Towards multicluster computations with Julia

Francisco H. de Carvalho Junior[1], **Tiago Carneiro**[2]

[1]Pós-Graduação em Ciência da Computação (MDCC)
Universidade Federal do Ceará (UFC)
Fortaleza, Brazil

[2]ExaScience: HPC software laboratory
Interuniversity Microelectronics Centre (IMEC)
Leuven, Belgium

heron@dc.ufc.br

Abstract. *The ability to aggregate the computational resources of multiple clusters is useful for solving large problems that can benefit from multicluster platforms. In this context, this paper introduces an extension to Julia that enables multilevel parallelism and allows users to exploit use cases of multicluster computation. The proposal is evaluated through a proof-of-concept case study based on the multizone version of the NAS Parallel Benchmarks (NPB-MZ), focusing on evaluating inter-cluster communication and load-balancing overheads.*

1. Introduction

The ability to aggregate the computational resources of multiple clusters is useful for solving large problems that can benefit from multicluster platforms. There are relevant use cases, including BigData processing over geographically distributed datasets that can not be moved across computing sites, computation offloading using cloud-based clusters, insufficiency of computational resources in a single on-premises cluster site, and taking advantage of specific computational resources (e.g., high-end accelerators) that are not present in the local computation infrastructure. However, deploying multiclusters is challenging for most technical and scientific computing programmers due to the requirement of third-party tools often outside the programming language ecosystem.

This paper presents an approach to enable multicluster computations in a programming language designed to reconcile productivity and high-performance computing (HPC) requirements. The testbed language is Julia [Bezanson et al. 2018], targeted at technical and scientific computing programmers. For cluster computing, it offers built-in support through the standard Distributed.jl package, which addresses loosely coupled parallel computations, where little or no communication requirements during the computation exist. For tightly coupled parallel computations, users have the third-party package MPI.jl, which supports message-passing through MPI (Message Passing Interface) [Dongarra et al. 1996]. Users may benefit from the complementarity between Distributed.jl and MPI.jl by using MPIClusterManagers.jl, a package for deploying processes that can interact through both packages across the nodes of a cluster.

Distributed.jl assumes that parallel programs are launched from within the cluster's network domain through an access node (e.g., bastion host) to reach the computation nodes, which generally do not have public IP addresses. Based on this assumption, Distributed.jl do not allow the recursive creation of processes from within other processes,

except the master process, which initiates the computation. Therefore, we propose an extension to Distributed.jl that removes such a restriction, inspired by the notion of multilevel parallelism [De Carvalho Junior and Carneiro 2023], and demonstrates how such an extension enables multicluster computations in Julia by exploiting the complementarity between Distributed.jl and MPI.jl to benefit users of high-end parallel computing platforms.

As a proof of concept, we present a case study based on the multizone version of the NAS Parallel Benchmarks (NPB) [Bailey and et al. 1991, Jin and Van der Wijngaart 2006], reporting the results of performance evaluation experiments mainly concerned with inter-cluster communication overheads and load balancing.

In what follows, Section 2 characterizes distributed and parallel computing systems to clarify the complementarity between Distributed.jl and MPI.jl. Also, it defines multilevel parallel programming, motivates multicluster computing, and introduces Julia. Section 3 first describes the existing support of Julia for cluster computing through Distributed.jl and MPI.jl. Then, it presents the multilevel extension to Distributed.jl and how it can handle multicluster computation deployment together with MPI.jl and MPIClusterManagers.jl. The proof-of-concept evaluation of such an extension through the multizone case study is presented in Section 4. Finally, Section 5 conclude this paper by discussing its relevance, pointing out future works, and presenting final remarks.

2. Background and Related Works

Distributed computing is concerned with using a set of autonomous computing systems that interact through a communication network to achieve the objectives of an application. In turn, *distributed-memory parallel computing* is concerned with using a set of computing devices (possibly non-autonomous) that cooperate by exchanging messages through a network interconnection to carry out a computing task in the shortest possible time.

Clusters, representing 88,5% of the parallel computing platforms listed in the June'2024 ranking of Top500, belongs to the intersection between distributed computing and distributed-memory parallel computing systems, whereas MPPs (Massive Parallel Processors), representing the other 11,5%, are not distributed computing systems, since their computing nodes in general cannot be viewed as autonomous computers.

