Introduction to IPS and Brief Overview

D. B. Batchelor

(Draft 1-26-2019)

Introduction to IPS

The Integrated Plasma Simulator (IPS) is a computational framework that permits serial and or parallel simulation codes to function inter-operably on the most advanced massively-parallel computers so as to provide a flexible capability for integrated, multi-physics simulation. It was developed under the SWIM project in the Scientific Advancement through Advanced Computation (SciDAC) program, jointly funded by the DOE offices of Fusion Energy and Advanced Computing Research. Continued development and maintenance of the IPS is presently (2019) supported under the AToM SciDAC project. The IPS has been extensively tested, is currently running on numerous computers around the world, and has demonstrated use of supercomputing resources in executing simulations comprised of a variety of different workflows. These range in complexity from simple time stepping simulations to complex event-driven workflows. IPS is implemented in framework/component architecture. It is not in any way restricted to plasma, or even to physics applications, and has found applications in several other fields, although many plasma physics related components are available for fusion applications. The IPS could reasonably be thought of as the Integrated ***Process*** Simulator.

IPS workflow components are typically independent codes, not originally developed with the intention of coupling to other components in an integrated way. The philosophy of IPS is to not require any modification of the component codes, in particular no modifications of the file I/O structure is required. The physics codes are 'wrapped' with interface coding (component-wrappers) that allows them to communicate and to operate with the IPS framework and other components.

The IPS framework itself is implemented in Python. It provides basic services needed by the components through a set of "managers". The Configuration Manager assembles and connects the needed components as specified in a simulation configuration file. The Task Manager manages the execution of the physics codes, which may be massively parallel. The Data Manager moves and archives the component input/output and simulation state files to make them available to the physics codes at each invocation. The Resource Manager efficiently manages access to compute resources for concurrently running processes. The Event Manager provides asynchronous publish/subscribe data exchange in a running simulation.

The IPS is designed primarily for use in a batch-processing environment, with a batch job typically comprising a single invocation of the framework, calling the individual component codes many times as the simulation progresses. IPS simulations are typically orchestrated by a "driver" component, although simulation-controlling logic can also be built into individual component-wrappers. The driver and component-wrapper interfaces are written in Python, and therefore can implement arbitrarily complex or conditional workflows – essentially anything that is programmable. The simulation configuration file (e.g., ips.config) specifies which components make up any given IPS workflow. The config file allows considerable flexibility in the location of data and codes, and in details of component code behavior. Evolving simulation data that must be communicated between components are contained in a set of files specified as *state* files, which are managed by the framework. The framework generates a directory structure for the workflow containing sub-directories for simulation results, restart data, progress logging, as well as individual work directories in which each component runs.

Some IPS details

Simulation State

In order to interact, codes must exchange data. In the IPS the exchange is accomplished by specifying a set of files, known as STATE files, containing data describing the current state of the simulation, which are moved between components by the framework before and after code executions. These files are specified in the simulation configuration file. The framework maintains a *state* work directory containing the current set of state files, which are updated after each component execution.

In general the output file of one component code will not be directly usable as input to another. It is a responsibility of the component-wrapper scripts to process the data in state files into the form need by that component. See discussion below. In several cases where all, or many, components require the same data, it has proved beneficial to develop a common data format and a single file which multiple components use. This reduces the number of conversion algorithms required and increases the probability that components are actually using the same data values. For fusion applications the SWIM project developed an extensive system called the “Plasma State” that supports a data structure containing core plasma data and system description data relevant to tokamak devices. The SWIM Plasma State system provides many useful functions such as multiple instances, loading, saving, interpolation, copying, and modification-tracking that greatly assist in maintaining consistency of data across components. It is supported but not required by the IPS.

Configuration file

The simulation configuration file is read by the IPS framework at startup and contains two kinds of parameters – global parameters that apply to the simulation as a whole and component specific parameters that are private to the individual components. The file specifies a set of “PORT”s, which point to the implementation of each component of the workflow. Each named PORT must have a section in the configuration file giving information about that PORT. Required global parameters include: IPS_ROOT, which is the path to the IPS framework itself, and NAMES, which is a list of abstract names of the components (or PORTS) to be used in the simulation. There are also required component parameters such as: SCRIPT, which is the path to the wrapper script that implements the component. The user is free to define other parameters, either global or component specific, although he/she is responsible for programming the retrieval and behavior of user defined parameters. Much more information about configuration files is available in [Integrated Plasma Simulator (IPS) Documentation, Release 2.1, October, 18, 2011].

Components

IPS component-wrapper python scripts provide the interface between the IPS framework and the component codes. Component-wrappers are python classes, which must inherit from the IPS class *Component*. Components are required to provide three functions, *init*, *step*, and *finalize*. Many also implement *checkpoint* and *restart* functions to use the IPS built-in checkpoint/restart capability. Users may also provide other component-wrapper functions, which for example drivers might use, but the framework makes no reference to them.

