Many thanks to all (past and present) members of the Electron Cloud Working Group for their invaluable help in the development of the code!
• Introduction
• PyE CLOUD simulations:
  • e-cloud buildup simulations
  • Simulations of e-cloud effects on beam dynamics (stability)
• Main development milestones
• Technical information
• The Particle-In-Cell engine: PyPIC
• Parallelization strategy for PyE CLOUD-PyHEADTAIL simulations (PyPARIS)
• Status and needs for computing resources
• Directions for future development
• **Introduction**
  • PyECLoud simulations:
    • e-cloud buildup simulations
    • Simulations of e-cloud effects on beam dynamics (stability)
  • Main development milestones
  • Technical information
  • The Particle-In-Cell engine: PyPIC
  • Parallelization strategy for PyECLoud-PyHEADTAIL simulations (PyPARIS)
  • Status and needs for computing resources
  • Directions for future development
Secondary Electron Emission can drive an avalanche multiplication effect filling the beam chamber with an electron cloud.

Trailing bunches of the train interact with a dense e-cloud:

- Transverse instabilities
- Transverse emittance blow-up
- Particle losses

e-cloud induces other unwanted effects like:

- Heat load on the beam chambers
- Vacuum degradation
**PyECloud** is the simulation code for the simulation of electron cloud effects

- **2D MacroParticle (MP) code**
- Developed and maintained at CERN since 2011 following the legacy of the **ECloud** code (F. Zimmermann et al., 1997...)

It can be used to *simulate different effects*:

- **Buildup simulations**: in stand-alone mode for the simulation of the e-cloud formation at a certain section of an accelerator
- **Effects on the beam dynamics** (e.g. instabilities, e-cloud detuning) in combination with the **PyHeadTail** code
- **Fast beam-ion instabilities** in combination with the **PyHeadTail** code (not covered in this presentation, described by L. Mether, **ABP-CWG meeting 24 Nov 2016**)

---

*Note: The text above was modified for clarity and to improve readability.*
• Introduction

• **PyE CLOUD simulations:**
  - e-cloud buildup simulations
  - Simulations of e-cloud effects on beam dynamics (stability)

• Main development milestones

• Technical information

• The Particle-In-Cell engine: PyPIC

• Parallelization strategy for PyE CLOUD-PyHEADTAIL simulations (PyPARIS)

• Status and needs for computing resources

• Directions for future development
Simulation of the **electron multipacting** process:

- The **beam is rigid** (assigned charge distribution)
- Electrons are tracked and **secondary emission** models are applied when particles impact the wall
Evaluate the electric field of beam at each MP location

Evaluate the $e^-$ space charge electric field

Compute MP motion (t→t+Δt)

Detect impacts and generate secondaries
“Primary electrons” are generated by:

- Ionization of the residual gas in the vacuum chamber
- Photoemission from the chamber’s walls due to synchrotron radiation
Evaluate the electric field of beam at each MP location

Evaluate the e⁻ space charge electric field

Compute MP motion \((t \rightarrow t + \Delta t)\)

Detect impacts and generate secondaries

\[ t = t + \Delta t \]

**Beam field** evaluated on a fixed grid and interpolated at each MP location

Several options implemented:

- **Bassetti – Erskine** formula with image charges (elliptic chambers)
- **Finite Difference Poisson Solver** (arbitrary shaped chambers)
- Field map imported from file

**Multiple beams** can be simulated (e.g. LHC triplets)
Evaluate the electric field of beam at each MP location

Evaluate the e⁻ space charge electric field

Compute MP motion (t→t+Δt)

Detect impacts and generate secondaries

t=t+Δt

Effect of the electrons on themselves evaluated by Particle In Cell (PIC) to solve the Poisson equation

• Arbitrary shaped chambers can be simulated

• Shortley-Weller method for refined approximation of curved boundaries on a rectangular grid

• Important speed-up achieved through C-implemented KLU factorization algorithm (cython)

• Available as a stand alone module (PyPIC) for different usages (e.g. space charge module for PyHEADTAIL)
Evaluate the electric field of beam at each MP location

Evaluate the \( e^- \) space charge electric field

Compute MP motion \((t \rightarrow t+\Delta t)\)

Detect impacts and generate secondaries

\[ t = t + \Delta t \]

The **dynamics equation** is integrated to update MP position and momentum. Two algorithms implemented:

- **Semi-analytic** algorithm inherited from the ECLOUD code (effective for dipole and field free regions)
- **Boris algorithm** with substeps (phase space volume preserving) → proved to be necessary for quadrupoles and combined function magnets
  - Speedup using **f2py and cython**
  - **Dipole and quadrupole** natively available, **arbitrary B-field maps** can be loaded from file
Evaluate the electric field of beam at each MP location

Evaluate the $e^-$ space charge electric field

Compute MP motion ($t \rightarrow t + \Delta t$)

Detect impacts and generate secondaries

$t = t + \Delta t$

- When a MP hits the wall Secondary Electron Emission models are employed to generate charge, energy and angle of the emitted charge
  - Secondary Electron Yield (SEY) can be non-uniform on the chamber surface
- According to the number of emitted electrons, the MP can be simply rescaled or new MPs can be generated
  - Part of a quite complex MP size management...
Peculiarity of these simulations: the **number of electrons grows exponentially**

- The **size and number of the MPs** needs to be adapted dynamically during the simulation
- All generation processes need to know an instantaneous reference MP size
- Periodic “cleanings” and “regenerations” need to be performed
e-cloud buildup simulations – CERN usage

Used for **studies on many machines** (e.g. PS, SPS, LHC, CLIC-DR, FCC-ee, FCC-hh) to:

- Identify **surface (SEY) requirements** on newly installed elements (multipacting thresholds)
- Compute **expected heat loads** on cryogenic components
- Compute **expected electron flux on beam chamber walls** (for vacuum estimates)
- **Reconstruct SEY from measured heat loads** (measured bunch-by-bunch properties can be loaded for simulations)
- Simulate **electron cloud detectors** (e.g. shielded pickup)
• Introduction

• **PyE CLOUD simulations:**
  • e-cloud buildup simulations
  • Simulations of e-cloud effects on beam dynamics (stability)

• Main development milestones

• Technical information

• The Particle-In-Cell engine: PyPIC

• Parallelization strategy for PyE CLOUD-PyHEADTAIL simulations (PyPARIS)

• Status and needs for computing resources

• Directions for future development
Electrons are attracted by the proton bunches and ‘fly” through it exerting
**significant electromagnetic forces on beam particles** $\rightarrow$ impact on beam dynamics:

- **Coherent effects:** tune shift, transverse instabilities
- **Incoherent effects:** tune spreads, non linear forces $\rightarrow$ losses, emittance blow-up

Electron density during a bunch passage

To simulate this kind of effects, **we need to track both the e-cloud particles and the beam particles**
Since 2014, we dropped the approach of having separate tools for buildup and instability

- **Use PyECloud** also simulate the interaction beam/ecloud **within PyHEADTAIL**

  → Possible thanks to the highly modular structure of the two codes (object oriented)

---

**Legend:**
- From PyHEADTAIL
- From PyECloud
- Developed ad hoc

---

**PyHEADTAIL**

- Transverse tracking → with $Q'$, octupoles etc.
- Longitudinal tracking
- Transverse feedback
- Impedances
- Space charge
- ...

**PyECloud**

- **Initial e- distribution** (from PyECloud buildup sim.)

**PyHEADTAIL slicer**

- **For each slice**
  - Generate seed $e^-$
  - Evaluate beam slice electric field (Particle in Cell)
  - Evaluate the $e^-$ electric field (Particle in Cell)
  - Apply kick on the beam particles
  - Compute $e^-$ motion ($t \rightarrow t + \Delta t$) (possibly with substeps)
  - Detect impacts and generate secondaries

**PyHEADTAIL bunch**

- Transverse tracking → with $Q'$, octupoles etc.
- Longitudinal tracking
- Transverse feedback
- Impedances
- Space charge
- ...

