Swirls: A Platform for Enabling Multicluster and Multicloud Execution of Parallel Programs

Francisco Heron de Carvalho Junior[1], **Allberson Bruno de Oliveira Dantas**[2], **Claro Henrique Silva Sales**[1]

[1]Pós-Graduação em Ciência da Computação (MDCC)
Universidade Federal do Ceará (UFC)
Campus Universitário do Pici, Bloco 912 – Fortaleza – CE – Brazil

[2]Instituto de Engenharias e Desenvolvimento Sustentável
Universidade da Integração Internacional da Lusofonia Afro-Brasileira (Unilab)
Redenção – CE – Brazil

hpcshelf@dc.ufc.br

Abstract. *Swirls is a general purpose application for interactive building, deploying, and execution of message-passing parallel programs that address multicluster and multicloud requirements. It is implemented on HPC Shelf, a cloud-based platform for providing HPC services. Swirls enables the communication between MPI programs written in C#, C, C++, and Python across one or more clusters, either on-premise or cloud-based ones. At the current implementation status, The users of Swirls may use clusters formed by virtual machines over Amazon Elastic Compute Cloud (EC2) and Google Cloud Platform (GCP).*

1. Introduction

Cloud computing is now an alternative to the acquisition of large servers by organizations. The pay-as-you-go model facilitates continuous provisioned access to virtualized software and hardware, making it a viable alternative to a range of applications, notably those with an intermittent workload. Given the high demand for large-scale processing stemming from emerging applications (e.g., Big Data and Deep Learning), High Performance Computing (HPC) techniques and technologies have spread at a rapid pace [1].

The seasonality of the use of dedicated computing resources in a datacenter or private cluster severely impacts HPC applications. However, traditional HPC developers are still reluctant to move their services to the cloud, based on the assumption of maintaining full control over the resources offered by on-premises clusters and because they believe processor virtualization and interconnects by IaaS providers degrade performance. However, the second justification is not observed in practice today. By consequence, many new HPC developers have increasingly invested in cloud-based HPC, leveraging self-scaling strategies and geographical proximity between data and applications.

A recurring difficulty in HPC development consists in mastering parallelism techniques and the heterogeneous features of parallel architectures. This difficulty was increased by the advent of accelerators (GPUs, FPGAs, TPUs, etc.) and multicore/many-core processors, inaugurating the years of heterogeneous computing as a way to achieve exascale parallel computing systems [2]. In this sense, several research initiatives have led computer scientists to propose abstractions that reduce the programming burden and the

need for intrinsic knowledge of parallel architectures. This abstraction power has proven to be greater when it comes to cloud HPC since the complexity of abstracting certain parallel programming artifacts and the programmability over HPC architectures can be built into the virtualization layers of the IaaS infrastructure.

HPC Shelf[1] [3, 4] is an open platform proposal for providing services for the creation, deployment, and execution of component-oriented *parallel computing systems* aimed at large-scale parallel processing, i.e., employing several parallel computing platforms across different computational infrastructures, such as IaaS providers, HPC/supercomputing centers, on-premise clusters, etc. Through component-orientation features, especially a contextual contract system for component selection [4], applications that use HPC Shelf services may deal with heterogeneous computing requirements.

This paper introduces Swirls, an HPC Shelf application for interactive building, deployment, and execution of parallel computing systems comprising a set of MPI (Message Passing Interface) [5] programs running in distinct clusters possibly deployed at distinct computational infrastructure, by means of a command-line interface implemented for Jupyter notebooks and Linux shell. Through Swirls, these MPI programs may interact by means of message-passing connectors, enabling inter-cluster parallelism.

Swirls is an alternative to introduce multicloud and multicluster parallel computing for MPI programmers, using the same message-passing programming model in which they are specialized, by leaving all concerns about multicloud/multicluster deployment at the level of a command-line interface (CLI), currently supported by Jupyter notebooks and Linux shell. At present, it includes multicloud support for two popular IaaS providers: Amazon Elastic Compute Cloud (EC2) and Google Cloud Platform (GCP). Also, it supports the following programming languages: C, C++, Python, and C#. However, it can be extended to support other IaaS providers and programming languages that support MPI.

This paper comprises more five sections. Related works are presented in Section 2. Section 3 presents HPC Shelf, the HPC services platform on which Swirls has been implemented. Section 4 introduces Swirls, and two case studies to demonstrate its main features. Finally, Section 5 presents the final considerations about this research work.

