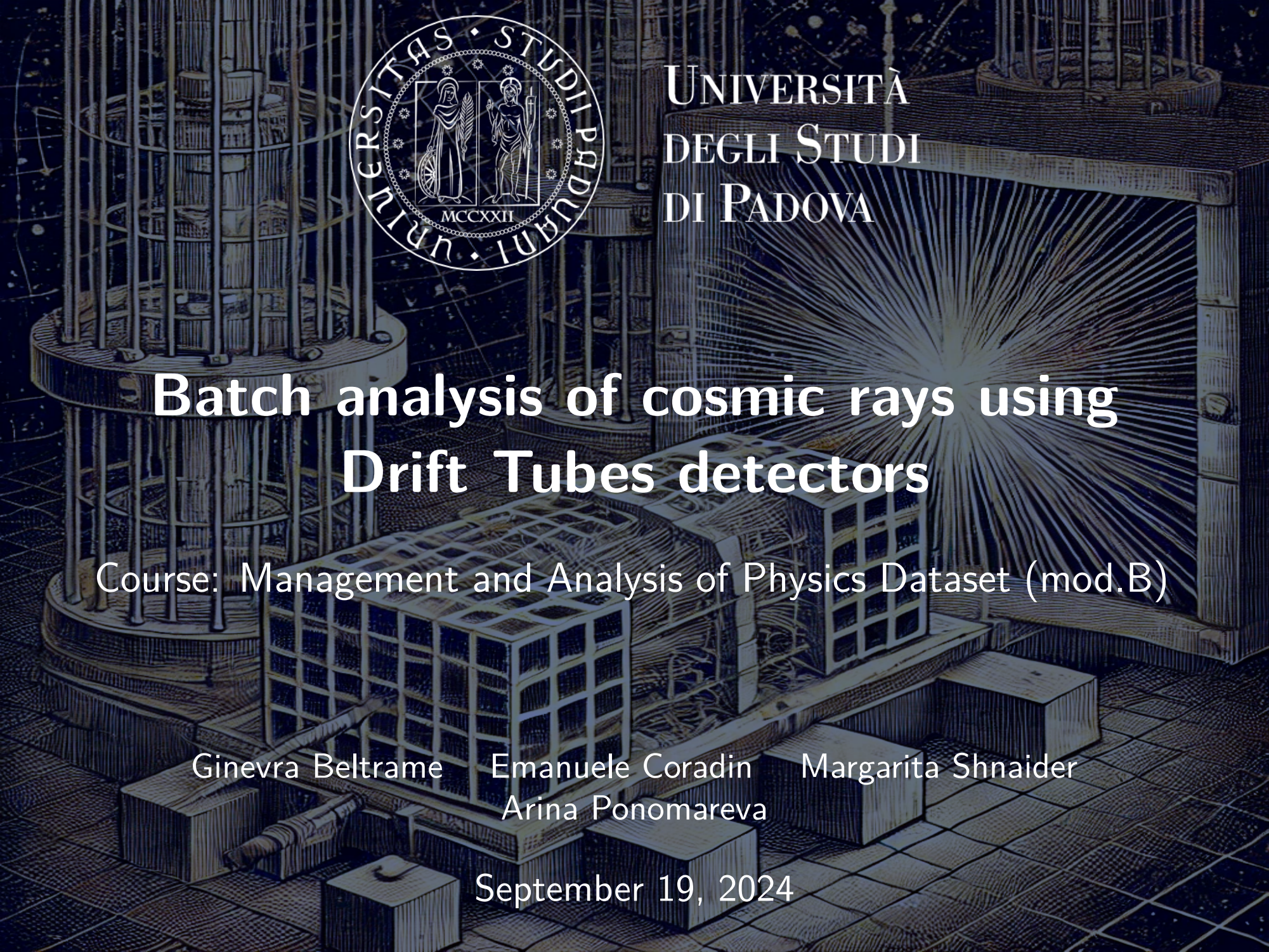


Università degli Studi di Padova

Batch analysis of cosmic rays using Drift Tubes detectors

Course: Management and Analysis of Physics Dataset (mod.B)

Ginevra Beltrame Emanuele Coradin Margarita Shnaider
Arina Ponomareva

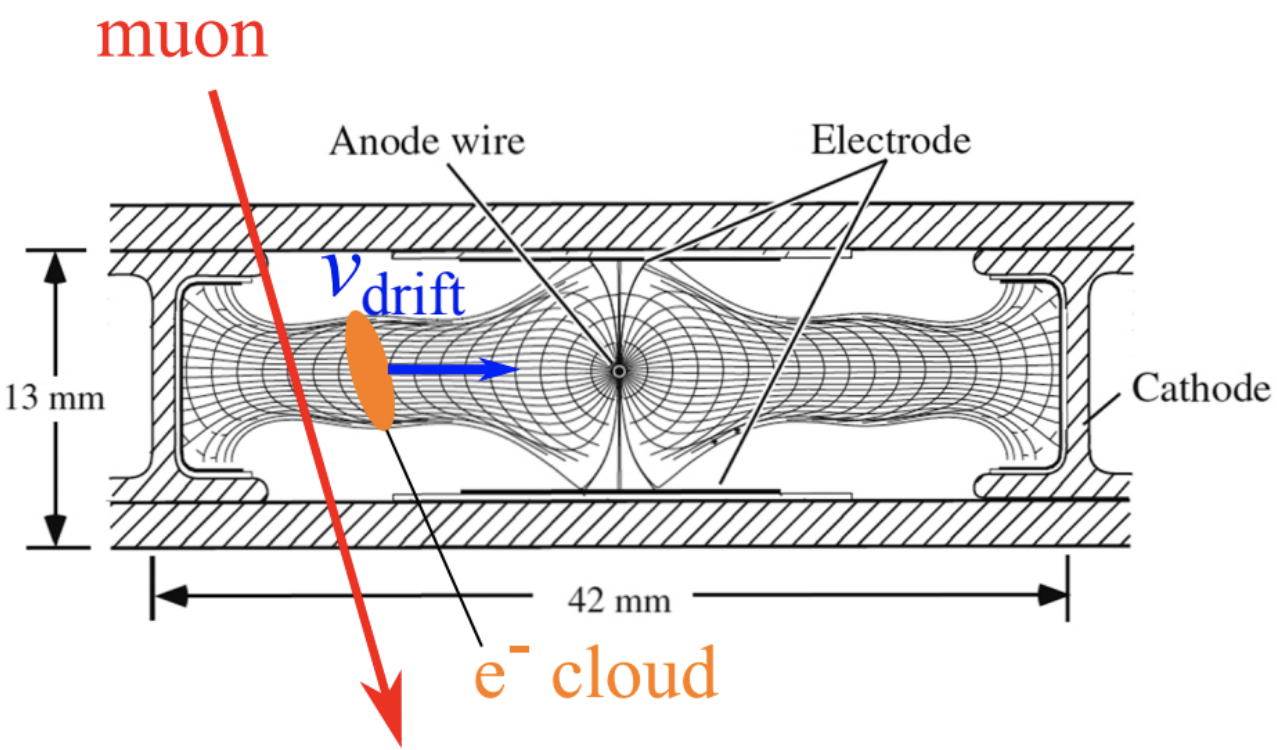
September 19, 2024

Table of Contents

Table of Contents

Introduction

The goal of this project is to analyze real data collected in a particle physics detector to reconstruct the trajectory of muons from cosmic rays.



