Nested Workflows For Loosely Coupled HPC Simulations *

Wael R. Elwasif*, Ane Lasa†, Philip C. Roth*, Timothy Reed Younkin†, Mark R. Cianciosa*
* Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA
† University of Tennessee, Knoxville, TN 37996, USA
Email: elwasifwr@ornl.gov, aesquisa@utk.edu, rothpc@ornl.gov,
tyounkin@vols.utk.edu, cianciosamr@ornl.gov,

***Abstract*—The increasing complexity of modern scientific simulations has given rise to the notion of re-usability as a means to reduce development time and effort. One form of re-usability involves hierarchical modelling, where the code and artifacts that model a physical phenomenon are used as the building blocks for more complex coupled physical systems. This re-usability mode allows for improvements in constituents sub-models to impact the overall fidelity of the entire system with minimal effort. This benefit, however, hinges upon the ability to provide a fairly loose coupling across model boundaries, where changes in any sub-model do not result in wholesale changes to other sub-models, or in the structure and details of the code for the entire physical system. In this paper, we present the design, implementation, and case studies for incorporating sub-workflows as building blocks in a framework for loosely coupled high performance simulations. We outline the issues involved in providing lightweight customization points for full workflows that allows their use *unchanged* in a nested-workflow setting, while providing each sub-workflow with a separate execution context that ensures non-interference with other parts of the simulation. We present several use cases that demonstrate the successful use of the proposed design and its flexibility in accommodating different customization and scaling requirements.**

I. Introduction

The increasing power and sophistication of high performance computing (HPC) platforms enables the use of higher fidelity models for the simulation of physical systems. Historically, many such models are instantiated using a monolithic approach where all underlying phenomena and models are integrated into a single application, typically using the Single Program Multiple Data (SPMD) parallel programming model. More recently, coupled physics simulations have emerged as a major focus for the HPC community as computing platforms have gained enough power to enable such demanding use [1]. Many coupled models adopt a a Multiple Program Multiple Data (MPMD) parallel programming model within a single parallel code. Where the underlying algorithms and data exchange profile between individual models permit, adopting a loosely coupled approach where individual sub-models are represented using stand-alone (possibly parallel) codes allows for more flexibility. This is particularly true for long lived complex codes that pose significant (tight) integration challenge due to the diversity of programming languages, support libraries, code governance, resource requirements, ..etc. Loosely coupled simulations can be considered as a workflow coordination problem, where aspects such as resource management, task scheduling, data management, and inter-model coordination are handled via a coupling framework. Effective reuse of coupled simulations as a full fledged sub-model in a hierarchical architecture demand a flexible design that allows sub-workflows to be seamlessly customized and integrated into a "higher" model, without the need to change their internal structure.

In this paper we present the design, implementation and use cases for nested workflows in a framework for component-based, loosely coupled HPC simulations. In section II we present details on the architecture of the framework and the services it provides to constituent components that make up a coupled simulation. Section III details the task and resource management capabilities of the simulation framework. In section IV we present the design of nested workflows and their integration into a hierarchical structure within the framework. In Section VI we discuss performance issues and profiling support for nested workflows in the simulation framework. Our conclusions and future work are presented in Section VII.

II. The Integrated Simulation Framework

Many physics applications of interest to have been in use and development for decades, and continue to be developed and used in different contexts. Therefore, it is important that such applications be changed as little as possible for integration in new coupled simulations. These physics applications also vary significantly in their parallel scalability, including codes which are strictly sequential, modestly scalable (typically tens of processors), and highly scalable (hundreds or thousands of processors).

The Integrated Plasma Simulator (IPS) simulation framework [2], [3] is a component-based lightweight coupled simulation framework implemented in Python. Components in the

* This manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

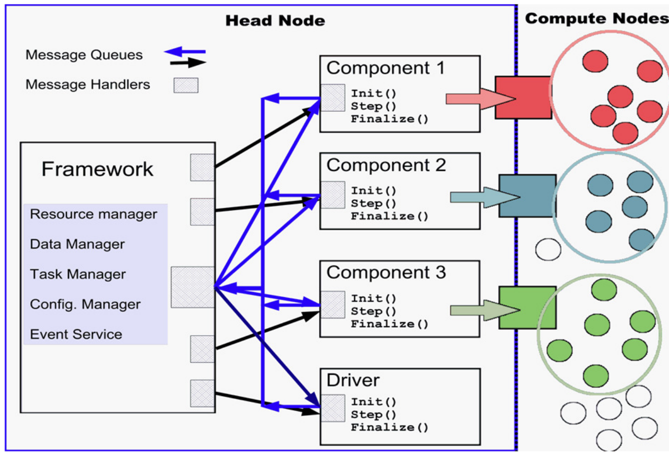

Fig. 1. An illustration of the relationships between the framework, components, services, and tasks in the Integrated Simulation Framework.

framework satisfy a simple interface with `init()`, `step()`, and `finalize()` as the primary methods. Components are unmodified executables of the physics applications, wrapped with Python. The wrapper, along with small "helper" executables, adapts the application's native inputs and outputs to a common inter-component data representation. Components generally exchange small amounts of data (typically a few megabytes) via state files with common formats that can be manipulated by all components in a coupled simulation.