2. Related Works

To find works related to Swirls, we carried out a bibliographic search of articles that propose platforms or frameworks for parallel programming aimed at multicluster or multicloud environments. For that, we have applied the search string "*(platform OR framework) AND parallel AND (multicluster OR multicloud)*" to ACM, ScienceDirect and IEEE databases for the last 5 years. From a total of 138 articles, we eliminated those whose title did not refer to the proposal of a platform or framework. The second round consisted of reading the abstracts of the remaining 32 articles and eliminating those that did not propose platforms or frameworks for multicluster/multicloud parallel programming. The remaining 9 works are divided into two groups, described in the following sections.

2.1. Management of multicluster and multicloud executions

The first group reports platforms or frameworks that propose strategies for managing executions on multicluster and multicloud, as well as other concerns, such as execution

[1] `http://www.hpcshelf.org`

plans, provisioning, deployment and fault tolerance. Flouris *et al* introduce FERARI, a prototype for complex events processing (CEP) over large volumes of data on distributed platforms [6]. It also brings an optimizer in the execution plan for minimizing inter-cloud communication latency, as well as query and visualization tools. Ferry *et al* propose the Cloud Modeling Framework (CLOUDMF), a model-oriented platform focused on DevOps issues [7]. It provides a domain-specific language for provisioning and deploying multicloud applications and an execution engine aimed at provisioning, deploying and adaptation. Wu *et al* proposes HDM-MC, a framework for big data processing capable of performing large-scale data analysis on multiclusters [8]. The authors argue for the minimal overhead incurred by the platform's scaling requirements. Maheshwari *et al* propose a multicloud scheduling workflow technique based on an execution performance model that takes into account the available resources and dynamic probes to assess throughput between clouds in execution [9]. Fakih and El Baz present a Peer-To-Peer HPC decentralized environment for the engagement of heterogeneous multiclusters in loosely synchronous applications[10]. Finally, Mosa *et al* introduce a scalable multicluster platform based on Hadoop [11]. The multicloud orchestration is provided through the MiCADO framework, aiming at the deployment and horizontal scaling of cloud resources.

2.2. PaaS for multicluster and multicloud

The two works found in this group report platforms for MapReduce over multiple clouds. Costa *et al* introduce Chrysaor, an execution system capable of scaling out MapReduce computations over multiclouds[12]. It allows detection of arbitrary failures, malicious failures and cloud outages. They also propose Medusa, a platform aimed at running MapReduce applications over multiclouds, capable of detecting different types of failures, in addition to not requiring modification of the original MapReduce code [13].

Swirls is innovative in two aspects. Firstly, it supports both multicluster and multicloud requirements. Secondly, it is general-purpose and not restricted to some parallel programming models, such as Bag-of-Tasks, MapReduce, or stream processing patterns.

3. HPC Shelf

HPC Shelf is an open platform proposal for providing services for the creation, deployment, and execution of component-oriented *parallel computing systems* [3, 4]. It uses the Hash component model as a basis [14]. The services are consumed by *applications*, each serving a community of domain specialists. In fact, parallel computing systems implement computationally demanding solutions to problems described by domain experts, requiring large-scale parallel processing, i.e. involving multiple parallel computing platforms, such as clusters and MPPs. These platforms can be deployed on the infrastructure of IaaS providers or HPC/Supercomputing centers serving academia and industry.

HPC Shelf offers SAFe (Shelf Application Framework) [3] to assist applications in building parallel computing systems, currently implemented as a C# API, but it does not prescribe the kind of interface that applications should offer to interact with domain specialists. Thus, applications may use different kinds of high-level interfaces, such as web portals, problem solving environments (PSEs), application programming interfaces (APIs), command-line interfaces (CLIs), Jupyter notebooks, and so on. In addition, the application interface may abstract away from the concrete nature of parallel computing

systems, by providing high-level abstractions to facilitate the description of problems by domain specialists, as well as possibly providing ways for domain specialists to help find the best computational solutions implemented by parallel computing systems.

3.1. Parallel Computing Systems

The components that comprise a parallel computing system are: a single *workflow* component, a single *application* component, and a set of *solution components*. Solution components may belong to one of the following component kinds: ***virtual platforms***, representing distributed-memory parallel computing platforms; ***data sources***, representing data repositories, possibly distributed, from which data that interest to applications may be retrieved; ***computations***, representing implementations of parallel algorithms that exploits the features of a class of virtual platforms; ***connectors***, which couple computations and data sources placed in distinct virtual platforms; ***service bindings***, which connect pairs of a *user* and a *provider port* belonging to components of any kind, so that a user component may consume a service offered by a provider one; ***action bindings***, which bind *action ports* that may belong to computations, connectors and the workflow component; and ***qualifiers***, used in contextual contracts to represent functional and non-functional assumptions, as well as assumptions in the implementation of components.

The application component is an abstraction to the application frontend. It intermediates the communication between the solution components and the application through service bindings, for providing inputs, receiving outputs, managing intermediary data, monitoring solution components, etc.