Cosmic Ray Detection with Drift Tubes

- **Muon detection** via drift tubes installed in Legnaro INFN Laboratories, near Padova.
- Charged particles ionize the gas inside the detector, creating electron-ion pairs.

Cosmic Ray Detection with Drift Tubes

- **Muon detection** via drift tubes installed in Legnaro INFN Laboratories, near Padova.
- Charged particles ionize the gas inside the detector, creating electron-ion pairs.
- Electric field guides electrons towards the anode wire at **constant drift velocity**:

$$v_{\text{drift}} = 53.8\,\mu m/ns$$

- Signals are amplified and digitized by the DAQ system to identify ionization events, called **hits**.

Detection and Hits

- Each hit corresponds to an ionization event in the detector.
- Position x_{hit} determined by the time-to-space conversion:

$$x_{\text{hit}} = (t_{\text{hit}} - t_0) \cdot v_{\text{drift}}$$

Detection and Hits

- Each hit corresponds to an ionization event in the detector.
- Position x_{hit} determined by the time-to-space conversion:

$$x_{\text{hit}} = (t_{\text{hit}} - t_0) \cdot v_{\text{drift}}$$

- **Challenge:** The time of passage t_0 is typically unknown and must be inferred.
- **Left-right ambiguity:** The hit time alone does not indicate if the muon passed on the left or right side of the detector cell.
- **Solution:** Multiple hits across cells are used to reconstruct the full muon track.

Detector Geometry and Setup

- **Chamber structure:** Each miniDT chamber has 64 cells arranged in 4 layers.
- **Cell dimensions:** $42 \times 13\,\mathrm{mm}^2$ (width $\times$ height), with staggered layers.
- **Muon telescope:** 4 stacked chambers form the telescope, with chamber 1 rotated 90 degrees for orthogonal measurements.
- **Timing:** External scintillators provide precise timing for muon passage t_0.

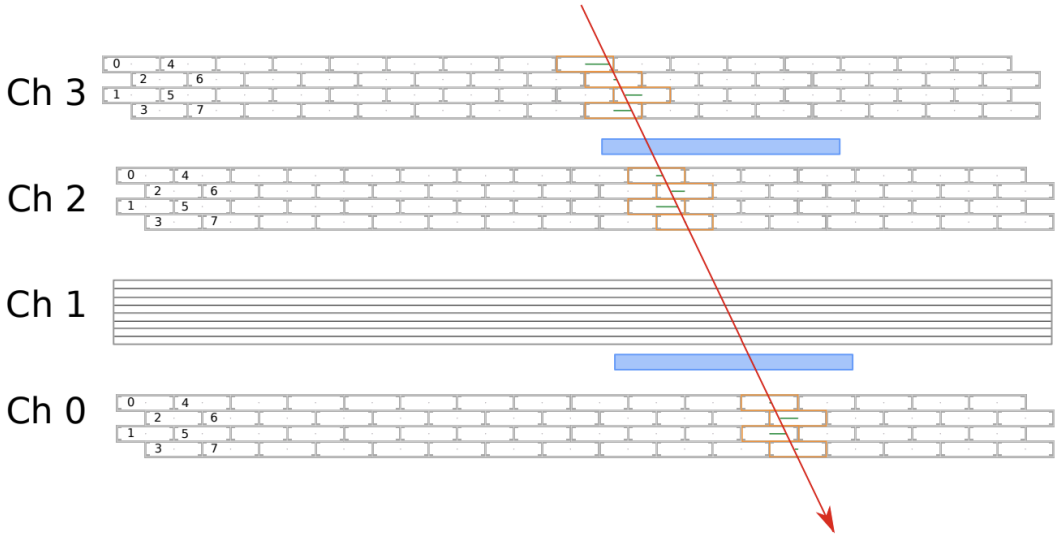
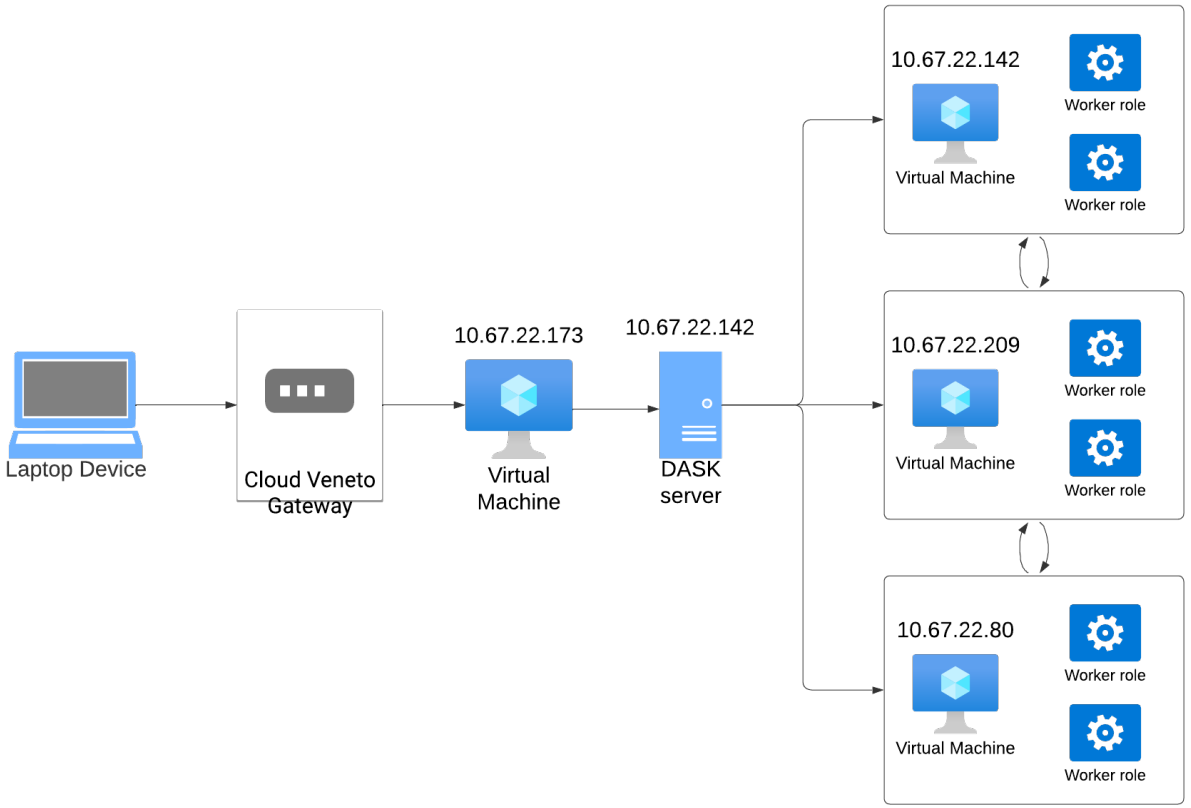