Clusters can be classified into tightly coupled and loosely coupled systems, as well as into capability and capacity systems. Examples of the former are high-end clusters with low-latency interconnections targeted at HPC, the competitors of MPPs in the top positions of Top500. They are suitable for fine-grained parallel computations, where inter-process communication interspersed with local computation is very frequent. They are mostly *capability systems*, i.e., specific-purpose supercomputers designed to apply all their computing power to solve a certain computationally challenging problem. In turn, examples of loosely coupled systems are *commodity clusters*, using off-the-shelf hardware, and *multiclusters*, where communication between processing nodes (clusters) may be performed over long-distance networks, such as the Internet. Loosely coupled systems suit embarrassingly and coarse-grained parallel computations, where inter-process communication does not exist, or its costs are negligible compared to computation costs. They are mostly *capacity systems*, whose computation power is divided among independent users interested in solving problems of their particular interest through queue systems.

2.1. Multilevel Parallel Programming

Modern parallel computing platforms have a hierarchical structure with multiple parallelism levels, each with a possibly distinct natural programming model to better exploit its potential performance [Ciccozzi et al. 2022]. To deal with the complexity of such platforms, multilevel parallel programming offers abstractions to develop efficient parallel code at the different parallelism levels [De Carvalho Junior and Carneiro 2023].

In a multicluster platform, we assume a parallelism hierarchy comprising the following levels: *multicluster-level*, having two or more clusters as processing elements that communicate through a high-latency network, such as the internet, when the clusters do not belong to the same organization or infrastructure provider; *cluster-level*, where the cluster nodes are the processing elements, having a dedicated network for communication; *multiprocessor-level*, having a set of processors and/or accelerators as processing elements, communicating through shared memory; and *multicore/manycore*-level, with processor cores as processing elements, also communicating through shared memory and sharing cache memory at one or more levels depending on the processor architecture.

Multiprocessor and multicore levels have shared-variables between threads as a natural programming model. In turn, message-passing and remote procedure calls (RPC) are the natural models at multicluster and cluster levels. While message-passing is better for programming parallel processes that act as interacting peers, commonly found among tightly-coupled systems, RPC and its variants are better when processes do not communicate with each other during computation, such as in embarrassingly parallelism patterns (e.g., master-slave, bag-of-tasks, and map-reduce) or there are hierarchical relations between them, such as client-server, commonly found among loosely-coupled systems.

2.2. Why Multicluster Computing?

The idea of combining the resources from a set of clusters to exploit their combined benefits in solving compute-intensive applications dates back to the end of the 1990s [Abawajy and Dandamudi 2003]. In the 2000s, it became a hot requirement in the research on grid computing systems [Foster and Kesselman 2004], mainly to tackle embarrassingly parallel computation patterns, such as bag-of-tasks, due to the high latency of the network connecting the clusters. In the 2010s, the interest in multicluster systems was boosted by cloud computing and Big Data analytics applications [Wu et al. 2017], justified by the need to deal with geographically distributed large amounts of data that cannot be copied outside organization boundaries due to transfer costs and/or data protection policies. In this context, the interest in large-scale parallel processing frameworks emerged, leading to the MapReduce frameworks (e.g., Hadoop) with several variants and extensions, such as stream processing frameworks (e.g., Spark and Flink) [Wang et al. 2012, Jayalath et al. 2014].

There are other real use cases for multicluster systems. For example, when users cannot find sufficient processing resources in a single cluster site, they may desire to look for more resources on other sites. Also, users may create clusters in cloud providers to offload computation when the computational resources offered by an on-premises cluster are insufficient for their needs. Finally, parts of the application may demand computational resources unavailable in the local environment or the current cloud provider, such as a high-end acceleration device that only a specific cloud provider can provide.

2.3. The Julia Programming Language

Julia appeared at the end of the 2000s, targeting scientific and technical computing applications [Bezanson et al. 2018]. It is maintained as an open-source project[1] by the JuliaHub company and a vibrant community of users and contributors, offering several packages for solving problems in different domains of science and engineering.

Julia aims to reconcile the productivity of dynamic languages like Python with the performance of native execution languages like Fortran and C/C++. For that, it combines just-in-time (JIT) compilation with a rich type system [Nardelli et al. 2018] designed to support a dynamic multiple dispatch mechanism. This approach makes it possible to generate efficient native code for methods that satisfy the *type stability* property [Pelenitsyn et al. 2021]. Since the performance of Julia's methods depends on how the code is structured, Julia's documentation includes a "performance tips" document to guide developers[2], as well as idiomatic coding guidelines.

In the next section, we present how Julia addresses issues of distributed and parallel computing and our contribution with a proposal to overcome the limitations of such an approach when dealing with multicluster computations.

