 familar with this type of process. Moreover, this type of processing is performed by using a given framework [21], [22], [23] where some issues related to dependencies, environment variables, libraries, and infrastructure could arise. This avoids developers to grant application immutability/portability, reducing interoperability and/or requiring for end-users to perform troubleshooting IT issues.

In this context, there is a need for solutions not only enabling organizations to add value to their contents but also including features required in real-world scenarios such as feasibility, flexibility and efficiency. $Feasibility$ is required for avoiding/reducing the need for troubleshooting IT procedures. $Flexibility$ is required 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 for processing contents by profiting as many resources as available in IT infrastructure for avoiding the side effects of these preprocessing tasks in the service experience of the end-users, which also should be performed without affecting the feasibility and flexibility of the solutions/systems.

This paper presents the design, development, and implementation of Kulla, a construction model that takes advance of lightweight and immutability features of virtual containers (VCs) in the composition of parallel and distributed agnostic applications.

In this model, the applications as well as dependencies and environment settings are encapsulated, with in-memory data storage mechanisms and I/O network/file call management systems into interoperable construction units called *Kulla-Blocks*. These units can be chained in the form of graphs and parallel patterns called *Kulla-Bricks*, which can be created without altering/modifying the code of applications.

To show the feasibility and flexibility of this model, parallel patterns such as *Divide-and-Containerize* (data parallelism), *Pipe&Blocks* (streaming) and *Manager/Blocks* (task parallelism) were developed by using Kulla instances.

Agnostic applications can be built, on-the-fly and on-demand, by using recursive combinations of Kulla instances by grouping them in deployment structures called *Kulla-Boxes*. The *Kulla-Boxes* are encapsulated into virtual containers and presented to end-users as a single system. This avoids organizations to perform troubleshooting processes when having container platform installed.

In kulla model all the software instances are self-similar to the smallest construction unit (Kulla-Block); as a result, the Kulla-Boxes also can be grouped in the form of Bricks (Bricks of Kulla-Boxes), which enables end-users/developers to create agnostic distributed and/or parallel applications.

In order to take advantage of the self-similarity property in the application composition process, we defined a set of Kulla-Box deployment strategies such as *Scale-In* and *Scale-Out* (or both).

These strategies were designed to improve the profitability of computing resources in infrastructures where Kulla-Boxes are deployed on and for enabling end-users and developers to create agnostic distributed and/or parallel applications.

Scale-In deployment strategy enables organizations to improve the utilization of computers with large number of cores, whereas Scale-Out is suitable to deploy solutions on a cluster of computers/virtual machines. Scale-Out/In solutions take advantage of both types of resources (cluster of nodes and cores in the nodes).

Different parallel and/or distributed Kulla solutions were created for processing satellite and medical imagery by combining real-world applications, which resulting in different case studies.

The experimental evaluation, where different scenarios were tested by performing comparison of Kulla-Boxes with serial and parallel (TBB) applications, showed the efficacy of self-similarity property to achieve interoprability to build agnostic solutions on different infrastructures in flexible and dynamic manners. In terms of performance, the evaluation showed the efficiency of Kulla model in comparison to traditional schemes.

The main contributions of this paper are:

- A container-based construction model for the composition of agnostic distributed and/or parallel applications. This model solves the interoprability issues and avoids to perform troubleshooting processes when deploying solutions on a given infrastructure.
- A scheme to build parallel patterns. The self-similarity property of the kulla software instances enables end-users/developers to create different patterns and combination of them to solve efficiency issues. Patterns methods were created to show the benefits of this property such as *Divide-and-Containerize* for data parallelism, *Pipe&Blocks* for streaming and *Manager/Blocks* for task parallelism as well as combinations of these patterns.
- A strategy for deploying Kulla solutions on different type of computing resources in a flexible and dynamic manner, which adds agnostic property to solutions and improves the profitability of resources.

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

2 Related Work

The encapsulation of applications into virtual containers has been explored in recent years [24] in areas such as bioinformatics [25], archaeology [26], software and web engineering [27], storage systems [28], [29], etc. This type of solutions are 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 [11], etc.

In previous works, we proposed a software architecture, named Sacbe, based on building blocks (BB) to manage multiple applications as black boxes [13]. A prototype of an architecture based on BBs implemented in Java was developed for building end-to-end applications that ensures and stores data in the cloud. Nevertheless, this model is only suitable for Java Virtual Machines and parallel patterns were not studied or solved, as only pipelines were defined. Moreover, this was a specific purpose model designed for end-to-end cloud storage solutions, whereas Kulla is focused on black boxes and parallelism patterns to exploit the shared memory management, which was not available in Sacbe model. Moreover, Kulla model can be used in a comprehensive manner for different types of applications, not only in storage systems.

In satellite or health image processing scenarios, the size of raw satellite images is around GB's, whereas the size of images corrected reaches hundreds of MB's. Thus, images are preprocessed to be ensured in terms of confidentiality (encryption/decryption) and reliability (fault tolerance based on error correcting). These images, in both domains, are also processed to achieve information to identify objects and producing information about these images. It is also expected that those images will be stored for large periods of time as are considered either heritage (in the case of satellite image) or extremely sensitive content (in the case of medical images). Archival produces then an information accumulation effect [30]. The development of solutions including preprocessing and processing stages [31], [32], [33] are examples of workflows. The combination of these solutions is required for fixing the application interoperability. Moreover, in practice, these workflows must be built in an interoperable manner. Nevertheless, the workflow frameworks are focused on the application interconnections and the resource profitability is not considered in this type of frameworks.

The former examples represent an important issue due to the volume of data to be processed and the response time experimented by end-users. Software processing pipelines were proposed to reduce those response times [18], while the use of data parallel mechanisms have been studied [34] for Big Data scenarios (Map-Reduce) [35]. The divide and containerize proposed in Kulla represents an alternative to pipelines for processing large files (as is the case of satellite or medical images) as Map-Reduce is commonly focused on data analytic/preprocessing [35].

The MapReduce processing model [35] is based on the model Single Program Multiple Data (SPMD) [36]. In these solutions, a given application is replicated to create workers, which execute the application in concurrent manner. For instance, data parallelism schemes used in Big Data scenarios (e.g. Map-Reduce [35] and design of pattern) enables organizations to improve the performance of their solutions by deploying applications in the form of workers on virtual spaces executed in physical resources. The SPMD model allows to split the input data into multiples blocks and to execute multiple copies of the same program simultaneously, so that each program processes its own block of data. Each copy of the program runs as an independent process and typically each process runs on its own processor [37]. Instead, in the divided and containerized proposal, each program is encapsulated within a Building Block (BB). All the BBs are encapsulated in a different container image, including its software dependencies such as: libraries, environment variables, etc.. The execution of different container images generates multiple virtual containers and those containers are interconnected between themselves creating a pipeline, where the segmentation and integration process can be added to create a SPMD solution. The combination of this proposal and the Kulla construction model enables organizations to deploy SPMD models on computers in an immutable manner, whereas the black box model enables developers to apply SPMD models to different types of applications.

Parallelism patterns templates has been proposed for developers to avoid dealing with complexity of parallelism frameworks such as MPI, OpenMP and TBB. A survey of the different parallel programming models and tools available today, with special consideration to their suitability for high-performance computing, was presented in [38]. More sophisticated patterns has been proposed in [39]. 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 [40], [41]. Kulla is not focused on routine parallelism, but also focused on data and tasks parallelism patterns, which enable developers to create parallel patterns without analyzing nor altering routines of application code. This is a quite interesting feature for 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.

3 Design principles of Kulla construction model

In this section, we describe the design principles of Kulla construction model, which are based on three types of structures. The first one are the construction units called Kulla-Blocks for developers to manage applications as building blocks (See Kulla-Blocks representation in Figure 1). The second one are pattern constructive units called Kulla-Bricks,

which have been designed for building parallel patterns by chaining Kulla-Blocks (See Kulla-Block representation in Figure 7). The last one are deployment structures called *Kulla-Box*, which were designed for deploying Kulla instances (either Blocks or Bricks) on different types of infrastructures such as cloud, cluster and server. (See Kulla-Blocks representation in Figure 7). These structures will be deeply described in next sections.

3.1 *Kulla-Block*: Building block structures

In Kulla model, a *Building Block* is a logical construction structure that integrates applications with dependencies and environment settings in a software instance called *Kulla-Block*. This instance also includes an abstraction called $Filter$, which represents a controller of native Kulla libraries for managing I/O calls (memory, network and file system) in a transparent manner and to launch applications in automatic and on-the-fly manners. In a Kulla-Block, each application is controlled by a filter; as a result, a Kulla-Block can be represented by the following notation: $App\_i = \{F\_App1, F\_App2, F\_AppN\} \in Kulla - Block\_j$ where Apps from 1 to N represents the libraries, classes and dependencies required by App_i to be successfully executed in a given infrastructure. This means all components of a given application are encapsulated into $Kulla - Block\_j$.

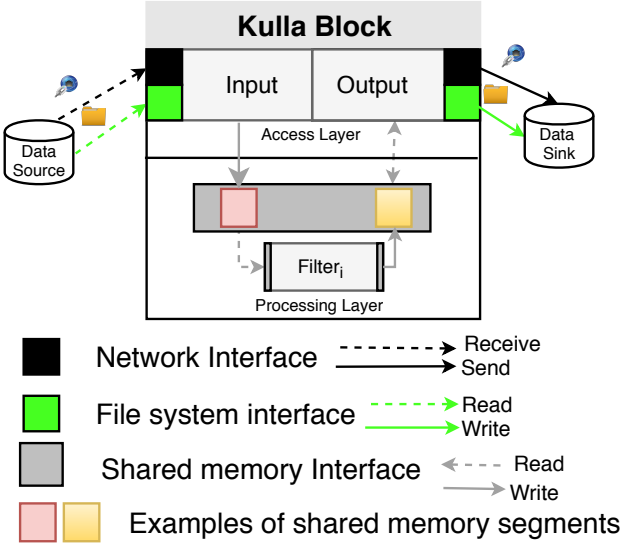

Fig. 1. A Kulla-Block conceptual representation.