Table of Contents

Network diagram

During the project we set up the following network



Deploy of the SSH Dask Cluster

To deploy the SSH cluster we have at our disposal 3 VMs with 4 CPUs and 8 GB of RAM memory each. We followed the steps below:

Deploy of the SSH Dask Cluster

To deploy the SSH cluster we have at our disposal 3 VMs with 4 CPUs and 8 GB of RAM memory each. We followed the steps below:

1. Set up a passwordless ssh connection between them:
    - `ssh-keygen -t rsa` (to create a new RSA key pair)
    - `ssh-copy-id` (to append the public key to the remote server's ∼`/.ssh/authorized_keys` file containing all public keys authorized to log in to the server without a password.)

Deploy of the SSH Dask Cluster

To deploy the SSH cluster we have at our disposal 3 VMs with 4 CPUs and 8 GB of RAM memory each. We followed the steps below:

1. Set up a passwordless ssh connection between them:
   - `ssh-keygen -t rsa` (to create a new RSA key pair)
   - `ssh-copy-id` (to append the public key to the remote server's ∼`/.ssh/authorized_keys` file containing all public keys authorized to log in to the server without a password.)
2. Create a conda environment installing Dask 2024.6.0 and export the packages in a .yml file:
   - `conda create -name DASK ...`;
   - `conda env export > DASK.yml`;
   - `conda env create -f DASK.yml` (to 'clone' the environment in the other machines).

Deploy of the SSH Dask Cluster

Finally in a jupyter notebook we ran the following command

```
cluster = SSHCluster(
    ["10.67.22.142", "10.67.22.142", "10.67.22.209", "10.67.22.80"],
    worker_options={"nthreads": nthreads, "n_workers": n_workers},
    scheduler_options={"port": 8786, "dashboard_address": ":8787"},
    connect_options={
        "known_hosts": "/home/coradin/.ssh/known_hosts",
        "username": "coradin"
    },
    remote_python="/miniconda3/envs/DASK/bin/python",

)

client = Client(cluster, timeout='180s')
client.wait_for_workers(n_workers)
```

