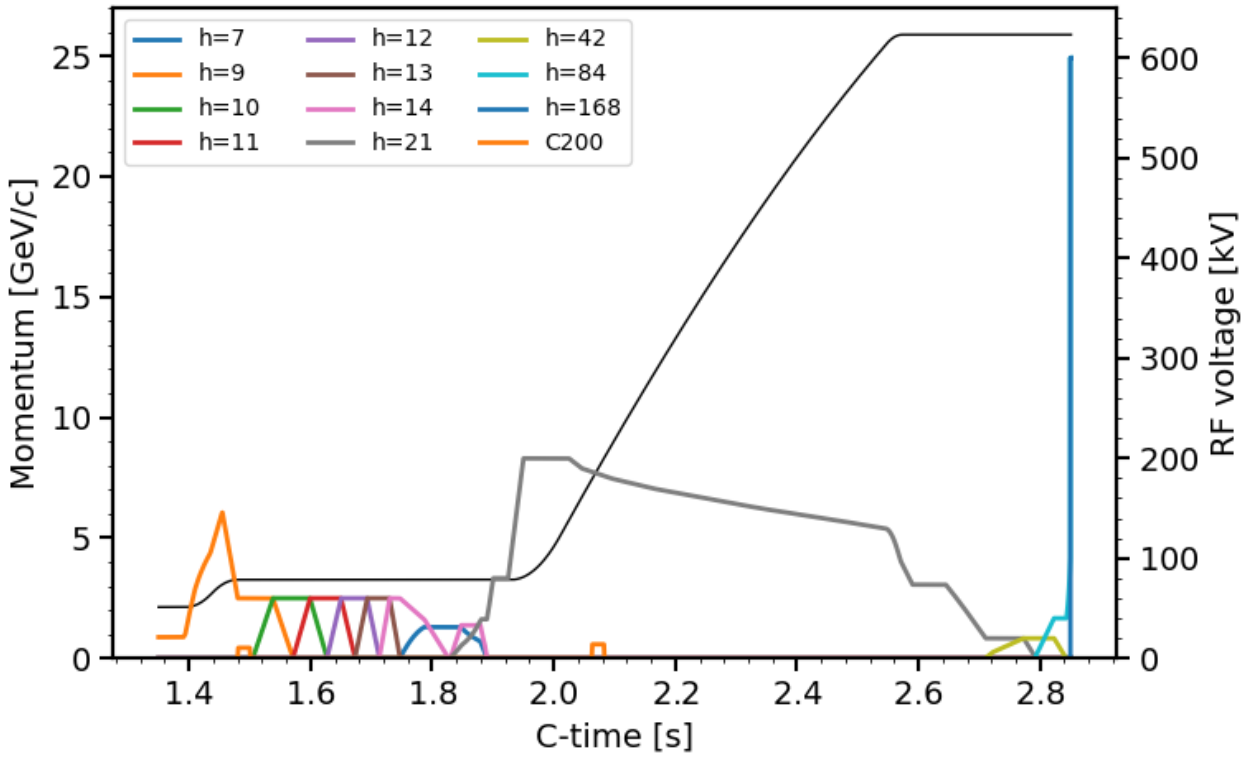
End-to-end simulations in the PS using MPI

