New tools for the simulation of coupled bunch instabilities driven by electron cloud

G. Iadarola

Many thanks to:

IT HPC team and in particular N. Hoimyr, C. Lindqvist, P. Llopis

INFN-CNAF HPC team and in particular A. Falabella

ABP Computing Working Group

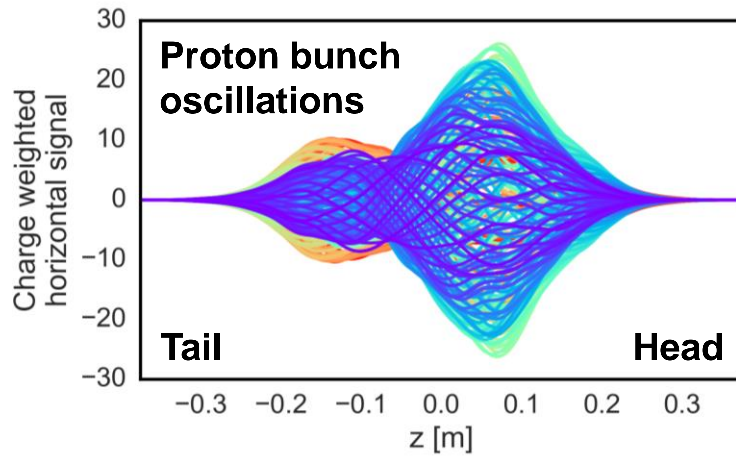
G. Arduini, X. Buffat, K. Li, L. Mether, E. Metral, A. Romano, G. Rumolo, L. Sabato

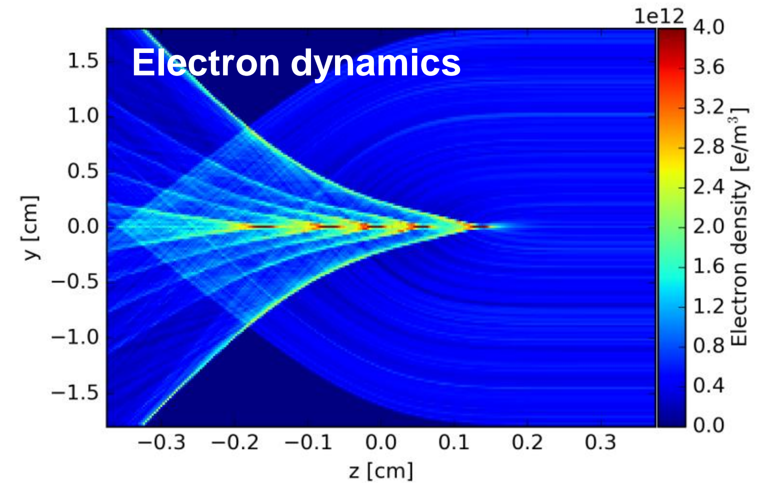
and J-L Vay (LBNL)

Outline

- **Introduction**
- **Parallelization strategy**
- **Extension of the PyPARIS parallelization layer**
  - Structure, interface and implementation
  - Beam generation and slicing
  - Beam data saving
- **Modifications to the PyECLOUD code**
  - Overall structure
  - Cloud data saving
  - Cloud simulation logics
  - Time discretization
- **Examples of simulation results**
- **Simulation time**
  - Scaling at constant number of bunches
  - Effect of hyperthreading
  - Scaling with number of bunches
- **Summary and next steps**

- **Instabilities** driven by **e-cloud** arise from the **coupling via electromagnetic forces** between the motion of the electrons and the dynamics of the proton beam





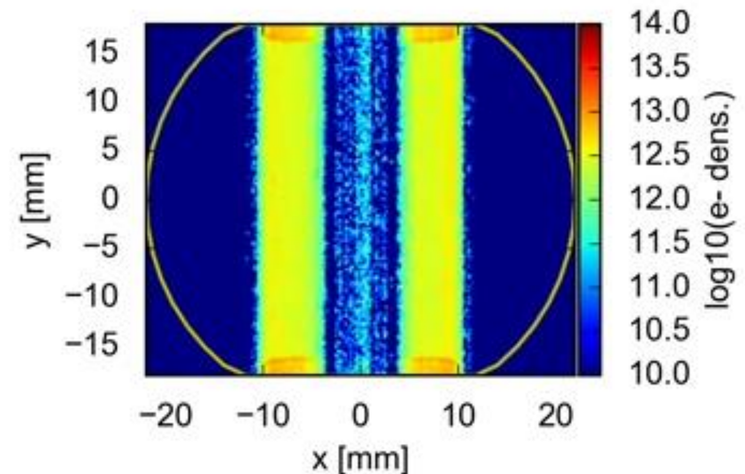
- Due to the **non-linear nature of the electron dynamics** it is difficult to study these instabilities using analytical treatments
- Modeling and understanding strongly relies on **numerical simulations** (macroparticle codes)
    - **PyECLOUD-PyHEADTAIL suite**[1], developed and maintained at CERN

[1] *G. Iadarola, E. Belli, K. Li, L. Mether, A. Romano, G. Rumolo, "Evolution of Python Tools for the Simulation of Electron Cloud Effects", Proceedings of IPAC17*

Introduction

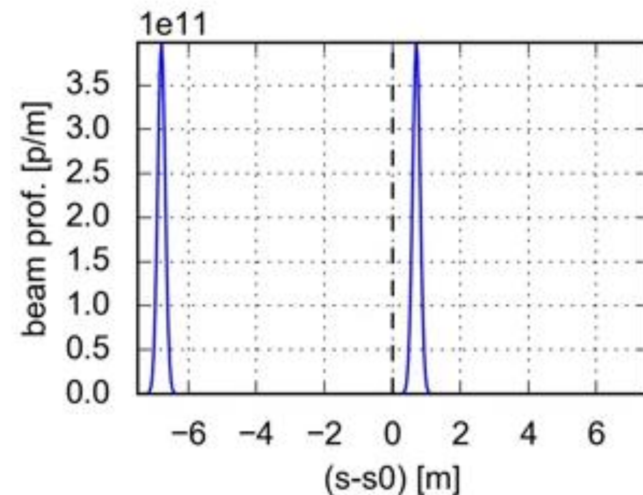
These simulations are **computationally very heavy**:

- Electron motion is very fast → requires very **short time steps** (~10 ps)
- Impact on the beam visible only on accumulated effect on **many turns** (~1 s)
- **Many macroparticles** are needed to minimize numerical noise (~250k per e-cloud interaction)



- Until recently we were able to simulate only **single-bunch effects**, i.e. coupling introduced by the electrons between **head and tail of the same bunch**:
  - Short interaction time (few ns)
  - Several simplifications possible

- We **could not simulate "coupled-bunch instabilities"**, i.e. coupling among different bunches within a bunch train:
  - Requires the simulation of the full **e-cloud buildup** process coupled with the beam dynamics
  - Too heavy for a standard computer…



Required computation time on single CPU core

Case study: instability of an **LHC bunch train** (72b) due to the interaction with the e-cloud in the **arc dipoles** at 450 GeV (modeled by 8 e-cloud interactions)

Task	Time
Single interaction of the train with the e-cloud	2.4 h
Single turn (8 e-cloud interactions)	19.2 h
Instability simulation (1000 turns)	19000h = 800 days = **2.2 years**

To make the simulation affordable,
we need to gain at least two orders of magnitude on the simulation time

→ Possible only using **High Performance Computing (HPC)** resources

Outline

- Introduction
- **Parallelization strategy**
- Extension of the PyPARIS parallelization layer
  - Structure, interface and implementation
  - Beam generation and slicing
  - Beam data saving
- Modifications to the PyECLOUD code
  - Overall structure
  - Cloud data saving
  - Cloud simulation logics
  - Time discretization
- Examples of simulation results
- Simulation time
  - Scaling at constant number of bunches
  - Effect of hyperthreading
  - Scaling with number of bunches
- Summary and next steps

Parallelization strategy: step 1

Parallelization over accelerator segments

Each CPU-core simulates a different portion of the machine (each containing at least one e-cloud interaction)

The different bunches are placed in a **queue** accessible by core 1

Parallelization strategy: step 1

Parallelization over accelerator segments

Core 2

Core 1

Core 3

Core 4

Bunch slot queue

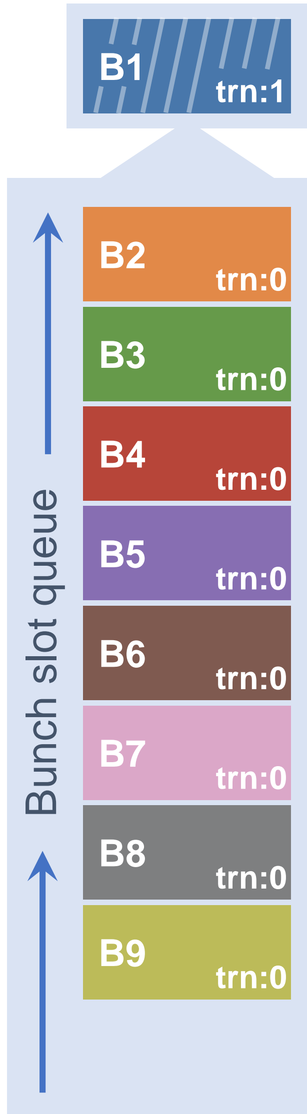
B1 trn:0
B2 trn:0
B3 trn:0
B4 trn:0
B5 trn:0
B6 trn:0
B7 trn:0
B8 trn:0
B9 trn:0

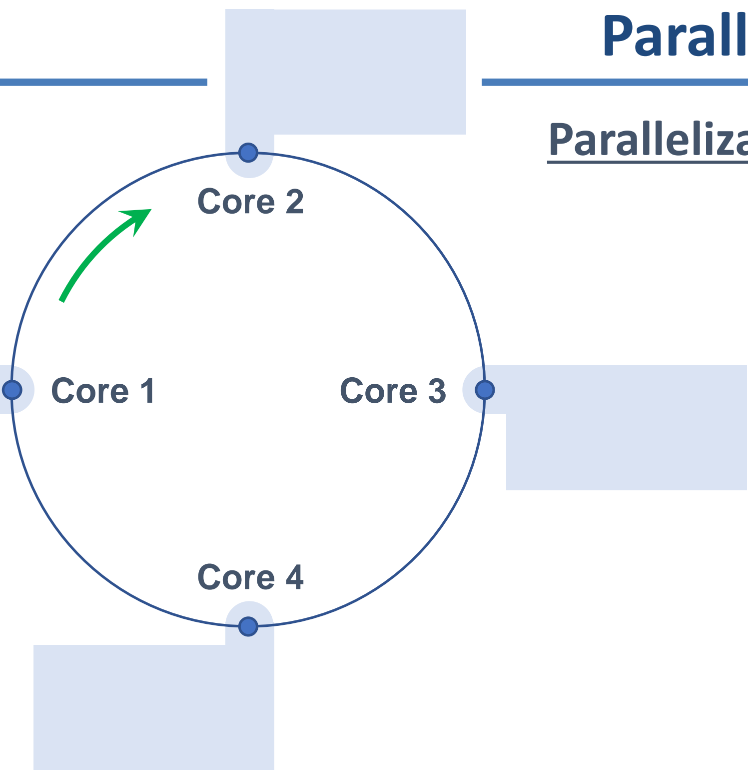
Each CPU-core simulates a different portion of the machine (each containing at least one e-cloud interaction)

First iteration:

Parallelization strategy: step 1

Parallelization over accelerator segments





Each CPU-core simulates a different portion of the machine (each containing at least one e-cloud interaction)

First iteration:

- **Core 1 pops a bunch** from the queue
- The bunch is **sliced longitudinally** (to compute the interaction with the e-cloud)
- The **interaction with the first accelerator segment** (including e-cloud) is computed
- The bunch is **passed to the next core**