Client

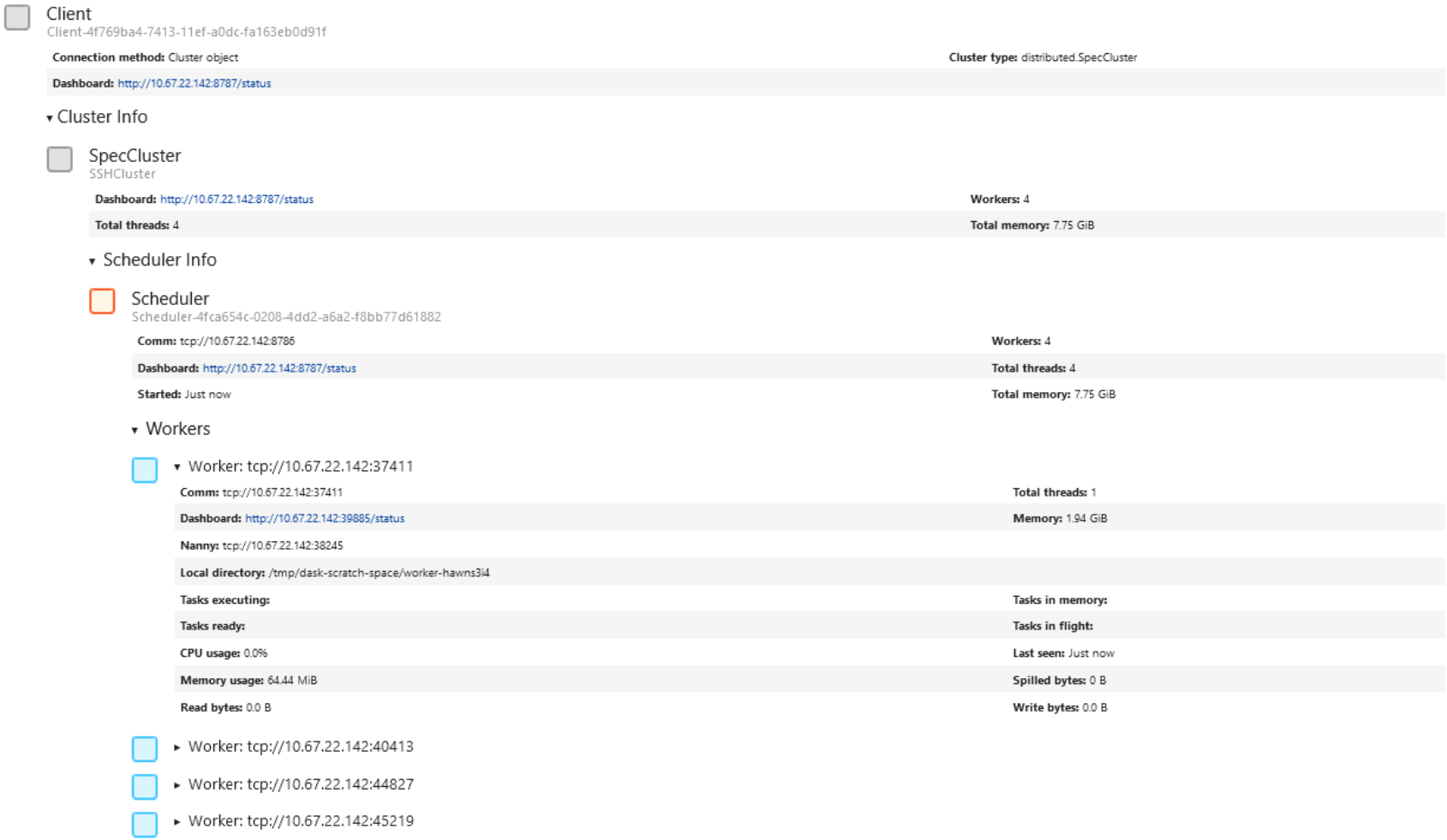

Client
Client-4f769ba4-7413-11ef-a0dc-fa163eb0d91f
Connection method: Cluster object
Cluster type: distributed.SpecCluster
Dashboard: http://10.67.22.142:8787/status
Cluster Info
SpecCluster
SSHCluster
Dashboard: http://10.67.22.142:8787/status
Workers: 4
Total threads: 4
Total memory: 7.75 GiB
Scheduler Info
Scheduler
Scheduler-4fca654c-0208-4dd2-a6a2-f8bb77d61882
Comm: tcp://10.67.22.142:8786
Workers: 4
Dashboard: http://10.67.22.142:8787/status
Total threads: 4
Started: Just now
Total memory: 7.75 GiB
Workers
Worker: tcp://10.67.22.142:37411
Comm: tcp://10.67.22.142:37411
Total threads: 1
Dashboard: http://10.67.22.142:39885/status
Memory: 1.94 GiB
Nanny: tcp://10.67.22.142:38245
Local directory: /tmp/dask-scratch-space/worker-hawns3i4
Tasks executing:
Tasks in memory:
Tasks ready:
Tasks in flight:
CPU usage: 0.0%
Last seen: Just now
Memory usage: 64.44 MiB
Spilled bytes: 0 B
Read bytes: 0.0 B
Write bytes: 0.0 B
Worker: tcp://10.67.22.142:40413
Worker: tcp://10.67.22.142:44827
Worker: tcp://10.67.22.142:45219

Table of Contents

Data Conversion and Detector Mapping

In order to clean the data, we're extracting specific fields from raw binary data using bitwise operations (on the right), filtering for `HEAD == 2`, and writing the results into a pandas DataFrame.

```
0-4 (5 bits) -> tdc_meas = (byte_array >> 0) & 0x1F
5-16 (12 bits) -> bx = (byte_array >> 5) & 0xFFF
17-48 (32 bits) -> orb_cnt = (byte_array >> 17) & 0xFFFFFFFF
49-57 (9 bits) -> tdc_chan = (byte_array >> 49) & 0x1FF
58-60 (3 bits) -> fpga = (byte_array >> 58) & 0x7
61-63 (3 bits) -> head = (byte_array >> 61) & 0x7
```

Data Conversion and Detector Mapping

In order to clean the data, we're extracting specific fields from raw binary data using bitwise operations (on the right), filtering for `HEAD == 2`, and writing the results into a pandas DataFrame.

```
0-4 (5 bits) -> tdc_meas = (byte_array >> 0) & 0x1F
5-16 (12 bits) -> bx = (byte_array >> 5) & 0xFFF
17-48 (32 bits) -> orb_cnt = (byte_array >> 17) & 0xFFFFFFFF
49-57 (9 bits) -> tdc_chan = (byte_array >> 49) & 0x1FF
58-60 (3 bits) -> fpga = (byte_array >> 58) & 0x7
61-63 (3 bits) -> head = (byte_array >> 61) & 0x7
```

We use the `FPGA` and `TDC_CHANNEL` values to determine which chamber each data point belongs to, assigning it to the chambers 0 through 3.

```
Chamber 0 → (FPGA = 0) AND (CHANNEL in [0-63])
Chamber 1 → (FPGA = 0) AND (CHANNEL in [64-127])
Chamber 2 → (FPGA = 1) AND (CHANNEL in [0-63])
Chamber 3 → (FPGA = 1) AND (CHANNEL in [64-127])
```

Scintillator Timing Alignment

- We extract the scintillator signal (`FPGA == 1`, `TDC_CHANNEL == 128`) to calculate the muon passage time (`t0`).
- Calculate the muon passage time (in ns), `t0`:

$$t0 = 25 \times (BX + \frac{\texttt{TDC\_MEAS}}{30})$$

- After removing the scintillator rows from the main data, we merge the `t0` information back with the TDC hits using `ORB_CNT`.

```python
def filter_groups_part1(df):
    # Filter with scintillator's information
    scintillator_df = df[(df.FPGA == 1) & (df.TDC_CHANNEL == 128)].drop_duplicates(subset='ORB_CNT', keep='first')
    scintillator_df['t0'] = 25 * (scintillator_df['BX'] + scintillator_df['TDC_MEAS'] / 30)
    scintillator_df = scintillator_df[['ORB_CNT', 't0']]

    # Remove scintillator info from the main DataFrame
    df = df[df.TDC_CHANNEL < 128]

    df = df.merge(scintillator_df, on='ORB_CNT', how='inner')
    return df
```

Chamber Assignment and Positioning

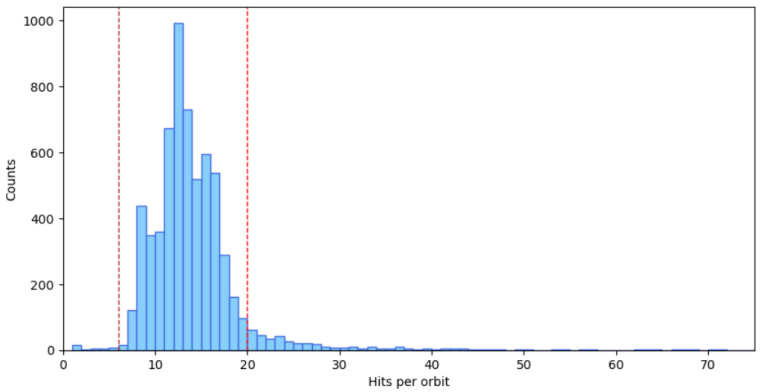
- We assign each hit to a chamber based on `FPGA` and `TDC_CHANNEL` as it was previously explained.
- Calculate the initial `relative_time` using `BX` and `TDC_MEAS`, then adjust by subtracting the muon passage time (`t0`).
- Apply chamber-specific time offsets to correct for delays in each chamber.
- Calculate hit positions (`x_left` and `x_right`) based on drift velocity.
- Filter out non-physical positions outside the detector's boundaries to keep only valid data.