Figure 1 shows the design of a *Kulla-Block*. As it can be seen, a Kulla-Block considers access and processing layers. The access layer includes *Input* and *Output* interfaces for filters to manage the incoming and outgoing data produced by their applications ($Kulla - Block\_j[in] = Input$ and $Kulla - Block\_j[out] = Output$)

The processing layer is an area for the deployment of $Filters$ where a data exchange area based on shared memory called $K - Storage$ is built by using shared resource pattern. K-Storage can be accessed through pointers that enables Filters to manage temporal storage to sharing information with either other filters, which avoids the usage of expensive I/O interfaces (i.e network and file system).

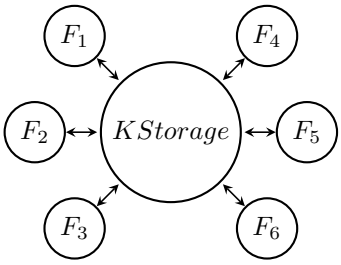

Fig. 2. Shared results allocated in a common space called KStorage

A Kulla-Block instance ($Node$) follows the traditional ETL process (Extract, Transform and Load), which means a Kulla-Block receives data from a *Data Source* (e.g. any of HD partition, cloud location, or another Kulla-Block) through the input interface (either network or file system). The filter(s) stores the data (d_x) retrieved from a edge $DataSource\_d \rightarrow Kulla - Block\_j(in) \rightarrow k - Storage\_Loc(in)$. The d_x are retrieved by the application (App_i) launched in the Kulla-Block by using $k - Storage\_Loc(in)$ and are processed to produce results $r\_x = k - Storage\_Loc(out)$, which are sent to either a *Data Sink* or another Kulla instance through another edge ($Kulla - Block\_j(out) \rightarrow Kulla - Block\_j[out] \rightarrow DataSink\_s$). This procedure is depicted in Figure 1.

3.2 *Kulla-Bricks*: Structures to build parallel patterns

At this point, a Kulla-Block has converted an application into an independent, interoperable and reusable piece of software. This means a Kulla-Block can be interconnected to other Kulla-Blocks through the I/O interfaces creating structures of directed graphs called *Kulla-Bricks*.

The chaining of Kulla instances in Kulla-Bricks (patterns) is managed in the form of directed graphs by using the notation defined for Kulla-Blocks structures as it is assumed that all nodes in a pattern are Kulla-Blocks.

In this model, we considered three basis patterns producing streaming/dataflow, task parallelism and data parallelism. The dataflow patterns are created by chaining applications in adjacent manner. These patterns are useful, for instance, to add value properties to contents. Task parallel patterns are built by chaining Kulla instances in parallel to improve the performance of a given application, which is quite useful when processing sets of contents. The Data parallel patterns are similar to the task parallel patterns but the tasks are focused on segments of data of a single content, which are processed in concurrent manner. This pattern is quite useful to process large length contents.

We designed and developed methods to create above patterns by using Kulla instances: *Pipe&Blocks* (streaming/dataflow), *Divide-and-Containerize* (data parallelism), and *Manager/Blocks* (task parallelism). These patterns are described in the folowing sections.

3.2.1 *Pipe&Blocks: A Kulla-Brick producing pipelines*

The grouping of Kulla-Blocks in adjacent fashion creates a Pipe&Filter patterns [2] for producing flows of data through

2. *Pipes and Filter* patterns are a sequence of f processing filters connected adjacently through communication interfaces (pipes) from a data source (DSr) to a data sink (DSk)

the applications encapusulated into the Kulla-Blocks considered in a Kulla-Brick. This type of pattern creates a continuous delivery of processed data to the filters (f) in the pipeline, where $f \geq 1$ is required [42], [43].

See this pattern build as a directed graph:

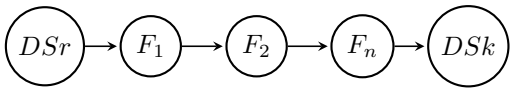

Fig. 3. Pipe & Filter pattern represented as directed graph

As it can be seen, in Pipe&Filters pattern the $filter$ represents a processing unit that receives an input data from a data source, applies a process (application), modifies the input data according to the performed process, and forwards either the modified data or a result to a data sink or another filter through a $pipe$ (edge), which represents a transport/communication interface.

In this pattern, each filter is independent and an autonomous piece of software. This means that filters can be added, modified, replaced or removed from one software pipeline without modifying other filters.

We defined a $Pipe\&Blocks$ method to create traditional Pipe&Filters by using Kulla instances. In this method, the Input and Output interfaces of Kulla-Blocks represent Pipes, whereas the Filters represent applications encapsulated into Kulla-Blocks. We developed a reserved Kulla instances such as $orchestrator$ and $launcher$, which were included in the Pipe&Block method to manage software pipelines. The $orchestrator$ filter is in charge of creating a big picture of the software pipeline solution in the form of a configuration files per each Kulla-Block included in this Brick by using the notation of a directed graph. This file includes information required to identify and deploy each Kulla-Block considered in the brick and the labels indicating the place of each Kulla-Block in the pipeline (any of $initiator$, $intermediary$ or $ending$). The orchestrator delivers the configuration files to the $launcher$ filter, which is in charge of deploying each Kulla-Block on the infrastructure and performing the chaining of these Kulla-Blocks in the form of a pipeline. The procedure performed by the launcher is described in Algorithm 22

The Input interface of Kulla-Block labeled as $beginning$ is linked to the Data Source and the Output to the Input of the first $intermediary$. The Input and Output interfaces of $intermediary$ Kulla-Blocks are linked to the interfaces of previous and next Kulla-Blocks to retrieve data and deliver the processed results respectively. The Output interface of Kulla-Block labeled as $ending$ is linked to the path for delivering the final results to a the Data Sink.

A reserved kulla instance called $Manager$ has been developed to establish controls over the exchange of data through the pipeline and the notifications of the ending of a processing task per each Kulla-Block as well as to inject workload to the $beginning$ Kulla-Block in the pipeline.

3.2.2 Divide-and-Containerize

We developed a method called $DivideandContainerize$[3] to create data parallel patterns by using Kulla-Blocks.

3. Inspired by traditional divide&conquer algorithm [44]

Algorithm 1 Pipe And Blocks Pattern.

Require: $NBinBrick \vee Brick-BlockIDMaps[] \vee Paths[]$
  $Source \leftarrow Paths[0]$
  $Sink \leftarrow Paths[1]$
  $Initiator \leftarrow BlockIDMaps[0]$
  $Kulla-Brick[0] \leftarrow Launch(Initiator)$
  $Previous \leftarrow Initiator$
  $NextBlock \leftarrow BlockIDMaps[i]$
  $Kulla-Brick[i] \leftarrow Launch(NextBlock)$
  $Kulla-Brick[0] \leftarrow Map(Initiator.InPar, Input = Source, Output = NextBlock.Input)$
  $Input = Previous.Output$
  while $NextBlock[i] \neq Ending$ **do**
    $Previous \leftarrow NextBlock[i]$
    $i++$
    $NextBlock \leftarrow BlockIDMaps[i]$
    $Kulla-Brick[i] = Launch(NextBlock)$
    $Kulla-Brick[i] \leftarrow Map(Previous.InPar, Input, Output = NextBlock[i].Input$
  end while
  if $NextBlock[i] == Ending$ **then**
    $Kulla-Brick[i] \leftarrow Map(Previous.InPar, Input, Output = Sink$
    $EXIT$
  else
    $ERROR(EndingPipeline)$
  end if

This $Divide-and-Containerize$ method considers a Kulla-Brick encapsulating a Pipe&Block pattern deploying three types of Blocks: $Divide$, $Containerize$, $Worker$ and $Conquer$, whereas reuses the $Orcherstator$, $Manager$ and $Launcher$ for the building of the Kulla-Brick.

The $orchestrator$ filter creates a big picture of the Brick in the form of a configuration files per each Kulla-Block in this Brick by using the notation of a directed graph. This pattern can be represented by the following pattern:

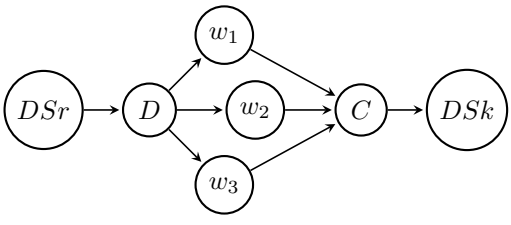

Fig. 4. Divide and Containerize pattern represented as directed graph

The configuration file includes information required to identify and deploy each Kulla-Block considered in a directed graph. In this configuration file, the $Orchestrator$ identifies the Kulla-Block that will represent the $Worker$ in this pattern, which one will represent $Divide$, $Containerize$, and $Conquer$. The configuration file includes the information of the I/O interfaces that will be used to perform the chaining of all this components in a pattern and delivers this file to the $Launcher$ filter, which create as many clones of the worker Kulla-Block as established the configuration file and launches the $Divide$, $Containerize$, and $Conquer$ software instances. The $Containerize$ in-

stance performs the chaining of the Kulla instances with the worker Kulla-Boxes by following the directed graph notation included in the configuration file.

When the pattern has been deployed on a given infrastructure, Divide (D) first retrieves data content-by-content from its data source (DSr) through the $Input$ interface by using the I/O interface defined in configuration file to do that (either network or file system I/O functions) and write the content in the $K - Storage$. The second step performed by Divide D is virtually to split each content stored in *K-Storage* into s segments for creating a list of K-Storage pointers (beginning and ending of each memory segment), which are sent to the Worker Kulla-Blocks.

