Synergia: Driving Massively Parallel Particle Accelerator Simulations with Python

James Amundson, Qiming Lu, and Eric Stern
Fermi National Accelerator Laboratory
Batavia, IL 60510
Email: amundson@fnal.gov

***Abstract*—Simulations of beam dynamics in particle accelerators have a wide range of computational requirements. The simplest calculations involve independent-particle tracking of a few thousand particles which can easily be accomplished on modern desktop computers. Calculations involving collective effects may require millions or even billions of particles and push the limits of modern supercomputers. We describe Synergia, a hybrid Python/C++ accelerator simulation package capable of dealing with the entire spectrum of beam dynamics simulations. We describe the motivations for the hybrid language design and discuss the issues that arise in the implementation. We also describe the most novel feature of our code, a hybrid C++/Python object serialization system, in detail. The same techniques are of general use in any domain by providing data and computationally intensive C++ frameworks with the advanced programming and user-friendly features of Python.**

I. Introduction

Modern particle accelerators are complex devices, often containing thousands or even tens of thousands of components, propagating beams of typically $\mathcal{O}(10^{12-13})$ particles, usually divided into well-separated bunches. A detailed understanding of the dynamics of beams in these machines inevitably requires computer simulations. Fortunately, in most cases, the physics of beams at leading order is determined by the interactions between individual beam particles and the electromagnetic fields generated by the accelerator components. The simulation of the dynamics of independent particles in particle accelerators is a well-understood problem and is tractable on desktop computers, even for the most complicated accelerators.

In order perform simulations beyond leading order, however, collective effects involving the beam particles must be included. The most important collective effects include space charge, the electrostatic repulsion between the particles in a bunch, and wake fields, the effects of fields induced in the beam pipe by the leading particles on the trailing particles. These collective effects are not tractable by brute force calculations; the large number of particles and even larger number of pairwise interactions push the problem well beyond the reach of today's, or even tomorrow's, supercomputers. Instead, most accelerator simulations of collective effects employ particle-in-cell (PIC) methods in order to reduce the problem to the simulation of a reasonable number of macroparticles combined with field solves on finite grids.

II. Synergia

Synergia [1] is a hybrid Python/C++ package for accelerator simulations utilizing PIC methods. The current version (2.1), which we describe here, is the evolution of the original Synergia [2], a Fortran90/C++ program with a Python user interface. It combines independent-particle and collective effects through the split-operator technique. When the Hamiltonian for a system can be split into independent (i) and collective (c) components,

$$\mathcal{H} = \mathcal{H}_i + \mathcal{H}_c, \tag{1}$$

The split-operator approximation for the time step evolution operator, $\mathcal{M}_{full}$ is given by

$$\mathcal{M}_{full}\left(t\right) = \mathcal{M}_i\left(\frac{t}{2}\right)\mathcal{M}_c\left(t\right)\mathcal{M}_i\left(\frac{t}{2}\right) + \mathcal{O}\left(t^3\right), \tag{2}$$

where $\mathcal{M}_c$ and $\mathcal{M}_i$ are the evolution operators corresponding to $\mathcal{H}_c$ and $\mathcal{H}_i$, respectively. Synergia abstracts this technique by defining each simulation as a series of *steps*, each of which is defined by an ordered set of *operators*. A set of steps through an entire accelerator is called a *turn*. Typical simulations of circular accelerators may consists of thousands to a hundred thousand turns. Real accelerator cycles can be millions of turns. Accelerator operations often require changing the accelerator parameters such as altering magnet settings and RF phase shifts, possibly in response to beam conditions. Synergia can also handle linear accelerators, through which the beam only passes once. Then the (poorly named, in this context) number of turns is simply one.

A Synergia simulation consists of propagating a single bunch or train of bunches through a given number of turns. Along the way, various user-selected (and/or user-defined) *diagnostics* can be performed on the bunch(es) to monitor the state of the beam as it propagates through the machine. The simulation parameters are defined by a brief Python or C++ program written by the end user. The end-user program may use only existing Synergia classes, or may include custom extensions to those classes. The entire state of the simulation, including end-user extensions, may be checkpointed and/or resumed at any point. This checkpointing mechanism allows both for recovery from hardware failures and the chaining of multiple jobs in time-limited queues to complete one long simulation.