Code Sample - Chamber Assignment and Positioning


```python
def filter_groups_part2_a(df):
    # Assign chamber information
    df['Chamber'] = df.apply(assign_chamber, axis=1, meta=('Chamber', 'float64'))

    # Add further information
    df['relative_time'] = 25 * (df['BX'] + df['TDC_MEAS'] / 30)
    df = df.drop(columns=['BX', 'TDC_MEAS'])

    # Calculate relative time by subtracting t0
    df['relative_time'] = df['relative_time'] - df['t0']
    df = df.drop(columns=['t0'])

    # Add time offsets based on the chamber
    df['relative_time'] = df['relative_time'] + df['Chamber'].map(time_offset_by_chamber)

    # Calculate x_left and x_right positions based on the drift velocity
    df['x_left']  = - V_DRIFT*df['relative_time']
    df['x_right'] = + V_DRIFT*df['relative_time']
    df['z'] = 0

    # Drop non-physical positions
    df = df[(df.x_left.between(-CELL_WIDTH/2, 0)) & (df.x_right.between(0, CELL_WIDTH/2))]

    return df
```

Filtering and Geometric Adjustments

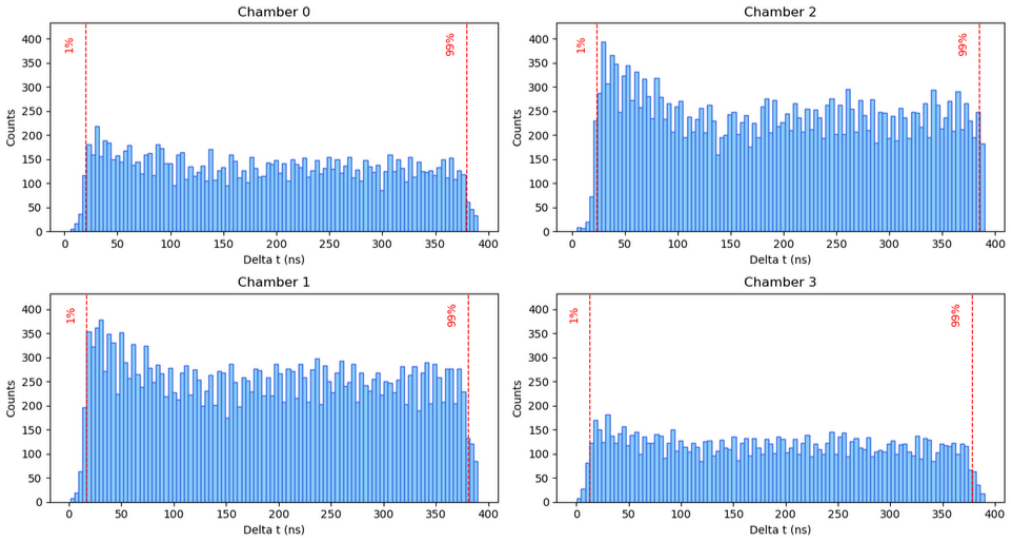
- Filter out events with fewer hits than the minimum required per chamber and those with total hits outside the valid range for each orbit.
- Adjust positions for non-Chamber 1 hits using `x_left` and `x_right` based on geometry from `df_bricks`.
- For Chamber 1, use y-coordinates for adjustments due to its unique layout.
- Adjust z-coordinates for all hits based on the detector layout from `df_bricks`.




```python
def filter_groups_part2_b(df):
    # Drop events based on number of hits

    #Remove hits of chambers with less than MIN_HITS_PER_CHAMBER hits
    grouped = df.groupby(['ORB_CNT', 'Chamber']).filter(lambda x: len(x) >= MIN_HITS_PER_CHAMBER)
    df = df.loc[df.ORB_CNT.isin(grouped.ORB_CNT)]

    # Remove events based on total number of hits
    grouped = df.groupby('ORB_CNT').filter(lambda x: MIN_HIT_ORB <= len(x) <= MAX_HIT_ORB)
    df = df.loc[df.ORB_CNT.isin(grouped.ORB_CNT)]

    # Create a mask for Chamber 1
    mask_chamber_1 = (df['Chamber'] == 1)

    # For Chambers other than 1, adjust x_left and x_right positions
    df.loc[~mask_chamber_1, 'x_left']  = df.loc[~mask_chamber_1, 'x_left']  + df_bricks.loc[(df.loc[~mask_chamber_1, 'Chamber']//2)*128
                                                                            + df.loc[~mask_chamber_1, 'TDC_CHANNEL'], 'x'].values

    df.loc[~mask_chamber_1, 'x_right'] = df.loc[~mask_chamber_1, 'x_right'] + df_bricks.loc[(df.loc[~mask_chamber_1, 'Chamber']//2)*128
                                                                            + df.loc[~mask_chamber_1, 'TDC_CHANNEL'], 'x'].values

    # For Chamber 1, adjust x_left and x_right positions
    df.loc[mask_chamber_1, 'x_left']  = df.loc[mask_chamber_1, 'x_left']  + df_bricks.loc[(df.loc[mask_chamber_1, 'Chamber']//2)*128
                                                                          + df.loc[mask_chamber_1, 'TDC_CHANNEL'], 'y'].values
    df.loc[mask_chamber_1, 'x_right'] = df.loc[mask_chamber_1, 'x_right'] + df_bricks.loc[(df.loc[mask_chamber_1, 'Chamber']//2)*128
                                                                          + df.loc[mask_chamber_1, 'TDC_CHANNEL'], 'y'].values

    # Adjust z positions based on the bricks information
    df['z'] = df['z'] + df_bricks.loc[(df['Chamber']//2)*128 + df['TDC_CHANNEL'], 'z'].values

    return df
```

Timing Distribution Investigation

- We observe a homogeneous distribution, indicating well-aligned timing across all chambers with no outliers.
- When plotting the hit counts per chamber, Chamber 2 consistently shows the highest number of hits compared to the other chambers.



