**Liam Huber**
*Grey Haven Solutions*
liamhuber@greyhavensolutions.com

Jörg Neugebauer
*Max Planck Institute für Nachhaltige Materlialien*

## *From concept to compute:*

# Accelerating research with workflow management in `pyiron`

github.com/liamhuber/scipy2024

pyiron

Grey Haven Solutions

### *Background & Motivation*

The pyiron project was conceived to accelerate the development of research codes, originally and primarily in the domain of computational materials science (atomistics). In its original formulation, pyiron is a python framework which abstracts away many of the pain-points computational researchers often encounter, including data management, scaling calculations to high performance computing (HPC), etc. In this form, the end result of a pyiron "workflow" is typically a Jupyter notebook. While powerful and flexible, this paradigm makes it difficult to track the provenance of data moving through the workflow, or to express the workflow in any form other than a python script. Further, we find that the object oriented programming of the framework makes it difficult for many researchers (who typically do not have formal computer science educations) to extend pyiron's functionality to new domains.

In this context, we have re-imagined pyiron as a more formal workflow management tool using graph node based computational units, `pyiron_workflow`. Here, we explore the philosophy and functionality of this revised paradigm, first using a simple "learning" example which requires only the core Workflow object and the code here, and then for a more realistic scientific application where additional packages of pre-constructed nodes are used (full code can be found at github.com/liamhuber/scipy2024).

### *Functional*

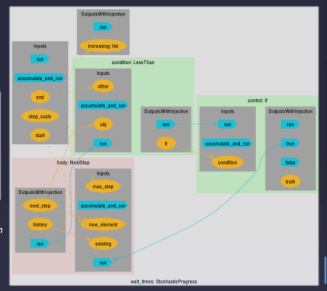
New node classes can be made with a decorator, so the barrier to entry is as low as writing a plain python function. Output labels can be provided, or scraped automatically from return values in the source code.

### *Composable*

Building up complex macros is simple as a python function defining how to construct a sub-graph from other nodes. These can be nested to arbitrary depth.

### *Rigorous*

By formulating the workflow as a graph, non-code representations (e.g. graphical below) are trivial. They are also unambiguously defined (no hidden state or execution order dependence associated with Jupyter notebooks); can be trivially re-executed with different input due to clearly defined interfaces; and can be re-stated as macros for inclusion in other workflows. *A-priori* workflow provenance is given by the graph structure, while *post-facto* provenance is tracked for both node execution and node completion (which can differ using parallel node computation).



*Above: top-level workflow graph*
*Right: While-loop macro sub-graph*

### *Flexible*

A formal separation of data and execution flows mean that directed acyclic graphs (DAGs) are automated and easy, but *cyclic* graphs are also possible! Syntactic sugar makes it easy to specify the flow of data by assigning node output to node input with `=`, while execution flow can be defined with `>>`.

### *Interoperable*

If provided, python's type hints are (optionally) strictly adhered to when forming data connections, turning workflows into a strongly-typed language. This improves reliability when combining nodes from different sources.

### *Practical*

Common routines are provided with a built-in standard node library. Further, almost all python operators are supported by dynamically adding new nodes to delay the operation until execution time; keep node definitions as close to writing plain python code as possible!

### *Scalable*

Each node is able to receive a `concurrent.futures.Executor`-compliant executor for parallel processing. This includes full compatibility with pyiron's `pympipool` package which has tools for flux, slurm, etc.

*Roadmap:* Complete integration with pyiron's `pysqa` to natively support sending node computations to HPC resources.

### *Storable*

Although new node classes are dynamically defined by decorating functions, the entire node ecosystem is fully `pickle` compliant by leveraging tools in `pyiron_snippets.factory`.

*Roadmap:* Hierarchical storage using `h5io` leveraged by some other pyiron packages is supported for special cases; extend the HDF5 interface to arbitrary python objects to support post-execution data scraping and partial loading

