Cloud-based parallel computing across multiple clusters in Julia

Francisco H. de Carvalho Junior
heron@dc.ufc.br
Departamento de Computação
Universidade Federal do Ceará
Fortaleza, Brazil

João Marcelo Uchôa de Alencar
joao.marcelo@ufc.br
Campus de Quixadá
Universidade Federal do Ceará
Quixadá, Brazil

Claro Henrique Silva Sales
clarohenriques@gmail.com
Pós-Graduação em Ciência da Computação (MDCC)
Universidade Federal do Ceará
Fortaleza, Brazil

Abstract

IaaS cloud providers are now a de facto alternative to HPC. They offer a rich catalog of virtual machine instances with high-end processors and accelerators connected through advanced network technology. This makes it possible to create cluster computing platforms rivaling the on-premises alternatives. This paper presents an Infrastructure as Code (IaC) approach to build parallel computing systems in Julia, both hardware and software elements, benefiting users of HPC applications in dynamic programming languages.

1 Introduction

Cloud computing platforms emerged as an alternative to providing services for HPC applications [16, 33], with offers at different abstraction levels, ranging from on-demand infrastructure to building high-end clusters (IaaS) to domain-specific end-user applications (SaaS), as well as high-level support to develop and deploy applications (PaaS).

IaaS providers allow users to create clusters of virtual machines with high-end processors, accelerators, and interconnections, taking advantage of state-of-the-art processing power without worrying about the obsolescence of hardware and scaling resources according to their current needs and budget. The first cloud-based supercomputers reached the top positions of Top500 [27] recently, with Microsoft Azure's Eagle [28] reaching the 3rd position in November 2023.

Julia is a dynamically compiled programming language that emerged in the late 2000s for scientific and technical computing applications [15]. It attempts to reconcile development productivity and HPC requirements, making it possible to write code that runs as fast as C or Fortran code [14, 34].

This paper contributes with an Infrastructure as Code (IaC) approach to deploy cloud-based cluster computing resources in Julia programs, where programmers declaratively create clusters comprising a set of virtual machine (VM) instances as compute nodes by specifying assumptions about their features. As an alternative to specific description languages, IaC benefits from a dynamic language like Julia for seamless integration between developers and IaaS providers, making it easier to deal with computationally intensive parallel code.

The product of this work is CloudClusters.jl, a Julia package for deploying clusters across multiple cloud providers to run parallel code based on the Distributed.jl and MPI.jl packages. CloudClusters.jl introduces *contextual contracts* on top of Julia's multiple dispatch to represent assumptions and features for selecting instance types for cluster nodes. Using PlatformAware.jl [21], the implementation of functions may be tuned to exploit features of clusters and their nodes.

Two case studies are presented for a proof-of-concept evaluation of the IaC approach behind CloudClusters.jl. The first comes from scientific computing: a Julia implementation of the multizone version of the NAS Parallel Benchmarks (NPB-MZ) [25]. The second one comes from artificial intelligence (AI): parallel deep learning using Flux.jl, a widely used deep learning package in Julia's ecosystem.

This paper comprises four more sections. Section 2 presents background and related works. Section 3 details the design of CloudClusters.jl. Section 4 presents and discusses the case studies and the proof-of-concept evaluation. Finally, Section 5 presents final remarks and lines for further works.

2 Background and Related Works

Cloud computing has become the backbone of today's digital society. Through a wide variety of services, clouds have something to offer for every use case. The abundance of alternatives implies that even researchers in HPC, whose goal is to extract all the potential performance from computing platforms, soon realized cloud-based platforms as an alternative to run parallel programs [16, 33].

The first experiments [2, 26] of using clouds for HPC showed that although clouds offers powerful computing resources, they had limitations, including the complexity of configuration and environmental dynamicity that jeopardized the assurance of QoS requirements. Nevertheless, cloud providers have improved the suitability of their offerings for HPC with tailored instance types and network connectivity [9, 10, 35]. Recently, the widespread interest in IA and Deep Learning [13, 31] motivated further investments and research that moved cloud-based HPC from a promise to a reality [16, 33]. For example, Microsoft Azure's Eagle and EOS NVIDIA DGX SUPERPOD, both cloud-based systems aimed at Deep Learning applications, recently reached the top 10 positions in the Top500 ranking of supercomputers [27].

To tackle the complexity of configuration, cloud providers support describing infrastructure resources as code (Infrastructure as Code - IaC). Instead of building clusters using the web console or prompt commands, developers can use declarative code to specify the characteristics of the computing nodes, interconnect networking, file system capacity, etc. Researchers can put this description under version control and recreate the environment whenever they need to run an application [36]. The downside of this approach is that the available tools, like Terraform [17], demand a specific description language for templates, so the developers must learn a new syntax instead of relying on the concepts of the programming language they already use in the parallel code.

2.1 The Julia Programming Language

Julia is a programming language targeting scientific and technical computing applications [15]. It is maintained as an open-source project[1] by the JuliaHub[2] company and a community of users, offering several packages for solving problems in different domains of sciences and engineering[3].

Julia aims to reconcile the productivity of dynamic languages like Python with the performance of native execution languages like FORTRAN and C [14]. For that, it combines the technology of just-in-time (JIT) compilation with a rich type system [32] designed to support a dynamic multiple dispatch mechanism for specifying and invoking different methods of a function, which vary according to the type of their formal parameters. This approach makes it possible for the JIT compiler to generate efficient native code for methods that satisfy the *type stability* property, for which the primitive type of local variables, especially the ones returned by the function, can be inferred from the parameter types [34].

2.1.1 Distributed-memory parallelism in Julia.

Julia supports distributed computing through Distributed.jl[4], a built-in package mainly for distributed computing. A distributed Julia program comprises P processes, numbered from 1 to P, where the process 1 is called *master* and the processes from 2 to P are called *workers*. Initially, the standalone program or REPL session initiated by the user has a single master process, which may create worker processes locally or in remote hosts/nodes by calling the addprocs function. Only the master process is allowed to call addprocs. The procs, nprocs, workers, nworkers, and myid functions may be used to inspect the number and identities of worker processes, and they can be removed by calling rmprocs.