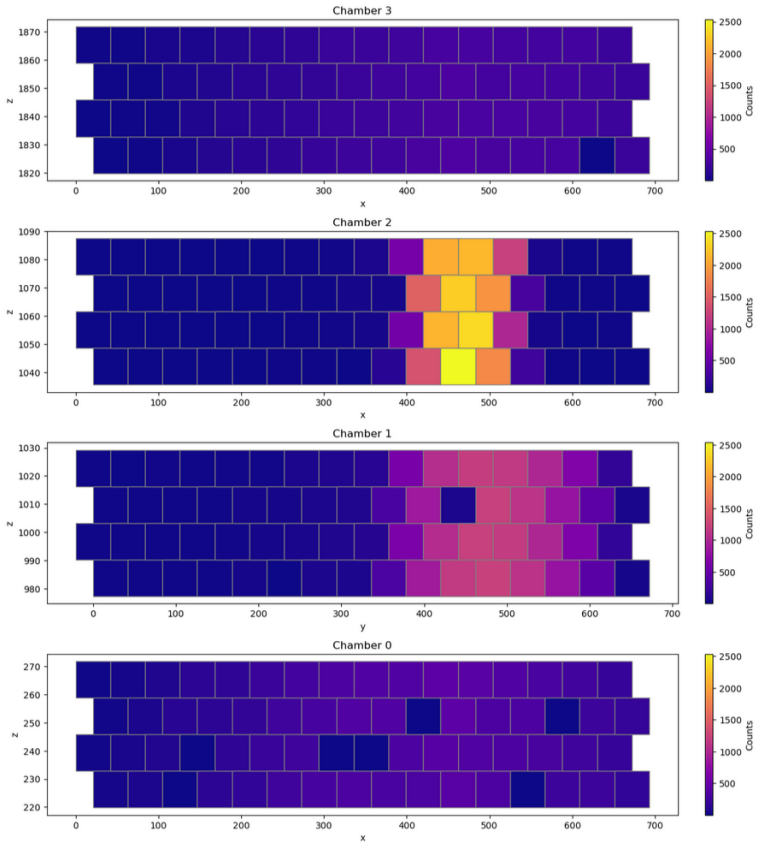


Table of Contents

Local Track Fitting Overview

- For each chamber, a linear fit reconstructs the muon trajectory.
- **Key Parameters**:
  - **Slope (angular parameter)**: Describes the angle of the muon's path.
  - **Intercept**: Defines where the muon crossed the chamber.

Local Track Fitting Overview

- For each chamber, a linear fit reconstructs the muon trajectory.
- **Key Parameters**:
  - **Slope (angular parameter)**: Describes the angle of the muon's path.
  - **Intercept**: Defines where the muon crossed the chamber.
- The fit selects the best combination of left/right hits by minimizing:

$$\chi^2/\text{ndof} = \frac{\sum \left(y_{\text{observed}} - y_{\text{fitted}}\right)^2}{\text{ndof}}$$

- The least-squares fit identifies the highest quality track by minimizing this value.

Fitting Algorithm

Step-by-Step Explanation of the Algorithm:

- **Grouping Hits by z**: Hits are grouped by their z coordinate, as multiple hits can occur at the same z.
- **Generating Combinations**: All possible combinations of left and right hits are generated using `np.meshgrid`.
- **Least-Squares Fit**: A design matrix A is constructed and the fit is solved using:

  `coeffs, residuals, _, _ = np.linalg.lstsq(A,` $\mathtt{x}_{\mathtt{combinations}}.T$`)`
- **Selection of Best Fit**: The combination with the lowest χ^2/ndof is selected.

Code Sample - Vectorized Fit

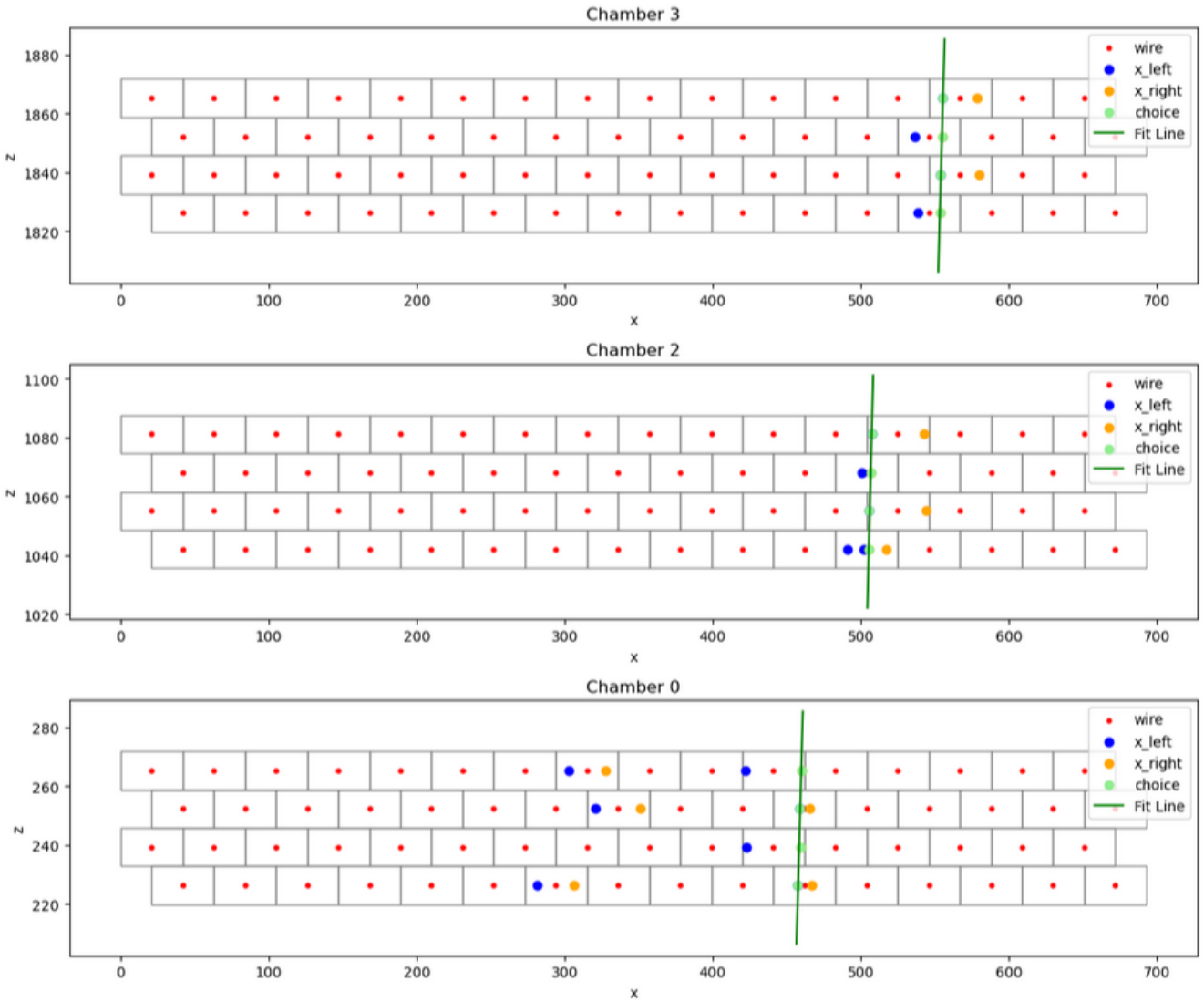

```python
def fit_tracks_vectorized(group):

    x_left = group['x_left'].values
    x_right = group['x_right'].values
    z = group['z'].values
    chamber = group['Chamber'].values[0]
    orb_cnt = group['ORB_CNT'].values[0]

    # Group the data by 'z' to ensure we are considering all combinations of points with the same z
    grouped_by_z = group.groupby('z')

    # List to store all combinations of x-values for different z values
    x_combinations = []
    list_stacks = []

    # Loop through each unique z value and get combinations of x_left and x_right
    for z_value, sub_group in grouped_by_z:
        x_left_sub = sub_group['x_left'].values
        x_right_sub = sub_group['x_right'].values

        # Stack left and right x-values for this z
        stack = np.column_stack((x_left_sub, x_right_sub)).flatten()
        list_stacks.append(stack)

    # Use numpy.meshgrid to create a multi-dimensional grid
    meshgrid_result = np.meshgrid(*list_stacks, indexing='ij')

    # Reshape and flatten the resulting arrays into a list of combinations
    x_combinations = np.array([grid.ravel() for grid in meshgrid_result]).T
    x_combinations_all = np.vstack(x_combinations)
    z_rows = np.sort(np.unique(group['z']))

    # Matrix for least squares: z-values and a column of ones for the intercept term
    A = np.vstack((z_rows, np.ones(len(z_rows)))).T

    # Perform least-squares fit on all combinations at once
    coeffs, residuals, _, _ = np.linalg.lstsq(A, x_combinations_all.T, rcond=None)

    # Compute chi-squared/ndof for all combinations
    chi_squared_ndof = residuals / (len(z_rows) - 2)  # Adjust degrees of freedom
```

