SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis | 979-8-3503-5554-3/24/$31.00 ©2024 IEEE | DOI: 10.1109/SCW63240.2024.00255

Parsl+CWL: Towards Combining the Python and CWL Ecosystems

Nishchay Karle*, Ben Clifford*, Yadu Babuji*†, Ryan Chard†, Daniel S. Katz† and Kyle Chard*†
* Department of Computer Science, University of Chicago, Chicago, USA
† Argonne National Laboratory, Lemont, USA
‡NCSA & CS & iSchool, University of Illinois Urbana-Champaign, Urbana, IL, USA

***Abstract*—The Common Workflow Language (CWL) is a widely adopted language for defining and sharing computational workflows. It is designed to be independent of the execution engine on which workflows are executed. In this paper, we describe our experiences integrating CWL with Parsl, a Python-based parallel programming library designed to manage execution of workflows across diverse computing environments. We propose a new method that converts CWL CommandLineTool definitions into Parsl apps, enabling Parsl scripts to easily import and use tools represented in CWL. We describe a Parsl runner that is capable of executing a CWL CommandLineTool directly. We also describe a proof-of-concept extension to support inline Python in a CWL workflow definition, enabling seamless use in Parsl's Python ecosystem. We demonstrate the benefits of this integration by presenting example CWL CommandLineTool definitions that show how they can be used in Parsl, and comparing performance of executing an image processing workflow using the Parsl integration and other CWL runners.**

I. Introduction

Scientific workflows are essential for automating complex computational tasks, enabling reproducibility, portability, and scalability of science applications. The pervasiveness of scientific workflows has led to the development of hundreds of workflow management systems that support the development and execution of workflows; however, most workflow systems are not interoperable and thus workflows developed using one workflow system are not usable in another. The Common Workflow Language (CWL) attempts to overcome this obstacle via a common workflow description that can be interpreted by many workflow systems. CWL has been widely adopted, and users have created a rich ecosystem of CWL workflows and tools. Importantly, the community has undertaken the significant effort to describe tools (i.e., applications and scripts) in CWL, including specifying input/output formats, command line invocation arguments, and environment requirement. These tool definitions are a critical step, irrespective of workflow system, in being able to programatically execute a tool as part of a workflow.

CWL is designed such that workflow definitions are independent of the *CWL runner*—the workflow system that executes the workflow. Many CWL runners have been implemented, such as CWLTool [1], Toil [2], Arvados [3], and previously Cromwell [4]. These runners provide distinct capabilities and thus have distinct user communities. An advantage of CWL is that workflows can be easily ported between these different runners enabling users to choose a runner that best matches their requirements.

Here, we describe work towards integrating CWL and Parsl [5]—a parallel programming library designed to enable parallel Python execution across different computing resources, from local clusters to cloud platforms. Parsl's dataflow model allows for intuitive definition of workflows directly in the Python programming language. Parsl's flexible execution framework enables scalable and efficient execution of workflows across many computing platforms, particularly at scale on large HPC systems. These capabilities make Parsl a potentially valuable runner for CWL workflows.

Our integration of CWL and Parsl aims to make it possible for Parsl developers to programmatically import and invoke CWL-defined tools directly in Python programs. Importantly, this integration removes the need for Parsl developers to manually specify and maintain tool definitions in Parsl's Python-based representation. Importing CWL tools directly into Parsl programs enables a richer programmatic approach to composing workflows that might include CWL tools, existing tools represented in Parsl, pure Python functions, and program logic written in Python to manage the execution of the workflow. We further implement a proof-of-concept CWL CommandLineTool runner that enables Parsl to execute CommandLineTools using Parsl's robust, scalable, and performant executors.

We see several benefits of integrating CWL and Parsl:

- **Portability**: CWL provides a common way to describe tools, ensuring that they can be executed on different platforms.
- **CWL ecosystem**: There are many CWL CommandLineTool definitions that describe input and output formats, command line interfaces, and environment requirements that can be used directly in Parsl without requiring developers to recreate these definitions in Python.
- **Scalability and Performance**: Parsl's runtime engine and various executors efficiently manage resources, allowing workflows to scale from personal computers to high-performance computing clusters.
- **Familiarity/Productivity**: Python is arguably the lingua franca of Science. Our CWL and Parsl integration enables the composition of workflows in Python while leveraging the curated tool definitions from the CWL ecosystem.