3. Multilevel Parallel Computing in Julia

Julia includes Distributed.jl[3], a built-in package designed for loosely-coupled cluster computing since it is based on a variant of RPC. A Distributed.jl program comprises a set of P processes, numbered from 1 to P, where the process 1 is called *master* and the others are called *workers*. A standalone program or REPL session is the master process responsible for creating and coordinating the worker processes, which may reside in other hosts, such as the compute nodes of a cluster. The communication between the master and worker processes, and between worker processes, is mainly performed by the remote evaluation of expressions, a variant of RPC, from a caller to a callee process.

The master process creates worker processes by calling the function addprocs. In turn, the functions procs, nprocs, workers, nworkers, and myid may be used to inspect the number and identities of existing processes. Workers can be removed using rmprocs.

The *cluster manager* argument of addprocs determines the cluster environment where worker processes will be instantiated. For that, addprocs has a method for each cluster manager, selected through multiple dispatch. Julia provides two builtin cluster managers: LocalManager, for launching workers on the same host of the master process, and SSHManager, for launching workers on remote hosts that accept ssh authentication.

The ClusterManager.jl[4] package offers cluster managers for popular job queue systems, such as Slurm, Kubernetes, LSF (Load Sharing Facility), SGE (Sun Grid Engine), PBS (Portable Batch System), etc. In fact, any contributor may develop cluster managers for different cluster environments. In this work, we are interested in MPIClusterManagers.jl[5], which provides support for message-passing between worker processes through MPI, the most popular message-passing interface for HPC [Dongarra et al. 1996].

[1] https://github.com/JuliaLang/julia
[2] https://docs.julialang.org/en/v1/manual/performance-tips
[3] https://github.com/JuliaLang/Distributed.jl
[4] https://github.com/JuliaParallel/ClusterManagers.jl
[5] https://github.com/JuliaParallel/MPIClusterManagers.jl

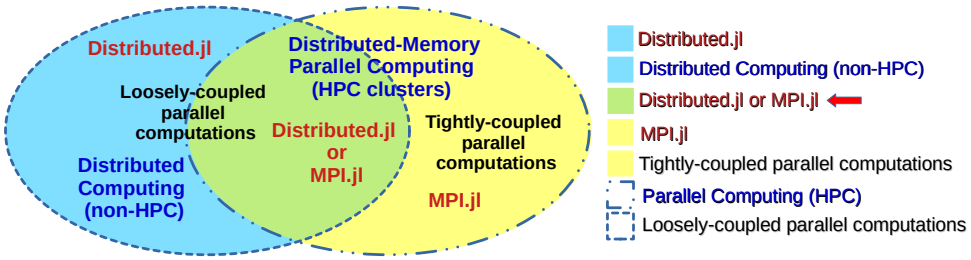

Figure 1. Distributed.jl versus MPI.jl

Master and worker processes interact through asynchronous remote evaluation of expressions. To evaluate an expression remotely, the caller process invokes the @spawnat macro, passing the callee process identifier and the expression as arguments. It immediately returns a value of Future type that may be passed to the @fetch macro (or fetch function) to wait and receive the result. There are variations of @spawnat, such as @spawn, which calls in an arbitrary worker, and @fetchfrom, which combines @fetch and @spawnat calls (synchronous evaluation). In turn, @everywhere evaluates the expression across a set of workers. Finally, the remotecall function provides support for asynchronous remote function calls, with synchronous variations: remotecall_fetch and remotecall_wait. Distributed.jl also offers high-level functions and macros for data distribution, such as @distributed and pmap, implementing the map/reduce paradigm.

3.1. The limitations of Distributed.jl

This paper addresses two limitations of Distributed.jl. The first is a *feature limitation*, as worker processes cannot create other worker processes recursively. If addprocs is called by a worker process, an exception is raised, and no worker is created. The second is an *expressiveness limitation*. Despite the programming model of Distributed.jl nicely fits client/server relations between processes, which applies to popular parallel computation patterns, such as bag-of-tasks and map/reduce, it is not suited to tightly-coupled parallel computing systems, since RPC and its variants are awkward and inefficient to implement peer-to-peer process interactions.

3.2. MPI.jl: message-passing in Julia

To face the expressiveness limitation of Distributed.jl, the Julia ecosystem offers MPI.jl[6], a package for using MPI (Message Passing Library) [Dongarra et al. 1996]. MPI is not a general distributed computing programming interface like Distributed.jl. It is designed for HPC, targeting tightly-coupled parallel systems and having implementations for clusters and MPPs. Figure 1 depicts how MPI.jl and Distributed.jl complement each other, showing that they are alternatives to each other for loosely-coupled parallel systems.