The interaction between the master and the worker processes is one-sided through asynchronous remote evaluation of expressions. To evaluate a Julia expression onto another process, one may invoke the @spawnat macro, passing the process identifier and executing code as arguments. Immediately, a future value of Future type is returned, which may be passed to the @fetch macro (or fetch function) to wait and receive the result of the remote evaluation.

There are some variants of @spawnat, such as @spawn, which executes the expression in an arbitrary worker, and @fetchfrom, which combines calls to @spawnat and @fetch. In turn, @everywhere evaluates an expression across all workers or a subset of workers, useful for data parallelism. Finally, the remotecall function provides support for asynchronous remote invocation of functions, with synchronous variations, remotecall_fetch and remotecall_wait, applied for functions that return and do not return a value, respectively. Distributed.jl also offers high-level operations for distributing data and computations across workers, such as @distributed and pmap, implementing the map/reduce paradigm.

The host where each worker process is instantiated is determined by passing a *cluster manager* as the first argument to addprocs. A cluster manager is an abstraction for the distributed environment where worker processes will be instantiated. So, addprocs has a different method for each cluster manager, selected through multiple dispatch. Julia provides two built-in cluster managers: LocalManager, for launching additional workers on the same host of the master process, and SSHManager, for launching workers on remote hosts that accept ssh authentication.

The ClusterManager.jl[5] package provides cluster managers for common job queue systems used on clusters, such as Slurm, Kubernetes, LSF (Load Sharing Facility), SGE (Sun Grid Engine), PBS (Portable Batch System), etc. In turn, MPIClusterManagers.jl[6] provides support for message-passing interaction among worker processes through MPI.jl. This third-party Julia package implements MPI [23], a standard message-passing library widely used in HPC.

2.2 Related Works

ElastiCluster [30] and ParallelCluster [4] are popular projects for creating HPC clusters on clouds through the IaC approach. Although they differ on the technological stack, their usage is similar. Users must define the cluster characteristics in a separate file. For ElastiCluster, the configuration file has the .ini format, while ParallelCluster uses YAML. They do not interact directly with the cloud. For that, they rely on other industry-leading projects to create clusters. ElastiCluster employs Ansible [5], and ParallelCluster uses CloudFormation [8]. Besides learning the syntax of the input file, the researcher interacts with the command line interface and uses SSH to log into the cluster and submit applications.

Recently, projects like HPC@Cloud [29] have improved the scheme of ElastiCluster and ParallelCluster with cost

[1] https://github.com/JuliaLang/julia
[2] https://juliahub.com/
[3] https://juliahub.com/ui/Packages
[4] https://github.com/JuliaLang/Distributed.jl
[5] https://github.com/JuliaParallel/ClusterManagers.jl
[6] https://github.com/JuliaParallel/MPIClusterManagers.jl

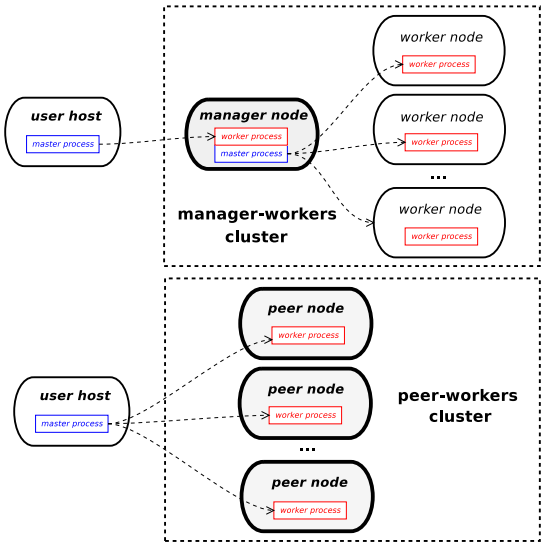

Figure 1. MW and PW clusters

management and elasticity support but still rely on Terraform, suffering from the same issues regarding the use of separate configuration languages. vClusters [3] has a similar approach but fully embraces the Cloud Native and DevOps development philosophies. With Kubernetes [6], administrators can create pipelines for deploying HPC applications on supercomputers and clouds with minimal user configuration. This is closer to allowing the researcher to focus on the application code but requires an established operations team, which is uncommon in many research facilities.

MCMPI [1] allows the programmer to create clusters and add or remove nodes (elasticity) directly through code without recompilation for MPI applications written in C. The researcher may use several clusters at once, but currently, MCMPI only supports AWS as a cloud provider. Even though the message-passing paradigm continues to be widespread, developers with parallel applications that do not adhere to it will have difficulty using MCMPI.

Users without the support of an operations team who want to uniformize the application and infrastructure code can use the cloud provider's software development kit (SDK), if any. Most cloud providers release SDKs for different programming languages. However, they are hardly good choices since they are merely a layer over the cloud REST services, providing poor abstractions and doubtful maintainability.

3 The CloudClusters.jl package

We have developed the CloudClusters.jl for deploying cloud-based clusters in Julia programs package[7]. It provides a set of macros and functions to create clusters of virtual machine (VM) instances, the cluster nodes, through the services of IaaS cloud providers. For that, the users specify a *contextual contract* declaring a set of assumptions about features of these instances to guide the selection of *instance types*. Also, they take advantage of the seamless integration with Distributed.jl to execute computations on the deployed clusters.

The current prototype of CloudClusters.jl only supports creating clusters through the Amazon's EC2 services [7], using the third-party AWS.jl package [18].

Contrarywise to the alternatives described in Section 2.2, CloudClusters.jl introduces an IaC approach where the code that configures and instantiates the clusters is specified using the same language used to write the code of the parallel algorithms that these clusters will execute. Also, it innovates by using contextual contracts [20] to specify cluster features. With contextual contracts, users of CloudClusters.jl may adhere to the view of *parallel computing system*, where they are concerned not only with the software aspects in the implementation of parallel algorithms but also with the hardware aspects, i.e., the features of parallel computing platforms where the code can execute efficiently by exploiting these features. This approach encourages *platform-aware programming*, i.e., code writing by making assumptions about the features of the target execution platforms. Indeed, CloudClusters.jl is being integrated to PlatformAware.jl, a Julia package for structured platform-aware programming [21].