979-8-3503-5554-3/24/$31.00 ©2024 IEEE
DOI 10.1109/SCW63240.2024.00255

Our experiences integrating Parsl and CWL highlighted the challenges of supporting CWL expressions—snippets of code written in JavaScript embedded in CWL YAML workflow definitions—given Parsl's Python-based environment. To overcome this mismatch we propose a prototype extension to CWL to support inline Python expressions in CWL workflow definitions. Inline Python allows for dynamic logic within workflows to be described entirely in Python.

This paper is structured as follows. §II describes CWL and Parsl. §III outlines our extensions to Parsl to import and run CWL CommandLineTools. §IV presents an example workflow and shows how the CWL workflow can be implemented in Parsl using our integration. §V describes how we support Python expressions in CWL. §VI compares the performance of our integration and Python expressions with other CWL runners. Finally, §VII describes related work and §VIII summarizes our contributions.

The code described in this paper is openly available on GitHub (https://github.com/Parsl/cwl-parsl) under the Apache-2 license.

II. Background

Here we describe the foci of our integration: CWL and Parsl.

A. Common Workflow Language (CWL)

The Common Workflow Language (CWL) [6] is an open specification that is designed to address the challenges of reproducibility and interoperability in scientific research. CWL achieves this goal by providing a common specification for workflows to ensure they can be shared and reused irrespective of the underlying workflow engine used.

CWL has two main abstractions: CommandLineTools and Workflows. *CommandLineTool* definitions, written in YAML, outline the interface to a command line tool (e.g., an application, script, or anything that can be invoked via the command line). The tool definition describes the input arguments, environment, and output files. The definition can then be used to invoke the command line tool, given suitable input arguments. The tool definition format allows definitions to be shared across workflows and referenced from registries. The community has invested significant effort cataloging tools and sharing definitions [7].

CommandLineTool definitions are used in a CWL *Workflow* definition, also written in YAML. The workflow links together CommandLineTools by specifying the exchange of input/output between tools. Importantly, while the workflow describes the various steps (and their input/outputs), execution of the tools is determined by dependencies rather than the order they are specified in the workflow definition. CWL supports software container technologies (e.g., Docker) to abstract execution environments.

While the CWL specification is a static representation of a workflow in YAML, there are many situations in which dynamic decisions need to be made as a workflow progresses, for example, to select a specific CommandLineTool to execute based on the output of a previous CommandLineTool or to modify arguments passed between CommandLineTools. CWL provides several built-in methods for common manipulations and also supports arbitrary *expressions*—snippets of JavaScript code that are evaluated during workflow execution.

A CWL workflow is executed by a CWL *runner*. The runner is responsible for managing the invocation of the CommandLineTools, determining when they can be executed, composing the execution command, monitoring execution and determining success or failure, and managing the exchange of data between CommandLineTools. *cwltool* [1] is the reference implementation of CWL runner. It is implemented in Python and maintained by the CWL community. cwltool is able to validate CWL descriptions, parsing the representation and ensuring that it is compliant with the CWL specification. It can also execute the workflow with user-supplied input arguments and a workflow definition. *toil-cwl-runner* is a Python-based CWL runner built on the Toil workflow engine. It validates CWL descriptions for compliance with the CWL specification and can execute workflows on both single-node setups and distributed cloud environments, using user-supplied inputs and workflow definitions.

B. Parsl

Parsl [5] is a parallel programming library for Python. Parsl allows developers to write programs entirely in Python and Parsl then manages execution of those programs across diverse computing resources. Parsl abstracts the complexities inherent in parallel computing by providing a straightforward functional programming model at the task level while maintaining procedural Python code for the wider program and task and data dependencies.

In Parsl, developers annotate Python functions as `apps` to specify that they can be executed concurrently. When an `app` is invoked, a *Future* is returned that tracks the asynchronous execution of the *app*. Dataflow is implicitly specified when a Future from one app is passed as input to another app. Parsl dynamically generates a task dependency graph and then maps the graph to available resources for execution, exploiting parallelism where possible by managing the creation of data objects and ensuring that app dependencies are met.

Parsl implements an extensible plugin model for its runtime execution system called *Executors*. Executors implement Python's `concurrent.futures.Executor` class and are responsible for executing a task and returning a future to the calling program. Parsl supports standard implementations of the `concurrent.futures.Executor` class, such as the *ThreadPoolExecutor*. It also includes several Parsl-specific Executors, such as the *HighThroughputExecutor*, and interfaces with other community Executors, such as *TaskVineExecutor* and *RadicalPilotExecutor*.

The most commonly used Executor, the *HighThroughputExecutor* (HTEX), employs a pilot job model to manage the execution of tasks on a paralel or distributed computer. This model introduces an abstraction layer that decouples task submission from resource allocation, thereby enabling efficient utilization of available computational resources. In the

pilot job model, a placeholder job—referred to as a pilot—is submitted to a batch scheduler. Tasks are then executed on the pilot job without interfacing with the batch scheduler. The pilot job model is particularly advantageous in environments with high variability in job queue time.