BLonD developers meeting 04/12/2020

A. Lasheen, K. Iliakis

PS end-to-end program

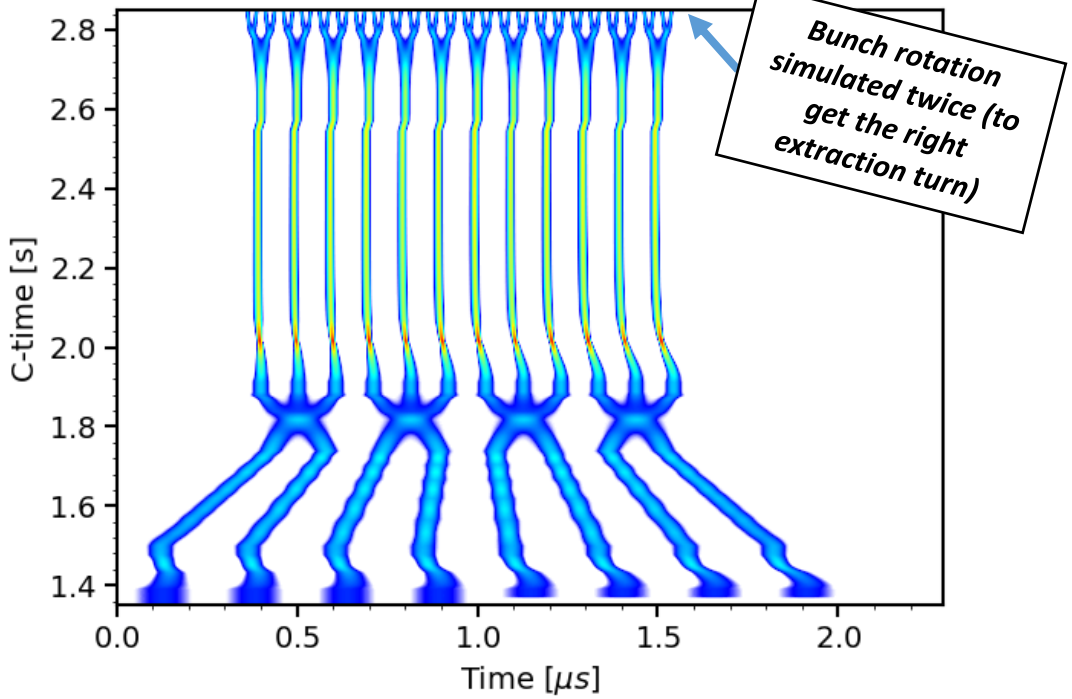
Momentum and RF programs for the BCMS cycle



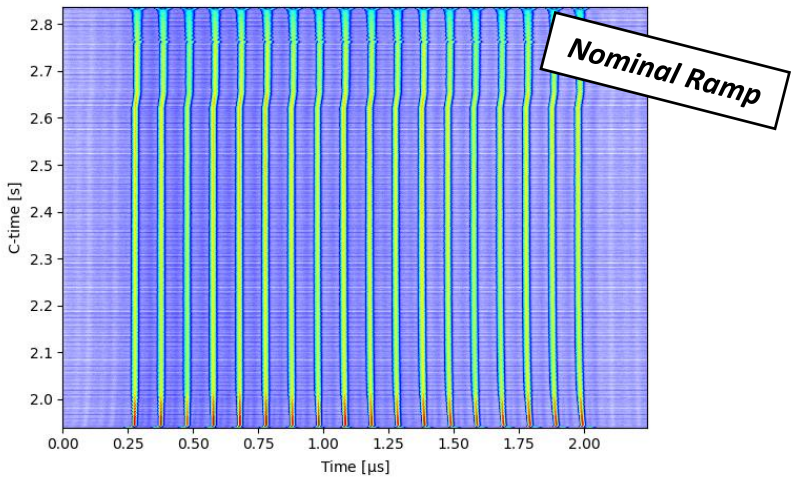
- All processes in the PS were simulated individually with Impedance/LLRF
- Some open questions, do we have long range effects like:
  - Influence of transition crossing to seed coupled bunch instabilities during the ramp?
  - Controlled emittance blow-up contributing to tails and losses at PS-SPS transfer?
- Can we simulate the whole PS cycle in general using the SLURM cluster?