3.3. MPIClusterManagers.jl: Reconciling Distributed.jl and MPI.jl

To implement multicluster computations in Julia, we argue that MPI.jl is the choice to implement tightly-coupled parallel computations inside a cluster. In contrast, Distributed.jl is better for coordinating the clusters and managing their communication.

[6] https://github.com/JuliaParallel/MPI.jl

By making N calls to addprocs using MPIClusterManagers.jl as the cluster manager, the master process can create N groups of worker processes. Workers in the same group can communicate through the MPI.COMM_WORLD communicator, as each worker group is launched by a distinct mpiexec call. Workers of distinct groups may interact through Distributed.jl. In what follows, we present an illustrative example where $N = 4$:

```
using Distributed
using MPIClusterManagers
SUM = Ref(0); function reduce_master(x) SUM[] += x end
group = Array{Array{Int}}(undef, 4)
clustersize = [4,8,4,8]
for i in 1:4
    group[i] = addprocs(MPIWorkerManager(clustersize[i]))
    @everywhere group[i] using MPI
end
Threads.@threads for i in 1:4
    @everywhere group[i] begin
          clusterid = $i
          MPI.Init()
          size = MPI.Comm_size(MPI.COMM_WORLD)
          rank = MPI.Comm_rank(MPI.COMM_WORLD)
          @info "my info: rank=$rank, size=$size, cluster=$clusterid"
          X = rand(1:10)
          r = MPI.Reduce(X, (x,y) -> x + y, 0, MPI.COMM_WORLD)
          rank == 0 && @spawnat 1 reduce_master(r)
          MPI.Finalize()
       end
    end

@info "The sum across all groups is $(SUM[])$"
```

In the above code, each worker group (group[i] for $i \in \{1, 2, 3, 4\}$) performs, concurrently to each other, a reduction operation to sum their X variables initialized with some value between 1 and 10 in the variable r of the worker with rank 0. Finally, these workers call reduce_master at the master to sum their r in the master's variable SUM.

Unfortunately, the master process can not launch worker groups across distinct clusters because addprocs works with a local MPI installation. MPIClusterManagers.jl has been designed to launch Julia/MPI computations in a single cluster. To circumvent this limitation, we propose a multilevel extension to Distributed.jl based on a multilevel parallel programming model introduced in a previous work [De Carvalho Junior and Carneiro 2023]. It is described in the next section.

3.4. A Multilevel Extension to Distributed.jl

The multilevel extension to Distributed.jl allows workers to create other workers recursively using addprocs, enabling multicluster computations using MPIClusterManagers.jl. In this context, the master is now called *driver process*, and its workers are called *entry processes*, each placed in the access node of a cluster. Then, each entry process, acting as a master in its cluster, calls addprocs to create a set of worker processes across the cluster's nodes, called *compute processes*. While entry processes can interact with each other and compute processes they created only through Distributed.jl, compute processes can select between MPI.jl, for implementing tighly-coupled parallel computations, or Distributed.jl, for loosely-coupled computations. This is illustrated in the following code using MPI.jl:

```
using Distributed
using MPIClusterManagers
SUM = Ref(0); function reduce_master(x) SUM[] *= x end
mid = Array{Int}(undef, 4)  # manager IDs
clustersize = [4,8,4,8]
for i in 1:4
  clusterid[i] = addprocs(["<host address>"]; #= kw parameters to connect to the access host =# ...)[1]
  @everywhere [clusterid[i]] thisclustersize = $(clustersize[i])
```

```
9  end
10 @everywhere mid using MPIClusterManagers
11 @everywhere mid addprocs(MPIWorkerManager(thisclustersize); #= kw parameters to configure MPI =# ...)
12 @everywhere mid @everywhere workers() using MPI
13 for i in 1:4
14    @spawnat mid[i] @everywhere workers() clusterid = $(mid[i])
15 end
16 @everywhere mid reduce_master(x) = @spawnat role=:worker 1 reduce_master(x)
17 @everywhere mid @everywhere workers() begin
18       MPI.Init()
19       size = MPI.Comm_size(MPI.COMM_WORLD)
20       rank = MPI.Comm_rank(MPI.COMM_WORLD)
21       @info "my info: rank=$rank, size=$size, cluster=$clusterid"
22       X = rand(1:10)
23       r = MPI.Reduce(X, (x,y) -> x + y, 0, MPI.COMM_WORLD)
24       rank==0 && @spawnat role=:worker 1 reduce_master(r)
25       MPI.Finalize()
26 end
27
28 @info "The sum across all clusters is $(SUM[])"
```

This code performs the same computation as the previous code. The entry process of each cluster is created in lines 6-9, and compute processes are created by nested calls to addprocs from entry processes using MPIClusterManagers in line 11. So, these compute processes exchange messages through MPI.jl. In line 16, each entry process defines a reduce_master function to forward, to the driver process, the result (r) sent by each compute process with rank 0 to the entry process (line 24). This is necessary because compute processes cannot communicate directly with the driver process.