Fitting Results and Visualization

- After fitting, the results include:
  - Best-fit slope and intercept.
  - Chi-squared per degree of freedom (χ^2/ndof) for each fit.
  - Best combination of hit positions.

	Chamber	ORB_CNT	x_values	z_values	Reg_Coefficient	Reg_Intercept	Chi_Squared_NDOF
0	0	1123056	[457.3068466666669, 460.26398666666654, 460.44...	[226.3, 239.3, 252.3, 265.3]	0.130101	428.246103	0.667025
1	0	1135298	[227.56031999999922, 232.68932, 236.3226800000...	[226.3, 239.3, 252.3, 265.3]	0.324264	154.539077	0.232197
2	0	1144838	[427.69698666666676, 427.42984666666666, 428.2...	[226.3, 239.3, 265.3]	0.016299	423.816884	0.128502
3	0	1190130	[463.3306533333335, 464.09282, 462.7048466666665]	[226.3, 252.3, 265.3]	-0.009566	465.748221	0.894160
4	0	1206251	[338.7653199999998, 344.0556533333333, 345.268...	[226.3, 252.3, 265.3]	0.171984	300.050004	0.586553
...	...	...	...	...	...	...	...
1743	3	48538547	[284.50071333333307, 281.6026200000008, 278.53...	[1826.3, 1839.3, 1852.3, 1865.3]	-0.203227	655.455789	0.194375
1744	3	48545828	[392.1619533333338, 388.73854666666693, 380.28...	[1826.3, 1839.3, 1852.3]	-0.456955	1227.538107	4.223560
1745	3	48547426	[565.586786666667, 566.2432133333331, 568.8327...	[1826.3, 1839.3, 1865.3]	0.085569	409.129580	0.116430
1746	3	48559795	[543.9878799999992, 547.5028799999992, 550.119...	[1826.3, 1839.3, 1852.3, 1865.3]	0.199530	180.050455	0.438301
1747	3	48576415	[253.16028666666674, 250.13845333333327, 248.0...	[1826.3, 1839.3, 1852.3, 1865.3]	-0.234707	681.982443	0.492624

Local fit plot



Dask Parallelization for Local Fitting

- The dataset is split into partitions, and fitting is performed on each partition independently:

  `df.map_partitions(...).apply(fit_tracks_vectorized)`

- Results from each partition are combined to produce the full dataset of fitted tracks.

Table of Contents

Global fit

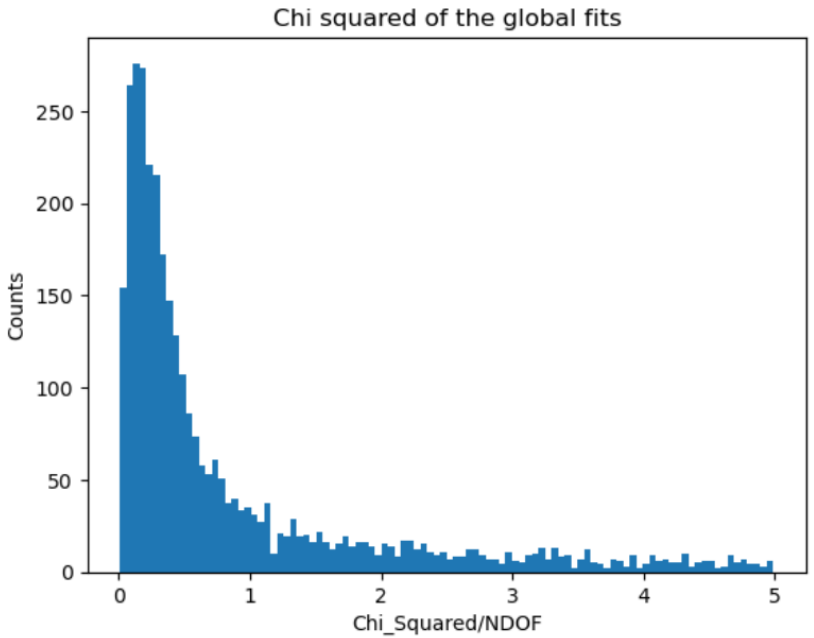
Starting from the local fit dataframe, we grouped the rows by orbit and fit the points included in the selected "best local fit" combinations and in Chambers 0, 2 and 3

	ORB_CNT	Reg_Coefficient	Reg_Intercept	Chi_Squared_NDOF
0	1118948	-0.262812	772.409738	0.260344
1	1123056	0.077788	441.086896	0.759312
2	1124284	-0.057558	486.015703	0.313131
3	1126947	0.091026	355.219645	0.521112
4	1130581	0.199468	285.058922	102.047355