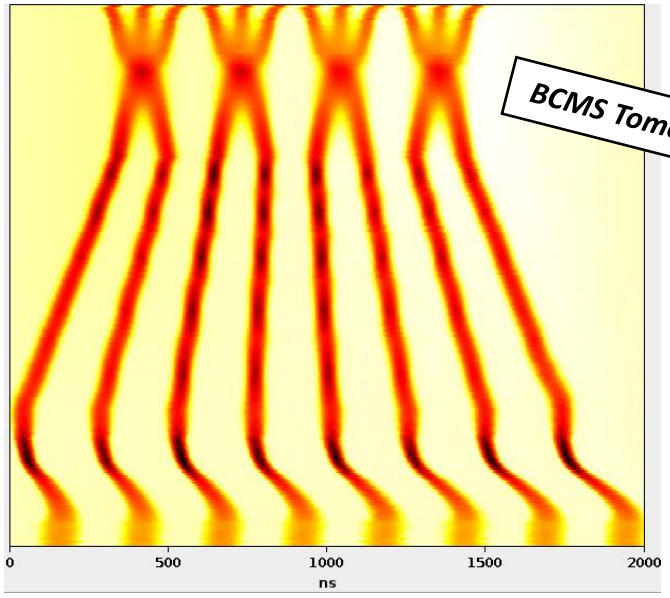
Simulation without impedance/llrf

Simulated profiles



Measured profiles

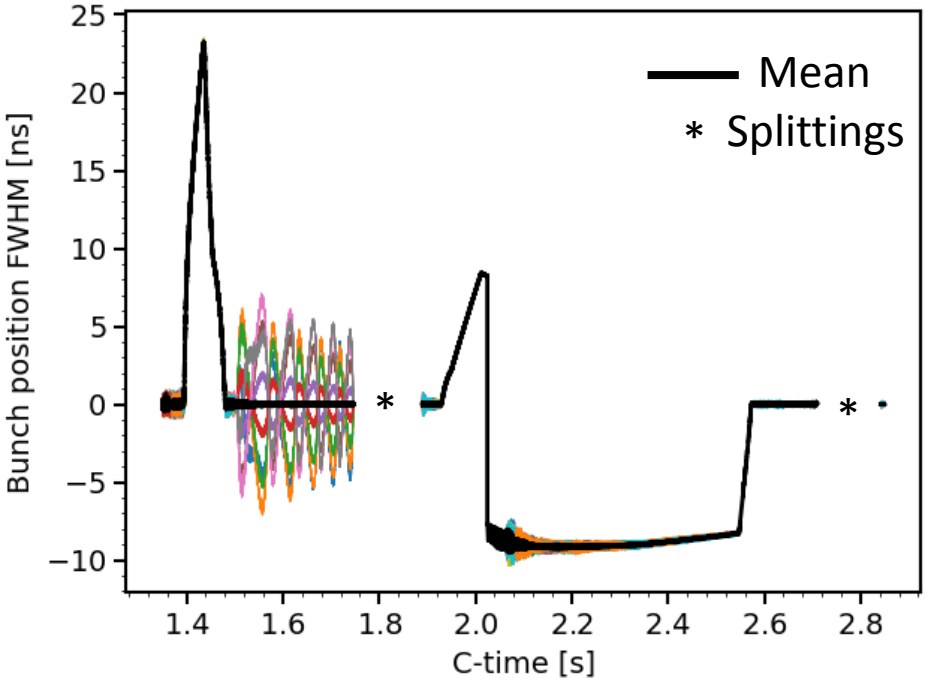




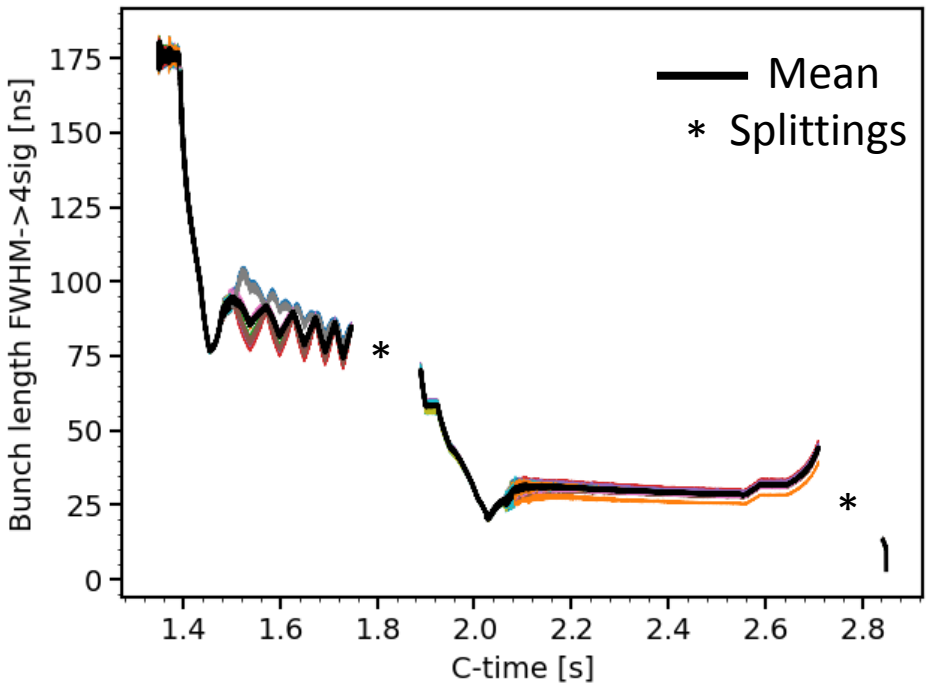
- Simulation without impedance/llrf ran successfully and fast on the SLURM cluster.
- Runtime ~3.4 hours with 48M macroparticles with 4 nodes (~800k sim. turns).
- Minor adaptations (e.g. squeezed long flat bottom, no synchro with SPS…)

