Title

PluginPlay: Enabling exascale scientific software one module at a time

Permalink

https://escholarship.org/uc/item/2xh2j8nc

Journal

The Journal of Chemical Physics, 158(18)

ISSN

0021-9606

Authors

Richard, Ryan M

Keipert, Kristopher

Waldrop, Jonathan

et al.

Publication Date

2023-05-14

DOI

10.1063/5.0147903

PluginPlay: Enabling Exascale Scientific Software One Module at a Time

Ryan M. Richard,[1,2,a)] Kristopher Keipert,[3] Jonathan Waldrop,[1] Murat Keçeli,[4] David Williams-Young,[5] Raymond Bair,[4] Jeffery Boschen,[1,2] Zachery Crandall,[1,2] Kevin Gasperich,[4] Quazi Ishtiaque Mahmud,[2] Ajay Panyala,[6] Edward Valeev,[7] Hubertus van Dam,[8] Wibe de Jong,[5] and Theresa L. Windus[1,2,b)]

[1)] *Ames National Laboratory, Ames, IA, 50011, USA.*

[2)] *Iowa State University, Ames, IA, 50011, USA.*

[3)] *NVIDIA, San Jose, CA, 95131, USA.*

[4)] *Argonne National Laboratory, Lemont, IL, 60439, USA.*

[5)] *Lawrence Berkeley National Laboratory, Berkeley, CA, 94720, USA.*

[6)] *Pacific Northwest National Laboratory, Richland, WA, 99354, USA.*

[7)] *Virginia Tech, Blacksburg, VA, 24061, USA.*

[8)] *Brookhaven National Laboratory, Upton, NY, 11973, USA.*

(Dated: 10 October 2023)

For many computational chemistry packages, being able to efficiently and effectively scale across an exascale cluster is a heroic feat. Collective experience from the Department of Energy's Exascale Computing Project suggests that achieving exascale performance requires far more planning, design, and optimization than scaling to petascale. In many cases, entire rewrites of software are necessary to address fundamental algorithmic bottlenecks. This in turn requires a tremendous amount of resources and development time, resources that can not reasonably be afforded by every computational science project. It thus becomes imperative that computational science transition to a more sustainable paradigm. Key to such a paradigm is modular software. While the importance of modular software is widely recognized, what is perhaps not so widely appreciated is the effort still required to leverage modular software in a sustainable manner. The present manuscript introduces PluginPlay, https://github.com/NWChemEx-Project/PluginPlay, an inversion-of-control framework designed to facilitate developing, maintaining, and sustaining modular scientific software packages. This manuscript focuses on the design aspects of PluginPlay, and how they specifically influence the performance of the resulting package. Although, PluginPlay serves as the framework for the NWChemEx package, PluginPlay is not tied to NWChemEx, or even computational chemistry. We thus anticipate PluginPlay to prove to be a generally useful tool for a number of computational science packages looking to transition to the exascale.

[a]Electronic mail: rrichard@ameslab.gov
[b]Electronic mail: twindus@iastate.edu

I. INTRODUCTION

Colloquially speaking, Moore's law[1] is the observation that the computing power of available processors doubles approximately every two years. Moore's law, combined with Dennard scaling[2] (the observation that despite the increase in processing power, the power consumption remains about the same for the same die size), has been responsible for a golden age of computational science, where one could run larger and larger simulations simply by waiting for new hardware. Unfortunately, Moore's law and Dennard scaling can not, and will not, continue indefinitely. Particularly within the last decade or so, scientists have come to appreciate that in order to simulate even larger systems, scientists will need to embrace the increasingly heterogenous hardware landscape.

When the Exascale Computing Project (ECP) started, the NWChem[3–6] team viewed the ECP as an opportunity to start anew with NWChemEx[6]. With NWChemEx being a ground-up rewrite of the original NWChem, the team invested a significant amount of time in software design with the intent of ensuring that NWChemEx will be a sustainable software package for years to come. The resulting design champions encapsulation and a separations-of-concerns on a somewhat unprecedented level - especially for exascale, at least for computational chemistry. More specifically, the entirety of NWChemEx is written as self-contained modules. The list of available modules, which module to call, and even how the modules are wired together can all be changed dynamically, at runtime, in a non-invasive manner. The present study argues that reaching the exascale is dramatically easier with sustainable, modular software. Key to NWChemEx's sustainability efforts is PluginPlay, a framework for supporting modular scientific software.

The present study "derives" PluginPlay https://github.com/NWChemEx-Project/PluginPlay. By this we mean that in Section II B we will start by formally enumerating the design considerations necessary for high-performance scientific computing (including exascale). From there, Section III presents a series of logical arguments showing how these design considerations motivated the architecture (defined here as the high-level design aspects pertaining to the overall structure of the PluginPlay library, and how it interacts with other stakeholders), and ultimately design of PluginPlay. We conclude with a high-level overview of the design of NWChemEx's Hartree-Fock (HF)/density functional theory (DFT) code, focusing in particular on how PluginPlay facilitates a modular implementation. The present study is part of the Journal of Chemical Physics's special topic on "High-Performance Computing in Chemical Physics" and a separate submission by Williams-Young and co-workers continues the discussion by focusing on the inner workings, performance, and initial applications of GauXC. While GauXC[7,8] was developed as part of the NWChemEx project to compute the exchange-correlation energy for NWChemEx's DFT code, the modular nature of the architecture allows the GauXC code to be used in other software packages as well. It is our hope that the current study will: clarify just how hard exascale scientific software development actually is, and serve as a template of sorts for future design efforts. We should also note that, while the first application of PluginPlay is NWChemEx, PluginPlay is a general reusable open-source framework for computational science that is not explicitly tied to NWChemEx (or even exascale computing).

II. BACKGROUND

A. Exascale Computing

The Department of Energy (DOE) ECP started in 2016[9,10] with the aim of building the United States' first exascale computers, as well as the exascale-capable software stack to accompany them. While it was widely appreciated[11–13] at the onset of ECP that moving from the petascale to the exascale would be more difficult than the transition from terascale to petascale, exactly how much more difficult was perhaps not as appreciated. At the risk of oversimplifying, the difficulty in transitioning to the exascale comes down to two things: strong versus weak scaling and the heterogeneous hardware of most exascale computers.

In high-performance computing (HPC) one discusses two types of scaling: weak and strong. An application is said to exhibit weak scaling if the time-to-solution remains constant upon scaling **both** the amount of work and the computational resources by the same factor. For example, the work required to compute a coupled cluster with single, double, and perturbative triple excitations (CCSD(T))[14] energy scales as $\mathcal{O}\left(N^7\right)$ (N being problem size). Assuming a particular CCSD(T) implementation exhibits perfect weak scaling, then if we double the problem size (which in turn produces $2^7 = 128$ times more work), then increasing the computational resources also by a factor of 128 would result in the same time-to-solution as the original problem. For a **fixed** amount of work, if we increase the amount of available computing resources by some factor, then assuming the application exhibits strong scaling, the time-to-solution would decrease by the same factor. For example, assuming our CCSD(T) implementation also exhibits strong scaling, then if t was our initial time-to-solution, upon increasing the computing resources by a factor of 128 for the same problem, the new time to solution would become $\frac{t}{128}$.

It is impossible to maintain strong scaling indefinitely; at some point one simply runs out of work to parallelize and the additional computing elements remain idle. In practice, most developers worry about strong scaling until the time-to-solution has been reduced to some reasonable amount, at which point concern shifts to weak scaling. "Reasonable" is highly situational and ultimately comes down to how long the user is willing to wait. For example, many people are willing to wait an hour for a single-point energy of a very large system computed using a high-level of theory, but an hour is still too long for a single-point energy computation if the goal is to run molecular dynamics (which typically seeks sub-second single-point energy evaluations).

So why does this all matter for exascale computing? Generally speaking, algorithms which exhibit weak-scaling are "embarrassingly parallel" and will continue to exhibit weak-scaling indefinitely. They thus can be adapted to the exascale reasonably easily. The problem is, especially for electronic structure theory, most algorithms are not embarrassingly parallel, and we need to contend with strong scaling. When it comes to strong scaling we are at the mercy of Amdahl's law[15]. Amdahl's law says the strong scaling efficiency of your algorithm is limited by the fraction which is not parallelized. In other words, if for a given problem size you want to maintain the same parallel efficiency while using x-times more computing resources, you need to decrease the serial portion of your algorithm by a factor of x. As an example, a code which has an 80% parallel-efficiency on petascale hardware, will need 1000 times less serial code in order to maintain 80% parallel-efficiency on exascale hardware. For practical reasons, developers usually choose to parallelize the code regions which take the longest first; meaning, it can be exponentially harder to parallelize the remaining serial fraction of the code, since it often is made up of many small routines rather than one large routine. On most exascale machines, processing power primarily comes from graphics processing units (GPUs), each of which are terascale computing resources; thus to scale from one GPU to an exascale machine requires a 1,000,000 times reduction in the amount of serial code.

This brings us to the second of the aforementioned problems, the heterogeneous hardware. Existing central processing unit (CPU) implementations are usually ill-suited for use on GPUs and must be rewritten. Making matters worse, at present, each GPU vendor uses its own programming language, with its own abstractions. So a GPU implementation targeting one vendor is usually not even portable to another vendor. While more general solutions, such as OpenMP[16] exist, they are still not as performant across all platforms as vendor specific solutions. Combine this with the more complicated memory hierarchies on exascale computers — standard dynamic random-access memory (DRAM), non-volatile random-access memory (NVRAM), solid-state drives (SSDs), and device memory — and even data movement is more difficult. Then the metaphorical cherry-on-top is that this all gets coupled back to Amdahl's law; so unless one is willing to rewrite every existing bottleneck CPU algorithm, strong scaling across an exascale machine will, in general, not be possible. Making matters still worse, while GPUs are the primary accelerators in play at the moment, a whole host of other accelerators — field-programmable gate arrays (FPGAs), tensor processing units (TPUs), and quantum computers — are waiting in the wings. In short, HPC is hard, and it has only gotten much harder with the arrival of the exascale era. Scientific software is often developed with limited resources, and the arrival of the exascale era promises to further strain those resources. It thus becomes essential for scientific software be developed in a sustainable manner.

B. Sustainable Scientific Software

As the author lists for most scientific software packages attest, it already takes a tremendous number of people to write performant scientific software. The last section painted a pretty grim picture of what it takes to develop exascale software and suggested how the situation will only continue to get worse. Frankly speaking, many scientific software packages are already struggling to balance their current research efforts with fundamental software maintenance (bug fixes, feature requests, performance tuning, and porting to new hardware). Increasing the complexity of the software, and needing to wholesale port a large number of existing CPU-based algorithms to new hardware, is unlikely to be feasible on a per-package basis. As we look to the future of scientific software it seems clear that sustainability will need to play a bigger role.

Unfortunately, achieving sustainability is not accidental and requires careful design and planning. Up until about the early 2000s, many legacy scientific software packages were developed without adhering to any real design principles. In many cases, the lack of design was not intentional, but stemmed from the fact that historically scientific software has been developed by scientists who often did not keep up with software engineering practices[17]. While legacy packages may have had different origin stories, the lack of design, particularly with respect to interoperability, means that when a feature of another package was needed it was often easier to re-implement the feature than to reuse the other package's feature. Within electronic structure theory, the net result is that each package's feature set has tended to converge to a set of standard algorithms, *e.g.*, DFT, second-order Møller-Plesset perturbation theory (MP2), CCSD(T). This is a substantial amount of duplicated work and few, if any, of these redundant implementations have been developed in a modular fashion, in turn perpetuating the cycle.

While many legacy package developers have, often in hindsight, realized the importance of better design, particularly when it comes to modularity, for many legacy packages it is too late. After decades of development, many legacy packages contain millions of lines of code, hundreds of features, a tightly-coupled code base, and mounds of technical debt (the cost, in developer time, needed to refactor a unit of code into the most effective solution). For such packages, changing the design requires repaying more technical debt than is practical, resulting in two choices: start from scratch or continue to rack up technical debt. While we do not purport to have a third option, we presently argue that the changing HPC landscape, specifically the need to create hardware-specific algorithms, can be seen as a golden opportunity for starting from scratch, and it is imperative that we use this opportunity to develop the most sustainable software we can in an effort to avoid the need to start from scratch again.

To be clear, while the contents of this subsection have so far been largely anecdotal, they are consistent with other anecdotal accounts of scientific software development. As a direct result of similar situations and realizations, a number of different research software sustainability groups[18–28], ranging from federal governments to individual research disciplines, have begun to form. Together these organizations have provided a wealth of guidance for developing sustainable research software; however, for brevity, we have distilled this guidance down to the following high-level design considerations:

II.B.1 **Sustainable**. Sustainability is defined as the ability for a software package to persist and remain useful beyond the original set of use cases. This includes, but is not limited to: new hardware, new applications, new developers, and new development paradigms. Sustainability does not imply that the software remains static, in fact our definition all but guarantees that the software must evolve over time. Moving forward, the time to develop performant components is likely to increase, and to avoid duplication, the goal should be to develop sustainable software, not to "just get something working."

II.B.2 **Reproducible**. Scientific software does science. A fundamental tenet of science is that results should be reproducible (running the same software, with the same inputs, should produce the same results). It is important that software be designed in a manner which facilitates reproducibility and ideally replicability (the ability to obtain consistent results with different software and similar inputs).

II.B.3 **Modular**. Modularity comes in two flavors: inter- and intra-package modularity. We are specifically interested in inter-package modularity where the modular software: is created, designed, and distributed *independently* of other software; fully encapsulates its dependencies; has well defined (and stable) application programming interfaces (APIs) and user interfaces (UIs); and is reusable by other software *as is* (no modifications required). By contrast, intra-package modularity is only concerned with reuse throughout the same package, and reuse outside the package may be impractical. Ultimately, most modern software development projects (scientific software included) are too large to tackle in any manner other than piecemeal, which is why modularity is so important.

II.B.4 **Interoperable**. In software engineering two components are interoperable if they "just work" together. This specifically means the components use the same data formats, protocols, and standardized APIs. In particular, if two components are interoperable they do not require adaptors, converters, or the like, in order to work together. Interoperability plays a key role in sustainability by increasing developer productivity and increasing the reuse of modular software. Again we can define intra- and inter-package interoperability. Inter-package interoperability ultimately amounts to the packages in a community coming together and establishing data and communication standards, which is beyond our present scope. Without community consensus, true inter-package interoperability is impossible and conversion tools become necessary. Intra-package interoperability is an easier ideal, and simply means that within a software package modules are interoperable.

II.B.5 **Performant**. Scientific simulations are computationally expensive. Even a small degradation in performance can cause a simulation to be intractable. Software design must not inhibit performance on current (or future) hardware. In practice, performance can not be an afterthought and it must be designed for from the beginning.

II.B.6 **Research-Based**. The end-goal of scientific software is research. Research is an inherently uncertain process. *A priori* it is difficult, if not impossible, to know what theories will pan out, how users will leverage your software, what new use cases will result from ongoing research, or which funding opportunities will be available. The ability to rapidly prototype is important for quickly discerning the viability of an idea, but ultimately robust, sustainable software needs to be the foundation of research efforts to ensure results are correct and reproducible.

II.B.7 **Complex**. Few computational scientific studies are able to publish results simply by running an existing program. Scientific workflows usually consist of some combination of: code development, *ad hoc* couplings between software and/or scripts, data analysis, and visualization. Software designs need to be extensible so as to not hinder current or future research workflows.

II.B.8 **Multidisciplinary**. Most computational science codes require expertise beyond the target scientific domain, *e.g.*, computer science, data science, computational mathematics, statistics, and/or software engineering. In computational chemistry, even the scientific domain expertise tends to be multidisciplinary with needs for expertise in physics, materials science, and biology.

II.B.9 **Decentralized**. Scientific research is conducted world-wide. Most existing scientific software packages do not have the luxury of having all developers reside locally. This complicates synchronization, and often means developers are not necessarily aware of other efforts, even when those efforts may be targeting the same package. Somewhat related, the resulting software stack tends to be decentralized as well. While there are some efforts[29] to unify the process of delivering a full working, scientific software stack, developers must contend with the fact that, at present, their software's dependencies tend to be scattered across the internet.

C. Prior Work

The considerations in Section II B ultimately motivated us to write PluginPlay. Before discussing the architecture of PluginPlay in Section III, we briefly review related efforts in computational chemistry, as well as relevant C++ frameworks.

One of the first attempts at modularizing the field of computational science was associated with the Common Component Architecture (CCA)[30] project. Each software component adhering to CCA conventions needed to adhere to the relevant standardized APIs. CCA attempted to span multiple programming languages and allow dynamic loading of modules to enable end-to-end, complex scientific HPC simulations. While the machinery needed to actually assemble software from CCA compliant components was fickle and cumbersome to use due to its prototype nature, CCA is notable in that it is one of the few prior efforts to really consider most of the considerations in Section II B. Although the CCA has been abandoned, some of its computational chemistry contributions live on in terms of combining multiple theory levels[31] and integral APIs[32], as well as the software engineering ideas associated with HPC.

