Digital Discovery

ROYAL SOCIETY OF CHEMISTRY

PAPER

View Article Online
View Journal | View Issue

Cite this: *Digital Discovery*, 2024, **3**, 2551

Navigating the Maize: cyclic and conditional computational graphs for molecular simulation

Thomas Löhr, *[a] Michele Assante,[bc] Michael Dodds,[ad] Lili Cao,[a] Mikhail Kabeshov,[a] Jon-Paul Janet,[a] Marco Klähn[a] and Ola Engkvist[ae]

Many computational chemistry and molecular simulation workflows can be expressed as graphs. This abstraction is useful to modularize and potentially reuse existing components, as well as provide parallelization and ease reproducibility. Existing tools represent the computation as a directed acyclic graph (DAG), thus allowing efficient execution by parallelization of concurrent branches. These systems can, however, generally not express cyclic and conditional workflows. We therefore developed Maize, a workflow manager for cyclic and conditional graphs based on the principles of flow-based programming. By running each node of the graph concurrently in separate processes and allowing communication at any time through dedicated inter-node channels, arbitrary graph structures can be executed. We demonstrate the effectiveness of the tool on a dynamic active learning task in computational drug design, involving the use of a small molecule generative model and an associated scoring system, and on a reactivity prediction pipeline using quantum-chemistry and semiempirical approaches.

Received 5th September 2024
Accepted 26th October 2024

DOI: 10.1039/d4dd00288a

rsc.li/digitaldiscovery

Introduction

Clearly defined workflows are essential for reproducibility in computational sciences.[1] They make it easier to reason about processes, and allow modularization, fast experimentation, and easy sharing. A workflow can be modelled as a graph, in which each node represents a step of computation, and each edge represents data being passed between steps. We can additionally consider parameters for each node that determine how the computation is performed. As an example, one can view a data processing pipeline as a simple linear workflow, in which data is first read, then processed with a certain set of parameters, and then saved to a new location. Workflows like this are described as directed acyclic graphs (DAGs, Fig. 1), because they are unidirectional and do not involve cyclic data flows. This means that data flows in one direction only, and each node is only executed a single time. DAGs are a popular model for workflows because they can represent many typical processing tasks, are easy to parallelize using topological sorting,[2] and simple enough to reason about. Many tools exist to execute DAGs, popular ones are Apache Airflow,[3] Luigi,[4] and Dagster.[5] Knime[6] is another popular tool, featuring a simplified flow-based architecture optimized for tabular data. Another recent example designed in particular for linear computational chemistry workflows is our tool Icolos.[7] Other recently developed tools focused on dynamic workflow creation are Jobflow,[8] allowing cases in which the number of computations is not known at workflow-compile time, as well as PerQueue,[9] allowing the use of nodes that can run in a cyclic manner. The workflow structure used by these tools is still fundamentally a DAG, although they are significantly more flexible than the

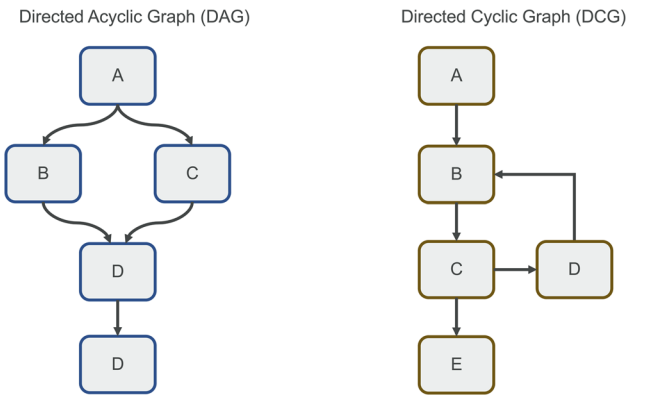

Fig. 1 Directed Acyclic Graphs (DAGs, left) and Directed Cyclic Graphs (DCGs, right). The latter workflow representation allows conditional and iterative execution, common in computational chemistry workflows.

[a]*Molecular AI, Discovery Sciences, R&D, AstraZeneca, 431 50 Gothenburg, Sweden. E-mail: thomas.lohr@astrazeneca.com*

[b]*Innovation Centre in Digital Molecular Technologies, Department of Chemistry, University of Cambridge, Lensfield Rd, Cambridge CB2 1EW, UK*

[c]*Compound Synthesis & Management, The Discovery Centre, Cambridge Biomedical Campus, 1 Francis Crick Avenue, AstraZeneca, CB2 0AA Cambridge, UK*

[d]*University of St Andrews, KY16 9AJ St Andrews, UK*

[e]*Department of Computer Science and Engineering, Chalmers University of Technology, Gothenburg, Sweden*

aforementioned workflow engines when it comes to conditional and cyclic computation.

However, many workflows do not conform to this DAG paradigm, but instead must be modelled as directed (cyclic) graphs (DCGs, Fig. 1). This is the case whenever data is passed through the same node repeatedly (without knowing the number of cycles in advance) or passed to different nodes depending on the nature of the data. Because of this, the convenient topological sorting method can no longer be used, so the graph must be modelled differently. One such approach is termed flow-based programming.[10,11] Here, each node in the graph is represented as a separate system process, with data being moved through uni-directional channels. Each node waits for data to be received and can perform computation as soon as all required data has arrived. Thus, every node is essentially independent from and agnostic to the surrounding graph structure. This model of computation has multiple advantages: first, due to each node operating in isolation, unexpected interactions and bugs resulting from different graph structures can be minimized. Second, parallelism is intrinsic to the graph, as each node operates as an independent process and can perform computation as soon as data is available. Third, the use of specific channels as edges makes it easier to reason about data inputs and outputs and provides modularity of components. Possible disadvantages are the potential overhead of many system processes running concurrently, the potentially unclear status of the graph execution (as halting of the computation cannot be readily predicted), and the sometimes-high complexity of the created graphs due to additional data manipulation. Interestingly, this programming model shows strong similarities to digital hardware design, specifically to the concurrent paradigms of hardware description languages such as Verilog and VHDL.