The fundamental macros offered by CloudClusters.jl are:

- **@cluster**, to specify a set of *assumptions* about a predefined set of cluster *features*, through a *contextual contract*, which will guide the creation of clusters of the following types, depicted in Figure 1:
  - *manager-workers* (MW), with a homogeneous set of compute nodes (*workers*) and a *manager node* through which worker nodes may be accessed.
  - *peer-workers* (PW), with a homogeneous set of directly accessible compute nodes (*peers*).
- **@resolve**, to resolve a contextual contract specified by a @cluster declaration by finding which instance type satisfies the assumptions specified in the contract for worker and manager nodes of MW clusters or worker nodes of PW ones;
- **@deploy**, to create a cluster from a resolved contextual contract by invoking the selected IaaS provider, putting it in the *running* state by instantiating VM instances (cluster nodes) of the instance types calculated in the contract resolution.
- **@interrupt**, to move a cluster from the *running* state to the *interrupted* state, pausing the use of cloud resources (e.g., the virtual machine instances that comprise their nodes);
- **@resume**, the inverse of @interrupt, i.e., to move an interrupted cluster back to the running state;
- **@terminate**, to move a cluster from the *running* or *interrupted* state to the *terminated* state, where all the cloud resources used by the cluster are freed.

[7] https://github.com/PlatformAwareProgramming/CloudClusters.jl

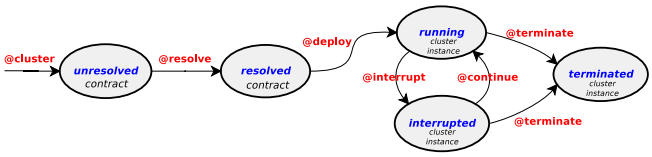

Figure 2. Lifecycle of contracts and clusters

```
contract_x = @cluster cluster_type=ManagerWorkers ⟨more assumptions⟩
contract_y = @cluster cluster_type=PeerWorkers ⟨more assumptions⟩

@resolve contract_x contract_y                  # resolve both contracts
@resolve contract_y                               # no effect

cluster_a, pid_master_1 = @deploy contract_x      # 1st cluster
cluster_b, pids_peers_2 = @deploy contract_y      # 2nd cluster
cluster_c, pid_master_3 = @deploy contract_x      # 3rd cluster

    launch computations over the three clusters using Distributed.jl

@terminate cluster_a cluster_c    # 1st and 3rd clusters terminated
@interrupt cluster_b                # 2nd cluster not necessary at this time

@resume cluster_b

    launch computations on the second cluster

@terminate cluster_b              # no active clusters from this point
```

Figure 3. Creating clusters

The diagram of Figure 2 depicts the states of the lifecycle of clusters and their contracts. Also, Figure 3 illustrates the operations by creating and using three clusters from two contracts. Contextual contracts are described in Section 3.1.

The @cluster and @resolve macros are designed to work in an interactive environment, such as Julia's REPL interface. For that, a call to @cluster returns a fresh symbol to make the role of a *contract handle*. After the call, such a handle of an unresolved contract can be applied in a call to @resolve to resolve it. The @resolve invocation is idempotent, i.e., it has no effect when applied to a resolved contract. Finally, the handle of a resolved contract can be applied in a call to deploy to create a cluster. A call to @deploy also returns a *cluster handle*, which may be applied to @interrupt, @resume, and @terminate macros. Notice that multiple cluster instances may be deployed from the same cluster contract.

Besides the cluster handle, **@deploy** also returns a single integer number or a list of integer numbers. They correspond to process identifiers (*pid*) of Distributed.jl processes. For a MW cluster, it is the pid of the process running in the manager node of the cluster, while they are pids of processes running in the peer nodes of the cluster in the case of PW clusters. In a MW cluster, remember that the processes at the worker nodes are only accessible from the process at the manager node that created them.

It is worth noticing that MW clusters are only possible with an extension of Distributed.jl[8] we have developed to make it possible to create processes from worker processes in distributed Julia programs, removing a restriction of addprocss. In the standard implementation of Distributed.jl, worker processes cannot make calls to addprocs, i.e., only the master process can create processes. We have found this extension is necessary to make it possible for Julia users to access clusters whose access from an external network to compute nodes is only possible through its master node. This is the case of most on-premises clusters, which do not offer IPv6 addressing to make compute nodes accessible, as well as a possible restriction in cloud providers where the use of public addresses incurs costs that the clients want to avoid.

CloudClusters.jl and Distributed.jl are fully integrated so that users may execute parallel code in the clusters through remote function invocation operations supported by Distributed.jl, which also give support to the communication between these clusters. Moreover, for MW clusters, processes at worker nodes can exchange messages through MPI, using the MPI.jl and MPIClusterManagers.jl packages. However, communication with processes at other clusters is only possible through the process at the master node. In fact, using CloudClusters.jl, a user may aggregate the computation power of multiple clusters running in the infrastructure of distinct cloud providers, forming multicluster environments.

3.1 Contextual Contracts

Contextual contracts come from HPC Shelf, a proposal of a component-oriented platform to offer HPC services through clouds [20]. For CloudCluster.jl, contextual contracts may be defined as a set of context parameters $n : \tau$, where n is the name of the context parameter and τ denotes an *assumption* about a *feature* of a cluster (or node). Assumptions are represented by a system of platform types with a subtyping relation (<:) between them, where assumptions in the leaf of the subtyping hierarchy represent features. Finally, a resolution mechanism matches contracts and instance types offered by the supported IaaS providers so that, from a given contextual contract C specifying a set of assumptions $\{\mathcal{A}_i \mid i = 1...n\}$, it may find one or more instance types whose set of features $\{\mathcal{F}_i \mid i = 1...n\}$ specified by their contracts satisfy the assumptions of C. We say $\mathcal{F}_i <: \mathcal{A}_i$, for $i = 1...n$.

