DistributedWorkflows.jl - A Julia interface to a task-based workflow management system.

Firoozeh Dastur[1, 2], Max Zeyen[3], and Mirko Rahn[2]

[1]RPTU Kaiserslautern
[2]Fraunhofer ITWM
[3]NorthStar Earth & Space

ABSTRACT

DistributedWorkflows.jl is a serializer-independent interface to a distributed task-based workflow management system. This package aims to simplify the process of writing distributed applications. Given a workflow pattern in the form of a Petri net and the Julia code for the workflow tasks, it can be used on a cluster, for example with Slurm, to automate the application's parallel deployment. This makes *DistributedWorkflows.jl* an invaluable addition to Julia's growing ecosystem of high-performance computing.

Keywords

Distributed workflows, Parallel computing, Task-based computing, Petri net workflows, High-Performance Computing (HPC), User experience (UX), User-centered design (UCD)

1. Introduction

Distributed computing is an essential research component and has led to significant advancements across many scientific domains. The availability and increasing performance of HPC systems enables the realization of experiments that are too costly, too dangerous, practically infeasible, or all of the above. Additionally, increasing data sizes and problem complexity have raised the necessity of adapting to a parallel world for many researchers.

For example, in the modeling and analysis of protein synthesis and DNA mutation, distributed systems enable efficient simulations of complex biological processes, providing insights into genetics and molecular biology (e.g. [28]). Similarly, weather simulations have greatly benefited from distributed systems, allowing for more accurate and faster predictions (e.g. [23]). Additionally, climate simulations are enabled to run over larger time ranges (e.g. [31]). Other examples include asteroid impact simulations[26], black hole mergers[17], and nuclear fusion simulations[22], just to name a few (see Figure 1). These achievements highlight the transformative power of distributed systems in scientific research.

However, programming distributed applications is a challenge of its own, especially for scientists who are not trained in the proper usage of HPC systems. Although this task is ideally outsourced to qualified computer scientists, this is not always a viable option due to many different reasons (e.g. budget).

Despite the clear advantages, many domain scientists struggle to write parallel applications. The complexity of parallel programming, coupled with the need for specialized knowledge in distributed systems, creates a barrier for scientists who are experts in their fields but not in computer science, particularly HPC models.

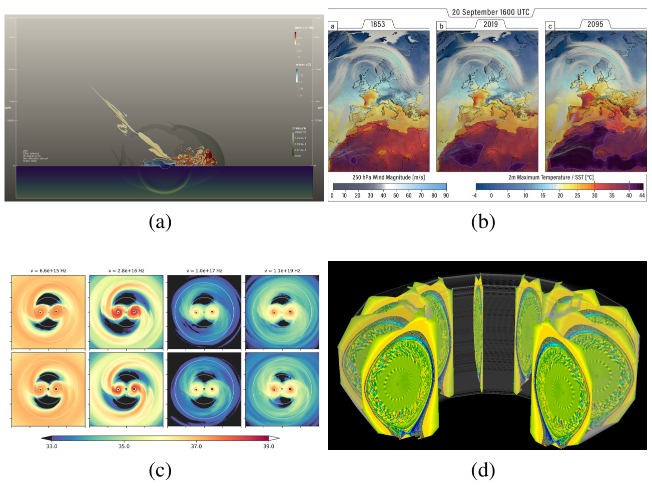

20 September 1600 UTC
1853
2019
2095
250 hPa Wind Magnitude [m/s]
2m Maximum Temperature / SST [°C]
(a)
(b)
(c)
(d)

Fig. 1. (a) Deep ocean meteorite impact [26], (b) evolution of future European heat waves [31], (c) supermassive black holes merger [17], (d) nuclear fusion simulation [22].

To address the challenges that arise due to large data sizes and application complexity, the concept of workflows has been introduced. Workflows provide a higher-level abstraction that simplifies the process of building parallel applications. By abstracting away much of the low-level complexity, workflows enable domain scientists to focus on the problem at hand rather than the intricacies of parallel programming. This approach significantly reduces the cognitive load required to implement distributed systems and makes parallel computing more accessible to non-computer scientists.

