The Integrated Plasma Simulator: A Flexible Python Framework for Coupled Multiphysics Simulation

Samantha S. Foley, Wael R. Elwasif, David E. Bernholdt
Computer Science and Mathematics Division
Oak Ridge National Laboratory, Oak Ridge, TN, USA
Email: {foleyss, elwasifwr, bernholdtde}@ornl.gov

***Abstract*—High-fidelity coupled multiphysics simulations are an increasingly important aspect of computational science. In many domains, however, there has been very limited experience with simulations of this sort, therefore research in coupled multiphysics often requires computational frameworks with significant flexibility to respond to the changing directions of the physics and mathematics.**

This paper presents the Integrated Plasma Simulator (IPS), a framework designed for loosely coupled simulations of fusion plasmas. The IPS provides users with a simple component architecture into which a wide range of existing plasma physics codes can be inserted as components. Simulations can take advantage of multiple levels of parallelism supported in the IPS, and can be controlled by a high-level "driver" component, or by other coordination mechanisms, such as an asynchronous event service.

We describe the requirements and design of the framework, and how they were implemented in the Python language. We also illustrate the flexibility of the framework by providing examples of different types of simulations that utilize various features of the IPS.

***Keywords*-coupled multiphysics simulation, python, experience report**

I. Introduction

High-fidelity coupled multiphysics simulations are an increasingly important aspect of computational science. As growth in computational capability has supported significant increases in the resolution, fidelity, and overall size of "single physics" simulations, researchers in many domains have recognized that the interactions between different physical phenomena are often more important to scientific understanding than further improvement in the individual models. Moreover, the same rapidly increasing computational capabilities make exploring complex coupled multiphysics increasingly tractable.

However for many areas of computational science, coupled multiphysics is, to a significant extent, uncharted territory—there is little practical experience with simulations of this sort. Where such experience does exist, the result is a variety of frameworks responding to the distinctive aspects of the physics and specific requirements of the domain, as well as to the backgrounds and preferences of the research team. While space does not permit a comprehensive list of prior work in coupled multiphysics, selected examples in climate modeling [1]–[4], rocket simulation [5], accidental fires and explosions [6], and astrophysics [7] are indicative of the diversity of approaches, both within and across domains.

In this paper, we focus on the simulation of fusion plasmas. The plasma physics community has a long history of modeling and simulation, but primarily of isolated physics phenomena in high fidelity while using much reduced physics in integrated modeling. However recognition that high-fidelity, fully integrated "whole device modeling" would be necessary to take full advantage of the ITER experimental fusion reactor [8] ignited interest throughout the international fusion community in coupled simulation, as embodied in the U.S. vision for a large-scale Fusion Simulation Project (FSP) [9]. Starting in 2005, the U.S. Dept. of Energy funded three projects to explore selected couplings as prototypes for a more fully integrated FSP [10]–[12], while similar projects were getting underway in Europe and Japan [13], [14].

The Center for Simulation of RF Wave Interaction with Magnetohydrodynamics (SWIM) "proto-FSP" project [12] is devoted to improving the understanding of the interactions of radio frequency (RF) waves with extended magnetohydrodynamic (MHD) phenomena, which plays an important role in the stability and control of plasmas. A typical problem for SWIM involves the time-dependent modeling of significant portions of a plasma discharge, ranging from a few tenths of a second to hundreds or several thousand seconds. There can be many different time scales involved in the phenomena being modeled, so appropriate time steps within the simulation can span several orders of magnitude, up to about a second. The physics of these simulations is relatively weakly and loosely coupled, in that components representing different physics phenomena can exchange modest amounts of data at the end of each time step, rather than having to be solved jointly (explicit vs. implicit coupling).

In order to support exploration of the loosely-coupled multiphysics phenomena of interest to the SWIM team, we developed the Integrated Plasma Simulator (IPS). The primary goal of the IPS was to provide a flexible framework that allowed the domain scientists to focus on the physics and math issues uncovered in the course of their research,

and anticipate the range of computational capabilities needed to facilitate novel solutions to these challenges. Given that the majority of multiphysics research is focused on more tightly coupled problems, where the issues are more problem-specific, a secondary longer-range goal of this work was to make the IPS sufficiently general for use on loosely-coupled problems outside of plasma physics, to explore the applicability of the computational abstractions developed for the SWIM project.

The choice of Python as the implementation language for the IPS was not dictated *a priori*, but rather came from an analysis of the requirements for the framework, and an assessment of several alternative approaches as described in the following sections. In Sec. II we lay out the key requirements for the IPS, and explain how they drove the framework's design (Sec. III). We then describe the Python-based implementation of the IPS, emphasizing some of the Python modules that greatly simplified the task (Sec. IV). Sec. V provides brief descriptions of some of the very different kinds of simulations for which the IPS is currently being used in order to give a sense of the flexibility provided by the framework's design and implementation. Finally, we close by summarizing our experience constructing a multiphysics simulation framework in Python, and ideas for future work (Sec. VI).

II. Requirements for the Integrated Plasma Simulator

At the start of the SWIM project, the fusion community had very limited experience with coupled simulation, so that flexibility of the framework to accommodate rapid experimentation involving components with wide variations in physics, mathematical, and computational characteristics was a fundamental requirement. To promote flexibility at both the scientific and framework levels during the design phase, different classes of physics modeling capabilities were identified, representing a superset of the physics needed for the SWIM project itself. Then for each class of physics, multiple existing implementations of the required capabilities were identified and analyzed to ensure that they could be integrated, from both the scientific and computational perspectives. A particularly illuminating strategy in this context is to consider both high-fidelity and reduced models of the physics, which are at opposite ends of the spectrum of physical fidelity.

The basic physics of interest to SWIM has, to a large extent, already been embodied in the many standalone physics codes which have been developed over decades of research in the community. While we anticipated the possible need to extend some codes as SWIM's investigations of the coupled physics exposed new issues and requirements, it was not considered feasible (or desirable) to create new physics codes from scratch solely for SWIM's needs. Moreover, the development of many of these codes continues under a variety of other projects, and SWIM needs to be able to take advantage of these outside developments quickly and easily, so "forking" off specialized versions of the codes was also not desirable. These factors led to the requirement that the IPS be able to support the coupling of physics components while absolutely minimizing the changes necessary to the underlying physics codes.