Parsl also implements an extensible *Provider* interface that facilitates the negotiation of computing resources from a range of batch systems, public clouds, and container orchestration systems like Kubernetes. *Providers* in Parsl are responsible for managing the lifecycle of compute resources, including provisioning, monitoring, and deprovisioning resources, thus enabling automatic scaling to match the needs of the workflow at runtime. This abstraction of the resource management system, allows Parsl to run on local clusters, cloud environments, and supercomputers with minimal configuration changes.

Parsl interacts with many other tools, such as TaskVine as mentioned before, as well as Globus [8]. Parsl is used in a range of scientific applications, has been shown to scale to some of the largest supercomputers, and is used as the basis for building other services, such as Globus Compute [9].

III. Integrating CWL and Parsl

Our integration of CWL and Parsl focuses on two primary areas: 1) importing CWL CommandLineTool definitions in Parsl, and 2) implementing a prototype CWL Parsl runner to execute CommandLineTools via Parsl.

A. Importing Tool Definitions

Given a CWL CommandLineTool definition, we seek to integrate the tool into a Parsl program such that it can be executed like any other Parsl app and therefore interwoven seamlessly in the program. The challenge here is that CWL CommandLineTool definitions are written in a YAML format while Parsl apps are represented as Python functions.

To overcome this difference we introduce a new Parsl app: *CWLApp*. The CWLApp reads a CWL CommandLineTool definition and transparently creates a Parsl *BashApp* that is configured to run the CWL CommandLineTool. The process of creating a CWLApp requires only that the developer specifies a file containing the CWL CommandLineApp. The CWLApp will read the definition and populate the input/output definition for the Parsl app. The CWLApp is callable as a Python function, allowing users to execute the tool by passing the required input arguments. These inputs, combined with the definitions from the CWL CommandLineTool, are used to construct and then execute the command.

The inputs specified in the CWL CommandLineTool are represented as keyword arguments in the CWLApp. When invoked, these arguments are processed according to the specifications in the CWL CommandLineTool definition. Prefixes and positions defined in the CommandLineTool's *inputBinding* definition are matched with arguments at runtime. Any inputs that are of type "File" are converted into Parsl's *File* type, which facilitate access regardless of the location where the app is executed.

The outputs specified in the CommandLineTool are managed as described in the definition. Both *stdout* and *stderr* are directed to their designated files. For any new file created during execution, a Parsl *DataFuture* object is created and returned. These DataFuture objects can then be passed as file inputs to other Parsl apps (including CWLApps) without needing to wait for the files to be available.

CWLApps can be created once and imported, allowing them to be reused across workflows. They can also be created in a Python module and then imported directly into other Python programs.

Listing 1 shows a CommandLineTool definition for the Linux echo command. It describes the required input argument, a string called message, and the output stdout file that is to be produced. Listing 2 shows how the CommandLineTool definition is used to create a CWLApp in Parsl and how the CommandLineTool can then be executed with Parsl.

```yaml
cwlVersion: v1.2

class: CommandLineTool

baseCommand: echo

inputs:
  message:
    type: string
    default: "Hello World"
    inputBinding:
      position: 1

outputs:
  output:
    type: stdout

stdout: hello.txt
```

Listing 1: CWL CommandLineTool definition for "echo"

```python
from parsl.configs.local_threads import config
from parsl_cwl.cwl_app import CWLApp

parsl.load(config)

echo = CWLApp("echo.cwl")

future = echo(
    message="Hello, World!",
    stdout="hello.txt",
)

# Wait for the future before reading the output
future.result()

with open("hello.txt", "r") as f:
    print(f.read())
```

Listing 2: An example Parsl program that first loads a Parsl configuration, loads the CWL CommandLineTool definition from the echo.cwl file, executes the CommandLineTool using Parsl, waits for the task to complete, and prints the contents of the output file.

B. Parsl CWL CommandLineTool Runner

We now describe how Parsl can act as a CWL runner to execute a single CWL CommandLineTool definition. This integration enables users to leverage Parsl's scalability and performance when running a CommandLineTool on high

performance parallel and distributed systems. Using the Parsl CWL CommandLineTool runner, a user can simply pass a CWL CommandLineTool definition to run with the Parsl CWL runner command: *parsl-cwl*. Currently, parsl-cwl can only execute CWL CommandLineTools directly; in the future we will extend this integration to support Workflow definitions.