In this paper, we introduce *DistributedWorkflows.jl*[15], a Julia package that aims to facilitate the writing of parallel applications. *DistributedWorkflows.jl* is a high-level wrapper around the HPC tool GPI-Space[18], which allows domain scientists to create efficient parallel applications with a user-friendly interface. This package simplifies the process of building distributed applications by providing a higher-level abstraction, enabling scientists to focus on their research rather than the challenges of parallel programming.

The paper is structured as follows: In Section 2, we provide an overview of related work in distributed computing with Julia and applications that utilize the underlying tool of DistributedWorkflows. Section 3 presents background information on Petri nets and GPI-Space. In Section 4, we highlight the main features of the package, and describe the design and methodology of the package. Section 5 provides a short description of how to use *DistributedWorkflows.jl*, while Section 6 discusses its limitations. Finally, Section 7 briefly describes future work and our concluding remarks.

2. Related Work

Julia's rich ecosystem already includes some packages for high-performance computing. The most popular of these packages are *Distributed.jl*[4] and *Dagger.jl*[12, 30].

Distributed.jl offers functionalities for creating and managing multiple Julia processes remotely, enabling distributed and parallel computing. It utilizes network sockets or other supported interfaces to facilitate communication between Julia processes and relies on Julia's serialization package from its standard library to efficiently transfer Julia objects between processes. The package includes a comprehensive set of utilities for creating and destroying Julia processes, adding them to a "cluster" (a collection of interconnected Julia processes), and enabling Remote Procedure Calls (RPC) between processes within the cluster. However, it requires the manual launching of tasks, which adds complexity to managing distributed workloads and makes it less user-friendly for domain scientists.

Dagger.jl is a scheduler that draws inspiration from Dask[14]. It efficiently executes computations represented as directed-acyclic graphs (DAGs) across multiple Julia worker processes and threads, as well as GPUs via *DaggerGPU.jl*. While DAGs are a powerful way to represent dependencies in a computation, as the name suggests, they cannot represent cyclic dependencies.

DistributedWorkflows.jl uses GPI-Space[18] as its underlying framework. GPI-Space has been utilized across a wide range of domains, from big-data processing [29], reverse time migration for seismic measurement data [21], to computational algebraic geometry [13]. It employs Petri nets as its workflow description language, which were named after Carl Adam Petri, who extensively analyzed these graphs in his dissertation[27]. Petri nets are directed bipartite graphs capable of modeling both concurrency and resource sharing more naturally. In Petri nets, the transitions and places provide a richer representation of parallelism, and workflows can easily capture more complex interactions in distributed systems. This makes Petri nets, as used in *DistributedWorkflows.jl*, a more powerful model for designing distributed applications, as they allow for greater expressiveness and ease in handling complex tasks and synchronization. Table 1 provides a feature comparison between *Distributed.jl*, *Dagger.jl*, and *DistributedWorkflows.jl*.

This package was created out of the need to provide a convenient way to parallelize computations in the Julia package *Oscar*[25]. *Oscar* is a computer algebra system that combines the capabilities of its four cornerstones *GAP*[19], *Polymake*[20], *Antic*[1] and *Singular*[16]. Each of these cornerstones come equipped with their own serializer, in addition to OSCAR's own serializer that works from within Julia. Hence, the need of *DistributedWorkflows.jl* to be serializer independent.

Singular already employs GPI-Space to write parallel applications, for example see [13]. However, *Singular*'s workflows are custom-designed for specific applications and are not reusable across different tasks. Another challenge is the need to compile the workflow Petri net along with the executable code, making it more difficult to debug and structure parallel applications. Although *Singular* has been using GPI-Space for some time, it lacks a user-friendly interface. Furthermore, *Singular*'s serializer is fixed to its custom `ssi` format.

In contrast, Julia's flexibility allows users to bring their own serializer, which makes *DistributedWorkflows.jl* a more versatile package in terms of serialization options.

Table 1. Comparison table reflecting different features of packages.

Feature	Distributed.jl	Dagger.jl	DistributedWorkflows.jl
Low-level	✓	✗	✗
Task based	✗	✓	✓
Automated task execution	✗	✓	✓
Serializer independent	✗	✗	✓
Out of the box cyclic workflows	✗	✗	✓
Built-in fault tolerance	✗	✓	✓

3. Background

In this section, we provide a short background on Petri nets used to define workflows in *DistributedWorkflows.jl* and its underlying distributed execution framework GPI-Space.