A simplifying aspect of SWIM's requirements is that the couplings between physics components are relatively loose—data exchanges are neither particularly frequent (typically once per time step, with each step requiring seconds to hours of wall clock time) nor particularly voluminous (typically tens of megabytes). This permits a file-based data transfer solution. However the level of flexibility desired to interchange physics components suggested the need for a centralized, standardized data store in order to avoid every component needing to know the unique data formats of every component with which it might need to exchange data.

These two requirements suggested an overall architecture in which SWIM components, are generally comprised of small wrappers around unmodified physics executables which, with the aid of small "helper" executables, adapt method invocations and data between the executable's native format and that used by the IPS framework. The "plasma state" [15], which was already in use in several fusion codes, was adopted and extended to define a standard format and location for data exchanged by components (data produced by a component that is *not* needed by others is left in the original form output by the physics code that produced it). The plasma state is defined on top of the netCDF self-describing file format [16].

Since the SWIM project is targeting large-scale simulations of fusion plasmas, the IPS must support computer platforms ranging from laptops and desktops (for development and testing) to Linux clusters to high-end "leadership class" systems, such as the Cray XT, XE, and XK lines, and the IBM Blue Gene line. (This requirement has been only partially met, as will be explained below.) Further, since these are typically batch-scheduled resources, the framework must be able to carry out a simulation involving many distinct invocations of the physics executables underlying the components in a single batch submission in order to avoid waits in the batch queue between each step of the simulation.

A final set of requirements comes from the perspective of the users of the framework. In the spirit of enabling rapid and facile experimentation, it is important the the IPS be easily used by domain scientists to create the coupled simulation applications they want, without having to read voluminous documentation or rely on the framework's developers to do it for them. The user's primary interactions with the framework relate to (a) the componentization of physics codes, to make them usable in the framework, (b) writing of the driver component, which uses the framework's services

to execute the desired simulation workflow, and (c) writing the configuration file describing the components of the simulation, data files that are used and exchanged and the execution characteristics of the tasks.

Based on earlier small-scale coupling experiments [17], we wanted the IPS to provide a simplified version of the Common Component Architecture (CCA) [18], retaining the key concepts, while simplifying the implementation by eliminating more sophisticated CCA features not required for the narrower scope of the IPS. We also followed the approach of the Earth System Modeling Framework (EMSF) [2] in defining a component interface which contains only a small number of functions and is uniform across all classes of physics, based on the operations `init()`, `step()` (advance a time step), and `finalize()`. As the project has evolved, additional operations, such as `checkpoint()` and `restart()` have been added, but the overall interface has been kept minimal. Experience has shown that a simple uniform interface makes it relatively easy for domain scientists to reason about both how to adapt their codes to work in the IPS environment, and how to use them in the context of writing a simulation driver.

Finally, users wanted to be able to develop drivers for their simulations and component wrappers for physics codes in a fashion that is not too far removed from procedural Fortran which still dominates the plasma physics community. This translates primarily into a requirement to support a full-featured programming language for these aspects of the system, rather than, for example, a workflow tool such as Kepler [19].

III. Design of the Integrated Plasma Simulator

Based on the requirements and some of the high-level design decisions described above, the IPS was designed to provide the components in the framework a modest set of services related to the management of computational tasks, management of the data files used by the component (and the framework), and a configuration management capability that allows an entire simulation to be described in a single input file. Resource management capabilities and an event service were added as the framework, and its uses, evolved. Here we summarize the design, which has been previously described in greater detail [20].

Fig. 1 schematically illustrates the services that comprise the IPS framework itself, a number of service components (DAOKTA Bridge, FTB Bridge, Portal Bridge, and Monitor), and a number of physics components, including the Driver, as they would typically be deployed on a high-end parallel computer. In the current design, components are spawned as separate processes,[1] which run on a head node, together with the framework itself. As mentioned earlier, components adapt the underlying physics executable to the IPS environment, which is almost negligible in a computational sense, and then invoke the physics code. Those executions, which comprise the computationally intensive portion of the simulation, are launched onto the system's compute nodes using appropriate mechanisms for the host system, such as `mpiexec` or `aprun` (Cray), just as any other HPC application would use the system.[2]

The **task manager** (TM) provides both services for inter-component method invocations and for the components themselves to launch and manage tasks. The TM API provides both blocking and non-blocking invocations, which allow both multiple tasks per component and multiple components within a simulation to execute concurrently. This permits IPS simulations to exploit three distinct levels of parallelism: individual computational tasks (the physics executables) can be parallel, a component can launch multiple tasks (for example to parallelize over flux surfaces in a plasma), and components can execute concurrently (where data dependencies permit) [21]. Using these same mechanisms, the framework also allows multiple simulations to be carried out concurrently in a single IPS invocation, sharing a common pool of compute nodes, thus providing a fourth level of parallelism [22].

The framework's **resource manager** (RM) tracks the utilization of the pool of compute nodes allocated to the batch job. It responds to resource requests from the TM using a simple first-come first-served (FCFS) algorithm with a first-fit backfill policy [23]. The TM also provides a *task pool* API that allows sets of tasks that do not have data dependencies among them to be specified and processed regardless of order [24]. So far, we have chosen to place the responsibility for handling dependencies on the domain scientist who writes the driver component for the simulation rather than providing elaborate capabilities in the framework to manage them. They understand the dependencies well, and have no problems coding them appropriately in the driver.

The **data manager** (DM) provides services that allow the components to conveniently specify and manage their input and output data staging requirements, as well as the overall simulation's archival needs. Although strictly speaking, the plasma state is distinct from the IPS, for convenience, the DM also mediates shared access to the plasma state through a mechanism of snapshotting the master plasma

[1] A thread-based rather than process-based design was considered, but we felt that thread-safety issues might cause problems for domain scientists not experienced in threaded programming in writing simulation drivers and component wrappers.

