ATL-SOFT-PROC-2024-001
22 July 2024

A Function-as-a-Task Workflow Management Approach with PanDA and iDDS

Wen Guan[1,*], *Tadashi* Maeno[1,**], *Torre* Wenaus[1], *Aleksandr* Alekseev[2], *Fernando Harald* Barreiro Megino[3], *Kaushik* De[3], *Edward* Karavakis[1], *Tatiana* Korchuganova[2], *FaHui* Lin[2], *Paul* Nilsson[1], *Zhaoyu* Yang[1], *Rui* Zhang[4], and *Xin* Zhao[1]on behalf of the ATLAS Computing Activity

[1]Brookhaven National Laboratory, Upton, NY, USA
[2]University of Pittsburgh, Pittsburgh, PA, USA
[3]University of Texas at Arlington, Arlington, TX, USA
[4]University of Wisconsin-Madison, Madison, USA

Abstract. The growing complexity of high energy physics analysis often involves running various distinct applications. This demands a multi-step data processing approach, with each step requiring different resources and carrying dependencies on preceding steps. It's important and useful to have a tool to automate these diverse steps efficiently.
With the Production and Distributed Analysis (PanDA) system and the intelligent Data Delivery Service (iDDS), we provide a platform that describes data processing steps as tasks and their sequences as workflows, seamlessly orchestrating their execution in a specified order and under predefined conditions, thereby automating the entire task sequence. In this presentation, we will start by giving an overview of the platform's architecture. Following that, we'll introduce a user-friendly interface where workflows are defined in Python with multiple code blocks, with each block being implemented as a Python function. We will then explain the process of converting Python functions into executable tasks, scheduling them across distributed heterogeneous resources, and managing their outputs through a messaging-based asynchronous result-processing mechanism. Finally, we'll showcase a practical example illustrating how this platform effectively converts a machine learning hyperparameter optimization processing to a distributed workflow in ATLAS at LHC.

1 Introduction

The Production and Distributed Analysis (PanDA) system [1, 2] manages workloads originating from distributed users across diverse computing resources. One of PanDA's key advantages is its ability to abstract underlying computing resources, offering users a unified interface to submit and manage workloads. This allows users to submit workloads without needing knowledge of the computational resources, providing essential transparency in a distributed environment.

*e-mail: wguan2@bnl.gov
**e-mail: tmaeno@bnl.gov
Copyright 2023 CERN for the benefit of the ATLAS Collaboration. CC-BY-4.0 license.

The intelligent Data Delivery Service (iDDS) [3, 4] is a workflow orchestration system in PanDA developed to automate the execution of complex and dynamic workflows. It offers advanced features such as workflow descriptions with Directed Acyclic Graphs (DAG), conditional branching, iterative sequences, and polymorphic workloads. Its flexibility in handling complex workflows makes it suitable for a range of automation scenarios, enhancing the ability of experiments to manage and process data efficiently.

With PanDA and iDDS, we have constructed a framework for distributed large-scale workflow management. It has been successfully applied to support various use cases in production, involving extensive data processing and different physics analyses in ATLAS [5] at the LHC [6], Rubin Observatory (LSST) [7, 8] and other experiments. For example, in ATLAS, it supports fine-grained data carousels, hyperparameter optimization (HPO), and active learning; In the Rubin Observatory, it manages DAGs for sequencing data processing [9].

As the number of use cases increases, managing different logical requirements and dependencies has become more complex. We have developed a new approach using Python decorators to convert local functions into distributed PanDA workloads. This enables easy definition and management of workflow conditions and dependencies using Python.