The IPS is designed to execute in self-contained fashion as a batch job or interactively. The core of the framework runs as a single Python process which may be run on a head node, service node, or in the compute partition depending on the system architecture and the capabilities or limitations of each class of nodes. Components are spawned as separate processes, which in turn invoke the underlying physics codes. The framework provides the environment in which components are assembled and executed as a coherent simulation. The IPS makes certain *services* available to components to invoke as appropriate during the various stages of execution. Such services can be broadly categorized into component instantiation and configuration management, resource management, task and remote method execution coordination, and events management. (Figure 1).

Coupled simulations in the framework are typically controlled by "driver" components written in Python. This approach provides users with a familiar procedural way of expressing the simulation workflow with complete flexibility. The driver is responsible for controlling the execution of components that implement the coupled physics. The driver is also responsible for ensuring that any data dependencies between the coupled components are satisfied in the course of executing the simulation. Simulations are typically time-stepped, with multiple components run at each time step in sequence dictated by their data dependencies. The details of these simulations vary with the physics of interest, but in many of the simulations we tackle, time steps range from a few minutes to an hour or more of wall-clock time, and involve many hundreds to a few thousand steps.

Framework services are provided by several *managers*, which collectively constitute the bulk of the framework. The *data manager* is responsible for staging the input, output, and state data for each component, and enforcing the directory structure. The *configuration manager* handles simulation configuration, ensuring separate execution contexts for different components and simulations. It also provides components with querying capability to access configuration parameters. The event service implements a typical publish/subscribe event service for asynchronous communication between components. The task and resource managers work together to execute tasks in a single batch allocation, and are described in detail in Section III. The remaining code in the framework mainly handles simulation start-up and shutdown, and routing of service requests to appropriate managers. Services are remotely accessed by the individual components via an embedded *ServicesProxy* object which implements the Application Programming Interface (API) for the framework services. The *ServiceProxy* object communicates with the central framework managers using a Remote Method Invocation (RMI) protocol, implemented on top of message queues which are provided by the Python multiprocessing standard library module. Details of the different services available through the framework managers can be found in [2].

III. Task and Resource Management

In this section, we describe in some detail the operation of the IPS resource and task managers. We outline the implementation and mechanisms these two subsystems use to enable the execution of component methods, as well as physics tasks launched by the components that make up coupled simulations in the framework.

The task manager mediates two types of activities initiated by components. The first activity is the inter-component invocation of methods defined in the component's API. Inter-component method invocations are handled through the *ServiceProxy* object, which marshals the call (along with any arguments) to the central task manager. The task manager, in turn, routes the method invocation to the target component. Upon method completion, the return code (if any) is routed back to the calling component.

It should be noted that the framework does not support *in situ* changes to arguments of inter-component method invocation. As such, the IPS can be thought to implement a call-by-value regime for inter-component method invocation. The return value from a component method call can be a scalar, or any Python data structure such as a tuple, list, or dictionary. The current interface used by convention for most components consists of a simple method signature, with one argument indicating the physics time associated with the method invocation, and a scalar return code indicating success or failure of the called method.

The *ServiceProxy* and task manager also implement a distributed exception forwarding scheme that simplifies the

handling of abnormal situations in user code. An error in a component method is signaled using a Python Exception, which is either raised explicitly by the callee method, or is generated during the course of executing user code. Such an exception is forwarded by the task manager to the calling components. If the caller component does not adequately trap and process the exception, the exception makes it all the way through the call chain to the original component (typically, the Driver). If the exception is not handled by the Driver component, it is propagated to the framework, which in turn aborts the affected simulation.

The task manager is also responsible for managing the execution of the underlying physics tasks. The *ServiceProxy* object implements a `launch_task()` method that encapsulates the protocol for launching underlying executables. The task manager provides an interface that hides the details of the system task execution mechanisms (e.g., for controlling the number of computing elements to be used, and controlling the number of cores to use in multi-core compute nodes). Tasks may be launched in either "blocking" or "non-blocking" modes, which control the task manager's response when there are insufficient resources available to immediately execute the task. In blocking mode, the task is queued until resources are available to run it, and the `launch_task()` call returns when the task starts executing. In non-blocking mode, `launch_task()` returns immediately with an exception if there are insufficient resources. The task manager uses information provided by the resource manager to determine if the task can be run. Tasks that request more resources than allocated to the entire invocation produce an error in all cases.

In addition to interfacing with the resource manager to determine the availability of resources to launch a task, the task manager constructs the actual launch command that is used to launch the task in the *ServicesProxy* of the launching component. In doing so, the task manager uses information about the execution platform, including the specifics about the batch allocation and the appropriate parallel task execution commands. The task launch interface provides a convenient location for implementing run-time controls to deal with the peculiarities of high end multi-core based platforms. One such control involves assigning compute cores to the user job in such a way as to leave core 0 free (which Cray's Compute Node Linux uses to execute the bulk of the operating system and communication code).