[2] We should observe, however, that while this approach works well on clusters and HPC systems like the Cray, it is not compatible with current IBM Blue Gene/L and Blue Gene/P systems because they do not provide enough flexibility to have the many different processes and physics executable invocations in a single batch job. Since this has not been a problem for the SWIM project, and the trend in HPC systems (including Blue Gene) is to support richer capabilities on the compute nodes, we made a strategic decision to defer addressing this issue until it became a true limitation on the IPS's abilities to support science.

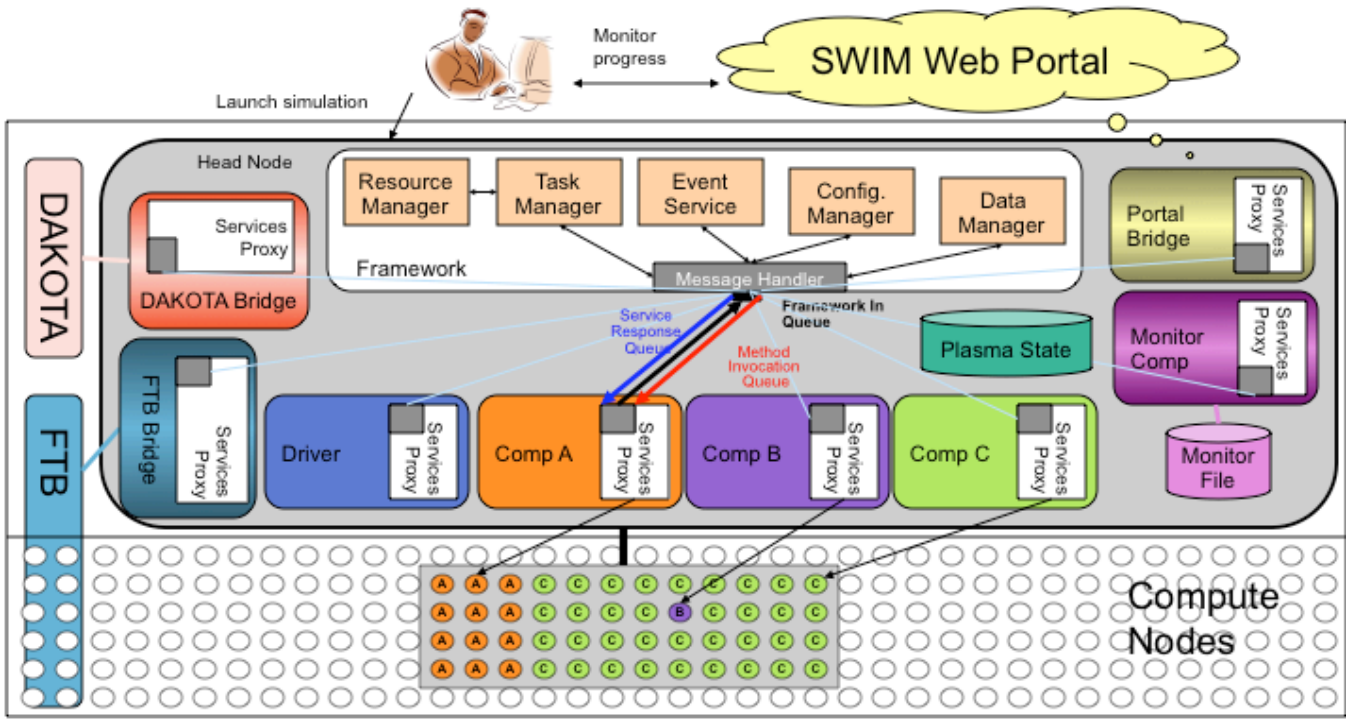


Figure 1. An illustration of the relationships between the framework, components, services, and tasks in the Integrated Plasma Simulator (IPS).

state, modifying the snapshot, and then merging changes back into the master.

The **configuration manager** (CM) allows a common input file to configure an entire multi-component simulation. The CM reads the input files when the IPS starts up and then provides a simple database capability which components can query for settings that effect them.

The **event service** (ES) allows communications among components through a simple publish–subscribe mechanism. The original purpose of the ES was to make fault-related information from the Fault Tolerance Backplane (FTB) [25] available within the IPS as part of research on fault tolerance in component-based multiphysics simulations [26]. The previously mentioned FTB Bridge component is a simple connector between the two event services, subscribing to FTB events and re-transmitting them to subscribers within the IPS, and vice versa. The IPS event service is also used to provide information about the progress of the simulation to an external web-based monitoring portal that allows IPS users to track the progress of their simulations in real time. Various methods within the framework have been instrumented with `send_portal_event()` calls which use the ES to send events (such as the invocation or termination of a component) to the Portal Bridge component, which in turn sends them to the web portal using a simple HTTP-based API. The portal can also display summaries of key simulation data which are extracted by the Monitor Component as the run progresses. The event service is also being used directly by some types of simulations, which can benefit from the asynchronous communication mechanism it provides (see for example, Sec. V-B and V-D).

A recent and noteworthy addition to the capabilities of the IPS is an interface with the DAKOTA engineering optimization and uncertainty analysis framework [27]. DAKOTA itself is a sophisticated framework designed to drive an external application through a sequence of invocations with different inputs, dictated by the optimization or analysis strategy. While DAKOTA is able to manage multiple concurrent invocations of the application, its capabilities in this regard are quite limited compared to the IPS. In order to take advantage of the best features of both packages, we run DAKOTA and the IPS concurrently, with DAKOTA configured to use the maximum concurrency permitted by the optimization algorithm being used. The IPS's DAKOTA Bridge component listens for application invocation requests from DAKOTA. Internally, it launches the simulations, which can then take advantage of all four levels of concurrency supported by the IPS within a single batch allocation, and returns the appropriate results to DAKOTA when they complete.

IV. Implementation of the Integrated Plasma Simulator

The requirements and design of the IPS, described above, require a great deal of versatility in the implementation language. On the one hand, we wanted to provide a fairly

high-level style of programming, utilizing component and object-oriented programming concepts. On the other hand, we needed the ability to easily interact with the system to launch and manage computational tasks, manipulate files, and even interact with remote systems via web protocols. Although it was not strictly necessary that the drivers and component wrappers be in the same language as the framework, we also wanted those aspects of the environment to be readily accessible to domain scientists.