Within electronic structure theory, there are several packages designed with inter-package modularity in mind. One of the first was Massively Parallel Quantum Chemistry (MPQC)[33,34]. The original design[33] defined a series of abstract classes to serve as internal APIs; derived classes then implemented the functionality. Originally, these classes only provided intra-package modularity; however, this restriction was later lifted[34] by allowing the implementations to treat MPQC as a dependent library. Like MPQC, Psi4[35–37] provides a number of internal APIs which can serve as customization points. Using Psi4's plugin mechanism developers can non-invasively extend Psi4 by writing plugins; however, these internal APIs are limited to high-level properties[36], such as total energies, and (similar to MPQC) require the developer to link against Psi4. The Psi4 team works closely with the Molecular Sciences Software Institute (MolSSI), and Version 1.4 of Psi4[37] relaxes these restrictions by using the MolSSI-developed Quantum Chemistry Archive (QCArchive) (*vide infra*). Perhaps Python-based Simulations of Chemistry Framework (PySCF)[38] comes the closest to the level of inter-package modularity targeted by PluginPlay. In particular, the PySCF objects are composable functors (callable objects associating a state with a routine). Using composition, users can manually wire together a new call graph containing a mix of PySCF and user-defined functors. Owing to how composition is implemented, and the fairly coarse-grained nature of the functors (*e.g.*, computing integrals, building integrals, and transforming integrals), the new call graph can often be represented as a one line command. Finally, since PySCF and Psi4 have Python interfaces, it is possible for the user, at runtime, to extend and override much of the functionality provided by PySCF and Psi4; however, without formal standards in place (such as the ones PySCF provides[38] for overriding the Hamiltonian) the resulting code is likely to be fairly fragile. This is because the resulting user code typically modifies the underlying software in ways in which the developers did not intend and/or are not actively supporting. So while this sort of non-invasive extension is great for rapid prototyping, it tends to be ill-suited for sustainable development.

More recently, inter-package modularity efforts have seen an increased interest in developing computational chemistry workflow tools. Included in this category are: Atomic Simulation Environment (ASE)[39], MolSSI Driver Interface (MDI)[40], Python-based Amsterdam Density Functional (ADF) (PyADF)[41], Python Materials Genomics (pymatgen)[42], QCArchive[43], and Quantum Chemistry Automation and Structure Manipulation (QChASM)[44] projects. Historically, these projects have also been referred to as "drivers" because they drive other software packages. Generally speaking, all of these tools interact with existing computational chemistry packages at a very high-level (typically the same level as an end-user), which in turn tends to limit them to primarily modularizing energies (and energy derivatives). That said, many of these workflow tools provide additional features (such as structure manipulations), but because these features tend to not be modularized, they can not easily be used by the underlying software. In turn, many of these additional workflow features need to be re-implemented by the underlying software, potentially leading to a large amount of redundancy. Many of these tools also address most of the considerations listed in Section II B, but with a restriction to high-level modularity. Perhaps the most notable omission among these tools is the multidisciplinary consideration (II.B.8); these tools tend to be overwhelmingly focused on interfacing to computational chemistry software.

Most computational chemistry packages have some level of intra-package modularity in the form of functions and/or libraries (or the language equivalents). The packages we have included here have been singled out because they also include some level of inter-package modularity. The resulting modules tend to be fairly heavy since they have the package being extended as a dependency. As we argue in Section III, the key to decoupling the module from the package to extend, is to build the package on top of an inversion-of-control (IOC)[45] framework. Because of the performance consideration, we need to be able to seamlessly use C/C++ modules with this framework and we limit ourselves to IOC frameworks written in C++. While a number of such frameworks exist[46–49], we were unable to locate any which are currently supported and under active development. Finally we note that PluginPlay grew out of the now abandoned Pulsar Computational Chemistry Framework project[50]. Many of the original design elements of Pulsar ended up in a beta version of Simulation Development Environment (SimDE)[51] (*N.B.*, we have since changed the abbreviation from SDE to SimDE to avoid confusion with "software development environment"). Since Reference 51, SimDE has undergone a significant amount of refactoring in order to better separate the parallel runtime, the IOC framework, the chemistry-specific classes, and the development environment into reusable components. As a result of this refactoring we have produced PluginPlay. So while some high-level design details of PluginPlay were included in Reference 51, those details: are not exhaustive, are intermixed with design details from other components, and are only used to motivate the introduction of the overall SimDE project. The exact relationship between SimDE, PluginPlay, and the other components of the NWChemEx software stack are summarized in Section IV.

III. PLUGINPLAY DESIGN

A. Motivation

Section II B ended with a series of considerations that scientific software should address. From the standpoint of the high-level architecture of a scientific software package, considerations research-based, complex, and multidisciplinary (II.B.6, II.B.7, and II.B.8 respectively) are arguably the most important. Together these considerations tell us the resulting software needs to be, from the beginning, extremely flexible in order to adapt to: unforeseeable research directions, new workflows, and the differing conventions/paradigms across scientific disciplines. While modularity will inevitably play a key role in the solution, the level

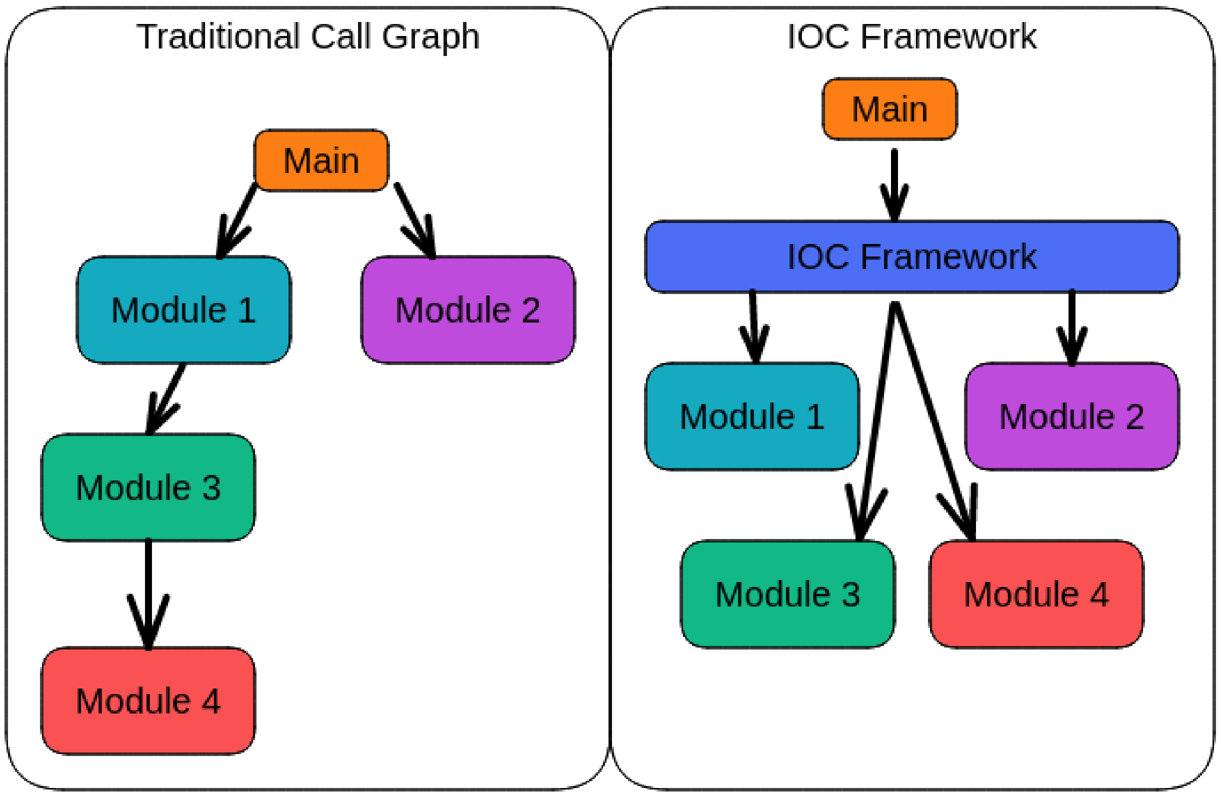


FIG. 1. Difference between a normal package and a package powered by an IOC framework. In both panels modules are called in a top-to-bottom, left-to-right ordering.

of flexibility required extends beyond modularity and raises a new consideration of avoiding as many *a priori* couplings as possible. In particular, designs inherently limit the applicability of the resulting software package if they couple to specific sets of: hardware, features, data types, or even components. Of these assumptions, avoiding coupling to a known set of components is arguably the most difficult to avoid. As alluded to in Section II, software engineers have already come up with a solution: IOC frameworks.

The difference between normal control graph execution and an IOC is shown in Figure 1. Generally speaking, most scientific software has either a single entry point (*e.g.*, `main` function for C/C++ executables and Python scripts) or a series of well-defined entry points (*e.g.*, UI of Python packages). Regardless of the number of entry points, each entry point is capable of calling a fixed number of subroutines, each of which are then also limited to calling a fixed number of subroutines, etc. The point is, in a traditional software package, once the user has called an entry point the software maintains control, and is responsible for delegating control flow from that point forward. This means the only way to add a new feature, use case, etc. is to modify the software, at each point where it needs to know about the feature, use case, etc. Conversely, when software is built on an IOC framework, the software dispatches control to the framework, and the framework decides the first subroutine to call based on the end-user's input. If that subroutine needs to relinquish control, including to call another subroutine, the subroutine first returns control to the IOC framework, and the IOC framework chooses whom to pass control to, *i.e.*, subroutines do not choose which subroutines they call, the IOC framework does. In turn, newly added features are immediately available throughout the call graph, without needing to modify the software. While an IOC framework avoids coupling components to each other until runtime, it does so by instead coupling every component to the IOC framework. In practice, this is typically not as bad as it sounds, since the components usually treat the IOC framework as a dependency, and thus the coupling to the IOC framework can be treated as an implementation detail of the component. The advantage of using an IOC framework is that it allows us to address considerations: research-based, complex, and multidisciplinary (II.B.6, II.B.7, and II.B.8).

Because of the modular consideration (II.B.3) we will consider our software as being comprised of components, and because of the performant consideration (II.B.5) it will be necessary for some of the components to tightly couple to a specific problem and/or specific hardware. We term such pieces "modules". In many computational chemistry applications, modules tend to focus on filling in tensors and/or consuming tensors (*N.B.*, we include in this statement scenarios where formally the theory and equations call for tensors, but in practice the full tensors are never actually assembled). Regardless of their contents, from the perspective of the IOC framework, modules form the nodes of the call graph. The IOC framework is thus ultimately charged with discovering new modules and wiring them together. Being able to manipulate modules from within the IOC framework is essential to sustainable (II.B.1) since it enables developers to non-invasively extend performance-critical pieces of the software to new hardware and use cases, all while having a minimal impact on the rest of the software.

Driven by performance (consideration II.B.5), and realizing the IOC framework will necessarily touch performance-critical modules, we have limited ourselves to C++ IOC frameworks. This is because the HPC community is overwhelmingly moving from Fortran to C/C++ (as evidence, note that the majority of ECP projects[10] assume C or C++) and because a C++-based framework is capable of leveraging C (and Fortran) and C++ modules. Thanks to projects such as Cppyy[52] and PyBind11[53], C++ software can easily be exposed to Python to improve the user experience (UX) without sacrificing performance[35–38], which facilitates rapid prototyping and the research-based consideration (II.B.6). Following from Section II, we were unable to locate any existing C++ IOC frameworks which are actively supported. Furthermore, to our knowledge, none of the existing computational chemistry efforts are designed to support IOC throughout the entire call graph, down to the lowest levels. Thus, in order to have a performant C++ IOC framework for scientific applications, we ultimately chose to develop PluginPlay.

As presently motivated, to address the considerations raised in Section II B, PluginPlay was designed subject to the following considerations:

III.A.1 **Sustainable**. Sustainability is even more important since every subroutine will couple to PluginPlay. Therefore sustainability of the downstream software assumes sustainability of PluginPlay itself.

III.A.2 **Performant**. With many calls routing through PluginPlay, PluginPlay must not inhibit the performance of those calls.

III.A.3 **Multidisciplinary**. PluginPlay needs to be capable of handling modules from a variety of disciplines. These disciplines use different terminology, types, and standards.

III.A.4 **Module discovery**. PluginPlay must be capable of dynamically discovering modules. The set of modules, the module authors, the deployment location of the module, and the relevancy of each module may change over time. It is thus unsustainable for PluginPlay to assume a static list of modules.

III.A.5 **Module wiring**. Since the set of modules is dynamic, PluginPlay must also wire the modules together dynamically. A dynamic call graph also ensures the resulting software is extensible and can be adapted to new use cases.

III.A.6 **Reproducible**. PluginPlay is designed as a framework for scientific software, and, therefore, results with PluginPlay must be reproducible. Considerations III.A.4 and III.A.5 result in unique challenges to reproducibility since what modules can be called, and which modules are wired together, can change from run to run.

The main goal of PluginPlay is to performantly build and manage a dynamic call graph comprised of performant modules.

B. Architecture

PluginPlay's architecture was designed in response to the considerations raised at the end of Section III A. For sustainability and performance reasons, PluginPlay's design adheres to usual C++ conventions, such as those underlying the C++ standard library. This facilitates using other C++ libraries (many of which also assume standards compliant code). In particular, PluginPlay is largely object-oriented, with stateful code being represented by classes, and pure functions being implemented by free functions. Given the complexity of scientific software, object-oriented programming is an essential tool for maintaining intra-package modularity.

From Section III A we established that PluginPlay's main tasks will be to discover new modules and wire them together. From the complexity consideration in Section II B, we expect the call graph for most scientific packages to be significantly more complicated than just calling one or two modules. In PluginPlay modules are represented by the `Module` class (see Figure 2). A "leaf" module, is a module which is a leaf of the call graph, *i.e.*, does not call any other modules. Modules which are not leaves will return control to PluginPlay at least once before the module completes; both leaves and non-leaves return control to PluginPlay upon completion. As a slight aside, in PluginPlay the dynamical nature of the call graph means that a module's leaf status can vary from run to run, or even within the same run! In turn, the effective call graph of the program is assembled by PluginPlay and comprised of modules. Since it will in general take many modules to implement a feature, we also introduce "plugins."

Simply put, plugins are collections of modules that developers choose to distribute together. The extent of a plugin is defined by the developers/maintainers of the plugin's distribution. Plugins are generally large enough to be an independent software repository. Though plugins are distributed as a single entity, for a given calculation, PluginPlay may wire the call graph to use the whole of the plugin or only select modules. For example, at present, the NWChemEx team develops and maintains plugins for self-consistent field (SCF)-like theories (including DFT), and coupled cluster. Users wanting the features included in

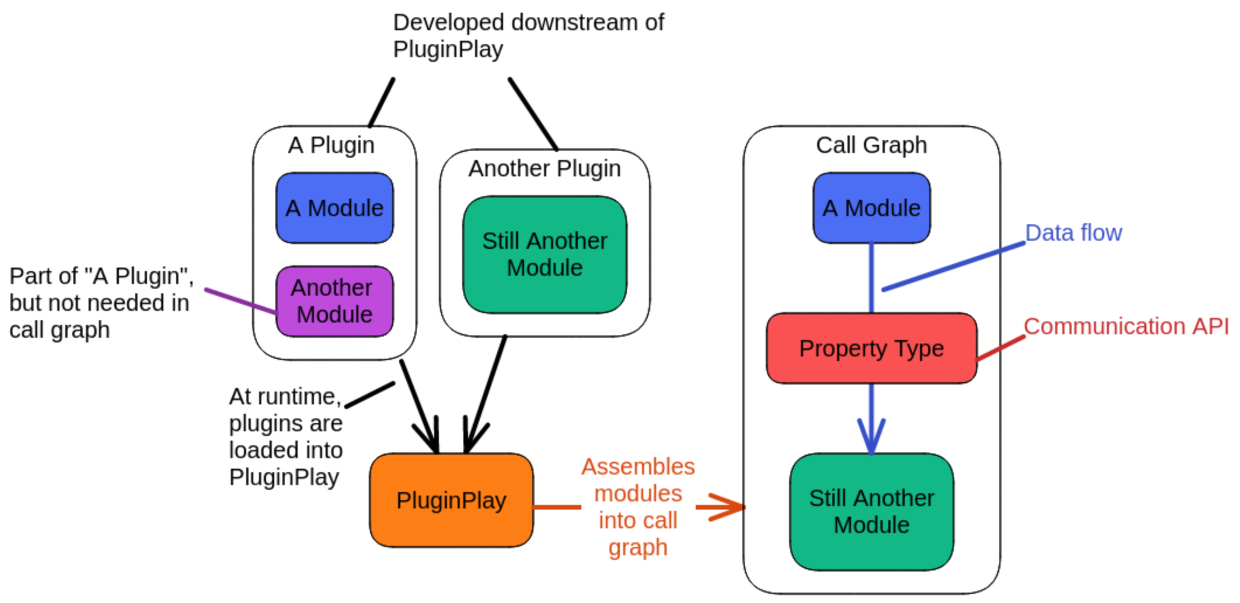

Developed downstream of PluginPlay
A Plugin
A Module
Another Module
Another Plugin
Still Another Module
Part of "A Plugin", but not needed in call graph
At runtime, plugins are loaded into PluginPlay
PluginPlay
Assembles modules into call graph
Call Graph
A Module
Data flow
Property Type
Communication API
Still Another Module

FIG. 2. Illustration of the key concepts in PluginPlay: modules, plugins, the literal PluginPlay framework, call graph assembly, and property types. See text in Section III B for a more detailed description.

a plugin should download the plugin, start up an instance of PluginPlay, and then load the plugin into the PluginPlay instance. Plugins help address the decentralized consideration (II.B.9) from Section. II B, by allowing research groups to maintain and focus on their own set of modules, in relative isolation, if they so choose.