The role parameter An entry process may perform two *roles* when executing any operation: *master* ($id = 1$), when interacting with its compute processes, and *worker* ($id > 1$), when interacting with the driver and other entry processes. This is valid for any intermediary process (non-root and non-leaf) in a hierarchy of processes created by recursive calls to addprocs. For that, an additional keyword parameter **role** has been added to each Distributed.jl operation, with possible values :master and :worker. Since keyword parameters are optional, existing Distributed.jl programs still works with the extension. For that, the default value of **role** is :master for the driver process and :worker for the other ones.

4. Case Study: Multizone NAS Parallel Benchmarks (NPBMZ.jl)

The NAS Parallel Benchmarks (NPB) was developed in the 1990s to evaluate parallel computing platforms for CFD applications [Bailey and et al. 1991]. The original NPB implementation comprised 5 kernels (EP, IS, CG, MG, and FT) and 3 simulated applications (SP, BT, and LU), coded in Fortran or C, with serial and parallel versions based on MPI and OpenMP. Over the years, NPB has been widely used to evaluate the performance of platforms and programming languages for parallel computing, and new official and unofficial versions have been developed to add new kernels and problem instances.

We have implemented a proof-of-concept for multicluster computations using the multilevel version of Distributed.jl based on the multizone version of NPB 3.4.3 [Jin and Van der Wijngaart 2006], which includes the simulated applications. In the original version, for single clusters, the grid is partitioned into zones distributed across cluster nodes using MPI. Then, each zone is parallelized across processor cores using OpenMP. In the alternative multicluster version we developed, written in Julia, zones are distributed across clusters and then partitioned into cells distributed across the cluster nodes. Inter-cluster and intra-cluster interactions are implemented in Distributed.jl and

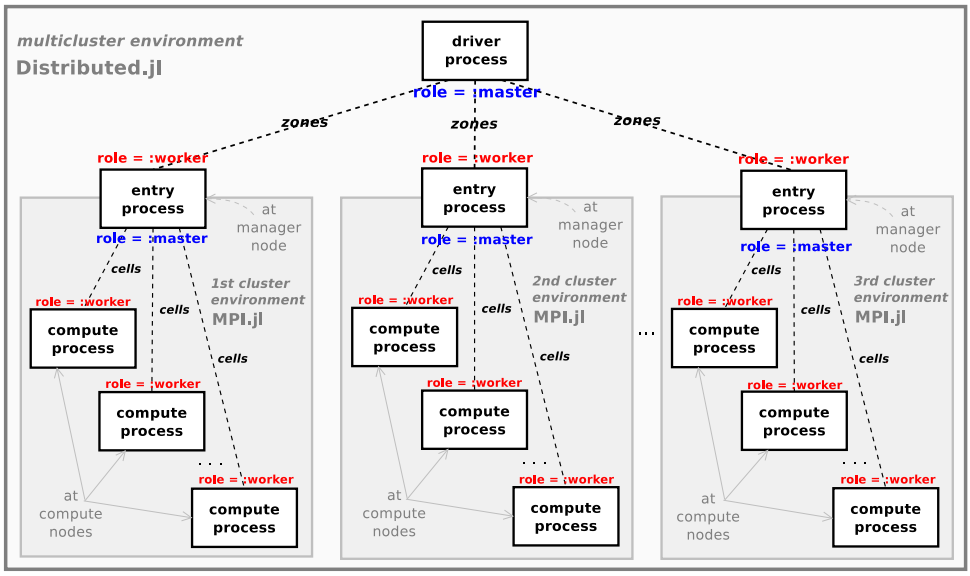

Figure 2. NPB Multizone General Architecture (SP, BT, LU)

MPI.jl, respectively. Figure 2 depicts such a parallelism hierarchy. A third level of parallelism is exploited by computing zones in different processor cores of the cluster nodes.

From Fortran to Julia To implement NPB 3.4.3-MZ in Julia, we firstly derived a Julia/MPI.jl version of SP, BT, and LU from NPB3.4-MPI, i.e., the original MPI versions of NPB, in Fortran, producing two versions for each program. The first version is a literal translation from Fortran to Julia. The second one applies Julia's performance tips [7] to reach competitive performance compared to the original Fortran version. Finally, the multi-zone version was built from the second version, implementing zone partitioning.