The segmentation process is described in Algorithm 2.

Algorithm 2 Segmentation Process (Divide).

Require: Content name C_{name},Data Content $|C|$, number of segments s, output list $outs = \{out_1, out_2, \ldots, out_s\}$

$currentSegment = 1$, $segmentsSent = 0$
while $currentSegment \leq s$ **do**
  $CreateThread()$
  $residue = |C| \bmod s$
  if ($residue == 0$) **then**
    $segmentSize = |C|/s$
  else
    if ($currentSegment \leq residue$) **then**
      $segmentSize = |C|/s + 1$
    else
      $segmentSize = |C|/s$
    end if
  end if
  if ($currentSegment > 1$) **then**
    $initialPosition = segmentSize * (currentSegment - 1)$
  else
    $initialPosition = 0$
  end if
  $segmentContent = ReadSegment(|C|, initialPosition, segmentSize)$
  if ($SendSegment(segmenContent, out_{currentSegment})$) **then**
    $sendedSegments+ = 1$
  end if
  $KillThread()$
end while
if ($segmentsSent == s$) **then**
  return $GenerateContentId(C_{name})$
else
  return -1
end if

As depicted in Algorithm 2, the workers uses the K-Storage pointers for applications encapsulated into the Kulla-Blocks to process the segments and puts the data in the K-Storage for $Conquer$ instance to consolidate results.

The $Conquer$ receives from workers the $K - Storage$ pointers, which are used to get the results and to consolidate the results into one single output (depending on the action suggested in each case). it also could deliver the results to any of a data sink (DSk), other filter, or another Kulla-Block/Brick depending on the configuration established in the construction of $Divide - Containerize$ Kulla-Brick.

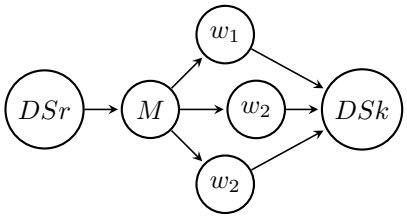

Fig. 5. Manager/Block (M/B) example

The integration process used by Conquer is described in Algorithm 3.

Algorithm 3 Integration process (Containerize).

Require: Content Id F_{id}, File Size $|F_{size}|$, number of segments s, input list $input = \{input_1, input_2, \ldots, input_s\}$, output path out

$i = 0$, $j = 0$, $recoveredSegments = [s]$
while $i \leq s$ **do**
  $CreateThread()$
  $recoveredSegments[i] = RecoverSegment(i, F_{id})$
  $KillThread()$
  $i+ = 1$
end while
$recoveredData = ReserveMemory(F_{size})$
while $j < s$ **do**
  $recoveredData = IntegrateContent(recoveredData, recoveredSegments(j), j)$
end while
$sendContent(recoveredData, F_{id}, out)$

As it can be seen, the results produced by a $Divide - and - Containerize$ Kulla-Brick are the very results produced by the application executed by a worker. Of course, the performance of $Divide - and - Containerize$ Kulla-Brick is better that the original application as it exploits data parallelism, which is achieved without altering the code of its routines and without changing the way in which the application is invoked for execution.

The Divide, Containerize, Conquer and Worker Kulla instances are in charge of invoking the functions and controlling the synchronization of access to $K - Storage$ in a transparent procedure; as a result, the end-users are only required for encapsulating the application into a Kulla-Block to be executed as worker in this pattern. It is assumed that the application does not produce dependencies and that the data segmentation is suitable for processing data when using this pattern.

3.2.3 Manager/Block*: M/B Kulla-Brick*

Manager/Block (M/B) method creates manager/worker patterns inside of a Kulla-Brick depicted in the following directed graph:

As it can be seen, the M/B method includes a $Manager$ instance that reuses $Orchestrator$ and $Launcher$ instances.

The pattern also includes a $Worker$ instance, which follows the very model ETL (Extract, Transform and Load) described in previous method: the worker receives a task (a content to be processed) as input parameter, executes a given application, which transforms that content and loads

the output results (transformed version of that content) to the K-Storage where other filter, Kulla-Block, or Brick can retrieve them to start another processing procedure.

The Manager performs the following steps: i) Launches w clones of the worker instance by reusing Launcher software instance. ii) Creates collections of tasks to process the contents stored in a Data source and creates maps that include the path of a given content in the data source, an identifier of the task to be processed, the parameters for the application invoked by the workers, and the path of either the data sink or another Kulla instance, which will be used by the worker to deliver the results. iii) Reads contents from data source by using the information registered in the tasks and puts these contents in the $K - Sotrage$ iv) Distributes the maps to the workers by using a load balancing algorithm ([45]). This algorithm assigns maps to the workers through an I/O interface and enables the manager to locate the results produced by the workers. vi) Recovers the messages returned by workers when finishing a given task, assigns new tasks to the workers and keeps control over the execution of the collection tasks.

Algorithm **??** describes the construction and the dataflow produced by this pattern.

Again, the Manager and Worker Kulla instances are in charge of invoking/managing the functions for data exchange, synchronization, and for the access to $K - Storage$, data source and data Sink in a transparent manner.

3.2.4 Recursive utilization of kulla instances and combined patterns

In kulla construction model, patterns based on complex graph systems can be built by using recursively patterns and/or by combining different types/numbers of *Kulla-Bricks* and to present these solutions to the endu-users as a single solution.

The following directed graph depicts a combined pattern including a Pipe&Block pattern, which includes in the first filter a Manager (M) of a M/B pattern. In This pattern, each worker (w_1, w_2, w_n) launches a divide-and-Containerize pattern (D/C_1, D/C_2, D/C_n).

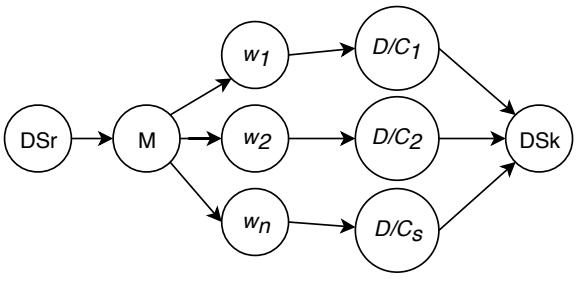

Fig. 6. A directed graph that exemplified recursive utilization of kulla instances and combined patterns

In a similar way to this directed graph, solutions can be built in a recursive manner. For instance, each worker of the M/B pattern can launch another M/B pattern and so on.

These Brick structures enable developers to add either parallelism or concurrent processing to their applications to improve the efficiency of solutions without altering/modifying routines in the codification of the applications. Examples of the development of this type of solutions based on directed graphs are described in the experimental evaluation section.

4 KULLA-BOXES: DEPLOYMENT STRUCTURES FOR THE COMPOSITION OF AGNOSTIC APPLICATIONS

The basic development structure in Kulla model is a software piece called *Kulla-Box*, which includes software instances to implement Kulla-Blocks and/or Kulla-Bricks. This software piece is encapsulated into a virtual container for deploying kulla solutions on a given infrastructure, which is presented to the end-users as a single application.

Kulla-Boxes were designed for end-users to avoid side-effects from data and infrastructure dependencies that arise in deployment time, which significantly reduces the time required by IT troubleshooting.

Therefore, a Kulla-Box converts applications into agnostic solutions, that are already to be executed and deployed by the end-users on a given infrastructure with container platform previously installed.

It is important to note that, all the solutions created in Kulla model reuse both the I/O interfaces and control software instances, which adds self-similarity to the Kulla-instances. For instance, a Kulla-Box and a Kulla-Brick are auto-similar to the smallest part (a Kulla-Block) as all the instances encapsulated into the virtual container of a Kulla-Box (any of filters, Kulla-Blocks or Kulla-Bricks) can get access to the very three I/O interfaces included in the Kulla-Blocks. This means a Kulla-Box ends up following the very ETL model used by Kulla-Blocks to process contents. For instance, a Kulla-Box could only to encapsulate one Kulla-Block, which cloud include a filer or a brick of filters (See a Kulla-Box encapsulating a Kulla-Block, which including patterns of filters in Figure 7). It also could includes a Kulla-Brick or a combination pf bricks, which also could includes a set of Kulla-Blocks (See a Kulla-Box encapsulating a Kulla-Brick in Figure 7) and so on.

4.1 Building agnostic distributed applications by using Bricks of Kulla-Boxes

Kulla-Box structures also can be converted into reusable and interopreable software pieces because of, as we already said, a $Kulla - Box$ follows the ETL model defined for all Kulla instances. This means that each Kulla-Box also retrieves contents from data source that are processed by an kulla solution that is executed in the virtual container as a single application, and delivers results to a Data Sink.

This self-similarity property adds interoprability to the Kulla-Boxes, which enables end-users/developers to chain Kulla-Boxes to another Kulla-Boxes to create a Bricks of Kulla-Boxes (See graph c in Figure 7).

This type of structures also can be represented in the form of a directed graph by using the very model created for Kulla-Blocks and Kulla-Bricks that have been previously described in this paper. This also enables end-users/Developers to create a distributed agnostic application when Bricks of Kulla-Boxes are deployed on a given distributed infrastructure (clusters, clouds, etc).