CloudClusters.jl assumes that a cluster is associated with a contextual contract with a subset of parameters representing assumptions about features of the instance type used to implement their nodes (*instance features*). To implement qualifier types (assumptions and features), it employs the approach of PlatformAware.jl, a Julia package for structured platform-aware programming [21], where Julia types represent qualifier and quantifier types.

With platform types, contract resolution may be implemented through Julia's dynamic multiple dispatch. For that, in the first loading of CloudCluster.jl, a set of methods for a function called resolve are dynamically created through metaprogramming for each instance type of each IaaS cloud

[8]The modified version of the standard Distributed.jl package is currently hosted at https://github.com/PlatformAwareProgramming/Distributed.jl.

provider supported, whose features are described in CSV files stored in the PlatformAware.jl repository for each provider. In the prototype used for the proof-of-concept evaluation reported in this paper, there are 795 methods for the resolve function, one for each instance/machine type of EC2 and GCP providers[9]. The parameters of each resolve method specify the features of an instance type. For example, the following method is called for selecting the EC2's instance type g4dn.xlarge from a set of compatible assumptions [10]:

```
function resolve (node_provider::Type{>:AmazonEC2},
                  node_machinetype::Type{>:EC2Type_G4DN_xLarge},
                  node_memory_size::Type{>:Tuple{AtLeast16G, AtMost16G, 1.7e10}},
                  node_vcpus_count::Type{>:Tuple{AtLeast4, AtMost4, 4.0}},
                  accelerator_count::Type{>:Tuple{AtLeast1, AtMost1, 1.0}},
                  accelerator_type::Type{>:GPU},
                  accelerator_architecture::Type{>:Turing},
                  accelerator_manufacturer::Type{>:NVIDIA},
                  accelerator::Type{>:NVIDIATesla_T4},
                  processor::Type{>:IntelXeon},
                  processor_manufacturer::Type{>:Intel},
                  processor_microarchitecture::Type{>:IntelMicroarchitecture},
                  storage_type::Type{>:StorageType_SSD},
                  storage_size::Type{>:Tuple{AtLeast64G, AtMost128G, 1.3e11}},
                  network_performance::Type{>:Tuple{AtLeast0, AtMost32G, 2.7e10}})
  return "g4dn.xlarge"
end
```

Thus, @resolve calls the resolve function to select an instance type from a set of assumptions that it specifies through the arguments. For example, consider the following @resolve call, which attempts to select an EC2 instance type with a single NVIDIA's Tesla GPU of Turing architecture, as well as between 16GB and 32GB of host memory:

```
my_contract = @cluster cluster_type => PeerWorkers
                       node_provider => PlatformAware.AmazonEC2
                       node_memory_size => (PlatformAware.@between 16G 32G)
                       accelerator => PlatformAware.Tesla
                       accelerator_count => (PlatformAware.@just 1)
                       accelerator_manufacturer => PlatformAware.NVIDIA
                       accelerator_architecture => PlatformAware.Turing
@resolve my_contract
```

For that, it generates the following call to resolve:

```
resolve (AmazonEC2, # node_provider
         MachineType, # node_machinetype
         Tuple{AtLeast16G, AtMost32G, M} where M, # node_memory_size
         Tuple{AtLeast1, AtMostInf, X} where X, # node_vcpus_count
         Tuple{AtLeast1, AtMost1, X} where X, # accelerator_count
         GPU, # accelerator_type
         Turing, # accelerator_architecture
         NVIDIA, # accelerator_manufacturer
         Tesla, # accelerator
         ProcessorModel, # processor
         Manufacturer, # processor_manufacturer
         ProcessorMicroarchitecture, # processor_microarchitecture
         StorageType, # storage_type
         Tuple{AtLeast0, AtMostInf, X} where X, # storage_size
         Tuple{AtLeast0, AtMostInf, X} where X # network_performance)
```

By dynamic multiple dispatch, the resolve method that selects the g4dn.xlarge instance type is called since it satisfies the assumptions. Dynamic multiple dispatch imposes that only a single instance must satisfy the assumptions. Otherwise, it causes an ambiguity error. This is the main drawback of Julia's multiple dispatch for contract resolution.

[9]In the current prototype, the support for deploying clusters using the GCP (*Google Computing Platform*) IaaS provider is not yet support.

[10]In this code, Type{>:*T*}, for Julia's type *T*, denotes the type including all the supertypes of *T*.

name	*instance feature ?*	*default type*
node_count	no	Integer
node_process_count	no	Integer
node_provider	yes	CloudProvider
cluster_locale	yes	Locale
node_machinetype	yes	InstanceType
node_memory_size	yes	@atleast 0
node_ecu_count	yes	@atleast 1
node_vcpus_unit	yes	@atleast 1
accelerator_count	yes	@atleast 0
accelerator_memory	yes	@atleast 0
accelerator_type	yes	AcceleratorType
accelerator_arch	yes	AcceleratorArchitecture
accelerator	yes	AcceleratorModel
processor	yes	ProcessorModel
processor_manufacturer	yes	Manufacturer
processor_microarchitecture	yes	ProcessorArchitecture
storage_type	yes	StorageType
storage_size	yes	@atleast 0
network_performance	yes	@atleast 0
image_id	no	String
user	no	String
key_name	no	String
subnet_id	no	String
placement_group	no	String
security_group_id	no	String

- The *instance features* are the ones used by the contract resolution procedure for selecting the instance type of cluster nodes;
- In the case of MW clusters, distinct instance features may be provided separately for the master and the worker nodes;
- Only **node_count** is mandatory, denoting the number of worker nodes for MW clusters or peer nodes for PW clusters.

Table 1. context parameters

Table 1 presents the feature types supported by the current prototype of CloudClusters.jl, highlighting the instance features that form the signature of the resolve methods.