3.1 Petri Nets

The programming model of *DistributedWorkflows.jl* uses Petri nets to define workflow patterns, which enable the modeling of concurrent and distributed systems.

A (basic) *Petri net* $\mathcal{PN}$ is a logic model that represents the order in which the events occur. We define a Petri net as a bipartite graph connecting nodes "places" to "transitions" via directed graphs, see Figure 2. In other words, a Petri net can be defined mathematically as a tuple $\mathcal{PN} := (P, T, F, M)$, where:

(1) P is a finite collection of places.
(2) T is a finite collection of transitions.
(3) F defines the arcs or flow relations, connecting a place $p \in P$ to a transition $t \in T$ or vice versa.
(4) M is the (initial) "Marking". It is a function from $P \to N$, where $N \in \mathbb{N}$ is the number of tokens in a place.

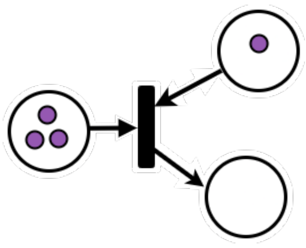


Fig. 2. A simple Petri net where the circular nodes are places, the black rectangle is a transition and the colored dots are tokens.

Some definitions also use an additional parameter W which is a weight function $W : F \longrightarrow \mathbb{N}^+$ assigning positive integers to each arc connecting places to transitions.

Petri nets have gotten multiple extensions over the years to improve their modeling convenience and/or add to their computational power. We list a few of them below:

(1) **Colored Petri nets** add data attributes (colors) to tokens, enabling easier modeling of complex systems with structured information.
(2) **Timed Petri nets** incorporate timing constraints on transitions, making them suitable for real-time systems.
(3) **Stochastic Petri nets** introduce probabilistic firing times, commonly used for performance and reliability analysis.
(4) **Hierarchical Petri nets** support modularity by nesting Petri nets within others, facilitating the modeling of large-scale systems.
(5) **Continuous Petri nets** generalize tokens to real numbers and transitions to continuous dynamics, ideal for systems like fluid flow.
(6) **Inhibitor Petri nets** include special arcs called inhibitors, which make the Petri net formalism Turing complete.
(7) **High-level Petri nets** combine features of these types, offering flexible modeling for abstract or multifaceted systems.

DistributedWorkflows.jl uses high-level, timed, and colored Petri nets to be consistent with GPI-Space, our underlying system. Additionally, we support a special kind of transition called `conditional transitions` which also comes from GPI-Space. This is a special case built on top of inhibitor Petri nets. These are required to make Petri nets non-deterministic and in turn Turing complete. For more details on Petri nets and its applications see [7, 11, 24, 32].

3.2 GPI-Space

GPI-Space is a task-based workflow management system for parallel applications written in C++. It utilizes the aforementioned Petri nets to define its workflows. Petri nets are defined using an `XML` dialect called `XPNet`. `XPNet` files are compiled into shared libraries for execution by GPI-Space applications.

The framework builds on an "agent-worker" architecture, as shown in the Figure 3. The agent houses the workflow engine and the scheduler. The workflow engine is responsible for determining which transitions in the Petri net can be fired. Next, exactly one token for each input place is consumed and bundled together with the transition into an activity. Activities are handed over to the scheduler to be queued up for execution with a worker. The worker processes that execute the tasks are distributed across the compute nodes. The Remote Interface Daemon (RIFD) on each host coordinates startup and shutdown steps. The Workers and the RIFDs together constitute the Distributed Runtime System (DRTS), as depicted in Figure 3. Finally, activity results are passed back from the scheduler to the workflow engine where the outputs are put as tokens into their corresponding place.