We felt that the "traditional HPC languages" (Fortran, C, and C++) lacked the system interaction capabilities, and in the case of Fortran and C, they additionally lacked the high-level language features we wanted for the core of the framework. Although Java is quite popular in general, it is much less so in the HPC community, and typically is not supported on high-end systems. Python, on the other hand, satisfied our requirements handily: it is popular in the HPC community, and has been ported to high-end systems. Further, we could leverage the extensive ecosystem of modules in the Python standard library and available from third parties to simplify and accelerate development. Finally, although most of the domain scientists on our project did not know it initially, they were willing to learn it and found it easy to pick up. Some of them now use it routinely for other scripting needs, outside of the IPS.

At present, the IPS framework comprises 6967 lines of Python code (including 1998 lines of comments and documentation), and leverages a number of modules for key functionality. Component wrappers are typically around 100 to 300 lines of actual code, though in many cases users prefer to write the data adapter helper codes in Fortran because they know precisely how to read/write the files produced by the underlying physics codes.

The **multiprocessing** module [28] plays a major role at the heart of the IPS framework. It provides several classes for managing multiple processes (e.g., `Process`, `Pool`), shared data (e.g., `Variable`, `Array`), communication (e.g., `Queue`, `Pipe`) and synchronization (e.g., `Lock`, `Manager`) on local and remote machines. Unlike the *threading* module, it provides true concurrency because the processes do not share a single Global Interpreter Lock [29]. Multiprocessing processes use the fork/join model: a process is created by `p = Process(<callable object>, *args)`, started using the start method `p.start()`, and joined `p.join()`. Objects for communication can be passed as arguments to processes and used by any process that has access to the object. `Queue()` returns a shared queue object that any process can `put()` or `get()` objects on, where the objects are serialized and deserialized (pickled) by the `Queue` object.

In the IPS, the configuration manager creates a `Process` for each component in each simulation passing it the component implementation as specified in the configuration file, and set of three `Queue` objects that allow the component to communicate with the framework throughout the life of the component. Each component process receives a shared queue object that allows it to send *service request* messages to the framework, a queue on which the component receives *method invocation* requests from the framework, and a queue on which the component receives *service response* messages, as depicted in Figure 1. Framework services are made available to the components through a component-side `ServiceProxy` object, which handles requests for framework services in collaboration with the central framework process. Service requests that require processing by the central framework process (e.g., `launch_task()`, `wait_call()`) are transmitted to the framework via the shared framework service request queue. This queue is shared by all components in all simulations and automatically orders all requests to the framework, eliminating complex logic for ordering and load balancing of service requests. Lastly, each component has a method invocation queue that the framework uses to invoke a method on the component. Since only one method of a component's exposed interface can be active at any given time, this queue simply ensures that invocations are ordered and at most one is active at a time.

The IPS uses the standard Python **logging** module [30] to implement a flexible, distributed logging service. The logging facility in Python streamlines the logging process, allowing for sophisticated control over both the log format and content, as well as the mechanics of collecting and archiving of logs. The module allows for fine control over the verbosity of generated logs by assigning a *severity level* to each log message (such as `DEBUG`, `INFO`, `WARNING`, `ERROR`, and `CRITICAL`) and allowing the user to control the level of messages allowed into the log. It also provides a set of *handlers* that control the mechanisms through which messages that make the cut are treated. In the IPS, we make use of two such handlers, `logging.FileHandler` which writes the log directly to a local file, and `logging.handlers.SocketHandler` which transmits log messages using a network protocol for processing by a separate remote process.

The IPS framework itself generates a log file that contains information pertaining to the internal operation of the framework, and is typically used only to debug the operation of the framework itself. This is done using a `FileHandler` logger that is accessed directly by the framework class. In addition, each simulation that is run within an IPS framework instance transmits log messages from all constituent component processes to a remote logging *daemon* process which is started by the framework. Remote communication with the logging daemon is done by extending the `logging.handlers.SocketHandler` class to use local Unix domain sockets. The IPS logging daemon uses an instance of the class `LogRecordSocketReceiver` which extends the

`SocketServer.ThreadingUnixStreamServer` class, from the standard library `SocketServer` module, to implement a multithreaded approach to log message processing, allowing for the concurrent logging of messages from multiple simulations running within the IPS.

The IPS has several requirements for managing configuration files. First, we want to be able to factor the inputs into two separate files, one for platform-specific parameters, and the other for parameters controlling the simulation itself. The information in these files needs to be merged appropriately, and made available for queries by both the framework and by individual components. Moreover it needs to be easy for domain scientists to extend the configuration file with new parameters in order to accommodate new features they wish to control in the components or underlying physics codes. After evaluating a number of modules, we settled on the open source, third-party **ConfigObj** [31] package as the most appropriate solution to our needs. ConfigObj reads and writes configuration files in a Windows INI-like style, with some very useful extensions. In addition to sections, lists, and key-value pairs, the module supports validation, string substitution, multi-line values, comments, and full unicode support. The configuration file is then represented in the program as a dictionary of key-value pairs, with sections represented as nested dictionaries. Thus their use is readily understood by most Python programmers, and easily mimicked from examples by those with less experience. While in most cases, the IPS only *reads* configuration files, the DAKOTA bridge, described above, uses it to *write* configuration files that incorporate the parameters selected by DAKOTA for each simulation run.

Beyond the IPS framework itself, many of the components within the IPS, including drivers and service components, such as the Monitor, take advantage of a number of other Python modules to simplify their implementation. **NumPy** [32] is a Python module that provides the basic building blocks of numerical libraries, such as N-dimensional arrays, linear algebra routines, and random number generators. **SciPy** [33] builds upon the features of NumPy to provide a collection of mathematical algorithms for optimization, interpolation, integration, signal processing, sparse matrix operations, statistics and more. Used together, they form a computing environment similar to Matlab [34]. Additionally, the **NetCDF4** [35] module provides a Python API for reading and writing the netCDF files used by the plasma state.[3] NumPy data structures are used for reading and writing values in the plasma state. Additionally, these modules allow domain scientists to do more of the necessary, yet modest, mathematical manipulations of simulation data (e.g., convergence tests) in Python, thus avoiding the need to "revert" to Fortran or another language to express them.