Here, we developed Maize, a workflow manager based on the principles of flow-based programming. The flow-based and non-linear nature sets it apart from our predecessor workflow engine Icolos.[7] Maize is written in and interfaces through Python, exposing a simple API to allow users to easily define workflows and add custom nodes. Data handling is accomplished with channels enforcing type safety, thus making the input and output requirements of individual nodes clearer and minimizing the potential for errors during execution. In addition, Maize can handle the sending of both small chunks of data in memory, as well as large files on disk while avoiding race conditions. System and workflow configuration are separated, allowing workflows to be transferable between systems. A feature unique to Maize is that multiple nodes can be grouped together into subgraphs, allowing easier reasoning and node reuse, as well as the construction of highly hierarchical workflows that allow multiple levels of granularity in the workflow specification. An important aspect of Maize, compared to a tool such as Knime,[6] is the use of Python throughout, including the ability to fully control the workflow execution and resource allocation. This makes it easier to quickly integrate custom software for computational scientists and allows seamless large-scale parallelism. To make the integration into production pipelines as straightforward as possible, workflows can also be specified in JSON or other serialization formats for automated deployments. A final focus has been the tight integration with high-performance computing (HPC) environments, *e.g.*, batch submission systems.

We will first discuss the underlying principles of Maize in more detail, discuss some of the useful emergent properties with regards to processing of large amounts of data and parallelism, and finally demonstrate its use on reinforcement learning and dynamic active learning tasks for early-stage small molecule drug discovery and a reactivity prediction task using quantum-chemical and semiempirical methods.

Design

Workflow definition

Maize is written in Python using an object-oriented approach. The computational graph is internally represented in a hierarchical manner as a tree (Fig. 2A), with the root as the full workflow graph, tree-nodes as (optional) subgraphs, and leaf-nodes as individual computation steps. Each leaf-node (henceforth termed 'node' for brevity) can declare one or more input or output ports representing data receivers and senders respectively, as well as parameters that are static for the duration of graph execution (Fig. 2B). The workflow is constructed by first initializing a *Workflow* object, followed by adding individual nodes or predefined subgraphs (Fig. 2C) to the workflow, and finally connecting specific inputs and outputs (Fig. 2D). This last step creates a *Channel* object that can pass both files and serialized in-memory data between nodes. The 'root' workflow object thus contains a list of child components, made up of nodes that perform individual computations, and possibly also subgraphs, themselves made up of nodes or subgraphs and so on, with arbitrarily deep nesting possible. Finally, the workflow can be transformed into an executable script, with all node parameters exposed on the command line. Alternatively, the workflow can also be specified using a suitable serialization or configuration system such as JSON or YAML.

Execution

Nodes are declared by inheriting from a base node class, declaring ports and parameters, and defining a *run()* method (Fig. 3A). When running the workflow (Fig. 3B), each node's *run()* method is executed in a separate process (using Python's *multiprocessing* library), potentially with a different Python interpreter, thus allowing the use of otherwise conflicting environments in a single workflow. Each node will perform any computations it can based on the data available to it through inputs and/or parameters and can send data through its outputs at any time. The *run()* method can either run a single time, causing the node to complete upon returning from the method, or run in a looped mode, *i.e.*, re-running the method upon returning. This latter mechanism allows the creation of cyclic workflows. Conditional execution is possible by sending data to one of multiple outputs, as a result only nodes that receive data will perform computation.

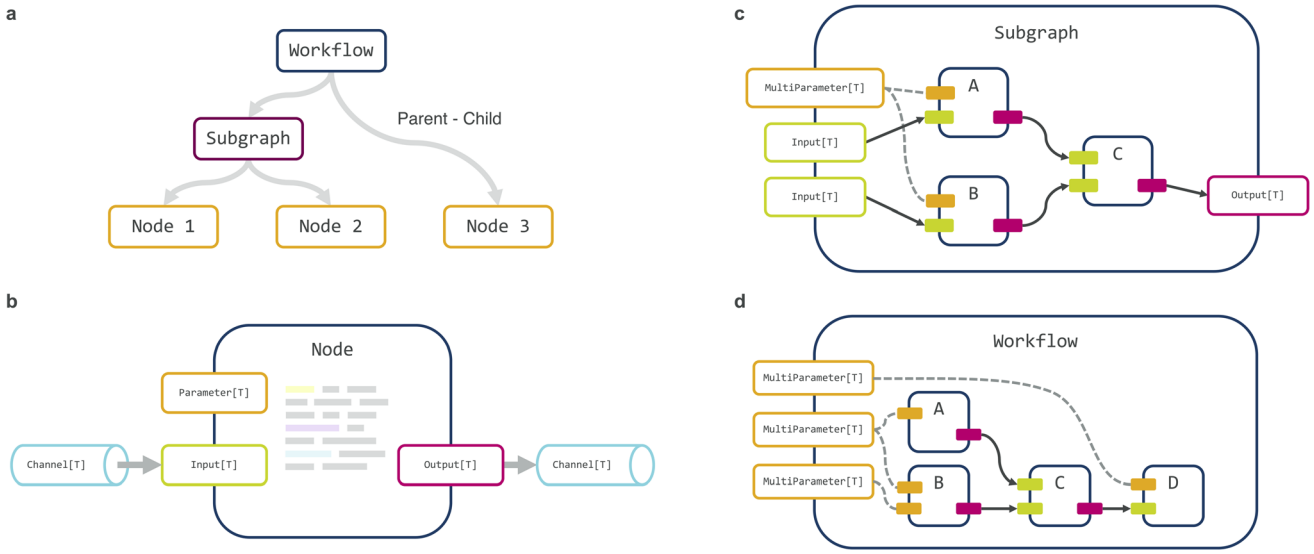

Fig. 2 (a) Internal workflow representation as a tree (connections, *i.e.* channel objects are stored separately), (b) node, (c) subgraph, and (d) workflow architecture. Nodes expose parameters (static values set prior to execution) and input and/or output ports (allowing data to be passed dynamically). These ports are connected to channels, allowing different nodes to be connected. Subgraphs can group multiple nodes together and themselves act like individual nodes with their own inputs and outputs. Workflows include many nodes and optionally subgraphs and can group parameters together and expose them externally, thus abstracting the underlying structure.