Bunch position and bunch length

Bunch-by-bunch position using FWHM



Bunch-by-bunch length using FWHM

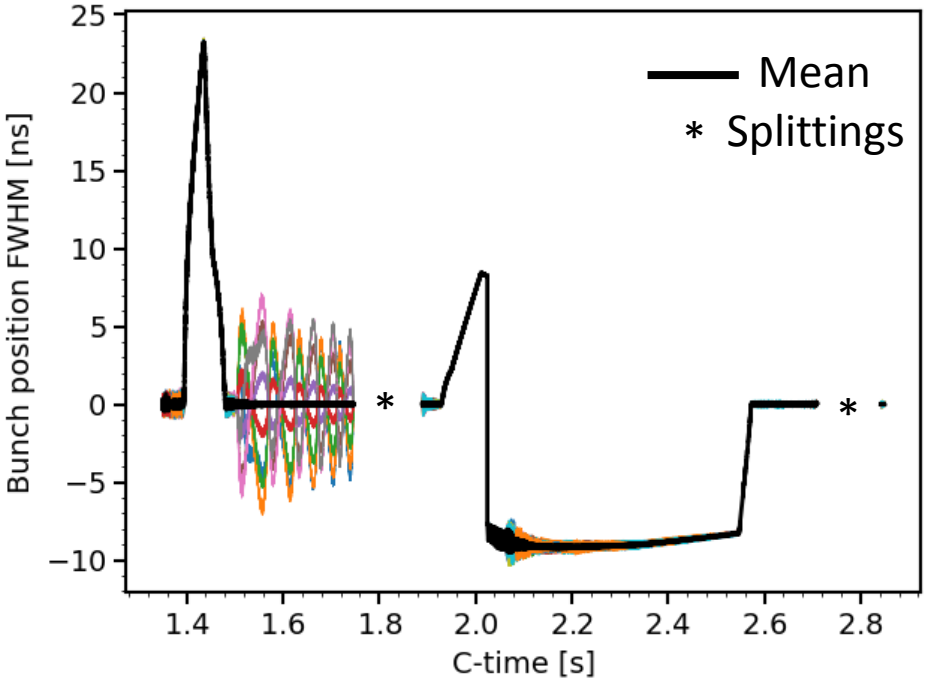


- Bunch position vs. center of bucket, effectively the bunch/beam phase. Bucket/phase definition at bit more delicate during batch rf manipulations…

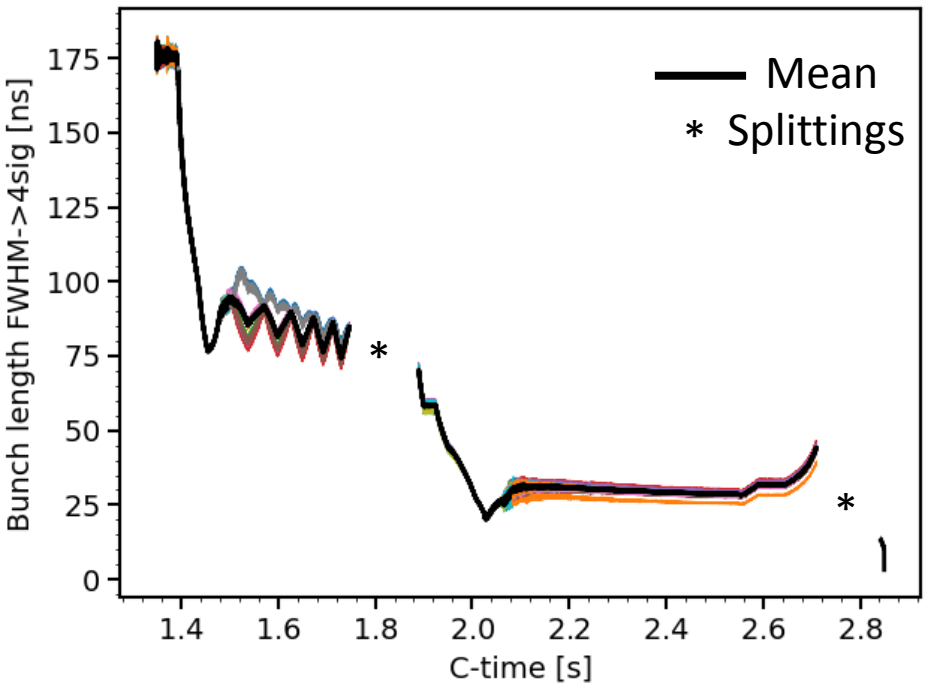
- Bunch length evolution follows qualitatively the expectation, only the controlled emittance blow-up needs to be checked (need to increase the voltage to get a good qualitative behavior)