Matplotlib [36], which is also built upon NumPy and SciPy provides a very flexible 2D graphing capability that is once again similar to that provided by Matlab. Physicists in the SWIM project have used Matplotlib to visualize plasma state values over the course of a simulation and also in a utility to compare these values across time steps and across runs. It is now an integral part of any SWIM IPS application and has facilitated the understanding of simulation results [37]. The IPS development team has also made extensive use of Matplotlib to graph resource utilization over time of both real IPS runs and simulations done in the Resource Usage Simulator (RUS), which is designed to simulate the IPS under different conditions [22] (e.g., Fig. 3(b)).

Finally, as the user base of the IPS has grown, so too has the need for documentation for both users and developers of the framework. For this purpose, we have turned to **Sphinx** [38], a third-party tool which is used to manage the documentation for Python itself. Key features for us include the ability to do both developer- and user-oriented documentation with the same tool, and produce good quality output in a variety of formats (particularly HTML and PDF).

V. IPS Applications

In this section, we briefly describe a number of different types of simulations for which the IPS is being used in production and exploratory contexts.

A. ITER Plasma Discharges

The ITER experimental fusion reactor [8] is an international effort to build a new large fusion device, which is designed to produce several hundred megawatts of fusion energy. Although construction of the project has barely begun, simulation of anticipated ITER plasma discharges already plays a very important role, both for design details of the device, and for planning of the scientific research program that will use it.

Current integrated modeling approaches for ITER typically require 1–1.5 months per run with moderate fidelity models running serially using the TSC/PTRANSP code [39]. Many such runs are required for parameter sweep and optimization studies. The SWIM project has demonstrated the capability to carry out much higher fidelity simulations while producing results much faster than the "standard" tools. Using the IPS, we have integrated components based on the TSC/GLF23 transport code [40], the NUBEAM neutral beam code [41], and the TORIC RF heating code [42]. Coupled simulations which exploit only the parallelism of the NUBEAM and TORIC codes can be completed in approximately 28 hours on 16 cores on the Cray XT4 at the National Energy Research Supercomputer Center (NERSC). Adding a second level of concurrency by running components simultaneously, where data dependencies permit, the time drops to 12 hours utilizing 24 cores. Further, the IPS

[3]Originally, we used the **Scientific.IO.netCDF**, but as it is no longer supported, we are migrating to the newer NetCDF4 module, starting with the Monitor component.

supports executing multiple distinct simulations concurrently in a single invocation of the framework. For example, a parameter study of *nine* of the aforementioned simulations were run in 16 hours on 128 cores to explore the location and height of the "pedestal" behavior at the edge of the plasma.

B. RF Control of MHD Stability

Another area of interest in SWIM is the use of RF waves to control instabilities that can lead to disruptions of the plasma. Simulations of so-called "slow MHD" phenomena of this type are being carried out in the IPS by integrating the NIMROD nonlinear magnetohydrodynamics code [43] and the GENRAY RF modeling code [44]. NIMROD is a very long-running code which, because of the time scales of the physics phenomena, needs updates from GENRAY only occasionally. Since NIMROD can tolerate small delays in incorporating the updated RF results, it is most efficient for NIMROD to run continuously rather than stopping it to run GENRAY and then restarting NIMROD. This type of simulation has been implemented in the IPS by using the framework's event service to signal GENRAY to begin an update, and when the results are available for NIMROD to incorporate.

C. Implicit Coupling with Discrete Components

One of the potential disadvantages of the loose explicit coupling approach taken by SWIM and the IPS is that there may be cases where couplings are highly non-linear and may require undesirably small time steps if they are to be explicitly coupled. However the usual approach to implicit coupling would involve extensive and pervasive changes to the relevant components. Following Sidi [45], SWIM project members Fred Jaeger, Ed D'Azevedo, John Wright, and Aaron Bader are using the IPS and existing physics components to implement an algorithm based on minimum polynomial expansion (MPE) and reduced rank extrapolation methods to couple the full-wave RF solvers AORSA [46] and TORIC [42] to the CQL3D Fokker-Planck solver [47]. The procedure is conceptually like a centered Crank-Nicholson method that can be implemented using quantities already computed by the existing codes, together with a Krylov subspace method for non-linear systems. The only computations beyond the explicit version of the coupling are an averaging operator, and a least squares solver. Fig. 2 illustrates the application of the method to approximately 0.165 seconds of a plasma discharge on the Alcator C-Mod experiment. The explicitly coupled solution required 125 time steps, and 8 hours of wall clock time on 1296 processors, while the implicit coupling scheme required just 5 time steps and completed in 2.3 hours [48].

