Parallel Reader for Large Data files

In [ ]:

In [1]:
```
#General libraries (not always need all)
%lsmagic
import time
import os
import numpy as np
import h5py
import matplotlib
import matplotlib.pyplot as plt
from matplotlib.colors import LogNorm
import seaborn as sns
from scipy import constants
import pandas.util.testing as tm
import pandas as pd

#Analysis Libraries
#Best not to import all of scipy when you have imported numpy. They have
from scipy.optimize import curve_fit
from statsmodels.stats.weightstats import DescrStatsW

#Python and DASK libraries used for parallel operation and handeling larg
import multiprocessing as mp
from concurrent.futures import ThreadPoolExecutor
from dask.distributed import Client

#Constants typically needed
pi=constants.pi
electron_mass=constants.electron_mass
c=constants.speed_of_light
eV=constants.electron_volt
electron_mass_eV=electron_mass*c**2/eV
electron_charge=constants.elementary_charge
```

```
/tmp/ipykernel_17882/2061179318.py:12: FutureWarning: pandas.util.testing
is deprecated. Use the functions in the public API at pandas.testing inst
ead.
  import pandas.util.testing as tm
```

Note: You can see/modify the information about the DASK Scheduler using client but only if you are on batch, devel or mem192 nodes not login nodes).
> client=Client()
> client

In [2]:
```
#client=Client()
#client
```

In [3]:
```
#The big bad wolf data at the end of ImpactZ (larger than memeory)
full_beam_file ='/p/scratch/xseed2024/niknejadi/FLASH2020+/S2E/FLASH2020P
#The Data with only one in every 100th particle saved
reduced_file ='/p/scratch/xseed2024/niknejadi/FLASH2020+/S2E/FLASH2020P/L
```

Info for pooling, sharing memory, ...

In [4]:
```
cpu_count = mp.cpu_count()
file_size = os.path.getsize(full_beam_file)
chunk_size = file_size // cpu_count
#cpu_count, chunk_size, file_size
```

Some examples with Python's ThreadPoolExecuter and ProcessPoolExecutor

Here we use the reduced file due to memory constraint

In [5]:
```
%%time
from concurrent.futures import ThreadPoolExecutor
tpe=ThreadPoolExecutor(cpu_count)

def read_impact_beam(file, index):
    with open(file) as f:
        temp=f.read().split()[index::9]
        return list(map(float, temp))

index=[0,1,2,3,4,5,6,7,8]
futures = []

for x in index:
    future=tpe.submit(read_impact_beam,reduced_file,x)
    futures.append(future)

results=[future.result()for future in futures]
```

```
CPU times: user 1min, sys: 25.8 s, total: 1min 26s
Wall time: 1min 22s
```

In [6]:
```
%%time
from concurrent.futures import ProcessPoolExecutor
ppe=ProcessPoolExecutor(cpu_count)

futures = []

for x in index:
    future=ppe.submit(read_impact_beam,reduced_file,x)
    futures.append(future)

results=[future.result()for future in futures]
```

```
CPU times: user 2.94 s, sys: 2.92 s, total: 5.86 s
Wall time: 15.2 s
```