Bunch position and bunch length

Bunch-by-bunch position using FWHM

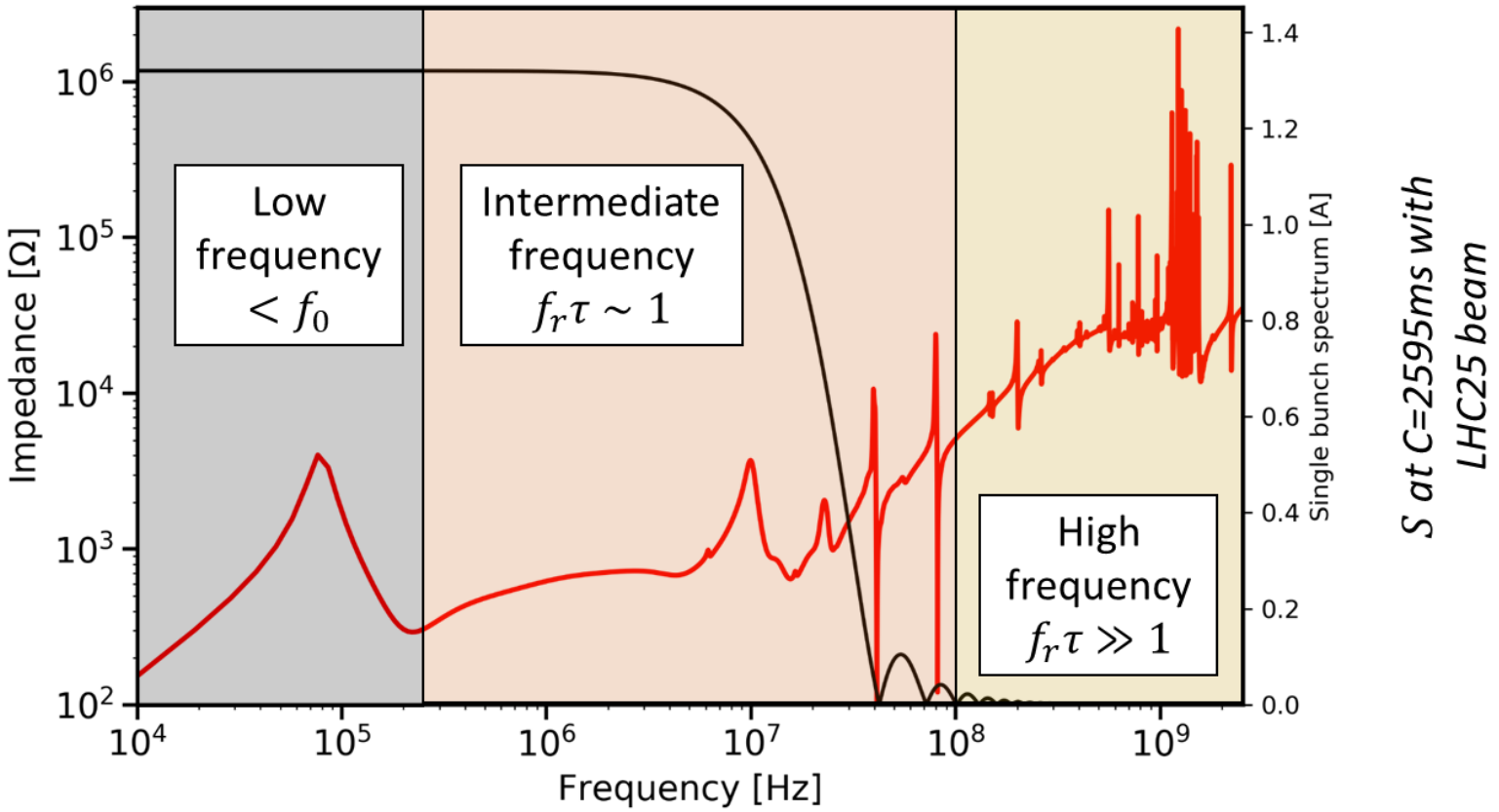


Bunch-by-bunch length using FWHM



- Need to compare with a consistent set of measured data:
  - ➢ Tomoscope spans the whole cycle, but not all locations are measured.
  - ➢ Other measurements along the ramp acquired in 2018 but no measurements before and on the plateau and after last splittings.