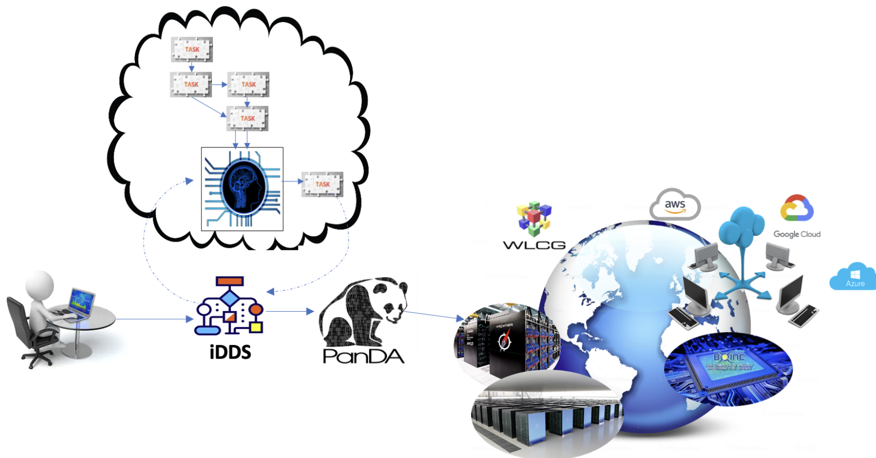


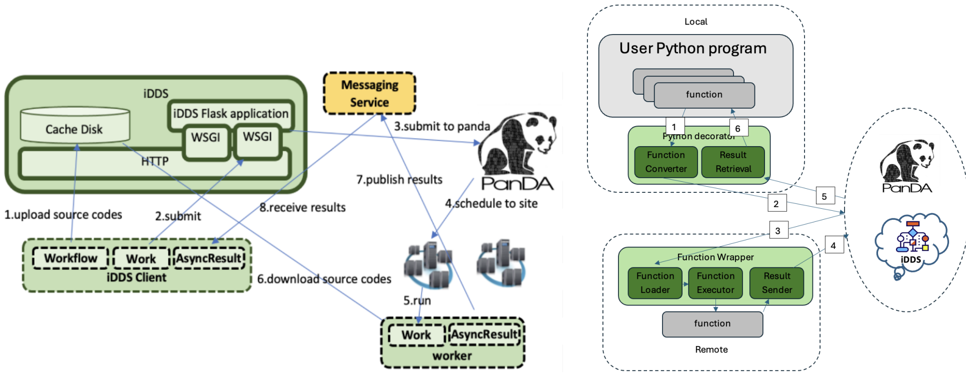
Figure 1: An integrated workflow with PanDA and iDDS, where iDDS automates complex and dynamic workflows, and PanDA schedules workloads to large-scale distributed heterogeneous computing resources.

2 Function-as-a-Task

iDDS coordinates and orchestrates task execution and data motion to streamline operations, improving automation and efficiency. It automatically aggregates results from previous workloads to trigger new ones, enhancing overall efficiency in large-scale operations.

However, emerging use cases, particularly those related to machine learning (ML), require sophisticated condition handling and task dependencies, which often defy straightforward descriptions in existing schemes. This complexity can result in discrepancies between user expectations and system behaviors when defining complex dependency logic.

The function-as-a-task scheme simplifies the user interface for defining workflows. In this approach, users write standard Python programs and indicate selected functions with Python



(a) The architecture to manage function-as-a-task workflow (b) Convert functions to distributed jobs

Figure 2: Function-as-a-task workflow: (a) The architecture and related services used to serve function-as-a-task workflows; (b) The flow to convert a local function with Python decorator to PanDA jobs and execute the function at distributed computing resources.

decorators to be executed as distributed workloads. It offers flexibility for defining various logics and dependencies within workflows.

A workflow with the function-as-a-task scheme involves three main parts, as shown in Figure 2a:

- **Managing Source Codes and Context**: User source codes and contexts are zipped into a file and uploaded to an HTTP cache service. Before loading functions on distributed computing resources, the function wrapper downloads the source codes from the HTTP cache and initializes the context. The execution context can be container-based, Conda-based [10], or a standard Python environment.
- **Converting and Executing Functions**: Local functions are converted to PanDA jobs and executed on distributed computing resources. Python decorators are used to convert local functions and their parameters into strings that can be wrapped into PanDA jobs. When these jobs are scheduled to distributed resources, the function wrapper loads and executes the functions.
- **Asynchronous Result Retrieval**: The function result is transferred back from the function executor to the function caller using an asynchronous result retrieval service. When the function finishes, the result is sent back to the original function caller via this service, as illustrated in Figure 2b.

The asynchronous result retrieval service facilitates the delivery of function outputs, supporting both STOMP [11]-based and HTTP REST [12]-based services. Initially, it uses STOMP to transfer data via an ActiveMQ [13] server efficiently. If there are issues loading STOMP or accessing the ActiveMQ server, it falls back to the HTTP REST service. The HTTP REST service communicates with the ActiveMQ server to keep a copy of the data temporarily. The data receiver subscribes to the ActiveMQ service if possible; otherwise, it falls back to the HTTP REST service.

The function-as-a-task scheme enables users to transparently run functions on distributed computing resources via the PanDA system. It offers useful solutions for managing user source codes, allowing the execution of complex functions. Leveraging the existing PanDA

infrastructure and grid middleware, it requires no additional setup for sites and can scale significantly. The asynchronous result retrieval service, based on a messaging publish-subscribe model, enhances efficiency between function execution and the function caller.