To address II.B.8, the multidisciplinary consideration, PluginPlay treats each module as a black-box. While some modules may be interchangeable, *e.g.*, modules implementing different eigensolvers or modules implementing different SCF guesses, many modules are not, *e.g.*, one can not use an eigensolver (by itself) as an SCF guess. Making matters more complicated, sometimes whether two modules are interchangeable is situationally dependent. For example, under normal conditions the total SCF energy is not interchangeable with the total MP2 energy; however, for the purposes of geometry optimization, the two may be used interchangeably. PluginPlay thus needs a mechanism for knowing when two modules can be used interchangeably, and in response we introduce the "property type" concept, modeled by the `PropertyType` class.

Each property type is a class, the name of which is associated with a property. For example, NWChemEx defines a class `Energy` to be a property type for the total energy of a chemical system, and a different class `FockOp` to be the property type for computing the Fock operator. To avoid biasing PluginPlay towards any one discipline, we have designed PluginPlay's `PropertyType` class so that the exact details of each property type are actually declared and defined downstream of PluginPlay, in the domain-specific software leveraging PluginPlay. In practice property types serve a twofold purpose; in addition to stating which property a module can compute, each property type also establishes the API for computing the titular property. For example, the NWChemEx `Energy` property type declares that modules which compute energies take the chemical system to compute the energy of and return the energy as a double precision number. When a module needs to compute a property, the module signals this to PluginPlay by requesting PluginPlay provide a module to call which satisfies the corresponding property type. PluginPlay then chooses the module to run, runs the chosen module, and returns the requested property. The NWChemEx team is in the process of creating SimDE, a (currently private) GitHub repository which will serve as a centralized community resource for standardizing property types across quantum chemistry. The use of a common set of property types community-wide will facilitate interoperability (consideration II.B.4). An initial release of SimDE is slated for summer 2023.

Property types are used to define how PluginPlay obtains properties from modules. In practice, this means module developers have tremendous flexibility when it comes to how they can implement a module. For example, a module which computes say the DFT energy of a molecular system can do so by wrapping a call to an existing electronic structure package, or by implementing the DFT method itself. As long as the modules satisfy the same property type, they can be used interchangeably. It is worth noting that a module is allowed to satisfy multiple property types. So a module which wraps an existing electronic structure package would, in general, satisfy a lot of property types (in theory one for every property the package can compute). As discussed in Section III D, thanks to memoization calling the same module multiple times, and with the same inputs, will introduce minimal additional cost beyond the first call. In turn, even though each call to a module only returns a single property, modules which compute multiple properties can still be used efficiently.

In our opinion, property types are one of the key pieces missing from all of the existing computational chemistry efforts summarized in Section II. In particular, property types are key to PluginPlay's ability to address II.B.8, the multidisciplinary considerations raised in Section III A, namely by allowing downstream developers to non-invasively extend the

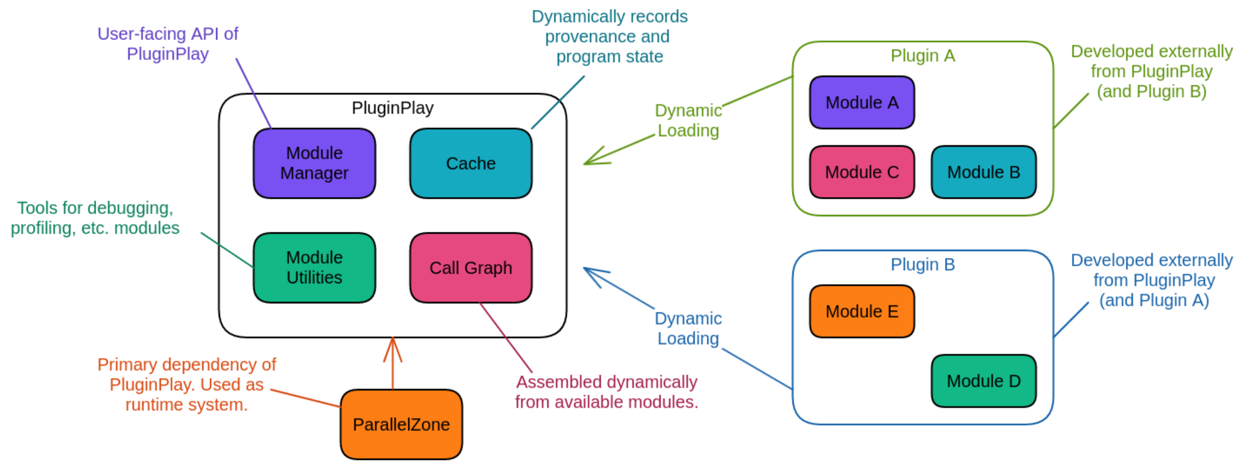

User-facing API of PluginPlay
Dynamically records provenance and program state
PluginPlay
Module Manager
Cache
Tools for debugging, profiling, etc. modules
Module Utilities
Call Graph
Primary dependency of PluginPlay. Used as runtime system.
ParallelZone
Assembled dynamically from available modules.
Dynamic Loading
Plugin A
Module A
Module C
Module B
Developed externally from PluginPlay (and Plugin B)
Dynamic Loading
Plugin B
Module E
Module D
Developed externally from PluginPlay (and Plugin A)

FIG. 3. Illustration of the key components of PluginPlay. See text in Section III B for description.

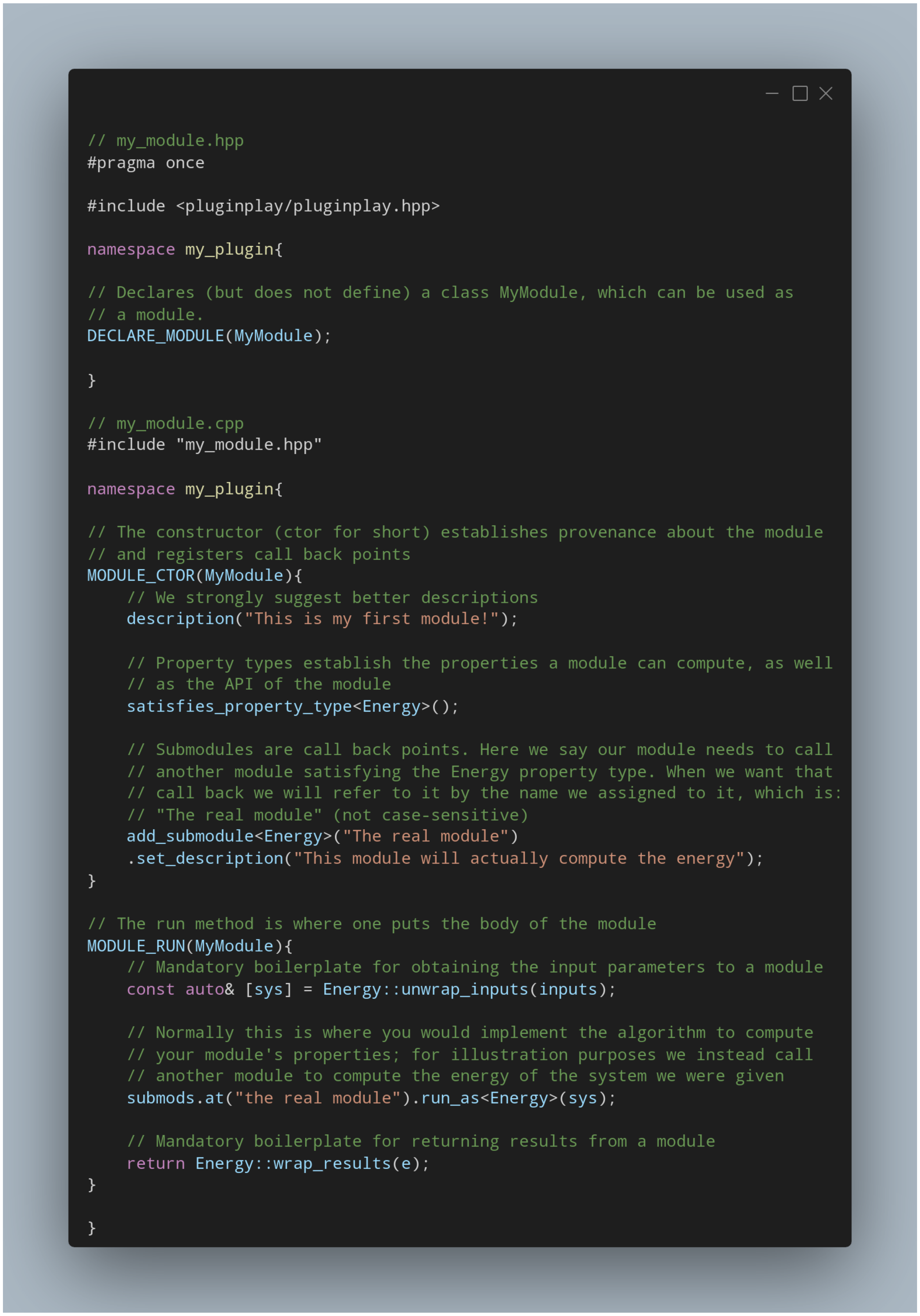
framework to new or domain-specific properties. Furthermore, since the downstream developers can define the property types in terms of native domain-specific object types, property types also allow us to avoid performance issues (consideration II.B.5) at module interfaces, by not requiring data to be serialized, written to disk, or otherwise converted. Finally, property types create a mechanism for enforcing intra-package interoperability by requiring modules to adhere to standardized APIs. If the community as a whole adheres to the same standardized APIs, then property types are also compatible with inter-package interoperability (consideration II.B.4). The PluginPlay concepts introduced so far are summarized in Figure 2.

From the above discussion we have motivated the major concepts of PluginPlay: modules, plugins, and property types. The present architecture of PluginPlay is shown in Figure 3. PluginPlay is built on ParallelZone[54], a runtime system which at present provides object-oriented C++ bindings to the Message Passing Interface (MPI) and some rudimentary hardware information pertaining to what hardware is present. Full details on the design, scope, and features of ParallelZone are beyond our present scope. We direct interested readers to the ParallelZone repository[54] for more details. At present, PluginPlay primarily uses ParallelZone to track the MPI context and forward it to modules; however, ongoing development efforts are looking at expanding that use to include coarse-grained parallelism over the call graph. PluginPlay ensures that each module has access to the current runtime system by providing them direct access to ParallelZone; the module developer is then free to use ParallelZone for their parallel needs or to grab the MPI communicator and drop down to MPI if they so choose (at present each module is responsible for its own thread management). Descriptions and designs for the four major components of PluginPlay (the module manager, cache, module utilities, and call graph) are the subject of the next four subsections. The remainder of Figure 3 serves as a reminder that plugins and modules (and property types) are developed downstream of PluginPlay and added to PluginPlay in a dynamic manner, *i.e.*, at runtime and on-the-fly.

While this subsection motivated the architecture of PluginPlay, it did so starting from very general considerations for scientific software. As other software packages target exascale platforms, many of them will face similar problems as NWChemEx already has, and we close this subsection by summarizing how exascale HPC considerations specifically shaped the architecture of PluginPlay. As stated in Section II A, many scientific software applications will seek strong scaling on exascale hardware, which in turn will require parallelizing nearly the entirety of the package. Being able to piece together already parallelized components will dramatically speed-up such efforts, but assembling disparate components is only viable for highly flexible software packages. Maintaining parallelism over the entire program is facilitated by having a literal representation of the call graph. Using the call graph it is possible to programatically assess what tasks are present, and how they are coupled. The modular nature of the call graph allows performance tuning on a per-module basis. By being able to dynamically swap modules we are able to port large portions of code, most modules of which are not so hardware-specific, to new platforms relatively easily. The introduction of property types played a key role in this design as it facilitated writing PluginPlay in a domain-agnostic manner that supports a modular software stack, without requiring serialization and/or conversions at module interfaces, both of which can be serious bottlenecks for performance.

1. Writing PluginPlay Plugins

Listing 1. “C++ source code showing how to write a module.”

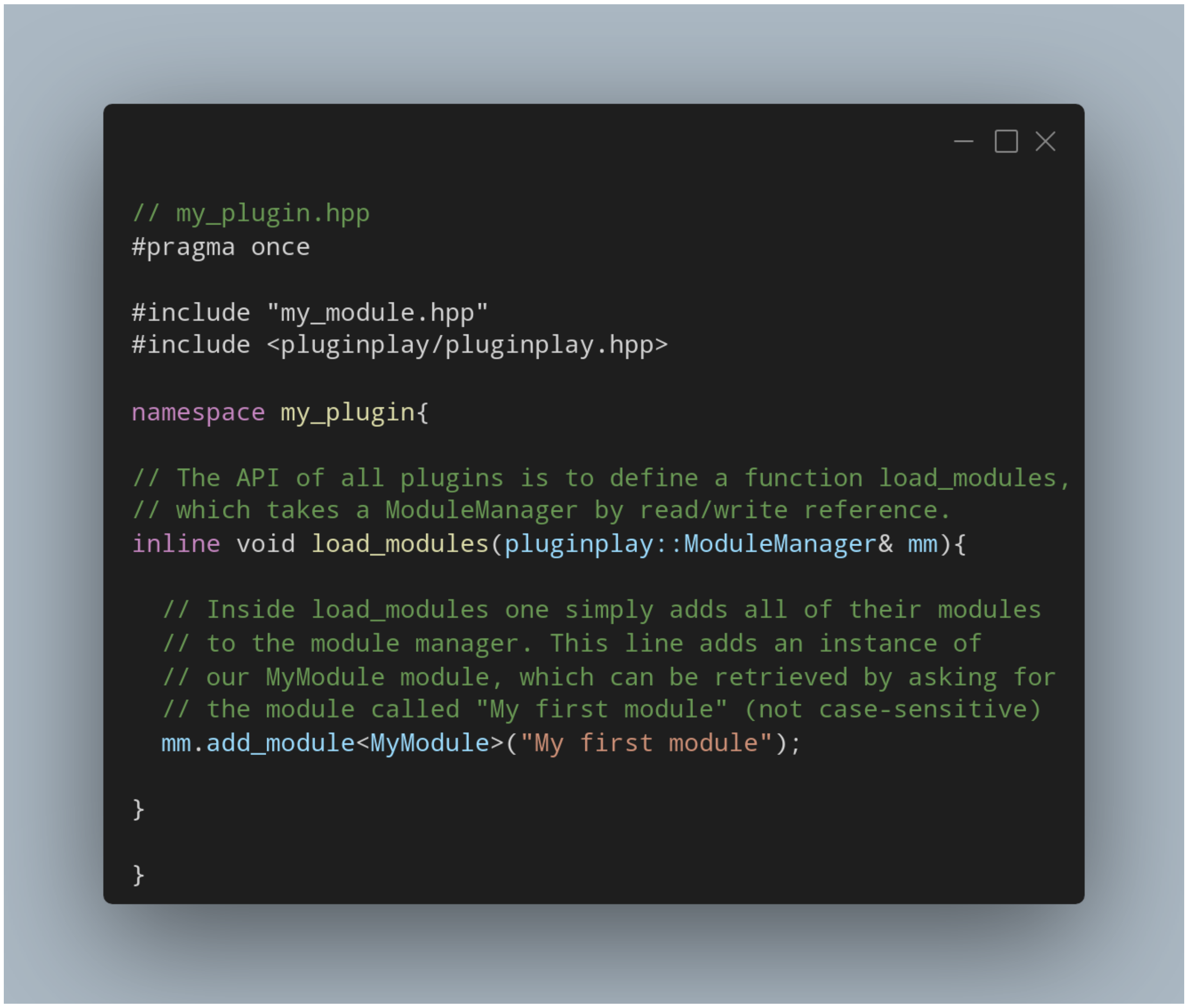


```cpp
// my_module.hpp
#pragma once

#include <pluginplay/pluginplay.hpp>

namespace my_plugin{

// Declares (but does not define) a class MyModule, which can be used as
// a module.
DECLARE_MODULE(MyModule);

}

// my_module.cpp
#include "my_module.hpp"

namespace my_plugin{

// The constructor (ctor for short) establishes provenance about the module
// and registers call back points
MODULE_CTOR(MyModule){
    // We strongly suggest better descriptions
    description("This is my first module!");

    // Property types establish the properties a module can compute, as well
    // as the API of the module
    satisfies_property_type<Energy>();

    // Submodules are call back points. Here we say our module needs to call
    // another module satisfying the Energy property type. When we want that
    // call back we will refer to it by the name we assigned to it, which is:
    // "The real module" (not case-sensitive)
    add_submodule<Energy>("The real module")
    .set_description("This module will actually compute the energy");
}

// The run method is where one puts the body of the module
MODULE_RUN(MyModule){
    // Mandatory boilerplate for obtaining the input parameters to a module
    const auto& [sys] = Energy::unwrap_inputs(inputs);

    // Normally this is where you would implement the algorithm to compute
    // your module's properties; for illustration purposes we instead call
    // another module to compute the energy of the system we were given
    submods.at("the real module").run_as<Energy>(sys);

    // Mandatory boilerplate for returning results from a module
    return Energy::wrap_results(e);
}

}
```

In an attempt to better illustrate the various PluginPlay concepts we show how one could write a trivial C++ plugin. More detailed tutorials, using the most up to date PluginPlay feature set can be found in PluginPlay's documentation[55].