Listing 1. Launching MPB-MZ (BT, class E)

```
1  using Distributed
2  addprocs(... #= launching parameters for the entry process of the 1st cluster =#)
3  addprocs(... #= launching parameters for the entry process of the 2nd cluster =#)
4  process_count = [(2,4),(3,4)]
5  @everywhere workers() using MPIClusterManagers
6  @everywhere workers() using MPI
7  for (w,np) in process_count
8      fetch(@spawnat w addprocs(MPIWorkerManager(np); threadlevel=:multiple))
9  end
10 @everywhere workers() @everywhere workers() using NPBApps
11
12 using NPBApps
13 BT.go(BT.CLASS_E; itimer=2, npb_verbose=3)
```

Listing 1 outlines the code of a driver process launching a multizone NPB program (i.e., BT, class E) across two clusters with four nodes. Only compute processes, created in lines 7-9, perform relevant computations. The entry processes, created in lines 2-3, only manage the communication between clusters (exchange of zone faces) and communicating with the driver process to receive parameters and send results, by setting the **role** parameter to :worker, as well as communicating with compute processes to mediate parameter and result passing between them and the driver, by setting **role** to :master.

[7] https://docs.julialang.org/en/v1/manual/performance-tips/

cluster	locale	provider	CPU mark*	nodes count	memory per node	CPUs per node	CPU model	cores per node	network
grvingt	Nancy	Grid'5000	20,411	4	192GB	2	Intel Xeon Gold 6130	32	100Gbs Omni-Path
roazhon8	Rennes	Grid'5000	11,676	2	256GB	2	Intel Xeon E5-2630 v4	24	10Gbs Ethernet
p2.xlarge	us-east-1	AWS EC2	20,900	4	16GB	1	Intel Xeon E5-2686 v4	4	10Gbs Ethernet

* CPU mark according to `https://www.cpubenchmark.net/cpu_list.php`

Table 1. Clusters Characteristics

This experiment aims to evaluate the feasibility of deploying multicluster computations in Julia programs, as well as to assess performance overheads related to inter-cluster communication and load balancing. Speedup by employing multiple clusters is not a concern, as we argue that multicluster is an alternative only when running on a single cluster is not feasible. The methodology of the experiment follows in the next section.

4.1. Experimental Scenarios and Methodology

Table 1 describes the testbed clusters. In turn, Table 2 describes the problem classes D and E (workloads) applied in the three scenarios, for SP, BT, and LU.

Table 3 describes the experimental scenarios and performance results and deserves special attention. First, to fill the tables (a), (b), and (c) of Table 3, single cluster and multicluster executions of SP, BT, and LU have been executed for problem classes D and E over the grvingt and roazhon8 clusters. Single cluster executions, i.e., over each cluster individually, have been performed only for D instances since the E instances do not fit the memory of a single cluster, requiring the use of the two clusters in parallel.

The column **procs** sets the number of MPI processes launched across the nodes of each cluster. For all the programs, it must be a square number. So, it was the maximum square number less than the number of processor cores across the cluster nodes, but restricted by the partitioning of zones in cells (cells must have no less than $3\times3\times3$ dimension) [8]. This is why only 16 processes have been employed for BT in both clusters.

Trying to quantify the main sources of multicluster parallelism overhead, the execution times (per iteration) are determined by the sum of three components, in columns **comm** (inter-cluster communication time), **comp** (intra-cluster execution time), and **idle** (idle time). Since processes across all clusters synchronize at each iteration, the execution time is determined by the time from the beginning of an iteration to the finish of the execution on the "slower" cluster. The measurements in Table 3 are averages for the iterations of a single run of each program (see the number of iterations in Table 2).

[8] I would be better to run a single MPI process per node and distribute the zones to be processed among threads, one per core. Unfortunately, the use of threads and MPI in Julia has some technical issues outside the scope of this paper. See at `https://github.com/JuliaParallel/MPI.jl/issues/725`.

		grid dimensions			zone count			iterations
		x	*y*	*z*	*x*	*y*	*total*	
D	SP	1632	1216	34	32	32	1024	500
	BT							250
	LU							300
E	SP	4224	3456	92	64	64	4096	500
	BT							250
	LU							300