The workflow component drives the execution of computations and connectors through action bindings that bind their action ports to the action ports of the workflow component. Action ports binded through the same action binding carry a common set of *action names*. Components activate actions by referring to action names. An activation of an action name n in a given port p remains blocked until a pending activation of n exists in each port binded to p through an action binding.

Connectors may *orchestrate* computations and other connectors through action bindings, just as the workflow component, or they may support choreographs among computations, data sources, and other connectors through service bindings. For that, they comprise a set of *facets*, each one placed in a virtual platform, allowing direct communication through service bindings with other components.

3.2. Contextual Contracts

In parallel computing systems, solution components are associated to *contextual contracts*, which guide the selection of appropriate component implementations according to a set of *contextual assumptions*, including the features of the target parallel computing platform and application requirements, as well as Quality-of-service (QoS) and cost constraints [4]. A contextual contract is the application of the *context signature* of an *abstract component* to a set of contextual contracts. An abstract component represents a collection of components that implement a given concern under a set contextual assumptions represented by *contextual parameters* (this is contextual signature). A contextual parameter has a name and a bound, which is a contextual contract that constraints the contextual contracts that may be applied to the contextual parameter. For that, there is a compati-

bility relation between contextual contracts, specified by means of subtyping rules. For a detailed description of Alite, the contextual contract system of HPC Shelf, read [4].

3.3. Stakeholders

There are four stakeholder in HPC Shelf. Domain ***specialists*** are the end-users. Using applications, they do not need to be aware of the component nature of resources, neither that computational solution will execute over a parallel computing infrastructure. Application ***providers*** have expertise on the best computational solutions for the problems specified by specialists. So, they know which components provide the necessary software and hardware resources to assemble the appropriate parallel computing system to solve each problem. However, they are not experts in parallel computing, contrariwise to component ***developers***, which have parallel programming skills and know how to exploit the performance of virtual platforms efficiently. Platform ***maintainers*** offer parallel computing infrastructures for deploying virtual platforms. Finally, data ***managers*** offer large data repositories that interest applications.

3.4. Architecture

The architecture of HPC Shelf has three elements: *Frontend*, *Core*, and *Backend*. The *Frontend* is SAFe (*Shelf Application Framework*) [3], offered to application providers as a C# API for building, deploying, and running parallel computing systems. In turn, the *Core* implements the component catalog, where developers and maintainers register components. Also, it implements Alite, the contextual contract system. Through SAFe, applications may access the *Core* services for taking the components of parallel computing systems, by resolving their contextual contracts and controlling their lifecycle. Once instantiated, applications directly orchestrate components. Finally, a *Backend* is a service that a *maintainer* offers to the *Core* for instantiating virtual platforms over a parallel computing infrastructure. Virtual platforms may communicate directly with *Core* for instantiating components. *Backend* services for providing access to large data repositories may also be possible. In fact, data source components are implemented as virtual platforms from which data may be accessed through service ports.

4. Swirls

Swirls is a general-purpose application of HPC Shelf aimed at interactive building, deployment, and execution of parallel computing systems comprising MPI programs in cloud-based computational infrastructures, by means of a command-line interface (CLI) currently implemented in Jupyter notebooks and Linux shell. So, each MPI program may run in its own cluster (virtual platform) with a set of characteristics specified in a contextual contract, including a IaaS provider. Currently, EC2 and GCP are supported. Also, the MPI programs may interact through message-passing connectors offered by the Swirls framework for enabling inter-cluster parallelism, which aggregates the computational power of an heterogeneous set of virtual platforms.

For supporting EC2 and GCP, Swirls offers two component frameworks to build virtual platforms from any instance/machine type supported by these IaaS providers until the end of 2020. Using the same approach, one can introduce other IaaS providers with some API support with a relatively low effort.