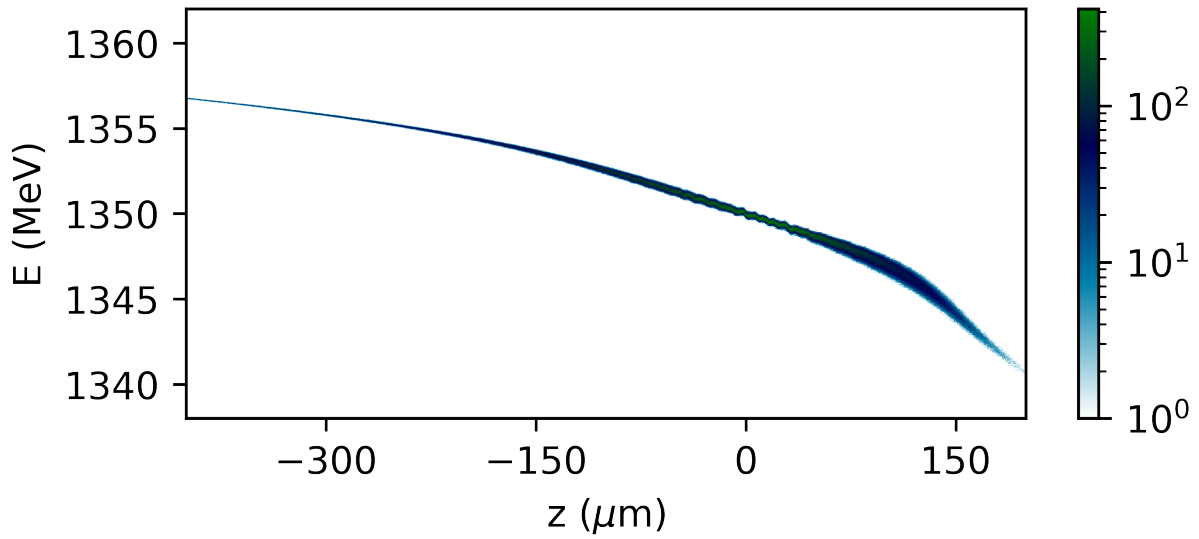
We need the referance file info (for converting to SU)

In [7]:
```
th=np.array(results[4])
delE=np.array(results[5])
mc=np.array(results[7])
pid=np.array(results[8])
```

In [8]:
```
fileloc ='/p/scratch/xseed2024/niknejadi/FLASH2020+/S2E/FLASH2020P/LINAC/
ref_particle_file=fileloc+'fort.18'
ref_KE_MeV=np.loadtxt(ref_particle_file)[:, 3]
final_ref_KE_MeV=ref_KE_MeV[-1]
E=-delE*electron_mass_eV+(final_ref_KE_MeV*1e6)
ScalingFrequesncy=1.3000000000000000e+09
z=-th/(2*pi)*c/ScalingFrequesncy
#E=delE+final_ref_KE_MeV
print("Total number of slices in Mod1 slice length: ",round(np.max(z)-np.
print("Weight of the macro particles is", np.mean(mc)/electron_charge)
```

```
Total number of slices in Mod1 slice length:  10257
Weight of the macro particles is 3.9945658076549098
```

In [9]:
```
%%time
%matplotlib inline
fig,bx = plt.subplots(1,1, figsize=(5,2),dpi=600)
h=bx.hist2d(-(np.mean(z)-z)*1e6, E*1e-6,bins=6000,cmap='ocean_r',norm=Log
bx.set_ylabel(r'E (MeV)')
bx.set_xlabel(r'z ($\mu$m)')
#bx.set_ylim(-12,12)
bx.set_ylim(1338,1362)
bx.xaxis.set_major_locator(plt.MaxNLocator(5))
bx.set_xlim(-400,200)
fig.colorbar(h[3])
plt.show()
```

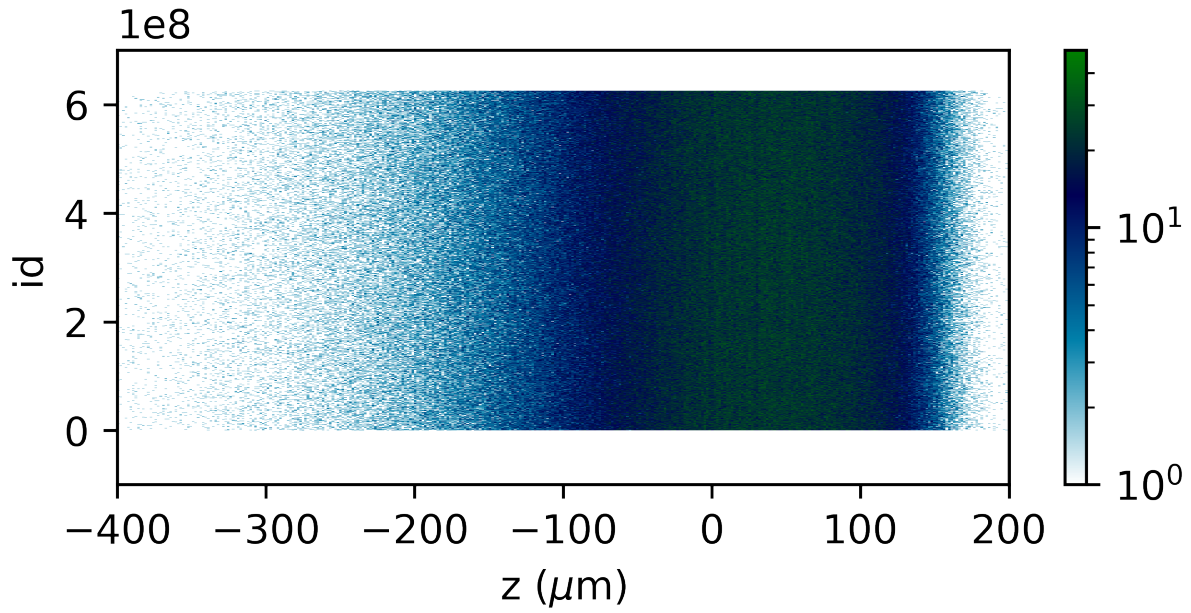



```
CPU times: user 15.6 s, sys: 1.38 s, total: 17 s
Wall time: 17.2 s
```