4. Design and Method

The *DistributedWorkflows.jl* package started as an interface to GPI-Space[18], but it has grown to include features that significantly improves user experience. The primary focus being on ease of use

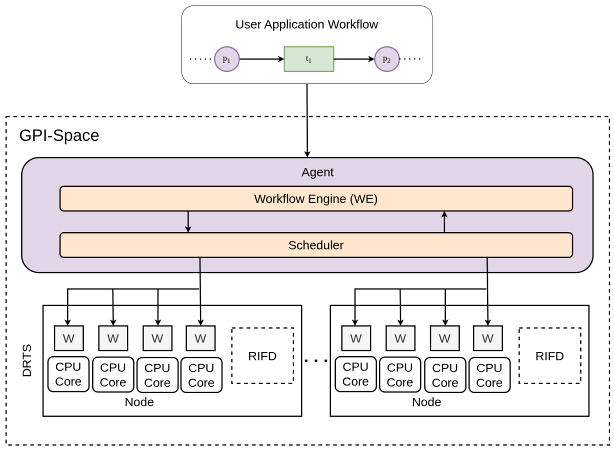


Fig. 3. Top block shows a local portion of the user application workflow. The lower block shows how GPI-Space handles the information internally. For source image see [18].

and flexibility, this package extends the capabilities of the underlying system. Below is a summary of its key features:

—**Ease of use**
Designed to simplify user interactions, with a fully documented public API, consisting of examples for every method.

—**Serializer Independent**
Supports Julia's built-in serializer, as well as other serializer formats like JLD2[8], HDF5[6], or any other custom serializer see for example the serializer for Julia package Oscar[25], allowing for flexibility in data handling.

—**Simplified Workflow Creation**
Reduced complexity of writing parallel applications and the Petri net workflows, making it accessible to experts and non-experts alike.

—**Workflow Component Creation**
Allows users to create Petri net workflow components without requiring a fully functional application.

—**Workflow Component Editing**
Has the ability to add and/or remove parts of any Petri net component, giving users greater control over the workflow design.

—**Workflow Visualization**
Features a visualization tool using Graphviz[5] to generate and view Petri nets, aiding in workflow design.

—**XML File Generation and Compilation**
Includes a convenience function to generate and compile workflows into XML files required by GPI-Space, directly within Julia.

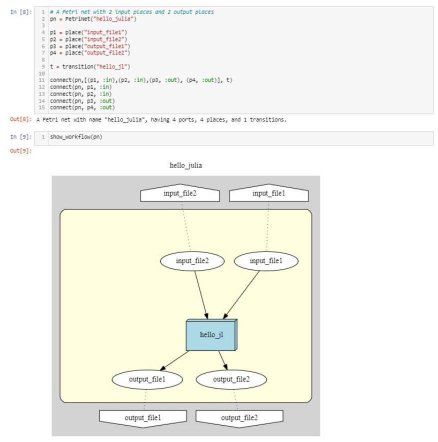
—**Reusable Workflows**
The compiled workflows are reusable and can be applied to various applications following the same workflow pattern. That is, the same workflow can be used by multiple applications by simply modifying the Julia code for the transitions in the application configuration.

—**Local Testing**
Enables local testing of applications before deploying them on expensive cluster resources.

—**Easy Application Debugging**
Enables easy debugging for individual transitions in the Petri net from within Julia.

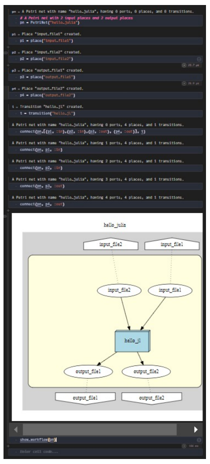
—**Direct Job Submission**
Supports job submission directly from Julia, streamlining deployment.

—**Notebook Support**
Integrates with Jupyter[10] and Pluto[9] notebooks, enabling workflow viewing and editing in an interactive environment, see Figure 4.



```
In [8]:
# A Petri net with 2 input places and 2 output places
pn = PetriNet("hello_julia")

p1 = place("input_file1")
p2 = place("input_file2")
p3 = place("output_file1")
p4 = place("output_file2")

t = transition("hello_jl")

connect(pn,[(p1, :in),(p2, :in),(p3, :out), (p4, :out)], t)
connect(pn, p1, :in)
connect(pn, p2, :in)
connect(pn, p3, :out)
connect(pn, p4, :out)

Out[8]: A Petri net with name "hello_julia", having 4 ports, 4 places, and 1 transitions.

In [9]: show_workflow(pn)
Out[9]:
```

(a)



(b)

Fig. 4. Snippet of a simple Petri net workflow created and viewed using a *Jupyter* notebook in (a), and a *Pluto* notebook in (b).