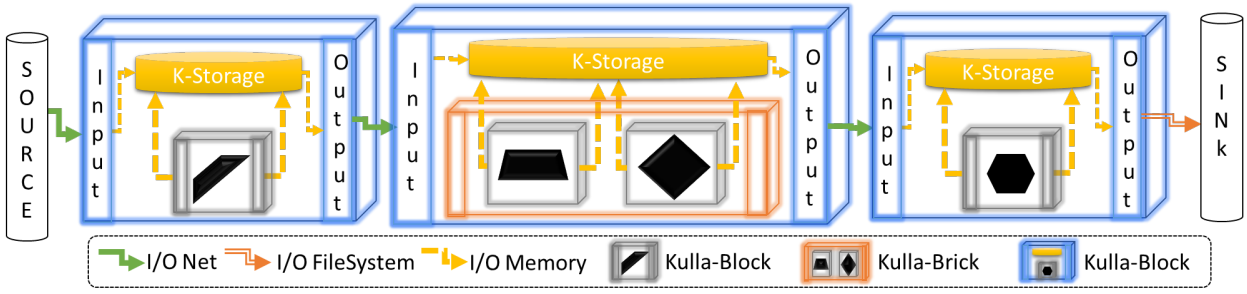

Fig. 7. Representation of Kulla-Boxes abstraction

4.1.1 Deployment scheme and resource profitability strategies

In order end-users/developers to create agnostic distributed applications in felxible and dynamic manners, we developed and implemented a deployment scheme for Kulla-Boxes based on strategies including preparation and launching phases.

In the preparation phase, configuration files for the Kulla-Boxes are created by following the notation of a directed graph. These files includes information such as list of all kulla-Box images considered in a Brick of Kulla-Boxes, the type of patterns considered in each Brick, the input parameters for each Brick (e.g. the number of workers in a manager/worker pattern). The role of each Kulla-Box in each pattern is also included in this file. For instance, in a Manager/Block pattern, a Kulla-Box image should represent a manager and another one a worker. This means, the graph will indicate which Kulla-Box image will be used to launch a Kulla-Box Manager and which image will be used to launch a Kulla-Box worker as well as the number of clones of this image should be launched and deployed on a infrastructure.

In order to implement the launching phase of the deployment scheme, we reused control kulla instances such as $Orchestrator$ and $Launcher$, which were previously defined when describing the management of patterns Kulla-Bricks.

The $Orchestrator$ of Kulla-Boxes is a software instance placed in the first Kulla-Box of a Brick of Kulla-Boxes and it is in charge of creating "solution layouts" for each Brick of Kulla-Boxes by following a directed graph configuration file created in the preparation phase.

Based on this "solution layout" the Orchestrator creates a configuration files for each Kulla-Image where is indicating information such as the ports (an IP when it required) of Input and Output interfaces of the Kulla-Boxes, the paths of the volumes that will be used as Data Sources and Data Sinks for each case, the resources assigned per each Kulla-Box (number of cores, RAM). In the case of this information has not been provided by the developer of the distributed application, the Orchestrator can get it from a virtual containers manager (e.g. either Kubernetes or Docker Swarm that was the one used by our Orchestrator).

The $Launcher$ is included in each kulla-Box image considered in the solution layout.

This enables the Orchestrator to invoke the launchers to deploy the Kulla images considered by the distributed application (Brick of Kulla-Boxes) in the form of instances of virtual containers on a given distributed infrastructure (e.g. Cluster or a Cloud). This also means that the scheme will clone the Kulla-Box worker images w times to launch w Kulla-Box instances.

The launchers add to the Kulla-Boxes a deployment configuration file sent by the Orchestrator for each Kulla-Box deploys its kulla instances required by that Kulla-Box (e.g. managers, filters, workers, divide, conquers, containerize, etc). In this phase, the scheme assumes the images of Kulla-Boxes considered in a directed graph have already created and are available to be launched in a given infrastructure.

The deployment of a solution layout is finished when the automatic chaining of all the Kulla-Boxes has been fished in the launching phase. At this point, a single agnostic solution is available to be deployed on different types of infrastructures by using a resource profitability strategy such as scale-in, scale-out and both (mixing scale-in and scale-out).

Scale-In is a strategy that enables the developers to compose applications by deploying Kulla-Bricks into a single Kulla-Box. In this strategy, all the Kulla-Blocks/Bricks are launched trying to use all the cores in that server to create agnostic parallel applications. This is feasible because developers can bind a given Kulla-Block with a given core. This deployment strategy considers the creation of $K - Storage$ in memory, which is used as a communication channel for the data exchange among the Filters, Blocks, and Bricks considered in a given Kulla-Box.This strategy is suitable to deploy parallel agnostic applications on any of virtual machines, servers or computers including several cores.

Scale-Out: This strategy enables the developers to compose applications by deploying Bricks of Kulla-Boxes on clusters of either computers or cloud virtual machines. In this strategy, a K-Storage is created per Kulla-Box in the Brick and the data exchange is performed through the network interfaces (Sockets for clusters and Curl for Cloud). This strategy is suitable to deploy agnostic distributed applications.

Scale-Out/In: This greedy strategy enables developers to create workflows of agnostic distributed and parallel applications by mixing the aforementioned policies. In this strategy, Bricks of Kulla-Boxes are deployed on different types of infrastructures and each Kulla-Box deploys parallel patterns (Kulla-Bricks), which also deploys Kulla-Boxes trying to use as many resources (i.e. cores) as available in a infrastructure.

4.2 Implementation of solutions based on Kulla model

In this section, we describe implementation details about the development of construction, processing and deployment structures and schemes considered in the design section. We first establish that all the Kulla instances were developed by using virtual container images created with Docker platform. The control kulla filters 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) were available in Java, Python and C++.

All the software instances and Kulla-Boxes were stored in a service we called Kull-Silo.

4.2.1 Kulla-Silo: Repository of Kulla-Boxes, I/O libraries and control instance software

In order to simplify the implementation of solutions based on Kulla model, we develop a repository service to manage I/O libraries, software instance (versions of the libraries in the form of services), control kulla instances (reserved) and Kulla-Boxes images already created by end-users/developers.

In the $Kulla - Silo$ repository, the Kulla instances are classified as user-defined and reserved. Kulla-Boxes created by end-users or developers are examples of user-defined Kulla instances, whereas filters such as Managers, Orchestrators, Launchers, Divide, Containerize, Conquer and Workers as well as the implementation of I/O libraries such as K-Storage, Input and Output interfaces are considered as reserved. These instances are available in virtual container images that can be used to build Kulla-Box instances.

The Kulla-Silo service includes functions to Put, Get, $Update$, $List$ and $Delete$ Kulla-Boxes images as well as functions to ADD, $REMOVE$, $CHANGE$ reserved kulla software instances to/from a given Kulla-Box image. This means the end-users can choose from existent and available Kulla-Blocks and/or Kulla-Bricks in the Silo by using above functions to create user-defined Kulla-Boxes. The end-users also can create Bricks of Kull-boxes by choosing the Kulla-Boxes from the Silo and by configuring the type of pattern to be created and the roles the Kulla-Boxes in that patterns (Brick).

The input parameters for Kulla-Silo service are locations maps, which include information such as image identifier (Id-Block), type of image (Id-Type), owner identifier (Id-Owner) and the "father" of the image (to identify bricks and boxes), inventory of reserved Kulla instances to be included into a Kulla-Box image.

Kulla-Silo service was implemented as a pair of containers: one including a PostgreSQL database for the indexing of Kulla images and another including an instance of a cloud storage service called SkyCDS [46] for storing the images indexed in the Silo.

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 in the form of filters and organize them as Kulla-Blocks and/or Kulla-Bricks.

The following libraries are available in templates considered in this repository:

- The management library includes functions for Kulla instances to create, launch, list and delete filters 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).
- The shared memory management library enables Kulla-Blocks to implement a K-Storage. This library includes functions calls for filters to PUT/GET data to/from K-Storage and to keep monitoring the K-Storage as well as the management of the retrievals and deliveries requested by applications. This library was developed in C++ IPC [47]. Functions written in Python, C, and C++ for applications to access to K-Storage were also included in this library.
- The I/O library has been developed for filters to access Input and Output interfaces of the Kulla-Blocks. This library includes sockets and curl I/O functions for Kulla-Blocks to establish communication with other Kulla instances through the Network and/or File System. This library enables Kulla-Blocks to receive/deliver data to/from either a data sink/source.

4.3 Prototyping

We developed filters for the management of a set of applications to create Kulla-Blocks, which were developed a set of Kulla-Boxes based on Kulla-Bricks such as Divide-and-Containerize, Pipe&Filters and Manager/Blocks methods as well as Bricks of Kulla-Boxes by combining Kulla-Boxes including different types of patterns (Kulla-Bricks).

The Kulla-Boxes were indexed in he Kulla-Silo, which was installed in a cluster of computers by using the Docker platform (Compose). The kulla-Boxes images designed for the evaluation defined in this paper, which including either Bricks or Blocks. These images were added and indexed in the Kulla-Silo using Docker Swarm for creating distributed clusters of virtual containers (for the deployment of Bricks of Kulla-Boxes).

The Kulla-Boxes were extracted from the Kulla-Siilo and were deployed on the infrastructure described in Table 1 by following a given resource profitability and deployment strategy (scale In and Scale Out/In).

5 Experimental Evaluation and Results

The assessment and evaluation of kulla model was conducted through an experimental evaluation in the form of study cases based on real-world application composition based on Kulla instances. A methodology of two phases was defined to perform this experimental evaluation. In the first one the solutions were deployed by using *Scale-In* strategy, whereas in the last one *Scale-In/Out* was used.

We developed a Kulla-Block called $Client$ that includes a $workload$ producer bot in a filter for sending requests to the Kulla instances. These instances assume that the workload generated by the bot is valid, as far as valid credentials of real end-users are provided. This $Client$ Kulla-Block includes a filter for capturing the metrics used in the experimental evaluation.

In the experimental evaluation of the case studies we used the following metrics:

- *Service Time (ST)*: This metric represents the time elapsed in which a content is processed by the application(s) encapsulated in a Kulla-Box.
- *Response Time (RT)*: This metric represents the time spent by a Kulla-Box solution to successfully dispatch requests sent by the client bot. This metric represents the sum of the ST produced by each Kulla instance considered in a solution.
- *Percentage of performance gain*: Represents the resultant percentage of gain when comparing response times produced by different solutions/configurations.

TABLE 1
Infrastructure used in experimental evaluation scenarios.

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

.