The bulk of any plugin consists of the modules. Listing 1 shows annotated C++ source code for writing a very simple module. Modules are ultimately C++ classes and must be declared and implemented. To simplify this process as much as possible, PluginPlay defines a series of C preprocessor macros (source code in all capital letters); the preprocessor macros hide the boilerplate (and template meta-programming) needed to create and register the classes with PluginPlay. Users are encouraged to use the C preprocessor macros as the underlying declarations are not considered stable parts of the PluginPlay API. In Listing 1 we declare a single module `MyModule`, and then implement `MyModule`'s two methods: the constructor and `run`. A module's constructor is used primarily to associate metadata with the module, register the property type(s) the module satisfies, and register any call back points the module defines. Here we only set the module's description, but each module also has a number of other metadata fields including: authors, references, and version. The remainder of the constructor in Listing 1 establishes that `MyModule` satisfies the `Energy` property type, and that it provides one call back location for another module also satisfying the `Energy` property type. The `run` member of a module is the member which is actually called when a caller runs a module. Most `run` implementations involve a bit of boilerplate relating to unwrapping/wrapping inputs/results. Aside from that, the remainder of the `run` method is the actual module implementation. In Listing 1 we implement our module by calling another module satisfying `Energy` and simply returning its result.

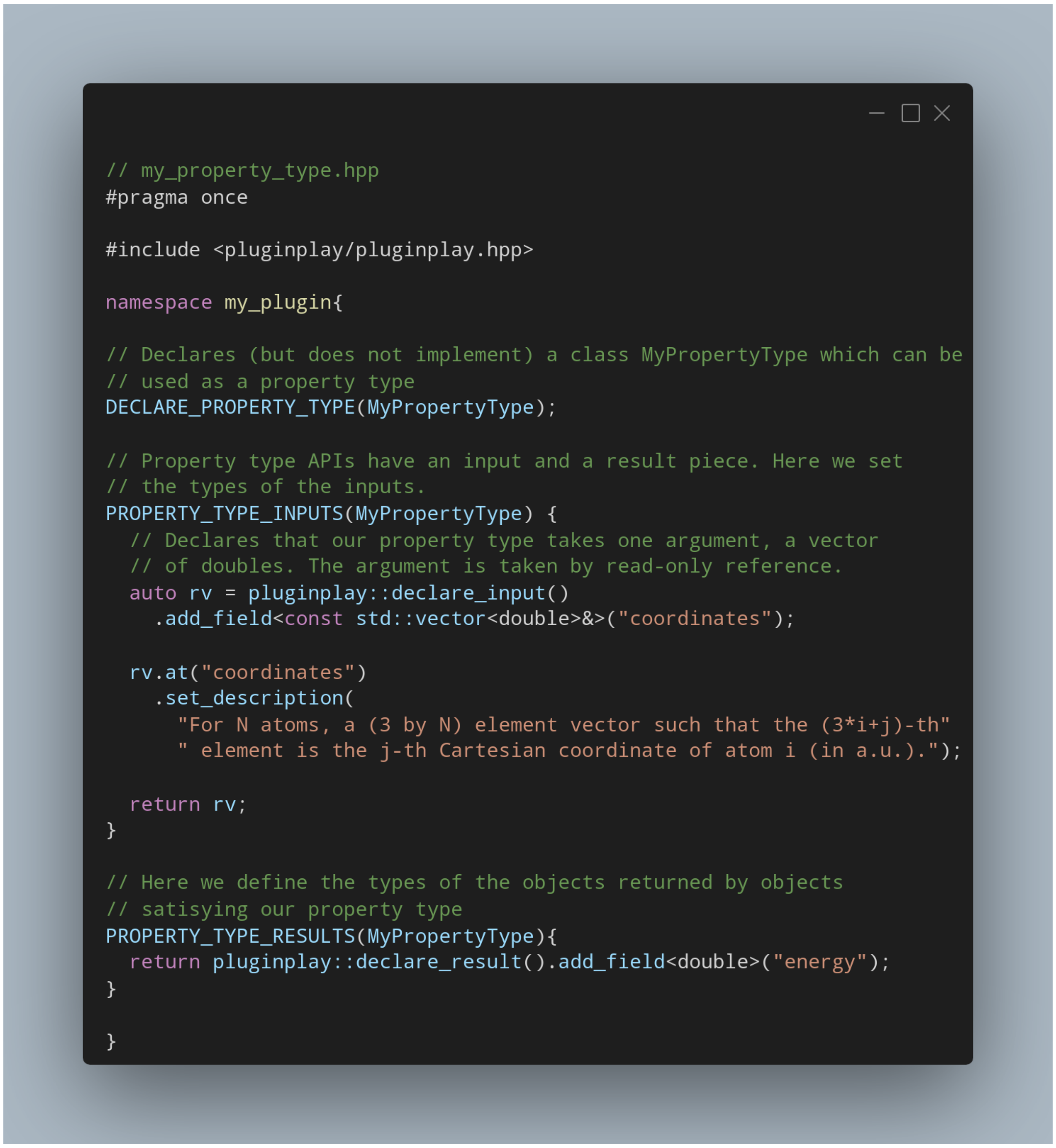
Listing 2. “C++ source code showing how to write a plugin.”



```cpp
// my_plugin.hpp
#pragma once

#include "my_module.hpp"
#include <pluginplay/pluginplay.hpp>

namespace my_plugin{

// The API of all plugins is to define a function load_modules,
// which takes a ModuleManager by read/write reference.
inline void load_modules(pluginplay::ModuleManager& mm){

  // Inside load_modules one simply adds all of their modules
  // to the module manager. This line adds an instance of
  // our MyModule module, which can be retrieved by asking for
  // the module called "My first module" (not case-sensitive)
  mm.add_module<MyModule>("My first module");

}

}
```

Most users of PluginPlay will focus primarily on creating modules, which they will likely add to an existing plugin. As mentioned in Section III B, plugins are nothing more than collections of modules which are distributed together. Listing 2 illustrates how to create a plugin. In order to discover the plugin, PluginPlay requires that all plugins expose a `load_modules` function which takes a mutable `ModuleManager` instance. Inside the `load_modules` function, one simply adds their plugin's modules to the provided module manager.

Listing 3. “C++ source code showing how to write a property type.”



```cpp
// my_property_type.hpp
#pragma once

#include <pluginplay/pluginplay.hpp>

namespace my_plugin{

// Declares (but does not implement) a class MyPropertyType which can be
// used as a property type
DECLARE_PROPERTY_TYPE(MyPropertyType);

// Property type APIs have an input and a result piece. Here we set
// the types of the inputs.
PROPERTY_TYPE_INPUTS(MyPropertyType) {
  // Declares that our property type takes one argument, a vector
  // of doubles. The argument is taken by read-only reference.
  auto rv = pluginplay::declare_input()
    .add_field<const std::vector<double>&>("coordinates");

  rv.at("coordinates")
    .set_description(
      "For N atoms, a (3 by N) element vector such that the (3*i+j)-th"
      " element is the j-th Cartesian coordinate of atom i (in a.u.).");

  return rv;
}

// Here we define the types of the objects returned by objects
// satisying our property type
PROPERTY_TYPE_RESULTS(MyPropertyType){
  return pluginplay::declare_result().add_field<double>("energy");
}

}
```

For completeness Listing 3 shows how to create a property type. Property types should ideally be PluginPlay implementations of community standards. In turn, if a property type already exists for a property, users should prefer that property type over implementing a new one. The result is that users only need to write new property types when their module computes a property for which no standard exists, and needing to write new property types is expected to become rarer as PluginPlay’s ecosystem grows. Since property types define

APIs, we need to define the input fields and result fields, *i.e.*, the types and names of the input arguments and the returned results. As shown in Listing 3, the process is largely identical for the inputs and results. First one calls `declare_input`/`declare_result` for defining inputs or results respectively. Then one chains `add_field` calls for each field (the syntax is a bit odd because behind the scenes a large amount of template meta-programming is occurring). Optionally, the developer may also provide descriptions describing what/how each input/result is used/obtained. The property type shown in Listing 3 declares one input, a read-only reference to `std::vector<double>` object holding the atomic positions, and one result which is of type `double`.

C. Call Graph Design

From the perspective of PluginPlay, a program's call graph is a directed acyclic graph (DAG) where the nodes are modules and the edges point from the calling module to the callee module. Actually discovering the modules, and then subsequently assembling the call graph, is the responsibility of the module manager component described in Section III E. The call graph component focuses purely on representing the DAG. From our discussion so far, we have the following considerations which pertain to the call graph component:

III.C.1 **Extensible**. Sustainability of the call graph component requires the call graph to be extensible to new use cases and features. It also requires us to avoid assuming any particular node is the head node of the DAG, since that limits extensibility.

III.C.2 **Performant**. To avoid introducing performance bottlenecks, assembling and traversing the DAG must be performant. Additionally, we anticipate that the DAG will prove useful down the line for automatic coarse-grained parallelism by allowing us to track task dependencies.

III.C.3 **Multidisciplinary**. PluginPlay will interact with multiple domains, and thus nodes need to support domain-specific types in order to performantly model domain-specific concepts.

III.C.4 **Domain Agnostic**. Seemingly at odds with III.C.3, we also need to avoid coupling PluginPlay to any one domain.

III.C.5 **Dynamic**. Discovery and building the DAG needs to be dynamic to adapt to new use cases and science drivers.

While actually assembling the DAG is the responsibility of the module manager component, we describe the assembly process here that ensures the call graph component has the features necessary to enable assembly. DAG assembly ultimately starts with developers writing modules, since these modules will end up forming the nodes of the DAG. To address the domain-agnostic consideration (III.C.4), the `Module` class will have the same API regardless of its contents (the property type will differentiate the different module interfaces), and to address extensibility (consideration III.C.1) we require all nodes to be instances of the `Module` class, *i.e.*, the root and leaf nodes are not treated any differently. In practice, PluginPlay does not need to actually know the types of the objects traversing the module boundaries. Hence to satisfy III.C.3, the multi-disciplinary consideration, we use a technique known as type erasure. Conceptually, type erasure in C++ can be thought of as simulating a common base class for all objects. Inside PluginPlay, we then pass inputs/results to/from a module via the common base class, and the users downcast them back to their original types. Of note, type erasure is fully type-safe and is implemented using template meta-programming and pointer casts, which allows it to avoid serialization and/or copies, *i.e.*, type erasure also satisfies consideration III.C.2 regarding performance.

Figure 4 illustrates the actual data traversal process. The keys to the entire process are the property types (which we remind the reader are defined downstream of PluginPlay). Property types define the typed API of the module, and they also wrap the machinery to type-erase the domain-specific objects. When a user calls a module they specify the property type to run the module and provide it domain-specific types. This all happens in the user's code. The property type then type-erases the inputs (`AnyField` is the class responsible for the type erasure) and forwards them to PluginPlay. At this point, PluginPlay dispatches to the selected module, and forwards the type-erased inputs into the module developer's code. Inside the module developer's code, the type-erasure process is reversed by passing the type-erased inputs into the property type, which in turn unwraps them back to their domain-specific types. Returning data from a module uses the same process, but in reverse. From the perspective of the call graph's design, the key points are: PluginPlay only interacts with type-erased data and control passes through PluginPlay on its way to/from module developer's code from/to the calling code.

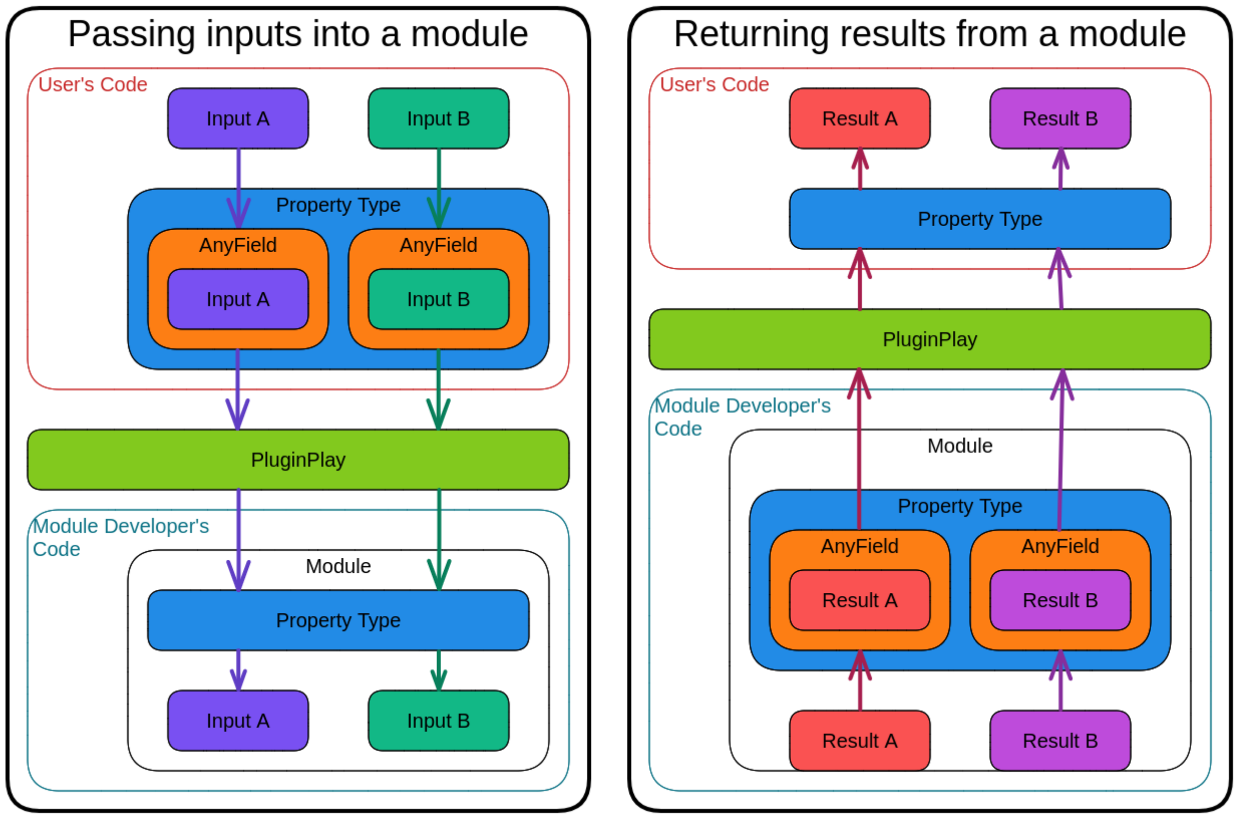


FIG. 4. Illustration of how data traverses module boundaries in PluginPlay. See Section III C for more details.

The fact that control always passes through PluginPlay before/after entering/leaving a module is the key to satisfying consideration III.C.5, dynamic. DAG creation starts when the end user calls the first module. Control flows through the interior of that module, until the module needs to call another module. At this point, the module asks PluginPlay to call a module with a specified property type, and PluginPlay is then able to dynamically decide which of the registered modules to call. In practice, PluginPlay actually assembles the entire DAG upon calling the first module. This is to ensure that the full DAG can actually be formed. By ensuring the DAG can be formed, we avoid running a potentially expensive computation, for a long period of time, only to find that it is not possible to complete the computation.