During execution, all nodes communicate their status, log messages, and possible errors to the main parent process through separate message queues. The workflow is stopped if one of the nodes raises an unrecoverable exception, all nodes are completed, or a shutdown signal is set by one of the nodes or an external process. Maize uses several heuristics to determine when to shut down a node, as some nodes may be running in a loop without necessarily performing useful computation. When a node has finished computation and exits, it will close its ports and by extension channels. This closing is communicated to a connected node, which can use its own set of rules to determine if it should also shutdown. Thus, node completion can cascade through the workflow graph.

Patterns

In the flow-based programming paradigm, some useful patterns can emerge (Fig. 4):

- Batch processing: if a very large number of datapoints needs to be processed in a sequential workflow, it can be especially efficient to process it in batches. In Maize, this process is parallel by default, as one batch can be processed on the second node while the next batch is processed on the

a

```python
from maize.core.node import Node
from maize.core.interface import Input, Output
from maize.utilities.chem import IsomerCollection

class Smiles2Molecules(Node):
    """Converts SMILES codes into a set of molecules."""

    inp: Input[list[str]] = Input()
    """SMILES input"""

    out: Output[list[IsomerCollection]] = Output()
    """Molecule output"""

    n_variants: Parameter[int] = Parameter(default=1)
    """Maximum number of stereoisomers to generate"""

    def run(self) -> None:
        smiles = self.inp.receive()
        mols = [IsomerCollection.from_smiles(
                    smi, max_isomers=self.n_variants.value)
                for smi in smiles]
        self.out.send(mols)
```

b

```python
from pathlib import Path
from maize.core.workflow import Workflow
from maize.steps.mai.docking import Vina
from maize.steps.io import LogResult

flow = Workflow(name="docking")
embed = flow.add(Smiles2Molecules)
dock = flow.add(Vina)
result = flow.add(LogResult)

flow.connect_all((embed.out, dock.inp), (dock.out, result.inp))

embed.inp.set(["Nc1nc(F)nc(c12)n(CCCC)c(n2)Cc3cc(OC)ccc3OC"])
embed.n_variants.set(4)
dock.receptor.set(Path("./receptor.pdbqt"))
dock.search_center.set((3.3, 11.5, 24.8))
flow.execute()
```

Fig. 3 Maize workflow code. Definition of a custom node embedding a small molecule from a SMILES code (a) and a workflow definition using this node in a linear workflow for docking (b).

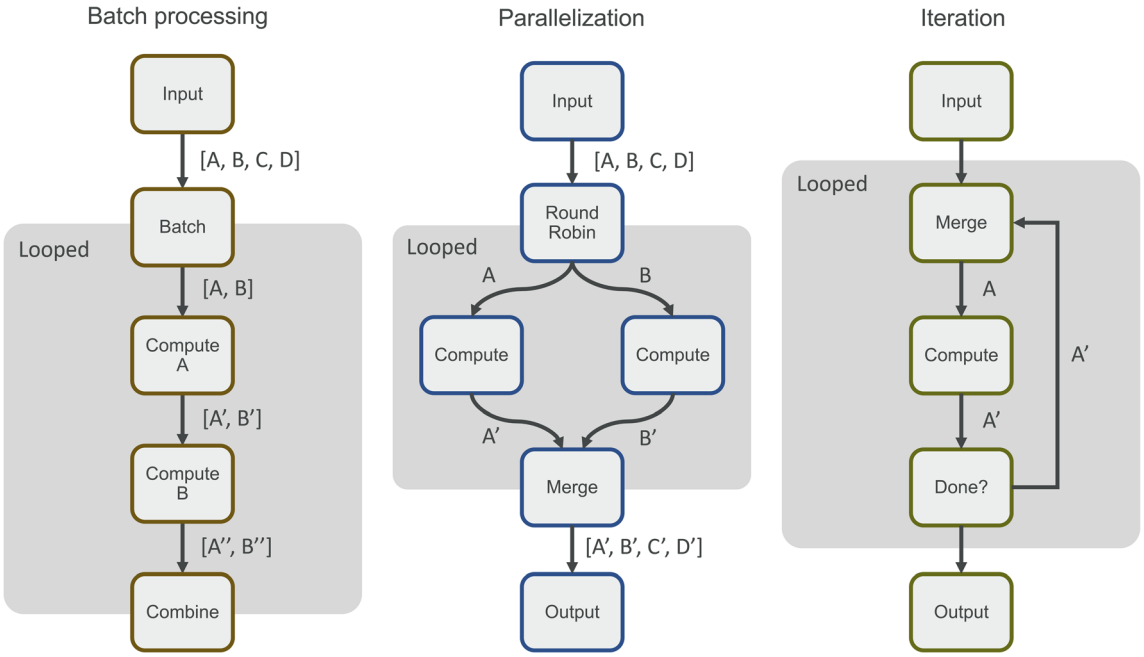

Fig. 4 Useful patterns in flow-based programming. Shaded areas indicate domains of the workflow that are run in a loop, and example data represents the first iteration. Batch processing (left) allows breaking up a large amount of data into chunks, and processing them in parallel, despite the sequential nature of the workflow. Parallelization (middle) allows splitting the data over multiple identical compute nodes. Iteration (right) allows the common pattern of checking a computation for completion and potential re-calculation.

previous node. An example of this is the process of docking small molecules to a target protein in early-stage drug discovery. The small molecule first needs to be prepared, a process that is typically performed on the CPU, and is then docked, an operation that can often be accomplished on the GPU. Thus, a batch of molecules can be prepared on the CPU, while the previous batch is docking on the GPU.