Parallelization strategy: step 1

Parallelization over accelerator segments

B1 trn:1
Core 2
B2 trn:1
Core 1
Core 3
Core 4
Bunch slot queue
B3 trn:0
B4 trn:0
B5 trn:0
B6 trn:0
B7 trn:0
B8 trn:0
B9 trn:0

Each CPU-core simulates a different portion of the machine (each containing at least one e-cloud interaction)

At each iteration a new bunch is taken from the queue

Parallelization strategy: step 1

Parallelization over accelerator segments

Each CPU-core simulates a different portion of the machine (each containing at least one e-cloud interaction)

At each iteration a new bunch is taken from the queue

Parallelization strategy: step 1

Parallelization over accelerator segments

Each CPU-core simulates a different portion of the machine (each containing at least one e-cloud interaction)

After the bunch interacts with the last segment:

- Slices are re-merged
- Synchrotron motion is applied

Parallelization strategy: step 1

Parallelization over accelerator segments

Each CPU-core simulates a different portion of the machine (each containing at least one e-cloud interaction)

After the bunch interacts with the last segment:

- Slices are re-merged
- Synchrotron motion is applied
- The bunch is pushed into the queue for the next turn

Parallelization strategy: step 1

Parallelization over accelerator segments

Each CPU-core simulates a different portion of the machine (each containing at least one e-cloud interaction)

Loop continues for the required number of turns…

Parallelization strategy: step 1

Parallelization over accelerator segments

Core 2
Core 1
Core 3
Core 4
B5 trn:1
B6 trn:1
B4 trn:1
B3 trn:1
B7 trn:0
B8 trn:0
B9 trn:0
B1 trn:1
B2 trn:1
Bunch slot queue

Each CPU-core simulates a different portion of the machine (each containing at least one e-cloud interaction)

Loop continues for the required number of turns…

Parallelization strategy: step 1

Parallelization over accelerator segments

Core 2

B6 trn:1

Core 1

B7 trn:1

Core 3

B5 trn:1

Core 4

B4 trn:1

Bunch slot queue

B8 trn:0

B9 trn:0

B1 trn:1

B2 trn:1

B3 trn:1

Each CPU-core simulates a different portion of the machine (each containing at least one e-cloud interaction)

Loop continues for the required number of turns…

Parallelization strategy: step 1

Parallelization over accelerator segments

B7 trn:1

Core 2

B8 trn:1

Core 1

Core 3

B6 trn:1

B9 trn:0

B1 trn:1

B2 trn:1

B3 trn:1

B4 trn:1

Bunch slot queue

Core 4

B5 trn:1

Each CPU-core simulates a different portion of the machine (each containing at least one e-cloud interaction)

Loop continues for the required number of turns…

Parallelization strategy: step 1

Parallelization over accelerator segments

Each CPU-core simulates a different portion of the machine (each containing at least one e-cloud interaction)

Loop continues for the required number of turns…

Parallelization strategy: step 1

Parallelization over accelerator segments

Each CPU-core simulates a different portion of the machine (each containing at least one e-cloud interaction)

Loop continues for the required number of turns…

Parallelization strategy: step 1

Parallelization over accelerator segments

B1 trn:2

Core 2

B2 trn:2

Core 1

Core 3

B9 trn:1

Core 4

B3 trn:1

B4 trn:1

B5 trn:1

B6 trn:1

B7 trn:1

B8 trn:1

Bunch slot queue

Each CPU-core simulates a different portion of the machine (each containing at least one e-cloud interaction)

Loop continues for the required number of turns…

Parallelization strategy: step 1

Parallelization over accelerator segments

B2 trn:2

Core 2

B3 trn:2

Core 1

Core 3

B1 trn:2

B4 trn:1

B5 trn:1

B6 trn:1

B7 trn:1

B8 trn:1

Core 4

B9 trn:1

Bunch slot queue

Each CPU-core simulates a different portion of the machine (each containing at least one e-cloud interaction)

This approach **allows exploiting a number of cores equal to the number of e-cloud interactions**
→ In our case study (8 interactions) we can gain up to a factor of 8 but not more…

Parallelization strategy: step 2

Parallelization over different turns

The simulation starts as before…

Parallelization strategy: step 2

Parallelization over different turns

The simulation starts as before…

Parallelization strategy: step 2

Parallelization over different turns

Core 1
Core 2
Core 3
Core 4

B1 trn:1
B2 trn:1
B3 trn:0
B4 trn:0
B5 trn:0
B6 trn:0
B7 trn:0
B8 trn:0
B9 trn:0

Bunch slot queue

The simulation starts as before…

Parallelization strategy: step 2

Parallelization over different turns

B2 trn:1
Core 2
B3 trn:1
Core 1
Core 3
B1 trn:1
Core 4
B4 trn:0
B5 trn:0
B6 trn:0
B7 trn:0
B8 trn:0
B9 trn:0
Bunch slot queue

The simulation starts as before…

Parallelization strategy: step 2

Parallelization over different turns

B3 trn:1

Core 2

B4 trn:1

Core 1

Core 3

B2 trn:1

Core 4

B1 trn:1

Bunch slot queue

B5 trn:0

B6 trn:0

B7 trn:0

B8 trn:0

B9 trn:0

The simulation starts as before…

Parallelization strategy: step 2

Parallelization over different turns

Hypothesis: the e-cloud decays between consecutive turns (abort gap)

Parallelization strategy: step 2

Parallelization over different turns

Hypothesis: the e-cloud decays between consecutive turns (abort gap)

→ Once the bunch has completed the first turn, instead of going to the queue, it can be sent to a **second set of CPU-cores** to simulate the following turn

Parallelization strategy: step 2

Parallelization over different turns

Hypothesis: the e-cloud decays between consecutive turns (abort gap)

→ Once the bunch has completed the first turn, instead of going to the queue, it can be sent to a **second set of CPU-cores** to simulate the following turn

Parallelization strategy: step 2

Parallelization over different turns

While a set of cores is simulating the first turn for the tail of the train another set of cores is already simulating the second turn for the head of the train

Parallelization strategy: step 2

Parallelization over different turns

While a set of cores is simulating the first turn for the tail of the train another set of cores is already simulating the second turn for the head of the train

Parallelization strategy: step 2

Parallelization over different turns

While a set of cores is simulating the first turn for the tail of the train another set of cores is already simulating the second turn for the head of the train

Parallelization strategy: step 2

Parallelization over different turns

While a set of cores is simulating the first turn for the tail of the train another set of cores is already simulating the second turn for the head of the train

Parallelization strategy: step 2

Parallelization over different turns

While a set of cores is simulating the first turn for the tail of the train another set of cores is already simulating the second turn for the head of the train

B9 trn:1

Core 2

B1 trn:3

Core 1

Core 3

B8 trn:1

B2 trn:2

Core 4

B7 trn:1

B5 trn:2

Core 6

Bunch slot queue

B6 trn:2

Core 5

Core 7

B4 trn:2

Core 8

B3 trn:2

Parallelization strategy: step 2

Parallelization over different turns

While a set of cores is simulating the first turn for the tail of the train another set of cores is already simulating the second turn for the head of the train

B1 trn:3
Core 2
B2 trn:3
Core 1
Core 3
B9 trn:1
B3 trn:2
Core 4
B8 trn:1
B6 trn:2
Core 6
Bunch slot queue
B7 trn:2
Core 5
Core 7
B5 trn:2
Core 8
B4 trn:2

Parallelization strategy: step 2

Parallelization over different turns

While a set of cores is simulating the first turn for the tail of the train another set of cores is already simulating the second turn for the head of the train

B2 trn:3
Core 2
B3 trn:3
Core 1
Core 3
B1 trn:3
B4 trn:2
Core 4
B9 trn:1
Bunch slot queue
B7 trn:2
Core 6
B8 trn:2
Core 5
Core 7
B6 trn:2
Core 8
B5 trn:2

Parallelization strategy: step 2

Parallelization over different turns

While a set of cores is simulating the first turn for the tail of the train another set of cores is already simulating the second turn for the head of the train

Parallelization strategy: step 2

Parallelization over different turns

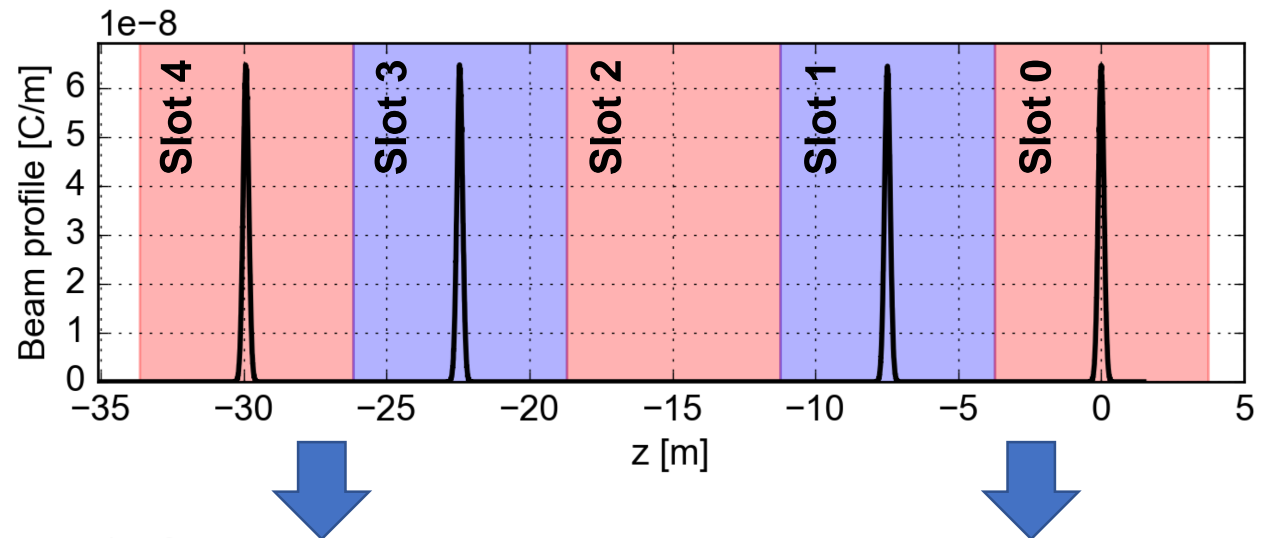
- This trick can be repeated multiple times (multiple rings of CPUs)
- The **number of CPU-cores** that can be exploited is **limited by the number of bunches** (72 in our test case)

Parallelization strategy: step 3

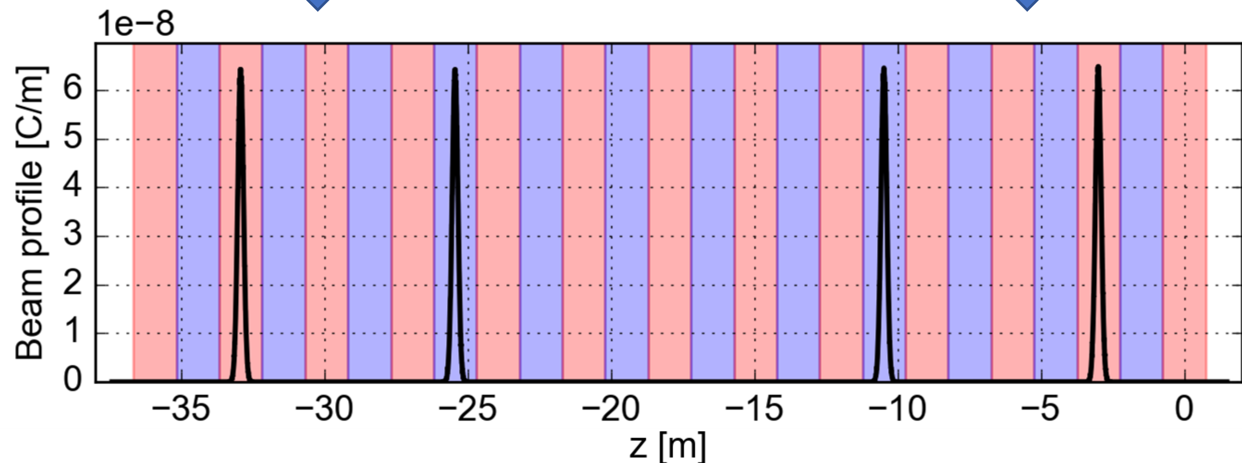
Electrons need to be tracked **also in between bunches** (large fraction of the computation time)