Figure 5 illustrates how the call graph is actually built. When a module developer writes a module, they specify not only the property type the module satisfies, but also the potential callback points within the module (*n.b.*, the logic inside the module is free to

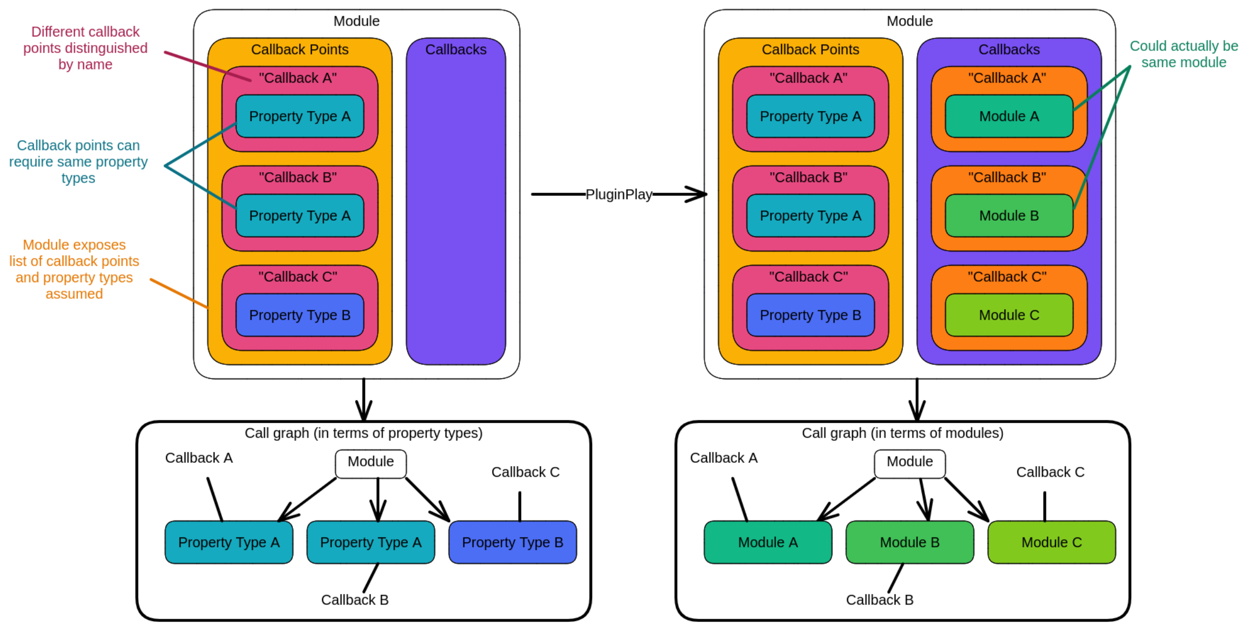


FIG. 5. Illustration of how PluginPlay assembles the call graph. See Section III C for more details.

choose which callback points it actually invokes, and it need not call all of them). Each callback point is assigned a label (`‘‘Callback A’’`, `‘‘Callback B’’`, and `‘‘Callback C’’` in Figure 5) for disambiguation; for example, without labels, and relying only on property type, it would not be possible to distinguish between `‘‘Callback A’’` and `‘‘Callback B’’` in Figure 5. Conceptually this creates the DAG shown in the bottom-left of Figure 5. When DAG creation is triggered by the root module, PluginPlay loops over the registered callback points and assigns them modules (the modules are chosen from among the module pool in the module manager, *vide infra*). The result is the DAG in the bottom-right of Figure 5. This process is then repeated recursively for each of the modules added in the previous step. At present, each module provides defaults for the callbacks in order to facilitate assembling the DAG; users can override the defaults, before DAG creation, in order to call whichever module they choose (so long as the module satisfies the correct property type).

While conceptually simple, the DAG assembly process just described relies on the use of very inflexible property types. The most obvious problem with this design is that many algorithms, and therefore modules, require more inputs than those the property types allow. For example, the `Energy` property type described above only takes as input the chemical system, but many electronic structure methods also need the atomic orbital (AO) basis set, *i.e.*, the contraction coefficients and expansion coefficients for each primitive Gaussian.

There are two potential solutions to this problem: add “module-specific inputs” or, obtain the additional inputs by calling another module. PluginPlay supports both solutions. Module-specific inputs are inputs which must be set by the user before the DAG is created (in practice most module-specific inputs have default values and only need to be set if the user wants a non-default value). Module-specific inputs work best for algorithmic parameters which are rarely changed, or when they are changed, can easily be set by the end user,*e.g.,* convergence tolerances or the maximum number of iterations. For obtaining more complicated inputs, such as the AO basis set, PluginPlay recommends calling another module (in this case a module which returns the AO basis set given the chemical system). In passing we note that modules may return additional information either by satisfying multiple return types, or by defining module-specific results. Satisfying multiple property types works best when a module is being called by another module. In this case, even though the caller defines multiple call back points (one for each property type) each call is wired to the same module. Similar to module-specific inputs, module-specific results work best when it is the end user who is retrieving the result; this is because to retrieve module-specific results one must know they exist without recourse to a property type.

D. Cache Design

The fact that property types are necessarily rigid, combined with the fact that different algorithms need different inputs, led to the PluginPlay design decision of “call another module to obtain the additional inputs.” This is a rather elegant solution in that it allows the property type to capture the inputs common to each algorithm, while still allowing the module developer to encapsulate the process of obtaining the additional inputs. The downside is that not all additional inputs are trivial to compute. In order to adhere to consideration III.A.2, *i.e.*, performant, we need to avoid recomputing non-trivial inputs. Thankfully, software engineers have again already devised a solution: “memoization”, a process where results from expensive function calls are cached to avoid recomputing them.

Conceptually, memoization in PluginPlay is relatively straightforward. For each module, PluginPlay maintains a cache. The cache is an associative array whose keys are the full set of inputs (including the module-specific inputs and any callback modules), and the associated value is the set of results from calling the module with those inputs. Before calling a module, PluginPlay checks the cache to see if the module has already been called with this set of inputs. If it has, the results are simply returned; if it hasn't, the module is run and the results are cached for future usage. While caching/memoization may at first seem like a PluginPlay specific design consideration, it actually turns out to be heavily related to considerations regarding reproducibility (III.A.6 and II.B.2). In turn the considerations for the cache component are:

III.D.1 **Memoization**. The need for the cache component was brought on by needing to memoize execution of the DAG.

III.D.2 **Performant**. The cache component is needed to ensure the call graph component can satisfy consideration III.C.2. It is therefore essential that the call graph component also be performant.

III.D.3 **Reproducible**. The result of running a computation with memoization disabled should be identical to running it with memoization enabled (aside from the time to solution). In turn, the cache also must capture all of the information needed to reproduce the calculation.

Fundamentally, memoization of a function assumes the function is a "pure function.", a function that returns identical results for identical inputs and has no side-effects. The former condition means the output of the function can not vary with changes to global variables, files, or any other inputs not directly passed into the function; as a corollary, it also means that the function must be deterministic. The "no side-effect" condition means the function can not modify global state, files, or any other results not directly returned from the function. Every pure function is trivially reproducible, since, by definition, a pure function is guaranteed to return the same results for the same inputs. Generalizing to the memoization of a module, we define a "pure module" by analogy to a pure function. A pure module: is a module that returns identical results for identical inputs, has no side-effects, and only calls other pure modules. Each pure module individually satisfies the reproducibility condition III.D.3, and it stands to reason that any DAG comprised of pure modules also satisfies III.D.3.

Knowing that a module is a pure module is not enough to satisfy the memoization consideration III.D.1, you also need to build the map from inputs to results. This requires capturing all of the inputs and results for each call and caching them. The full design of the cache is fairly complicated, and beyond our current scope; instead we point out some of the additional design considerations which stem from ensuring the cache satisfies the memoization and performant considerations (III.D.1 and III.D.2). These are considerations that potential solutions to the reproducibility consideration (II.B.2) will likely also need to contend with. These considerations include: data size, recording sufficient provenance, and checkpointing.

Arguably the largest complication is that some of the inputs/results are large data structures. Furthermore, some of these large data structures will be inputs or results to/from multiple modules. Storing multiple copies of a large object is inconsistent with the performance consideration (III.D.2). One potential solution is hashing. Using hashing, each object is associated with a hash, and copies of the hash are stored. Hashing has several problems though: First, hashing a large object can be expensive, and hashing a large distributed object is difficult to do in a performant manner. Second, in the most general scenarios, hashing tends to be invasive because it requires knowledge of the internal state of the object. Third, while hash collisions should be theoretically rare, in practice, naive hashing of objects can result in far more frequent hash collisions; a notable example is that many hashing implementations will compute the same hash for any empty container, even though different containers constitute different inputs. Instead of hashing, PluginPlay's cache component uses universally unique identifiers (UUIDs)[56]. Generating a UUID is a trivial process which is decoupled from the identity of the object (although care needs to be taken when generating UUIDs for distributed objects, since each rank generates a different UUID). Furthermore, UUIDs are guaranteed to be unique.

With the UUID solution, recording sufficient provenance to tell different inputs apart is relatively straightforward. PluginPlay associates a UUID with each object and module. When a particular module is called, PluginPlay recursively records the UUIDs for each input, and the UUIDs for each module which will be called. Realizing that memoization is only attempted with pure modules, this is then sufficient provenance to reliably apply memoization. Looking an input up in the cache involves comparing UUIDs (which are short strings) and mapping objects to UUIDs (which involve value comparisons); anecdotally, the rate-limiting step is almost always the latter. Unfortunately, the overall performance of value comparisons depends on the implementation of the objects being compared and is out of the control of PluginPlay. As such, the performance of value comparison can vary widely from fractions of a second to much worse depending on how the comparison is implemented. We note that the value comparisons in the cache usually result in false (thus performance benefits greatly from short-circuit logic) and value comparisons should scale (at worst) linearly in the size of the object.

Given the importance of the data in the cache, losing it may be detrimental to performance. For this reason, PluginPlay also supports backing the cache up to disk. If in addition to the cache, one also saves the object to UUID mapping, the cache can be used for checkpointing/restarting the calculation. In short, if one pre-populates the cache before running a calculation, control will quickly return to parity with the previous run thanks to memoization. It is worth mentioning that every memoizable module gets checkpoint/restart for free, simply by using PluginPlay. Before concluding this section, we want to note that the cache component is more complicated than alluded to here. Notably, PluginPlay contains mechanisms for deciding what is/is not cached, support for memory hierarchies (combinations of memory and disk), and a mechanism for adding additional database backends. It is entirely possible to use PluginPlay with no cache at all; however, without a cache any performance gains from memoization will be lost. At present the user must be involved with many of these decisions, but our intent is to automate as much of the cache's operation as possible. We also want to note that the user can directly access the cache if they wish.

E. Module Manager Design

The module manager component of PluginPlay is designed to respond to the remaining considerations raised in Section III A:

III.E.1 **Stable**. III.A.1 requires PluginPlay to be extremely stable as instability jeopardizes the entire ecosystem. Developers want guarantees that their plugins will continue to work long term.

III.E.2 **UX**. Also related to III.A.1, the module manager is the API of PluginPlay. Long term sustainability requires the module manager to provide a good UX.

III.E.3 **Module management**. III.A.3 and III.A.4 mean that PluginPlay will need to interact with a diverse, dynamic set of modules. Modules will need to be discovered,

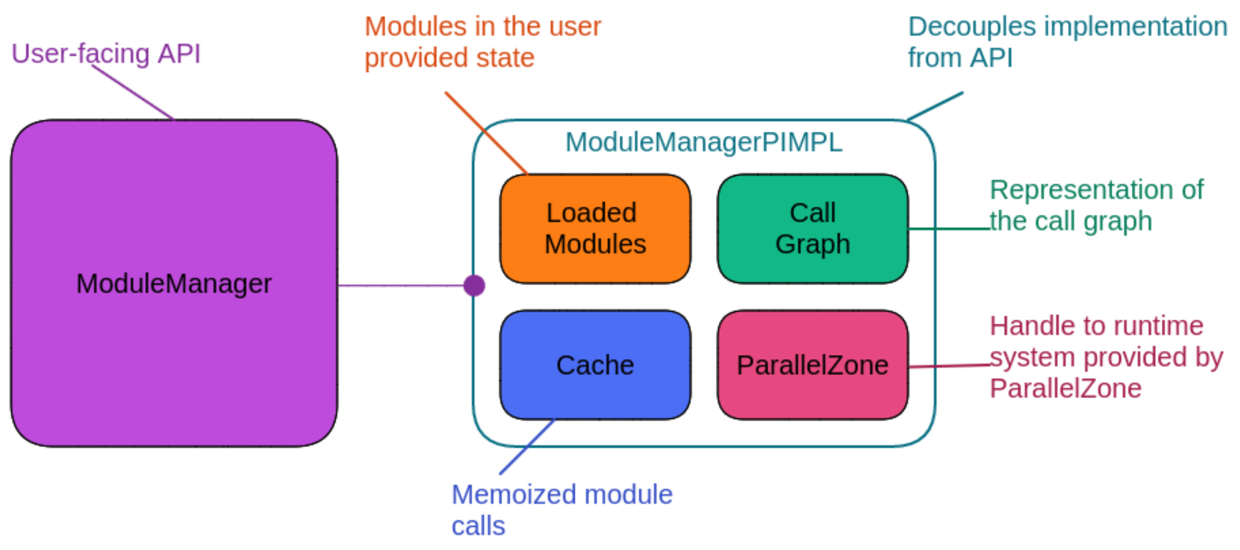

User-facing API
Modules in the user provided state
Decouples implementation from API
ModuleManagerPIMPL
Loaded Modules
Call Graph
Representation of the call graph
ModuleManager
Cache
ParallelZone
Handle to runtime system provided by ParallelZone
Memoized module calls

FIG. 6. High-level design aspects pertaining to PluginPlay's module manager component. See Section III E for more details.

added, modified (usually by changing input values), and potentially removed, all at runtime.

III.E.4 **Wiring**. While wiring was discussed in Section III C, actually responding to III.A.5 falls to the module manager.

The module manager component is modeled by the `ModuleManager` class, the basic design of which is shown in Figure. 6.

The `ModuleManager` class addresses the UX consideration (III.E.2) by providing the API for interacting with PluginPlay. Internally, the `ModuleManager` uses the pointer to implementation (PIMPL) idiom to separate the API from the implementation details, which is a direct response to the stability consideration (III.E.1). With PIMPL, PluginPlay can maintain backwards compatibility, even if the implementation details need to be overhauled. Since a `ModuleManager` instance is essentially an instance of PluginPlay, it is the home for many of the high-level architecture details such as the ParallelZone handle, the cache, and the call graph. For the purposes of this subsection, the only additional feature of the module manager component is the module pool of loaded modules.

To address module management (III.E.3), we have added a module pool to the module manager. The module pool is an associative array where the keys are labels for modules (termed "module keys") and the values are the modules. The module keys are arbitrary strings, aside from the fact that each key must be unique; however, typically keys are actually short descriptions of the algorithm in the module and any notable option values. When a user selects a module to run, they do so from the module pool. This triggers the building of the call graph, and thus consideration III.E.4 is also satisfied by the module pool.

F. Module Utilities

While together the module manager, call graph, and cache components addressed all of the considerations raised in Section III A, PluginPlay actually includes a fourth "module utilities" component. This component additionally helps with III.A.1, III.A.2, and III.A.6 by providing generic tools designed to facilitate software development and maintenance. At present this includes three utilities: documentation generation, profiling, and checkpoint/restart.

When developing a module, the module developer registers the inputs/results the module takes/returns (via its property type) and the callback points it provides. This is done to programatically expose the API of the module to PluginPlay. Since PluginPlay knows the API of the module, PluginPlay can autogenerate API documentation for the module. To increase the usefulness of the generated documentation (and to provide additional provenance to support III.A.6), PluginPlay allows developers to provide additional module metadata such as: module authors, papers to cite, and extended descriptions. At present, the resulting API documentation is generated in reStructured Text (reST) format and is suitable for inclusion in Sphinx[57] documentation; however, tools such as Pandoc[58] can be used to convert to a different markup language. Generating the documentation from the source code has the notable advantage that changes to the source are automatically synchronized with the documentation.

As noted a number of times, PluginPlay was designed with performance, specifically exascale performance, in mind. Key to achieving such performance is profiling. PluginPlay's ability to non-invasively inject tooling into the call graph is a boon for profiling modules. More specifically, at present PluginPlay is able to time each call to a module without the developer needing to add timers. This is useful for determining which routines are causing bottlenecks. The same mechanism can be extended to monitor additional resource usage (*e.g.*, memory, disk, or GPU) on a per module basis. While this functionality is redundant with that provided by existing tools such as Intel's VTune[59], having this functionality native to PluginPlay facilitates more fine-grained profiling, down to a per module invocation basis.