- Parallelization/load-balancing: another commonly seen pattern is parallelization. In Maize, this can be accomplished by creating multiple identical workflow branches and distributing the incoming datapoints over all branches. This workflow pattern can be automatically generated and implemented as a subgraph, allowing any kind of computation to be parallelized naturally without having to worry about locks or race conditions.
- Iteration: many workflows in computational chemistry require performing costly computations until some final condition is fulfilled. This is possible in Maize by creating a node that checks if the computation has completed, sending it either to some final node or back to the computation node for another iteration.

Additional features

Maize exposes several convenience functions to make the definition and running of complex workflows easier and more flexible. These include the ability to submit jobs to a queuing system instead of executing locally, re-executing failed nodes multiple times, loading modules (using the LMOD system) from the Python interpreter, automatically connecting nodes based on their port types, renaming and combination of multiple parameters into one, and shortcuts to create for instance the parallelization pattern mentioned above. Because each node runs in its own separate process (using Python's multiprocessing module), each node can run a different Python environment, as long as it contains a Maize installation. This is because both the interpreter path and the sys.path global variable (indicating to Python where it can find installed modules) can be changed directly before starting the new process. This allows the use of potentially conflicting packages within the same workflow.

Implemented software

We have implemented interfaces to various software packages common in computational chemistry as Maize nodes. So far, these include quantum chemistry software Gaussian,[12] semi-empirical packages xTB[13] and CREST,[14,15] small molecule docking tools such as, AutoDock-GPU,[16] AutoDock Vina,[17] GNINA,[18] and GLIDE,[19] GROMACS[20,21] for molecular dynamics (MD) trajectory analysis, Gypsum-DL[22] for small molecule embedding, and our in-house developed tools REINVENT[23] for AI-based small molecule *de novo* drug design and QpTuna, a tool that automatically generates machine learning models for compound property prediction,[24] as well as various input/output functionality. The domain-agnostic part of Maize also features nodes to enable easier data movement, such as copying, merging, and splitting data. The scope of Maize interfaces is currently expanding rapidly to encompass various tools related to MD simulations including free energy perturbation methods, quantum chemical software and other tools.

Applications

De novo design

Motivation. In this first example we apply Maize on a complex drug discovery workflow, small molecule generation with reinforcement learning.[25,26] The hit-to-lead drug design process typically begins with small molecule hits for a particular target protein. The atomistic structure of these protein–ligand complexes is often available and details the exact position and orientation of the ligand in the protein binding pocket. These initial hit compounds usually exhibit suboptimal properties – they are often not strong and specific binders, and they may have problematic pharmacokinetic properties. It is therefore necessary to find small molecule binders with improved properties using computational approaches, while making use of the information gained from our initial hits. Potential candidates can either be picked from existing compound libraries or created *de novo* using small molecule generative models such as REINVENT.[27–29] The latter method allows guided generation using reinforcement learning,[30] *i.e.*, we can feed back a score for each generated molecule indicating if it should be considered favorable or not. As a result, over many iterations, REINVENT will learn to create more suitable molecules. The scoring function used can take many different forms, but here we will be focusing on the docking score, in which a small molecule is fit into a binding pocket by various geometric transformations and the binding energy evaluated using a physics-based approach.[31]

Implementation. We implemented the workflow described above in Maize, using nodes for REINVENT,[27] AutoDock GPU,[16] Gypsum-DL,[22] and various data-handling (Fig. 5). The parameter system in Maize allows different configurations of the involved software, as well as changes in how the data is piped through the system. In practice the workflow exhibits some additional complexity: the generated small molecules first need to be embedded, *i.e.*, the SMILES[32] codes need to be converted to an actual 3D representation, which also involves selecting an adequate protonation state and stereo-isomer for the corresponding compound (using Gypsum-DL[22]). To demonstrate Maize's control flow abilities, we added an additional docking node with higher precision that is triggered whenever the root-mean-square deviation of the docked small molecule to the original reference compound is above a certain threshold.

A flow-based implementation of such a workflow has multiple advantages: first, nodes can be treated completely independently, and are isolated from one another, reducing possible side-effects. Second, the docking node can be re-used in two locations, with the only difference being a slightly different set of parameters. Third, because every node runs in its own process, environments can be kept separate, and code can run in parallel.

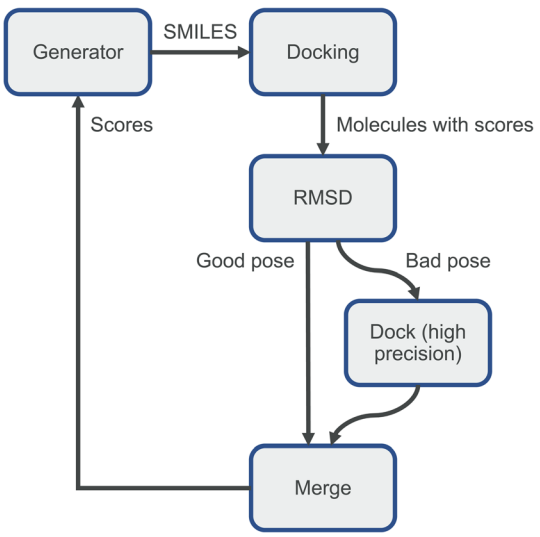

Fig. 5 Small-molecule generation reinforcement learning workflow. Molecules are generated and evaluated by docking them to a target protein structure. Molecules with a large deviation from the reference pose are docked again with higher precision and more conformational sampling. The resulting scores are fed back to the generative model and the process repeated.

Active learning

Motivation. As the number of iterations required to find more favorable small molecules can be quite high, and some scoring methods are often computationally expensive, we would like to replace some of these calculations with a simple machine learning model that can learn an approximation of a physics-based score. This way, instead of always calculating a score using the expensive scoring function, we can in some cases fall back on our fast-to-evaluate approximate model.