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

5.1 Scenario Scale-In: Experiments and Results

In this phase of evaluation, the Kulla-Boxes including Kulla-Bricks such as Divide-and-Containerize, Pipe&Blocks and Manager/Blocks were created by using real-world applications. These Kulla-Boxes were deployed on a PC and a single server by using the *Scale-In* strategy. The infrastructure used on each case is described in the case study.

In this scenario, we specifically created three Kulla-Boxes.

The first Kulla-Box only including a Divide-and-Containerize Kulla-Brick adding reliability value property to the contents before sending them to the cloud or sharing them with other users/partners.

The second Kulla-Box included a two stage Pipe&Block Kulla-Brick where in the first stage is a Kulla-Block adding frugality property to the contents (a compression application was encapsulated into this Kulla-Block), whereas the Divide-and-Containerize Kulla-Brick used in the first solution was placed in in the second stage of the pattern encapsulated into this kulla-Box.

In the last Kulla-Box, a Manager/Block Kull-Brick was added to the bricks created in the second solution. In this Brick, each worker executed a Pipe&Block of two stages: in the first one, a compression application was executed and a Divide-and-Containerize Brick was executed in the second stage.

5.2 Divide-and-Containerize Kulla-Box

The Divide, Containerize and Conquer software instances of the Divide-and-Containerize method were encapsulated each into a Kulla-Block. The Containerize Kulla-Block was configured to launch worker Kulla-Blocks in an incremental manner (one-by-one from 1 to 5).

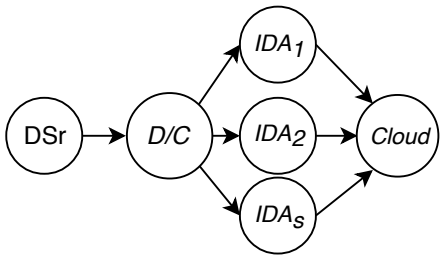

Fig. 8. First evaluated Kulla Solution represented as directed graph

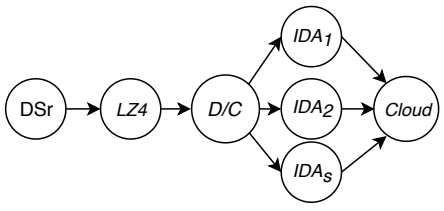

Fig. 9. Second evaluated Kulla Solution represented as directed graph

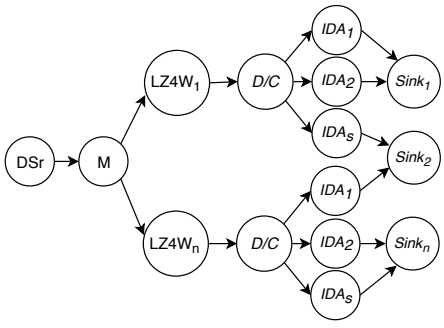

Fig. 10. Third evaluated Kulla Solution represented as directed graph

The worker Kulla-Block image included the implementation of a dispersal information algorithm (IDA) [**?**], [48] based on a serial/sequential implementation of the IDA algorithm [48] available in the literature [**?**], [49], [50], [51].

The IDA algorithm provides contents with reliability features to withstand failures of data (data missing, data bit errors, unavailability of data, etc). This algorithm considers a fault-tolerant technique that splits each content $|C|$ of length L into n pieces called 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 $n * (L_C/m)$, which means the overhead is $Ov = (n * L_{dfs}) - L_C$.

In order to understand the benefits and limitations of the implementation of this algorithm in straightforward manner, let us to consider a content of length L_C = 1MB processed by an IDA configuration n = 5 and m = 3. In such a configuration, IDA splits that content $|C|$ into five segments dfs of length $L_{dfs} = (L_C = 1MB/m = 3)$=.33MB

where three of them ($m = 3$) sufficient to reconstruct $|C|$. In this example, the capacity overhead is $Ov = ((n = 5 * L_{dfs} = .33MB) - L_C = IMB) = .667MB$, which results in 66% of extra capacity (less than one replica of the source data) for the system to withstand the failure/unavailability of two $(n-m)$ missing dfs. As it can be seen, in practice, the

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

5.2.1 Configurations and evaluated solutions

The following configurations of Kulla-Brick and related solutions were evaluated in the experimental evaluation:

- Serial IDA ($IDA - S$): This configuration executes a serial implementation of IDA (implemented in C) [50].
- Parallel IDA ($TBB - IDA$): This configuration represents a parallel version of IDA algorithm [46] developed by using Intel TBB framework [21], 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 and CURL libraries).
- $Kulla - IDA$: In this configuration, the Divide-and-Containerize method was developed to execute IDA serial without modifying the routines of the 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).

5.2.2 Divide-and-Containerize Kulla-Box result analysis

In this section, we present the results of the experiments performed with the configurations and solutions tested in this case study.

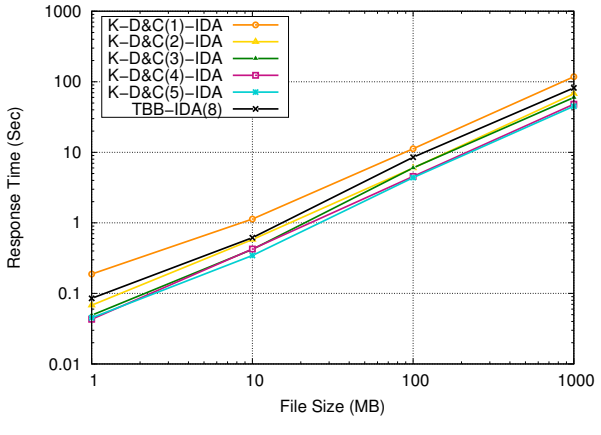

Fig. 11. Encoding response time of studied configurations

Figure 11 shows the response time produced by the solutions evaluated in the controlled experimentation for different file sizes.

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

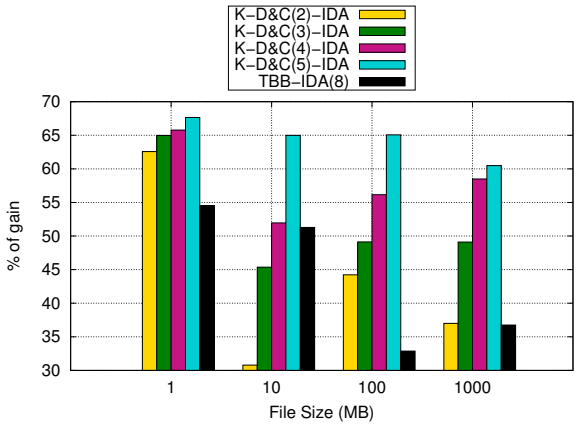

Fig. 12. Gain performance percentage of studied configurations in comparison with Serial solution.

Figure 11 also shows the Divide-and-Containerize method encapsulated into a Kulla-Brick launched with different configurations depending on the number of workers (from 1 to 5 workers) invoked by Containerize block (Kulla-IDA2-5). Kulla-IDA2 configuration is not competitive in comparison with TBB-IDA configuration for small files, which is an expected behavior considering that Kulla-IDA (based on Divide&Containerize method) configurations are focused on data parallelism. As it expected, Kulla-IDA configuration is not quite efficient with few workers and small tasks. This premise is evident when Kulla-IDA2 becoming competitive for large files in comparison with TBB-IDA configuration. It is also evident when increasing the number of the worker instances in the Kulla-Brick (Kulla-IDA3,4,5).

Two causes produce the improvement effect observed in the response time produced by Kulla-IDA configurations: the first one is the size of the tasks (*Task Size=FileSize/Number of workers*) managed by each worker, which results in that the more the workers, the less the size to be processed by the applications of the workers. The second one is the in-memory exchange of information only produces two I/O operations sent to the file system (Read from the data source and write to the Data Sink), which also reduces the service times produced by the solution.

The impact of the evaluated solutions on the performance of the algorithm is showed in Figure 12 where the percentage of gain obtained by the solutions evaluated produced for different file sizes is presented. The improvement of TBB-IDA decreases form 54% (for 1MB files) to 33% in comparison with the serial version (IDA-S), whereas Kulla-IDA-5 produces an improvement from 68% to 60%, Moreover, Kulla-IDA-5 produces a performance improvement, depending on the file size to be processed, between 14% and 27% in comparison with TBB-IDA.

In order to understand the effects, not only of the data parallelism, but also of the in-memory management on the performance for the evaluated solutions, we added in-memory management to the TBB-IDA solution.

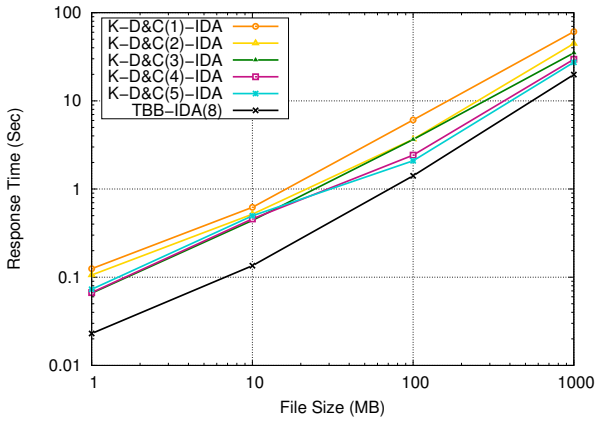

Fig. 13. Response time obtained for the three configurations with all their versions.

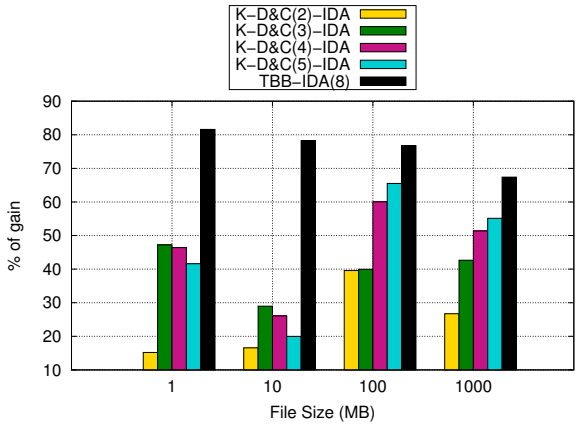

Fig. 14. Gain obtained by the K-D&C-IDA and TBB-IDA to IDA Pipeline.