The resource manager is responsible for discovering and allocating the resources available to the framework. Currently it only manages available compute nodes but may, in the future, be extended to manage other resources like storage, memory, I/O or bandwidth. The resource manager queries the native job scheduler and execution environment for the configuration of the computing environment available to the simulation. The resource manager is then used by the task manager to allocate resources to computational tasks as they are launched. The usage model allows both the task manager and resource manager to simply implement FIFO scheduling with backfill based on the number of processes a task requires. Additional

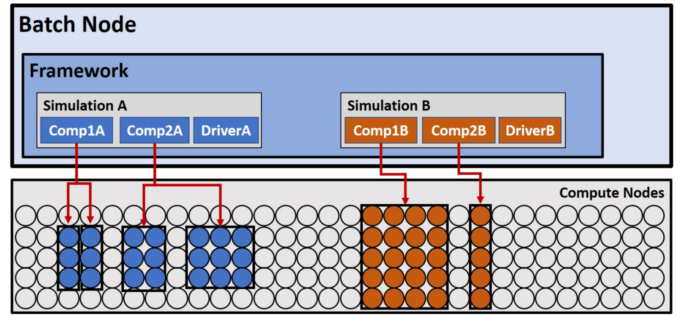

Fig. 2. Multi-level Concurrency in the Simulation Framework

sophistication with regard to allocation and scheduling is not required at this time, thus commercial schedulers and resource managers (like PBS [4]) are not used, rather the framework uses a simple implementations that provides a powerful and flexible compute environment for multiple levels of parallelism. In the current design, the resource manager is used internally by the task manager. No resource manager services are directly accessible by the individual components.

Existing modeling codes span a wide range of parallelism. Some codes remain strictly sequential, while others are quite scalable. Many are in between. In order to allow better control over the trade-off between resource utilization and execution time, the framework supports multiple levels of concurrency, providing the flexibility for users to maximize the number of compute nodes in use at any point in the simulation [5]. Individual computational tasks (the physics executables underlying components) can be parallel, components can launch multiple computational tasks simultaneously (either separately or as part of a single *task pool* that contains an ensemble of independent tasks), multiple components can be active concurrently in a simulation (provided data race conditions are avoided), and multiple simulations can also be run simultaneously. Furthermore, multiple simulations can be dynamically instantiated within a running instance of the framework, all drawing on resources from the common pool allocated when the framework was launched. Figure 2 shows an illustration of these levels of concurrency, with two independent simulations running concurrently within the framework, each with a separate set of components launching different tasks that are scheduled and executed on available compute nodes. It should be noted that this ability to (dynamically) instantiate and execute independent simulations concurrently within a single instance of the framework forms the foundation upon which the IPS provides support for nested workflows as outlined in the following section.

IV. Nested Workflows Design and Implementation

The integration of nested workflows in the simulation framework builds upon the capability to dynamically create new simulations (i.e. workflows) at run time as detailed in [5]. This capability was designed to allow *peer workflows* to be instantiated and executed dynamically within a single instance of the simulation framework, and leveraging a single set of

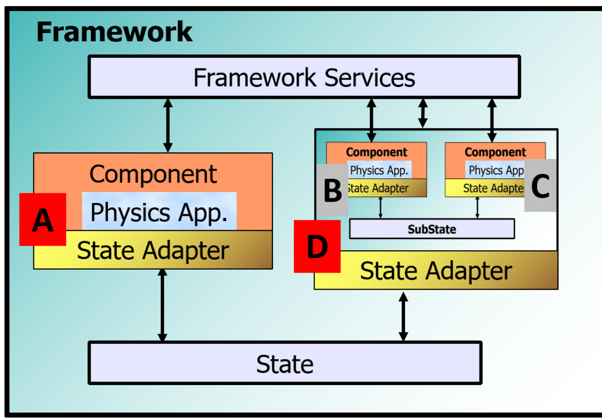

Fig. 3. Nested Workflows in Simulation Framework

compute resources that is shared across all constituent components. This capability underpins the use of the framework as the execution engine for coupled simulations parameter sweeps and optimization [6]. To adapt this capability to support nested workflows, the following design aspects are addressed.

Sub workflow instantiation: A sub workflow is dynamically instantiated when a component in a workflow invokes the framework service `create_sub_workflow()`. Figure 3 shows a schematic for the structure of a nested workflow within the simulation framework. Components **A** and **D** constitute the *main* workflow. Components **B** and **C** constitute a sub workflow that is instantiated by component **D**. The sub workflow executes in its own context (file system space, shared state files, and inter-component communication). The enclosing component **D** is the only component in the outer workflow with access to the state of the **B-C** sub-workflow, and as such, component **D** is responsible for **B-C** configuration and input/output adaptation for integration into the outer workflow. In this regard, the sub-workflow **B-C** is analogous to an executable program that is directly launched in **D**, but with the added property that **B-C** has access to the same framework services as the outer workflow **A-D**.

```
1  sub_name = "subflow1"; cfg_file = "sub.conf"
2  cfg_override = {};
3  input_dir = "sub_data/input"
4  (sim_id, _, driver_h) = \
5      services.create_sub_workflow(sub_name,
6      cfg_file, override, input_dir)
7  services.call(driver_h, "init", "0.0")
8  services.call(driver_h, "step", "0.0")
9  services.call(driver_h, "finalize", "0.0")
10 out_files = \
11     services.stage_subflow_output_files(sim_name)
```