In [10]:
```
%%time
fig,bx = plt.subplots(1,1, figsize=(5,2),dpi=600)
h=bx.hist2d(-(np.mean(z)-z)*1e6, pid,bins=2000,cmap='ocean_r',norm=LogNor
bx.set_ylabel(r'id')
bx.set_xlabel(r'z ($\mu$m)')
bx.set_ylim(-0.1e9, .7e9)
bx.set_xlim(-400,200)
fig.colorbar(h[3])
```

```
CPU times: user 1.43 s, sys: 107 ms, total: 1.54 s
Wall time: 1.54 s
```

Out[10]:
```
<matplotlib.colorbar.Colorbar at 0x153e3702e940>
```



Reading and Processing the Full Beam File

Tip: ImpactZ output files have no header, sometimes they are comma sperated sometimes space sperated. For what each info on content of the files refer to the manual.

Reading with Pandas

In [11]:
```
%%time
a=pd.read_csv(full_beam_file, sep='\s+', header=None, dtype=np.float64).v
# Then you can read each coulumn as x=a[:,0]
```

```
CPU times: user 16min 47s, sys: 1min 15s, total: 18min 2s
Wall time: 18min 6s
```

In [12]:
```
%%time
#To make things nicer and life easier we create a panda dataframe
df = pd.DataFrame(a, columns=['x', 'px', 'y', 'py', 'theta', 'E - E0', 'c
```

```
CPU times: user 305 μs, sys: 30 μs, total: 335 μs
Wall time: 341 μs
```

Note 1: Without Pandas, we will be able to do most everything, it will be less efficient.
>\# Printing type, shape, size, and type of elements
>print("Array is of type: ", type(a))
>print("No. of dimensions: ", a.ndim)
>print("Shape of array: ", a.shape)
>print("Size of array: ", a.size)
>print("Array stores elements of type: ", a.dtype)
Note 2: for better memory handling, float parameters in pandas might be rounded and deviate slightly from actual parameter but the difference is in 1e-55 range

In [13]:
```
# Printing type, shape, size, and type of elements
print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 625000000 entries, 0 to 624999999
Data columns (total 9 columns):
 #   Column               Dtype  
---  ------               -----  
 0   x                    float64
 1   px                   float64
 2   y                    float64
 3   py                   float64
 4   theta                float64
 5   E - E0               float64
 6   charge over mass     float64
 7   macroparticle size   float64
 8   particle id          float64
dtypes: float64(9)
memory usage: 41.9 GB
None
```