This package has been designed with a user-centric approach from its inception. Our primary goal has been to remove the complexities of writing parallel applications and make distributed computing more accessible to domain scientists. As a result, users can focus on parallelizing their algorithms and designing a Petri net to define the workflow, rather than dealing with system setup intricacies. To achieve this, we abstracted the C++ components of GPI-Space into a library and integrated its communication functionality into Julia using CXXWrap[3].

Since GPI-Space is a workflow management tool not natively designed for Julia, users need to configure their environment to utilize it. GPI-Space itself can be downloaded by following the installation steps provided in [18]. *DistributedWorkflows.jl* on the other hand, provides binaries for selected operating systems and architectures, that can be downloaded from the GitHub repository [15]. Following the step by step instructions, the user can setup their environment to use the package.

For users who wish to visualize workflows before compilation, it is recommended to install Graphviz[5], which allows visualization in various formats. Once the setup is complete, the *DistributedWorkflows.jl* package can be installed directly in Julia using its package manager. At this stage, users are ready to parallelize and execute their applications with *DistributedWorkflows.jl*.

Without the interface of *DistributedWorkflows.jl*, the user will have to embed their parallel Julia algorithm within the workflow which is written in the `XML` based `XPNet` format. The workflow can only be visualized after the `XML` is compiled with functioning code. For an example snippet see Listing 1.

```xml
<defun name="hello_world">
  <in name="Input" place="Input" type="string"/>
  <out name="Output" place="Output" type="string"/>
  <net>
    <place name="Input" type="string"/>
    <place name="Output" type="string"/>
    <transition name="greet">
      <defun>
        <in name="Input" type="string"/>
        <out name="Output" type="string"/>
        <module name="hello_world"
                function="greet (Input, Output)">
          <cinclude href="string"/>
          <code><![CDATA[
            Output = "Hello " + Input;
          ]]></code>
        </module>
      </defun>
      <connect-in place="Input" port="Input"/>
      <connect-out place="Output" port="Output"/>
    </transition>
  </net>
</defun>
```

Listing 1

The *DistributedWorkflows.jl* package provides a workflow builder in the form of a Petri net, developed entirely in Julia, which can be visualized and edited without compiling the workflow or having a running code available. See for example Listing 2.

```julia
pnet = Workflow_PetriNet("hello_world")
p1 = place("Input")
p2 = place("Output")
t1 = transition("greet")
connect(pnet,[(p1,:in),(p2,:out)],t1)
connect(pnet,[(p1,:out),(p2,:in)])

# Visualizing the workflow before generating the XML file
show_workflow(pnet)
# Generating the XML file
generate_workflow(pn, "tmp/new_workflow")
```

Listing 2

Once the algorithm has been parallelized in the form of a Petri net workflow, users require to write a workflow launcher, compile everything, and write an executable script, in case of using GPI-Space (see GPI-Space documentation [18]). We simplify the process of creating a workflow launcher and the process of compiling and executing by providing easy to use API functions in Julia. For examples, see the examples directory in the GitHub repository [15].

Additionally, since GPI-Space is agnostic to the serializer in Julia, we provide an easy way of using any kind of serializer format, in case the user's application does not use the serializer from Julia's standard library. See `examples/other_serializers` in [15] for an example with a custom serializer.

5. How to Use *DistributedWorkflows.jl*

In this section, we will give a simple example showcasing the use of *DistributedWorkflows.jl* for parallel applications in the Julia ecosystem.

Let us consider an example from geometry, where we want to compute the *Minkowski sum* for $A_1, \ldots, A_n$, where each A_i is a set of position vectors in the Euclidean space.

The *Minkowski sum* of two sets of position vectors U and V in the Euclidean space is computed by adding each vector in U to each vector in V, i.e.

$$U + V := \{u + v \mid u \in U,\ v \in V\}.$$

A Petri net describing the workflow to compute the *Minkowski sum* $\sum_{i=1}^{n} A_i$, for $n \in \mathbb{N}$, in parallel is given in Figure 5.