Fig. 4. Sub-Workflow instantiation and invocation

Figure 4 shows an outline for the instantiation and invocation for a single sub-workflow. The call to `create_sub_workflow()` in lines $4 - 6$ takes as input the name of the sub-workflow (which should be unique for all sub-workflows created in any single component). The parameter `cfg_file` refers to the full path of the (original) configuration file for the sub-workflow. Entries in this file can be overriden (e.g. to adjust control parameters to the current stage in he simulation) using the `override` Python dictionary. Input files used in the sub-workflow are (by default) copied to the working directory of the invoking component. This allows the invoking component to make any required updates to those files (if needed). When no changes are needed (and the only customization happens via the control parameters in the configuration file), then the parameter `input_dir` will be used as the location of all input files for the sub-workflow, and no copying is performed during instantiation.

The call to `create_sub_workflow()` returns a three-element tuple that contain a framework generated globally unique simulation id, a handle to the `INIT` component for the sub-workflow (if present), and a handle to the `DRIVER` component in the sub-workflow. The parent component can then proceed to invoke the public methods for the sub-workflow (in the listing shown in Figure 4, no `INIT` component is present in the sub-workflow, so this part of the tuple is ignored. The newly created sub-workflow executes in a unique file system context rooted at the working directory of the invoking component, keeping its data separate from the main workflow, and avoiding any name collisions.

Access to sub-workflow data: A standalone executable has a well-defined specification of its artifacts for configuration, inputs, and outputs. While a sub-workflow can be thought of as a special class of executable code, the output specification of sub-workflows is not immediately obvious. The simulation framework provides data management services that allow individual components to archive their own output files to a separate *simulation results* area. This can happen once per time step, or one time (e.g. when the `finalize()` component method is invoked), or using other scenarios. For sub-workflows to be re-usable from an enclosing component, outputs from the *entire* coupled simulation need to be defined. We have adopted the notion that since the `DRIVER` component is the main entry point into the sub-workflow, and the component and that usually orchestrates the execution from all other components, output from the `DRIVER` component is considered to be the output from the entire sub-workflow. The call on lines $10 - 11$ in Figure 4 copies the output files present in the work directory of the sub-workflow `DRIVER` component to the working directory of the invoking component for further processing. The `stage_subflow_output_files()` method can be used to retrieve outputs from all launched sub-worflows by using the special argument string `"ALL"`. It should be pointed out that the sub-workflow can be executed asynchronously (by using the `call_nonblocking()` method to invoke the `step()` method for the sub-worflow `DRIVER` component. In that scenario, the call to `stage_subflow_output_files()` can be used to get snapshots of the sub-workflow outputs files

as needed while the sub-workflow is executing.

V. Use Cases for Nested Workflows

In this section, we provide an overview of two cases where nested sub-workflows are used to provide seamless hierarchical model re-use. The two use cases presented here are from the general domain of fusion energy modelling, where the complexity of the underlying physics and the interaction between various physics regiments present a great challenge for coupled simulation environments.

A. Inverse problems

```
1  for_each parameter:
2      setup_sub_workflow()
3      run_sub_workflow()
4  while not_at_minimum:
5      for_each parameter:
6          perturb_parameter()
7          run_sub_workflow()
8          reset_sub_state()
9      compute_new_parameters()
10     for_each parameter:
11         set_scaled_parameters()
12         run_sub_workflow()
13     retain_smallest_state()
```

Fig. 5. Pseudo code for Qausi-Newton Control of Sub-Workflow

```
1  set_up_current_state()
2  set_up_child_workflow()
3  run_child_workflow()
4  get_child_result()
5  if child_result_changed:
6      run_current()
7  return result_current_to_parent
```

Fig. 6. Pseudo code for a nested workflow implementation.

Inverse methods seek optimal parameters of a model necessary to match experimental observations. From a model, synthetic signals are calculated. Then parameter space is searched to minimize the mismatch between observed and modeled signals. At the minimum point, the model is now the most probable description of the actual experiment. Having obtained a model of the experiment it is not possible to infer quantities that cannot be directly measured. In fusion, inverse methods are most commonly used to reconstruct the plasma equilibrium [7]–[9].

Inverse methods are typically composed of two parts, the forward model and an optimization searching algorithm. In order to create a generalized optimization workflow, it is best to treat the forward model as a nested workflow. As a proof of principle, the quasi-newton optimization algorithm from V3FIT [8] was re-implemented as a coupled simulation, using a nested workflow as the forward model. The forward model is composed of the VMEC [10] code to evaluate the equilibrium model and the V3POST code [8] to compute synthetic signals. The intent of this workflow is to apply reconstruction techniques to more complex coupled models.