Here, we show an example of how a CWL CommandLineTool can be executed by specifying the CommandLineTool definition (`echo.cwl`), defining the Parsl configuration to use (`config.yml`), and specifying inputs either as command line arguments or as a YAML file (`inputs.yml`):

```
$ parsl-cwl config.yml echo.cwl inputs.yml
```

```
$ parsl-cwl config.yml echo.cwl --message='Hello'
```

We adopt a YAML-based Parsl configuration to match the CWL ecosystem. Specifically, we use the format defined in the TaPS benchmark suite [10] to specify key configuration options. The configuration includes the executor and provider, as well as various options such as number of nodes, number of workers per node, accelerators to be used, and Python or container environment.

IV. Example Workflow

We now present an example image processing workflow written in CWL and ported to Parsl using our integration.

A. Image Processing Workflow in CWL

Listing 3 shows the CWL Workflow definition. The workflow encodes a sequence of image processing tasks: resizing an image, applying a sepia filter, and blurring the image. Each stage is implemented as a separate CWL *CommandLineTool*. The workflow takes an image as input, and produces a blurred and possibly sepia-filtered image as output. The three stages of the workflow are:

1) **Image Resizing (`resize_image.cwl`)**: This stage takes an input image and resizes it to the specified dimensions. The target size is provided as an input parameter, allowing for flexible resizing according to user requirements.
2) **Image Filtering (`filter_image.cwl`)**: This stage applies a sepia filter to the resized image. The filter is controlled by a boolean parameter, enabling a user to apply the sepia effect as needed.
3) **Image Blurring (`blur_image.cwl`)**: The final stage blurs the filtered image using a specified blur radius. This step allows a user to soften the image, with the degree of blurring controlled by an input parameter. The final blurred image is saved as `blurred.png`.

The workflow takes four input arguments.

- `input_image`: The original image file to be processed.
- `size`: The dimensions to which the image should be resized.
- `sepia`: A boolean flag indicating whether the sepia filter should be applied.
- `radius`: The radius used for the blur operation.

The workflow produces one output file:

- `final_output`: The final blurred image, output from the `blur_image` stage.

```yaml
cwlVersion: v1.2

class: Workflow

doc: This CWL workflow processes images by
    performing a series of tasks - resizing,
    filtering, and blurring

requirements:
  - class: StepInputExpressionRequirement

inputs:
  input_image:
    type: File
    doc: The original image to be processed

  size:
    type: int
    doc: The target sizeXsize for resizing

  sepia:
    type: boolean
    doc: Whether to apply the filter

  radius:
    type: int
    doc: The amount of blur to apply

outputs:
  final_output:
    type: File
    outputSource: blur_image/output_image

steps:
  resize_image:
    run: resize_image.cwl
    in:
      input_image: input_image
      size: size
      output_image:
        valueFrom: "resized.png"
    out: [output_image]

  filter_image:
    run: filter_image.cwl
    in:
      input_image: resize_image/output_image
      sepia: sepia
      output_image:
        valueFrom: "filtered.png"
    out: [output_image]

  blur_image:
    run: blur_image.cwl
    in:
      input_image: filter_image/output_image
      radius: radius
      output_image:
        valueFrom: "blurred.png"
    out: [output_image]
```

Listing 3: CWL Image Processing Workflow

B. Image Processing Workflow in Parsl

Listing 4 shows the image processing workflow from Listing 3 implemented in Parsl by importing CWL CommandLineTool definitions and orchestrating the execution of each

CommandLineTool in sequence. We present this example to illustrate how the CWL CommandLineTools can be integrated in a Parsl program and to highlight the advantages of using a Python interface to specify the control and data flow between steps. In this example, we take a modular and Pythonic approach by defining a function representing the three sequential steps, which can then be used repeatedly (e.g., in a loop or in another function) to process multiple images concurrently. Parsl will derive a DAG of the individual tasks and execute them in an interleaved fashion when their dependencies are met (i.e., it will not wait for the three steps to be completed before running a step from another loop).

```
import glob

from myconfigs import perlmutter_config
from parsl_cwl.cwl_app import CWLApp


parsl.load(perlmutter_config)

resize_image = CWLApp("resize_image.cwl")
filter_image = CWLApp("filter_image.cwl")
blur_image = CWLApp("blur_image.cwl")

def process_img(image: str) -> Future:
    resized_img_future = resize_image(
        input_image=image,
        size=1024,
    )

    filtered_img_future = filter_image(
        input_image=resized_img_future.outputs[0],
        sepia=True,
    )

    blurred_img_future = blur_image(
        input_image=filtered_img_future.outputs
            [0],
        radius=1,
    )

    return blurred_img_future


final_imgs = [process_img(img) for img in glob.
    glob(r'**/*.png', recursive=True)]

concurrent.futures.wait(
    final_imgs, return_when=concurrent.futures.
        ALL_COMPLETED
)
```

Listing 4: Python Script Example

We describe in listing 4 the various parts of the Parsl program to illustrate how one can use the CWL integration.

