NCSA SPIN Internship Research Plan

Project Title: Building Coherence between Parsl and CWL

Investigators: Daniel S. Katz, Assistant Director for Scientific Software and Applications

Yuhang Ren, Undergraduate Student

Abstract

The central topic of this project is to build coherence between Parsl and CWL, two parallel programming solutions to the problems of running a series of application tasks (apps) with dependencies. To achieve such coherence, I studied how to define CWL apps based on defined apps in Parsl programs, as well as how to capture the data about a Parsl program after it has been run into a machine-written CWL workflow.

Methodology

During this project, I first acquired knowledge of Parsl and CWL to form an idea of what these languages do. After organizing the information about these two languages, I prescribed how to define and run Parsl bash_apps in a way that CWL can reproduce them. Finally, I worked on a program that captures running data about a Parsl program after it has been run and builds a CWL workflow that is intended to reproduce the procedures in the program.

Introduction

While Python is a well-known programming language widely used due to its simplicity, ability to integrate with other languages, and its interpreted nature, it lacks ability to run programs in parallel. One means of allowing parallel programming in Python is Parsl, a Python library for programming and executing data-oriented workflows (dataflows) in parallel [1]. Python programs that use Parsl allow selected Python functions and external applications (called apps) to be connected by shared input/output data objects into flexible parallel workflows.

However, while Parsl provides substantial parallel programming support for Python, it does so in a different way from what is specified by Common Workflow Language (CWL), a specification common in the life sciences (and to a lesser extent, in other fields) for describing analysis workflows and tools in a way that makes them portable and scalable across a variety of software and hardware environments [2]. Parsl uses decorators to programmatically indicate which functions should be run in parallel, and at runtime uses this information to build up an internal representation of a graph of tasks and dependencies, while CWL requires the user to explicitly represent a program as a

set of apps and dependencies, stored in a data structure, rather than a program. This difference forms a potential obstacle for programmers using both Parsl and CWL, where they might have to separately write two versions of a code that performs the same function, adding unnecessary cost.

To save the cost of writing the same codes in two different languages, we can explore making Parsl and CWL as interoperable as possible, so that programmers writing in either language can share and use codes of the other without having to rewrite the complete application in their own language again. In practice, going from Parsl to CWL has potential utility in that CWL can store the exact set of apps that were run in Parsl, while going from CWL to Parsl has less clear utility, so I have studied representing Parsl programs in CWL.

As the name suggests, CWL is a (somewhat) common workflow language that is accepted by a wide range of communities. Hence one practical way to obviate the obstacle between Parsl and CWL is to make Parsl programs CWL-friendly, which includes following a set of restrictions to make Parsl programs able to be captures into CWL programs, as well as to automatically translate Parsl programs to CWL, saving human labor.

To create a program that automatically translate a Parsl program into CWL, I use the monitoring module in Parsl to create a database that captures all the tasks executed in the program, because interpreting a Parsl program which is written in Python by machine requires an exhaustive and undesirable effort.

Results

1. The following restrictions on Parsl programs are required:
    a) Bash apps cannot call Python functions. CWL cannot execute codes written in Python, so we need to avoid using Python functions in a bash app. String formatting is still ok to translate into CWL.
    b) Limit the input and output data types of apps. Inputs that CWL can handle include primitive types such as string, int, long, float, double and null, complex types i.e. arrays and records, and special types such as File, Directory and Any. Outputs for CWL are either files or content in files.
    c) Avoid using environment variables in the return value of a bash_app. Environment variables needs redefinition in CWL.
    d) Include the non-input arguments in the input keyword. This is because bash_app parameters outside inputs/outputs keywords cannot be stored in

the database produced by execution of the program.

2. Program reproducing a Parsl program in CWL: My program succeeded in transforming a sample Parsl workflow into a CWL workflow. The program as well as demonstration and instructions to utilize it can be acquired on GitHub: https://github.com/Parsl/student-projects.

Example

1. To demonstrate how the restrictions on Parsl programs work, the following is an example of a strictly formatted Parsl bash_app, which calls a shell script with one argument. The command is first formatted as a string and is then returned. The sting to be returned stands for the command to be executed, where an executable is called with arguments following it. The executable and arguments should be specified in inputs and outputs of the bash_app, as prescribed in Restriction d.

```
@bash_app
def echo (inputs=[], outputs=[], stdout=parsl.AUTO_LOGNAME,
stderr=parsl.AUTO_LOGNAME):
    command = '{exe} {output}'.format(
        exe    = inputs[0],
        output = outputs[0]
    )
    return command
```

2. The following is an example workflow that my program handles.

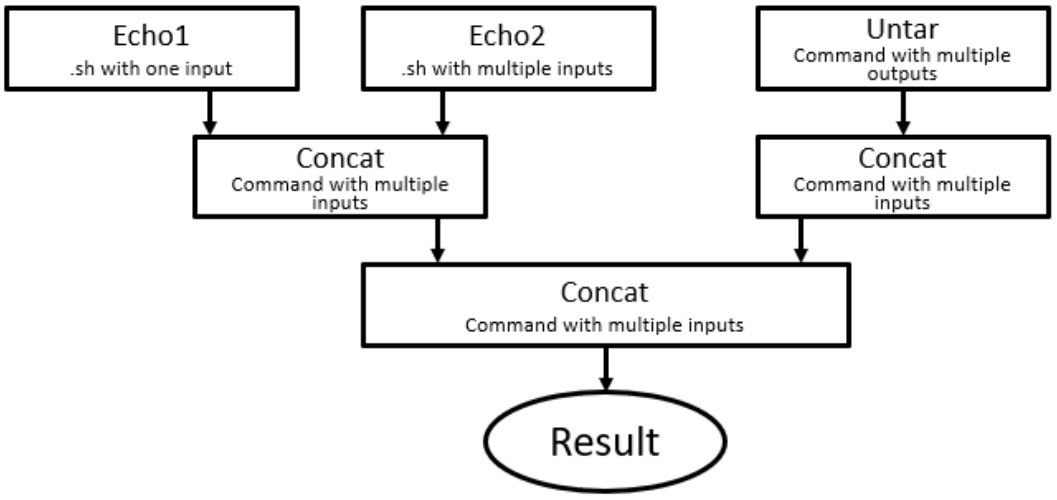

Figure 1: Example workflow. Boxes stand for bash_apps, and arrows suggest dependencies. The program at the bottom generates the final result.

Conclusion

Parsl is a useful way to write parallel workflows in Python. However, to make it possible to capture these workflows as run into CWL, the Common Workflow Language, some restrictions must be followed. This also allows Parsl programmers to transform Parsl programs into the CWL language, which also serves to reproduce

relatively dynamic Parsl programs. I have built and tested a software tool that can be utilized for this purpose.

References

[1] Y. Babuji, A. Woodard, B. Clifford, Z. Li, D. S. Katz, R. Chard, R. Kumar, L. Lacinski, J. Wozniak, I. Foster, M. Wilde, K. Chard, "Parsl: Pervasive Parallel Programming in Python," 28th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC 2019), doi: 2019.10.1145/3307681.3325400 (preprint: arXiv 1905.02158)

[2] Peter Amstutz, Michael R. Crusoe, Nebojša Tijanić (editors), Brad Chapman, John Chilton, Michael Heuer, Andrey Kartashov, Dan Leehr, Hervé Ménager, Maya Nedeljkovich, Matt Scales, Stian Soiland-Reyes, Luka Stojanovic (2016): **Common Workflow Language, v1.0**. Specification, *Common Workflow Language working group*. https://w3id.org/cwl/v1.0/ doi:10.6084/m9.figshare.3115156.v2