1) Quasi-Newton workflow: The basic operation of the quasi-newton method is to move through parameter space by traveling along the quickest path to the minimum. The quickest path to the minimum is determined by evaluating derivatives of the model with respect to each reconstruction parameter. At each point during the optimization, each parameter is perturbed independently to obtain a numerical derivative of the change in signals with respect to the change in parameter. From the final gradients, uncertainty in the experimental measurements is propagated into the reconstructed parameter space using the same method of V3FIT.

Since the derivative evaluation are independent, they can be run in parallel. A sub-workflow instance for the forward model is created for each parameter that is reconstructed. Sub-workflows are created once at the beginning then reused through the optimization process. Since the sub-workflows are reused, the quasi-newton workflow tracks the state files and resets the sub-workflow state as necessary. Since the perturbations to parameters are relatively small, sub-workflows can make use of restarting functionality to allow quicker convergence to the new perturbed state.

Listing 5 shows pseudo code of the quasi-newton method. After computing the derivatives, a step in parameter space is chosen based on various metrics. Using the parallel capacity setup to compute the derivatives, multiple step sizes are attempted. The step size and sub-workflow state that exhibited the greatest decrease in the optimization figure of merit is retained and synchronized across the sub-workflow instances.

2) Nested Forward Model: For this case, the forward model was implemented as a V3POST workflow containing a nested VMEC workflow. To facilitate the use as nested components, workflows, including the quasi-newton parent, implement the same basic functionality. Listing 6 shows the basic operation of a nest-able component. On the first call of the current workflow, any child workflows are setup once at the beginning and retained for the lifetime of the workflow. In turn, on the first call of the child workflows, their child workflows are recursively created until the last workflow is reached.

To simplify the management of the state files necessary to run a workflow and its potentially deeply nested sub-workflows, each workflow's state files are archived in a zip file. By housing state files in single zip state, the framework only needs to stage one file. Components extract, replace or add files as they are needed. An embedded JSON file, contains meta-data about the zip state. A *state* key entry of the JSON file tracks changes in the state by holding one of the following states.

- `needs_update`
- `updated`
- `unchanged`

By tracking these changes, a component can avoid re-computation when nothing has changed.

Using this system, the state files of child workflows can be easily managed by embedding the child zip state file into the parent state file. In turn, child zip states contain the embedded states of their children workflows. This simplifies the development of deeply nested workflow by hiding details that are unimportant from the parent workflow states. Deeply nested workflows can be swapped out with minimal to no impact on parent workflows. The results of child workflows are passed back to the parent by returning the complete zip state file. Again by checking if a child state is `unchanged`, a parent workflow component can avoid unnecessary computation if it depends on a child workflow result. Additionally, this zip state simplifies restarting deeply workflows since the parent state contains the complete state of the system.

B. Plasma-Surface Interactions (PSI) Nested Workflow

The Plasma-Surface Interactions nested workflow is an integrated simulation composed of high fidelity models which describe physics phenomena in magnetically confined fusion tokamak experimental rectors. In previous code development in the field, the natural division between the physics of the plasma (ionized particles and electrons) and of the materials that face the plasma was set at the material surface. The same division is followed in the present example, where the surface boundary becomes the natural point at which two established work-flows (the edge plasma and impurity fluxes, along with the evolution of the material surface) could be joined as nested sub-flows into a "superflow". The building-block structure of nested work-flows enabled integration of the two physic models with minimal code and component development.

The plasma-surface interactions integrated workflow is composed of two non-concurrent sub-workflows. Sub-workflow1 is first run to completion. This sub-workflow involves the sequential execution of codes in four components. Data generated from sub-workflow1 is consumed by sub-workflow2. This data is then used to launch a set of concurrent independent framework tasks for parallel iterative simulations between two components.

Sub-workflow1 is comprised of a C++ program responsible for processing SOLPS [**?**], [11] data on a single node. Processing time is on the order of 10s of seconds and is therefore not parallelized. SOLPS provides plasma profiles of species, temperature, density, and flux to material target. The next component is responsible for processing data from hPIC [**?**], a particle-in-cell code that provides the distribution in energy and angle of ions near surface. Following the hPIC component, the F-TRIDYN [**?**] component is responsible for launching a parallel program that runs many instances of the Python shared library executable of the FORTRAN code F-TRIDYN (using F2py). F-TRIDYN is a binary collision approximation code that models collisions occurring when plasma ions enter the material surface, with the ability to treat rough surfaces. F-TRIDYN outputs sputtering yields, reflection coefficients, energy-angle distributions of sputtered particles and depth profiles of implanted ions. F-TRIDYN is launched using mpi4py, with one master thread and many parallel workers - typically one per CPU process of the entire job allocation. Typically many thousands of cases of F-TRIDYN are run in order to build and compile a database file which is used in the next component. The final component in sub-workflow1 is GITR, the global impurity transport code for tracing eroded surface particles through the plasma. GITR is a hybrid OPENMP/MPI code which also makes use of the entire job allocation - decomposing simulated particles across nodes. Both number of compute nodes and OPENMP threads must be set accurately to optimize the performance of GITR.