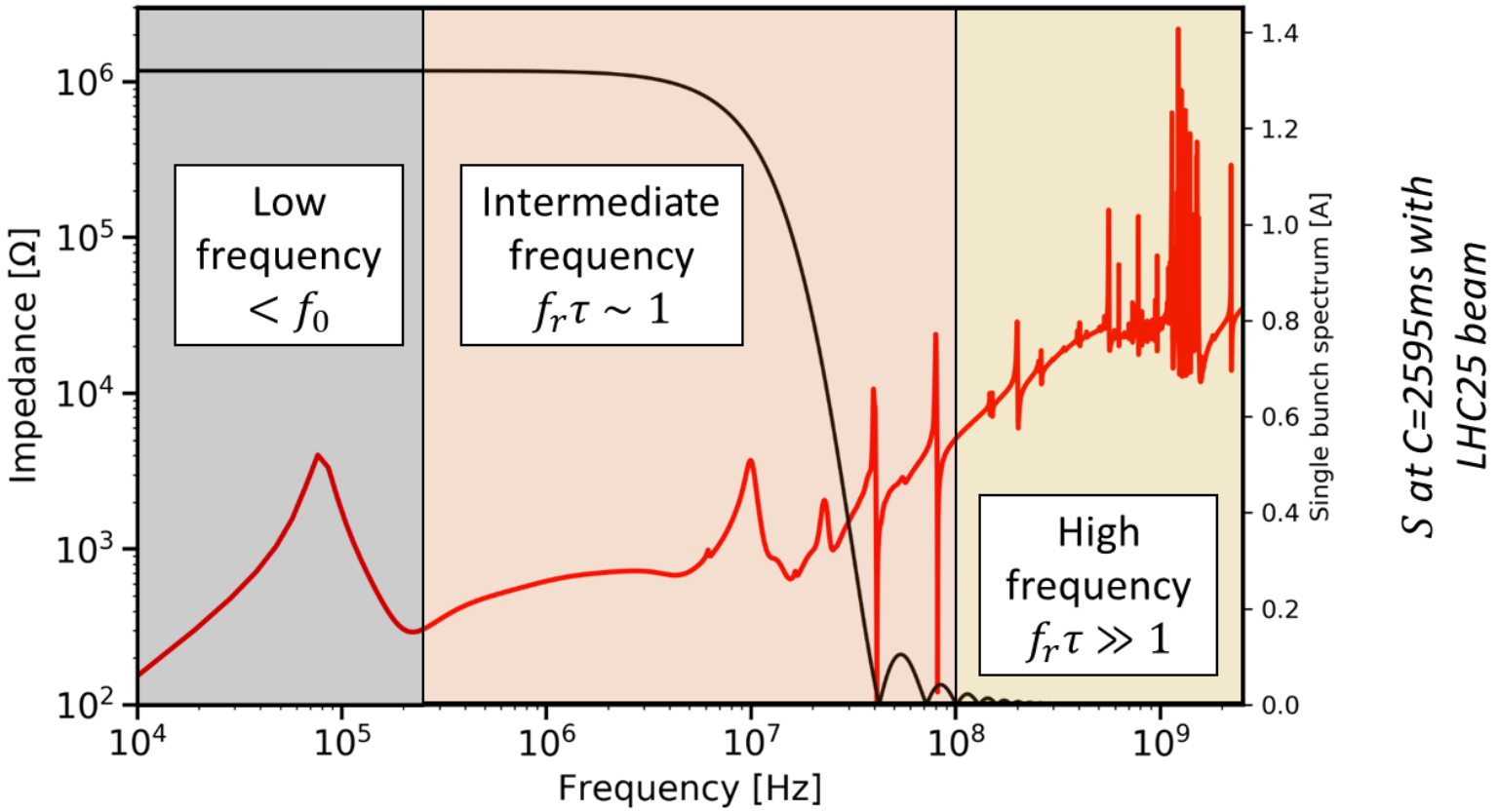
PS impedance model



- Impedance model in PS covering large range in frequency, and bunch length is varying a lot during the ramp

- Impedance sources are changing over time, mostly the cavities (tuning, impedance reduction with closed gap relays, feedbacks, damping…). The impedance needs to be updated regularly.