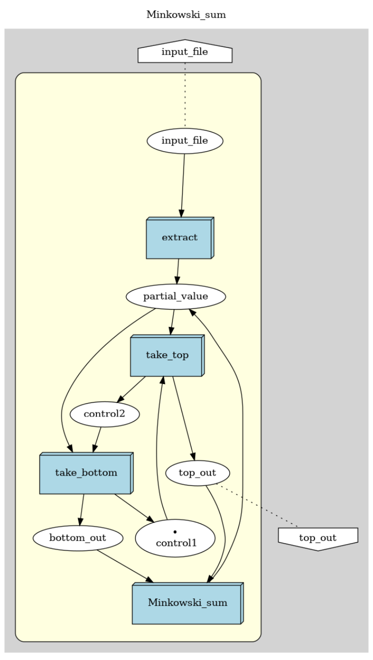


Fig. 5. A Petri net (generated using *DistributedWorkflows.jl*) modeling the workflow to compute the *Minkowski sum* of a collection of sets of points in $\mathbb{R}^2$.

This Petri net can be written in Julia using *DistributedWorkflows* as shown in Listing 3.

```julia
using DistributedWorkflows

# create an empty workflow
pnet = Workflow_PetriNet("Minkowski_sum")

# define all the places
p1 = place("input_file")
p2 = place("partial_value")
p3 = place("control1", :control_init)
p4 = place("control2", :control)
p5 = place("top_out")
p6 = place("bottom_out")

# define all the transitions
t1 = transition("extract")
t2 = transition("take_top", :exp)
t3 = transition("take_bottom", :exp)
t4 = transition("Minkowski_sum")

# connect the places to transitions with respective arcs
connect(pnet,[(p1,:in),(p2,:out)],t1)
connect(pnet,[(p2,:in),(p3,:in),(p4,:out),(p5,:out)],t2)
connect(pnet,[(p2,:in),(p3,:out),(p4,:in),(p6,:out)],t3)
connect(pnet,[(p5,:in),(p6,:in),(p2,:out)],t4)

# connect the input/output ports
connect(pnet,[(p1,:in), (p5,:out)])
```

Listing 3

Now, we can visualize the workflow using `show_workflow()` or it can be saved using a preferred format using `savefig()` methods, see Figure 5. Once we have a satisfactory workflow, an XML file can be generated using `generate_workflow()` and later compiled using `compile_workflow()` methods.

Assuming that we have a method called `Minkowski_sum(A, B)`, where the method takes as input two sets A and B and returns a set C which is the *Minkowski sum* of the input sets. Next, we set up our workflow launcher by configuring our application as well as the workflow as shown in Listing 4.

```julia
# name of the input port from where the
# serialized data enters the workflow
impl_port = "implementation_1"

jl_impl = "/path/to/code/per/transition.jl"

# name of functions corresponding to each transition
fname0 = "extract"
fname1 = "Minkowski_sum"

# configure the application
app = application_config(impl_port,
                         jl_impl,
                         [fname0, fname1])
out_dir = "path/to/output"

# workflow configuration
workflow_cfg = workflow_config("Minkowski_sum.pnet",
                               out_dir, app)

# set the input parameter for the initial state
input_stream = input_pair("input_file",
                          "/path/to/serialized_data")

input_var = [input_stream]
```

Listing 4

Note, this can be saved as a Julia script to execute together with our compilation step.

At this stage, if we are happy with the workflow and our transition code, we can compile and locally execute the application to test it, before using expensive cluster resources, as shown in Listing 5.

```
resource_list = joinpath(tmp_dir, "nodefile")
# start the client locally with the number of workers
# based on the available resources.
client = client(4, resource_list , "local")

# submit the workflow to the client with
# the workflow configuration
submit_app = submit_workflow(client, workflow_cfg, input_var)
```

Listing 5

Finally, depending on the cluster settings, we can run this example on a cluster. Note that since clusters are not standardized, the cluster execution script may vary depending on the cluster manager. For details on cluster managers in Julia see [2].

As a more complex example, a Petri net workflow (see Figure 6) that models different repair synthesis pathways of DNA as in [28] can also be tested using *DistributedWorkflows.jl*. In this case, since the application setup has been simplified due to the Julia interface, the most complex part will be to have the transition code in Julia that would be executed using a given workflow pattern.