Configuration and Executor Setup: We first configure a `HighThroughputExecutor` to manage execution of the workflow. In this case, we load a configuration for the Perlmutter Supercomputer at NERSC. We note that any Parsl executor can be used in the program by using the appropriate configuration. For example, one can easily use a configuration for a specific machine, from their prior use of that machine, configurations published by HPC sites or from the Parsl documentation for many ACCESS and institutional HPC clusters.

Creating CWLApps: As shown in listing 4, we create a Parsl CWLApp for each CommandLineTool by specifying the CWL file.

Defining the workflow: We define the workflow by creating a Python function that includes the three stages. Each consumes the output of the prior stage, which establishes the dataflow graph.

Stage 1: Image Resizing: The first stage uses the CWLApp, `resize_image`, to resize each image to a specified size. A DataFuture is returned that is passed on to the next stage.

Stage 2: Image Filtering: The `filter_image` CWLApp is used to apply a sepia tone filter to each resized image by setting the sepia input argument to True. The input image to this step is extracted from the DataFuture from the previous resizing step A new DataFuture is returned to reference the filtered image and is passed on to the next stage.

Stage 3: Image Blurring: The final stage involves blurring each filtered image using the `blur_image` CWLApp. The output images, obtained from the DataFutures of Stage 2, are used as input for for the blurring operation.

Starting the workflow: As the workflow is written in Python, we have access to the full capabilities of Python. In this case, we use a list comprehension and a glob pattern to identify all "png" files in the subdirectories and to start an instance of the workflow for each. We maintain a list of Futures for all processed images.

Wait for Results: The use of Futures enables concurrent execution of the workflow stages for all images. Parsl handles the workflow automatically, ensuring that the result of each future is available before executing the subsequent stage. The pipeline waits for all processing futures to complete before concluding, ensuring all images are fully processed.

C. Discussion

We have shown how the image processing workflow can be implemented in Parsl by importing the CWL CommandLineTool definitions. As shown, the Parsl implementation offers an intuitive Python implementation using the full power of the Python programming language, enabling modular workflow definition, simple looping over inputs, asynchronous execution, and access to Parsl's executor and provider ecosystem. The same workflow can therefore be easily moved between computers and scaled from laptops to supercomputers.

V. Python Expressions in CWL Workflows

It is clear that static workflow representations are not suitably expressive for many scientific workflows. Indeed, such observations motivated the development of Parsl and it's predecessor Swift [11], while other static specification languages like CWL have evolved to support more dynamic behavior via inclusion of code.

CWL supports specification of dynamic *expressions* within workflow definitions. These expressions, written in JavaScript, enable CWL workflows to adapt their behavior dynamically. For example, expressions can be used to modify input arguments to pass to CommandLineTools, modify arguments

passed between stages, or operate on results from CommandLineTools. Here, we propose a new Python-based expression that better matches the execution environment of Parsl. Further, Python is a widely-used, easy-to-understand, and productive language.

We approach this problem by defining a new type of expression: **InlinePythonRequirement**. We follow closely the CWL *InlineJavascriptRequirement* used for JavaScript. We allow inclusion of Python code, including functions, to be specified in the InlinePythonRequirement section. As a referenceable requirement, the expression can then be reused throughout the workflow definition.

When invoking the Python expression, we need a way to refer to variable attributes in the workflow (e.g., input arguments) and to select the arguments to be passed to the expression. We adopt a simple notation for referencing attributes in the workflow: `$(inputs.input)`.Here, the "$" indicates the reference, "inputs" the input arguments, and "input" the specific argument. We adopt a Python f-string-like [12] syntax to template arguments to be passed to the InlinePythonRequirement. This allows expressions to be used anywhere in the CWL workflow definition.

```
cwlVersion: v1.2
class: CommandLineTool
requirements:
  - class: InlinePythonRequirement
    expressionLib:
      - |
        def capitalize_words(message):
            """
            Capitalize each word in the given
                message.

            Args:
                message (str): The input message.

            Returns:
                str: The message with each word
                    capitalized.
            """
            return message.title()

baseCommand: echo

inputs:
  message:
    type: string

arguments:
    - f"{capitalize_words($(inputs.message))}"

outputs: []
```