---

**Legend:**
- From PyHEADTAIL
- From PyECloud
- Developed ad hoc
Since 2014, we dropped the approach of having separate tools for buildup and instability

- **Use PyECloud** also simulate the interaction beam/ecloud within PyHEADTAIL
  - Possible thanks to the highly modular structure of the two codes (object oriented)
  - Using PyHEADTAIL philosophy → the author writes a script

In this framework *we could significantly improve* our modeling of e-cloud effects on beam dynamics:

- Accurate **tracking** of the electrons in dipoles and quadrupoles (Boris push)
- **Multipacting** can be included
- Realistic **boundary conditions**
- Possibility to record the e-cloud pinch to evaluate **detuning along the bunch** (footprint)
- Combine **different e-clouds** in the same simulation (e.g. dipoles+quadrupoles)
- Non uniform **optics functions** (e.g. LHC Insertion Region), off-centered beams...
- We could generalize it for **fast-ion instabilities**

And of course **more efficient in terms of development and maintainance** w.r.t. having two separate tools with largely duplicated code
e-cloud - beam dynamics simulations: CERN usage

Used up to now especially for SPS and LHC to estimate:

- **Stability** thresholds
- **Tune shifts and tune spreads**

Including **several effects:**

- **e-cloud in different machine elements** (dipoles and/or quadrupoles and/or drifts)
- And all **beam dynamics** modeling available in PyHEADTAIL:
  - Non uniform optics
  - Transverse damper
  - Chromaticity
  - Octupoles
  - RF quadrupole and Q” (ongoing work by Michael)
• Introduction

• PyE CLOUD simulations:
  • e-cloud buildup simulations
  • Simulations of e-cloud effects on beam dynamics (stability)

• Main development milestones

• Technical information

• The Particle-In-Cell engine: PyPIC

• Parallelization strategy for PyE CLOUD-PyHEADTAIL simulations (PyPARIS)

• Status and needs for computing resources

• Directions for future development
Main development milestones

Roughly one main release per year:

- **PyECLOUD 1 (2012)**
  
  **Basic features**: buildup only, elliptical chamber, only dipole field, not object oriented

- **PyECLOUD 2 (2013)**
  
  **Object oriented reorganization, generic magnetic field map** (semi-analytic), arbitrary chamber shape, non-uniform SEY (simulation of e-cloud detectors)

- **PyECLOUD 3 (2014)**
  
  **Multiple beams** (simulation of LHC common regions), improved **electron tracking** (Boris) → quartupole and combined function magnets, improved **boundary conditions** (Shortely-Weller), **performance optimization** (KLUB, cython), Particle-In-Cell in a separate module (**PyPIC**, see later)

- **PyECLOUD 4 (2015)**
  
  **Interface with PyHEADTAIL** for beam dynamics simulations (PyEC4PyHT), “weak-strong” mode for footprint evaluation, **non-convex boundary conditions**

- **PyECLOUD 5 (2016)**
  
  **Parallelization** for beam dynamics simulation (via PyPARIS, see later), introduced manual, examples and **test suite**

- **PyECLOUD 6 (2017)**
  
  - Simulation of **Fast Beam-Ion Instabilities**, most of the code generalized to handle arbitrary ions instead of electrons
• Introduction

• PyECloud simulations:
  • e-cloud buildup simulations
  • Simulations of e-cloud effects on beam dynamics (stability)

• Main development milestones

• Technical information
  • The Particle-In-Cell engine: PyPIC
  • Parallelization strategy for PyECloud-PyHEADTAIL simulations (PyPARIS)
  • Status and needs for computing resources
  • Directions for future development
PyECloud is **open source**, available at [https://github.com/PyCOMPLETE/PyECloud](https://github.com/PyCOMPLETE/PyECloud)

**Programming languages:**
- Mostly **Python** (95%)
- **FORTRAN** (linked via f2py) and **C** (linked via cython) for computationally intensive parts

**Programming paradigm:**
- **Object oriented** (makes it so much easier! e.g. to implement two beams, different chamber shaped chambers, different trackers...)

**Parallelization strategy:**
- **Buildup** simulations are **serial**
- **PyPARIS parallelization layer** developed for **beam stability simulations** (see later)

**Operative systems:**
- **Linux** (tested extensively on Ubuntu and SLC 5 and 6)

**Other prerequisites:**
- **Python** 2.7+
- Open source **python libraries**: numpy (including fpy), scipy, matplotlib, cython
- Other CERN **developed library**: PyPIC (open source)
Github proved to be very effective to manage the collaborative development

<table>
<thead>
<tr>
<th>Owners</th>
<th>Feb</th>
<th>Mar</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>28</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PyCOMPLETE</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>jsopouse</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Available documentation and resources:

- **Reference manual** (documenting input and output)
- **Git Wiki pages** including:
  - Instructions for installation
  - A set of references on physics models and implementations
- A set of **examples and test scripts**:  
  - Used also as validation before releasing new versions (very extremely useful)