3.2 Usage example

A user is interested in building a PW cluster whose nodes are equipped with at least four GPUs of NVIDIA's Ampere architecture. First, it may query CloudClusters.jl to check if the supported cloud providers offer some instance type that satisfies this assumption by running the following command through Julia's REPL:

```
query_result = @select accelerator_count=>(@atleast 4)
                       accelerator_architecture=>PlatformAware.Ampere
```

The current version of CloudClusters.jl uses the instance type database of PlatformAware.jl. So, it returns, through the query_result variable, a dictionary presenting the features of five instance types: g5.12xlarge, g5.24xlarge, g5.48xlarge, p4de.24xlarge, and p4d.24xlarge. They are accelerated instance types offered by the AWS EC2 provider.

If the user executes @cluster over the above assumptions, followed by @resolve, the resolution fails due to ambiguity since it cannot decide which instance type to select:

```
my_contract = @cluster cluster_type => PeerWorkers
                       accelerator_count => (@atleast 4)
                       accelerator_architecture => PlatformAware.Ampere
@resolve my_contract
```

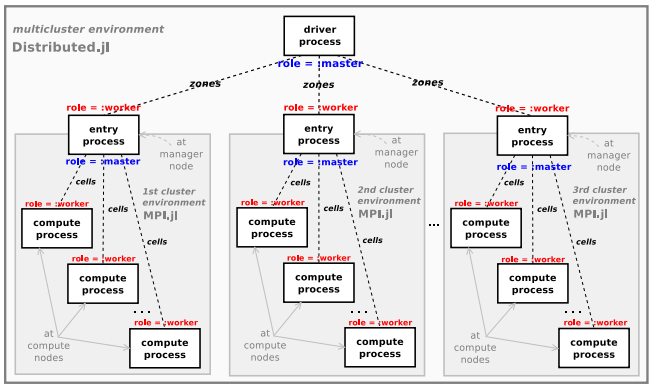

Figure 4. Multizone NPB's Architecture (SP, BT, LU)

As mentioned, this behavior results from using Julia's multiple dispatch to implement the contract resolution. To complete the resolution successfully, the user may either select an instance type explicitly or add new assumptions that make @select return a single instance type. Both cases are illustrated below:

```
contract_1 = @cluster cluster_type=>PeerWorkers
                      node_machinetype=>PlatformAware.EC2_G5_12xLarge

contract_2 = @cluster cluster_type => PeerWorkers
                      accelerator_count => (@just 4)
                      accelerator_architecture => PlatformAware.Ampere
                      node_memory_size => (@atmost 256G)

@resolve my_contract_1 my_contract_2    # both return only "g5.12xlarge"
```

Using the first contract, g5.12xlarge is directly selected to satisfy the assumptions. The result is the same using the second contract, with two restrictions added. First, the peer nodes must have 4 GPU devices, excluding p4d.24xlarge, p4de.24xlarge, and g5.48xlarge instance types, which offer 8 GPUs per instance. Second, peer nodes must have at most 256GB of memory, excluding g5.24xlarge.

4 Case Studies

Two case studies have been designed as a proof-of-concept for CloudClusters.jl. The first employs MW clusters for multicluster parallelism in a Julia implementation of a well-known parallel computing benchmark suite. The second employs PW clusters for distributed deep learning using Flux.jl.

4.1 Multizone NPB

NPB (NAS Parallel Benchmarks) is a benchmark suite developed in the early 1990s that became popular for evaluating parallel computing platforms and programming interfaces [11, 22]. It comprises a set of programs and standard problem classes that evolved over time, now including official serial and parallel versions written in C, Fortran, HPF, and Java, with parallel versions targeting both distributed-memory and shared-memory parallel computers.

This case study uses the multi-zone version of NPB (NPB3.4-MZ-MPI) [24, 25], including the simulated applications SP, BT, and LU, where the grid is partitioned into zones distributed across the cluster's nodes through MPI. In each node, zone processing is parallelized using OpenMP directives.

```
using CloudClusters, PlatformAware, NPBApps

contract_1 = @cluster cluster_type => ManagerWorkers
                      cluster_location => PlatformAware.EC2Region_USEast1
                      node_count => 4
                      node_machinetype => PlatformAware.EC2Type_T3
                      node_memory_size => (@atleast 16G)

contract_2 = @cluster cluster_type => ManagerWorkers
                      cluster_location => PlatformAware.EC2Region_SAEast1
                      node_count => 4
                      node_machinetype => PlatformAware.EC2Type_T3
                      node_memory_size => (@atleast 16G)

@resolve contract_1 contract_2

cluster_1, _ = @deploy contract_1
cluster_2, _ = @deploy contract_2

function run_experiment(N)
    for npb in [SP, BT, LU]
        npb.benchmark(npb.CLASS_C, N)    # running benchmark N times.
    end
end

@everywhere workers() @everywhere workers() using NPBApps

run_experiment(5) # run experiment with two clusters.

@terminate cluster_2

run_experiment(5) # run experiment with a single cluster.
```

Figure 5. Running SP, BT, LU over MW clusters

We have developed a Julia version of NPB3.4-MZ-MPI that exploits *multicluster parallelism* at the first parallelism level and *cluster parallelism* at the second level, whose architecture is depicted in Figure 4. The set of zones is distributed across clusters through Distributed.jl, which becomes responsible for the data exchange between adjacent zones' faces in different clusters. In a cluster, each zone is partitioned into cells that are distributed across cluster nodes using MPI.jl, which is responsible for communication between the faces of adjacent cells and zones assigned to the same cluster.

Figure 5 outlines the code for a scenario where a user creates two clusters at different EC2 regions, US East (Virginia) and South America (São Paulo), to run an experiment with SP, BT, and LU. Both clusters comprise four nodes of t3.xlarge instance type. The experiment measures the impact of inter-cluster communication, with clusters placed at distinct regions, for the problem size of class C. For adjacent zones deployed at distinct clusters, the inter-zone face exchange is performed through Distributed.jl. Otherwise, for adjacent zones in the same cluster, MPI.jl is used.