In [14]:
```
pd.set_option("display.max.columns", None)
df.head()
```

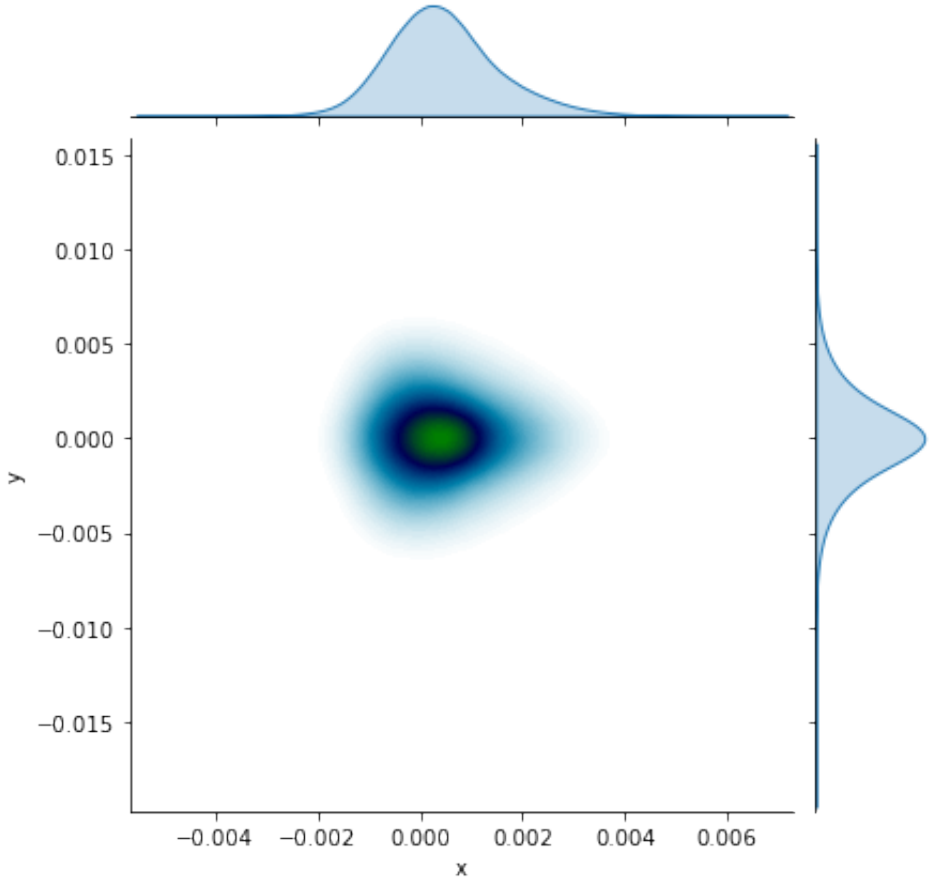
Out[14]:

	x	px	y	py	theta	E - E0	charge over mass	macropar
0	-0.000164	0.033211	-0.019200	0.145186	-0.004524	13.122991	0.000002	6.400000
1	-0.000634	-0.022001	-0.016979	0.125634	-0.003101	6.475946	0.000002	6.400000
2	-0.000558	-0.006560	-0.016907	0.115405	-0.003433	7.904633	0.000002	6.400000
3	-0.001002	-0.037772	-0.016882	0.112978	-0.002512	5.115332	0.000002	6.400000
4	-0.000244	-0.017483	-0.016858	0.102722	0.000057	-0.910063	0.000002	6.400000

In [15]:
```
%%time
#we can still reduce the particle files for ploting
rdf = df.iloc[::100]
#df_chart = sns.lineplot(x="theta", y="E - E0", data=rdf).set_title('ener
#plt.show()
```

```
CPU times: user 72 µs, sys: 7 µs, total: 79 µs
Wall time: 84.9 µs
```

In [16]:
```
%%time
df_xy_profile = sns.jointplot(x='x', y='y', data=rdf, kind='kde',cmap='oc
plt.show()
```




```
CPU times: user 1h 6min 47s, sys: 13.3 s, total: 1h 7min
Wall time: 1h 7min 1s
```

In [17]:
```
now = time.localtime() # get struct_time
time_string = time.strftime("%Y-%m-%d %H:%M:%S", now)
print('Conversion time: ',time_string)
```

```
Conversion time:  2022-11-23 04:51:07
```