→ By slicing the beam in **shorter slots** (made by an integer number of RF buckets) we can **share the workload over a larger number of CPUs**

25 ns slots:



5 ns slots:



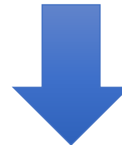
Outline

- Introduction
- Parallelization strategy
- **Extension of the PyPARIS parallelization layer**
  - Structure, interface and implementation
  - Beam generation and slicing
  - Beam data saving
- Modifications to the PyECLOUD code
  - Overall structure
  - Cloud data saving
  - Cloud simulation logics
  - Time discretization
- Examples of simulation results
- Simulation time
  - Scaling at constant number of bunches
  - Effect of hyperthreading
  - Scaling with number of bunches
- Summary and next steps

Reminder on PyPARIS

Main objectives:

- Hide as much as possible the parallelization technology → **keep physics and parallelization code physically separated**
- The physics should **be extendable by a developer who is unaware about the parallelization details**
- Keep possibility **to change parallelization technology** (e.g. MPI vs. multiprocess) with **no intervention on the physics code**
- Minimize **changes in existing tools** (PyECLOUD, PyHEADTAIL, PyPIC etc…) to avoid painful re-validation phases



Developed an **additional python layer taking care only of the parallelization**, separate from PyHEADTAIL and PyECLOUD, and working together with them:

PyPARIS

Python Parallel Ring Simulator

More info at https://github.com/PyCOMPLETE/PyPARIS/wiki

The PyPARIS parallelization layer

The simulation is managed through **two python objects** (actually each process will have an instance of the two classes)

RingOfCPUs object
(implemented in PyPARIS)

Takes care of the parallelization

Handles a **set of processes** organized in a **ring structure**:

- Data transfer between processes
- Synchronization
- Handles special tasks performed by the master process
- Handles "messages" broadcasted from the master to all processes

This object is **fully abstract** (no physics included)→ specific tasks to be performed are defined by the Simulation object

Uses:

Communicator (MPI or MPI-like)

Simulation object
(can be modified by the user)

Contains the physics

Defines the **task to be performed by the master and worker** processes at the different stages.

Physics of the simulation can be defined using Python tools like:

PyHAEDTAIL

PyECLOUD

PyPIC

NAFFlib

User defined Python (or Python-callable) code

The PyPARIS parallelization layer

A **new class called "ring_of_CPUs_multiturn"** has been implemented in PyPARIS

→ Made available for production in version 2.0.0

Search or jump to... Pull requests Issues Marketplace Explore

PyCOMPLETE / **PyPARIS**
forked from giadarol/PyParaSlice

Unwatch 2 Unstar 1 Fork 1

Code | Pull requests 0 | Projects 0 | Wiki | Insights

Releases | Tags | Edit release | Delete

Latest release

v2.0.0

8684a41

PyPARIS Version 2.0.0

giadarol released this 13 days ago

Assets 2

Source code (zip)

Source code (tar.gz)

New feature:

- Capability of handling **multi-bunch beams** implementing parallelisation over machine segments and **pipe-line over different turns.**

Additional info:

- An example of setup to simulate a multi-bunch instability driven by e-cloud is available here.
- A description of the multi-bunch interface can be found in the wiki pages.

More info at https://github.com/PyCOMPLETE/PyPARIS/wiki

PyPARIS multi-bunch: physics description

As for single-bunch parallel simulations, physics is described by writing a **simulation class** (description and full example available in github)

Each running process has an instance of the simulation class

```
import PyECLOUD, PyHEADTAIL, ...

class Simulation(object):
    def __init__(self):

        self.N_turns = 5000
        self.N_parellel_rings = 10

    def init_all(self):
        # Executed on all cores at the beginning of the simulation
        #  - Generate the portion of the machine to be
        #    simulated by the specific core.
        #  - Insert and initialize the e-cloud elements
        #  - At end-ring: prepare for global bunch operations
        if self.ring_of_CPUs.I_am_at_end_ring:
             self.non_parallel_part=\
                  self.machine.one_turn_map[-n_non_sliceable:]

    def init_master(self):
        # Executed on the “master”, i.e. first core of first ring:
        #  - Initialize the queue with the bunches to be simulated
        return list_bunches

    def init_start_ring(self):
        # Executed at each core that is at the start of a ring:
        #  - Prepare bunch monitor
        self.bunch_monitor = ...

                                                            [...]
```

PyPARIS multi-bunch: physics description

As for single-bunch parallel simulations, physics is described by writing a **simulation class** (description and full example available in github)

```
import PyECLOUD, PyHEADTAIL, ...

class Simulation(object):
    def __init__(self):

        self.N_turns = 5000
        self.N_parellel_rings = 10

    def init_all(self):
        # Executed on all cores at the beginning of the simulation
        #  - Generate the portion of the machine to be
        #    simulated by the specific core.
                  sert and initialize the e-cloud elements
                   end-ring: prepare for global bunch operations
                  .ring_of_CPUs.I_am_at_end_ring:
                  f.non_parallel_part=\
                    self.machine.one_turn_map[-n_non_sliceable:]

                  ster(self):
                  ted on the "master", i.e. first core of first ring:
                  itialize the queue with the bunches to be simulated
                  list_bunches

                  art_ring(self):
                  ted at each core that is at the start of a ring:
                  epare bunch monitor
                  nch_monitor = ...

                                                                [...]
```

Each core is aware of its role in the topology through a specific **object "ring_of_CPUs"** attached by PyPARIS to the simulation object.

```
# General
self.ring_of_CPUs.N_nodes
self.ring_of_CPUs.myid
self.ring_of_CPUs.I_am_the_master

# Specific of multibunch
self.ring_of_CPUs.N_nodes_per_ring
self.ring_of_CPUs.myring
self.ring_of_CPUs.myid_in_ring
self.ring_of_CPUs.I_am_at_start_ring
self.ring_of_CPUs.I_am_at_end_ring
```

PyPARIS multi-bunch: physics description

As for single-bunch parallel simulations, physics is described by writing a **simulation class** (description and full example available in github)

Each running process has an instance of the simulation class

```
class Simulation(object):
                                                        [...]

    def perform_bunch_operations_at_start_ring(self, bunch):
        # Executed at each turn by cores at the start of each
        # ring:
        #  - Save bunch momenta

    def slice_bunch_at_start_ring(self, bunch):
        # Executed by cores at the start of each ring:
        #  - Pop a bunch and slice it
        return list_slices

    def treat_piece(self, slice):
        # Executed by all cores:
        #  - Simulate the interaction of a slice with the
        #    assigned part of the ring

    def merge_slices_at_end_ring(self, list_slices):
        # Executed by cores at the end of each ring:
        #  - Merge the slices back into a single bunch object
        return bunch

    def perform_bunch_operations_at_end_ring(self, bunch):
        # Executed by cores at the end of each ring:
        #  - Physics that needs to be performed globally on
        #    the bunch (e.g. lumped longitudinal tracking,
        #    bunch-by-bunch feedback)
```

PyPARIS: behind the scene

PyPARIS has been developed using **mpi4py** (python wrapper for MPI) to implement the parallelization:

- It can run **without MPI** using embedded "**dummy MPI communicator**", based on python multiprocessing (it can be used to run on a single multi-core machine)

Communication features:

- A single **array of floats** is passed at each iteration
- The PyHEADTAIL **particle object is transformed into a buffer of float**, which is transmitted to another process, where it is re-translated back into a particle objects
  - **Helper functions** to perform these operations are available in PyPARIS
- Several buffers (beam slices) can be transferred together (a list can be passed)
- There is **no other data going around** (apart from stop signal at end-simulation)
  - Additional information to manage the simulation is **attached as an extra member (dictionary) to the bunch object**. This dictionary is "casted" to a float array and included in the buffer

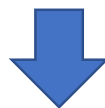
PyPARIS: beam generation and slicing

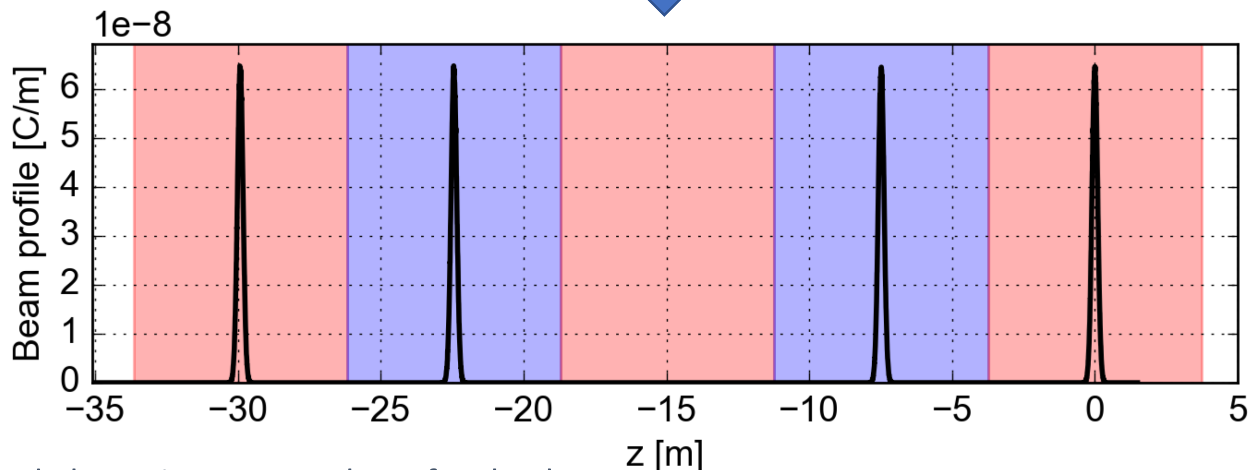
A module has been included in PyPARIS to **generate a PyHEADTAIL multibunch beam with meta-data for usage in e-cloud simulations** (PyHEADTAIL slicing under the hood), could be moved to PyHEADTAIL generators module in the future

```
from machines_for_testing import SPS
machine = SPS(machine_configuration = 'Q20-injection')

filling_pattern = [1., 1., 0., 1., 1.]
non_linear_long_matching = True

list_bunches = gmb.gen_matched_multibunch_beam(
         machine, n_mpart_bunch, filling_pattern,
         b_spac_s, bunch_intensity, epsn_x, epsn_y, sigma_z,
         non_linear_long_matching, min_inten_slice4EC)
```