Synergia consists of a core set of C++ classes which are exposed to Python via Boost.Python [3] as described later in this paper. The independent-particle dynamics are handled by the CHEF [4] C++ libraries. Both Synergia and CHEF were developed at Fermilab. Synergia also depends on a variety of packages representing the current state-of-the-art in scientific computing. Fig. 1 displays the full set of Synergia dependencies.

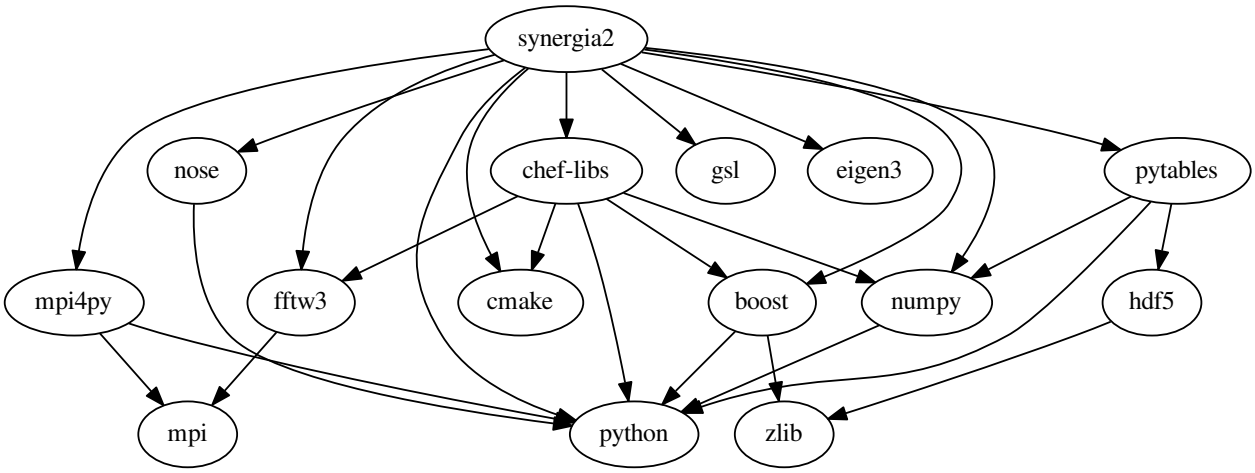

Fig. 1. Dependencies of Synergia. CHEF (chef-libs) is a set of independent-particle accelerator libraries. The other nodes consist of widely-used scientific computing and/or Python packages.

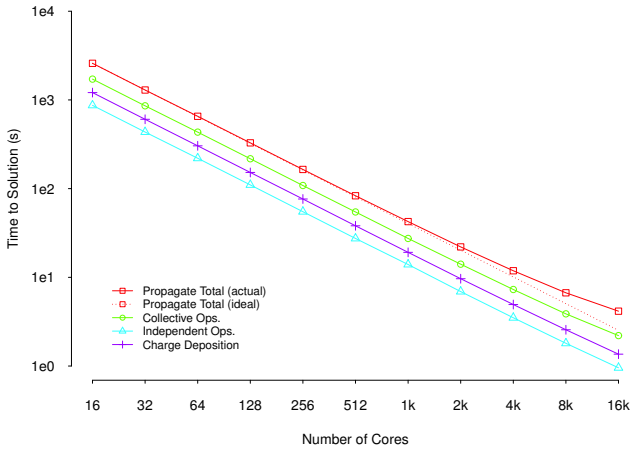

Fig. 2. Strong scaling of a single-bunch space charge simulation on ALCF's Mira, a Blue Gene/Q machine. The space charge calculation uses a $32 \times 32 \times 1024$ grid and 100M macroparticles. The red curve is the total time; other curves detail the scaling of the most computationally demanding portions of the calculation.

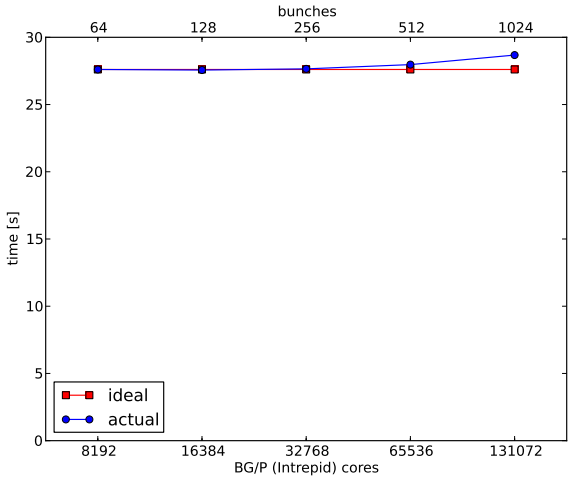

Fig. 3. Weak scaling of a multiple-bunch space charge simulation on ALCF's Intrepid, a Blue Gene/P machine. The space charge calculation uses a $32 \times 32 \times 1024$ grid and 100M macroparticles per bunch; the largest simulation has a total of over 13B macroparticles.