Sub-workflow2 is made up of a specified number of loosely coupled simulations using the Xolotl [**?**] component in a Pickard iteration with the F-TRIDYN component to evolve the surface morphology, implanted gas populations, and self-interstitial and vacancy profiles. With input from sub-workflow1 and updated boundary conditions (implantation profiles) from the F-TRIDYN component, the Xolotl component integrates the time evolution of the surface height and bulk concentration profiles. In these cases, a number of independent simulations are run to simulate many different spatial locations. Using the optimal number of nodes needed by Xolotl, these parallel runs fill the job allocation size. The iterations between F-TRIDYN and Xolotl continue until the simulation reaches its target simulation time or the compute job runs out of time. The net results of this coupled simulation include plasma ion distributions at the surface, net erosion of material, plasma impurity concentration, surface height changes, and gas inventory in the materials.

VI. Performance of Nested Workflows

Understanding the performance of any single application can be challenging, but composing several such applications into nested workflows using a coupling framework can cause a substantial increase in difficulty. The addition of interactions between multiple applications, and between those applications and the framework, greatly increases the complexity of the computation whose performance must be measured and analyzed compared to any of the constituent applications. But the type of framework used, and the functionality it provides, can help control the difficulty caused by this increase in complexity.

As described in Sections I and II, the applications involved in a coupled workflow typically interact with each other via the file system using state files. In a single-level workflow, this application-framework interaction provides a natural serialization point that isolates each application run from others in the coupled simulation. A performance analysis of this type of workflow *could* involve studying each application in isolation, plus an analysis of the I/O costs of reading and writing the plasma state files. A nested workflow such as the plasma surface interaction (PSI) workflow described in Sec. V is similar to a single-level workflow. Although it involved multiple applications, it did not involve running multiple leaf-level workflows concurrently. Nested workflows that involve sub-workflows that are invoked concurrently, or sub-workflows that manage their own collection of concurrent application

runs (e.g., the F-TRIDYN sub-workflow in the PSI workflow), present the greatest performance analysis challenge because of the need to distinguish events occurring in multiple concurrent activities.

The PSI workflow provides an interesting case study for performance analysis of a real-world, nested workflow. The simulation framework does not currently provide functionality explicitly designed for workflow performance analysis. The framework, however, exposes some information that the PSI project team found useful. When managing a workflow, the framework generates a trace of workflow-level events to a log, such as when a workflow step begins or ends. This log can be visualized by a web-based interface, or written to a file. In our studies, the framework was configured to write its log to a file, which can then be post-processed to generate an Open Trace Format version 2 [**?**] (OTF2) event trace file that could be visualized using the Vampir [**?**] event trace visualization tool. Figure **??** shows one such visualization, taken from an early version of the PSI workflow simulating the PISCES linear fusion test stand. The PSI team's strategy provided almost immediate dividends: when presented with this visualization, team members recognized that GITR was running for much longer than it should for the problem being simulated (as shown by the line labeled "gitr_comp" in the figure), and hypothesized that the workflow driver scripts were not configuring OpenMP so that it made use of all the available processor cores. Fixing that configuration problem resulted in much better GITR performance, which greatly improved the overall experiment throughput by allowing the workflow to conduct many more Xolotl-F-TRIDYN iterations than were previously possible (see Figure **??**, comparing the width of the lines labeled "gitr_comp" and the number of F-TRIDYN/Xolotl iterations between the two figures).

The PSI workflow also exposed several ways that the framework could better support performance analysis of nested workflows. One deficiency was that the information captured in the event log was incomplete. For instance, in some cases the log indicated when an activity started but not when it ended. Also, the logs did not provide information about which allocated system nodes are used by a given sub-workflow. Both are needed to determine important workflow characteristics such as when allocated resources were idle, which helps the analyst identify work imbalance within the workflow. The PSI team worked around this problem by having application driver scripts output ad hoc strings containing some of the needed information, and these strings were captured in the same file as the framework event log. However, because the frameowork did not serialize output to this file, there were situations where the output from framework itself and the output from the workflow driver scripts were interleaved, requiring manual separation before the event log could be post-processed into an OTF2 file. Finally, the PSI team's approach did not support capturing application-level performance data that was correlated with the events generated by the framework and the workflow driver scripts. Given these limitations, one potential direction for future work involves the framework providing a low-overhead performance event logging service that the framework, driver scripts, and applications themselves can use to capture performance information about interesting events. To be used by workflow scripts and applications, such a service would have to provide bindings for Python and for traditional HPC programming languages like C++ and Fortran.

VII. Conclusions and Future Work

In this paper we outlined the design, implementation and several use cases of nested workflows in the IPS component-based framework for loosely coupled HPC simulations. We showed how a simple framework service API can be used to adapt an existing workflow for execution in a separate context within a higher level "superflow". The design does not limit this nesting to a single level, and an arbitrary deep hierarchy of workflows can be constructed and executed within a single framework instance, making use of the framework's data, task and resource management services. The fusion plasma simulation use cases demonstrated the utility of the design, and the ability to build sophisticated hierarchical scalable simulations using this approach. This work also identified challenges in performance profiling for this class of coupled HPC codes. Providing better support for standard profiling interfaces will be a focus of our future work. Another focus of future work is enhancing data access for sub-workflows to allow more granular access to intermediate results during the lifetime of the sub-workflow.