Table 2 presents the performance measures for the experiment. The execution time is the average (μ) of three in five executions, eliminating the two higher values. From a single cluster to two clusters, the inter-zone communication cost for SP and BT increased from inexpressive 4.32% and 1.54% to highly significant 64.7% and 65.9%. For LU, the impact of

	seq	1 cluster				2 clusters					
	total	total	computation	communication		total		computation		communication	
	$\mu_{\pm\sigma}$(s)	$\mu_{\pm\sigma}$(s)	$\mu_{\pm\sigma}$(s)	$\mu_{\pm\sigma}$(s)	fraction	$\mu_{\pm\sigma}$(s)	speedup	$\mu_{\pm\sigma}$(s)	speedup	$\mu_{\pm\sigma}$(s)	fraction
SP	$240.9_{\pm 2.81}$	$361.0_{\pm 12.0}$	$345.4_{\pm 12.0}$	$15.6_{\pm 0.09}$	4.32%	$465.4_{\pm 11.7}$	0.8	$164.2_{\pm 3.9}$	2.1	$301.1_{\pm 13.4}$	64.7%
BT	$475.7_{\pm 6.13}$	$360.1_{\pm 5.62}$	$354.6_{\pm 5.4}$	$5.56_{\pm 0.21}$	1.54%	$480.6_{\pm 20.5}$	0.8	$163.7_{\pm 6.0}$	2.2	$316.8_{\pm 26.3}$	65.9%
LU	$87.8_{\pm 0.29}$	$109.6_{\pm 1.9}$	$109.1_{\pm 1.95}$	$0.5_{\pm 0.01}$	0.45%	$162.9_{\pm 14.2}$	0.7	$117.0_{\pm 7.57}$	0.9	$45.8_{\pm 6.58}$	28.1%

Table 2. NPB-MZ performance measures

multicluster execution for inter-cluster communication is lower, but it also increased significantly, from 0.45% to 28.1%.

The speedups are poor, between 0.7 and 0.8, due to the high inter-cluster communication overheads. However, disregarding inter-zone communication time, only considering intra-cluster execution, SP and BT speedups are linear (2.0). This is not true for LU due to a high increase in intra-cluster communication for two clusters.

4.1.1 Discussion. The poor multicluster speedups are due to the SP, BT, and LU parallelism model, not an intrinsic limitation of CloudClusters.jl or multicluster execution. These programs implement tightly coupled computations designed to run efficiently in MPPs and clusters with low-latency interconnections. Nobody would run such parallel programs on multiple clusters expecting a significant speedup when the communication between the clusters is intensive. In fact, besides performing a proof-of-concept test and quantifying communication overheads between cloud-based clusters in an extreme scenario, this experiment evidences that it is only justified to run tightly coupled parallel computations on multiple clusters when the problem size does not fit the memory capacity of a single cluster and the schedule constraints restrictions for having the results available are compatible with the overhead due to inter-cluster communication. For example, a single cluster of t3.xlarge instances cannot run problem classes D, E, and F. So the user must compare the cost of using bigger (and costly) instance types or increasing the cluster size by allocating more instances. However, allocating more resources in the same region may be impossible or restricted due to a lack of resources or cloud provider policies. In this case, depending on the kind of parallel program, the multicluster deployment feature allows the user to benefit from allocating a cluster in either another region or through another cloud provider to run a parallel program.

4.2 Distributed Deep Learning

In recent years, there has been a rapid increase in interest in artificial intelligence (AI) applications, most of which is explained by the evolution of machine learning techniques for deep neural networks (DNNs). This led to the emergence of many tools to support *deep learning* in programming language ecosystems, especially Python.

```
using CloudClusters, PlatformAware, MLDatasets: CIFAR10

% build the cluster
contract = @cluster cluster_type => PeerWorkers
                    node_count => P              # for P = 1, P = 2, and P = 4
                    node_process_count => N      # for N = 1 and N = 2
                    accelerator_count => @just(1)
                    accelerator_architecture => PlatformAware.Turing
                    accelerator_memory => @atleast(16G)
@resolve contract % EC2's g4dn.xlarge will be selected
cluster, _ = @deploy contract

# call the distributed deep learning function over the deployed cluster
# • model: ResNet-18 (https://doi.org/10.1109/CVPR.2016.90)
# • dataset: CIFAR10 (https://www.cs.toronto.edu/ kriz/cifar.html)
teach_the_model(:resnet18, CIFAR10, cluster)
```

Figure 6. Running distributed training on a PW cluster

HPC techniques have also contributed to the success of DNNs, using accelerators, especially GPUs and their variants. In a vicious cycle, the relevance of challenging IA applications has motivated the industry to invest in advancing accelerator technology by developing new devices specially designed to accelerate DNN training and inference, and IaaS cloud providers have offered instance types equipped with such AI accelerators to their customers. In the near future, any computing device, from smartphones to personal computers, will be equipped with neural processing units (NPUs).

Distributed computing complements accelerator-based computing, enabling exploiting the combined performance of multiple accelerators and processors to address challenging AI applications [12]. As evidence, MPPs and clusters specially designed for AI workloads have been classified in the Top500 ranking (June 2024), some now occupying the top positions, such as Intel's Aurora (#2), Microsoft Azure's Eagle (#3), and EOS NVIDIA DGX SUPERPOD (#10) systems.

Flux.jl is a package that enables machine learning using deep neural networks (DNNs) in Julia programs [19]. In this case study, a PW cluster comprising four workers is deployed to study a distributed training strategy for a Flux.jl implementation of the ResNet18 model to solve an image classification problem over the CIFAR10 dataset, where each worker has a dataset partition to train a copy of the global model locally. Then, a new global model is generated by combining the local models through an *aggregation function*.