## *Learning*

A complete example

```python
import pickle
from random import random
from time import sleep, time

from pyiron_workflow import Workflow


@Workflow.wrap.as_function_node("seconds", "minutes", "hours")
def Sleep(sleep_seconds: float | int) -> tuple[float, float, float]:
    sleep(sleep_seconds)
    return float(sleep_seconds), sleep_seconds / 60., sleep_seconds / 3600.


@Workflow.wrap.as_macro_node()
def NextStep(self, existing: list | None, new_element, max_step):
    self.history = Workflow.create.standard.AppendToList(
        existing=existing,
        new_element=new_element,
    )
    self.random = Workflow.create.standard.PureCall(random)
    self.next_step = self.history[-1] + max_step * self.random
    return self.history, self.next_step


@Workflow.wrap.as_macro_node("increasing")
def StochasticProgress(
    self,
    start: int | float = 0,
    end: int | float = 3.13,
    step_scale: int | float = 1.0,
) -> list[float]:
    self.body = NextStep(None, start, step_scale)
    self.body.inputs.existing = self.body.outputs.history
    self.body.inputs.new_element = self.body.outputs.next_step

    self.condition = self.body.outputs.next_step < end
    self.control = Workflow.create.standard.If(condition=self.condition)

    self.starting_nodes = [self.body]
    self.body >> self.condition >> self.control
    self.control.signals.output.true >> self.body

    return self.body.outputs.history


wf = Workflow("demo")

wf.wait_times = StochasticProgress()
wf.wait = Workflow.create.for_node(
    body_node_class=Sleep,
    iter_on=("sleep_seconds",),
    sleep_seconds=wf.wait_times
)

t0 = time()
with Workflow.create.ThreadPoolExecutor(max_workers=8) as exe:
    wf.wait.body_node_executor = exe
    wf()
t1 = time()

print(
    f"Body node sleep times "
    f"{[round(t, 2) for t in wf.wait_times.outputs.increasing.value]}s"
)
>>> Body node sleep times [0, 0.59, 1.35, 2.2, 2.89]s

print(f"Total runtime {(t1 - t0):.2f}s")
>>> Total runtime 3.07s

print(
    "Loop body node execution provenance",
    [l.replace("body_", "") for l in wf.wait.provenance_by_execution if "body" in l]
)
>>> Loop body node execution provenance ['4', '3', '2', '1', '0']

print(
    "Loop body node completion provenance",
    [l.replace("body_", "") for l in wf.wait.provenance_by_completion if "body" in l]
)
>>> Loop body node completion provenance ['0', '1', '2', '3', '4']

reloaded = pickle.loads(pickle.dumps(wf))
print(all(reloaded.wait.outputs.df.value == wf.wait.outputs.df.value))
>>> True

wf.draw(depth=0).render("workflow", format="eps")
wf.wait_times.draw().render("while_macro", format="eps")
```

## DOING

Importing additional nodes

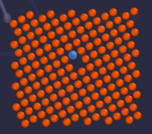
```python
from ase import Atoms
from pyiron_workflow import Workflow

import ase_nodes as an
import my_nodes as mn

@Workflow.wrap.as_macro_node()
def SizeConverged(
    self,
    host: str,
    solutes: list[str],
    optimizer: an.optimizers.DelayedOptimizer,
    filter_: an.filters.DelayedFilter,
    cubic_host: bool = True
) -> tuple[int, Atoms]:
    """Find the largest solute based on ASE's guess of the per-atom volume."""
    self.biggest_solute = mn.BiggestSpeciesGuess(species=solutes)
    self.unit = an.atoms.Bulk(host, cubic=cubic_host)
    self.cheap_calculator = an.calculators.EMT()
    self.converged_repeats = mn.IncreaseSizeWhileEnergyChanges(
        host_unit_cell=self.unit,
        solute=self.biggest_solute,
        calculator=self.cheap_calculator,
        optimizer=optimizer,
        filter_=filter_,
        parent=self,
        energy_threshold=0.001  # Intentionally "large" for the sake of speed
    )
    self.supercell = an.atoms.Repeat(self.unit, self.converged_repeats)
    return self.converged_repeats, self.supercell

@Workflow.wrap.as_macro_node("repeats", "supercell")
def ScanCandidates(
    self,
    supercell: Atoms,
    solutes: list[str],
    optimizer: an.optimizers.DelayedOptimizer,
    filter_: an.filters.DelayedFilter,
) -> tuple[str, list[Atoms]]:
    """Scan binding energies for multiple solutes with a classical simulator"""
    self.binding_energies = Workflow.create.for_node(
        mn.LAMMPSSoluteVacancyBinding,
        iter_on=("solute",),
        bulk=supercell,
        solute=solutes,
        optimizer=optimizer,
        filter_=filter_,
        parent=self,  # So the semantic path is available at instantiate for file paths
    )

    self.strongest_solute = self.binding_energies.outputs.df["solute"](
        Workflow.create.standard.PureCall(
            self.binding_energies.outputs.df["binding_energy"].idxmin, parent=self
        )
    )

    self.is_strongest = Workflow.create.standard.Equals(
        self.binding_energies.outputs.df["solute"],
        self.strongest_solute
    )
    self.relaxed = self.binding_energies.outputs.df["atoms"][self.is_strongest].iloc[0]
    return self.strongest_solute, self.relaxed

@Workflow.wrap.as_macro_node("energy")
def DFTBinding(
    self,
    binding_structures: list[Atoms],
    optimizer: an.optimizers.DelayedOptimizer,
    filter_: an.filters.DelayedFilter,
) -> float:
    """Finally, do a quantum mechanical calculation"""
    self.expensive_calculator = an.calculators.Espresso(
        input_data={
            "tstress": True,
            "electron_maxsteps": 1,
            "scf_must_converge": False,
        },  # Just the cheap parameters -- this is a demo on a laptop, not real science
        n_cores=4,
        oversubscribe=True,
    )
    self.dft_binding = an.physics.BindingEnergy(
        a=binding_structures[0],
        b=binding_structures[1],
        ab=binding_structures[2],
        neither=binding_structures[3],
        calculator=self.expensive_calculator,
        optimizer=optimizer,
        minimize_inputs=an.physics.MinimizeInput(steps=0),  # Again, be cheap not right
        filter_=filter_,
        parent=self,  # So the semantic path is available at instantiate for file paths
    )
    return self.dft_binding.outputs.binding_energy


wf = Workflow("binding_example")

wf.host = Workflow.create.standard.UserInput("Al")
wf.host.inputs.user_input.type_hint = str
wf.solutes = Workflow.create.standard.UserInput(["Cu", "Au", "Ni", "Pd", "Pt"])
wf.solutes.inputs.user_input.type_hint = list[str]

wf.optimizer = an.optimizers.BFGS()  # For ionic relaxation
wf.filter = an.filters.UnitCellFilter()  # For cell relaxation

wf.converged = SizeConverged(
    host=wf.host,
    solutes=wf.solutes,
    optimizer=wf.optimizer,
    filter_=wf.filter
)
wf.scanned = ScanCandidates(
    supercell=wf.converged.outputs.supercell,
    solutes=wf.solutes,
    optimizer=wf.optimizer,
    filter_=wf.filter
)
wf.dft_binding = DFTBinding(
    binding_structures=wf.scanned.outputs.relaxed,
    optimizer=wf.optimizer,
    filter_=wf.filter
)

with Workflow.create.ProcessPoolExecutor(max_workers=4) as exe:
    wf.scanned.binding_energies.body_node_executor = exe
    wf()

print(wf.outputs.to_value_dict())
>>> {'converged__repeats': 3,
... 'scanned__strongest_solute': 'Pt',
... 'dft_binding__energy': -0.09048996041383361}
```