Table 2. Problem classes used in the experiments

(a) **BT**

class	clusters	procs	comm	comp	idle	lbal
D	grvingt	16	0.14	10.9	-	100%
	roazhon8	16	0.33	16.3	-	100%
	grvingt roazhon8	16 16	3.1	3.5 10.5	7.0 0.0	50% 50%
E	grvingt roazhon8	16 16	5.1	106.9 152.0	45.1 0.0	50% 50%

(b) **LU**

class	clusters	procs	comm	comp	idle	lbal
D	grvingt	64	0.01	1.1	-	100%
	roazhon8	36	0.01	2.2	-	100%
	grvingt roazhon8	64 36	0.37	0.6 1.1	0.5 0.0	50% 50%
E	grvingt roazhon8	64 36	40.4	9.0 24.6	15.6 0.0	50% 50%

(c) **SP**

class	clusters	procs	comm	comp	idle	lbal
D	grvingt	64	0.1	2.5	-	100%
	roazhon8	36	0.5	4.8	-	100%
	grvingt roazhon8	64 36	1.9	1.3 2.4	1.2 0.0	50% 50%
E	grvingt roazhon8	64 36	9.0	13.0 26.9	13.9 0.0	50% 50%

(d) **SP** (*cloud offloading*)

class	clusters	procs	comm	comp	idle	lbal
D	grvingt roazhon8 p2-xlarge	64 36 4	11.4	1.0 1.8 4.2	3.2 2.4 0.0	33.3% 33.3% 33.3%
	grvingt roazhon8 p2-xlarge	64 36 4	5.4	1.6 1.7 1.7	0.18 0.07 0.00	55.9% 30.9% 13.2%
E	grvingt roazhon8 p2-xlarge	64 36 4	21.8	19.7 15.4 18.2	0.0 4.3 1.5	58.9% 31.7% 9.04%

Table 3. Performance measures (per iteration)

Finally, the **lbal** column informs *load balancing*, i.e., the distribution of zones across the target clusters. In Table 3(a-c), they are uniformly distributed across them.

Table 3(d) evaluates two aspects. First, it exemplifies the offloading of computations on cloud-based clusters by including a third cluster from AWS EC2 provider comprising eight p2.xlarge instances as nodes. Second, it demonstrates the impact of loading balancing across the clusters. For that, at first, the zones are uniformly distributed across the grvingt, roazhon8, and p2-xlarge clusters. Then, the zones are redistributed across these clusters by using the performance results to minimize idle time. This methodology works for class D but fails for E because the uniform distribution does not fit the memory capacity of the offloading cluster (p2-xlarge). So, for E, we have assigned to the p2-xlarge cluster the maximum possible number of zones and balanced the number of zones assigned to grvingt, roazhon8 according to the performance results of Figure 3(c).

4.2. Discussion

The numbers of Table 3 show two sources of multicluster parallelism overhead:

- The high latency of communication between the clusters, especially when they must communicate through the internet;
- The different processing powers of the clusters result in high idle times without proper load balancing (zone assignment).

Using **comm**, we define the inter-zone communication overhead as the percentage of execution time that the programs waste synchronizing zone faces. For instance, the minimum communication overhead for single cluster execution has been measured for LU by less than 1% (negligible), while the maximum has been measured for SP: 4.2% for grvingt and 8.9% for roazhon8. These measures have been 1.3% and 2.0% for BT. In a single cluster, all interzone communication is performed through MPI, and the higher overhead for roazhon8 is justified because grvingt uses a faster interconnection compared to roazhon8 (see Table 1). In contrast, still for class D, multicluster execution makes

the communication overhead of LU, SP, and BT increase to 25.2%, 44.2%, and 22.8%, respectively, while, for class E, the values are 62.1%, 25.1%, and 3.2%.

Another important source of overhead in multicluster execution is the lack of load balancing. Due to the different processing capabilities of clusters, this leads to inefficient use of computational resources, resulting in significant idle times. For instance, in Table 3, the difference in processing speed between grvingt, roazhon8, and p2.xlarge clusters may be observed by looking at the **comp** column for the experimental cases where the workload (zones) is uniformly distributed across the clusters.

As pointed out in Section 4.1, Table 3(d) presents performance measures by balancing the number of zones assigned to the three clusters for SP, using the information of uniformly distributed executions and satisfying memory restrictions of the offloading cluster (p2.xlarge). For class D, the execution time of SP over the three clusters (**comm** + **comp** + **idle**) moves from $15.6s$ to $7.1s$, with negligible idle time. For class E, we can not compare with the execution time of uniform load balancing because it was not possible to run this case due to memory restrictions of the p2.xlarge cluster, but the maximum idle time compared to the total execution time is around 10%, less than a half compared with other results for class E using two clusters (24% for LU, 39% for SP, and 29% for BT).