Figure 13 shows the response time obtained by the Kulla-Box configurations (Kulla-IDA2-5) and TBB-IDA in the decoding process for contents of different sizes. Figure 14) 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-Boxes.

The behavior showed in Figure 13 is produced by two causes: the first (and the obvious one) is the in-memory management of the I/O calls added to TBB-IDA solution, which reduces the calls performed by this solution to the file system, which enables this solution to process data faster than in the original version. The second one is the reduction of the data processed in the decoding procedure (only three $df\,s$ are processed, whereas five $df\,s$ are processed in encoding procedure). For instance, when processing an file of 1GB, in a encoding procedure are managed 1,66 GB because of the 66% of redundancy produced by the IDA algorithm, whereas in the decoding process only 1GB is processed; as a result, the workers receives less data to process in a decoding process than in encoding procedure. As already we said, the data parallelism is more suitable for processing large files, which explains the improvement of Kulla solutions for large file sizes in comparison with TBB-IDA.

The results of this scenario revealed that, when the developers have enough experience to identify the routines suitable to be executed in parallel and to develop shared memory functions for these routines, they can create solutions producing a better performance than the performance produced by Kulla scheme based on patterns deployed on virtual containers and in-memory processing/storing.

However, when this is not the case, Kulla represents a good deal for developers requiring dynamic, rapid and efficient solutions, as the management of data parallelism processing and in-memory storage is performed by Kulla transparently. Moreover, the construction of the solutions is almost immediate as the parallel pattern is built in advance and the developer/user only requires to incorporate an application to a worker and choosing the number of workers to be launched for obtaining a parallel solution delivering a competitive performance.

A third option is encapsulating a parallel application into Kulla-Box to add, in a rapid manner, in-memory processing/storing feature to this application (We performed this in TBB-IDA configuration showed in Figure 13). In this case, the in-memory processing is transparent for developers/end-users as this process is managed inside of the Kulla-Box.

5.3 Pipe&Block Kulla-Box processing satellite imagery

As aforementioned, Pipe&Block Kulla-Box is a solution based on a two stage Pipe&Filter Brick: In the first stage, we encapsulated the *LZ4 Lossless compression algorithm (lZ4)* [4] into a Kulla-Block, which reads a given file as input and produces, as output, a compressed/decompressed version of original file, which is sent forward to the next stage in the pipeline. The second stage of this pattern is the very Divide-and-Containerize Brick evaluated in previous section. This Kulla-Box is represented by the *Kulla-Lz4+IDA* configuration. We reused the parallel configuration (TBB-IDA) and a software pipeline (LZ4-IDA) created by using the serial configuration previously evaluated (Serial-IDA)

This kulla-Box allowed us to show an interesting feature of 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.

In this context, we conducted a case study based on the processing of satellite imagery repository by using that Kulla-Box. This repository includes the catalogs per each sensor (LandSat, Terra and AQUA) captured by an antenna 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 with one processing level) and must be preserved as heritage for large periods of time as these images are used in the creation of earth observation products. Each image is processed 31 times by the studied solutions and the median of the metrics is captured for corresponding evaluation.

4. Implemented by using the libraries described in [52]

5.3.1 Pipe&Block Kulla-Box experimental results

The Kulla-Box (*Kulla-Lz4+IDA*) adds frugality and reliability properties to the satellite images in a combined manner. The aim of this type of properties combination is to improve the storage utilization, is a quite interesting feature, for instance, for missions of earth observation. 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) 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 improvement of the storage utilization produced by *Kulla-Lz4+IDA* also improves the service time of the Kulla-Box because the contents are first compressed before to be sent to the Divide&Containerize Brick of IDA. This means the workers of IDA pattern receives less data to the encoding/decoding procedures, which also reduces the response time of the *Kulla-Lz4+IDA*.

Figure 15 depicts this effect when showing, in left vertical axis (left), the capacity produced when applying the IDA fault-tolerant technique to the satelltie images with (See *Capacity LZ4-IDA* bars) and without (See *Capacity IDA* bars) compression produced by *Kulla-IDA* when coding satellite images of different size (horizontal axis). The costs of the redundancy can be easily observed by comparing the capacity produced by both configurations. As it can be seen, the reliability costs can be significantly reduced when combining encoding with compression depending on the compression degree achieved by the first stage of this pattern. For instance, the encoding and decoding response times for last image where the compression only could reduce the size of the file in 15% producing, which also affecting the next stage producing a increasing of redundancy overhead of 18%, whereas the 40% of reduction was obtained for the rest of images, which even reducing the costs of reliability to zero.

Figure 15 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 *K-Lz4-D/C(6w)* and *K-D/C(6w)*).

As it can be seen, it is evident that the combination of compression with encoding/decoding not only reduces the the capacity to be processed but also the response time is reduced in such a process. Although an increment in the response times is expected by adding compression to the original solution devoted to encode/decode satellite images, the combination of features produced in *Kulla-Lz4+IDA* not only avoids this increment but also produces a reduction in the times (20%).

Insights from the *Kulla-Lz4+IDA* comparison with Kulla-IDA(6 cores) showing the efficacy of combining solutions to reduce not only the storage utilization but also the processing performance, encouraged us to perform a comparison between the performance of *Kulla-Lz4+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 be close to reach the performance of a routine paralleled solution and in-memory solution (TBB-IDA-Mem).

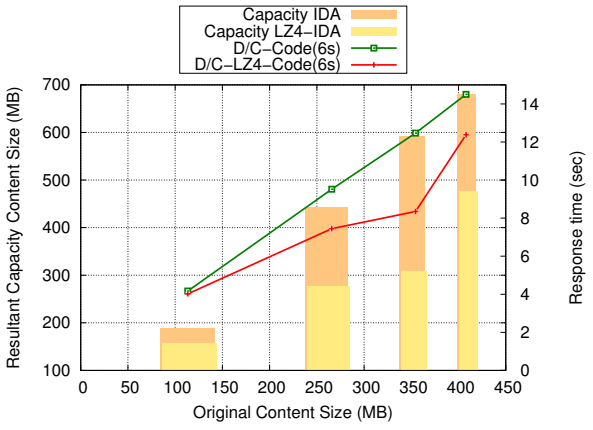

Fig. 15. Resultant capacity space produced by evaluated solutions.

Figure 16 shows, in vertical axis, the response times produced by *Kulla-Lz4+IDA* and *TBB-IDA* when encoding/decoding satellite images by using the studied configurations. As it can be seen, the more the workers, the less the size of data processed by the workers of Kulla configurations, which yields the higher improvement of the response times of these configurations. When *Kulla-Lz4+IDA* processing large satellite images, it is possible for this configuration to reach the performance of *TBB-IDA*.

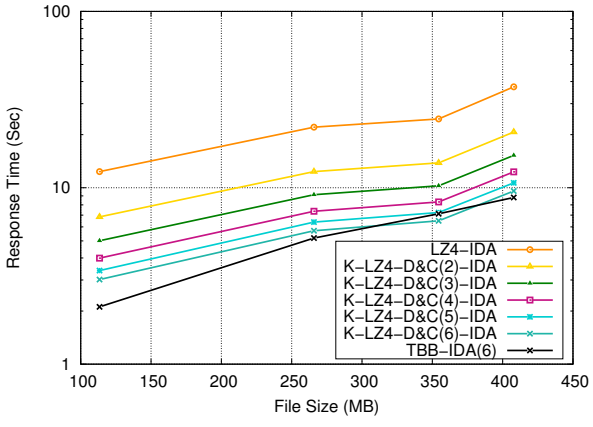

Fig. 16. Response time when increasing the number of workers Kulla-Blocks in the codification process.

5.4 Combining all patterns in a Kulla-Box for processing medical images

At this point, we have showed how a serial applications can be competitive in comparison with routine-based parallel applications by using parallel patterns built by using Kulla model. Furthermore, the evaluation also showed how the self-similar and modular properties of this model enabled us 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.

In this context, we add a new pattern to the previous Kulla solution by converting the LZ4 Kulla-Block into a Kulla-Brick implementing a Manger/Block pattern (M/B).

We encapsulated this solution into a Kulla-Box. See this Kulla-Box in Figure 17. The master of M/B Kulla-Brick reads files from the data source and sends one file to one worker, which launches a D&C pattern. This pattern splits the received file into 5 segments that are sent to five workers, which perform IDA encoding process on each segment.

As it can be seen, the basic idea is to reduce the awaiting time of D/C-IDA Kulla-Brick. The performance of this Kulla-Box was comapred with the performance of *TBB-IDA*.

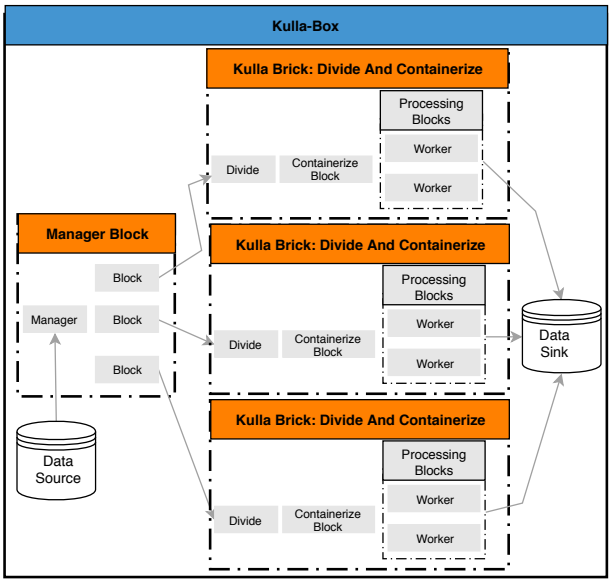

Fig. 17. Kulla-Box combining Manager/Block and Divide-and-Containerize patterns.