Just last week running the PyECLCoud tests we spotted a little nasty bug introduced by PyHEADTAIL 1.11
PyE CLOUD and its Python friends

PyE CLOUD is **integrated** with other python tools developed at CERN

Toolbox that can be setup with one single command:

- **Beam dynamics**
- **Sparse linear system solution**
- **Parallel computing**
- **Particle In Cell solvers**
- **Frequency analysis**
• Introduction

• PyE Cloud simulations:
  • e-cloud buildup simulations
  • Simulations of e-cloud effects on beam dynamics (stability)

• Main development milestones

• Technical information

• **The Particle-In-Cell engine: PyPIC**
  • Parallelization strategy for PyE Cloud-PyHEADTAIL simulations (PyPARIS)
  • Status and needs for computing resources
  • Directions for future development
A key component of the e-cloud simulator is the **Particle In Cell (PIC) Poisson solver**

Initially PyECLoud included a simple Particle-In-Cell solver

→ We decided to **reorganize** our Particle In Cell (PIC) Poisson solvers

→ We wrote a **Python library (PyPIC)** including different PIC solvers having the same interface which can be used as **plug-in modules for PyECLoud** but also for other applications (e.g. space charge, beam-beam)

PyPIC is now available on the **PyCOMPLETE git repository**: [https://github.com/PyCOMPLETE/PyPIC/](https://github.com/PyCOMPLETE/PyPIC/)
PyPIC includes the following solvers:

- Open boundary FFT
- Perfect Electric Conductor (PEC) rectangular FFT
- PEC arbitrarily shaped boundary – Finite Differences, staircase approx. of curved boundaries
- PEC arbitrarily shaped boundary – Finite Differences, Shortley-Weller method for curved boundaries (more accurate)
- Fast Polar Poisson solver (thanks Xavier!)
- GPU FFT solvers (thanks Adrian!)

**Multigrid mode** introduced since PyPIC 2.0 → **several grids** with different cell size can be **nested**, for example to have increased resolution only where needed (e.g. at the beam location)

Some reference with more details at: [https://github.com/PyCOMPLETE/PyPIC/wiki](https://github.com/PyCOMPLETE/PyPIC/wiki)
• Introduction
• PyECLoud simulations:
  • e-cloud buildup simulations
  • Simulations of e-cloud effects on beam dynamics (stability)
• Main development milestones
• Technical information
• The Particle-In-Cell engine: PyPIC
• **Parallelization strategy for PyECLoud-PyHEADTAIL simulations (PyPARIS)**
• Status and needs for computing resources
• Directions for future development
Intability simulations (especially LHC at ~7 TeV) are **intrinsically very heavy**

**Multi-scale problem:**

- **In space:** small beam (~100 μm) in a big chamber (4 cm)
- **In time:** 1 ns for the e⁻ motion, 1 to 10 s for instability development

Recent work on **performance improvement**:

- Introduced **multi-grid** Particle in Cell solver (as already discussed)
- Exploit **parallel computing** through a new parallelization layer (PyPARIS)
• We launch a **set of processes** logically organized in a **ring structure**
• Each **process takes care of the bunch interaction with the corresponding part of the ring** (an octant in this case)

More details at: [https://github.com/PyCOMPLETE/PyPARIS/wiki](https://github.com/PyCOMPLETE/PyPARIS/wiki)
Parallelization strategy

- One of the processes, the “master”, will have special duties.
- At the beginning of the simulation the master generates the coordinates of the simulated bunch.

On the master process:

To be treated:

Already treated:

More details at: [https://github.com/PyCOMPLETE/PyPARIS/wiki](https://github.com/PyCOMPLETE/PyPARIS/wiki)
Parallelization strategy

- The **bunch is sliced** up by the master...