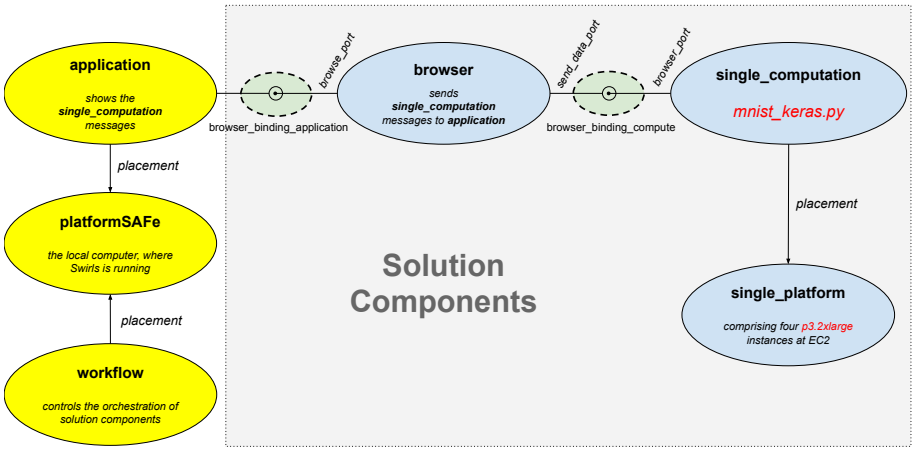

Figure 1. A Simple MPI Launcher Parallel Computing System

The main contribution of Swirls is to simplify the access to multicluster and multicloud parallel computation capabilities to MPI programmers, without being limited to specific parallel computing patterns, such as Bag-of-Tasks, MapReduce, and their extensions. In addition, it inherits the component-oriented features of HPC Shelf, which are useful for the architecture of large-scale parallel computing systems.

We use two examples for demonstrating the features of Swirls. First, **Horovod/MNIST** shows how to launch a simple MPI program in a single cluster by choosing the target IaaS cloud provider among the ones supported by Swirls. In turn, **GEMM** demonstrates the launching of a multicluster parallel program comprising a set of MPI programs deployed at different clusters across distinct cloud providers. **Horovod/MNIST** and **GEMM** are described in the next two sections.

The source codes of these case studies may be found are accessible from a *supplementary material folder*, at `https://gitlab.com/carvalho.heron/supplementary-material/-/tree/master/WSCAD2021-Swirls`.

4.1. Horovod/MNIST: Distributed Deep Learning using Horovod

Figure 1 depicts a parallel computing system for **Horovod/MNIST**, comprising three solution components: **single_computation**, of kind computation, encapsulating an MPI program; **single_platform**, of kind virtual platform, where **single_computation** will run; and **browser**, of kind connector, that allows the MPI program to send messages to the Swirls frontend (i.e., the Jupyter notebook).

Any MPI program could be encapsulated in **single_computation**. In this case study, it is a Python/MPI program, called keras_mnist.py, that performs a well-known deep learning computation: recognition of digits from the MNIST database[15]. For that, it employs Horovod [16], a distributed deep learning framework, which has been installed in the virtual machine images used by EC2 and GCP backend services.

keras_mnist.py is one of the examples offered by Horovod. Minor modifications in the original code were necessary, including calls to the `browse` subroutine for sending progress messages to the Swirls frontend through the **browser** connector.

The command "`new system horovod_example`" creates the initial system. Then,

the following command creates **single_platform** as a cluster of four EC2 instances with Tesla V100 GPUs and at least 32GB of memory, located at us-east-1 (Virginia) locale:

```
new platform single_platform
    --contract=name=org.hpcshelf.platform.Platform;
               maintainer=org.hpcshelf.my.EC2_backend;
               node-locale=org.hpcshelf.platform.locale.northamerica.Virginia;
               node-accelerator-model=
                              org.hpcshelf.platform.node.accelerator.model.Tesla_V100;
               node-memory-size=32
               node-count=4
```

In EC2, `p3.2xlarge` are the accelerated computing instances that supports the required kind of GPU (up to 8 per instance). Also, it has 61GB of memory, satisfying the required 32GB. So, the contextual contract system will find a component called `EC2_P3_Large2x`, among the subtypes of `org.hpcshelf.platform.Platform`, for **single_platform**. Alternatively, the user could refer to `EC2_P3_Large2x` directly in the `name` parameter. However, using the indirect approach, if the user decides to move the computation from EC2 to GCP subject to same platform constraints, it may only change the value of `maintainer` to `org.hpcshelf.my.GCP_backend`. In this case, for example, the contextual contractual resolution system will select a cluster of GCP instances of machine type `a2-highgpu-1g` located at `us-east4` region, each one with 85GB of memory.

The following command will encapsulate keras_mnist.py into a component named `org.hpcshelf.Horovod_MNIST`, becoming available in the *Core*'s catalog:

```
create computation org.hpcshelf.Horovod_MNIST --source=keras_mnist --language=Python
```

The `---language` flag specifies that Python is the language in which the wrapped MPI program is written, so that the name of the MPI source code pointed by the `---source` flag is `keras_mnist.py`. An instance of `org.hpcshelf.Horovod_MNIST`, the so-called **single_computation**, may be created using the following command:

```
new computation single_computation --contract=name=org.hpcshelf.Horovod_MNIST
                                   --platform=single_platform
```

It specifies that **single_computation** will run in **single_platform** by means of the `---platform` flag. The contract specified through `---contract` is the simplest possible, only specifying the component type of **single_computation** through the `name` context parameter. It could also include platform context parameters to constraint the required features of the target virtual platform, such as the use of accelerators, processor architectures, interconnection type, etc. However, this is not so useful in a single Swirls configuration, where platform constraints are all specified in the contract of virtual platforms.