PS impedance model



- Various approaches are possible:
  - Having several profiles with different bin size (several FFTs and tracking operation)
  - Adapting the bin size to the bunch length (one FFT with varying length, not ideal for multi-turn wake calculation)
  - Keeping fixed bin size (one very large FFT, large number of points to recompute the impedance and large number of particles needed)

Impedance update during tracking

```
ind_voltage_freq = InducedVoltageFreq(...)

total_ind = TotalInducedVoltage(..., [ind_voltage_object])

# Tracking loop
for turn in range(n_turns):

    my_new_real_Z, my_new_imag_Z = impedance_calc(frequency_array)

    ind_voltage_freq.impedance_source_list[...].Re_Z_array_loaded = my_new_real_Z
    ind_voltage_freq.impedance_source_list[...].Im_Z_array_loaded = my_new_imag_Z

    ind_voltage_freq.sum_impedances(frequency_array)

    total_ind.induced_voltage_sum()

```

- The sum impedances recompute the impedance if a Resonator object is passed, reinterpolate if InputTable

- Can be slow, an option to avoid reinterpolating could be foreseen…

Impedance update during tracking

```
1  ind_voltage_object = InducedVoltageFreq(...)
2
3  total_ind = TotalInducedVoltage(..., [ind_voltage_object])
4
5  # Tracking loop
6  for turn in range(n_turns):
7
8      my_new_impedance = impedance_calc(frequency_array)
9
10     ind_voltage_object.total_impedance[:] = \
11         (my_new_impedance +...) / profile.bin_size
12
13     total_ind.induced_voltage_sum()
14
```

- Bypassing the sum_impedances() already speed things up.

- Warning: the loaded impedance is not updated and is not consistent with the total_impedance anymore, and the new impedance should always be calculated on the same frequency array.

Partial impedance update

```
ind_voltage_object = InducedVoltageFreq(...)

total_ind = TotalInducedVoltage(..., [ind_voltage_object])

n_points_update = np.where(frequency_array < my_freq_limit)[0][-1]

# Tracking loop
for turn in range(n_turns):

    my_new_impedance = impedance_calc(frequency_array[:n_points_update])

    ind_voltage_object.total_impedance[:n_points_update] = \
        (my_new_impedance +...) / profile.bin_size

    total_ind.induced_voltage_sum()

```

- The PS impedance is only changing at relatively low frequencies, the impedance can be partially update by targeting a certain frequency band

- Warning: need to keep margin at high frequency to have proper decay of the impedance

Using MPI

```
1  ind_voltage_object = InducedVoltageFreq(...)
2
3  total_ind = TotalInducedVoltage(..., [ind_voltage_object])
4
5  n_points_update = np.where(frequency_array < my_freq_limit)[0][-1]
6  frequency_array = frequency_array[:n_points_update]
7
8  frequency_array = worker.scatter(frequency_array)
9
10 # Tracking loop
11 for turn in range(n_turns):
12
13     my_new_impedance = impedance_calc(frequency_array) / profile.bin_size
14
15     my_new_impedance = worker.allgather(my_new_impedance)
16
17     ind_voltage_object.total_impedance[:n_points_update] = \
18         (my_new_impedance +...)
19
20     total_ind.induced_voltage_sum()
21
```

Each node works on one frequency range

All nodes get back a copy of the full impedance

- The MPI worker as designed in the BLonD utils can be used to distribute the impedance calculation and update across the nodes in a very easy and elegant way.

- Warning: our functions to compute impedance like Resonators.imped_calc() in C++ assume that the first point correspond to f=0, this function may need some adaptation so that we can use with MPI together with openMP.

Using multiprocessing (1)

```
10  my_new_impedance_real_mp = mp.Array(ctypes.c_double,[0]*n_points_update)
11  my_new_impedance_real = np.frombuffer(my_new_impedance_real_mp.get_obj())
12
13  my_new_impedance_imag_mp = mp.Array(ctypes.c_double,[0]*n_points_update)
14  my_new_impedance_imag = np.frombuffer(my_new_impedance_imag_mp.get_obj())
15
16  shared_globals = {
17      'frequency_array':frequency_array,
18      'whatever_parameter_you_need':whatever_parameter_you_need}
19  shared_mp_arrays = {
20      'my_new_impedance_real_mp':my_new_impedance_real_mp,
21      'my_new_impedance_imag_mp':my_new_impedance_imag_mp}
22
23  pool = mp.Pool(processes=n_cores,
24            initializer=mp_init_func,
25            initargs=(shared_globals,
26                      shared_mp_arrays))
27
28  list_indexes_mp = split_indexes_mp(n_points_update, n_cores)
```

- For computation on local PC, or to profit from the several cores/nodes, functions can be written in C/C++ to profit from openMP.

- Some functions are too tedious to write in C/C++, multiprocessing can be used for these cases.