The last notable utility included in PluginPlay is checkpoint/restart. While briefly discussed in Section III D, software built on PluginPlay gets checkpoint/restart for free as a side effect of memoization. At least within electronic structure theory, most packages have very limited checkpoint/restart capabilities. If they exist they are usually limited to reading in SCF/DFT molecular orbitals. Being able to save/load a calculation on a much more fine-grained scale is somewhat of a game changer for a field where week-long, and even month-long, calculations are not uncommon. Furthermore, with the increase in the number of hardware pieces involved in an exascale system, hardware failures are expected to become more common[11]. Being able to checkpoint/restart thus offers a mitigation strategy. Finally, we note that checkpoint/restart is also likely to enable cloud computing applications, where jobs may be suspended and/or moved to make room for other, higher-priority, jobs.

G. Design Summary

The point of Section II and Section III was to enumerate the considerations facing scientific software development at the advent of the exascale era, and to explain how those considerations impacted the architecture and high-level design decisions leading to PluginPlay. Achieving exascale performance requires a large number of software components to execute together in concert. Simply developing a series of disjoint, performant components will not suffice; one needs a means of wiring the performant components together. As this section showed, development of PluginPlay needed to overcome a number of design considerations and challenges. Software design is ultimately a soft science, meaning PluginPlay is not the only possible solution which addresses the considerations raised in Sections II and III and it is possible to develop other solutions; however, to our knowledge, no other solution addressees every consideration raised in Sections II and III. To facilitate the development of additional solutions, we end this section by summarizing the key high-level considerations other potential modular solutions must contend with:

- Sustainability. A lot of time and effort will need to be poured into each component. It is thus imperative that the resulting components be sustainable, and the framework for combining the components also be sustainable.

- Extensible. The fact that the software is for research means that its scope will continue to expand. New use cases and features will need to fully integrate into existing packages. Ensuring hooks exist for these features is complicated by the fact that we often do not know what these features will look like ahead of time, nor do we necessarily know the performance considerations.

- Reproducible. Recording all of the provenance needed to reproduce a calculation is complicated. PluginPlay actually records a fair amount of provenance automatically as a side-effect of memoization. Full automation requires PluginPlay to capture additional provenance (including code versions, and call graphs). At present, PluginPlay is not able to do this, but such features are planned.

- Decentralized. Scientists have a tendency to develop code in relative isolation. Solutions need mechanisms for connecting together potentially disparate contributions.

- Domain-specific types. Many existing solutions make assumptions about the data types at interfaces. Needing to convert, serialize, or use a format like JSON, will be a serious bottleneck for objects with a large amount of state. Furthermore, the objects storing this data are unlikely to be standardized across domains.

- Algorithm inputs. Even if components compute the same property, they may need different inputs. This leads to contention with interoperability efforts which require the components to have the same API.

- Data reuse. Data reuse is extremely complicated when you do not know the inner workings of a component. At the same time, the computational complexity of many scientific algorithms means that missing a reuse opportunity can lead to significant performance bottlenecks.

- Checkpoint/restart. As the number of hardware components in an HPC system increases, hardware failures are expected to be more common. Even if it turns out that hardware failures are still rare for exascale systems, jobs crashing is unlikely to be rare. This is because even in the exascale era, jobs will continue to run out of wall time, power failures will still happen, and users will still specify incorrect/incompatible

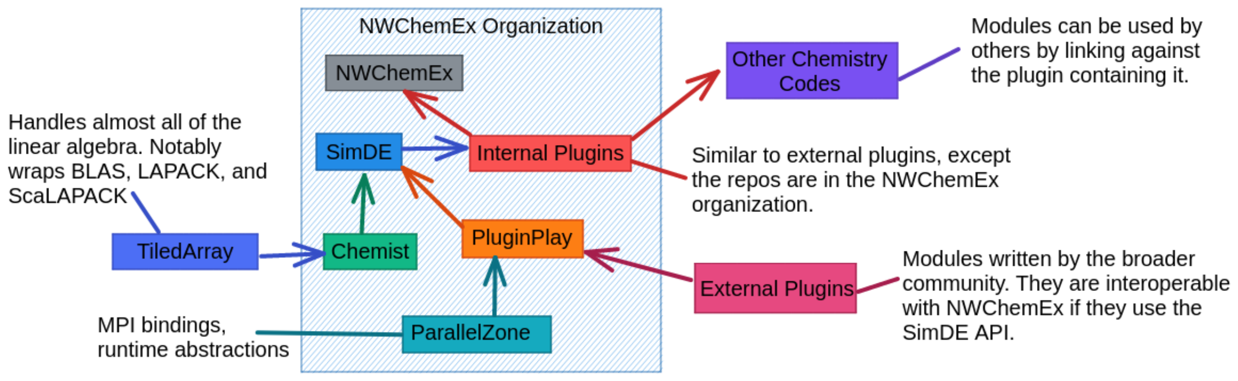

NWChemEx Organization
NWChemEx
SimDE
Internal Plugins
Chemist
PluginPlay
ParallelZone
TiledArray
Other Chemistry Codes
External Plugins
Handles almost all of the linear algebra. Notably wraps BLAS, LAPACK, and ScaLAPACK
MPI bindings, runtime abstractions
Modules can be used by others by linking against the plugin containing it.
Similar to external plugins, except the repos are in the NWChemEx organization.
Modules written by the broader community. They are interoperable with NWChemEx if they use the SimDE API.

FIG. 7. Overview of the NWChemEx ecosystem based on PluginPlay. NWChemEx developers write modules based on SimDE which is PluginPlay plus Chemist plus property types. Chemist defines a series of computational chemistry-specific classes designed for HPC use. The property types provided by SimDE are defined in terms of Chemist's classes. Plugins are managed by PluginPlay, with APIs defined by SimDE.

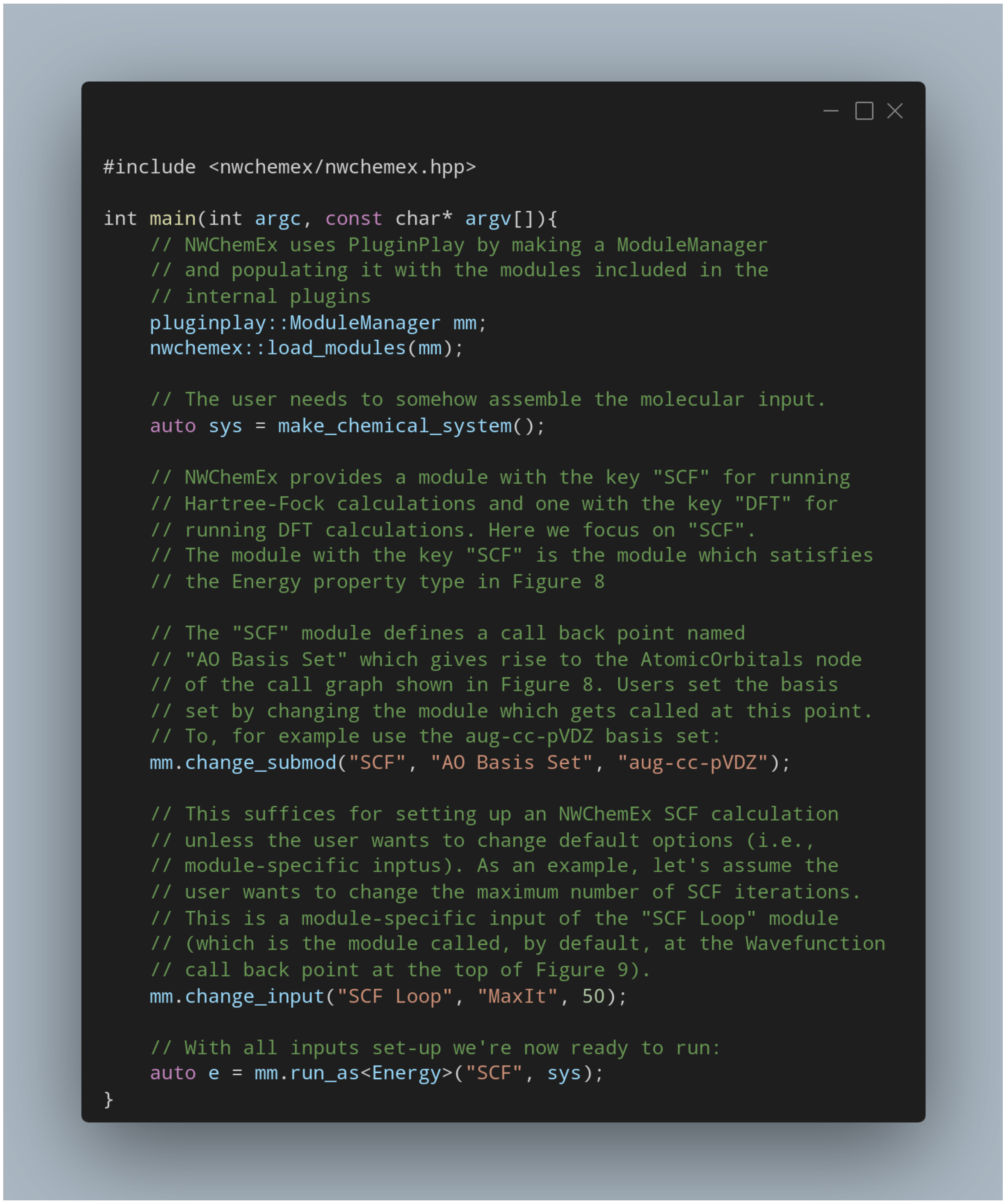
settings. Losing an exascale run is 1000 times more costly than losing a comparable petascale run.

- Performance. Exascale means every step of the process needs to be performant and every one of the aforementioned considerations must be handled in a performant manner.

IV. PLUGINPLAY APPLICATION: HF AND DFT

The present section outlines how the HF and DFT codes are implemented in NWChemEx, with a specific focus on PluginPlay's role in the implementation. For perspective, Figure 7 summarizes the overall NWChemEx ecosystem. Consistent with considerations II.B.3 and II.B.5, Chemist is a stand-alone library containing performant implementations of classes modeling commonly encountered chemistry concepts, including: the chemical system, operators, tensors, and wavefunctions. SimDE uses these classes to define the property types used by NWChemEx's modules. Thus, as the name suggests, SimDE serves as a development environment for developers wanting to write modules which are interoperable with the NWChemEx software stack. The "Internal Plugins" component of Figure 7 contains the first-party PluginPlay plugins developed by the NWChemEx team, a broad term for those contributing to the NWChemEx software stack. The top-level “NWChemEx” component provides a unified UI to NWChemEx akin to a traditional electronic structure software package. Under the hood, the NWChemEx component simply loads the first-party NWChemEx plugins into PluginPlay and relies on PluginPlay to power the software. Most end users interact with the NWChemEx ecosystem purely through the NWChemEx component and do not need to know the remaining details of the software stack. At the time of publication Chemist, SimDE, the HF/DFT plugin, and NWChemEx are still under heavy development and not yet publicly available. We intend initial public releases for the entire stack by the end of 2023.

Listing 4. “Pseudocode for running a Hartree-Fock calculation.”



```cpp
#include <nwchemex/nwchemex.hpp>

int main(int argc, const char* argv[]){
    // NWChemEx uses PluginPlay by making a ModuleManager
    // and populating it with the modules included in the
    // internal plugins
    pluginplay::ModuleManager mm;
    nwchemex::load_modules(mm);

    // The user needs to somehow assemble the molecular input.
    auto sys = make_chemical_system();

    // NWChemEx provides a module with the key "SCF" for running
    // Hartree-Fock calculations and one with the key "DFT" for
    // running DFT calculations. Here we focus on "SCF".
    // The module with the key "SCF" is the module which satisfies
    // the Energy property type in Figure 8

    // The "SCF" module defines a call back point named
    // "AO Basis Set" which gives rise to the AtomicOrbitals node
    // of the call graph shown in Figure 8. Users set the basis
    // set by changing the module which gets called at this point.
    // To, for example use the aug-cc-pVDZ basis set:
    mm.change_submod("SCF", "AO Basis Set", "aug-cc-pVDZ");

    // This suffices for setting up an NWChemEx SCF calculation
    // unless the user wants to change default options (i.e.,
    // module-specific inptus). As an example, let's assume the
    // user wants to change the maximum number of SCF iterations.
    // This is a module-specific input of the "SCF Loop" module
    // (which is the module called, by default, at the Wavefunction
    // call back point at the top of Figure 9).
    mm.change_input("SCF Loop", "MaxIt", 50);

    // With all inputs set-up we're now ready to run:
    auto e = mm.run_as<Energy>("SCF", sys);
}
```

As also shown in Figure 7 the NWChemEx ecosystem contains a number of contact points with the broader computational chemistry community. The first contact point is with the first-party plugins/modules. NWChemEx's first-party plugins are simply C++ libraries, meaning each plugin can easily be used as a dependency of another computational

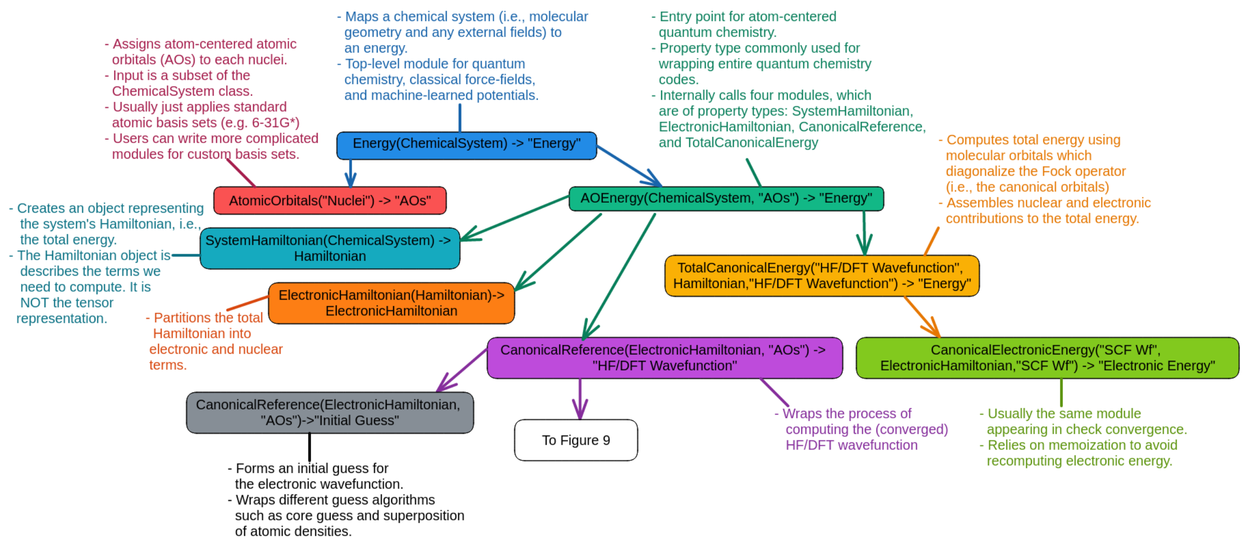

- Maps a chemical system (i.e., molecular geometry and any external fields) to an energy.
- Top-level module for quantum chemistry, classical force-fields, and machine-learned potentials.
- Entry point for atom-centered quantum chemistry.
- Property type commonly used for wrapping entire quantum chemistry codes.
- Internally calls four modules, which are of property types: SystemHamiltonian, ElectronicHamiltonian, CanonicalReference, and TotalCanonicalEnergy
- Assigns atom-centered atomic orbitals (AOs) to each nuclei.
- Input is a subset of the ChemicalSystem class.
- Usually just applies standard atomic basis sets (e.g. 6-31G*)
- Users can write more complicated modules for custom basis sets.
Energy(ChemicalSystem) -> "Energy"
AtomicOrbitals("Nuclei") -> "AOs"
AOEnergy(ChemicalSystem, "AOs") -> "Energy"
- Computes total energy using molecular orbitals which diagonalize the Fock operator (i.e., the canonical orbitals)
- Assembles nuclear and electronic contributions to the total energy.
- Creates an object representing the system's Hamiltonian, i.e., the total energy.
- The Hamiltonian object is describes the terms we need to compute. It is NOT the tensor representation.
SystemHamiltonian(ChemicalSystem) -> Hamiltonian
ElectronicHamiltonian(Hamiltonian)-> ElectronicHamiltonian
TotalCanonicalEnergy("HF/DFT Wavefunction", Hamiltonian,"HF/DFT Wavefunction") -> "Energy"
- Partitions the total Hamiltonian into electronic and nuclear terms.
CanonicalReference(ElectronicHamiltonian, "AOs") -> "HF/DFT Wavefunction"
CanonicalElectronicEnergy("SCF Wf", ElectronicHamiltonian,"SCF Wf") -> "Electronic Energy"
CanonicalReference(ElectronicHamiltonian, "AOs")->"Initial Guess"
To Figure 9
- Wraps the process of computing the (converged) HF/DFT wavefunction
- Usually the same module appearing in check convergence.
- Relies on memoization to avoid recomputing electronic energy.
- Forms an initial guess for the electronic wavefunction.
- Wraps different guess algorithms such as core guess and superposition of atomic densities.