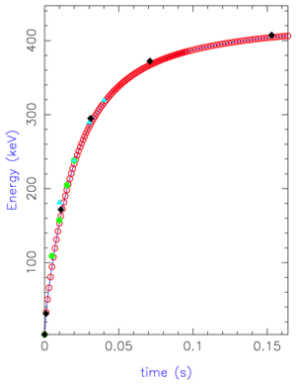


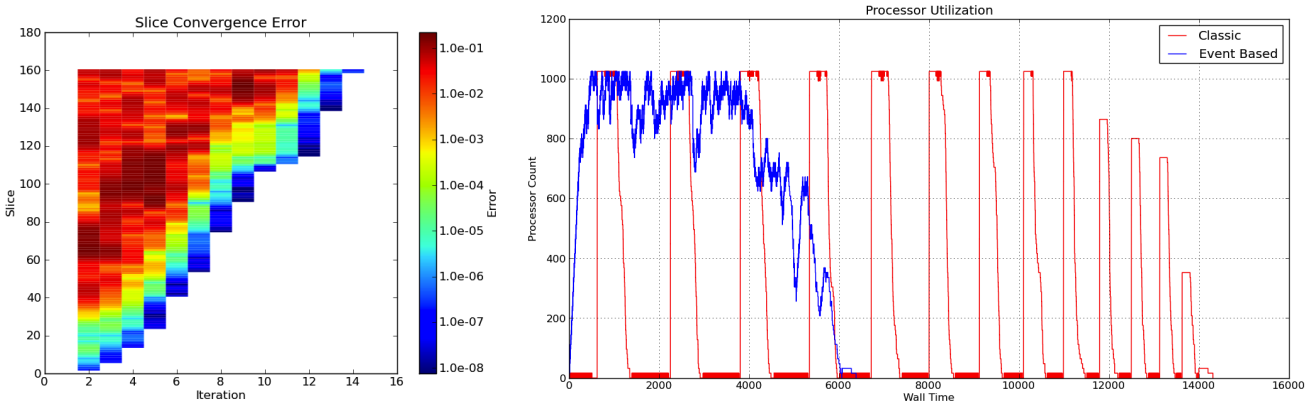
Figure 2. "Standard" direct coupling (red circles) versus MPE implicit coupling (black diamonds) produce the results with similar accuracy but significantly reduced run time.

D. Parallel-in-Time Algorithms

Parallelization of the temporal dimension of time-dependent partial differential equation based problems is of significant interest as computational scientists strive to uncover new parallelism in their applications in order to better utilize the rapidly increasing numbers of processors on high-end parallel systems. SWIM researchers are collaborating with colleagues at the University of Alaska, ITER, and University Carlos III de Madrid to explore the parareal algorithm [49] in plasma physics and other domains. As depicted in Fig. 3(a), the parareal algorithm iteratively refines approximate solutions to all of the time slices of interest simultaneously (vertical axis), with one or more slices converging to the final value at each iteration of the algorithm (horizontal axis). The capabilities of the IPS to support multi-level parallelism and the framework's event service inspired a novel parareal implementation [50], [51] which is driven by data dependencies of the computations rather than the simple loops in the traditional form of the algorithm. The components for the coarse and fine solvers maintain separate pools of ready-to-run tasks, using the framework's task management capabilities to manage the concurrent execution of these tasks, and the event service to signal completion of tasks, which in turn satisfy dependencies. This increases the amount of parallelism available at any point, thus improving resource utilization compared to the traditional parareal implementation, as illustrated in Fig. 3(b).

E. Battery Simulation

The IPS framework is currently being used as the basis for multi-physics simulations aimed at evaluating the safety and enhancing the performance of lithium-ion batteries. The Computer Aided Design of BATteries (CAEBAT) project, funded by the U.S. Dept. of Energy, works toward developing a modeling capability that integrates multi-scale models of sub-components of a battery pack (battery pack, cell, electrodes) in a unified simulation environment. This work involves studying the impact of phenomena such as charge and thermal transport; electrochemical reactions; mechanical stresses across the porous 3D structure of the electrodes (cathodes and anodes) and the solid or liquid electrolyte system on both the performance and safety of lithium-ion batteries. The loose coupling file-based approach of the IPS



(a) Parareal: Convergence

(b) Parareal: Resource Utilization

Figure 3. (a) Heat map representation of convergence rate for a parareal problem. The vertical axis represents the 160 time slices into which the time domain has been decomposed. The horizontal axis represents successive iterations of the parareal algorithm. Colors indicate distance from convergence, with red representing the largest errors, and blue representing converged results. (b) Resource utilization graph of classic parareal (red) vs. event-based (blue) parareal the same 160-slice problem on 1024 cores.

facilitates the integration of proprietary sub-models from industrial partners in the CAEBAT project into a coupled simulation, without undue hurdles stemming from the need to protect intellectual property in such models. The ability to use the DAKOTA optimization toolkit to drive IPS simulations is crucial in this effort, as the work is fundamentally a constrained optimization problem that aims to maximize the performance of a battery pack, while keeping it operating within an acceptable cost and safety envelope.

VI. Conclusions

In this paper we have presented the IPS as a flexible, powerful, and expressive framework for loosely-coupled multiphysics on high performance computers. It has been shown to work for a variety of applications and execution models, from time-stepped simulations to event-based data flow problems to parameter sweeps driven by an external optimization package. The IPS component concept and framework's core capabilities of resource, task, data and configuration management, along with the event service provide multiple levels of parallelism, file-based data coupling, and multiple modes of simulation composition and orchestration. In addition to these core capabilities, bridge components provide optional services for visualization, job monitoring, external application drivers and fault tolerance to support the scientific studies being done by our users. Many of the capabilities implemented in the IPS for one purpose have found additional, and very different uses in other types of problems.

Much of the design and implementation was influenced by features of the Python language, including its built-in and easy to use data structures, system interaction utilities and object-oriented design, and the wide range of external modules that encapsulate powerful tools in easy to use APIs. This allowed us to provide the capabilities of the IPS in a small amount of code that is easily maintained and extended.

Looking forward, we plan to continue using and developing the IPS for fusion simulation and battery simulation, and are seeking users in additional domains. One area of interest for fusion is integration of multiple frameworks to support more complex simulations with both tightly- and loosely-coupled aspects. These ideas have recently been prototyped by encapsulating the FACETS tight-coupling framework [52] as an IPS component, allowing a high-fidelity core/edge models in FACETS to couple to existing IPS components. We are also interested in extending the IPS to support higher performance approaches in which data can be exchanged in memory rather than via files, and the computational core of components are directly linked into the Python framework rather than being invoked as a separate executable.

VII. Code Availability

We have plans to migrate the IPS code base to a public repository in the near future. In the meantime, we encourage interested parties to contact the authors for access to the repository.

Acknowledgments

This work has been supported by the U. S. Department of Energy, Office of Science, Offices of Advanced Scientific Computing Research (ASCR) and Fusion Energy Sciences (FES). It has also been supported by by the ORNL Postmasters and Postdoctoral Research Participation Programs and the ORNL Higher Eduction Research Experiences Program which are sponsored by ORNL and administered jointly by ORNL and by the Oak Ridge Institute for Science and Education (ORISE). ORNL is managed by UT-Battelle, LLC for the U. S. Department of Energy under Contract No. DE-AC05-00OR22725. ORISE is managed by Oak

Ridge Associated Universities for the U. S. Department of Energy under Contract No. DE-AC05-00OR22750.