The Synergia design includes parallelism at its core. Simple Synergia simulations can be run on a desktop machine with a single core, but one can easily utilize a range of parallel resources from ranging from multi-core desktops to $100,000+$ core supercomputers. The central model is based on MPI, but extensions including hybrid MPI/OpenMP and GPU-based models exist in preliminary form. Figs. 2 and 3 demonstrate the scalability of the code. We find excellent strong scaling behavior over a range of a little less than a thousand. We also obtain excellent weak scaling in the number of macroparticles used (not shown) and number of bunches (Fig. 3).

A. *Why Python?*

Working particle accelerators have many operational parameters. Some of these are varied from run to run, while others may be adjusted by pre-determined algorithms or feedback loops. Realistic accelerator simulations will therefore require a very complicated description of simulation parameters. Synergia's use of Python as a driver gives the end user access to a full programming language to use in simulation description. In older accelerator simulation packages it was most common to either have a very limited declarative description of a simulation, or, in more advanced packages, an *ad hoc* language to enable limited programability. Using Python allows Synergia vastly greater flexibility while eliminating the development

```
from synergia import Propagate_actions, Pickle_helper

class Ramp_actions(Propagate_actions, Pickle_helper):
    def __init__(self, ramp_turns, turns_to_extract, initial_k1,
                 final_k1, final_k2l, rfko_kicker):
        Propagate_actions.__init__(self)
        Pickle_helper.__init__(self, ramp_turns, turns_to_extract,
                               initial_k1, final_k1, final_k2l, rfko_kicker)
        # ...

    def turn_end_action(self, stepper, bunch, turn_num):
        synergia_elements = \
            stepper.get_lattice_simulator().get_lattice().get_elements()
        # ...
        if turn_num <= self.ramp_turns:
            index = 0
            for element in synergia_elements:
                if element.get_type() == "multipole":
                    new_k2l = self.final_k2l[index]*turn_num / \
                        self.ramp_turns
                    element.set_double_attribute("k2l", new_k2l)
                    index += 1
            if turn_num == 1:
                old_intensity = bunch.get_total_num()
                n0 = old_intensity
                avg_rate = n0 / float(self.turns_to_extract - self.ramp_turns)
        # ...
```

Fig. 4. User-defined propagate actions class in Python.

effort needed to create yet another language.

The usefulness of Python in Synergia extends beyond static description. Many of the Synergia classes are designed to be extended in either C++ or Python. By doing so, the end-user can create dynamic simulations including time-varying machine parameters and active feedback. Such simulations can very closely model the actual behavior of the machine, allowing highly realistic simulations.

A real-world example of the power of Python in Synergia simulations can be seen in the Synergia model of the resonant extraction scheme for the proposed Mu2e experiment at Fermilab. The resonant extraction scheme involves a circuit that ramps non-linear magnets to create a moving resonance that progressively captures portions of the beam, causing it to be extracted at a (roughly) constant rate. A second circuit monitors the extraction rate and uses that information to drive a system known as RF knockout (RFKO) to enhance the uniformity of the rate. In the Synergia simulation of the resonant extraction process, a Python class extending the Synergia C++ class `Propagate_actions` provides the logic mimicking the action of both circuits. Portions of the class are shown in Fig. 4.

Another use for `Propagate_actions` is active feedback. A simple model of an accelerator damping system can be constructed as follows: measure the position of the bunch centroid using a beam position monitor (BPM) installed a given location. At a given point downstream, shift the particles toward the center of the beam pipe by an amount proportional to the offset at the BPM. This model can be implemented in Synergia using the `Damper_actions` class in Fig. 5.