FIG. 8. Non-iterative portion of the call graph for NWChemEx's HF/DFT plugin. A top-down, left-to-right call order is implied. Nodes are labeled with the property type a module must satisfy in order to be callable at that location. The syntax depicted on each node is: the name of the property type, the inputs to the module (in parentheses), and the results (with C++ trailing return syntax). Items appearing in quotes are descriptive placeholders meant to clarify and/or simplify the figure and are not classes contained in Chemist. All other text is the actual type of the object or property type as one would find in Chemist or SimDE respectively.

chemistry code. While not explicitly depicted in Figure 7, NWChemEx actually maintains a series of plugins, with one per major electronic structure theory. This makes it easier for another package to pick and choose which NWChemEx plugins they want to use, *e.g.*, it is possible for another package to use the SCF plugin as a base for many-body theories. The other major contact point is that anyone can contribute to NWChemEx through the external plugins component. This component allows NWChemEx, via PluginPlay, to reuse content developed primarily for use with other computational chemistry packages. More specifically, it is possible to use existing modular software with PluginPlay by wrapping it so that it has a compatible API. Moreover, if the original developers use the property types defined by SimDE, then the resulting modules will exhibit true inter-package interoperability with NWChemEx.

Given the foundational role of HF/DFT in electronic structure theory, the first plugin the NWChemEx team wrote was the HF/DFT plugin. Listing 4 shows an example C++

main function for running an SCF calculation in NWChemEx. The listing shows how this done directly with PluginPlay. The generality of PluginPlay admittedly leads to a verbose API. That said, software packages can use source code like that shown in Listing 4 to implement a more user-friendly UI of their choice. Notably, it is relatively straightforward to map the user interface for most existing electronic structure packages onto a series of PluginPlay calls. Figure 8 and 9 show the call graph PluginPlay generates upon loading the SCF plugin maintained by the NWChemEx team. The call graph has been split into two pieces to enhance readability. Each node in Figures 8 and 9 is labeled with the property type a module must satisfy to be called at that location. So the first module that the user calls must satisfy the `Energy` property type. Following the property type, the values in the parenthesis are the types of the objects which are passed into the module at that point. So the input to the first module will be an object of type `ChemicalSystem`, which is the Chemist class describing the input to the simulation including the molecule(s) and any fields the molecule lives in. After the inputs is the type of the return (using C++ trailing return type syntax). When a type is in double quotes, it means that the type shown in Figure 8 is not the actual type used in the code and defined in Chemist, but a stand-in type used for clarity and/or to simplify the figure. So the return of the first module is an object describing the `"Energy"` of the input chemical system, but since "energy" is in double quotes, we know that the actual returned object's type is not actually depicted in Figure 8 (in this case the actual return type is `double`, which we choose not to show for clarity reasons).

As mentioned in Section III, PluginPlay allows the user to non-invasively change the call graph. Even though Figures 8 and 9 are the default call graphs of NWChemEx's HF/DFT, every node in this call graph represents a customization point that can be changed, at runtime, by the user, all without needing to modify any part of NWChemEx. Thus users are free to use their own modules, or modules from other plugins, throughout the HF/DFT. This is particularly powerful if the user wants to, say, capture the Fock matrix. In this case, the user can write a simple wrapper using a lambda function, which wraps the normal Fock matrix module, and does additional processing of the Fock matrix, *e.g.*, copying it to a variable captured by the lambda function. It is also particularly powerful if the user wants to add additional terms to the Hamiltonian (*n.b.*, the user is not restricted to adding terms to the core Hamiltonian, but can even add density-dependent terms).

While being able inject functionality as just described may seem like "syntactic sugar",

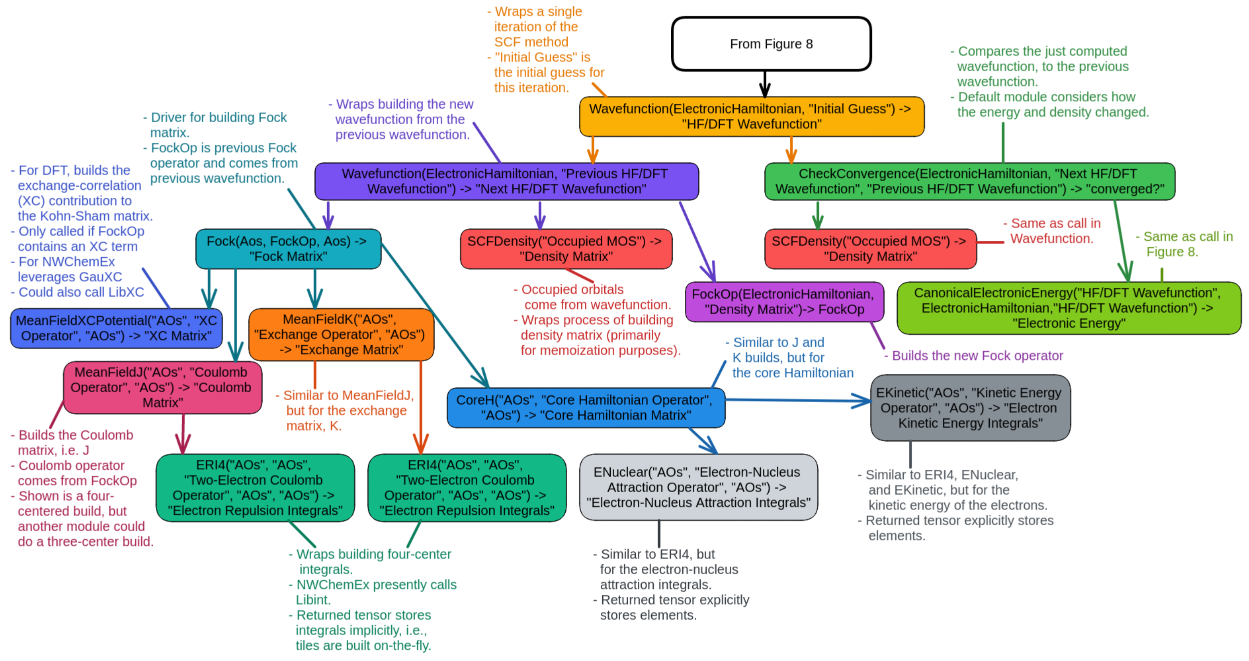

From Figure 8
- Wraps a single iteration of the SCF method
- "Initial Guess" is the initial guess for this iteration.
Wavefunction(ElectronicHamiltonian, "Initial Guess") -> "HF/DFT Wavefunction"
- Compares the just computed wavefunction, to the previous wavefunction.
- Default module considers how the energy and density changed.
- Wraps building the new wavefunction from the previous wavefunction.
- Driver for building Fock matrix.
- FockOp is previous Fock operator and comes from previous wavefunction.
Wavefunction(ElectronicHamiltonian, "Previous HF/DFT Wavefunction") -> "Next HF/DFT Wavefunction"
CheckConvergence(ElectronicHamiltonian, "Next HF/DFT Wavefunction", "Previous HF/DFT Wavefunction") -> "converged?"
- For DFT, builds the exchange-correlation (XC) contribution to the Kohn-Sham matrix.
- Only called if FockOp contains an XC term
- For NWChemEx leverages GauXC
- Could also call LibXC
Fock(Aos, FockOp, Aos) -> "Fock Matrix"
SCFDensity("Occupied MOS") -> "Density Matrix"
SCFDensity("Occupied MOS") -> "Density Matrix"
- Same as call in Wavefunction.
- Same as call in Figure 8.
FockOp(ElectronicHamiltonian, "Density Matrix")-> FockOp
CanonicalElectronicEnergy("HF/DFT Wavefunction", ElectronicHamiltonian,"HF/DFT Wavefunction") -> "Electronic Energy"
MeanFieldXCPotential("AOs", "XC Operator", "AOs") -> "XC Matrix"
MeanFieldK("AOs", "Exchange Operator", "AOs") -> "Exchange Matrix"
- Occupied orbitals come from wavefunction.
- Wraps process of building density matrix (primarily for memoization purposes).
- Similar to J and K builds, but for the core Hamiltonian
- Builds the new Fock operator
MeanFieldJ("AOs", "Coulomb Operator", "AOs") -> "Coulomb Matrix"
- Similar to MeanFieldJ, but for the exchange matrix, K.
CoreH("AOs", "Core Hamiltonian Operator", "AOs") -> "Core Hamiltonian Matrix"
EKinetic("AOs", "Kinetic Energy Operator", "AOs") -> "Electron Kinetic Energy Integrals"
- Builds the Coulomb matrix, i.e. J
- Coulomb operator comes from FockOp
- Shown is a four-centered build, but another module could do a three-center build.
ERI4("AOs", "AOs", "Two-Electron Coulomb Operator", "AOs", "AOs") -> "Electron Repulsion Integrals"
ERI4("AOs", "AOs", "Two-Electron Coulomb Operator", "AOs", "AOs") -> "Electron Repulsion Integrals"
ENuclear("AOs", "Electron-Nucleus Attraction Operator", "AOs") -> "Electron-Nucleus Attraction Integrals"
- Similar to ERI4, ENuclear, and EKinetic, but for the kinetic energy of the electrons.
- Returned tensor explicitly stores elements.
- Wraps building four-center integrals.
- NWChemEx presently calls Libint.
- Returned tensor stores integrals implicitly, i.e., tiles are built on-the-fly.
- Similar to ERI4, but for the electron-nucleus attraction integrals.
- Returned tensor explicitly stores elements.

FIG. 9. Iterative portion of the call graph for NWChemEx'sSCF plugin. See the caption for Fig. 8 and the text in Section IV more details.

keep in mind that the traditional method of adding such functionality is to modify the code with additional logic (*e.g.*, if the user wants the value, do something). Unfortunately, obtaining performance on many accelerators, including GPUs, requires minimizing conditional logic. In turn, PluginPlay's injection mechanism actually makes it easier to port algorithms, one at a time, to GPUs. Our current GPU porting strategy exploits this by starting our porting efforts at the bottom of Figure 9 and working up. Each time we write a new GPU-based module it can immediately be used with the rest of the plugin, no additional dispatch logic required. The black-box nature of each module also facilitates "fusing" modules together. For example, while the CPU implementations of HF/DFT can be written in a performant manner using the call graph shown in Figure 9, achieving excellent performance on GPU requires fusing the integral building and digestion into a single GPU kernel. The resulting kernel can be inserted at the `MeanFieldJ` and `MeanFieldK` nodes, in turn subsuming the `ERI4` modules needed by the CPU versions. The full details of our GPU accelerated HF/DFT plugin are beyond our current scope for this article and interested readers are encouraged to read more details in the article by Williams-Young and co-workers also appearing in this special issue.

The last point we want to make is that the vast majority of the call graph shown in Figures 8and 9 is expected to remain the same regardless of whether or not the leaves of the graph are run on CPUs or GPUs or some other hardware. In turn, it provides a straightforward path to port this HF/DFT plugin to additional hardware architectures and software interfaces. Of note, most of the differences between restricted, unrestricted, and restricted open shell HF occur prior to the leaves of the graph. In turn, while we are at the moment exclusively concerned with porting our restricted HF code to GPU, we anticipate the resulting efforts to immediately enable GPU-enabled unrestricted and restricted open-shell versions as well.

Designing the aforementioned HF/DFT plugin was admittedly a trial-and-error process. The encapsulated nature means that all state a module needs must either be passed into the module, or come from calling a module. While conceptually a simple requirement, note that because each module is developed in relative isolation, and without knowledge of exactly which modules will call it, or be called by it, we need a way to programatically forward assumptions and to explicitly indicate what a module should compute. We found that it was quite natural to express, and manipulate, these assumptions/instructions with a quantum chemistry domain-specific language (DSL) comprised of operators, orbital spaces, and wavefunctions. In particular our implementation of the DSL strives to allow users to compose with these objects in manners akin to how one would actually derive the theory. For example, note near the top of Fig. 8 how the Hamiltonian is split into the electronic Hamiltonian and the (not explicitly shown) nuclear-nuclear repulsion operator. This allows us to programatically, and explicitly, choose the modules for computing the various energy terms. This is an important design aspect because, although we as humans can tell from the names of classes what they are supposed to compute, the program can not. By creating objects to represent the various terms in the Hamiltonian we are able to correctly dispatch in a programatic manner. While beyond the scope of the current article, we have found that similar considerations are necessary for correlated methods. For example, we are able to correctly dispatch between say MP2 and coupled cluster with single, and double excitations (CCSD) based on the wavefunction ansantz (*i.e.*, is the correlated wavefunction created by transforming it by a resolvent or an exponentiated excitation operator).

V. OUTLOOK

Writing software capable of using exascale resources is complicated. In many cases, it will require rewriting entire swaths of code to use new hardware. Often the effort needed for these rewrites is non-trivial and it is imperative that as scientific software targets exascale computing, it does so in a sustainable manner. A key aspect of sustainable computing will be writing reusable, modular software. Ideally, this modular software should be fairly fine-grained, to facilitate easier refactoring as bottlenecks arise and/or porting to new hardware. The present study argues that simply having the modules is not sufficient, one also needs a mechanism for assembling the modules. To this end, we have developed PluginPlay. Developing exascale-ready software is a complicated, iterative process. PluginPlay primarily facilitates this process by simplifying the refactoring efforts inherent to such a process. We specifically want to stress that using PluginPlay offers no more guarantee of exascale performance, than using C++ does. In both cases, the potential for exascale software exists, but whether the final software achieves exascale performance ultimately depends on how the developer uses PluginPlay, or C++. That said, on going efforts within the NWChemEx project offer anecdotal evidence (see for example the concurrent submission, to this special issue, by Williams-Young and co-workers) that it is possible to develop exascale software using PluginPlay.

PluginPlay is an open-source, domain-agnostic, IOC framework for writing scientific software. Using PluginPlay, new algorithms are written as "modules". Modules are the fundamental nodes of the software's call graph. Modules are wired together in order to compute specific features, and the modules needed to support a particular feature are distributed as "plugins." Each plugin is typically developed by a small group of developers working closely together. Similar to web browser plugins, users then choose the plugins they want, add them to PluginPlay, and run the resulting software. Since PluginPlay assembles the call graph dynamically, it is possible to non-invasively modify the call graph for new hardware and/or use-cases without needing to directly modify the existing plugins or modules. In addition to providing runtime functionality for manipulating the call graph, PluginPlay also provides checkpoint/restart, API documentation, and profiling tools which facilitate development of the downstream package. PluginPlay is ultimately a framework for writing software. Computationally expensive algorithms are developed as modules, and anecdotal evidence suggests that PluginPlay itself contributes negligible overhead (fractions of a second) to such algorithms. This is easily understood by virtue of the fact that many of PluginPlay's features are implemented in terms of core C++-language features such as inheritance and pointer casts. The largest caveat to the aforementioned performance statements comes from usual C++ memory considerations, namely users may see performance degradation if they copy large data objects in to/out of PluginPlay; however, PluginPlay fully supports modern C++ techniques for avoiding copies including move semantics and smart pointers.

One of the first applications built on PluginPlay is NWChemEx. As a case study, the current study shows how NWChemEx leverages PluginPlay to non-invasively modify the HF/DFT code to port bottleneck kernels to GPU. Efforts are nearing completion to extend similar functionality to traditional many-body methods and linear-scaling variants of many-body methods. Outside the NWChemEx organization, other applications such as GhostFragment[60], a software suite for fragment-based methods, are being built as PluginPlay plugins in order to directly leverage the PluginPlay ecosystem. While we are optimistic that the ecosystem will continue to grow, we note that to users of GhostFragment, PluginPlay is an implementation detail. Thus even if the PluginPlay ecosystem never extends beyond NWChemEx, it is possible for GhostFragment to still remain useful to other packages, albeit with a little bit of glue code to adapt the property type-based APIs to the APIs expected by the package.

Adding existing libraries to the PluginPlay ecosystem is possible. More specifically, developers must identify the property type(s) an existing library should satisfy, and write glue code to convert the inputs provided by the property type to the inputs the existing library expects. Similar considerations exist for the results as well. Assuming one uses the property types already established in SimDE, the resulting plugin will depend on the initial library and SimDE (and its dependencies including PluginPlay). This process can even be done non-invasively in a separate repository from the original library, if the developer likes.