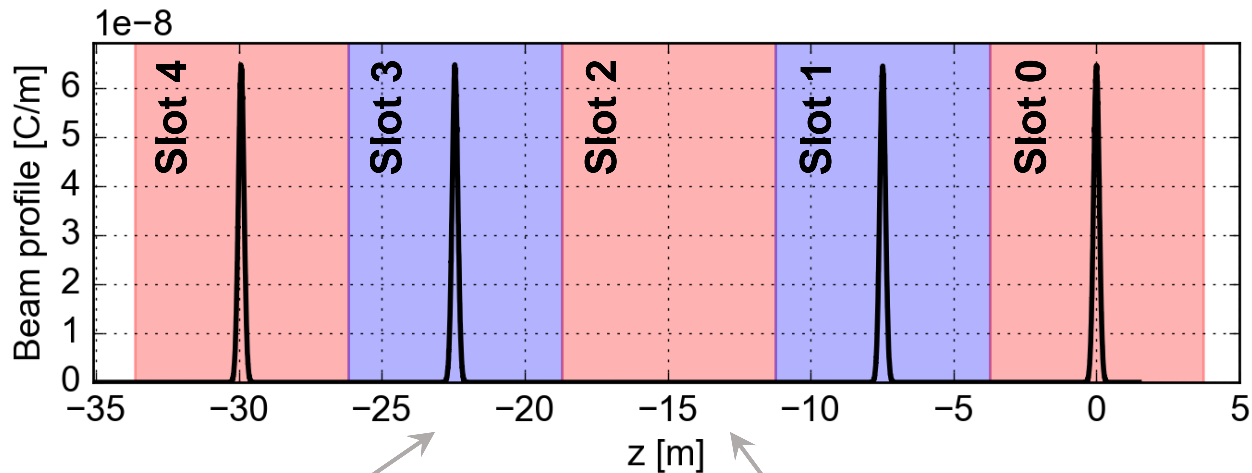


Each bunch slot is made by an integer number of RF buckets

PyPARIS: beam generation and slicing

Empty slots need to be taken into account for the electron dynamics

- **Each slot is a PyHEADTAIL Particles object** with attached a **dictionary with metadata**
  - This includes **flags** defining whether the slot needs to be sub-sliced and whether it interacts with the e-cloud (kicks need to be applied)



bun3.slice_info

```
{'N_bunches_tot_beam': 5,
 'i_bunch': 3,
 'i_turn': 0,
 'interact_with_EC': True,
 'slice_4_EC': True,
 'z_bin_center': -22.439,
 'z_bin_left': -26.179,
 'z_bin_right': -18.699}
```

bun2.slice_info

```
{'N_bunches_tot_beam': 5,
 'i_bunch': 2,
 'i_turn': 0,
 'interact_with_EC': False,
 'slice_4_EC': False,
 'z_bin_center': -14.959,
 'z_bin_left': -18.699,
 'z_bin_right': -11.219}
```

PyPARIS: beam generation and slicing

a_slice.slice_info

```
{'N_slices_tot_bunch': 22,
 'i_slice': 11,
 'interact_with_EC': True,
 'z_bin_center': -7.489,
 'z_bin_left': -7.499,
 'z_bin_right': -7.479,
 'info_parent_bunch’: {
     'N_bunches_tot_beam': 5,
     'i_bunch': 1,
     'i_turn': 0,
     'interact_with_EC': True,
     'slice_4_EC': True,
     'z_bin_center': -7.479,
     'z_bin_left': -11.219,
     'z_bin_right': -3.739},}
```

Written a **slicing tool** that slices bunches (again PyHEADTAIL’s slicing under the hood) and **attaches the required metadata** to each slice object

- **Gaps between bunches** are covered by long slices at head and tail that do not interact with the e-cloud
- **Info of the parent bunch** stored inside each slice

PyPARIS: beam generation and slicing

a_slice.slice_info

```
{'N_slices_tot_bunch': 22,
 'i_slice': 0,
 'interact_with_EC': False,
 'z_bin_center': -5.509,
 'z_bin_left': -7.279,
 'z_bin_right': -3.739,
 'info_parent_bunch': {
     'N_bunches_tot_beam': 5,
     'i_bunch': 1,
     'i_turn': 0,
     'interact_with_EC': True,
     'slice_4_EC': True,
     'z_bin_center': -7.479,
     'z_bin_left': -11.219,
     'z_bin_right': -3.739},}
```

Written a **slicing tool** that slices bunches (again PyHEADTAIL's slicing under the hood) and **attaches the required metadata** to each slice object

- **Gaps between bunches** are covered by long slices at head and tail that do not interact with the e-cloud
- **Info of the parent bunch** stored inside each slice

PyPARIS: beam data saving

Bunch data at each turn are saved by **"misusing" a standard PyHEADTAIL bunch monitor** at the entrance of each ring:

- The **bunch-id** and the **turn number** are attached to the bunch as additional methods. The PyHT bunch monitor is instructed to record them.
- Bunches pass one by one through the monitor and the information is logged 1D arrays
- A **separate file is saved by bunch monitor at entrance of each ring of CPUs**: if we have 4 rings, the first file contains turns [0, 4, 8, 12, …], the second file contains turns [1, 5, 9, 13, …] and so on) → **Data is reshuffled at the post-processing stage**

```
def init_start_ring(self):
    from PyHEADTAIL.monitors.monitors import BunchMonitor
    self.bunch_monitor = BunchMonitor(
        'bunch_monitor_ring%03d'%self.ring_of_CPUs.myring,
        n_stored_turns, {'Comment':'PyHDTL simulation’},
        write_buffer_every=1,
        stats_to_store=['mean_x', 'mean_xp’, ..., 'i_bunch', 'i_turn’])

def perform_bunch_operations_at_start_ring(self, bunch):
    if bunch.macroparticlenumber>0:
        bunch.i_bunch = types.MethodType(
            lambda self: self.slice_info['i_bunch'], bunch)
        bunch.i_turn = types.MethodType(
            lambda self: self.slice_info['i_turn'], bunch)
        self.bunch_monitor.dump(bunch)
```

PyPARIS: data saving

Correct implementation of the parallelization topology checked with a **simple simulation without e-cloud (tracking + ideal feedback)**

Bunch 3

1e−4

x [m]

1e−4

y [m]

Bunch 3

1e−2

Horizontal spectrum

1e−2

Vertical spectrum

Bunch 3

1e−2−2.24e1

z [m]

z spectrum

Expected

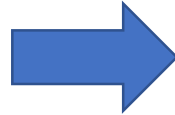
Synchrotron tune

Outline

- **Introduction**
- **Parallelization strategy**
- **Extension of the PyPARIS parallelization layer**
  - Structure, interface and implementation
  - Beam generation and slicing
  - Beam data saving
- **Modifications to the PyECLOUD code**
  - Overall structure
  - Cloud data saving
  - Cloud simulation logics
  - Time discretization
- **Examples of simulation results**
- **Simulation time**
  - Scaling at constant number of bunches
  - Effect of hyperthreading
  - Scaling with number of bunches
- **Summary and next steps**

Modifications to PyECLOUD

At each turn PyECLOUD **needs to perform a full buildup simulation** using the beam distribution **(MacroParticles) received from PyHEADTAIL**



Important changes needed

Boundary conditions:

- We want to **avoid creating a separate version** of PyECLOUD just for these simulations → very bad for future development and maintenance
- Preserve **clean code** structure and good readability, avoid duplications
- Changes should be **backwards compatible** (old buildup and single-bunch instability simulations should work with no changes)

Strategy:

- Define **intermediate milestones**, i.e. different needed features
- At each milestone merge changes:
    - Validate using **PyECLOUD test suite** (strengthened at each step)
    - **Deploy in production** version to verify that there was no impact on real simulation campaign

Changes gradually deployed over four versions: v7.2.0 (April), v7.3.0 (June), v7.4.0 (July), v7.5.0 (August)

Outline

- **Introduction**
- **Parallelization strategy**
- **Extension of the PyPARIS parallelization layer**
  - Structure, interface and implementation
  - Beam generation and slicing
  - Beam data saving
- **Modifications to the PyECLOUD code**
  - Overall structure
  - Cloud data saving
  - Cloud simulation logics
  - Time discretization
- **Examples of simulation results**
- **Simulation time**
  - Scaling at constant number of bunches
  - Effect of hyperthreading
  - Scaling with number of bunches
- **Summary and next steps**

PyECLOUD: structure modification

PyECLOUD Version < 7.0.0

Buildup simulation object	**PyEC4PyHT** object
	Usable an a PyHEADTAIL element
Beam and timing	Custom Code handling beam slicing and fields
MP system	**MP system** (no regen)
Dynamics (e^- track)	**Dynamics** (e^- track)
Impact manager	**Impact manager**
Gas ionization	**Field solver (PIC)**
Photoemission	
Data saver	Kicks to beam particles
Field solver (PIC)	Diagnostics

Buildup simulations are performed by a ***Buildup simulation object*** which includes a set of objects implementing different parts of the simulation

Instability simulations are performed using a ***PyEC4PyHT object***, built using some of the buildup modules and custom python code.

- The electron physics code is recycled
- Still there is some code duplication in the initialization of the objects

→ Using this approach for the coupled-bunch would significantly increase the duplication and make further extensions too heavy…

PyECLOUD: structure modification

PyECLOUD Version ≥ 7.0.0

Buildup simulation object	**PyEC4PyHT** object
	Usable an a PyHEADTAIL element
Beam and timing	Custom Code handling beam slicing and fields
Cloud 1, Cloud 2, Cloud 3, each with: **MP system**, **Dynamics** (e⁻ track), **Impact manager**, **Gas ionization**, **Photoemission**, **Data saver**	**MP system** (no regen)
	Dynamics (e⁻ track)
	Impact manager
	Field solver (PIC)
	Kicks to beam particles
Field solver (PIC)	Diagnostics

Moreover, as of version 7.0.0 the possibility of simulating the buildup using **multiple species** has been implemented (by Lotta) by instantiating **multiple cloud objects**

- Each cloud has its own sets of dedicated objects
- At this stage the PyEC4PyHT class was left unchanged

PyECLOUD: structure modification

PyECLOUD Version ≥ 7.0.0

Buildup simulation object

Beam and timing

MP system

Dynamics (e$^-$ track)

Impact manager

Gas ionization

Photoemission

Data saver

Cloud 1

Cloud 2

Cloud 3

Field solver (PIC)

PyEC4PyHT object

Usable an a PyHEADTAIL element

Custom Code handling beam slicing and fields

MP system (no regen)

Dynamics (e$^-$ track)

Impact manager

Field solver (PIC)

Kicks to beam particles

Diagnostics

This was the status when the coupled-bunch development started

→ Decided to change the code organization

PyECLOUD: structure modification

PyEC4PyHT object now **includes a full buildup simulation object** an not only some of its parts

Buildup simulation object

- **Beam and timing**
- **MP system**
- **Dynamics** (e⁻ track)
- **Impact manager**
- **Gas ionization**
- **Photoemission**
- **Data saver**
- Cloud 1
- Cloud 2
- Cloud 3
- **Field solver (PIC)**

PyEC4PyHT object

- **Buildup simulation** object
  - **Beam and timing**
  - **MP system**
  - **Dynamics** (e⁻ track)
  - **Impact manager**
  - **Gas ionization**
  - **Photoemission**
  - **Data saver**
  - Cloud 1
  - Cloud 2
  - Cloud 3
  - **Field solver (PIC)**
- Code handling beam slicing
- Kicks to beam particles
- Diagnostics

PyECLOUD Version ≥ 7.2.0

PyECLOUD: structure modification

PyEC4PyHT object now **includes a full buildup simulation object** an not only some of its parts

In this way we do have a full e-cloud simulation within a PyEC4PyHT object and we get rid of all code duplication

Beam and timing object is different w.r.t. buildup case:

- Same interface exposed to the buildup simulation object (e.g. method providing electric fields at arbitrary position)
- Predefined map (rigid beam) used for the buildup
- Map coming from PIC of the beam particles for instability simulations (PyEC4PyHT)

With this solution electron simulation far all cases goes trough the same class
→ No duplication at all

PyEC4PyHT object

Buildup simulation object

Code handling beam slicing

Beam and timing

MP system

Dynamics (e^- track)