REFERENCES

[1] A. P. Craig, R. Jacob, B. Kauffman, T. Bettge, J. Larson, E. Ong, C. Ding, and Y. He, "CPL6: The New Extensible, High Performance Parallel Coupler for the Community Climate System Model," *International Journal of High Performance Computing Applications*, vol. 19, no. 3, pp. 309–327, 2005.

[2] N. Collins, G. Theurich, C. DeLuca, M. Suarez, A. Trayanov, V. Balaji, P. Li, W. Yang, C. Hill, and A. da Silva, "Design and implementation of components in the Earth System Modeling Framework," *International Journal of High Performance Computing Applications*, vol. 19, no. 3, pp. 341–350, 2005.

[3] S. Buis, A. Piacentini, D. Déclat, and the PALM Group, "PALM: a Computational Framework for Assembling High-Performance Computing Applications," *Concurrency and Computation: Practice and Experience*, vol. 18, pp. 231–245, 2006.

[4] S. Valcke, E. Guilyardi, and C. Larsson, "PRISM and ENES: A European approach to Earth system modeling," *Concurrency and Computation: Practice and Experience*, vol. 18, pp. 247–262, 2006.

[5] X. Jiao, G. Zheng, P. A. Alexander, M. T. Campbell, O. S. Lawlor, J. Norris, A. Haselbacher, and M. T. Heath, "A system integration framework for coupled multiphysics simulations," *Eng. with Comput.*, vol. 22, no. 3, pp. 293–309, 2006.

[6] C. R. Johnson, S. G. Parker, and D. M. Weinstein, "Component-Based Problem Solving Environments for Large-Scale Scientific Computing," *Concurrency and Computation: Practice and Experience*, vol. 14, pp. 1337–1349, 2002.

[7] T. Goodale, G. Allen, G. Lanfermann, J. Massó, T. Radke, E. Seidel, and J. Shalf, "The Cactus framework and toolkit: Design and applications," in *Vector and Parallel Processing - VECPAR '2002, 5th International Conference*. Springer, 2003.

[8] "ITER: The way to new energy," http://www.iter.org.

[9] J. Dahlburg, J. Corones, D. Batchelor, R. Bramley, M. Greenwald, S. Jardin, S. Krasheninnikov, A. Laub, J.-N. Leboeuf, J. Lindl, W. Lokke, M. Rosenbluth, D. Ross, and D. Schnack, "Fusion Simulation Project: Integrated Simulation and Optimization of Fusion Systems," *J. Fusion Energy*, vol. 20, no. 4, pp. 135–196, December 2001.

[10] "Center for Plasma Edge Simulation," http://www.cims.nyu.edu/cpes/.

[11] "Framework for Core-Edge Transport Simulations," http://facetsproject.org.

[12] "Center for Simulation of RF Wave Interactions with Magnetohydrodynamics," http://cswim.org/.

[13] "EFDA Task Force on Integrated Tokamak Modelling," http://www.efda-taskforce-itm.org.

[14] "TASK Code Home Page," http://bpsi.nucleng.kyoto-u.ac.jp/task/.

[15] D. McCune, "Plasma State," http://w3.pppl.gov/ntcc/PlasmaState/.

[16] Unidata Program Center, "NetCDF (Network Common Data Format)," http://www.unidata.ucar.edu/software/netcdf/.

[17] W. R. Elwasif, D. B. Batchelor, D. E. Bernholdt, L. A. Berry, E. F. D'Azevedo, W. A. Houlberg, E. F. Jaeger, J. A. Kohl, and S. Li, "Coupled fusion simulation using the Common Component Architecture," in *Computational Science – ICCS 2005 5th International Conference, Atlanta, USA, May 22–25, 2005, Proceedings, Part I*, ser. Lecture Notes in Computer Science, V. Sunderam, G. D. van Albada, P. M. A. Sloot, and J. J. Dongarra, Eds., vol. 3514. Atlanta, Georgia, USA: Springer, 22–25 May 2005, pp. 372–379.

[18] B. A. Allan, R. Armstrong, D. E. Bernholdt, F. Bertrand, K. Chiu, T. L. Dahlgren, K. Damevski, W. R. Elwasif, T. G. 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," *Intl. J. High-Perf. Computing Appl.*, vol. 20, no. 2, pp. 163–202, Summer 2006.

[19] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger-Frank, M. Jones, E. Lee, J. Tao, and Y. Zhao, "Scientific Workflow Management and the Kepler System," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1039–1065, 2006.

[20] 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.

[21] 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, 16–17 October 2008.

[22] S. Foley, W. Elwasif, D. Bernholdt, A. Shet, and R. Bramley, "Many-task applications in the Integrated Plasma Simulator," in *3rd IEEE Workshop on Many-Task Computing on Grids and Supercomputers, MTAGS 2010*, Nov 2010.

[23] D. Lifka, "The ANL/IBM SP scheduling system," in *Job Scheduling Strategies for Parallel Processing.*, ser. IPPS'95 Workshop Proceedings. Berlin: Springer-Verlag, 1995, pp. 295–303.

[24] 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 *ACS/IEEE International Conference on Computer Systems and Applications*, 2011.

[25] R. Gupta, P. Beckman, H. Park, E. Lusk, P. Hargrove, A. Geist, D. K. Panda, A. Lumsdaine, and J. Dongarra, "CIFTS: A Coordinated Infrastructure for Fault-Tolerant Systems," in *Proceedings of 38th International Conference on Parallel Processing (ICPP) 2009*, September 2009.

[26] A. G. Shet, W. R. Elwasif, S. S. Foley, B. H. Park, D. E. Bernholdt, and R. Bramley, "Strategies for Fault Tolerance in Multicomponent Applications," in *Proceedings of the International Conference on Computational Science, ICCS 2011*, vol. 4, 2011, pp. 2287 – 2296.