This is the main idea behind dynamic active learning:[25,33–36] we first generate a set of small molecules to score against our target protein. In the first iteration, these molecules are evaluated using our physics-based oracle function such as docking, and the scores fed back to our generator, as well as used to train a simple surrogate model emulating our oracle function for future iterations. In subsequent iterations we start by predicting a score for each molecule using this surrogate model. Next, we pick a subset of these compounds using an acquisition function to send to our oracle and use the calculated scores to re-train our surrogate model. Finally, we send all scores back to the small molecule generator and repeat the process. This process has large potential savings in computational time, as the accurate but expensive physics-based calculations are reduced. Additionally, the resulting surrogate model can feature high accuracy despite being a simple model such as a random forest due to the very narrow domain. Here, model training will be limited to a single target protein and is thus non-transferable to other targets. Commonly used acquisition functions use various strategies: we could pick a random subset of molecules, pick the ones predicted to have the highest scores (greedy sampling), use a combination of both (epsilon-greedy), or pick ones with a high uncertainty in their score prediction (*e.g.*, using the upper-confidence bound).

Implementation. Building on the reinforcement learning workflow described above, we implemented an active learning system. We used the same nodes as described above, with the addition of Qptuna[37,38] to provide the surrogate model (Fig. 6).

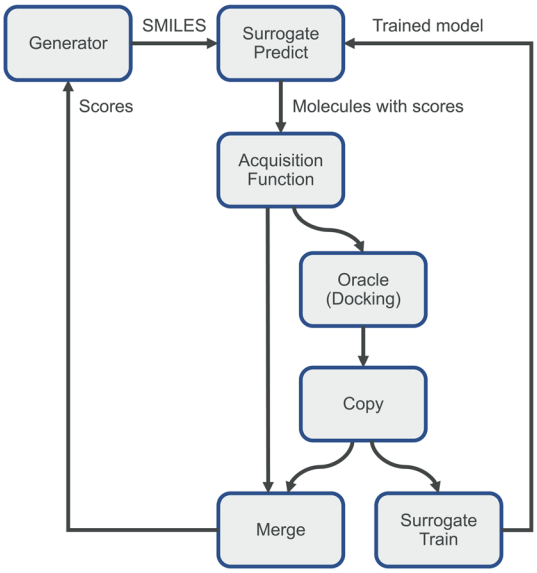

Fig. 6 Simplified active learning workflow. The generator (in this case REINVENT) proposes several molecules, which are fed to a surrogate machine learning model predicting how well these molecules may bind the target protein. Based on these scores, the acquisition function sends a small fraction of the molecules to the computationally expensive oracle (in this case docking). The computed scores are merged with the predicted ones and sent back to the generator to update it. A copy of the scored molecules is sent to the surrogate model for retraining.

The surrogate model is split into separate nodes for training and prediction to simplify the graph dependencies. Finally, the first n iterations will be pooling runs to build up the first training dataset for the surrogate model, *i.e.* during this initial phase all compounds are scored by the oracle only.

As a result of this design, parallelization emerges naturally from the graph definition, for instance, the surrogate model can be re-trained while the next batch of molecules is generated, despite this independence not being explicitly accounted for. The nodes for the tools mentioned above are run in separate Python environments, thus avoiding conflicting dependencies.

To evaluate the efficiency gains from the above-mentioned parallelism, we ran the active learning workflow in a sequential and fully parallel manner and compared execution times. We limited the run to 10 iterations and used a batch size of 512 generated compounds at each iteration, with 128 acquired molecules to be evaluated by the oracle. The parallel workflow was 13% faster than the naïve sequential workflow due to the more efficient resource utilization. For a more detailed demonstration of the capabilities of dynamic active learning, see ref. 36.

Automated first-principles calculations

Motivation. First-principles calculations, such as Quantum-Mechanics and Density Functional Theory, can offer great insight into chemical systems. Molecular properties obtained with such calculations can predict reactivity of chemical compounds and be used as advanced features in data-driven models to increase performances.[39–41] A promising application is the integration of the above-mentioned features in reaction prediction and optimization routines, especially if experimental data is scarce. However, these types of methods often require significant computational resources to be performed as well as specific expertise to be initialized and analyzed correctly. In this context, automation of first-principles calculations could be a remedy to this limitation by providing a faster, more reliable, and more systematic way to perform such calculations. Indeed, to achieve accurate results, a correct description of the system is required; choice of functional, basis set, molecular flexibility and solvent environment are some of the aspects to take in account when defining the system of interest.[42] At the same time, it is crucial to find the right balance between the level of accuracy to be achieved and the computational resources available. In practice, this results in the selection of different computational methods for different tasks, often involving the utilization of multiple software. In this sense, a workflow manager able to orchestrate the requirements of several computational chemistry software is crucial to achieve automation of first-principles calculations and ultimately the integration of such techniques in data-driven methods.

Implementation. Information about the reaction and the chemical structure of its components is received in tabular format and loaded into a control node (Fig. 7). Here, depending on the type of reaction, 3D-geometries for relevant chemical structures in the reaction are generated either through the rdkit package[43] or with custom in-house functions. These input geometries are sent to the first sub-workflow. Initially a molecular mechanics pre-optimisation step removes potential artifacts from the geometries, these are then used as inputs for the conformer generation step performed with semiempirical based metadynamics calculations.