3 Experiments

3.1 HyperParameter Optimization (HPO)

Listing 1: A Distributed HPO Example

```
@work(map_results=True)
def optimize_work(opt_params, retMethod=None, hist=True,
                  saveModel=False, input_weight=None, **kwargs):
...
@workflow
def optimize_workflow():
    ...
    for i in range(n_iterations):
        print("Iteration %s" % i)
        multi_jobs_kwargs_list = []
        for j in range(n_points_per_iteration):
            x_probe = bayesopt.suggest(util)
            multi_jobs_kwargs_list.append(x_probe)

        results = optimize_work(opt_params=params, opt_method=opt_method,
                                retMethod=opt_method, multi_jobs_kwargs_list=multi_jobs_kwargs_list)
    ...
```

request id	username	workflow status	graph	workflow name	created on (UTC)	total tasks	tasks	transform type	total files	released files	unreleased files	finished files
6128	Wen Guan	Finished	plot	optimize_iworkflow.optimize_workflow_2024_03_06_13_18_47	2024-03-06 13:18:47	11	Finished(10)	N/A	0	0	0	100%
6127	Wen Guan	Finished	plot	optimize_iworkflow.optimize_workflow_2024_03_06_13_18_41	2024-03-06 13:18:45	11	Finished(10)	N/A	0	0	0	100%
6126	Wen Guan	Finished	plot	optimize_iworkflow.optimize_workflow_2024_03_06_08_55_01	2024-03-06 08:55:04	6	Finished(5)	N/A	0	0	0	100%
6125	Wen Guan	Finished	plot	optimize_iworkflow.optimize_workflow_2024_03_06_08_54_32	2024-03-06 08:54:36	3	Finished(2)	N/A	0	0	0	100%

(a) Several example HPO workflows

tasks, sorted by jeditaskid-desc

ID Parent	Task name TaskType/ProcessingType Campaign Group User Errors Logged status	Task status Nfiles	Input files Nlost Nfinish % Nfail %
168807	optimize_iworkflow.optimize_iworkflow.optimize_work_2024_03_06_11_12_39_6125_47808 iDDS Wen Guan Errors	done 20	20 100%
168804	optimize_iworkflow.optimize_iworkflow.optimize_work_2024_03_06_08_57_10_6125_47805 iDDS Wen Guan Errors	done 20	20 100%
168801	optimize_iworkflow.optimize_workflow_2024_03_06_08_54_32_6125 iDDS Wen Guan Errors	done 1	1 100%

(b) Example tasks for a workflow

Figure 3: Example HPO workflows: (a) Several HPO workflows: Each iteration is mapped to a PanDA task. For instance, workflow 6128 has completed 10 tasks, corresponding to 10 iterations (The total number of tasks is 11 because the main workflow function is also mapped to a PanDA task in this case). (b) Example tasks for workflow 6125: This workflow has 2 optimization iterations with 20 concurrent jobs per iteration.

We applied the function-as-a-task scheme to a HyperParameter Optimization (HPO) workflow. As shown in Listing 1, the Python decorators **@workflow** and **@work** are used

to convert a local HPO program into a distributed workflow. In this example, the function **optimize_work** is converted into PanDA tasks and executed at remote sites.

Figures 3a and 3b show some experiment jobs using the HPO workflow. In this workflow, each iteration is mapped to a PanDA task with multiple concurrent jobs.

The function-as-a-task service allows users to update only a few lines of the original program to map it to a distributed program, making everything else transparent. This greatly simplifies the process of running tasks on distributed computing resources.

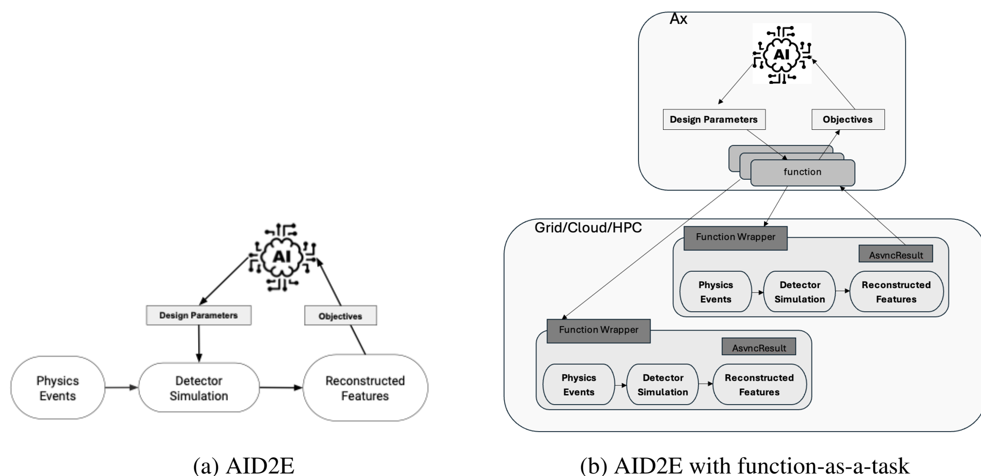
3.2 AI-assisted Detector Design at EIC (AID2E)

Our next step is to integrate the function-as-a-task scheme with the AID2E [14] project in EIC [15]. This project employs AI and an iterative approach to optimize detector design parameters, considering various detector objectives, as shown in Figure 4a.