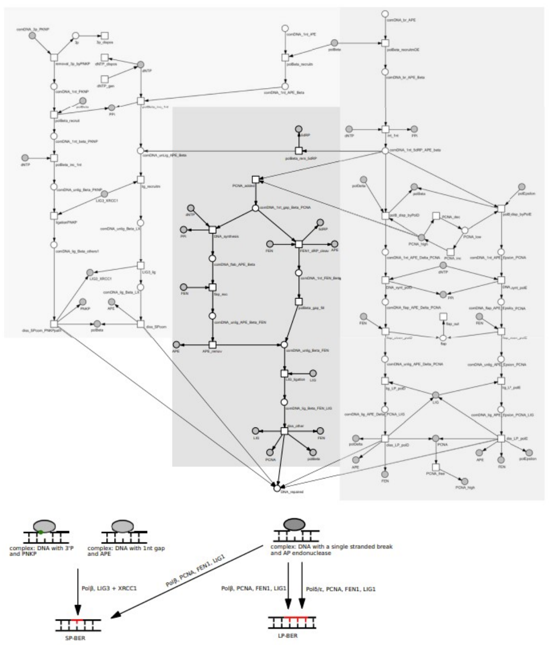

complex: DNA with 3'P and PNKP
complex: DNA with 1nt gap and APE
complex: DNA with a single stranded break and AP endonuclease
Polβ, LIG3 + XRCC1
Polβ, PCNA, FEN1, LIG1
Polβ, PCNA, FEN1, LIG1
Polδ/ε, PCNA, FEN1, LIG1
SP-BER
LP-BER

Fig. 6. Part of a Petri net modeling different repair synthesis pathways of DNA as in [28].

6. Limitations

Like any software, the *DistributedWorkflows.jl* package has certain limitations. It is an interface package and hence relies on tools outside of Julia. It has somewhat of a setup process before being able to parallelize applications. It is currently, recommended for long-running processes due to I/O overhead and requires a shared filesystem. Official support for the workflow manager is limited to Ubuntu 20 and 22 LTS, with compatibility restricted to Linux distributions, excluding macOS and Windows. Additionally, the package relies on Spack for binary installation, which may introduce additional setup complexity for some systems.

7. Conclusions and Future Work

This paper and the *DistributedWorkflows.jl* package introduce a user-friendly interface for managing distributed, task-based workflows. It enables users to generate, visualize, compile, and launch workflows represented as Petri nets through simple methods. The package provides a fully documented public API, allowing users to locally test applications before deploying them on expensive clusters, ensuring cost efficiency. With binaries available for multiple Linux distributions, the tool is currently best suited for long-running processes.

Upcoming updates aim to enhance the package with specialized transition types for reduced boilerplate code, additional workflow examples to guide users, and convenience functions that streamline the process of creating and managing workflows. Furthermore, improvements to the user interface are planned, making the tool even more intuitive. These features are expected to expand the package's utility, enabling broader adoption and more efficient handling of distributed workflows across various applications.

8. ACKNOWLEDGEMENTS

We extend our gratitude to Fraunhofer ITWM for funding the Ph.D. studies of the first author. As well as, enabling her to work on and develop *DistributedWorkflows.jl*. We are also deeply grateful to Prof. Dr. Anne Frühbis-Krüger and Prof. Dr. Claus Fieker for their invaluable insights into the needs of domain scientists, which guided the design of this package with a strong focus on user experience. Our sincere thanks goes to Dr. Tiberiu Rotaru for his expertise in GPI-Space, particularly during the initial stages of integrating our Julia prototype with C++.

9. References

[1] ANTIC – Algebraic Number Theory In C. `https://github.com/flintlib/antic`.
[2] ClusterManagers Julia Package. `https://github.com/JuliaParallel/ClusterManagers.jl`.
[3] CxxWrap.jl: Package to make C++ libraries available in Julia. `https://github.com/JuliaInterop/CxxWrap.jl`.
[4] Distributed.jl: Create and control multiple Julia processes remotely for distributed computing. ships as a Julia stdlib. `https://github.com/JuliaLang/Distributed.jl`.
[5] Graphviz. `https://www.graphviz.org/`.
[6] HDF5.jl File Format. `https://juliaio.github.io/HDF5.jl/stable/`.
[7] Introduction to Petri Nets. `https://link.springer.com/chapter/10.1007/978-1-4471-4276-8_10`.
[8] JLD2 File Format. `https://github.com/JuliaIO/JLD2.jl`.
[9] Pluto.jl — interactive Julia programming environment. `https://plutojl.org/`.
[10] Project Jupyter. `https://jupyter.org/`.

[11] Understanding Petri Nets: Modeling Techniques, Analysis Methods, Case Studies. `https://link.springer.com/book/10.1007/978-3-642-33278-4`.