This study aims to evaluate the impact of an aggregation function that averages the model parameters on accuracy improvement and overall training time. Figure 6 depicts the

nodes	processes	time★	speedup	epochs	time per epoch	speedup per epoch
seq	-	1904	1.00	7	253	1.00
1	1	1854	1.03	7	252	1.00
1	2	1996	0.95	11	173	1.46
2	2	1571	1.21	11	132	1.93
2	4	1514	1.26	15	94	2.48
4	4	1310	1.45	16	75	3.38
4	8	2885	0.66	45	62	4.10

★ *training time in seconds (disregarding accuracy testing time).*

Table 3. Deep learning case study - performance results

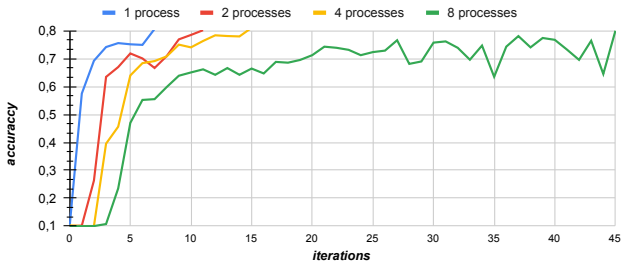

Figure 7. Convergence to 0.8 accuracy

code for that. The contract requires worker nodes with a single NVIDIA Turing GPU and at least 16GB of memory. The resolution returns the EC2's g4dn.xlarge instance type, equipped with a single Tesla T4 GPU.

The *teach_the_model* function performs distributed training. Since Flux.jl does not natively support distributed training, it is explicitly programmed by launching a set of P worker processes at distinct cluster nodes using Distributed.jl. They traverse P dataset partitions of the same size in parallel at each epoch. After each epoch, the local models calculated by the workers are aggregated in a global model by averaging parameters. Then, the global model is sent to each worker to start the next epoch. The computation ends when a 0.8 accuracy is achieved. The experiment has been performed for P equal to 1, 2, and 4. For each P, two scenarios were considered: one with a single process and another with two processes at the same node, sharing the GPU. This resulted in a total of 6 experimental cases. The training time (per epoch) and the number of epochs (iterations) to achieve the predefined accuracy (80%) are presented in Table 3.

In Figure 7, although the training time per epoch presents a sublinear decrease with the number of processes, it is compensated by the increase in epochs to achieve 0.8 accuracy. Overall, the total training time is reduced until the number of processes reaches 4, presenting a modest 1.45 speedup across 4 nodes after 16 iterations, doubling the 7 iterations needed by the sequential version. Indeed, for 8 processes on 4 nodes, the number of epochs to reach 0.8 accuracy increases to 45, becoming slower than the sequential version, with a 0.66 speedup. The measures show overhead by placing 2 processes in a single GPU to increase occupancy. For example, the speedup achieved for 4 processes on 2 nodes was 1.26.

5 Conclusions

Infrastructure-as-Code (IaC) is a popular alternative for describing and deploying HPC clusters over clouds of infrastructure services. However, using specific-purpose configuration languages and separate tools for this purpose may be cumbersome to many users, and SDKs aimed at IoC integrated into programming languages, when they exist, offer poor abstractions and doubtful maintainability.

CloudClusters.jl, the main artifact of this research work, is a Julia package to exercise the idea of HPC programmers using the same programming language to deal with software and hardware elements of parallel computing systems instead of thinking separately about them. Compared to configuration languages, this approach makes it easier to describe and deploy the cluster infrastructure and execute parallel code. In addition, such an approach encourages performance engineering through platform-aware programming, i.e., programming by making assumptions about the underlying parallel computing platform features. For that, CloudClusters.jl is integrated with PlatformAware.jl, a package of Julia's ecosystem for structured platform-aware programming.

Another innovation behind CloudClusters.jl is using contextual contracts to describe assumptions about cluster features and the resolution mechanism to create clusters from these contracts. i.e., that satisfies contract assumptions. Implementing contextual contracts on top of Julia's type system for using dynamic multiple dispatch to implement contract resolution also deserves special attention.

The ideas behind CloudClusters.jl can be ported to other programming languages through a library of functions/subroutines implementing the CloudClusters.jl primitives. Also, using metaprogramming, if it is supported, one may embed a DSL to increase abstraction. The main difficulty is implementing contract resolution without dynamic multiple dispatch. However, implementing contract resolution without native multiple dispatch makes it possible to circumvent some limitations, such as dealing with ambiguous contracts. For example, like in the contextual contract system of HPC Shelf [20], context parameters representing QoS and cost assumptions could be introduced to resolve ambiguities in the resolution of instance types that satisfy a contract. Dynamic multiple dispatch is also important in implementing structured platform-aware programming, but research on finding other ways to implement it is welcome.

CloudClusters.jl will be published in Julia's ecosystem after concluding its proof-of-concept studies, partially reported in this paper. By then, we plan to include the Google Cloud Platform (GCP) support. Finally, in-depth studies will be performed on platform-aware programming in parallel computing systems deployed through CloudClusters.jl.

References

[1] C. A. T. Aguni, L. M. Sato, and E. T. Midorikawa. 2024. MCMPI: A library with elasticity for multi-domain and public cloud environments. *Concurrency and Computation: Practice and Experience* (2024).

[2] S. Akioka and Y. Muraoka. 2010. HPC Benchmarks on Amazon EC2. In *24th IEEE International Conference on Advanced Information Networking and Applications Workshops*. 1029–1034.

[3] S. R. Alam, M. Gila, M. Klein, M. Martinasso, and T. C. Schulthess. 2023. Versatile software-defined HPC and cloud clusters on Alps supercomputer for diverse workflows. *The International Journal of High Performance Computing Applications* 37, 3-4 (2023), 288–305.

[4] Amazon Web Services (AWS). 2024. *AWS ParallelCluster - HPC for the Cloud.* https://github.com/aws/aws-parallelcluster

[5] Ansible. 2024. *Ansible Collaborative.* https://www.ansible.com

[6] The Kubernetes Authors. 2024. *Kubernetes.* https://kubernetes.io

[7] Amazon Web Services (AWS). 2024. *Amazon Elastic Cloud Computing (EC2).* https://aws.amazon.com/ec2

[8] Amazon Web Services (AWS). 2024. *AWS Cloud Formation.* https://aws.amazon.com/cloudformation/

[9] Amazon Web Services (AWS). 2024. *High Performance Coputing.* https://aws.amazon.com/hpc

[10] Microsoft Azure. 2024. *Azure high-performance computing.* https://azure.microsoft.com/pt-br/solutions/high-performance-computing/

[11] D. H. Bailey and et al. 1991. The NAS Parallel Benchmarks. *International Journal of Supercomputing Applications* 5, 3 (1991), 63–73.