The following commands instantiate the connector **browser** and the service bindings **browser_binding_application** and **browser_binding_compute**, respectively:

```
new connector browser --contract=name=org.hpcshelf.mpi.wrapper.WBrowserConnector
                      --platform=local:0 --platform=single_platform:1

new service-binding browser_binding_application
   --contract=name=org.hpcshelf.common.BrowserBinding;
              browser_port_type=org.hpcshelf.common.browser.RecvDataPortType
   --user-port=application --provider-port=browser.browse_port

new service-binding browser_binding_computation
   --contract=name=org.hpcshelf.common.BrowserBinding;
              browser_port_type=org.hpcshelf.common.browser.wrapper.WSendDataPortType
   --user-port=single_computation.browser_port --provider-port=browser.send_data_port
```

Once the parallel computing system is configured, the lifecycle operation for resolving, deploying and instantiating the solution components may be executed:

```
resolve single_platform single_computation browser
        browser_binding_application browser_binding_computation

deploy single_platform single_computation browser
       browser_binding_application browser_binding_computation

instiantiate single_platform single_computation browser
             browser_binding_application browser_binding_computation
```

Figure 2 shows the screenshot of a notebook fragment including the `browse` and `run` commands for this case study. The `browse` command waits for messages sent by **single_computation** through the **browser** connector. For that, the name of the binding that connects the application to the **browse_port** of **browser**, i.e., **browse_binding_application**, follows the `browse` keyword.

The browse messages appear as outputs of the `browse` command as they are received by the application. Since `browse` and `run` must execute concurrently, one of them must be executed asynchronously (`--async` flag).

```
In [45]: browse --async=_ browser_binding_application
Out[49]: 1:  60000 train samples
Out[50]: 1:  10000 test samples
Out[51]: 1:  x_train shape: (60000, 28, 28, 1)
Out[52]: 0:  x_train shape: (60000, 28, 28, 1)
Out[53]: 0:  60000 train samples
Out[54]: 0:  10000 test samples

In [46]: run single_computation browser
Out[47]: The component 'browser' of system 'testing_horovod' is running.
Out[48]: The component 'single_computation' of system 'testing_horovod' is running.
```

Figure 2. Browse and Run Horovod/MNIST

4.2. GEMM: Multicluster Matrix Multiplication of General Matrices

This case study demonstrates the parallel execution of MPI programs over multicloud infrastructure in order to support multicluster parallel computing. For that, we have developed a MPI program called GEMM. Inspired in the GEMM subroutines of BLAS (level 3) [17], it performs parallel multiplication of general matrices in three independent parallelism levels: *multicluster*, across virtual platforms; *cluster*, across the nodes of virtual platforms; and *multicore*, across processor cores of a single node. For enabling multicluster parallelism, it is encapsulated in a computation component of HPC Shelf.

The GEMM component performs $\mathbf{C} = \beta \times \mathbf{C} + \alpha \times \mathbf{A} \times \mathbf{B}$, where α and β are scalars, and $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are matrices of dimensions $M \times N$, $N \times P$, $M \times P$, respectively. A multicluster matrix multiplication will be performed by a cohort of GEMM instances organized in a $X \times Y$ grid ($\mathbf{gemm}\_r\_c$, for $r \in \{0 \ldots X-1\}$ and $c \in \{0 \ldots Y-1\}$), each one deployed in a distinct virtual platform ($\mathbf{platform}\_r\_c$), by assuming that input matrices are block-cyclically distributed across them. The same parallelism strategy and matrix distribution are applied to the cluster parallelism level. The dimensions of blocks for each input matrix are, respectively: $m \times n$, $p \times n$, and $m \times p$. For simplifying the