[12] R. Alomairy, F. Tome, J. Samaroo, and A. Edelman. Dynamic Task Scheduling with Data Dependency Awareness using Julia. pages 1–6, 2024.

[13] Janko Böhm and Anne Frühbis-Krüger. Massively parallel computations in algebraic geometry. In *Proceedings of the 2021 International Symposium on Symbolic and Algebraic Computation*, ISSAC '21, page 11–14, New York, NY, USA, 2021. Association for Computing Machinery.

[14] Dask Development Team. *Dask: Library for dynamic task scheduling*, 2016.

[15] F. Dastur, M. Zeyen, and M. Rahn. DistributedWorkflows.jl - A Julia interface to a distributed task-based workflow management system. `https://firoozehdastur.github.io/DistributedWorkflows.jl/stable/`, January 2024.

[16] W. Decker, G.M. Greuel, G. Pfister, and H. Schönemann. SINGULAR 4-4-0 — A computer algebra system for polynomial computations. `http://www.singular.uni-kl.de`, 2024.

[17] S. d'Ascoli, S.C. Noble, D.B. Bowen, M. Campanelli, J.H. Krolik, and V. Mewes. Electromagnetic Emission from Supermassive Binary Black Holes Approaching Merger. *The Astrophysical Journal*, 865(2):140, oct 2018.

[18] Competence Center High Performance Computing Fraunhofer ITWM. GPI-Space. `https://www.gpi-space.de`, September 2020.

[19] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.14.0*, 2024.

[20] E. Gawrilow and M. Joswig. *polymake: a Framework for Analyzing Convex Polytopes*, pages 43–73. Birkhäuser Basel, Basel, 2000.

[21] D. Gruenewald, N. Ettrich, M. Rahn, and F.J. Pfreundt. FRTM - A Productive Framework for Reverse Time Migration. 2014.

[22] J. Kress, D. Pugmire, S. Klasky, and H. Childs. Visualization and Analysis Requirements for in Situ Processing for a Large-Scale Fusion Simulation Code. In *2016 Second Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV)*, pages 45–50, 2016.

[23] J. Michalakes. *HPC for Weather Forecasting*, pages 297–323. Springer International Publishing, Cham, 2020.

[24] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[25] OSCAR – Open Source Computer Algebra Research system, Version 1.3.0-DEV, 2024.

[26] J. Patchett, F. Samsel, K. Tsai, G. Gisler, D. Rogers, G. Abram, and T. Turton. Visualization and Analysis of Threats from Asteroid Ocean Impacts. 2016. Winner, Best Scientific Visualization and Data Analytics Showcase; LA-UR-16-26258.

[27] C. A. Petri. *Kommunikation mit Automaten*. Phd thesis, Technische Hochschule Darmstadt, 1962. Available at `https://www2.informatik.uni-hamburg.de/TGI/mitarbeiter/profs/petri/PetriDis.pdf`.

[28] M. Radom, M. A. Machnicka, J. Krwawicz, J. M. Bujnicki, and P. Formanowicz. Petri net–based model of the human DNA base excision repair pathway. *PLOS ONE*, 14:1–26, 09 2019.

[29] T. Rotaru, M. Rahn, and F.J. Pfreundt. MapReduce in GPI-Space. In *Euro-Par 2013: Parallel Processing Workshops: BigDataCloud, DIHC, FedICI, HeteroPar, HiBB, LSDVE, MHPC, OMHI, PADABS, PROPER, Resilience, ROME, and UCHPC 2013, Aachen, Germany, August 26-27, 2013. Revised Selected Papers 19*, pages 43–52. Springer, 2014.

[30] J. Samaroo, R. Alomairy, M. Giordano, and A. Edelman. Efficient Dynamic Task Scheduling in Heterogeneous Environments with Julia. 2024.

[31] A. Sánchez-Benítez, H. Goessling, F. Pithan, T. Semmler, and T. Jung. The July 2019 European Heat Wave in a Warmer Climate: Storyline Scenarios with a Coupled Model Using Spectral Nudging. *Journal of Climate*, 35(8):2373 – 2390, 2022.

[32] D. A. Zaitsev. Toward the Minimal Universal Petri Net. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 44(1):47–58, 2014.