[12] T. Ben Nun and T. Hoefler. 2019. Demystifying Parallel and Distributed Deep Learning: An In-depth Concurrency Analysis. *ACM Computung Surveys* 52, 4 (Aug. 2019), 65:1–65:43.

[13] J. L. F. Betting, C. I. De Zeeuw, and C. Strydis. 2023. Oikonomos-II: A Reinforcement-Learning, Resource-Recommendation System for Cloud HPC. In *30th IEEE International Conference on High Performance Computing, Data, and Analytics (HiPC)*. 266–276.

[14] J. Bezanson, J. Chen, B. Chung, S. Karpinski, V. B. Shah, J. Vitek, and L. Zoubritzky. 2018. Julia: Dynamism and Performance Reconciled by Design. *Proceedings of ACM Programming Languages* 2, OOPSLA, Article 120 (oct 2018), 23 pages.

[15] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. 2017. Julia: A Fresh Approach to Numerical Computing. *SIAM Review* 59, 1 (2017), 65–98.

[16] E. Borin, L. M. A. Drummond, Gaudiot J-L., A. Melo, M. M. Alves, and P. O. A. Navaux. 2023. *High Performance Computing in Clouds: Moving HPC Applications to a Scalable and Cost-Effective Environment.* Springer.

[17] Terraform Community. 2024. *Automate infrastructure on any cloud with Terraform.* https://www.terraform.io

[18] AWS.jl contributors. 2024. *Julia Interface for AWS.* https://github.com/JuliaCloud/AWS.jl

[19] Flux.jl contributors. 2024. *Flux: The Julia Machine Learning Library.* https://fluxml.ai/Flux.jl

[20] F. H. de Carvalho Junior, W. G. Al Alam, and A. B. de O. Dantas. 2021. Contextual Contracts for Component-Oriented Resource Abstraction in a Cloud of High Performance Computing Services. *Concurrency and Computation: Practice and Experience* 33, 18 (2021), e6225.

[21] F. H. de Carvalho Junior, A. B. Dantas, J. M. Hoffiman, T. Carneiro, C. S. Sales, and P. A. S. Sales. 2023. Structured Platform-Aware Programming. In *XXIV Simpósio em Sistemas Computacionais de Alto Desempenho (SSCAD'2023)* (Porto Alegre, RS). SBC, Porto Alegre, Brazil, 301–312.

[22] NASA Advanced Supercomputing (NAS) Division. 2024. *NAS Parallel Benchmarks.* https://www.nas.nasa.gov/software/npb.html

[23] J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, and A. White. 2003. *Sourcebook of Parallel Computing.* Morgan Kauffman publishers.

[24] H. Jin, D. Jespersen, P. Mehrotra, R. Biswas, L. Huang, and B. Chapman. 2011. High performance computing using MPI and OpenMP on multi-core parallel systems. *Parallel Computing* 37, 9 (2011), 562–575.

[25] H. Jin and R. F. Van der Wijngaart. 2006. Performance characteristics of the multi-zone NAS parallel benchmarks. *Journal of Parallel and Distributed Computing* 66, 5 (2006), 674–685. IPDPS'04 Special Issue.

[26] P. Mehrotra, J. Djomehri, S.e Heistand, R. Hood, H. Jin, A. Lazanoff, S. Saini, and R. Biswas. 2012. Performance evaluation of Amazon EC2 for NASA HPC applications. In *Proceedings of the 3rd Workshop on Scientific Cloud Computing* (Delft, The Netherlands). ACM, New York, NY, USA, 41–50.

[27] H. Meuer, E. Strohmaier, J. Dongarra, and H. D. Simon. 2013. *Top 500 Supercomputer sites.* http://www.top500.org

[28] Microsoft Azure. 2024. *Eagle - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR.* https://top500.org/system/180236/

[29] V. Munhoz and M. Castro. 2024. Enabling the execution of HPC applications on public clouds with HPC@Cloud toolkit. *Concurrency and Computation: Practice and Experience* 36, 8 (2024), e7976.

[30] Riccard Murri. 2024. *ElastiCluster.* https://github.com/elasticluster/elasticluster

[31] O. O. Napoli, R. K. Tesser, D. L. Fonseca, and E. Borin. 2023. *Cost Effective Deep Learning on the Cloud.* Springer, 283–307.

[32] F. Z. Nardelli, J. Belyakova, A. Pelenitsyn, B. Chung, J. Bezanson, and J. Vitek. 2018. Julia Subtyping: A Rational Reconstruction. *Proceedings of the ACM Programming Languages* 2, Article 113 (oct 2018), 27 pages.

[33] M. A. S. Netto, R. N. Calheiros, E. R. Rodrigues, R. L. F. Cunha, and R. Buyya. 2018. HPC Cloud for Scientific and Business Applications: Taxonomy, Vision, and Research Challenges. *ACM Computing Surveys* 51, 1 (Jan. 2018), 1–29.

[34] A. Pelenitsyn, J. Belyakova, B. Chung, R. Tate, and J. Vitek. 2021. Type Stability in Julia: Avoiding Performance Pathologies in JIT Compilation. *Proceedings of ACM Programmming Languages* 5, OOPSLA, Article 150 (oct 2021), 26 pages.

[35] Google Cloud Platform. 2024. *High Performance Coputing.* https://cloud.google.com/solutions/hpc

[36] P. Vaillancourt, B. Wineholt, B. Barker, P. Deliyannis, J. Zheng, A. Suresh, A. Brazier, R. Knepper, and R. Wolski. 2020. Reproducible and Portable Workflows for Scientific Computing and HPC in the Cloud. In *Practice and Experience in Advanced Research Computing* (Portland, OR, USA) *(PEARC'20)*. Association for Computing Machinery, New York, NY, USA, 311–320.