The generated conformers are later inspected by the control node for any inconsistencies or artifacts and later submitted to the second sub-workflow. Here all the conformers of each component undergo geometry optimization at semiempirical

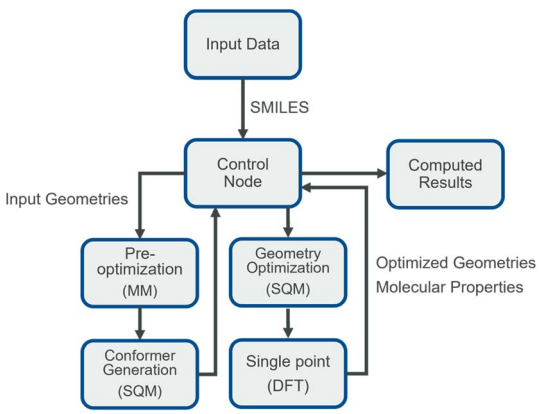

Fig. 7 Reaction prediction workflow based on first-principles calculations. Control node initially generates molecular structures for reactants, products, intermediates, and transition states. Pre-defined structural templates are used to generate geometries for intermediates and templates.

level and single point calculation at DFT level. Once completed, the calculations results are redirected back to the control node which in turn handles potential errors in calculations. Depending on the types of error, individual jobs can be either removed from the workflow or re-submitted. Successful calculations are sent to a return node, which simply reports results in a tabular format.

Discussion

We have presented Maize, a workflow manager capable of executing cyclic and conditional workflows as commonly found in computational chemistry and early-stage drug discovery. We detailed the design and demonstrated its use on a complex active learning workflow to identify possible new small molecule drug candidates. We showed how parallelization emerges naturally from the graph structure, enabling efficiency improvements in possibly unexpected ways. We also detailed useful patterns providing potentially significant speedups to certain workflows.

While Maize was written with computational chemistry in mind, its architecture and design were deliberately kept domain-agnostic to enable its use in other fields. To enable ease of contributing custom nodes, subgraphs, and workflows, the core domain-agnostic part of Maize is a separate package, and all domain-specific components and utilities are in a separate contribution namespace package. This mechanism allows straightforward extensions and simplifies code reuse.

However, Maize is not necessarily suitable for all workflows: while communication between nodes is fast, it is not intended for low-latency, high-frequency, or inter-processor message passing – here, a system such as the Message Passing Interface (MPI) would be more suitable. Related to the previous point is that Maize is not intended to be run on multiple compute nodes the way that MPI applications are, instead Maize can submit jobs to existing job queuing systems such as SLURM and wait for jobs to complete. This means that compute-intensive workflows that potentially require multiple compute nodes will run on a single compute node but submit jobs to other compute nodes and collect the results. Additionally, while we have not observed slowdowns, the use of many Python processes – one for each workflow node – in complex workflows could result in undesirable overheads. Python essentially features two parallelization primitives: threads and processes. Threads have low overhead and allow the use of shared memory but are currently limited by Python's global interpreter lock (GIL), which only allows a single thread to make use of the interpreter at a time. Maize by contrast is implemented using Python's processes, which have a higher overhead and no shared memory, but allow distribution over multiple cores. Future versions of Python will likely remove the GIL, thus in principle allowing the use of threads for Maize, reducing the overhead for very large workflows.

To conclude, we envision Maize as a useful and versatile tool to handle the complexity and many diverse workflows in molecular simulation, computational chemistry, and drug design. It is distributed under the permissive Apache 2.0 license and available at **https://github.com/MolecularAI/maize** and **https://github.com/MolecularAI/maize-contrib**. The latter includes several prepared workflows for common computational chemistry tasks.

Data availability

The code for Maize can be found at **https://github.com/MolecularAI/maize** and **https://github.com/MolecularAI/maize-contrib**. The versions of the code employed for this study are maize 0.8.3 and maize-contrib 0.5.5.

Author contributions

Conceptualization – TL, MaK, OE; Software – TL, MA, MD, LC, MiK, JPJ, MaK; writing – original draft – TL, MA; writing – review & editing – TL, MA, MD, LC, MiK, JPJ, MaK, OE.

Conflicts of interest

The authors declare no conflict of interest.

References

1 S. Cohen-Boulakia, K. Belhajjame, O. Collin, J. Chopard, C. Froidevaux, A. Gaignard, K. Hinsen, P. Larmande, Y. L. Bras, F. Lemoine, F. Mareuil, H. Ménager, C. Pradal and C. Blanchet, Scientific Workflows for Computational Reproducibility in the Life Sciences: Status, Challenges and Opportunities, *Future Gener. Comput. Syst.*, 2017, **75**, 284–298, DOI: **10.1016/j.future.2017.01.012**.

2 A. B. Kahn, Topological Sorting of Large Networks, *Commun. ACM*, 1962, **5**(11), 558–562, DOI: **10.1145/368996.369025**.

3 Apache Airflow, 2023, **https://github.com/apache/airflow**, accessed 2023-10-17.

4 Spotify/Luigi, 2023, **https://github.com/spotify/luigi**, accessed 2023-07-31.

5 Dagster-Io/Dagster, 2023, **https://github.com/dagster-io/dagster**, accessed 2023-07-31.

6 M. R. Berthold, N. Cebron, F. Dill, T. R. Gabriel, T. Kötter, T. Meinl, P. Ohl, C. Sieb, K. Thiel, B. Wiswedel, KNIME: The Konstanz Information Miner, in *Data Analysis, Machine Learning and Applications*, ed. C. Preisach, H. Burkhardt, L. Schmidt-Thieme, R. Decker, Studies in Classification, Data Analysis, and Knowledge Organization, Springer, Berlin Heidelberg: Berlin, Heidelberg, 2008, pp 319–326, DOI: **10.1007/978-3-540-78246-9_38**.

7 J. H. Moore, M. R. Bauer, J. Guo, A. Patronov, O. Engkvist and C. Margreitter, Icolos: A Workflow Manager for Structure-Based Post-Processing of de Novo Generated Small Molecules, *Bioinformatics*, 2022, **38**(21), 4951–4952, DOI: **10.1093/bioinformatics/btac614**.

8 A. S. Rosen, M. Gallant, J. George, J. Riebesell, H. Sahasrabuddhe, J.-X. Shen, M. Wen, M. L. Evans, G. Petretto, D. Waroquiers, G.-M. Rignanese, K. A. Persson, A. Jain and A. M. Ganose, Jobflow: Computational

Workflows Made Simple, *J. Open Source Softw.*, 2024, **9**(93), 5995, DOI: 10.21105/joss.05995.