On the master process:

To be treated: 10 9 8 7 6 5 4 3 2 1

Already treated:

More details at: [https://github.com/PyCOMPLETE/PyPARIS/wiki](https://github.com/PyCOMPLETE/PyPARIS/wiki)
• The **bunch is sliced** up by the master...
• ... and the slices are made to **circulate around the ring** for transverse tracking and e-cloud interactions

**On the master process:**

To be treated: 10, 9, 8, 7, 6, 5, 4, 3, 2

Already treated: 1

More details at: [https://github.com/PyCOMPLETE/PyPARIS/wiki](https://github.com/PyCOMPLETE/PyPARIS/wiki)
Parallelization strategy

- The **bunch is sliced** up by the master...
- ... and the slices are made to **circulate around the ring** for transverse tracking and e-cloud interactions

**On the master process:**

To be treated: 10 9 8 7 6 5 4 3

Already treated:

More details at: [https://github.com/PyCOMPLETE/PyPARIS/wiki](https://github.com/PyCOMPLETE/PyPARIS/wiki)
Parallelization strategy

- The **bunch is sliced** up by the master...
- ... and the slices are made to **circulate around the ring** for transverse tracking and e-cloud interactions

On the master process:

To be treated: 10 9 8 7 6 5 4

Already treated:

More details at: [https://github.com/PyCOMPLETE/PyPARIS/wiki](https://github.com/PyCOMPLETE/PyPARIS/wiki)
Parallelization strategy

- The **bunch is sliced** up by the master...
- ... and the slices are made to **circulate around the ring** for transverse tracking and e-cloud interactions

**On the master process:**

To be treated: 10 9 8 7 6 5

Already treated:

More details at: [https://github.com/PyCOMPLETE/PyPARIS/wiki](https://github.com/PyCOMPLETE/PyPARIS/wiki)
Parallelization strategy

- The bunch is sliced up by the master...
- ... and the slices are made to circulate around the ring for transverse tracking and e-cloud interactions

On the master process:

To be treated: 10 9 8 7 6

Already treated:

More details at: https://github.com/PyCOMPLETE/PyPARIS/wiki
Parallelization strategy

- The **bunch is sliced** up by the master...
- ... and the slices are made to **circulate around the ring** for transverse tracking and e-cloud interactions

**On the master process:**

To be treated: 10 9 8 7

Already treated:

More details at: [https://github.com/PyCOMPLETE/PyPARIS/wiki](https://github.com/PyCOMPLETE/PyPARIS/wiki)
Parallelization strategy

- The **bunch is sliced** up by the master...
- ... and the slices are made to **circulate around the ring** for transverse tracking and e-cloud interactions

**On the master process:**

To be treated: 10 9 8

Already treated:

More details at: [https://github.com/PyCOMPLETE/PyPARIS/wiki](https://github.com/PyCOMPLETE/PyPARIS/wiki)
Parallelization strategy

- The **bunch is sliced** up by the master...
- ... and the slices are made to **circulate around the ring** for transverse tracking and e-cloud interactions

**On the master process:**

To be treated: 10 9

Already treated:

More details at: [https://github.com/PyCOMPLETE/PyPARIS/wiki](https://github.com/PyCOMPLETE/PyPARIS/wiki)
Slices are **recollected by the master** at the end of the turn.

**On the master process:**

- To be treated: 10
- Already treated: 1

More details at: [https://github.com/PyCOMPLETE/PyPARIS/wiki](https://github.com/PyCOMPLETE/PyPARIS/wiki)
• Slices are **recollected by the master** at the end of the turn

**On the master process:**

To be treated: 2 1

Already treated: 2 1

More details at: [https://github.com/PyCOMPLETE/PyPARIS/wiki](https://github.com/PyCOMPLETE/PyPARIS/wiki)
Parallelization strategy

- Slices are **recollected by the master** at the end of the turn

**On the master process:**

To be treated:

Already treated:

More details at: [https://github.com/PyCOMPLETE/PyPARIS/wiki](https://github.com/PyCOMPLETE/PyPARIS/wiki)
Parallelization strategy

- Slices are **recollected by the master** at the end of the turn

**On the master process:**

- To be treated:
- Already treated:

More details at: [https://github.com/PyCOMPLETE/PyPARIS/wiki](https://github.com/PyCOMPLETE/PyPARIS/wiki)
Parallelization strategy

- Slices are **recollected by the master** at the end of the turn

**On the master process:**

To be treated:

Already treated:

More details at: [https://github.com/PyCOMPLETE/PyPARIS/wiki](https://github.com/PyCOMPLETE/PyPARIS/wiki)
• Slices are **recollected by the master** at the end of the turn

**On the master process:**

To be treated: 6 5 4 3 2 1

Already treated: 6 5 4 3 2 1

More details at: [https://github.com/PyCOMPLETE/PyPARIS/wiki](https://github.com/PyCOMPLETE/PyPARIS/wiki)
Parallelization strategy