Impact manager

Gas ionization

Photoemission

Data saver

Cloud 1

Cloud 2

Cloud 3

Field solver (PIC)

Kicks to beam particles

Diagnostics

PyECLOUD Version ≥ 7.2.0

Outline

- Introduction
- Parallelization strategy
- Extension of the PyPARIS parallelization layer
    - Structure, interface and implementation
    - Beam generation and slicing
    - Beam data saving
- **Modifications to the PyECLOUD code**
    - Overall structure
    - **Cloud data saving**
    - Cloud simulation logics
    - Time discretization
- Examples of simulation results
- Simulation time
    - Scaling at constant number of bunches
    - Effect of hyperthreading
    - Scaling with number of bunches
- Summary and next steps

PyECLOUD: saving of buildup simulation results

In a coupled bunch simulation we want to **save all the information on the electron dynamics** that we save in a **buildup simulation** (possibly at each turn)

- **PyEC4PyHT module** used for single bunch simulations **had its own (quite limited) diagnostics** for the electron motion

- **Buildup simulation** object is equipped with a **saver object** that monitors and saves on file the different relevant quantities of the buildup process
  - The module needed to know the number of time steps at the beginning of the simulation (to allocate memory) → not appropriate when slice come one by one from PyHEADTAIL
  - The module was assuming uniform time step → non convenient as it forces to slice the beam and sample the gaps with the same time step (see later…)

→ **Saver module has been restructured** to allow **dynamic memory allocation** and **non-uniform time steps** (it was a good occasion for quite some cleanup)

Restructured saver deployed in **PyECLOUD v7.2.0**
Buildup saver usable in PyEC4PyHT in **PyECLOUD v7.4.0**

Outline

- Introduction
- Parallelization strategy
- Extension of the PyPARIS parallelization layer
  - Structure, interface and implementation
  - Beam generation and slicing
  - Beam data saving
- **Modifications to the PyECLOUD code**
  - Overall structure
  - Cloud data saving
  - Cloud simulation logics
  - Time discretization
- Examples of simulation results
- Simulation time
  - Scaling at constant number of bunches
  - Effect of hyperthreading
  - Scaling with number of bunches
- Summary and next steps

PyECLOUD: cloud simulation logics

a_slice.slice_info

```
{'N_slices_tot_bunch': 22,
 'i_slice': 11,
 'interact_with_EC': True,
 'z_bin_center': -7.489,
 'z_bin_left': -7.499,
 'z_bin_right': -7.479,
 'info_parent_bunch': {
     'N_bunches_tot_beam': 5,
     'i_bunch': 1,
     'i_turn': 0,
     'interact_with_EC': True,
     'slice_4_EC': True,
     'z_bin_center': -7.479,
     'z_bin_left': -11.219,
     'z_bin_right': -3.739},}
```

No command sent from the master to ecloud objects.

All information travels together with the bunches in the slice_info dictionary

If the flag **interact_with_EC is True**:

- **PIC on beam particles** is performed to evaluate the beam field
- Beam **forces applied to cloud MPs**
- Electron **forces are applied to beam MPs**

PyECLOUD: cloud simulation logics

a_slice.slice_info

```
{'N_slices_tot_bunch': 22,
 'i_slice': 0,
 'interact_with_EC': False,
 'z_bin_center': -5.509,
 'z_bin_left': -7.279,
 'z_bin_right': -3.739,
 'info_parent_bunch': {
     'N_bunches_tot_beam': 5,
     'i_bunch': 1,
     'i_turn': 0,
     'interact_with_EC': True,
     'slice_4_EC': True,
     'z_bin_center': -7.479,
     'z_bin_left': -11.219,
     'z_bin_right': -3.739},}
```

No command sent from the master to ecloud objects.

All information travels together with the bunches in the slice_info dictionary

If the flag **interact_with_EC is False**:

- **Beam forces** are **not computed** nor applied
- **Electrons are tracked** taking into account only forces from externally applied magnetic fields and their own space-charge forces

PyECLOUD: cloud simulation logics

a_slice.slice_info

```
{'N_slices_tot_bunch': 22,
 'i_slice': 11,
 'interact_with_EC': True,
 'z_bin_center': -7.489,
 'z_bin_left': -7.499,
 'z_bin_right': -7.479,
 'info_parent_bunch': {
     'N_bunches_tot_beam': 5,
     'i_bunch': 1,
     'i_turn': 0,
     'interact_with_EC': True,
     'slice_4_EC': True,
     'z_bin_center': -7.479,
     'z_bin_left': -11.219,
     'z_bin_right': -3.739},}
```

To work correctly and save buildup data within a PyHEADTAIL simulation, the **buildup module needs to know** (via the beam_and_timing object):

- Time within the turn and current time-step size
- Bunch passage within the turn
- Beam charge line density
- Slice transverse position and r.m.s. size (for primary electron generation)

These info are **prepared by the PyEC4PyHT module based on the slice_info dictionary** attached to the slice and on the macroparticles in the slice

Outline

- **Introduction**
- **Parallelization strategy**
- **Extension of the PyPARIS parallelization layer**
  - Structure, interface and implementation
  - Beam generation and slicing
  - Beam data saving
- **Modifications to the PyECLOUD code**
  - Overall structure
  - Cloud data saving
  - Cloud simulation logics
  - Time discretization
- **Examples of simulation results**
- **Simulation time**
  - Scaling at constant number of bunches
  - Effect of hyperthreading
  - Scaling with number of bunches
- **Summary and next steps**

PyECLOUD: time discretization

PyECLOUD
Version < 7.3.0

Buildup simulation:

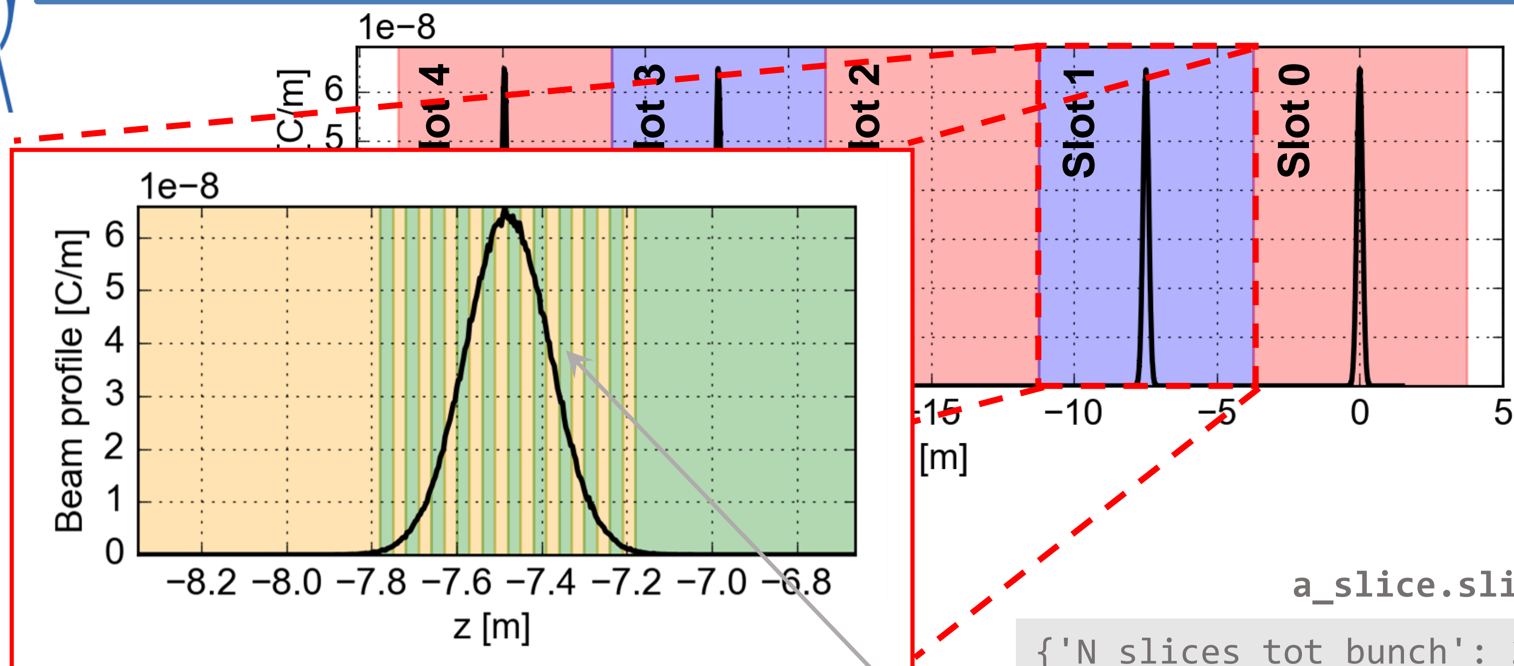
- **Δt:** uniform time-step defined by the used
- **Substeps** used to resolve e^- cyclotron motion
- **Δt_{sc}** larger time step used for electron PIC re-calculation

PyEC4PyHT:

- Time step **defined by slicing** coming from PyHEADTAIL
- **Substeps** used to resolve e^- cyclotron motion
- Electron **PIC re-calculation at each time-step**

→ For **coupled-bunch simulations**, the first approach is too coarse, the second is too heavy…

PyECLOUD: time discretization



The user still provides two parameters:

- **Δt:** time-step required to resolve e^- dynamics (sub-steps can still be used for cyclotron motion)
- **Δt_{sc}** larger time step used for electron PIC re-calculation only for slices that do not interact with the e-cloud

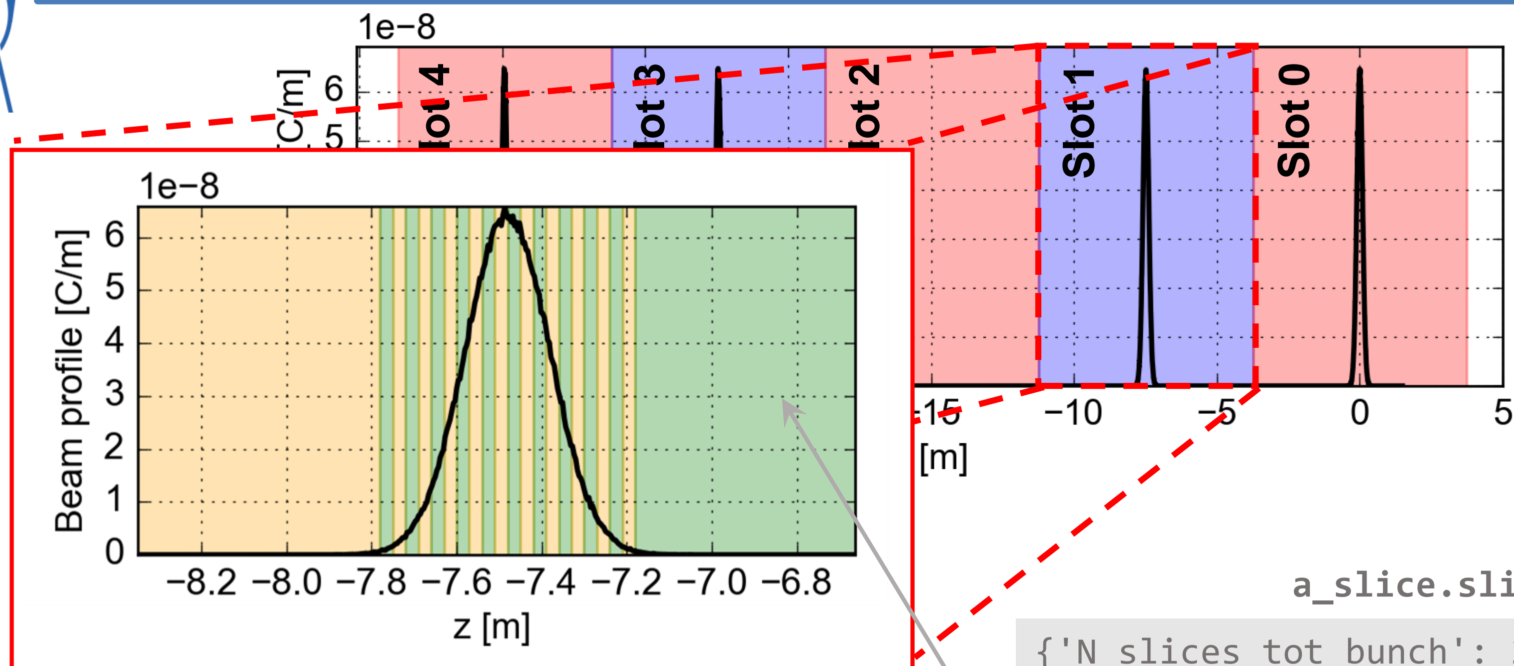
If the beam **slice interacts with the e-cloud**:

- The code **checks that the slice is shorter than Δt** (if not simulation stops and informs the user)
- Electron dynamics **time-step is equal to slice length**
- Recalculation of electron field map **(PIC) forced for each slice** (Δt_{sc} is ignored)

a_slice.slice_info

```
{'N_slices_tot_bunch': 22,
 'i_slice': 11,
 'interact_with_EC': True,
 'z_bin_center': -7.489,
 'z_bin_left': -7.499,
 'z_bin_right': -7.479,
 'info_parent_bunch’: {
     'N_bunches_tot_beam': 5,
     'i_bunch': 1,
     'i_turn': 0,
     'interact_with_EC': True,
     'slice_4_EC': True,
     'z_bin_center': -7.479,
     'z_bin_left': -11.219,
     'z_bin_right': -3.739},}
```

PyECLOUD: time discretization



The user still provides two parameters:

- **Δt:** time-step required to resolve e^- dynamics (sub-steps can still be used for cyclotron motion)
- **Δt_{sc}** larger time step used for electron PIC re-calculation only for slices that do not interact with the e-cloud

If the **beam slice does not interact with the e-cloud:**

- If the **slice is longer that Δt**, the slice gets **divided in several time steps** just smaller than Δt.
- Recalculation of electron field map **(PIC) happens every Δt_{sc}**

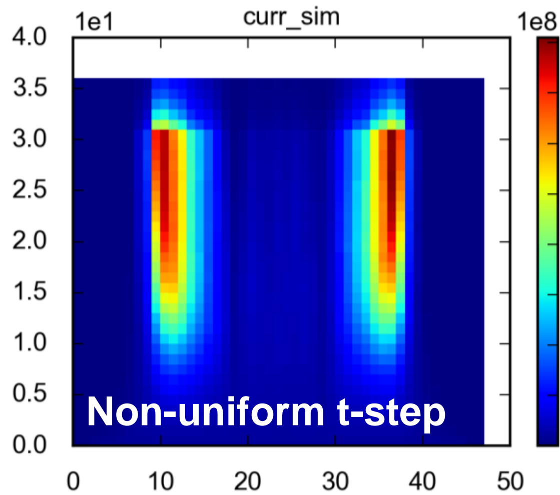
a_slice.slice_info

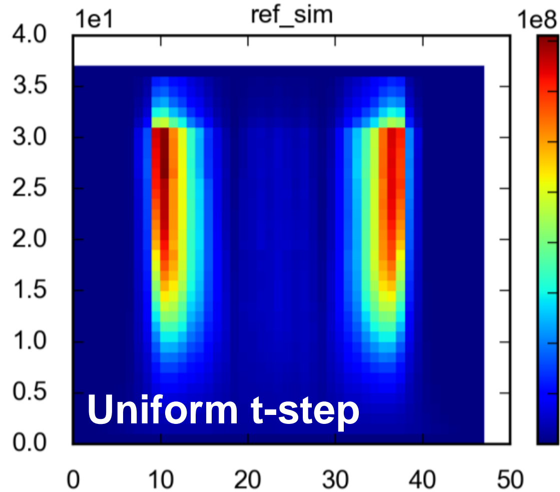
```
{'N_slices_tot_bunch': 22,
 'i_slice': 0,
 'interact_with_EC’: False,
 'z_bin_center': -5.509,
 'z_bin_left': -7.279,
 'z_bin_right': -3.739,
 'info_parent_bunch’: {
     'N_bunches_tot_beam': 5,
     'i_bunch': 1,
     'i_turn': 0,
     'interact_with_EC': True,
     'slice_4_EC': True,
     'z_bin_center': -7.479,
     'z_bin_left': -11.219,
     'z_bin_right': -3.739},}
```

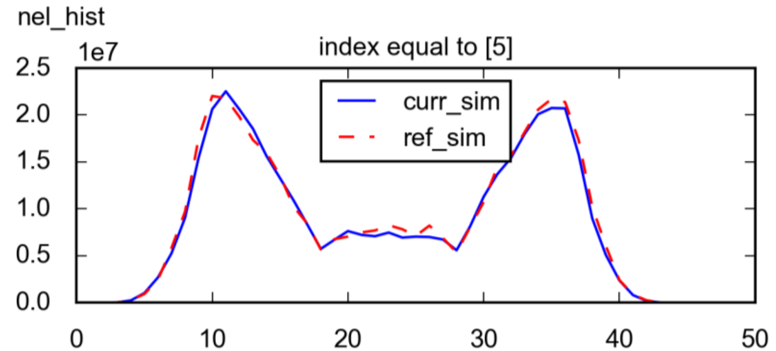
PyECLOUD: time discretization

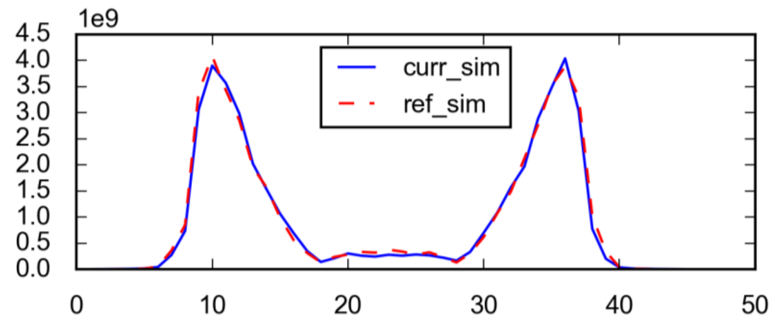
To be compatible with the model described before, the **cloud simulator needs to learn how to handle a non uniform time-step**

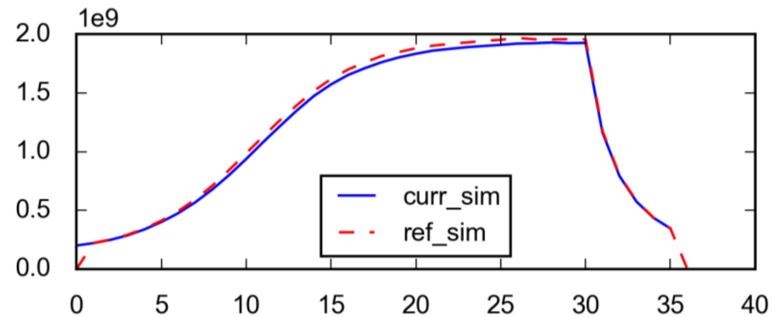
→ Implemented and deployed in **PyECLOUD v7.3.0**











PyECLOUD: reinitialization

At the **end of each turn** the buildup simulation **needs to be re-initialized**:

- Buildups simulation object was not conceived for that
- Constructing a new object at each turn would be too heavy

→ Needed to introduce in the buildup object the **memory of its initial state** and the capability of **reinitializing** itself when it sees the first slice of the first bunch

a_slice.slice_info

```
{'N_slices_tot_bunch': 22,
 'i_slice': 11,
 'interact_with_EC': True,
 'z_bin_center': -7.489,
 'z_bin_left': -7.499,
 'z_bin_right': -7.479,
 'info_parent_bunch': {
     'N_bunches_tot_beam'
     'i_bunch': 1
     'i_turn'
                           rue,
                       rue,
         center': -7.479,
       _bin_left': -11.219,
     'z_bin_right': -3.739},}
```

→ Implemented and deployed in **PyECLOUD v7.5.0**

PyECLOUD: a global view on applied changes

As of version 7.5.0, PyECLOUD has all the functionalities for the simulation of coupled-bunch instabilities (changes applied between v7.1.2 and v.7.5.0)

- The new functionalities were added **without affecting too much the code size**, in fact it was an occasion for some reorganization and clean-up
- The **test suite was significantly strengthened** (added ~1000 lines) to validate new features and stress modules that were modified

Statistics from “git diff --numstats”

Module	Lines removed(*)	Lines added(*)	Total lines v7.5.0
pyecloud_saver.py	337	496	620
PyEC4PyHT.py	172	310	599
PyEC4PyHT_fastion.py	263	0	263
buildup_simulation.py	91	146	226
init.py	13	41	420
MP_system.py	18	35	525
beam_and_timing.py	5	26	383
Total	899	1054	3036

Only modules in which changes were made are listed here

(*) Between v7.1.2 and v7.5.0

Functionality incorporated in PyECPyHT module (thanks Lotta!)

PyECLOUD: some checks

Results of **simulations with PyHEADTAIL beam** (made of macroparticles) are **fully consistent with corresponding standard buildup simulation** (rigid analytical beam distribution)

PyECLOUD: some checks

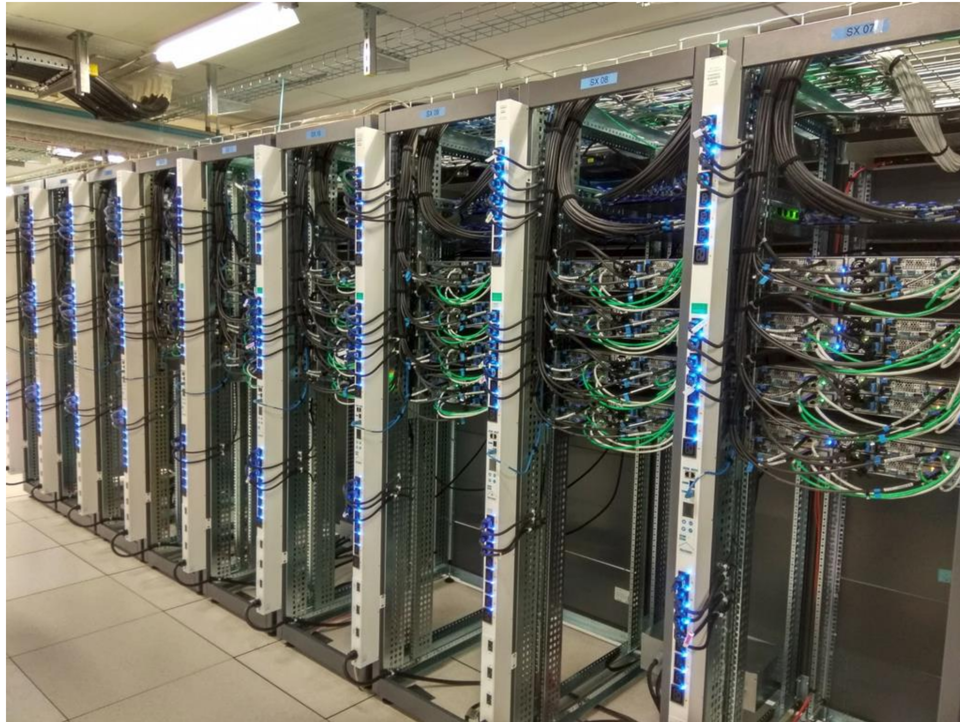
Response to a transverse beam displacement along the bunch train is clearly visible in the electron dynamics