There are many other ways in which end-users can extend Synergia through the Python interface. For example, Synergia already contains a rich set of *diagnostics* classes for measuring beam characteristics during the course of a simulation, but the possible set of such measurements is enormous. These classes can easily be extended in Python. Furthermore, the set of possibly interesting points in the simulation to perform such measurements is also enormous. A class analogous to `Propagate_actions`, `Diagnostic_actions`, can be extended to allow for arbitrary logic in measurements.

```
from synergia import Propagate_actions, Pickle_helper,
    bunch.Core_diagnostics

class Damper_actions(Propagate_actions, Pickle_helper):
    def __init__(self, bpm_location, damper_location):
        Propagate_actions.__init__(self)
        Pickle_helper.__init__(self, bpm_location, damper_location)
        self.bpm_location = bpm_location
        self.damper_location = damper_location
        self.bunchx = 0.0
        # ...

    def step_end_action(self, stepper, step, bunch, turn_num, step_num):
        # Measure the bunch position at the pickup (BPM) location
        if step_num == self.bpm_location:
            self.bunchx = Core_diagnostics().calculate_mean(bunch)[0]

        # Shift the bunch position at the damper location
        elif step_num == self.damper_location:
            # kick x momentum to restore position
            bunch.get_local_particles()[:, 1] += -gain*bunchx/self.betax
```

Fig. 5. Synergia implementation of a simple damper module.

B. *Why* Not *Python?*

Even though the advantages of Python-driven simulations are many, there are also drawbacks. The difficulties we have encountered in working with a Python/C++ package are, in order of importance, porting problems, complex debugging, difficulty in using external tools such as profilers, and the extra work that goes into maintaining the C++-Python interface.

Portability has been, and continues to be our number one problem. The Synergia code itself, which is written in very standard-compliant C++, is almost never a portability problem. Our portability problems really split into three different issues: availability of shared libraries, Python, and other dependent packages. On modern mainstream Unix-like systems, none of these issues are particularly troublesome. In Fedora 19, for example, all of our dependent packages (except, naturally CHEF) are available as part of the operating system. On supercomputing and other cutting-edge architectures such as Intel MIC, however, the story is much more complicated. Supercomputing operating systems as recent as Blue Gene/L and Cray Xt 4 did not support shared libraries at all (or in a way useful to us.) Even though the subsequent releases of those systems, Blue Gene/P and Cray Xt 5, do provide support for shared libraries, it has been our experience on the systems we have used that it is common to find important system libraries such as libz available in only static versions. Once the shared library problem has been solved, there is the problem of getting a version of Python that is fully compatible with dynamically loaded libraries. The problem is made much more complicated by the fact that most cutting-edge systems require cross-compiling and the Python build system is incompatible with cross-compiling as shipped. Having dealt with the shared library and Python issues, we still have to deal with building the Python-related dependent packages. Out of the 14 dependent packages in shown in Fig. 1, five are related to Python. Fortunately, these packages are usually the least of our portability problems, but some work is still required to get them installed.

Debugging and external tool difficulties arise after portability has been dealt with. In general, there exist tools that can do debugging and profiling in mixed language environments, but they are not necessarily the best options in every case, nor are they available everywhere. It is always simplest to perform debugging and profiling operations in a pure-C++ environment.

C. Evolution of the Use of Python in Synergia

The current version of Synergia, 2.1, is the third major step in the evolution of its design. In the original version, the simulation itself was entirely contained in a hybrid Fortran 90/C++ application. The main loop came from a legacy Fortran 90 code that we augmented to call the (C++) CHEF libraries for independent-particle physics. This design arose more out of necessity than choice – it was too difficult to split the Fortran 90 code into components to be called by anything else. The user interface was Python based, but the entire function of the Python portion of the code was to generate the human-unfriendly input file format for the Fortran 90 code, which consisted of nothing but floating point numbers. This solution was usable, but highly inflexible; we were limited by design of the

In version 2 of Synergia, we broke away from the Fortran 90 main loop. For the first time, the code was driven by Python, with calls to small pieces written in C++ and even some chunks laboriously split off the Fortran 90 code. Eventually, we replaced all of the functionality that was being provided in Fortran 90 with our own much more flexible C++ code. Nonetheless, the overall design was a complex mixture of Python and C++ at multiple levels of granularity. It was an improvement over the version which also included Fortran 90, but it still left a great deal to be desired in terms of robust design.