- Slices are **recollected by the master** at the end of the turn

**On the master process:**

To be treated: 7 6 5 4 3 2 1

Already treated: 7 6 5 4 3 2 1

More details at: [https://github.com/PyCOMPLETE/PyPARIS/wiki](https://github.com/PyCOMPLETE/PyPARIS/wiki)
Parallelization strategy

- Slices are **recollected by the master** at the end of the turn

On the master process:

To be treated: 8 7 6 5 4 3 2 1

Already treated: 8 7 6 5 4 3 2 1

More details at: [https://github.com/PyCOMPLETE/PyPARIS/wiki](https://github.com/PyCOMPLETE/PyPARIS/wiki)
Parallelization strategy

- Slices are **recollected by the master** at the end of the turn

On the master process:

To be treated: 9 8 7 6 5 4 3 2 1

Already treated: 9 8 7 6 5 4 3 2 1

More details at: [https://github.com/PyCOMPLETE/PyPARIS/wiki](https://github.com/PyCOMPLETE/PyPARIS/wiki)
Parallelization strategy

• When all the slices are back at the master, the bunch is re-merged

On the master process:

To be treated:

Already treated: 10 9 8 7 6 5 4 3 2 1

More details at: https://github.com/PyCOMPLETE/PyPARIS/wiki
• When all the slices are back at the master, **the bunch is re-merged**

More details at: [https://github.com/PyCOMPLETE/PyPARIS/wiki](https://github.com/PyCOMPLETE/PyPARIS/wiki)
Parallelization strategy

- When all the slices are back at the master, the bunch is re-merged
- Then the longitudinal motion is computed

On the master process:

To be treated:

Already treated:

More details at: https://github.com/PyCOMPLETE/PyPARIS/wiki
• The bunch is **re-sliced for the following turn**

---

**On the master process:**

To be treated: 10 9 8 7 6 5 4 3 2 1

Already treated:

More details at: [https://github.com/PyCOMPLETE/PyPARIS/wiki](https://github.com/PyCOMPLETE/PyPARIS/wiki)
Parallelization strategy: expected speed-up

Since all slices need to be recollected at the master process, all processes will be waiting for some time at the beginning and at the end of each turn.

→ We can anticipate that the expected speedup will scale less than linearly with the number of processes.

The expected speed-up is given by:

\[
S(N_{\text{proc}}) = \frac{T_{\text{turn}}(1)}{T_{\text{turn}}(N_{\text{proc}})} = \frac{N_{\text{slices}}N_{\text{proc}}}{N_{\text{proc}} + N_{\text{slices}} - 1}
\]

which approaches the ideal speedup:

\[
S(N_{\text{proc}}) \approx N_{\text{proc}}
\]

when \( N_{\text{proc}} \ll N_{\text{slices}} \)

Where:

- \( N_{\text{slices}} \) is the number beam slices
- \( N_{\text{EC}} \) is the number of e-cloud interactions
- \( N_{\text{proc}} \) is the number of processes
- \( T_{\text{EC, Slice}} \) is the computing time for one slice at one e-cloud interaction

More details at: [https://github.com/PyCOMPLETE/PyPARIS/wiki](https://github.com/PyCOMPLETE/PyPARIS/wiki)
My wish-list when I started:

• 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 details at: https://github.com/PyCOMPLETE/PyPARIS/wiki
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

**Communicator** (MPI or MPI-like)

---

**Simulation object**
(defined 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
- PySUSSIX

- User defined Python (or Python-callable) code

More details at: [https://github.com/PyCOMPLETE/PyPARIS/wiki](https://github.com/PyCOMPLETE/PyPARIS/wiki)
Parallel computing implementation

• PyPARIS has been developed using mpi4py (python wrapper for MPI) to implement the parallelization:
  
  ➔ Drawback: you need MPI installation even if you want to run using two cores on your laptop

• On the other hand, the Python Standard Library provides a module called "multiprocessing" to handle several processes on the same machine
  
  ➔ Could I get an alternative for non-MPI users?
  ➔ Unfortunately the interface is completely different...

I did not want to develop another implementation of the Ring of CPU based on multiprocessing since duplication is a nightmare for maintenance

Solution:

• Exploit **Python’s duck typing**. Written a little module based on multiprocessing which mimics an MPI communicator from the PyPARIS prospective

• Just **switch between the two modules 😊**
You can **choose the parallelization engine** when starting your simulation.