9 B. H. Sjølin, W. S. Hansen, A. A. Morin-Martinez, M. H. Petersen, L. H. Rieger, T. Vegge, J. M. García-Lastra and I. E. Castelli, PerQueue: Managing Complex and Dynamic Workflows, *Digital Discovery*, 2024, **3**(9), 1832–1841, DOI: 10.1039/D4DD00134F.

10 J. P. Morrison, Data Stream Linkage Mechanism, *IBM Syst. J.*, 1978, **17**(4), 383–408, DOI: 10.1147/sj.174.0383.

11 J. P. Morrison, Flow-Based Programming, *A New Approach to Application Development*, CreateSpace Independent Publishing Platform, Unionville, Ont, 2nd edn, 2010.

12 M. J. Frisch, G. W. Trucks, H. B. Schlegel, G. E. Scuseria, M. A. Robb, J. R. Cheeseman, G. Scalmani, V. Barone, G. A. Petersson, H. Nakatsuji, X. Li, M. Caricato, A. V. Marenich, J. Bloino, B. G. Janesko, R. Gomperts, B. Mennucci, H. P. Hratchian, J. V. Ortiz, A. F. Izmaylov, J. L. Sonnenberg, F. Williams; Ding, F. Lipparini, F. Egidi, J. Goings, B. Peng, A. Petrone, T. Henderson, D. Ranasinghe, V. G. Zakrzewski, J. Gao, N. Rega, G. Zheng, W. Liang, M. Hada, M. Ehara, K. Toyota, R. Fukuda, J. Hasegawa, M. Ishida, T. Nakajima, Y. Honda, O. Kitao, H. Nakai, T. Vreven, K. Throssell, J. A. Montgomery Jr, J. E. Peralta, F. Ogliaro, M. J. Bearpark, J. J. Heyd, E. N. Brothers, K. N. Kudin, V. N. Staroverov, T. A. Keith, R. Kobayashi, J. Normand, K. Raghavachari, A. P. Rendell, J. C. Burant, S. S. Iyengar, J. Tomasi, M. Cossi, J. M. Millam, M. Klene, C. Adamo, R. Cammi, J. W. Ochterski, R. L. Martin, K. Morokuma, O. Farkas, J. B. Foresman, D. J. Fox, *Gaussian 16 Rev. C.01*, 2016.

13 C. Bannwarth, E. Caldeweyher, S. Ehlert, A. Hansen, P. Pracht, J. Seibert, S. Spicher and S. Grimme, Extended Tight-Binding Quantum Chemistry Methods, *Wiley Interdiscip. Rev.: Comput. Mol. Sci.*, 2021, **11**(2), e1493, DOI: 10.1002/wcms.1493.

14 P. Pracht, F. Bohle and S. Grimme, Automated Exploration of the Low-Energy Chemical Space with Fast Quantum Chemical Methods, *Phys. Chem. Chem. Phys.*, 2020, **22**(14), 7169–7192, DOI: 10.1039/C9CP06869D.

15 S. Grimme, Exploration of Chemical Compound, Conformer, and Reaction Space with Meta-Dynamics Simulations Based on Tight-Binding Quantum Chemical Calculations, *J. Chem. Theory Comput.*, 2019, **15**(5), 2847–2862, DOI: 10.1021/acs.jctc.9b00143.

16 D. Santos-Martins, L. Solis-Vasquez, A. F. Tillack, M. F. Sanner, A. Koch and S. Forli, Accelerating AutoDock4 with GPUs and Gradient-Based Local Search, *J. Chem. Theory Comput.*, 2021, **17**(2), 1060–1073, DOI: 10.1021/acs.jctc.0c01006.

17 O. Trott and A. J. Olson, AutoDock Vina: Improving the Speed and Accuracy of Docking with a New Scoring Function, Efficient Optimization, and Multithreading, *J. Comput. Chem.*, 2010, **31**(2), 455–461, DOI: 10.1002/jcc.21334.

18 A. T. McNutt, P. Francoeur, R. Aggarwal, T. Masuda, R. Meli, M. Ragoza, J. Sunseri and D. R. Koes, GNINA 1.0: Molecular Docking with Deep Learning, *J. Cheminf.*, 2021, **13**(1), 43, DOI: 10.1186/s13321-021-00522-2.

19 R. A. Friesner, J. L. Banks, R. B. Murphy, T. A. Halgren, J. J. Klicic, D. T. Mainz, M. P. Repasky, E. H. Knoll, M. Shelley, J. K. Perry, D. E. Shaw, P. Francis and P. S. Shenkin, Glide: A New Approach for Rapid, Accurate Docking and Scoring. 1. Method and Assessment of Docking Accuracy, *J. Med. Chem.*, 2004, **47**(7), 1739–1749, DOI: 10.1021/jm0306430.

20 M. J. Abraham, T. Murtola, R. Schulz, S. Páll, J. C. Smith, B. Hess and E. Lindahl, GROMACS: High Performance Molecular Simulations through Multi-Level Parallelism from Laptops to Supercomputers, *SoftwareX*, 2015, **1–2**, 19–25, DOI: 10.1016/j.softx.2015.06.001.

21 S. Páll, A. Zhmurov, P. Bauer, M. Abraham, M. Lundborg, A. Gray, B. Hess and E. Lindahl, Heterogeneous Parallelization and Acceleration of Molecular Dynamics Simulations in GROMACS, *J. Chem. Phys.*, 2020, **153**(13), 134110, DOI: 10.1063/5.0018516.

22 P. J. Ropp, J. O. Spiegel, J. L. Walker, H. Green, G. A. Morales, K. A. Milliken, J. J. Ringe and J. D. Durrant, Gypsum-DL: An Open-Source Program for Preparing Small-Molecule Libraries for Structure-Based Virtual Screening, *J. Cheminf.*, 2019, **11**(1), 34, DOI: 10.1186/s13321-019-0358-3.