In version 2.1 of Synergia, we have finally achieved a robust class structure. Because of (potential) difficulties with Python/C++ applications, we have refactored the code to be a optionally C++-only. By creating a pure-C++ class structure, we now allow the option to write pure-C++ Synergia simulations. In general use, we find the Python-driven approach to be superior. However, in cases where the Python-related difficulties described above arise, we are now able to proceed in the simpler mono-language mode. As a side-effect, we have also found that creating a single-language class structure leads to a cleaner design. However, this approach puts demands on the Python-C++ interface, as we describe later in this work.

III. C++ Core Classes

The Synergia C++ class structure mimics the physical system being simulated. The simulation is a bunch (or bunches) of particles propagating through an accelerator lattice. The core Synergia classes are `Bunch`, which holds a distributed set of macroparticles and `Lattice`, which hold an abstract description of the series of elements that make up the accelerator lattice. The `Propagator` class applies a series of `Step`s composed of `Operator`s, while periodically executing various `Action`s, including the application of `Diagnostics`. See Fig. 6.

Because we have designed Synergia so that fully-C++ simulations are possible in addition to Python-driven ones, for reasons described in Subsection II-B, it is logical to have a natural fully-C++ interface. The implementation therefore makes full use of appropriate C++ features, including classes, inheritance, overloading, templates and STL containers. It also includes extensive use of the Boost libraries [5]. The list of libraries used includes *Boost.MultiArray* for multidimensional arrays, *shared_ptr* for memory management, *Boost.Filesystem* for filesystem operations, etc. The *Test* library is used for unit tests. The use of two more Boost libraries, *Serialization* and *Boost.Python* are described in the following subsection and section, respectively.

A. Serialization

Checkpointing is an important feature for large-scale simulation applications. It allows for recovery from hardware failures. It also allows for simulations whose duration exceeds the time limits frequently imposed by batch queue systems. The long-duration problem is particularly important in accelerator simulations because real accelerators propagate beams for very long times – times that correspond to simulations that would take months or longer. Although month-long accelerator jobs are rare, simulations that exceed the typical 12 or 24 hour limits imposed by many batch systems are common.

If Synergia was a static application, checkpointing might be implemented by some sort of scheme that re-does initialization, then skips to the place where the last job left off. The fact that Synergia is really a library that end-users use to create their own applications makes that sort of approach impractical. Instead, Synergia includes an object serialization scheme that allows the state of each of its object to be written to and/or restored from disk.

The Synergia serialization implementation utilizes the Boost Serialization library, which requires each class to have a templated `serialize` method. In certain situations, separate `save` and `load` methods are substituted for the single `serialize` method. Each class must send/get its state, usually just the values of its member values, to/from the `Archive` object, which is a template parameter. Both (efficient) binary and (large and slow, but human readable) XML archive types are available. Synergia allows the user to decide at runtime which archive format is to be used. The Serialization library takes care of things like pointers and STL containers transparently. The flexible nature of this serialization scheme allows arbitrary end-user programs to save and restore their states transparently. The only burden on the end-user is the requirement to add a serialization method to new *C++* classes extending the Synergia classes. *Python* classes that extend the Synergia classes can be handled automatically; see the following Section.

IV. Python-C++ Interface

We considered many different Python-C++ binding generators for Synergia. Some were clearly intended for specific projects and it wasn't clear that they would be supported (or even supportable) on some of the relatively exotic architectures on which we planned to (and do) run Synergia. Many of the others were of the "C/C++" variety, meaning that they were really designed for C code with a few additional C++ features. Since we have a C++ class interface that includes many of the non-trivial features of C++, the "C/C++" binding generators were deemed insufficient for our needs. In the end we chose Boost.Python because it allows for the most transparent translation of C++ interfaces and data structures to Python with minimal portability issues. Another advantage for us of this choice was that we were already using several Boost libraries, so no additional dependency was added to Synergia.