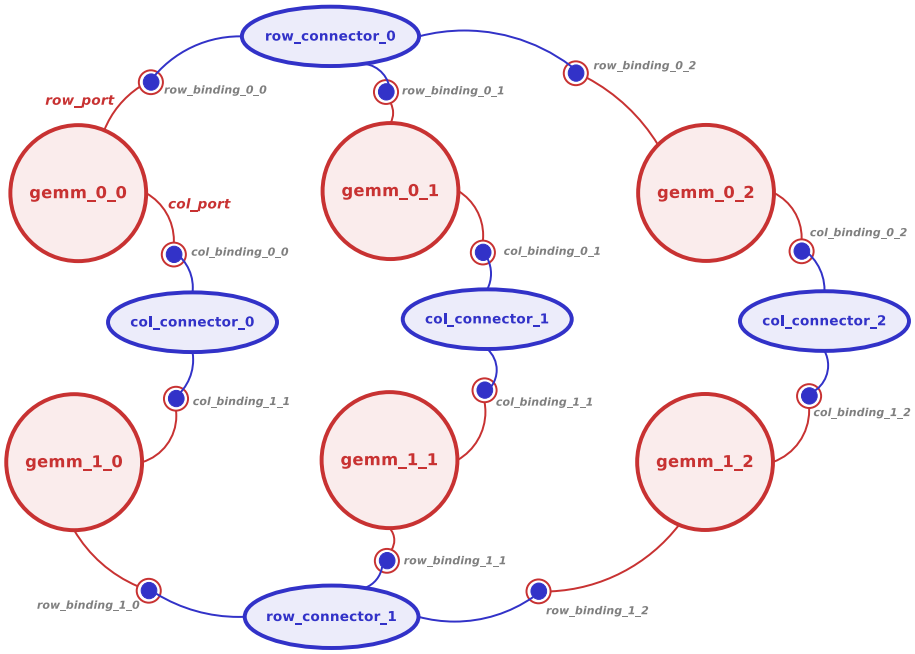

Figure 3. A 2×3 GEMM System

implementation, we assume that m, n, and p are divisible by M, N, and P, respectively. So, inside a process (MPI program), a matrix of blocks is stored for each input matrix.

The multicluster GEMM is useful in a context where the dimensions of the matrices are so large to fit a single computer (or even a local cluster accessible by the user), so that the user may decide to employ one or more clusters deployed at IaaS providers to perform its computation. In addition, the user may overcome restrictions on the number of nodes it is possible to instantiate in a cluster of a particular IaaS provider by employing clusters deployed across different providers or locations within the same provider.

Figure 3 depicts a GEMM system (multicluster level), comprising 6 GEMM instances organized in a 2×3 grid of virtual platforms. In a $X \times Y$ grid of processes, each $\mathbf{gemm}\_r\_c$ component, placed on $\mathbf{platform}\_r\_c$, is connected to two intercommunicator connectors: one for communication with processes in its row, through the user port **row_port**, and another one for communication with processes in its column, through the user port **col_port**. For that, $X + Y$ connectors are necessary, named $\mathbf{row\_connector}\_r$, for $r \in \{0, \ldots, X-1\}$, and $\mathbf{col\_connector}\_c$, for $c \in \{0, \ldots, Y-1\}$.

Let A, B, C be the matrices of blocks of matrices $\mathbf{A}$, $\mathbf{B}^T$, and $\mathbf{C}$, respectively, stored in the virtual platform $\mathbf{platform}\_r\_c$, having the following dimensions, respectively: $M/X \times N/Y$, $P/X \times N/Y$, and $M/X \times P/Y$. The block-cyclic algorithm performs X steps. In step i, $0 \leq i < X$, it first performs a local matrix multiplication $C' = \alpha \times A \times B$, where C' intermediary matrix with dimensions $M/X \times P/X$. In fact, this is a GEMM computation at the cluster parallelism level, using the same algorithm. Then, it performs two communication operations. The first is a sequence of Y reductions involving $\mathbf{gemm}\_i\_c$, for $0 \leq c < Y$, illustrated in Figure 4 for 2×3 grid, for accumulating the partial C' matrix in the global matrix C. The second are Y simultaneous 1-shift rotations, for each column of the grid. Indeed, in rotation j, for $0 \leq j < Y$, each