As we look to the future, we anticipate the number of modular scientific software libraries to continue to increase. This is driven by a number of factors including better software engineering practices, funding agency requirements, and the need to piecemeal port algorithms to new hardware. Such a landscape is prime real estate for PluginPlay, and the software packages built on top of PluginPlay. It is our vision that scientific software will move towards an app-store like future, where users pick the features/methods they want to use from a scientific app store. Users will be able to choose apps from their developers of choice, and for their hardware of choice. PluginPlay serves a key role in this vision by being able to wire those apps together. We also look forward to better integrating PluginPlay with existing quantum chemistry packages, particularly by making contact with concurrent efforts such as those by MolSSI[20]. While such collaborations would most likely initially focus on high-level couplings (energies, potential energy surface scans, geometry optimizations, *etc.*), over time, we anticipate lower-level couplings to become feasible as well. This in turn would facilitate the transition to exascale, not in just NWChemEx, but other packages as well.

ACKNOWLEDGMENTS

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

REFERENCES

[1]G. Moore, "Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff," Solid-State Circuits Newsletter, IEEE **11**, 33 – 35 (2006).

[2]R. Dennard, F. Gaensslen, H.-N. Yu, V. Rideout, E. Bassous, and A. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," IEEE Journal of Solid-State Circuits **9**, 256–268 (1974).

[3]M. Valiev, E. Bylaska, N. Govind, K. Kowalski, T. Straatsma, H. V. Dam, D. Wang, J. Nieplocha, E. Apra, T. Windus, and W. de Jong, "NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations," Computer Physics Communications **181**, 1477–1489 (2010).

[4]R. A. Kendall, E. Aprà, D. E. Bernholdt, E. J. Bylaska, M. Dupuis, G. I. Fann, R. J. Harrison, J. Ju, J. A. Nichols, J. Nieplocha, T. Straatsma, T. L. Windus, and A. T. Wong, "High performance computational chemistry: An overview of NWChem a distributed parallel application," Computer Physics Communications **128**, 260–283 (2000).

[5]E. Aprà, E. J. Bylaska, W. A. de Jong, N. Govind, K. Kowalski, T. P. Straatsma, M. Va-

liev, H. J. J. van Dam, Y. Alexeev, J. Anchell, V. Anisimov, F. W. Aquino, R. Atta-Fynn, J. Autschbach, N. P. Bauman, J. C. Becca, D. E. Bernholdt, K. Bhaskaran-Nair, S. Bogatko, P. Borowski, J. Boschen, J. Brabec, A. Bruner, E. Cauët, Y. Chen, G. N. Chuev, C. J. Cramer, J. Daily, M. J. O. Deegan, T. H. Dunning, M. Dupuis, K. G. Dyall, G. I. Fann, S. A. Fischer, A. Fonari, H. Früchtl, L. Gagliardi, J. Garza, N. Gawande, S. Ghosh, K. Glaesemann, A. W. Götz, J. Hammond, V. Helms, E. D. Hermes, K. Hirao, S. Hirata, M. Jacquelin, L. Jensen, B. G. Johnson, H. Jónsson, R. A. Kendall, M. Klemm, R. Kobayashi, V. Konkov, S. Krishnamoorthy, M. Krishnan, Z. Lin, R. D. Lins, R. J. Littlefield, A. J. Logsdail, K. Lopata, W. Ma, A. V. Marenich, J. M. del Campo, D. Mejia-Rodriguez, J. E. Moore, J. M. Mullin, T. Nakajima, D. R. Nascimento, J. A. Nichols, P. J. Nichols, J. Nieplocha, A. O. de-la Roza, B. Palmer, A. Panyala, T. Pirojsirikul, B. Peng, R. Peverati, J. Pittner, L. Pollack, R. M. Richard, P. Sadayappan, G. C. Schatz, W. A. Shelton, D. W. Silverstein, D. M. A. Smith, T. A. Soares, D. Song, M. Swart, H. L. Taylor, G. S. Thomas, V. Tipparaju, D. G. Truhlar, K. Tsemekhman, T. V. Voorhis, Á. Vázquez-Mayagoitia, P. Verma, O. Villa, A. Vishnu, K. D. Vogiatzis, D. Wang, J. H. Weare, M. J. Williamson, T. L. Windus, K. Woliński, A. T. Wong, Q. Wu, C. Yang, Q. Yu, M. Zacharias, Z. Zhang, Y. Zhao, and R. J. Harrison, "NWChem: Past, present, and future," The Journal of Chemical Physics **152**, 184102 (2020).

[6]K. Kowalski, R. Bair, N. P. Bauman, J. S. Boschen, E. J. Bylaska, J. Daily, W. A. de Jong, T. Dunning, N. Govind, R. J. Harrison, M. Keçeli, K. Keipert, S. Krishnamoorthy, S. Kumar, E. Mutlu, B. Palmer, A. Panyala, B. Peng, R. M. Richard, T. P. Straatsma, P. Sushko, E. F. Valeev, M. Valiev, H. J. J. van Dam, J. M. Waldrop, D. B. Williams-Young, C. Yang, M. Zalewski, and T. L. Windus, "From NWChem to NWChemEx: Evolving with the computational chemistry landscape," Chemical Reviews **121**, 4962–4998 (2021).

[7]A. Petrone, D. B. Williams-Young, S. Sun, T. F. Stetina, and X. Li, "An efficient implementation of two-component relativistic density functional theory with torque-free auxiliary variables," The European Physical Journal B **91** (2018), 10.1140/epjb/e2018-90170-1.

[8]D. B. Williams-Young, W. A. de Jong, H. J. van Dam, and C. Yang, "On the efficient evaluation of the exchange correlation potential on graphics processing unit clusters," Frontiers in Chemistry , Accepted (2020).

[9]P. Messina, "The exascale computing project," Computing in Science & Engineering **19**, 63–67 (2017).

[10] “Exascale computing project,” `https://www.exascaleproject.org/`, accessed: 2-22-2023.

[11] P. Kogge, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Keckler, D. Klein, and R. Lucas, “Exascale computing study: Technology challenges in achieving exascale systems,” Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Techinal Representative **15** (2008).

[12] E. Abraham, C. Bekas, I. Brandic, S. Genaim, E. B. Johnsen, I. Kondov, S. Pllana, and A. Streit, “Challenges and recommendations for preparing hpc applications for exascale,” (2015).

[13] G. Da Costa, T. Fahringer, J. A. R. Gallego, I. Grasso, A. Hristov, H. D. Karatza, A. Lastovetsky, F. Marozzo, D. Petcu, G. L. Stavrinides, D. Talia, P. Trunfio, and H. Astsatryan, “Exascale machines require new programming paradigms and runtimes,” Supercomputing Frontiers and Innovations **2**, 6–27 (2015).

[14] K. Raghavachari, G. W. Trucks, J. A. Pople, and M. Head-Gordon, “A fifth-order perturbation comparison of electron correlation theories,” Chemical Physics Letters **157**, 479–483 (1989).

[15] G. M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS ’67 (Spring) (Association for Computing Machinery, New York, NY, USA, 1967) p. 483–485.

[16] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel programming in OpenMP* (Morgan Kaufmann, 2001).

[17] J. Durrani, “Computational chemistry faces a coding crisis,” `https://www.chemistryworld.com/news/chemistrys-reproducibility-crisis-that-youve-probably-never-heard-of/4011693.article` (2020).

[18] “Better scientific software,” `https://bssw.io/`, accessed: 12-30-2022.

[19] “German society for research software engineers,” `https://de-rse.org/de/index.html`, accessed: 12-30-2022.

[20] “The molecular sciences software institute,” `https://molssi.org/`, accessed: 12-28-2022.

[21] “The nordic research software engineers association,” `https://nordic-rse.org/`, accessed: 12-30-2022.

[22] "Research software alliance," https://www.researchsoft.org/, accessed: 12-30-2022.

[23] "The RSE association of australia and new zealand," https://rse-aunz.github.io/, accessed: 12-30-2022.

[24] "The society of research software engineering," https://society-rse.org, accessed: 12-30-2022.

[25] "Software engineering for science," https://se4science.org/, accessed: 12-30-2022.

[26] "The software sustainability institute," https://www.software.ac.uk/, accessed: 12-30-2022.

[27] "The united states research software engineer association," https://us-rse.org/, accessed: 12-30-2022.

[28] "Working towards sustainable software for science: Practice and experiencies," https://wssspe.researchcomputing.org.uk/, accessed: 12-30-2022.

[29] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral, "The Spack Package Manager: Bringing Order to HPC Software Chaos," (Austin, Texas, USA, 2015) lLNL-CONF-669890.

[30] D. E. Bernholdt, B. A. Allan, R. Armstrong, F. Bertrand, K. Chiu, T. L. Dahlgren, K. Damevski, W. R. Elwasif, T. W. Epperly, M. Govindaraju, D. S. Katz, J. A. Kohl, M. Krishnan, G. Kumfert, J. W. Larson, S. Lefantzi, M. J. Lewis, A. D. Malony, L. C. McInnes, J. Nieplocha, B. Norris, S. G. Parker, J. Ray, S. Shende, T. L. Windus, and S. Zhou, "A component architecture for high-performance scientific computing," International Journal of High-Performance Computing Applications **20**, 163 – 202 (2006).

[31] T. P. Gulabani, *Development of high performance scientific components for interoperability of computing packages*, Phd thesis, Iowa State University (2008).

[32] J. P. Kenny, C. L. Janssen, E. F. Valeev, and T. L. Windus, "Components for integral evaluation in quantum chemistry," Journal of Computational Chemistry **29**, 562–577 (2008).

[33] C. L. Janssen, E. T. Seidl, and M. E. Colvin, "Object-oriented implementation of parallel ab initio programs," in *Parallel Computing in Computational Chemistry* (American Chemical Society, 1995) Chap. 4, pp. 47–61.

[34] C. Peng, C. Lewis, X. Wang, M. Clement, F. Pavosevic, J. Zhang, V. Rishi, N. Teke, K. Pierce, J. Calvin, J. Kenny, E. Seidl, C. Janssen, and E. Valeev, "The massively parallel quantum chemistry program (MPQC), version 4.0.0-beta.1," http://github.com/ValeevGroup/mpqc, accessed: 12-29-2022.

[35]J. M. Turney, A. C. Simmonett, R. M. Parrish, E. G. Hohenstein, F. A. Evangelista, J. T. Fermann, B. J. Mintz, L. A. Burns, J. J. Wilke, M. L. Abrams, N. J. Russ, M. L. Leininger, C. L. Janssen, E. T. Seidl, W. D. Allen, H. F. Schaefer, R. A. King, E. F. Valeev, C. D. Sherrill, and T. D. Crawford, "Psi4: an open-source ab initio electronic structure program," WIREs Computational Molecular Science **2**, 556–565 (2012).

[36]R. M. Parrish, L. A. Burns, D. G. A. Smith, A. C. Simmonett, A. E. I. DePrince, E. G. Hohenstein, U. Bozkaya, A. Y. Sokolov, R. Di Remigio, R. M. Richard, J. F. Gonthier, A. M. James, H. R. McAlexander, A. Kumar, M. Saitow, X. Wang, B. P. Pritchard, P. Verma, H. F. I. Schaefer, K. Patkowski, R. A. King, E. F. Valeev, F. A. Evangelista, J. M. Turney, T. D. Crawford, and C. D. Sherrill, "Psi4 1.1: An open-source electronic structure program emphasizing automation, advanced libraries, and interoperability," Journal of Chemical Theory and Computation **13**, 3185–3197 (2017).

[37]D. G. A. Smith, L. A. Burns, A. C. Simmonett, R. M. Parrish, M. C. Schieber, R. Galvelis, P. Kraus, H. Kruse, R. Di Remigio, A. Alenaizan, A. M. James, S. Lehtola, J. P. Misiewicz, M. Scheurer, R. A. Shaw, J. B. Schriber, Y. Xie, Z. L. Glick, D. A. Sirianni, J. S. O'Brien, J. M. Waldrop, A. Kumar, E. G. Hohenstein, B. P. Pritchard, B. R. Brooks, H. F. Schaefer, A. Y. Sokolov, K. Patkowski, A. E. DePrince, U. Bozkaya, R. A. King, F. A. Evangelista, J. M. Turney, T. D. Crawford, and C. D. Sherrill, "PSI4 1.4: Open-source software for high-throughput quantum chemistry," The Journal of Chemical Physics **152**, 184108 (2020).

[38]Q. Sun, T. C. Berkelbach, N. S. Blunt, G. H. Booth, S. Guo, Z. Li, J. Liu, J. D. McClain, E. R. Sayfutyarova, S. Sharma, S. Wouters, and G. K.-L. Chan, "PySCF: the python-based simulations of chemistry framework," WIREs Computational Molecular Science **8**, e1340 (2018).

[39]A. H. Larsen, J. J. Mortensen, J. Blomqvist, I. E. Castelli, R. Christensen, M. Dułak, J. Friis, M. N. Groves, B. Hammer, C. Hargus, E. D. Hermes, P. C. Jennings, P. B. Jensen, J. Kermode, J. R. Kitchin, E. L. Kolsbjerg, J. Kubal, K. Kaasbjerg, S. Lysgaard, J. B. Maronsson, T. Maxson, T. Olsen, L. Pastewka, A. Peterson, C. Rostgaard, J. Schiøtz, O. Schütt, M. Strange, K. S. Thygesen, T. Vegge, L. Vilhelmsen, M. Walter, Z. Zeng, and K. W. Jacobsen, "The atomic simulation environment—a python library for working with atoms," Journal of Physics: Condensed Matter **29**, 273002 (2017).

[40]"MolSSI Driver Interface (MDI) Library," `https://github.com/MolSSI-MDI/MDI_Library`, accessed: 12-30-2022.

[41] C. R. Jacob, S. M. Beyhan, R. E. Bulo, A. S. P. Gomes, A. W. Götz, K. Kiewisch, J. Sikkema, and L. Visscher, "PyADF — a scripting framework for multiscale quantum chemistry," Journal of Computational Chemistry **32**, 2328–2338 (2011).

[42] S. P. Ong, W. D. Richards, A. Jain, G. Hautier, M. Kocher, S. Cholia, D. Gunter, V. L. Chevrier, K. A. Persson, and G. Ceder, "Python materials genomics (pymatgen): A robust, open-source python library for materials analysis," Computational Materials Science **68**, 314–319 (2013).

[43] D. G. A. Smith, D. Altarawy, L. A. Burns, M. Welborn, L. N. Naden, L. Ward, S. Ellis, B. P. Pritchard, and T. D. Crawford, "The MolSSI QCArchive project: An open-source platform to compute, organize, and share quantum chemistry data," WIREs Computational Molecular Science **11**, e1491 (2021).

[44] V. M. Ingman, A. J. Schaefer, L. R. Andreola, and S. E. Wheeler, "QChASM: Quantum chemistry automation and structure manipulation," WIREs Computational Molecular Science **11**, e1510 (2021).

[45] B. Ralph E.; Foote, "Designing reusable classes," (1988).

[46] "Autowiring: A C++ inversion of control framework," `https://github.com/leapmotion/autowiring`, accessed: 12-28-2022.

[47] "ioc: Inversion of control container c++11," `https://github.com/unixdev0/ioc` (), accessed: 12-28-2022.

[48] "ioc-cpp: Inversion of control/dependency injection container for c++03," `https://github.com/mrts/ioc-cpp` (), accessed: 12-28-2022.

[49] "Pococapsule: An IoC and DSM framework for C/C++ applications," `https://code.google.com/archive/p/pococapsule`, accessed: 12-28-2022.

[50] "Pulsar computational chemistry framework," `https://github.com/pulsar-chem`, [Accessed: 2-6-2023].

[51] R. M. Richard, C. Bertoni, J. S. Boschen, K. Keipert, B. Pritchard, E. F. Valeev, R. J. Harrison, W. A. de Jong, and T. L. Windus, "Developing a computational chemistry framework for the exascale era," Computing in Science & Engineering **21**, 48–58 (2019).

[52] W. T. L. P. Lavrijsen and A. Dutta, "High-performance python-C++ bindings with PyPy and Cling," in *Proceedings of the 6th Workshop on Python for High-Performance and Scientific Computing*, PyHPC '16 (IEEE Press, 2016) p. 27–35.

[53] W. Jakob, J. Rhinelander, and D. Moldovan, "pybind11 – seamless operability between

c++11 and python," (2017), https://github.com/pybind/pybind11.

[54]N. Organization, "Parallel runtime for nwchemex," (2020), https://https://github.com/NWChemEx-Project/ParallelZone.

[55]N. Organization, "Pluginplay," (2023), https://nwchemex-project.github.io/PluginPlay/index.html.

[56]P. Leach, M. Mealling, and R. Salz, "A universally unique IDentifier (UUID) URN namespace," (2005), 10.17487/RFC4122.

[57]"Sphinx python documentation generator," https://www.sphinx-doc.org/en/master/index.html, [Accessed: 2-22-2023].

[58]J. MacFarlane, "Pandoc: A universal document converter," (2006), https://pandoc.org/index.html.

[59]"Intel vtune profiler user guide," `https://www.intel.com/content/www/us/en/develop/documentation/vtune-help/top.html`, accessed: 2-27-2023.

[60]R. M. Richard, "Ghostfragment," (2020), https://github.com/rmrresearch/GhostFragment.