Using PanDA/iDDS for highly scalable distributed workflows enables AID2E to address complex optimization problems across multiple detector parameters and design objectives at a scale that would otherwise be intractable.

In the initial iteration, we select multiple sets of detector parameters, conduct a detector simulation for each set, and evaluate various detector objectives, such as resolution. Subsequently, we determine new sets of detector parameters based on the objectives measured in the previous iteration and perform detector simulations to obtain the objectives, which are then used for the next iteration. This iterative process continues until we achieve satisfactory detector objectives.

Using the function-as-a-task service, we will map the local detector evaluation functions in AID2E to remote functions executed on distributed resources such as Grid [16], Cloud [17], or HPC [18], as illustrated in Figure 4b.



(a) AID2E

(b) AID2E with function-as-a-task

Figure 4: AID2E: (a) A framework with AI assistance suggests design parameters for multiple objectives [19]. For each group of design parameters, the EIC detector simulation evaluates these parameters to meet the multiple objectives. (b) With function-as-a-task, we convert local functions in AID2E to PanDA jobs and then execute the function at distributed computing resources.

4 Conclusions

The PanDA and iDDS systems have provided workflow orchestration to streamline data processing across diverse computing resources. In this paper, we introduced a new user-friendly interface for using workflows in PanDA and iDDS. This interface allows users to easily integrate local Python functions with PanDA and iDDS. The asynchronous result retrieval service enhances data transfer between different functions, improving workflow management.

In the future, we will continue our efforts to improve the system's robustness, making it a valuable tool for large scale distributed computing and machine learning in experiments such in ATLAS at LHC, Rubin Observatory and EIC.

Acknowledgements

This manuscript has been authored by employees of Brookhaven Science Associates, LLC under Contract No. DE-SC0012704 with the U.S. Department of Energy. The publisher by accepting the manuscript for publication acknowledges that the United States Government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes.

References

[1] T. Maeno et al., *PanDA: Production and Distributed Analysis System*, Computing and Software for Big Science (to be submitted)

[2] T. Maeno et al., *Overview of ATLAS PanDA Workload Management*, J.Phys.: Conf. Ser. **331**, 072024 (2011)

[3] W. Guan et al., *An intelligent Data Delivery Service for and beyond the ATLAS experiment*, EPJ Web Conf. **251**, 02007 (2021)

[4] W. Guan et al., *Towards an Intelligent Data Delivery Service*, EPJ Web Conf. **245**, 04015 (2020)

[5] ATLAS Collaboration, *The ATLAS Experiment at the CERN Large Hadron Collider*, J. Inst. **3**, S08003 (2008)

[6] L. Evans and P. Bryant (editors), *LHC Machine*, J. Inst. **3**, S08001 (2008)

[7] Z. Ivezic et al., *LSST: From Science Drivers To Reference Design And Anticipated Data Products*, Astrophys. J., **873**, 111 (2019)

[8] LSST Science Collaboration, *Lsst science book, version 2.0* (2009), `arXiv:0912.0201`

[9] W. Guan et al., *Distributed Machine Learning Workflow with PanDA and iDDS in LHC ATLAS*, EPJ Web Conf. **295**, 04019 (2024)

[10] *Conda package manager*, https://docs.conda.io/en/latest/

[11] *Simple Text Oriented Messaging Protocol*, https://stomp.github.io/

[12] *Representational State Transfer*, https://fr.wikipedia.org/wiki/Representational_state_transfer

[13] *Apache ActiveMQ: Flexible and powerful open source multi-protocol messaging*, https://activemq.apache.org/

[14] M. Diefenthaler et al., *Ai-assisted detector design for the eic (aid(2)e)*, Journal of Instrumentation **19**, C07001 (2024)

[15] R. Ent, *Electron ion collider, in the proceedings of the jluo meeting on nsac long range plan*, https://indico.jlab.org/event/589/contributions/10644/attachments/8403/12002/EICTJJLUOLRP2022.pptx (2022)

[16] *WLCG*, https://wlcg.web.cern.ch/

[17] F. Barreiro Megino et al., *Seamless Integration of Commercial Clouds with ATLAS Distributed Computing*, EPJ Web Conf. **251**, 02005 (2021)

[18] *High-performance computing*, https://en.wikipedia.org/wiki/High-performance_computing

[19] C. Fanelli et al., *Ai-assisted optimization of the ecce tracking system at the electron ion collider*, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment **1047**, 167748 (2023)