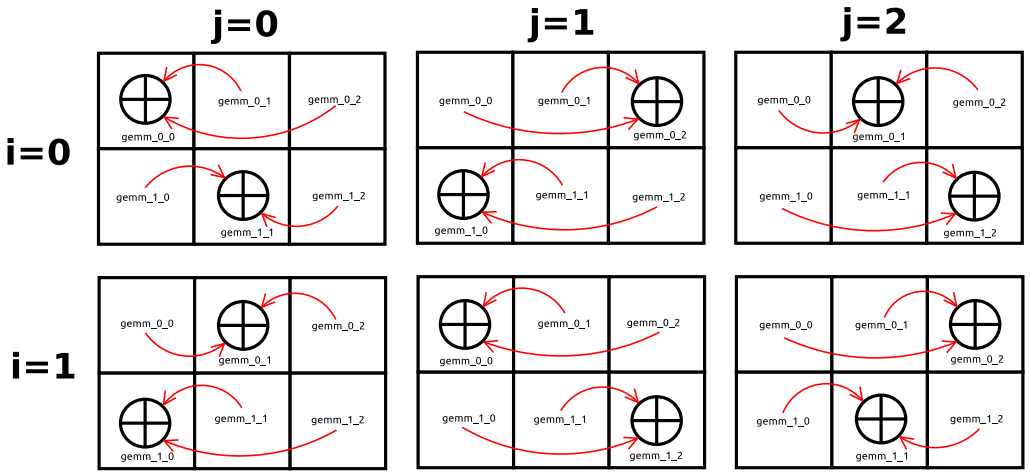

Figure 4. Rowwise reduction stages of C' matrix blocks

$\mathbf{gemm}\_k\_j$, for $0 \leq k < X$, sends its B block to $\mathbf{gemm}\_k'\_j$, where $k' = k+1 \bmod X$. The reduction steps are performed through the $\mathbf{row\_connector}\_*$ connectors, whereas the rotation steps are performed through the $\mathbf{row\_connector}\_*$ ones. Currently, such connectors offer basic message-passing primitives whose C signatures are:

```
int HPCShelf_Send(void *buf, int count, MPI_Datatype datatype, int target_facet,
                  int target_rank, int tag, HPCShelf_Connector conn);

int HPCShelf_Recv(void *buf, int count, MPI_Datatype datatype, int source_facet,
                  int source_rank, int tag, HPCShelf_Connector conn);
```

In the near future, we plan to implement collective message-passing operations like that ones supported by MPI, for promoting structured parallel programming and increasing the performance of complex communication patterns.

The signatures of `HPCShelf_Send` and `HPCShelf_Recv` are clearly inspired in MPI, but requiring two parameters for making reference to communication partners: *target_facet*/*target_rank* and *source_facet*/*source_rank*, respectively. The facet parameter refers to one of the MPI programs, whereas rank refers to a process of the MPI program. The `conn` parameter refer to an intercommunicator connector, making the role of MPI communicators. The handle for making reference to the intercommunicator connector may be obtained by calling `HPCShelf_Get_Port`, passing the name of the connector in the parallel computing system as an argument to the parameter port:

```
int HPCShelf_Get_port(char* port_name, HPCShelf_Port* port);
```

There are operations for querying connector configurations, through which one can get the local facet of the calling process, the number of facets of the connector, the number of processes in each facet, and the global facets of connector partners. They are:

```
int HPCShelf_Get_facet(HPCShelf_Port port, int* facet);
int HPCShelf_Get_facet_count(HPCShelf_Port port, int* facet_count);
int HPCShelf_Get_facet_size(HPCShelf_Port port, int* facet_size);
int HPCShelf_Get_facet_instance(HPCShelf_Port port, int* facet_instance);
```

In HPC research community, matrix multiplication is commonly considered in trivial case studies of parallel programming. However, in parallel processing beyond the limits of a single cluster, where loosely coupled distributed memory parallel programming

patterns are common alternatives, such as Bag-of-Tasks, MapReduce, and stream-based processing (see related works), the two-level (inter-cluster and intra-cluster) block-cyclic algorithm of GEMM, where the process topology may differ in each cluster, is challenging due to the relatively complex communication pattern and volume of data in communication. In fact, GEMM was chosen for this case study because it proves to be an appropriate alternative for exercising Swirls's multicluster and multicloud deployment capabilities for general-purpose message-passing parallel programming.

In the supplementary material, we provide the configuration of a 2×2 GEMM parallel computing system, where **gemm**_0_0 and **gemm**_1_1 components perform 2×2 intra-cluster GEMM computations over GCP cluster formed by `n2-standard-2` virtual machine instances, and **gemm**_0_1 and **gemm**_1_0 performs 2×3 intra-cluster GEMM computations over EC2 clusters formed by `t2.micro` vm instances. So, the four clusters instantiate 20 virtual machines at all (4 for each **platform**_0_0 and **platform**_1_1, and 6 for each **platform**_0_1 and **platform**_1_0). One may note that these are modest cluster configurations. In fact, they are the cheaper ones for EC2 and GCP, chosen only for testing and concept validation. The matrices are generated automatically at each cluster node. Indeed, they have been scaled to fit the available memory in the virtual machines.

5. Conclusions

Swirls is a contribution to HPC users interested in moving their MPI code to take advantage of multicluster and multicloud capabilities through a programming interface they already know (message-passing), without being restricted to particular parallel programming models. Indeed, all the concerns about multicluster/multicloud deployment is moved to the level of a command-line interface (CLI), supported by Jupyter notebooks and Linux shell. Besides to run parallel programs such as the ones presented in the case studies, we are successfully applying Swirls as an auxiliary tool in teaching parallel programming for undergraduate and graduate students in a traditional HPC course, giving the opportunity of introducing HPC in cloud computing platforms to students.

Swirls is a tool under continuous development, with the purpose of extending features, such as the support for new IaaS providers and programming languages. Also, we are working on implementing realistic use cases (e.g. deep learning and multiphysics simulation) with the purpose of attracting interested users, developers and research partners.