### *Domain*

Computational materials science for metallurgy often probes material systems at the level of individual atoms, using either classical "potential" models for ions, or fully quantum mechanical representations that also solve for the electronic state.

### *Problem*

Material behaviour often depends strongly on atomic-scale details of how chemistry (alloying species) interact with spatial defects in the host crystal lattice. The simplest example of this is an isolated solute atom interacting with an isolated lattice vacancy. The potential energy binding these two defects together is relatively straightforward but still of real scientific interest. The schematics below show a pseudo-2D configuration of host and solute atoms with a vacancy, and how to compute the binding energy in four calculations with negative energies as favourable binding.



$$E_{\mathrm{bind}} = (E_{\mathrm{bulk}} + E_{X,\mathrm{vac}}) - (E_X + E_{\mathrm{vac}})$$

### *Attack*

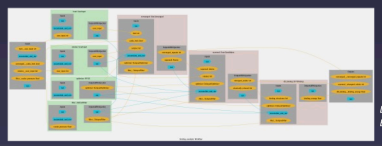
Atomic simulators for metals typically use periodic boundary conditions, where the simulation domain effectively repeats itself infinitely in each direction, so we first need to converge the size of this simulation domain. For this we find the "largest" solute (likely to have the strongest elastic interaction with its own periodic images), then use a very simple model to converge its binding energy. Next, we use a more sophisticated (but still classical) model to scan across a variety of solutes to quickly find an interesting candidate. Finally, we perform a fully quantum mechanical simulation of the most interesting solute.

### *Implementation*

We compose the workflow of three macros, one for each of the attack steps above. These macros in turn use `pyiron_workflow` nodes which have been written to wrap the popular ase (atomic simulation environment) python library (all code available on the presentation repo linked near the title). The choice of `ase` over one of pyiron's own toolsets for atomistic simulations is intentional to demonstrate the flexibility of the `pyiron_workflow` paradigm to adapt to a wide variety of existing codes, even those with distinctly non-functional (i.e. very OOP) paradigms. In turn, `ase` gives us a variety of atomistic tools, including parsers to write and read file-based IO for the classical LAMMPS simulator and Quantum Espresso density functional theory (DFT) code.

### *Result*

This example exposes the host and solute species as input, and `pyiron_workflow` allowed us to build up a simple, physically meaningful graph by composing together multiple levels of nested macro node (below). This workflow strings together two completely independent file-based executables and shows how we can parallelize over them by launching multiple python subprocesses, or by passing extra cores to an mpi invocation. The final result is easily digestible, and various components can be re-used and re-combined freely in other workflows.



*Left: The top-level workflow*
*Below: The workflow drawn to its full depth, resolving the bodies of each nested macro*

### *Looking forward*

This graph-based approach to workflows lends itself very well to visual scripting. We have constructed a prototype GUI for this, `ironflow`, which additionally leverages `pyiron_ontology` to provide ontological typing for node connections, providing a sort of *dynamic typing* that depends on the up- or down-stream use of data. E.g., in the figure to the right, the `Murnaghan` node requires an `engine` carrying "bulk-like" information. As long as the `Lammps` node output flows into this, it only sees the `BulkStructure` node (placement menu) and its output (graph connection, white) as recommendations; if this `Lammps`–`Murnaghan` connection is broken, additional structure generators appear as recommendations.