23 H. H. Loeffler, J. He, A. Tibo, J. P. Janet, A. Voronov, L. H. Mervin and O. Engkvist, Reinvent 4: Modern AI–Driven Generative Molecule Design, *J. Cheminf.*, 2024, **16**(1), 20, DOI: 10.1186/s13321-024-00812-5.

24 L. Mervin, A. Voronov, M. Kabeshov and O. Engkvist, QSARtuna: An Automated QSAR Modeling Platform for Molecular Property Prediction in Drug Design, *J. Chem. Inf. Model.*, 2024, **64**(14), 5365–5374, DOI: 10.1021/acs.jcim.4c00457.

25 D. E. Graff, E. I. Shakhnovich and C. W. Coley, Accelerating High-Throughput Virtual Screening through Molecular Pool-Based Active Learning, *Chem. Sci.*, 2021, **12**(22), 7866–7881, DOI: 10.1039/D0SC06805E.

26 I. Filella-Merce, A. Molina, M. Orzechowski, L. Díaz, Y. M. Zhu, J. V. Mor, L. Malo, A. S. Yekkirala, S. Ray and V. Guallar, Optimizing Drug Design by Merging Generative AI With Active Learning Frameworks, *arXiv*, 2023, preprint, arXiv:2305.06334, DOI: 10.48550/arXiv.2305.06334.

27 T. Blaschke, J. Arús-Pous, H. Chen, C. Margreitter, C. Tyrchan, O. Engkvist, K. Papadopoulos and A. Patronov, REINVENT 2.0: An AI Tool for De Novo Drug Design, *J. Chem. Inf. Model.*, 2020, **60**(12), 5918–5922, DOI: 10.1021/acs.jcim.0c00915.

28 J. He, E. Nittinger, C. Tyrchan, W. Czechtizky, A. Patronov, E. J. Bjerrum and O. Engkvist, Transformer-Based Molecular Optimization beyond Matched Molecular Pairs, *J. Cheminf.*, 2022, **14**(1), 18, DOI: 10.1186/s13321-022-00599-3.

29 J. P. Janet, L. Mervin and O. Engkvist, Artificial Intelligence in Molecular de Novo Design: Integration with Experiment, *Curr. Opin. Struct. Biol.*, 2023, **80**, 102575, DOI: 10.1016/j.sbi.2023.102575.

30 M. Olivecrona, T. Blaschke, O. Engkvist and H. Chen, Molecular De-Novo Design through Deep Reinforcement Learning, *J. Cheminf.*, 2017, **9**(1), 48, DOI: **10.1186/s13321-017-0235-x**.

31 J. Li, A. Fu and L. Zhang, An Overview of Scoring Functions Used for Protein-Ligand Interactions in Molecular Docking, *Interdiscip. Sci.: Comput. Life Sci.*, 2019, **11**(2), 320–328, DOI: **10.1007/s12539-019-00327-w**.

32 D. Weininger, SMILES, a Chemical Language and Information System. 1. Introduction to Methodology and Encoding Rules, *J. Chem. Inf. Comput. Sci.*, 1988, **28**(1), 31–36, DOI: **10.1021/ci00057a005**.

33 J. Sacks, S. B. Schiller and W. J. Welch, Designs for Computer Experiments, *Technometrics*, 1989, **31**(1), 41–47, DOI: **10.1080/00401706.1989.10488474**.

34 D. R. Jones, M. Schonlau and W. J. Welch, Efficient Global Optimization of Expensive Black-Box Functions, *J. Glob. Optim.*, 1998, **13**(4), 455–492, DOI: **10.1023/A:1008306431147**.

35 J. Yu, X. Li and M. Zheng, Current Status of Active Learning for Drug Discovery, *Artif. Intell. Life Sci.*, 2021, **1**, 100023, DOI: **10.1016/j.ailsci.2021.100023**.

36 M. Dodds, J. Guo, T. Löhr, A. Tibo, O. Engkvist and J. Paul Janet, Sample Efficient Reinforcement Learning with Active Learning for Molecular Design, *Chem. Sci.*, 2024, **15**(11), 4146–4160, DOI: **10.1039/D3SC04653B**.

37 T. Akiba, S. Sano, T. Yanase, T. Ohta, M. Koyama, Optuna: A Next-Generation Hyperparameter Optimization Framework, in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '19; Association for Computing Machinery, New York, NY, USA, 2019, pp 2623–2631, DOI: **10.1145/3292500.3330701**.

38 QPTUNA: QSAR Using Optimization for Hyper-Parameter Tuning, 2023, **https://github.com/MolecularAI/Qptuna**, accessed 2023-07-31.

39 M. H. Samha, L. J. Karas, D. B. Vogt, E. C. Odogwu, J. Elward, J. M. Crawford, J. E. Steves and M. S. Sigman, Predicting Success in Cu-Catalyzed C–N Coupling Reactions Using Data Science, *Sci. Adv.*, 2024, **10**(3), eadn3478, DOI: **10.1126/sciadv.adn3478**.

40 S. M. Maley, D.-H. Kwon, N. Rollins, J. C. Stanley, O. L. Sydora, S. M. Bischof and D. H. Ess, Quantum-Mechanical Transition-State Model Combined with Machine Learning Provides Catalyst Design Features for Selective Cr Olefin Oligomerization, *Chem. Sci.*, 2020, **11**(35), 9665–9674, DOI: **10.1039/D0SC03552A**.

41 K. Jorner, T. Brinck, P.-O. Norrby and D. Buttar, Machine Learning Meets Mechanistic Modelling for Accurate Prediction of Experimental Activation Energies, *Chem. Sci.*, 2021, **12**(3), 1163–1175, DOI: **10.1039/D0SC04896H**.

42 M. Bursch, J.-M. Mewes, A. Hansen and S. Grimme, Best-Practice DFT Protocols for Basic Molecular Computational Chemistry, *Angew. Chem.*, 2022, **134**(42), e202205735, DOI: **10.1002/ange.202205735**.

43 RDKit: Open-Source Cheminformatics. **https://www.rdkit.org**.