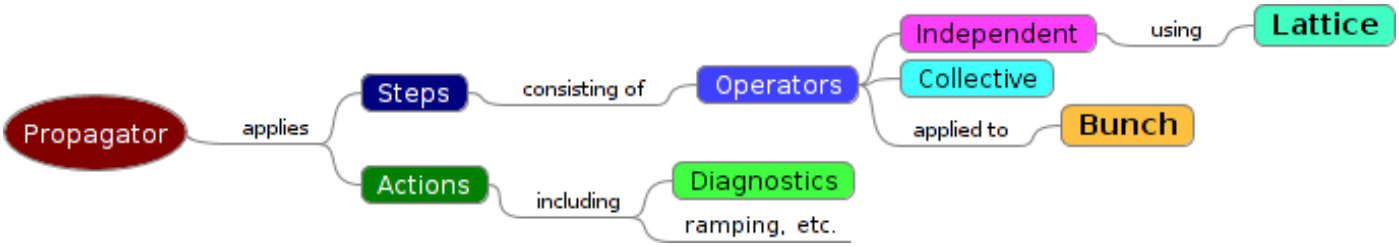

Fig. 6. Overview of Synergia structure.

```
template<class Archive>
    void
    Chef_lattice_section::serialize(Archive & ar, const unsigned int version)
    {
        ar & BOOST_SERIALIZATION_NVP(begin_index);
        ar & BOOST_SERIALIZATION_NVP(end_index);
        ar & BOOST_SERIALIZATION_NVP(chef_lattice_sptr);
    }
```

Fig. 7. Example of a class serialize method. The `BOOST_SERIALIZATION_NVP` macro allows variables to been written in XML archives as name-value pairs, enhancing readability of the archive itself.

Boost.Python has many features that make it easy to expose our C++ interfaces to Python:

- Classes can be wrapped straightforwardly, preserving inheritance relations. Python classes can inherit from C++ classes.
- C++ classes can call inherited methods in Python classes through callbacks.
- Other structures can also be wrapped in a straightforward manner, including enums, static data member and constants.
- Overloaded C++ signatures can be translated to Python with a minimum of effort.
- Classes and shared pointers to classes can be interchanged transparently at the Python level.

One C++ construct which is hard to translate to Python are functions which take mutable atomic types as arguments. While we generally avoid such constructs in our C++ code as a matter of style, there are a few occasions where mutable atomic types are the simplest interface. Boost.Python allows us to write a custom wrapper for cases like these. We invariably map them to Python functions which return multiple values.

Another feature of Boost.Python we use extensively is container conversion. While wrapping our application-specific classes and all of their methods is a logical choice, for the basic STL containers `std::vector` and `std::list` it is more natural to convert to the native Python `list` or `tuple` types. Boost.MultiArray types are naturally converted to Python Numpy arrays – this can be accomplished without copying the underlying data. An excellent example of this interface can be seen in the last line of Fig. 5, where the particle coordinate data, which is stored in a C++ Boost.MultiArray, is simply modified in Python through the Numpy interface. In the end Boost.Python allows our C++ code to look like C++ code and our Python code to look like Python code.

```
# !/usr/bin/env synergia
import synergia

from synergia.foundation import Four_momentum, Reference_particle, pconstants
from synergia.lattice import Mad8_reader, Lattice
from synergia.bunch import Bunch, Diagnostics_basic
from synergia.simulation import Independent_stepper_elements, Bunch_simulator, \
    Propagator
from fodo_options import opts
from mpi4py import MPI
import sys

# We wrap the entire simulation in a try..except block in order to allow
# for graceful failures under MPI.
try:
    # Read the lattice named "fodo" from the Mad8 file "fodo.lat"
    lattice = Mad8_reader().get_lattice("fodo", "fodo.lat")

    # Define the simulation steps
    stepper = Independent_stepper_elements(lattice, opts.map_order,
                                           opts.steps_per_element)

    # Define the parameters for the bunch
    bunch = synergia.optics.generate_matched_bunch_transverse(
                stepper.get_lattice_simulator(),
                opts.x_emit, opts.y_emit, opts.z_std, opts.dpop,
                opts.real_particles, opts.macro_particles,
                seed=opts.seed)

    # Apply basic diagnostics every step
    diagnostics = Diagnostics_basic("diagnostics.h5")
    bunch_simulator = Bunch_simulator(bunch)
    bunch_simulator.add_per_step(diagnostics)

    # Perform the simulation
    propagator = Propagator(stepper)
    propagator.propagate(bunch_simulator, opts.turns, opts.max_turns,
                         opts.verbosity)

except Exception, e:
    sys.stderr.write(str(e) + '\n')
    MPI.COMM_WORLD.Abort(777)
```

Fig. 8. A simple Synergia simulation of a focusing-defocusing (FODO) accelerator cell.

A. A Simple Synergia Python Simulation

In Fig. 8 we demonstrate a complete Synergia simulation.