Scale the Data

In [22]:
```
%%time
rf_frequesncy=1.3000000000000000e+09
rf_omega=2*pi*rf_frequesncy
#x and y are normalized to c over omega
df['x'] = df['x'].apply(lambda element: element*c/rf_omega)
df['y'] = df['y'].apply(lambda element: element*c/rf_omega)
```

```
CPU times: user 5min 12s, sys: 1min 16s, total: 6min 28s
Wall time: 6min 29s
```

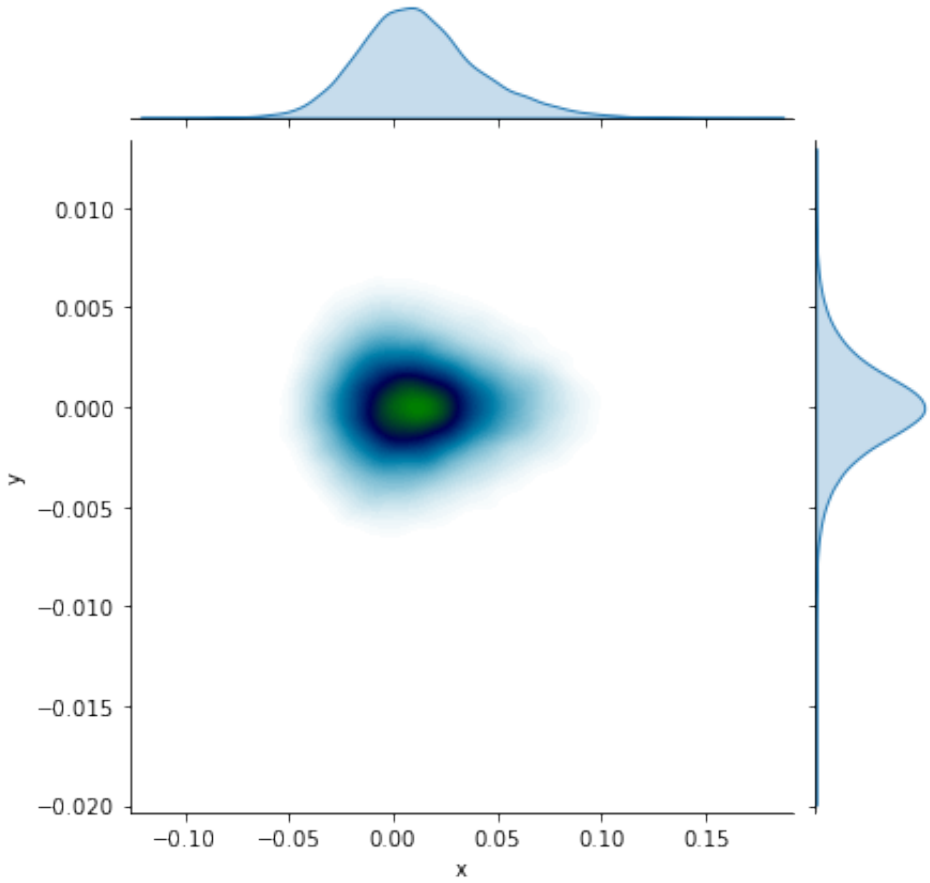
In [23]:
```
df.head()
```

Out[23]:

	x	px	y	py	theta	E - E0	charge over mass	macropar
0	-0.004468	0.033211	-0.019200	0.145186	-0.004524	13.122991	0.000002	6.400000
1	-0.017286	-0.022001	-0.016979	0.125634	-0.003101	6.475946	0.000002	6.400000
2	-0.015193	-0.006560	-0.016907	0.115405	-0.003433	7.904633	0.000002	6.400000
3	-0.027297	-0.037772	-0.016882	0.112978	-0.002512	5.115332	0.000002	6.400000
4	-0.006642	-0.017483	-0.016858	0.102722	0.000057	-0.910063	0.000002	6.400000

In [24]:
```
rdf = df.iloc[::10000]
```

In [26]:
```
%%time
df_xy_profile = sns.jointplot(x='x', y='y', data=rdf, kind='kde',cmap='oc
plt.show()
```




```
CPU times: user 54.8 s, sys: 198 ms, total: 55 s
Wall time: 51.6 s
```

In [ ]:
```
```