Listing 5: InlinePythonExpression capitalizing words

Listing 5 and Listing 6 show examples of how the InlinePythonExpression can be used. Listing 5 shows a simple CommandLineTool that calls the echo command and uses the Python Expression to capitalize the result. Here the InlinePythonExpression defines a *capitalize_words* function to capitalize the input message (a string). The InlinePythonExpression is called in the argument attribute (arguments are used in CWL to create additional options or modify workflow inputs for invoking a CommandLineTool). The invocation of the *capitalize_words* function is enclosed within a Python 'f-string', signaling to *parsl-cwl* that it is a Python expression requiring evaluation. The argument processes the user input message, calls the InlinePythonExpression to capitalize the input, and then invokes the echo CommandLineTool with the capitalized message as input.

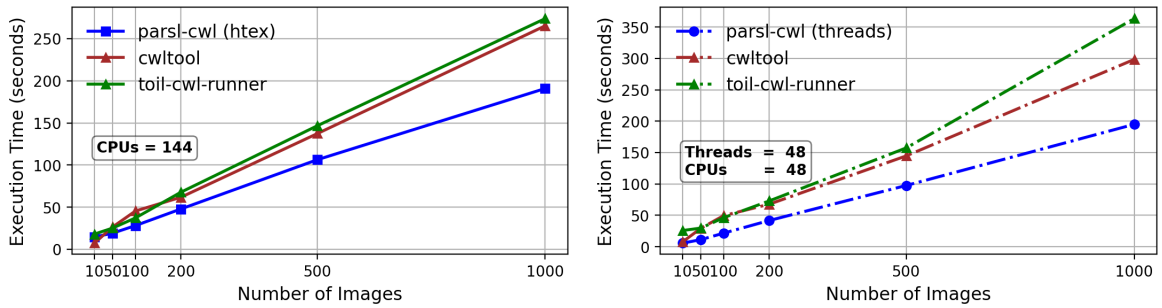
InlinePythonExpressions enable the inclusion of Python expressions anywhere in the workflow, which is particularly useful for input validation. Listing 6 shows this with the use of a *validate* field for each input. In this example, the CWL CommandLineTool has a validate field that invokes a Python function *valid_file* to verify that the input data file is of type '.csv'. This validation is done before the execution of the CWL CommandLineTool allowing validation to be incorporated directly within the CWL file rather than this being handled externally.

```
cwlVersion: v1.2
class: CommandLineTool

requirements:
  - class: InlinePythonRequirement
    expressionLib:
    - |
      def valid_file(file, ext):
          """
          Check if a file is valid

          Args:
              file (str): Path to the file
              ext (str): Expected file extension

          Raises:
              Exception: If the file is invalid
          """

          if not file.lower().endswith(ext):
              raise Exception(f"Invalid file.
                  Expected '{ext}'")

baseCommand: cat

inputs:
  data_file:
    type: File
    validate: |
        f"{valid_file($(inputs.data_file), '.csv')
            }"
    inputBinding:
      position: 1

outputs:
  validated_output:
    type: stdout
```

Listing 6: InlinePythonExpression example to verify that a given input file is a CSV file

These Python expressions can support multiple use cases:

- **Dynamic Input Validation**: Python can be used to implement complex input validation logic, ensuring data integrity and preventings runtime errors. For example, fields can be checked for valid ranges, formats, or dependencies, with exceptions raised for invalid data.
- **Error Handling**: Python's exception-handling mechanisms can be utilized to manage errors, enhancing workflow reliability.



(a) Three nodes

(b) Single node

Fig. 1: Runtimes for CWL image processing workflow using CWLTool, Toil and Parsl-CWL on three nodes and one node

- **Conditional Defaults**: Inline Python allows the specification of default values based on other inputs, enabling dynamic parameter management. This capability is particularly useful for workflows that require adaptive default values derived from related inputs.
- **External Python files**: External python files can be imported making the functions and variables available to be used in other parts of the CWL document.

VI. Evaluation

The primary contribution of our work is the integration of CWL and Parsl and support for inline Python expressions in CWL. The examples presented illustrate the functional capabilities realized from this work. Here, we briefly explore the performance of our CWL and Parsl integration as well as Python expressions.

1) CLW + Parsl: We evaluate our integration by comparing execution of a CWL workflow using Parsl with ones executed using cwltool and Toil.

We implement a CWL workflow using the image processing workflow detailed in Listing 3 with a wrapper to process a list of images. The wrapped workflow uses the scatter method to call the sub workflow on each image individually. This approach ensures that the execution of the image processing workflow on each image is independent, allowing cwltool and Toil to leverage parallel execution of these independent steps.