Outline

- Introduction
- Parallelization strategy
- Extension of the PyPARIS parallelization layer
  - Structure, interface and implementation
  - Beam generation and slicing
  - Beam data saving
- Modifications to the PyECLOUD code
  - Overall structure
  - Cloud data saving
  - Cloud simulation logics
  - Time discretization
- **Examples of simulation results**
- Simulation time
  - Scaling at constant number of bunches
  - Effect of hyperthreading
  - Scaling with number of bunches
- Summary and next steps

CERN High Performance Computing (HPC) cluster

- First tests (mainly debugging) conducted on a dedicated node at **INFN-CNAF cluster**
- Larger studies launched on **CERN HPC cluster** (managed by IT department)
    - **Two clusters** available for accelerator studies:
        - **“BE cluster”**: ~140 nodes, 20 CPU-cores/node → 2800 cores
        - **“Batch cluster”**: ~100 nodes, 16 CPU-cores/node → 1600 cores
- Equipped with **low-latency network** (InfiniBand on BE cluster) for fast communication among nodes using the MPI protocol

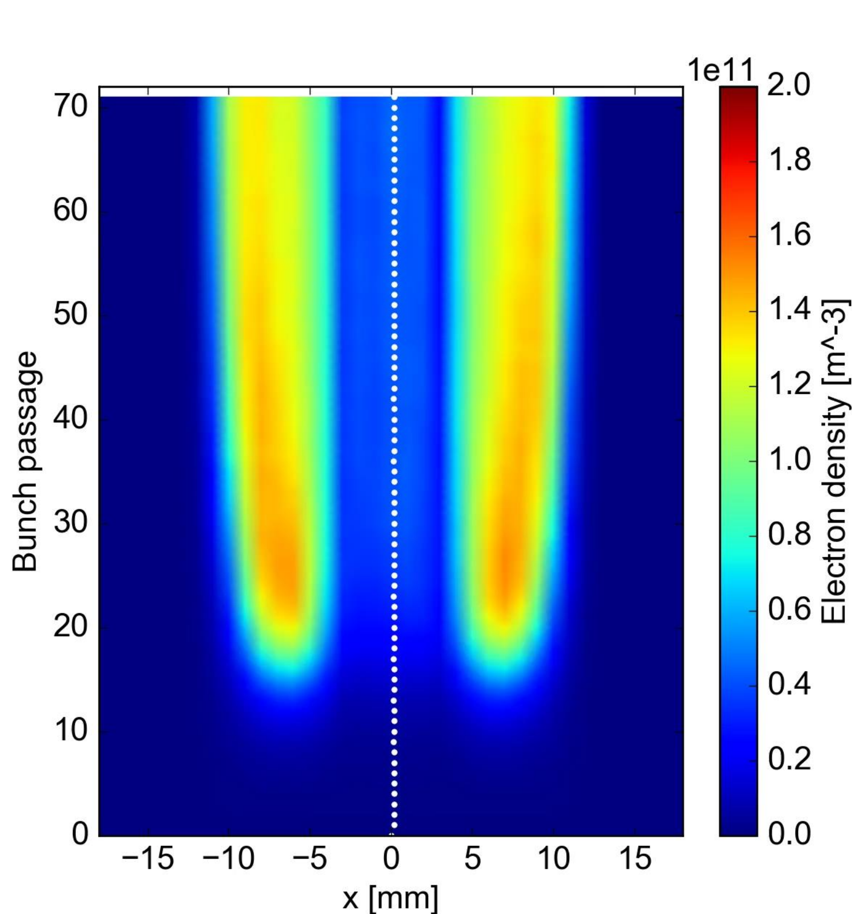


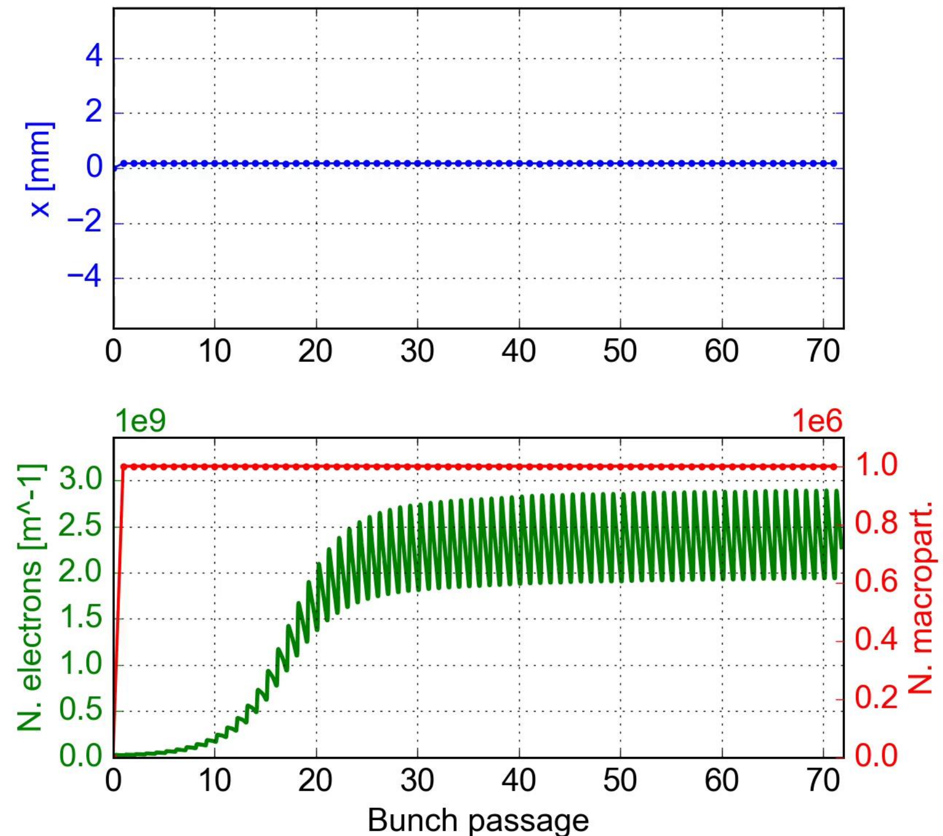
https://cern.service-now.com/service-portal/faq.do?se=High-Performance-Computing

Example of simulation results 1/2

- **Simulation scenario:** LHC, 450 GeV, **72 bunches**, e-cloud in the dipole magnets
- **Numerical parameters:** 10^6 macroparticles per proton bunch, 2.5×10^5 macroparticles at each e-cloud interaction, **360 CPU cores**
- In total 72×10^6 beam macroparticles and 90×10^6 electron macroparticles

Turn 0





Example of simulation results 2/2

- **Simulation scenario:** LHC, 450 GeV, **288 bunches**, e-cloud in the dipole magnets
- **Numerical parameters:** 10^6 macroparticles per proton bunch, 2.5×10^5 macroparticles at each e-cloud interaction, **1200 CPU cores**
- In total 288×10^6 beam macroparticles and 300×10^6 electron macroparticles

Turn 0

Bunch passage
x [mm]
1e11
Electron density [m^-3]
x [mm]
1e9
1e6
N. electrons [m^-1]
N. macropart.
Bunch passage

Outline

- Introduction
- Parallelization strategy
- Extension of the PyPARIS parallelization layer
  - Structure, interface and implementation
  - Beam generation and slicing
  - Beam data saving
- Modifications to the PyECLOUD code
  - Overall structure
  - Cloud data saving
  - Cloud simulation logics
  - Time discretization
- Examples of simulation results
- **Simulation time**
  - Scaling at constant number of bunches
  - Effect of hyperthreading
  - Scaling with number of bunches
- Summary and next steps

Impact of the parallelization

All three parallelization steps allow **reducing the computation time by using a larger number of CPU-cores**

Simulations performed on the CERN HPC cluster

Time per 1000 turns [days]

1000
100
10
1

1
10
100

N. CPU cores

Step 1:
Parallelization over segments

Step 2:
Parallelization over turns

Step 3:
Shorter bunch slots

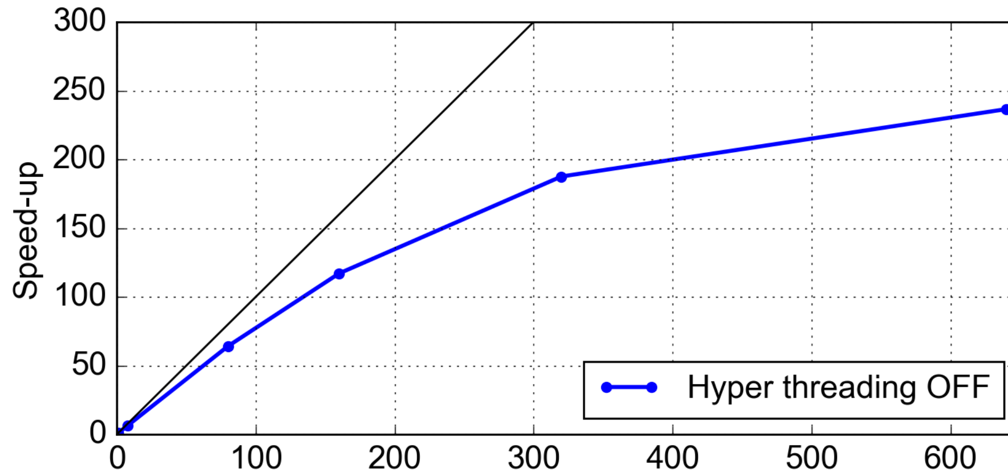
Test case for scaling studies:

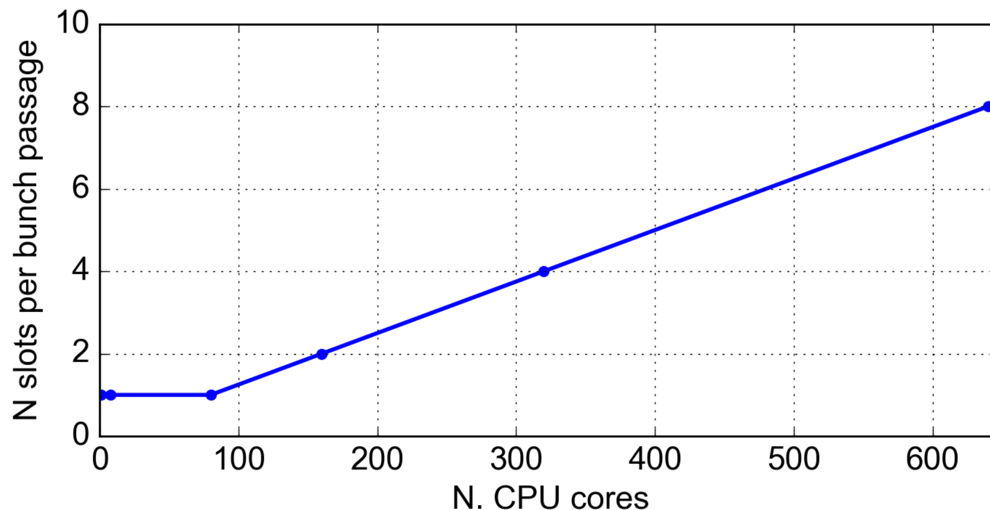
- bunch spacing 20 ns
- 80 bunches,
- 8 EC interactions / turn

Impact of the parallelization

We focus on the last parallelization step:

- Using **shorter bunch slots** allows using more cores ang gives a visible gain
- Speed-up is **significantly less than linear**, likely due to **asynchronous operations on the clouds** (e.g. regenerations) that keep other cores waiting





Test case for scaling studies:

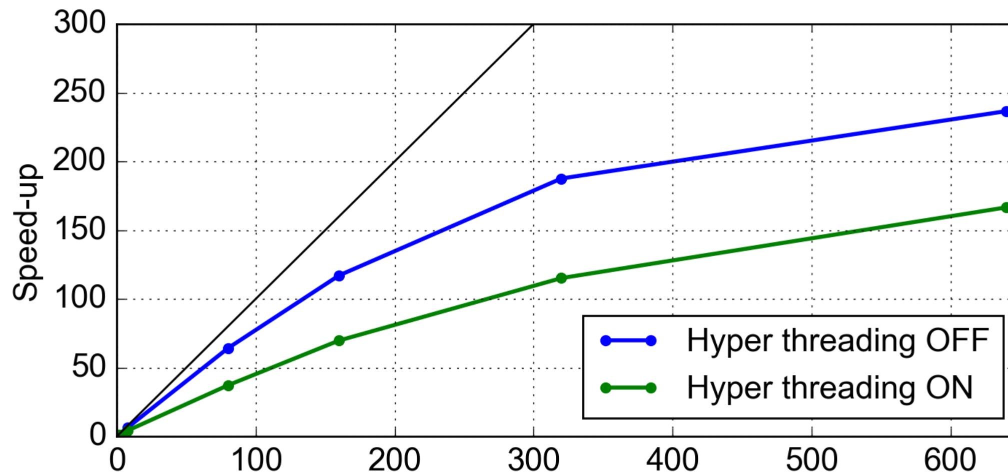
- bunch spacing 20 ns
- 80 bunches
- 8 EC interactions / turn

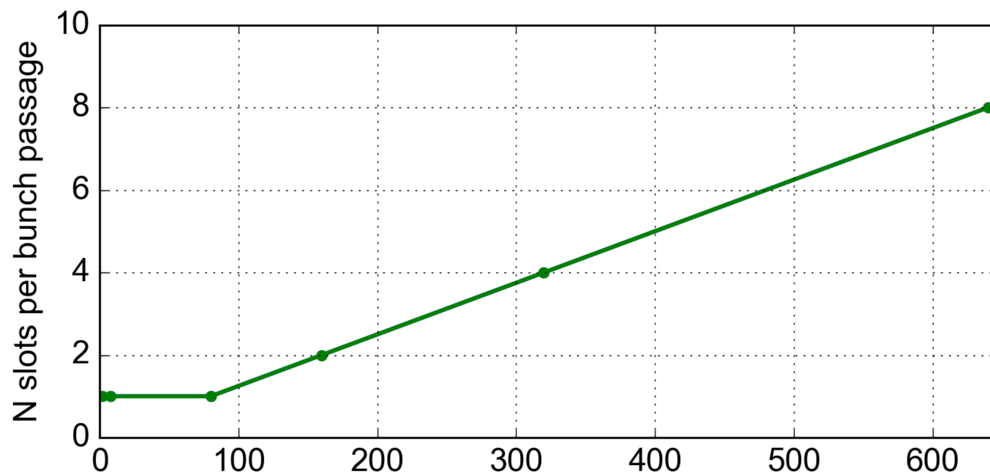
Effect of Hyper-Threading

CERN HPC cluster allows the user to decide whether to exploit **HyperThreading** or not

- Expected loss of performance is observed with HyperThreading ON (using x2 less physical CPU cores), but **performance loss is less than a factor of 2**

  → **Resources of waiting CPU cores can be used by busy ones**





Test case for scaling studies:

- bunch spacing 20 ns
- 80 bunches
- 8 EC interactions / turn

For all cases the time with 1 CPU core is measured with HT off

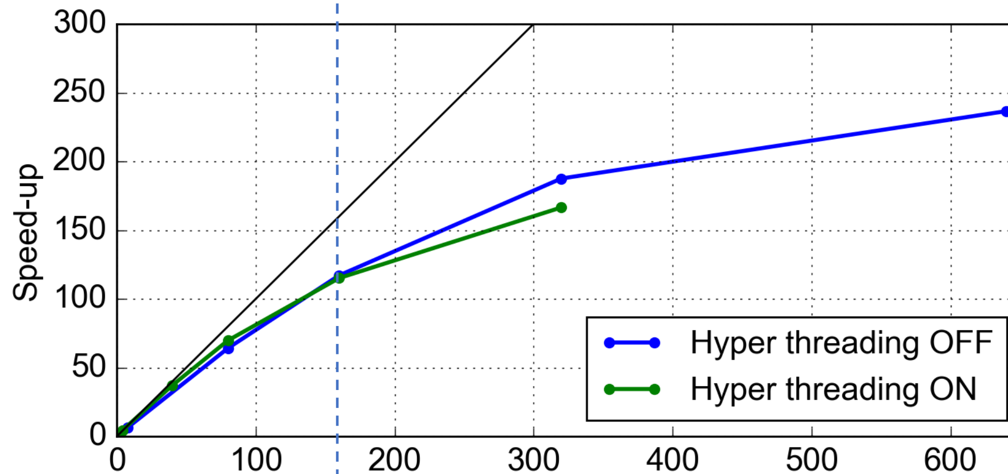
Virtual cores in case HT is ON

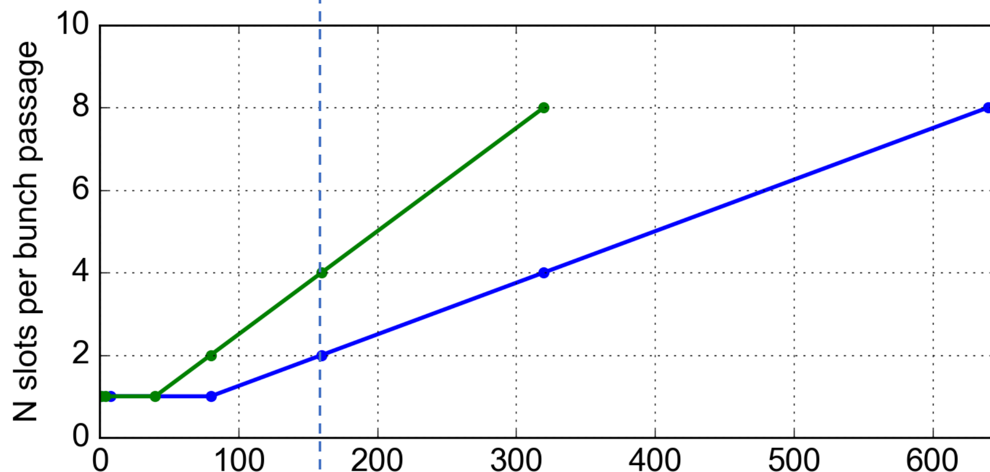
Effect of Hyper-Threading

We can plot the data **as a function of real CPU cores that are really used**

→ The two modes of operation are practically equivalent

A good working point for larger simulations:
Slots of 5 ns, HT ON





Real cores

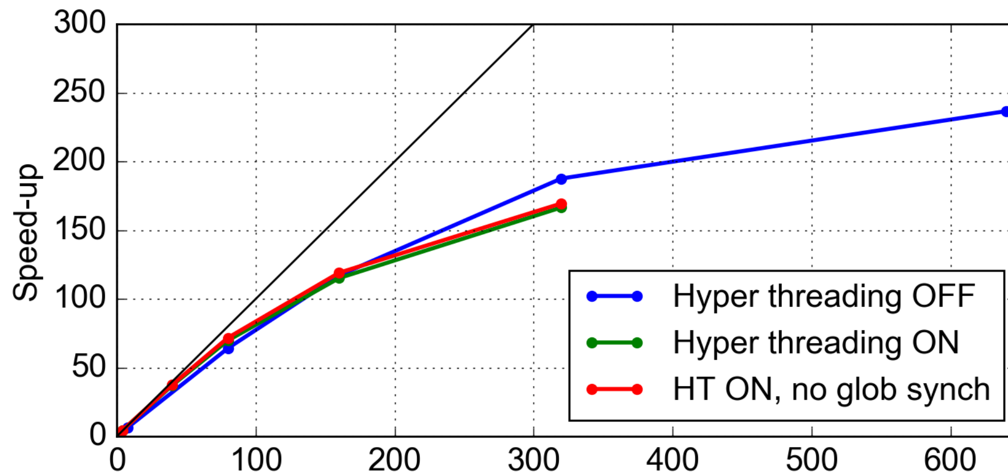
Test case for scaling studies:

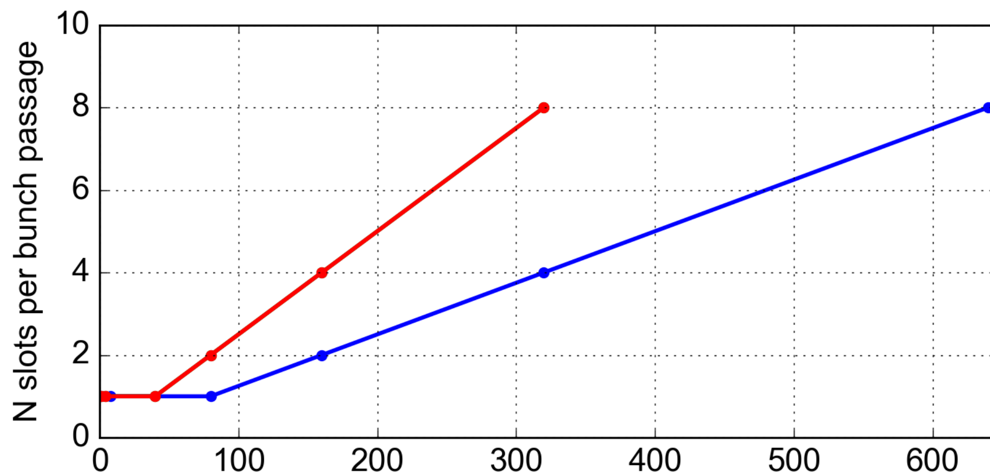
- bunch spacing 20 ns
- 80 bunches,
- 8 EC interactions / turn

For all cases the time with 1 CPU core is measured with HT off

Effect of Hyper-Threading

- To avoid mistakes during development a **global synchronization** is performed among all cores at each iteration, not strictly needed
- Interesting to observe that **this has absolutely no effect** (time of single iterations changes, but average stays exactly the same)





Real cores

Test case for scaling studies:

- bunch spacing 20 ns
- 80 bunches
- 8 EC interactions / turn

For all cases the time with 1 CPU core is measured with HT off

Scaling with number of bunches

- When **increasing the number of bunches**, the number of CPU cores can be increased accordingly → **simulation time stays roughly constant**

Slots of 5 ns, HT on

Time per 1000 turns [days]

0 1 2 3 4 5 6

0 50 100 150 200 250 300 350

N. CPU cores

0 200 400 600 800 1000 1200

0 50 100 150 200 250 300 350

N. bunches

Conclusion and next steps

PyPARIS and PyECLOUD have been updated to simulate coupled-bunch instabilities driven by electron cloud exploiting MPI parallelization

Next steps (not necessarily in this order):

- Implement diagnostics for intra-bunch motion
- Implement distributed bunch generation
- Introduce non-ideal transverse feedback (if needed)
- Assess possibility of further performance enhancement (collaboration with IT)
- Perform first real studies → compare against simplified models

How to transfer these dictionaries

```
pss = pickle.dumps(sinfo, protocol=2)

# Pad to have a multiple of 8 bytes
s1arr = np.frombuffer(pss, dtype='S1')
ll = len(s1arr)
s1arr_padded = np.concatenate((s1arr, np.zeros(8-ll%8, dtype='S1')))

# Cast to array of floats
f8arr = np.frombuffer(s1arr_padded, dtype=np.float64)
sinfo_float_buf = np.concatenate((np.array([ll], dtype=np.float64), f8arr))
```