Swirls is freely available to users and developers interested in simply using it or even modifying it to satisfy non-commercial needs. Information on how to install and use Swirls is available from the HPC Shelf web site, especially by visiting `https://www.hpcshelf.org/#h.8eglmidqf5uh`.

References

[1] M. A. S. Netto, R. N. Calheiros, E. R. Rodrigues, R. L. F. Cunha, and R. Buyya, "HPC Cloud for Scientific and Business Applications: Taxonomy, Vision, and Research Challenges," *ACM Computing Surveys*, vol. 51, no. 1, pp. 1–29, Jan. 2018. [Online]. Available: http://doi.acm.org/10.1145/3150224

[2] M. Zahran, "Heterogeneous Computing: Here to Stay," *Communications of the ACM*, vol. 60, no. 3, pp. 42–45, Feb. 2017. [Online]. Available: http://doi.acm.org/10.1145/3024918

[3] F. H. de Carvalho Junior, J. C. Silva, and A. B. O. Dantas, "A Scientific Workflow Management System for Orchestration of Parallel Components in a Cloud of Large-Scale Parallel Processing Services," *Science of Computer Programming*, vol. 173, pp. 95–127, Mar. 2019.

[4] F. H. de Carvalho Junior, W. G. Al Alam, and A. B. O. Dantas, "Contextual Contracts for Component-Oriented Resource Abstraction in a Cloud of High Performance Computing Services," *Concurrency and Computation: Practice and Experience*, vol. 33, no. 18, p. e6225. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.6225

[5] J. Dongarra, S. W. Otto, M. Snir, and D. Walker, "A Message Passing Standard for MPP and Workstation," *Communications of ACM*, vol. 39, no. 7, pp. 84–90, 1996.

[6] I. Flouris, V. Manikaki, N. Giatrakos, A. Deligiannakis, M. Garofalakis, M. Mock, S. Bothe, I. Skarbovsky, F. Fournier, M. Stajcer, T. Krizan, J. Yom-Tov, and T. Curin, "Ferari: A prototype for complex event processing over streaming multi-cloud platforms," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 2093–2096. [Online]. Available: https://doi.org/10.1145/2882903.2899395

[7] N. Ferry, F. Chauvel, H. Song, A. Rossini, M. Lushpenko, and A. Solberg, "Cloudmf: Model-driven management of multi-cloud applications," *ACM Trans. Internet Technol.*, vol. 18, no. 2, Jan. 2018. [Online]. Available: https://doi.org/10.1145/3125621

[8] D. Wu, S. Sakr, L. Zhu, and H. Wu, "Towards big data analytics across multiple clusters," in *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, ser. CCGrid '17. IEEE Press, 2017, p. 218–227. [Online]. Available: https://doi.org/10.1109/CCGRID.2017.73

[9] K. Maheshwari, E.-S. Jung, J. Meng, V. Morozov, V. Vishwanath, and R. Kettimuthu, "Workflow performance improvement using model-based scheduling over multiple clusters and clouds," *Future Generation Computer Systems*, vol. 54, pp. 206–218, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X15000795

[10] B. Fakih and D. El Baz, "Heterogeneous computing and multi-clustering support via peer-to-peer hpc," in *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, 2018, pp. 292–296.

[11] A. Mosa, T. Kiss, G. Pierantoni, J. DesLauriers, D. Kagialis, and G. Terstyanszky, "Towards a cloud native big data platform using micado," in *2020 19th International Symposium on Parallel and Distributed Computing (ISPDC)*, 2020, pp. 118–125.

[12] P. A. R. S. Costa, F. M. V. Ramos, and M. Correia, "Chrysaor: Fine-grained, fault-tolerant cloud-of-clouds mapreduce," in *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, ser. CCGrid '17. IEEE Press, 2017, p. 421–430. [Online]. Available: https://doi.org/10.1109/CCGRID.2017.89

[13] P. A. R. S. Costa, X. Bai, F. M. V. Ramos, and M. Correia, "Medusa: An efficient cloud fault-tolerant mapreduce," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2016, pp. 443–452.

[14] F. H. de Carvalho Junior and C. A. Rezende, "A Case Study on Expressiveness and Performance of Component-Oriented Parallel Programming," *J. of Parallel and Distributed Computing*, vol. 73, no. 5, pp. 557–569, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731512002882

[15] L. Deng, "The MNIST Database of Handwritten Digit Images for Machine Learning Research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[16] A. Sergeev and M. Del Balso, "Horovod: Fast and Easy Distributed Deep Learning in TensorFlow," *arXiv preprint arXiv:1802.05799*, 2018.

[17] J. Dongarra, "Basic Linear Algebra Subprograms Technical Forum Standard I," *International Journal of High Performance Applications and Supercomputing*, vol. 16, no. 2, pp. 115–199, feb 2002.