PyPARIS **provides three different executables** to run simulations with the different parallelization modes:

- To run in **parallel with multiprocessing** (without MPI):
  
  ```
  PyPARIS/multiprocexec.py -n 3 sim_class=Simulation.Simulation
  ```

- To run in **parallel with MPI**:
  
  ```
  mpiexec -n 4 PyPARIS/withmpi.py sim_class=Simulation.Simulation
  ```

- To run **serial** (useful for debugging, profiling etc..):
  
  ```
  PyPARIS/serialexec.py sim_class=Simulation.Simulation
  ```

Here the Simulation module can be exactly the same in all three cases
Impact on performance

Computing time for 1024 turns
(exacted at CNAF)

- Single grid: 46 days
- Multigrid: 19 days
- PyPARIS: 3 days

Numerical settings:
- PIC resolution: 3 points per beam sigma over a region of 10 beam sigmas
- 700k MP per e-cloud (need to populate such a fine grid)
- Contrary to the serial case you cannot re-use the same e-cloud for many kicks
  ➔ We are physically allocating and simulating >20 millions MP!!!
• Introduction
• PyECL OUD simulations:
  • e-cloud buildup simulations
  • Simulations of e-cloud effects on beam dynamics (stability)
• Main development milestones
• Technical information
• The Particle-In-Cell engine: PyPIC
• Parallelization strategy for PyECL OUD-PyHEADTAIL simulations (PyPARIS)
• Directions for future development
• Status and needs for computing resources
Future development

**e-cloud buildup simulations:**
- Implementation ongoing for **high order multipoles** (to be validated)
- It would be very useful to introduce **other methods for Secondary Emission** (e.g. Furman-Pivi), possibly with input by surface experts
- Simulations with **super-imposed RF fields** (cavities)

**e-cloud instability simulations:**
- Significant **development done in 2016** → **need computing resources** to really profit of it
  - We **need to make experience** with this new simulation regime (10k turns) to really identify what is still missing (and in what priority)
  - Many effects are already implemented but never studied (interplay with impedance and beam-beam), non uniform optics (IRs, ATS)
- For the further future: full simulations of **coupled bunch instabilities** (lower priority...)

- **Further development for the ion cloud simulations?** Different species at the same time?
  - Depends on interest and resources...

**Resources for development and maintenance** shared with those for e-cloud studies remodulating according to needs → it worked quite well up to now...
• Introduction
• PyECLoUD simulations:
  • e-cloud buildup simulations
  • Simulations of e-cloud effects on beam dynamics (stability)
• Main development milestones
• Technical information
• The Particle-In-Cell engine: PyPIC
• Parallelization strategy for PyECLoUD-PyHEADTAIL simulations (PyPARIS)
• Directions for future development
• Status and needs for computing resources
Computing resources

Buildup simulations:
- Typically run on \textit{lxbatch} (single core), local machines when looking at details (e.g. e$^-$ trajectories)
- Run times ranging \textbf{from 1 h} (72b, dipole field) \textbf{to 2-3 days} (full LHC beam, quadrupole field)
- \textbf{Parametric scans} are essential for these studies (typically running up to \textbf{2000 jobs})
- Reasonably \textbf{happy} with lxbatch performance (on a good day...)

\textbf{PyECLOUD-PyHEADTAIL simulations (stability studies)}
- These simulations are \textbf{heavier} \rightarrow need to run parallel:
  - Typical setup \textbf{8 - 16 CPU cores per job} \rightarrow but still simulations can take \textbf{2-3 weeks} (10k turns)
  - Need \textbf{parametric scans} \rightarrow several simulations at the same time
- At the moment the only resource available is \textbf{INFN-CNAF cluster}:
  - Negotiated \textbf{168 cores} dedicated to CERN studies (always running since a few months)
  - \textbf{Only 21 simulations} at the same time
    \rightarrow progress with LHC studies is very slow, HL-LHC studies cannot really start...
- To effectively profit of the available manpower \textbf{we need significantly more computing resources} (ideally by a factor of 10)!
  \rightarrow We don’t strictly need infiniband, \textbf{groups of 8-16 cores} on the same node can do the job (can we negotiate this on lxbatch?)
Thanks for your attention!