Using multiprocessing (2)

```
10  my_new_impedance_real_mp = mp.Array(ctypes.c_double,[0]*n_points_update)
11  my_new_impedance_real = np.frombuffer(my_new_impedance_real_mp.get_obj())
12
13  my_new_impedance_imag_mp = mp.Array(ctypes.c_double,[0]*n_points_update)
14  my_new_impedance_imag = np.frombuffer(my_new_impedance_imag_mp.get_obj())
15
16  shared_globals = {
17      'frequency_array':frequency_array,
18      'whatever_parameter_you_need':whatever_parameter_you_need}
19  shared_mp_arrays = {
20      'my_new_impedance_real_mp':my_new_impedance_real_mp,
21      'my_new_impedance_imag_mp':my_new_impedance_imag_mp}
22
23  pool = mp.Pool(processes=n_cores,
24            initializer=mp_init_func,
25            initargs=(shared_globals,
26                      shared_mp_arrays))
27
28  list_indexes_mp = split_indexes_mp(n_points_update, n_cores)
```

Specific arrays/variable to share and update

Generic functions

- A pool of processes should be open before the tracking loop.
- Arrays and other variables are shared across the different processes (independent python interpreters).
- Generic wrapper functions could be done on the model of the MPI worker to simplify the process.

Using multiprocessing (3)

```
31  # Tracking loop
32  for turn in range(n_turns):
33
34      pool.starmap(impedance_calc_mp,
35                   zip(list_indexes_mp,
36                       repeat(turn),
37                       ...),  #
38                   chunksize=1)
39
40      my_new_impedance = (my_new_impedance_real +
41                          1j * my_new_impedance_imag) / profile.bin_size
42
43      my_new_impedance = worker.allgather(my_new_impedance)
44
45      ind_voltage_object.total_impedance[:n_points_update] = \
46          (my_new_impedance +...)
47
48      total_ind.induced_voltage_sum()
```

Function and arguments

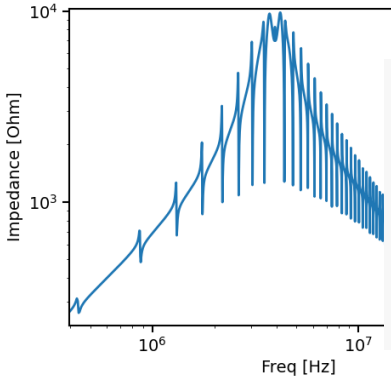
Other static variables can be accessed via the shared dictionaries and the impedance should be updated via the mp.Array

These numpy arrays are automatically updated by reference

- pool.starmap is used to call the impedance_calc function which was adapted (need to get the need variables from the shared dictionaries)

- If the frequency_array was split with MPI before the multiprocessing code of previous slide, then MPI can be combined with multiprocessing and the result is gathered after the pool.starmap operation.

Timings for C10 update




```
All_C10_1TFB_Parametric_model.load_model(
    main_harmonic,
    revolution_frequency,
    impedance_reduction_target,
    main_harmonic_FB=False,
    manual_params=None,
    ZFactor=1.):
```

NB: not systematic study, the values are obtained with time.perf_counter() and the most representative timing is picked by hand…

Local PC (4 cores, 8 hyperthreaded)

Freq. range	noOpti	pyMP	Speedup
46k (200 MHz)	34 ms	30 ms	x1.1
1M (4.7 GHz)	510 ms	230 ms	x2.2

SLURM (4 nodes, 80 cores)

Freq. range	noOpti	pyMP	Speedup	MPI	Speedup	BothOpti	Speedup
46k (200 MHz)	28 ms	28 ms	x1.0	13 ms	x2.2	36 ms	x0.8
1M (4.7 GHz)	510 ms	205 ms	x2.0	90 ms	x5.6	137 ms	x3.7

- MPI gives very good results regarding how easily the implementation can be used! (NB: MPI timings were extremely stable)
- Multiprocessing can be a good help for local computation, thoughts for generic implementation in BLonD in utils?
- Both MPI/multiprocessing don’t seem to go too well, especially for small arrays…

Overall speedup

- Massive speedup from the original tracking code with known routines to the optimized impedance update + usage of MPI

Complete tracking turn on local PC, before optimization	**Complete tracking turn on SLURM, after optimization**
4.8 M macro ~900 ms/turn	
48 M macro ~1080 ms/turn	*48 M macro ~127 ms/turn*
480 M macro ~KABOOM ms/turn	***480 M macro ~280 ms/turn***

- A full simulation is ~800k turns, further work to be completed, but an end-to-end simulation with all effects is expected to last ~2.5 days and seem within reach.
- First guess is that the massive FFT with multi turn wake is the present bottleneck.
- First simulations done with 50 effective turns of memory (100 turns needed for the FFT in total). To be checked if it can be reduced, considering the long response time of feedbacks.