VIII. Acknowledgmant

This work has been supported by the U. S. Department of Energy, Offices of Fusion Energy Sciences and Advanced Scientific Computing Research. This work used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231 and the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory which is managed by UT-Battelle, LLC for the U. S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] P. E. Thornton, K. V. Calvin, A. D. Jones, A. V. Di Vittorio, B. P. Bond-Lamberty, L. P. Chini, X. Shi, J. Mao, W. D. Collins, J. Edmonds, and G. C. Hurtt, "Biospheric feedback effects in a synchronously coupled model of human and Earth systems," *AGU Fall Meeting Abstracts*, Dec. 2017.

[2] W. Elwasif, D. Bernholdt, A. Shet, S. Foley, R. Bramley, D. Batchelor, and L. Berry, "The Design and Implementation of the SWIM Integrated Plasma Simulator," in *18th Euromicro Int'l. Conf. on Parallel, Distributed and Network-based Processing (PDP)*, Pisa, Italy, 17–19 February 2010.

[3] S. S. Foley, W. R. Elwasif, A. G. Shet, D. E. Bernholdt, and R. Bramley, "Incorporating Concurrent Component Execution in Loosely Coupled Integrated Fusion Plasma Simulation," in *Component-Based High-Performance Computing (CBHPC)*, Karlsruhe, Germany, 2008.

[4] R. Henderson, "Job scheduling under the portable batch system," in *Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. Feitelson and L. Rudolph, Eds. Springer Berlin / Heidelberg, 1995, vol. 949, pp. 279–294.

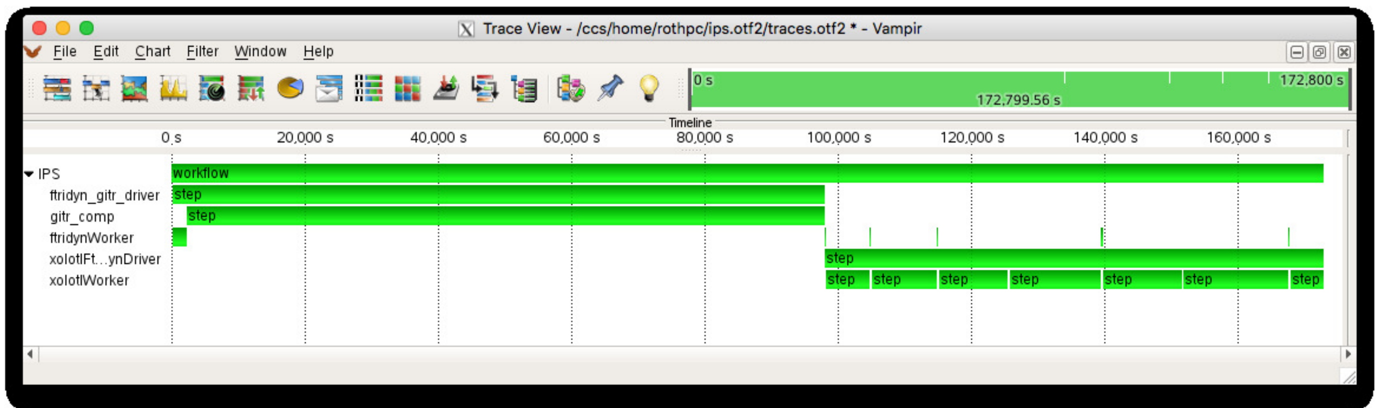

Fig. 7. Timeline visualization of initial 48-hour coupled F-TRIDYN/GITR/Xolotl simulation of He-D plasma in the PISCES linear test stand.