[27] B. Adams, W. Bohnhoff, K. Dalbey, J. Eddy, M. Eldred, D. Gay, K. Haskell, P. Hough, and L. Swiler, "DAKOTA, A Multilevel Parallel Object-Oriented Framework for Design Optimization, Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis: Version 5.0 User's Manual," Sandia National Laboratories, Tech. Rep. SAND2010-2183, December 2009.

[28] "Multiprocessing - Process based "Threading" Interface," http://docs.python.org/library/multiprocessing.html.

[29] "Thread State and the Global Interpreter Lock," http://docs.python.org/c-api/init.html#threads.

[30] "Logging facility for python," http://docs.python.org/library/logging.html.

[31] M. Foord and N. Larosa, "ConfigObj 4.2.7," http://www.voidspace.org.uk/python/configobj.html, February 2010.

[32] "NumPy," http://numpy.scipy.org/.

[33] "SciPy," http://www.scipy.org/.

[34] MathWorks, "MATLAB - The Language Of Technical Computing," http://www.mathworks.com/products/matlab/index.html.

[35] J. Whitaker, "Python Module NetCDF4 (0.9.7)," http://code.google.com/p/netcdf4-python/, August 2011.

[36] J. Hunter *et al.*, "Matplotlib," http://matplotlib.sourceforge.net/.

[37] D. Batchelor, G. Alba, E. D'Azevedo, G. Bateman, D. Bernholdt, L. Berry, P. Bonoli, R. Bramley, J. Breslau, M. Chance, J. Chen, M. Choi, W. Elwasif, S. Foley, G. Fu, R. Harvey, E. Jaeger, S. Jardin, T. Jenkins, D. Keyes, S. Klasky, S. Kruger, L. Ku, V. Lynch, D. McCune, J. Ramos, D. Schissel, D. Schnack, and J. Wright, "Advances in simulation of wave interations with extended MHD phenomena," in *SciDAC 2009, 14–18 June 2009, California, USA*, ser. Journal of Physics: Conference Series, H. Simon, Ed. Institute of Physics, 2009.

[38] G. Brandl and the Pocoo Team, "Sphinx," http://sphinx.pocoo.org/.

[39] S. Jardin, M. Bell, and N. Pomphrey, "TSC simulation of Ohmic discharges in TFTR," *Nucl. Fusion*, vol. 33, no. 3, pp. 371–382, March 1993.

[40] S. C. Jardin, N. Pomphrey, and J. Delucia, "Dynamic Modeling of Transport and Positional Control of Tokamaks," *J. Computat. Phys.*, vol. 66, no. 2, pp. 481–507, 2 October 1986.

[41] A. Pankin, D. McCune, R. Andre, G. Bateman, and A. Kritz, "The tokamak Monte Carlo fast ion module NUBEAM in the National Transport Code Collaboration library," *Computer Phys. Comm.*, vol. 159, no. 3, pp. 157–184, 1 June 2004.

[42] J. C. Wright, P. T. Bonoli, M. Brambilla, F. Meo, E. D'Azevedo, D. B. Batchelor, E. F. Jaeger, L. A. Berry, C. K. Phillips, and A. Pletzer, "Full Wave Simulations of Fast Wave Mode Conversion and Lower Hybrid Wave Propagation in Tokamaks," *Phys. Plasmas*, vol. 11, pp. 2473–2479, 2004.

[43] C. R. Sovinec, A. H. Glasser, T. A. Gianakon, D. C. Barnes, R. A. Nebel, S. E. Kruger, D. D. Schnack, S. J. Plimpton, A. Tarditi, and M. S. Chu, "Nonlinear magnetohydrodynamics simulation using high-order finite elements," *J. Comput. Phys.*, vol. 195, no. 1, pp. 355–386, 2004.

[44] A. Smirnov, R. Harvey, and K. Kupfer, "A general ray tracing code GENRAY," *Bull Amer. Phys. Soc.*, vol. 39, no. 7, p. 1626, 1994.

[45] A. Sidi, "Efficient implementation of minimal polynomial and reduced rank extrapolation methods," *Journal of Computational and Applied Mathematics*, vol. 36, no. 3, pp. 305 – 337, 1991.

[46] E. F. Jaeger, L. A. Berry, E. F. D'Azevedo, D. B. Batchelor, M. D. Carter, K. F. White, and H. Weitzner, "Advances in Full-Wave Modeling of Radio Frequency Heated, Multidimensional Plasmas," *Physics of Plasmas*, vol. 9, no. 5, pp. 1873–1881, 2002.

[47] R. Harvey and M. McCoy, "The CQL3D Fokker-Planck code," in *Proceedings of the IAEA Technical Committee Meeting on Simulation and Modeling of Thermonuclear Plasmas*, Montreal, Canada, 1992.

[48] E. F. Jaeger, private communication, May 2011.

[49] J.-L. Lions, Y. Maday, and G. Turinici, "A"parareal" in time discretization of PDE's," in *Mathématique*, ser. 1, vol. 332, no. 7. Paris: Comptes rendus de l'Académie des sciences, 2001, pp. 661–668.

[50] L. A. Berry, W. R. Elwasif, J. M. Reynolds-Barredo, D. Samaddar, R. Sanchez, and D. E. Newman, "Event-based parareal: A data-flow based implementation of parareal," *Journal of Computational Physics*, 2011, submitted.

[51] W. R. Elwasif, S. S. Foley, D. E. Bernholdt, L. A. Berry, D. Samaddar, D. E. Newman, and R. Sanchez, "A Dependency-Driven Formulation of Parareal: Parallel-in-Time Solution of PDEs as a Many-Task Application," in *4th IEEE Workshop on Many-Task Computing on Grids and Supercomputers, MTAGS 2011*, November 2011, accepted.

[52] J. R. Cary, A. Hakim, M. Miah, S. Kruger, A. Pletzer, S. Shasharina, S. Vadlamani, A. Pankin, R. Cohen, T. Epperly, T. Rognlien, R. Groebner, S. Balay, L. McInnes, and H. Zhang, "FACETS – a Framework for Parallel Coupling of Fusion Components," in *18th Euromicro Int'l. Conf. on Parallel, Distributed and Network-based Processing (PDP)*, Pisa, Italy, 17–19 February 2010.