There are two distinguished PORTS called by the framework itself – INIT, and DRIVER. The workflow INIT performs any initializations needed before the components perform their individual *init* functions. The DRIVER calls the *init* functions of the other component-wrappers, starts the workflow execution, it usually provides the logic for the progression of the workflow by calling the *step* functions of the components, and it calls the *finalize* function for each component-wrapper at the end of the workflow. The purpose of the component-wrapper *init* functions is to do any work needed to prepare the component for the beginning of the simulation. In particular it must generate any initial data that goes into the state files and update the state work directory using framework services. The *finalize* functions can perform any needed final calculations or clean up any mess left behind, but most often do nothing.

Physics component-wrapper scripts have three primary responsibilities as the simulation progresses:

1) To generate standard input files for the implementing component codes. Input data for the codes most often is a combination of code specific data derived from standard template input files of the given code, evolving data derived from the simulation state files, and perhaps parameters obtained from the configuration file. This is accomplished by calling on framework services to move input files and simulation state files into the work directory, then merging the data to generate updated standard code input files. It may involve invoking other codes in the process.
2) To call on framework services to assemble the computer resources needed and launch the physics code execution using the appropriate parallel protocol for the platform in use.
3) To process the code output to the form appropriate for the STATE files, for example converting to a common data format if one is specified. It should then call on framework services to update the collection of state files containing new data, and to archive any other output files that are to be kept as results.

Component-wrapper scripts are free to provide other functions to be called by the driver, or to launch other physics codes needed to fulfill the functionality of the component. Component-wrappers can get and employ global or component specific configuration parameters. Some component-wrappers have been developed which have extensive internal logic or which react to signals directly from other component-wrappers sent via the framework publish/subscribe event service.

Other IPS features

The IPS provides many other functions and services. We describe just a few here.

Framework event tracking – The framework generates a unique, human readable (although long) run identifier for each IPS workflow execution, which allows some level of provenance tracking. The identifier is available to component-wrappers as a global configuration parameter, RUNID. The IPS produces an event log file identified by the RUNID that contains a record of every action the framework takes throughout the run. This data is valuable for following the progress of the workflow, debugging, and chasing down timing issues. Periodically and at the end of the run the event log file is processed to an html file that can easily be displayed in a web browser. These two files reside in the */simulation_log/* subdirectory. In addition the user can specify a directory, possibly outside the IPS generated workflow directory, where the html files are sent. This way html log files for all of the user's IPS runs can be collected for reference. This directory, called /www/, can be accessed directly by web browser, at least it can at NERSC.

Data Monitoring – An IPS component called MONITOR is available which serves two functions. For simulations that use the Plasma State, at the end of each step, it extracts and processes selected plasma data from the current plasma state file (which is a time snapshot) and aggregates the data from multiple steps into time series, which are written to a monitor netcdf file. The current monitor file is copied to the/www/ directory each step. For any simulation the user can specify in the configuration file one or more state files to be copied to the /www/ directory at the end of each step

Coupling with Dakota Framework

Nested workflows

Getting started with IPS

The place to start with IPS is to download the framework and some simple example simulations and just run them. There are three examples that are particularly appropriate for getting started – hello-world, ABC_example, and sequential_model_simulation.

Instructions for cloning the needed github repos can be found at https://github.com/ORNL-Fusion/ips-examples. To give an idea of what is involved, the process as of 4/2018 is repeated here, but it is subject to change so the instructions at github should be considered definitive..

Install the IPS

1) Create an IPS directory and clone the IPS-framework, wrappers, and examples repos.

```
mkdir IPS
cd IPS
git clone https://github.com/HPC-SimTools/IPS-framework.git ips-framework
git clone https://github.com/ORNL-Fusion/ips-wrappers.git
git clone https://github.com/ORNL-Fusion/ips-examples.git
```

2) Export the IPS_DIR environment variable

```
export IPS_DIR=${PWD}
```

3) Add this to your .bashrc (or otherwise so it's there next time you open a shell). Note: Adapt for csh or otherwise.

```
echo 'export IPS_DIR='${PWD} >> ~/.bashrc
```

Run an example (e.g. hello-world)

1) Source the environment

```
cd $IPS_DIR
source ips-wrappers/env.ips
```

2) Run the example

```
cd ips- ips-examples/hello-world
ips.py --simulation=ips.config --platform=platform.conf
```

hello-world example

Hello_world is (almost) the simplest possible IPS run, exercising two components DRIVER, hello_driver.py, and WORKER, hello_worker.py. We say "almost" because it would actually be possible to have an IPS simulation with only a driver and no components. Hello world does not use any STATE files or use any physics components. There is one external connection, to an environment file (env.ips) that is sourced from $IPS_DIR/ips-examples/env.ips in the present setup.