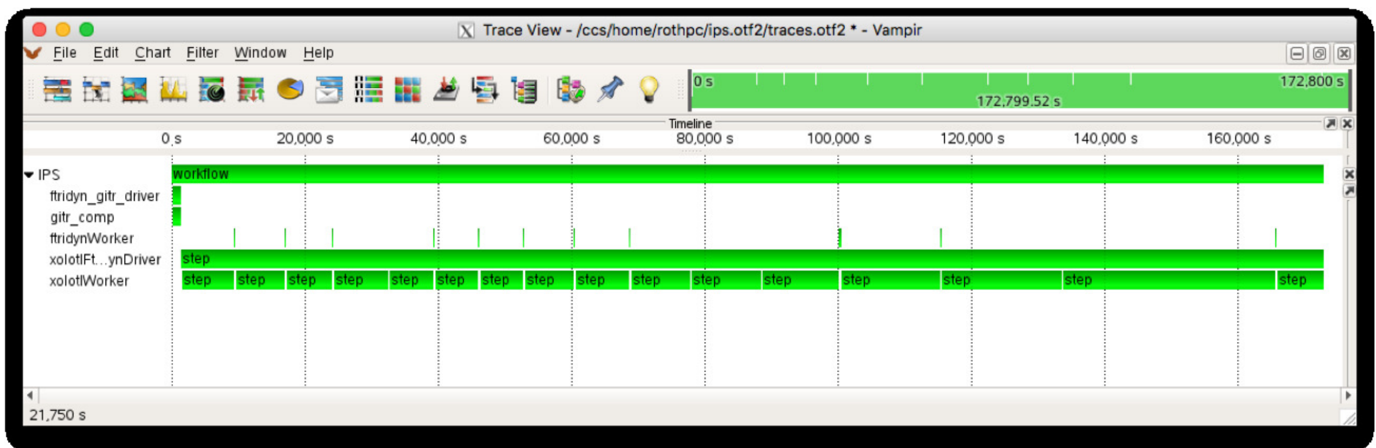

Fig. 8. Timeline visualization of 48-hour coupled F-TRIDYN/GITR/Xolotl simulation of He-D plasma in the PISCES linear test stand after correctly setting the OpenMP configuration for the GITR sub-workflow.

[5] W. R. Elwasif, D. E. Bernholdt, S. S. Foley, A. G. Shet, and R. Bramley, "Multi-level concurrency in a framework for integrated loosely coupled plasma simulations," in *2011 9th IEEE/ACS International Conference on Computer Systems and Applications (AICCSA)*, Dec 2011, pp. 188–195.

[6] W. R. Elwasif, D. E. Bernholdt, S. Pannala, S. Allu, and S. S. Foley, "Parameter sweep and optimization of loosely coupled simulations using the dakota toolkit," in *2012 IEEE 15th International Conference on Computational Science and Engineering*, Dec 2012, pp. 102–110.

[7] L. Lao, H. S. John, R. Stambaugh, A. Kellman, and W. Pfeiffer, "Reconstruction of current profile parameters and plasma shapes in tokamaks," *Nuclear Fusion*, vol. 25, no. 11, pp. 1611–1622, nov 1985. [Online]. Available: https://doi.org/10.1088%2F0029-5515%2F25%2F11%2F007

[8] J. D. Hanson, S. P. Hirshman, S. F. Knowlton, L. L. Lao, E. A. Lazarus, and J. M. Shields, "V3fit: a code for three-dimensional equilibrium reconstruction," *Nuclear Fusion*, vol. 49, no. 7, p. 075031, jul 2009. [Online]. Available: https://doi.org/10.1088%2F0029-5515%2F49%2F7%2F075031

[9] M. Cianciosa, S. P. Hirshman, S. K. Seal, and M. W. Shafer, "3d equilibrium reconstruction with islands," *Plasma Physics and Controlled Fusion*, vol. 60, no. 4, p. 044017, mar 2018.

[10] S. P. Hirshman and J. C. Whitson, "Steepestdescent moment method for threedimensional magnetohydrodynamic equilibria," *The Physics of Fluids*, vol. 26, no. 12, pp. 3553–3568, 1983. [Online]. Available: https://aip.scitation.org/doi/abs/10.1063/1.864116

[11] D. P. Coster, X. Bonnin, and M. Warrier, "Extensions to the SOLPS edge plasma simulation code to include additional surface interaction posibilities," *Physica Scripta*, vol. T124, pp. 9–12, apr 2006.

[12] S. Wiesen, D. Reiter, V. Kotov, M. Baelmans, W. Dekeyser, A. Kukushkin, S. Lisgo, R. Pitts, V. Rozhansky, G. Saibene, I. Veselova, and S. Voskoboynikov, "The new solps-iter code package," *Journal of Nuclear Materials*, vol. 463, pp. 480 – 484, 2015, pLASMA-SURFACE INTERACTIONS 21. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0022311514006965

[13] R. Khaziev and D. Curreli, "hpic: A scalable electrostatic particle-in-cell for plasmamaterial interactions," *Computer Physics Communications*, vol. 229, pp. 87 – 98, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0010465518301012

[14] J. Drobny, A. Hayes, D. Curreli, and D. N. Ruzic, "F-tridyn: A binary collision approximation code for simulating ion interactions with rough surfaces," *Journal of Nuclear Materials*, vol. 494, pp. 278 – 283, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S002231151730301X

[15] "https://github.com/ornl-fusion/xolotl," 2019.

[16] D. Eschweiler, M. Wagner, M. Geimer, A. Knpfer, W. E. Nagel, and F. Wolf, "Open Trace Format 2: The Next Generation of Scalable Trace Formats and Support Libraries," in *Applications, Tools and Techniques on the Road to Exascale Computing / ed.: K. De Bosschere, E.H. D'Hollander, G.R. Joubert, David Padua, Frans Peters, Mark Sawyer, IOS Press, 2012, Advances in Parallel Computing, Vol. 22. - 978-1-61499-040-6. - S. 481 - 490*, 2012, record converted from VDB: 12.11.2012. [Online]. Available: http://juser.fz-juelich.de/record/17929

[17] H. Brunst and M. Weber, "Custom hot spot analysis of hpc software with the vampir performance tool suite," in *Tools for High Performance Computing 2012*, A. Cheptsov, S. Brinkmann, J. Gracia, M. M. Resch, and W. E. Nagel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 95–114.