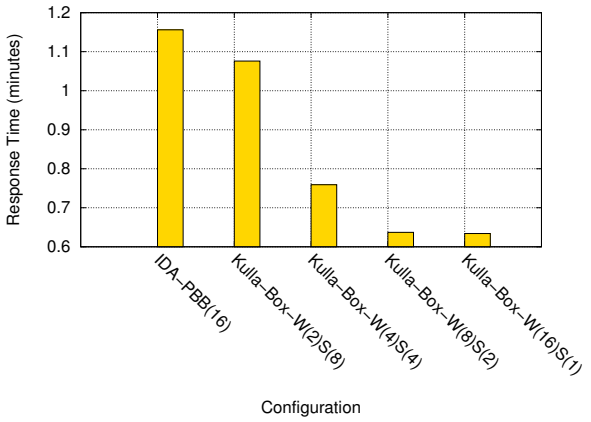

Fig. 18. Response time produced by TBB-IDA and different Kulla-Box configurations when using 16 physical cores

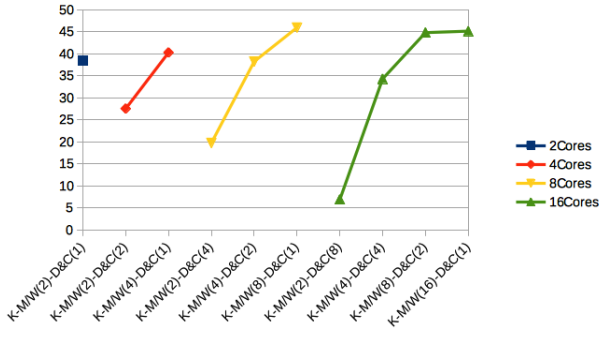

Fig. 19. % of performance gain

We conducted a case study based on computed tomography (CT) imaging repository where both solutions processed 55 images with quality (512x512), of 512MB each, of a crocodile produced by a tomograph. The medical imagery, with a volume of 27,5GB, was processed by the evaluated solutions.

In the case of the Kulla-Box, the experiments 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 ($D\&C$). In the case of *TBB-IDA*, the number of cores used by this configuration was increased from 1 to 16.

5.4.1 Experimental results when processing medical images

Figure 18 shows, in vertical axis, the response time produced by all the configurations of Kulla-Boxes (*k-M/B(x)-D&C(x)* and TBB-IDA(horizontal axis) when processing sets of medical images. To make a fair comparison, the configurations in this experiment to take advantage of all the available cores of the server used in the experimation.

As it can be seen, all configurations of $K - Box(w)$ produce better response time than *TBB-IDA* when using all available cores in the server. 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 improvement of the kulla solutions in comparison with TBB-IDA is exponentially increased from 6,88% when using two workers in the M/B pattern to 45,99% when using 16 workers in M/B per only one $D\&C$ pattern. Three are the causes of this effect of improvement in the performance observed in these experiments when processing medical images: i) the first stage (compression by LZ4) is performed in parallel, which reduces the awaiting time in the next stage (IDA). ii) LZ4 reduces the size of the data delivered to IDA stage, which reduces the service time of required by IDA in this stage to process the images. iii) Although the increment of D/C patterns increases the data parallelism, it also increases the I/O operations required to store the processed data, which also reduces the effectiveness of data parallelism.

In order to quantify the performance differences between both models, the client bot also captured the percentage of performance gain achieved by each Kulla-Box configurations. Figure 19 shows the percentage of performance gain of Kulla Boxes in comparison with *TBB-IDA*. 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)).

As showed in previous experiments, besides of performance improvement, $Kulla - Box$ configurations also reduce the costs of withstanding failures of services. In these solutions the combination of patterns encapsulated on kulla-Boxes not only reduces the storage utilization but also

improves the performance of the processing procedures, which was achieved without analyzing the code of the applications to find routines to be executed in parallel.

5.5 Scale Out/In Scenario: deploying agnostic distributed applications

In order to show the flexibility of Kulla model to provide applications with agnostic property in distributed environments, we deployed the kulla-Box combining patterns described in previous experiments in Figure 17 but now by using four servers and changing the scale-in deployment method by Scale-out/in. This deployment builds a distributed agnostic application that we called *Brick-Of-Kulla-Boxes*. In this type of deployment, a M/W pattern implementing a Pipe&Blocks encapsulated into a Kulla-Box is deployed on one server. The first stage of this Kulla-Brick included LZ4 and a launcher of three D&C patterns, which were encapsulated into another Kulla Box, which was cloned three times by the manager of the Brick of Kulla-Boxes. The Kulla-Boxes were deployed on one different server. Each D&C pattern included 1 master and as many workers as cores available in the servers.

The exchange of data between M/W and D&C Kulla-Boxes was performed through the network I/O interface (sockets chosen as all servers are placed at the same site). As a result of this deployment, the *BrickOf-Kulla-Boxes* solution used a M/B(3w),D&C(12w) configuration. Nevertheless, 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.

TBB-IDA was configured for encoding the medical images in a server by using all the cores in one server and then sending the encoding segments to the rest of servers.

Table XXX shows the features of the servers, which were installed in the form of a clusters of virtual containers.

We performed the very experiment described in previous section but now the 55 medical images were processed by *BrickOf-Kulla-Boxes* and *TBB-IDA* configurations. We compared the response time produced by both solutions to perform a performance assessment.

5.5.1 Experimental results when deploying Brick of Kulla-Boxes

BrickOf-Kulla-Boxes processed the image repository (28,5GB) in 6,1 minutes for a throughput of 75,90 GB/sec, whereas *TBB-IDA* spent 26.5 minutes to perform the same task. Moreover, *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 *BrickOf-Kulla-Boxes* was achieved for free as the comrpessin degree in this type of digital products was high (in a range of 30-40%) to withstand the same number of failures of traditional algorithm.

6 Conclusions and Future Work

This paper presented the design, development and implementation of Kulla, a construction model that takes advance of lightweight and immutability features of virtual containers (VCs) in the composition of agnostic distributed/parallel applications.

In Kulla model, the applications, including dependencies and environment settings, are encapsulated into construction units called *Kulla-Blocks*. These logical construction structures also include input/output interfaces (network, file system and memory) and in-memory data management schemes, which provides Kulla-Blocks with interoperability functionality for coupling/chaining several Kulla-Blocks in the form of processing structures called *Kulla-Bricks*. These structures enable developers/end-users to build parallel patterns without altering/modifying the applications code to improve the efficiency of these applications.

Kulla-Bricks including parallel patterns such as *Divide-and-Containerize* (producing data parallelism), *Pipe&Blocks* (producing pipelines) and *Manager/Blocks* (producing task parallelism) were developed to show the feasibility and flexibility of applying this model to real-world real-world applications for processing data.

In this model, deployment structures called Kulla-Boxes encapsulate Kulla-Blocks and Kulla-Bricks into virtual containers to provide applications with immutability and agnostic properties, which reduces the need for organizations to perform IT troubleshooting procedures. Bricks of Kulla-Boxes can be built to create distributed/parallel agnostic applications by chaining Kulla-Boxes and by using deployment strategies described in this paper (Scale In, Out, Out/In).

In this model, all the kulla instances reuse the I/O interfaces as are self-similar to the smallest construction unit (Kulla-Block) and are presented to the end-users as a Kulla-Box (a single application); 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 control software instances designed for this construction model.

Case studies based on satellite and medical imagery were conducted by using Kulla instances deployed on a cluster of virtual containers. The experimental evaluation revealed, in a comparison of Kulla-Boxes with serial and parallel (created by using intel TBB) applications, the efficiency and flexibility of Kulla model in the different evaluated scenarios. It also shows the agnostic and interoperable properties of this model as multiple solutions could be built by reusing applications and as these applications could be deployed on different types of IT infrastructures.

Now we are working on models for the building of processing fabrics, operation management models for improving the profitability of resources in automatic manner. By now the scripts and templates used in the solutions evaluated in this paper were created by Kilo service in automatic manner, which also creates the configurations for all the kulla instances (Blocks, Bricks, Boxes and Bricks of Kulla-Boxes). Nevertheless, a programming model also is under construction for developers can build Kulla solutions by using programmable scripts instead a GUI or the Kilo-Service.

Acknowledgments

This work has been partially supported by the "Spanish Ministerio de Economia y Competitividad" under the project grant TIN2016-79637-P "Towards Unification of HPC and Big Data paradigms".

REFERENCES

[1] M. Hayden and R. Carbone, "Securing linux Containers," *GIAC (GCUX) Gold Certification, Creative Commons Attribution-ShareAlike 4.0 International License*, vol. 19, 2015.

[2] M. Souppaya, J. Morello, and K. Scarfone, "Application Container Security Guide," *NIST Special Publication*, vol. 800, p. 190, 2017.

[3] A. Karmel, R. Chadromouli, and M. Iorga, "NIST Definition of Microservices, Application Containers and System Virtual Machines," *Natl Inst. of Standards and Technology (NIST) Special Publication*, pp. 800–180, 2016.

[4] Gartner, Inc. and/or its affiliates, "6 Best Practices for Creating a Container Platform Strategy," 2017, https://www.gartner.com/smarterwithgartner/6-best-practices-for-creating-a-container-platform-strategy/, Last accessed on 2018-11-15.

[5] I. A Forrester Consulting Thought Leadership Paper Commissioned By Dell EMC and R. Hat, "Containers: Real Adoption And Use Cases In 2017," https://www.gartner.com/smarterwithgartner/6-best-practices-for-creating-a-container-platform-strategy/, Last accessed on 2018-11-15.