We conducted our experiments on a high-performance computing cluster located in our department. We used two configurations: 1) a single-node configuration using local threads or processes; 2) a distributed configuration using three nodes. Each node in our cluster is equipped with two 12-core Intel x86_64 processors (48 logical CPUs) and 126GB of RAM. We configured cwltool with the `parallel` option and toil-cwl-runner with the `slurm` batch system. We configured the Parsl workflow using the example in Listing 4 and using the HighThroughputExecutor, for three nodes, and ThreadPoolExecutor for the single-node deployment. We configured each workflow system to use all cores available on the allocated nodes.

Figure 1 shows a linear scaling trend as the number of images increases in both three node and single node deployments. Using three nodes (Figure 1a), Parsl-CWL with the HighThroughputExecutor achieves approximately 1.5 times better performance than CWLTool when processing a workload of 1,000 images. Similarly, on a single node (Figure 1b), Parsl-CWL using the ThreadPoolExecutor outperforms CWLTool by about the same factor for the same workload.

2) Python Expressions in CWL: We now consider the time to evaluate InlinePythonExpressions compared to CWL's InlineJavaScriptExpressions. We use the simple workflow from Listing 5 where the expression simply changes the case of a set of words. We deploy the workflow on a single node on the HPC cluster. We scale the number of words and record the time to complete the workflow. In Fig. 2, we see a short time to process up to 1024 words using the InlinePythonExpression, a constant performance for the Inline Python Expressions. We see a superlinear increase in time for JavaScript Expressions using both cwltool and Toil.

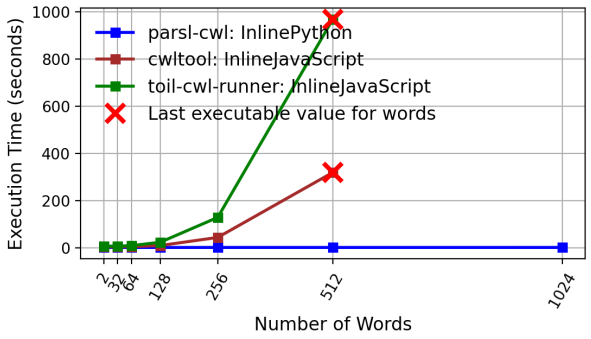


Fig. 2: Runtime for CWL InlineJavaScript processing using CWLTool, Toil and InlinePython using Parsl-CWL as we increase number of words from 2 to 1024

VII. Related Work

Many workflow systems—353 at the time of writing [13]—enable the orchestrated execution of multiple applications at both small and large-scale. Well-known examples include Pegasus [14], Galaxy [15], Swift [11], RCT [16], NextFlow [17], FireWorks [18], Apache Airflow [19], and Luigi [20]. These

systems are differentiated by their language, static workflow definition, explicit graph definition, or focus on specific domains or communities. Community workflow specifications such as CWL and WDL [21] have become increasingly popular; however, neither WDL or CWL are themselves workflow engines, and instead they may be executed using one of several supported workflow engines.

CWL has developed an extensive ecosystem of runners. The reference implementation CWLTool and Toil [2] are both implemented in Python. However, while they may be written in Python, the runner itself does not expose the Python side of the tool, instead providing a command line interface for users to run CWL workflows.

To the best of our knowledge, JavaScript is the only language supported by CWL expressions. Others have developed tools in Python and Java to support creation of CWL definitions directly from other languages and to provide parsing and validation of CWL definitions [22].

This paper builds on previous ideas, such as the concept of using common definitions for applications. For example, a previous paper [23] that included some of the authors of this paper suggested common configurations for both applications and systems, with the idea that the owners of those applications (developers) and systems (system administrators) could do a small amount of work to make these resources easy for researchers to use in their workflows.

VIII. Summary

The integration of CWL and Parsl enables Parsl workflows, written in Python, to easily leverage the ecosystem of CWL CommandLineTool definitions. It allows users to combine these two environments, allowing CWL CommandLineTools to be run in the flexible and scalable Parsl environment on a variety of computing resources. Our performance evaluation shows that Parsl can execute CWL CommandLineTools efficiently and comparably to existing CWL runners. To bring CWL towards the Python ecosystem we presented a prototype integration of Python Expressions within CWL workflow definitions. Our approach enhances workflow flexibility and expressiveness, enabling researchers to implement complex logic, validate inputs, and manage dependencies directly within their workflows using Python. Our future work focuses on developing our prototype into a robust toolkit for Parsl and adding support in Parsl to run complete CWL workflows.