5. Conclusions

This paper contributes to the problem of deploying multicluster computations in programs using multilevel parallel programming [De Carvalho Junior and Carneiro 2023], herein implemented by an extension to the Julia programming language. Such an extension removes the restriction of the built-in Distributed.jl package that only the master process may create worker processes, so that a hierarchy of processes may be created. Using this, the master process of a Julia program (driver process) may launch workers (entry processes) in the access nodes of distinct clusters, which in turn launch nested workers (compute processes) across the cluster's nodes. Using MPIClusterManagers, programmers may use MPI.jl to code tighly-coupled parallel computations involving compute processes in the same cluster (intra-cluster parallelism) and Distributed.jl to code loosely-coupled computations between entry processes (inter-cluster parallelism).

The ability to deploy multicluster computations in Julia may cover a number of relevant use cases, including BigData processing over large geographically distributed datasets, computation offloading in cloud-based clusters, insufficiency of computational resources in a single on-premises cluster site (e.g., like in the case study with Grid'5000), and taking advantage of specific computational resources (e.g., high-end accelerators) that are not present in the local computation infrastructure.

As a proof-of-concept, we developed a multicluster implementation for the multi-zone version of NPB in Julia, comprising the SP, BT, and LU simulated applications. Due to the intensive communication between clusters in these applications, it has been possible to analyze the overhead of multicluster execution, with special attention to high-latency communication and load imbalance.

We plan to continue this work by creating new case studies to explore suitable use cases of multicluster execution in Julia, and to work on proper abstractions to facilitate multicluster deployment and hierarchical process interactions.

Acknowledgments

The experiments presented in this paper were carried out on the Grid'5000 testbed [Bolze et al. 2006], hosted by INRIA and including several other organizations.

References

Abawajy, J. H. and Dandamudi, S. P. (2003). Parallel Job Scheduling on Multicluster Computing System. In *IEEE Intern. Conference on Cluster Computing*, pages 11–18.

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

Bezanson, J., Chen, J., Chung, B., Karpinski, S., Shah, V. B., Vitek, J., and Zoubritzky, L. (2018). Julia: Dynamism and Performance Reconciled by Design. *ACM Programming Languages*, 2(OOPSLA).

Bolze, R., Cappello, F., Caron, E., Daydé, M., Desprez, F., Jeannot, E., Jégou, Y., Lanteri, S., Leduc, J., and Melab, N. e. a. (2006). Grid'5000: a large scale and highly reconfigurable experimental grid testbed. *The International Journal of High Performance Computing Applications*, 20(4):481–494.

Ciccozzi, F., Addazi, L., Asadollah, S. A., Lisper, B., Masud, A. N., and Mubeen, S. (2022). A Comprehensive Exploration of Languages for Parallel Computing. 55(2).

De Carvalho Junior, F. H. and Carneiro, T. (2023). A Component Model for Multilevel Parallel Programming. In *XXVII Brazilian Symposium on Programming Languages*, SBLP'23, page 25–32, New York, NY, USA. Association for Computer Machinery.

Dongarra, J., Otto, S. W., Snir, M., and Walker, D. (1996). A Message Passing Standard for MPP and Workstation. *Communications of ACM*, 39(7):84–90.

Foster, I. and Kesselman, C. (2004). *The Grid 2: Blueprint for a New Computing Infrastructure*. M. Kauffman.

Jayalath, C., Stephen, J., and Eugster, P. (2014). From the Cloud to the Atmosphere: Running MapReduce across Data Centers. *IEEE Trans. on Computers*, 63(1):74–87.

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

Nardelli, F. Z., Belyakova, J., Pelenitsyn, A., Chung, B., Bezanson, J., and Vitek, J. (2018). Julia Subtyping: A Rational Reconstruction. *Proceedings of the ACM Programming Languages*, 2.

Pelenitsyn, A., Belyakova, J., Chung, B., Tate, R., and Vitek, J. (2021). Type Stability in Julia: Avoiding Performance Pathologies in JIT Compilation. *ACM Programmming Languages*, 5(OOPSLA).

Wang, L., Tao, J., Marten, H., Streit, A., Khan, S. U., Kolodziej, J., and Chen, D. (2012). MapReduce Across Distributed Clusters for Data-intensive Applications. In *26th Intern. Parallel and Distributed Processing Symposium*, pages 2004–2011.

Wu, D., Sakr, S., Zhu, L., and Wu, H. (2017). Towards big data analytics across multiple clusters. In *17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid '17, page 218–227. IEEE Press.