[6] P. Sharma, L. Chaufournier, P. Shenoy, and Y. Tay, "Containers and Virtual Machines at Scale: A Comparative Study," in *Proceedings of the 17th International Middleware Conference*. ACM, 2016, p. 1.

[7] R. Montella, A. Brizius, D. Di Luccio, C. Porter, J. Elliot, R. Madduri, D. Kelly, A. Riccio, and I. Foster, "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 Generation Computer Systems*, 2018.

[8] J. F. Smart, *Jenkins: The Definitive Guide*. O'Reilly Media, Inc., 2011.

[9] I. Taylor, M. Shields, I. Wang, and A. Harrison, "The triana workflow environment: Architecture and applications," *Workflows for e-Science*, pp. 320–339, 2007.

[10] R. Montella, D. Kelly, W. Xiong, A. Brizius, J. Elliott, R. Madduri, K. Maheshwari, C. Porter, P. Vilter, M. Wilde *et al.*, "FACE-IT: A science gateway for food security research," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 16, pp. 4423–4436, 2015.

[11] T. J. Skluzacek, K. Chard, and I. Foster, "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, 2016, pp. 31–36.

[12] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, "Swift: A language for distributed parallel scripting," *Parallel Computing*, vol. 37, no. 9, pp. 633–652, 2011.

[13] J. Gonzalez-Compean, V. Sosa-Sosa, A. Diaz-Perez, J. Carretero, and J. Yanez-Sierra, "Sacbe: A building block approach for constructing efficient and flexible end-to-end cloud storage," *Journal of Systems and Software*, vol. 135, pp. 143–156, 2018.

[14] G. A. Vazquez-Martinez, J. Gonzalez-Compean, V. J. Sosa-Sosa, M. Morales-Sandoval, and J. C. Perez, "CloudChain: A novel distribution model for digital products based on supply chain principles," *International Journal of Information Management*, vol. 39, pp. 90–103, 2018.

[15] K. Abushab, M. Suleiman, Y. Alajerami, S. Alagha, M. ALnahal, A. Najim, and M. Naser, "Evaluation of advanced medical imaging services at governmental hospitals-gaza governorates, palestine," *Journal of Radiation Research and Applied Sciences*, vol. 11, no. 1, pp. 43–48, 2018.

[16] R. Marcelín-Jiménez and S. Rajsbaum, "Cyclic strategies for balanced and fault-tolerant distributed storage," in *Latin-American Symposium on Dependable Computing*. Springer, 2003, pp. 214–233.

[17] C. G. Riso Andrea M., "NASA Cloud Computing Platform: Nebula." 2010.

[18] J. Gonzalez-Compean, V. J. Sosa-Sosa, A. Diaz-Perez, J. Carretero, and R. Marcelin-Jimenez, "FedIDS: a federated cloud storage architecture and satellite image delivery service for building dependable geospatial platforms," *International Journal of Digital Earth*, pp. 1–22, 2017.

[19] D. del Rio Astorga, M. F. Dolz, J. Fernández, and J. D. García, "Paving the way towards high-level parallel pattern interfaces for data stream processing," *Future Generation Computer Systems*, vol. 87, pp. 228–241, 2018.

[20] R. M. Badia, J. Conejero, C. Diaz, J. Ejarque, D. Lezzi, F. Lordan, C. Ramon-Cortes, and R. Sirvent, "Comp superscalar, an interoperable programming framework," *SoftwareX*, vol. 3, pp. 32–36, 2015.

[21] J. Reinders, *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc., 2007.

[22] W. Gropp, R. Thakur, and E. Lusk, *Using MPI-2: Advanced features of the message passing interface*. MIT press, 1999.

[23] T. White, *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[24] C. Boettiger, "An introduction to Docker for reproducible research," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015.

[25] P. Belmann, J. Dröge, A. Bremges, A. C. McHardy, A. Sczyrba, and M. D. Barton, "Bioboxes: standardised containers for interchangeable bioinformatics software," *Gigascience*, vol. 4, no. 1, p. 47, 2015.

[26] B. Marwick, "Computational reproducibility in archaeological research: basic principles and a case study of their implementation," *Journal of Archaeological Method and Theory*, vol. 24, no. 2, pp. 424–450, 2017.

[27] J. Cito, V. Ferme, and H. C. Gall, "Using Docker containers to improve reproducibility in software and web engineering research," in *International Conference on Web Engineering*. Springer, 2016, pp. 609–612.

[28] P. Morales-Ferreira, M. Santiago-Duran, C. Gaytan-Diaz, J. Gonzalez-Compean, V. J. Sosa-Sosa, and I. Lopez-Arevalo, "A Data Distribution Service for Cloud and Containerized Storage Based on Information Dispersal," in *Service-Oriented System Engineering (SOSE), 2018 IEEE Symposium on*. IEEE, 2018, pp. 86–95.

[29] H. G. Reyes-Anastacio, J. Gonzalez-Compean, M. Morales-Sandoval, and J. Carretero, "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, 2018, pp. 201–206.

[30] J. Gantz and D. Reinsel, "The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east," *IDC iView: IDC Analyze the future*, vol. 2007, pp. 1–16, 2012.

[31] R. Montella, S. Kosta, and I. Foster, "DYNAMO: Distributed leisure Yacht-carried sensor-Network for Atmosphere and Marine data crOwdsourcing applications," in *Cloud Engineering (IC2E), 2018 IEEE International Conference on*. IEEE, 2018, pp. 333–339.

[32] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. F. da Silva, M. Livny *et al.*, "Pegasus, a workflow management system for science automation," *Future Generation Computer Systems*, vol. 46, pp. 17–35, 2015.

[33] J. Liu, E. Pacitti, P. Valduriez, and M. Mattoso, "A survey of data-intensive scientific workflow management," *Journal of Grid Computing*, vol. 13, no. 4, pp. 457–493, 2015.

[34] B. Barney *et al.*, "Introduction to parallel computing," *Lawrence Livermore National Laboratory*, vol. 6, no. 13, p. 10, 2010.

[35] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[36] F. Darema, "The spmd model: Past, present and future," in *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*. Springer, 2001, pp. 1–1.

[37] V. Pieterse and P. E. Black, ""single program multiple data"," in *Dictionary of Algorithms and Data Structures* [online], Vreda Pieterse and Paul E. Black eds., dec 2004, available from: https://xlinux.nist.gov/dads/HTML/singleprogrm.html (accessed 14 March 2018).

[38] J. Diaz, C. Munoz-Caro, and A. Nino, "A survey of parallel programming models and tools in the multi and many-core era," *IEEE Transactions on parallel and distributed systems*, vol. 23, no. 8, pp. 1369–1386, 2012.

[39] D. del Rio Astorga, M. F. Dolz, J. Fernández, and J. D. García, "Supporting advanced patterns in g r ppi, a generic parallel pattern interface," in *European Conference on Parallel Processing*. Springer, 2017, pp. 55–67.

[40] J. G. Blas and J. D. Garcıa, "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, 2016, p. 74.

[41] R. Sotomayor, L. M. Sanchez, J. G. Blas, J. Fernandez, and J. D. Garcia, "Automatic cpu/gpu generation of multi-versioned opencl kernels for c++ scientific applications," *International Journal of Parallel Programming*, vol. 45, no. 2, pp. 262–282, 2017.

[42] M. Grawinkel, M. Mardaus, T. Süß, and A. Brinkmann, "Evaluation of a hash-compress-encrypt pipeline for storage system applications," in *Networking, Architecture and Storage (NAS), 2015 IEEE International Conference on*. IEEE, 2015, pp. 355–356.
[43] F. Buschmann, K. Henney, and D. C. Schmidt, *Pattern-oriented software architecture, on patterns and pattern languages*. John wiley & sons, 2007, vol. 5.
[44] E. A. Posner, K. E. Spier, and A. Vermeule, "Divide and conquer," *Journal of Legal Analysis*, vol. 2, no. 2, pp. 417–471, 2010.
[45] P. Morales-Ferreira, M. Santiago-Duran, C. Gaytan-Diaz, J. Gonzalez-Compean, V. J. Sosa-Sosa, and I. Lopez-Arevalo, "A Data Distribution Service for Cloud and Containerized Storage Based on Information Dispersal," in *Service-Oriented System Engineering (SOSE), 2018 IEEE Symposium on*. IEEE, 2018, pp. 86–95.
[46] J. L. Gonzalez, J. C. Perez, V. J. Sosa-Sosa, L. M. Sanchez, and B. Bergua, "Skycds: A resilient content delivery service based on diversified cloud storage," *Simulation Modelling Practice and Theory*, vol. 54, pp. 64–85, 2015.
[47] J. M. G. (auth.), *Performance Modeling of Operating Systems Using Object-Oriented Simulation: A Practical Introduction*, 1st ed., ser. Series in Computer Science. Springer US, 2002.
[48] M. O. Rabin, "Efficient Dispersal of Information for Security, Load Balancing, and Fault Tolerance," *J. ACM*, vol. 36, no. 2, pp. 335–348, Apr. 1989. [Online]. Available: http://doi.acm.org/10.1145/62044.62050
[49] R. Marcelin-Jimenez, S. Rajsbaum, and B. Stevens, "Cyclic storage for fault-tolerant distributed executions," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 9, pp. 1028–1036, 2006.
[50] M. Quezada Naquid, R. Marcelín Jiménez, and M. López Guerrero, "Fault-tolerance and load-balance tradeoff in a distributed storage system," *Computación y Sistemas*, vol. 14, no. 2, pp. 151–163, 2010.
[51] J. L. Gonzalez and R. Marcelín-Jiménez, "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, 2011, pp. 282–287.
[52] Y. Collet, "LZ4 - Extremely fast compression," https://github.com/lz4/lz4, 2017.