IX. Acknowledgements

This work was supported in part by the National Science Foundation awards 2209919 and 2209920, the Chan Zuckerberg Initiative, and the U.S. Department of Energy under Contract DE-AC02-06CH11357.

References

[1] P. Amstutz, M. R. Crusoe, K. Ghose, J. Chilton, M. Franklin, B. Gavrilovic, S. Soiland-Reyes *et al.*, "Common Workflow Language (CWL) workflow description, v1.2.1," https://w3id.org/cwl/v1.2/, 2020.

[2] J. Vivian, A. A. Rao, F. A. Nothaft, C. Ketchum, J. Armstrong, A. Novak, J. Pfeil, J. Narkizian, A. D. Deran, A. Musselman-Brown, H. Schmidt, P. Amstutz, B. Craft, M. Goldman, K. Rosenbloom, M. Cline *et al.*, "Toil enables reproducible, open source, big biomedical data analyses," *Nature Biotechnology*, vol. 35, no. 4, pp. 314–316, Apr. 2017. [Online]. Available: https://doi.org/10.1038/nbt.3772

[3] Arvados Project, "Arvados," https://arvados.org, accessed: 2024-08-07.

[4] Broad Institute, "Cromwell," https://github.com/broadinstitute/cromwell, accessed: 2024-08-07.

[5] Y. Babuji, A. Woodard, Z. Li, D. S. Katz, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J. M. Wozniak, I. Foster, M. Wilde, and K. Chard, "Parsl: Pervasive parallel programming in python," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '19, 2019, p. 25–36.

[6] M. R. Crusoe, S. Abeln, A. Iosup, P. Amstutz, J. Chilton, N. Tijanić, H. Ménager, S. Soiland-Reyes, B. Gavrilović, C. Goble, and the CWL Community, "Methods included: standardizing computational reuse and portability with the Common Workflow Language," *Communications of the ACM*, vol. 65, no. 6, pp. 54—63, May 2022.

[7] C. Goble, "Implementing fair digital objects in the eosc-life workflow collaboratory," https://zenodo.org/record/4605654, Mar. 2021.

[8] K. Chard, S. Tuecke, and I. Foster, "Efficient and secure transfer, synchronization, and sharing of big data," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 46–55, 2014.

[9] R. Chard, Y. Babuji, Z. Li, T. Skluzacek, A. Woodard, B. Blaiszik, I. Foster, and K. Chard, "funcX: A federated function serving fabric for science," in *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, Jun 2020.

[10] J. G. Pauloski, V. Hayot-Sasson, M. Gonthier, N. Hudson, H. Pan, S. Zhou, I. Foster, and K. Chard, "Taps: A performance evaluation suite for task-based execution frameworks," in *IEEE Conference on eScience*, 2024.

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

[12] E. V. Smith, "Literal string interpolation," Python Software Foundation, PEP 498, 2015.

[13] "Computational data analysis workflow systems," https://s.apache.org/existing-workflow-systems.

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

[15] J. Goecks, A. Nekrutenko, and J. Taylor, "Galaxy: A comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences," *Genome Biology*, vol. 11, no. 8, p. R86, 2010.

[16] V. Balasubramanian, S. Jha, A. Merzky, and M. Turilli, "RADICAL-Cybertools: Middleware building blocks for scalable science," 2019, arXiv 1904.03085.

[17] P. Di Tommaso, M. Chatzou, E. W. Floden, P. P. Barja, E. Palumbo, and C. Notredame, "Nextflow enables reproducible computational workflows," *Nature Biotechnology*, vol. 35, no. 4, p. 316, 2017.

[18] A. Jain, S. P. Ong, W. Chen, B. Medasani, X. Qu, M. Kocher, M. Brafman, G. Petretto, G.-M. Rignanese, G. Hautier, D. Gunter, and K. A. Persson, "FireWorks: A dynamic workflow system designed for high-throughput applications," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 17, pp. 5037–5059, 2015. [Online]. Available: https://doi.org/10.1002/cpe.3505

[19] "Airflow," 2024, https://airflow.apache.org/.

[20] "Luigi," https://github.com/spotify/luigi.

[21] "Workflow description language," https://github.com/openwdl/wdl/blob/main/versions/1.0/SPEC.md.

[22] "CWL Java SDK," https://github.com/common-workflow-language/cwljava.

[23] J. Stubbs, S. Marru, D. Mejia, D. S. Katz, K. Chard, M. Dahan, M. Pierce, and M. Zentner, "Toward interoperable cyberinfrastructure: Common descriptions for computational resources and applications," in *Practice and Experience in Advanced Research Computing*, 2020.