B. MPI and the Python/C++ Interface

Notice that fodo example, Fig. 8, only contains two explicit references to MPI: The `mpi4py` module is imported and `MPI.COMM_WORLD.Abort` is called if an exception is caught. The former has the implicit effect of properly calling `MPI_Init` and `MPI_Finalize` at appropriate times. The latter is important for a clean exit in cases where (C++ and/or Python) exceptions are thrown in a subset of processes. Most end-user usage will not require any other explicit references to MPI. The C++ code manages MPI communicators through a thin wrapper class, `Commxx`, which handles creation, destruction, and serialization of communicators. `Commxx` contains simple member to access the rank and size of a communicator, as well as access to the underlying raw MPI object. The Python interface to `Commxx` is handled through Boost.Python just like any other Synergia C++ class.

```
class Pickle_helper:
    __getstate_manages_dict__ = 1
    def __init__(self, *args):
        self.args = args
    def __getinitargs__(self):
        return self.args
    def __getstate__(self):
        return self.__dict__
    def __setstate__(self, state):
        self.__dict__ = state
```

Fig. 9. The implementation of the Synergia `Pickle_helper` class.

```
    template<class Archive>
        void
        save(Archive & ar, const unsigned int version) const
        {
            ar &
            BOOST_SERIALIZATION_BASE_OBJECT_NVP(Propagate_actions);
            std::string pickled_object(
                    extract<std::string >(
                            import("cPickle").attr("dumps")(self)));
            ar & BOOST_SERIALIZATION_NVP(pickled_object);
        }
    template<class Archive>
        void
        load(Archive & ar, const unsigned int version)
        {
            ar &
            BOOST_SERIALIZATION_BASE_OBJECT_NVP(Propagate_actions);
            std::string pickled_object;
            ar & BOOST_SERIALIZATION_NVP(pickled_object);
            str pickle_str(pickled_object);
            self = import("cPickle").attr("loads")(pickle_str);
        }
    BOOST_SERIALIZATION_SPLIT_MEMBER()
```

Fig. 10. The implementation of the `load` and `save` methods required by Boost Serialization in the callback struct used wraps the `Propagate_actions` class.

C. Serialization of Python extensions

As we argued in the section on serialization in the C++ code, object serialization is the only general way to implement checkpointing in a set of libraries like Synergia, as opposed to a static application where various initializations can rerun at resume time. Although Synergia provides a transparent C++ object serialization system, can it also serialize end-user objects written in Python? Indeed, it can.

The Synergia Python extension serialization mechanism relies on the built-in Python pickle module. The C++ serialization mechanism calls pickle to serialize the Python object to a string, then writes that string to the C++ archive. The pickle module imposes a few requirements on the end-user Python class. These can be met for most classes by simply inheriting from the `Pickle_helper` class provided by Synergia. The example in Fig. 4 does just that. Figs. 9 and 10 show the Python and C++ implementations of this mechanism, respectively.

V. Conclusion

Synergia is a modern, flexible accelerator simulation framework. The design is parallel at its very core, resulting in scalability to over $100,000$ cores. By using Python as a driver language, Synergia allows end-users to create realistic accelerator simulations of arbitrary complexity. Synergia provides end-user simulations with checkpointing ability with a minimum of user intervention. In this paper we described how we arrived at the current state of the code including some of the challenges along the way. All of the code described above is available from the web site in Ref. [1]. The idea of pairing a core C++ application framework with a Python driver with two way communication between the two parts can be applied in other domains where complex and data intensive C++ application would benefit from a rich and flexible programming language.

Acknowledgment

This work was performed at Fermilab, operated by Fermi Research Alliance, LLC under Contract No. De-AC02-07CH11359 with the United States Department of Energy. It was also supported by the ComPASS project, funded through the Scientific Discovery through Advanced Computing program in the DOE Office of High Energy Physics. We also used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

References

[1] J. Amundson, "Synergia2.1: A framework for accelerator simulations." [Online]. Available: https://cdcvs.fnal.gov/redmine/projects/synergia2

[2] J. Amundson, P. Spentzouris, J. Qiang, and R. Ryne, "Synergia: An accelerator modeling tool with 3-d space charge," *J. Comp. Phys.*, vol. 211, pp. 229–248, 2006.

[3] "The boost.python c++ library." [Online]. Available: http://www.boost.org/doc/libs/1_54_0/libs/python/doc/index.html

[4] L. Michelotti, "C++ objects for beam physics," in *Proceedings of the 14th IEEE Particle Accelerator Conference*, 1991.

[5] "The boost c++ libraries." [Online]. Available: http://www.boost.org/index.html