ABC_example

The ABC simulation is a somewhat more realistic example of IPS usage. It is completely written in python and has no dependencies other than IPS. As such it should run about anywhere, at least it runs on Macs and at NERSC. The input and output files are all human readable text. It largely has the structure of a real simulation in that it has three components that interact. The “simulation” solves a coupled set of 2 ODEs by bonehead step-by-step integration.

$$\dot{x} = Ax + Bxy$$
$$\dot{y} = Cy + Dxy$$

There are 3 “physics” codes: *X_dot_code.py, Y_dot_code.py*, and *integrator.py*. Component wrapper scripts drive these “physics” codes: *A_component.py, B_component.py*, and *C_component.py* respectively. In addition there are simulation initializer and driver components: *basic_init.py* and *basic_driver.py*. The *basic_init.py* and *basic_driver.py* components are very generic and might well be usable directly in other simple workflows.

The code is in the *_exec* directory and all of the input data is in the *_input* directory. Each code has a template input file and produces one output file. The STATE files consist of output files from the A and B components and a common state.dat file. It also will send the eventlog html file to a www directory. So it pretty much has the structure of a real simulation.

Typically a real simulation would be run as a batch job, submitted via a batch script. In trying to make this like a real simulation running as a batch job, a batch script for Edison and a shell script for Mac are provided. The exporting of IPS_DIR and sourcing of ips.env are included in these scripts so these two steps described above can be omitted. The command lines to run from the /ips-examples/ABC_example/ directory are:

On Edison – sbatch Edison_run

and on Mac – ./Mac_run

This example also demonstrates simulation restart which is invoked with the Edison_restart and Mac_restart scripts.

sequential_model_simulation

This model simulation is intended to look almost like a real simulation, short of requiring actual physics codes and input data. Instead typical simulation-like data is generated from simple analytic (physics-less) models for most of the quantities that are assembled into time series by the MONITOR component. It includes time stepping, time varying scalars and profiles, and checkpoint/restart. And importantly it gives a simple demonstration of coupling of components using the SWIM Plasma State framework. Here "sequential" refers to each component being run sequentially, as opposed to concurrently, which will be dealt with in another example.

The PORTS (i.e. components) that exercised are:

Simulation initialization = INIT generic_ps_init.py

Driver = DRIVER - generic_driver.py

Equilibrium and profile advance = EPA - model_epa_ps_file_init.py

Ion cyclotron heating = RF_IC - model_RF_IC_2_mcmd.py

Neutral beam heating = NB - model_NB_2_mcmd.py

Fusion heating and reaction products = FUS - model_FUS_2_mcmd.py

Simulation time history monitoring = MONITOR - monitor_comp.py

The driver and monitor components are full components as used in many of the real simulations. In this case the driver is generic_driver.py, which implements a simple time stepping workflow using any mixture of eight specific physics components. The other components are simple analytic models, with parameters that can be modified in the component input data files.

Other examples and workflows

There are a number of other examples available that use real physics codes. Also available are a number of simulation workflows and simulation configuration files which might fit the users needs with no modification, other than change of input parameters. However these will generally require access to prebuilt codes and wrappers that are not provided with the installation above.

Plasma physics, wrappers, and maintained installations

The hello-world and ABC_example are entirely Python and require no compilation or other build steps. However most of the applications of IPS involve significant physics codes, primarily plasma physics, which do require compilation and building. Up to now IPS has not attempted to distribute the physics codes themselves, but relies on code developers to maintain and distribute them. The IPS does maintain a collection of interface component-wrappers to allow many physics codes to be used in the IPS (i.e. the contents of the ORNL-Fusion/ips-wrappers.git repo cloned above). Many of the wrappers require compilation and building, so such component subdirectories contain makefiles. It is not always a trivial matter to build these wrappers, particularly ones that couple to the SWIM Plasma State which itself is not simple to build. To obviate the need for a user to go through this process, the IPS team maintains built IPS installations, along with collections of built physics codes at several computer centers. Such installations have been made at many locations around the world, although the principal ones active now are at NERSC, General Atomics, and ORNL. If a user wishes to use the existing IPS component base it is easiest to do it through one of the maintained installations, although other installations can be made, and we anticipate making it easier in the future.

For a green-field development in which the user has in hand the physics codes to be coupled, and there is no need for any of the compiled IPS components (some of the IPS components are Python and require no compilation), the IPS system cloned above can be used locally. The user will just need to provide the interface wrappers for his particular physics codes.

Developing new components

The required functions of component-wrapper scripts are all the same, as described above. As such much of the code is identical between the wrappers, particularly that needed for communicating with the IPS framework. The logic may differ between components but that is generally easy to program in Python. The biggest jobs are assembling the data in the proper format to produce the physics code's input file/files, and putting data from the code's output into the state files for use by other components.

There already exist a wide variety of driver components and physics components that can serve as templates for new development. Some of the drivers and initializers are quite generic and might be directly useable for new workflows. The advice is, before launching into a new component development, talk to someone on the IPS team who has done it. It is essentially never necessary to start from scratch.