Global fit

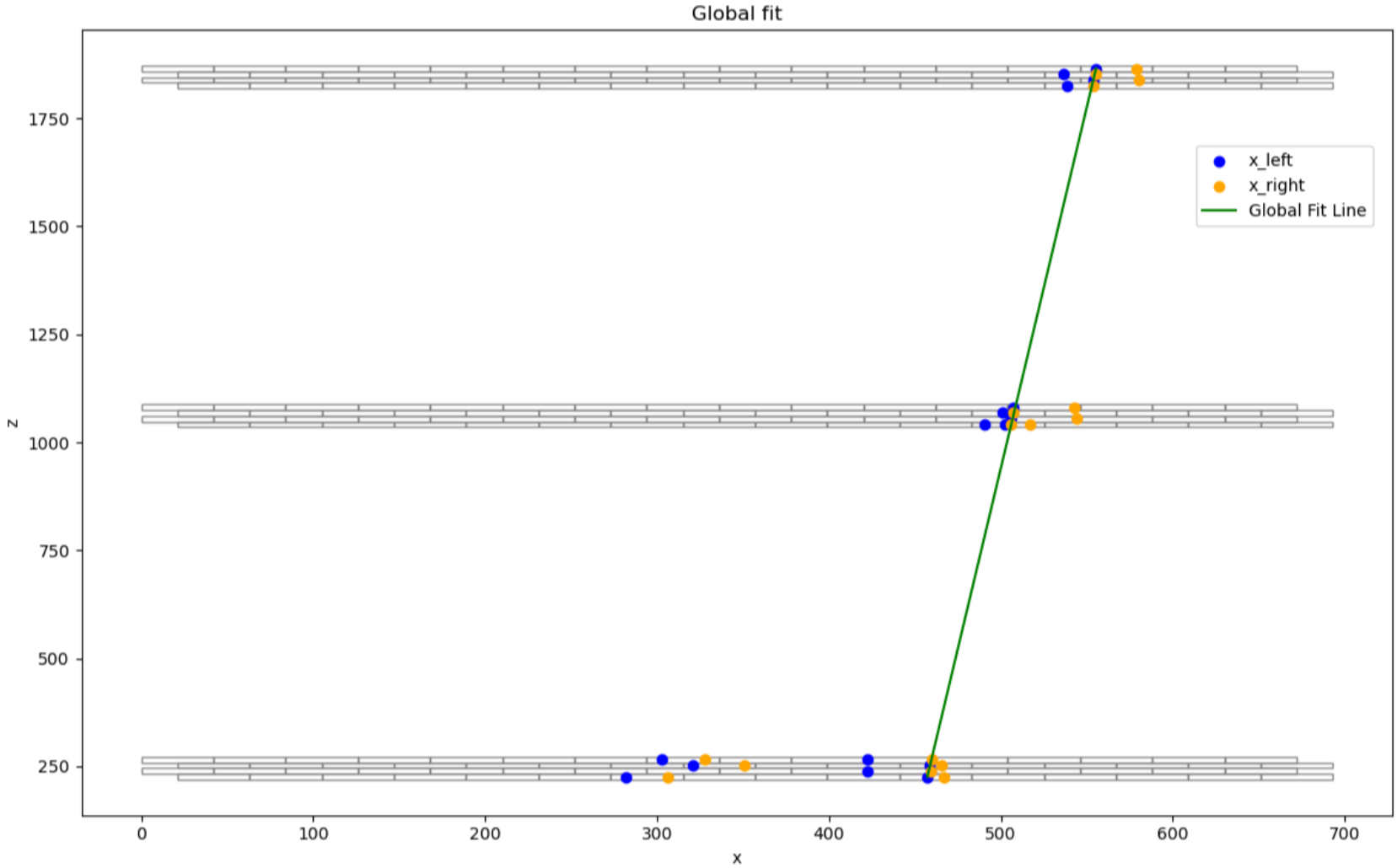
Starting from the local fit dataframe, we grouped the rows by orbit and fit the points included in the selected "best local fit" combinations and in Chambers 0, 2 and 3

	ORB_CNT	Reg_Coefficient	Reg_Intercept	Chi_Squared_NDOF
0	1118948	-0.262812	772.409738	0.260344
1	1123056	0.077788	441.086896	0.759312
2	1124284	-0.057558	486.015703	0.313131
3	1126947	0.091026	355.219645	0.521112
4	1130581	0.199468	285.058922	102.047355



We computed a simple linear regression: once again we used $\frac{\chi^2}{ndof}$ to evaluate the fit

Single-orbit trajectory plot



Execution


```
global_tracks_df_all = df_regression_all.map_partitions(
    lambda partition: (
        partition
        .groupby(['ORB_CNT'])
        .apply(fit_global_tracks, include_groups=True)
    ),
    meta=meta).dropna().reset_index(drop=True).rename(columns={'ORB_CNT_': 'ORB_CNT'}).repartition(npartitions=1)

global_tracks_df_all.visualize(True)
```

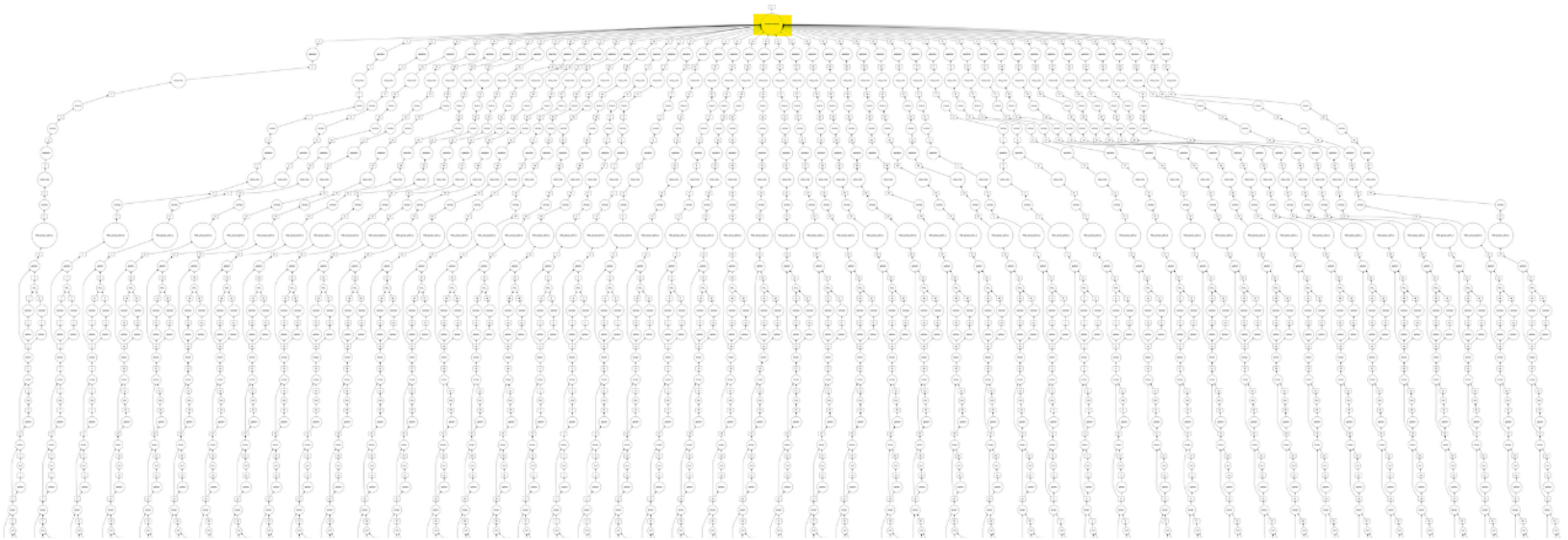


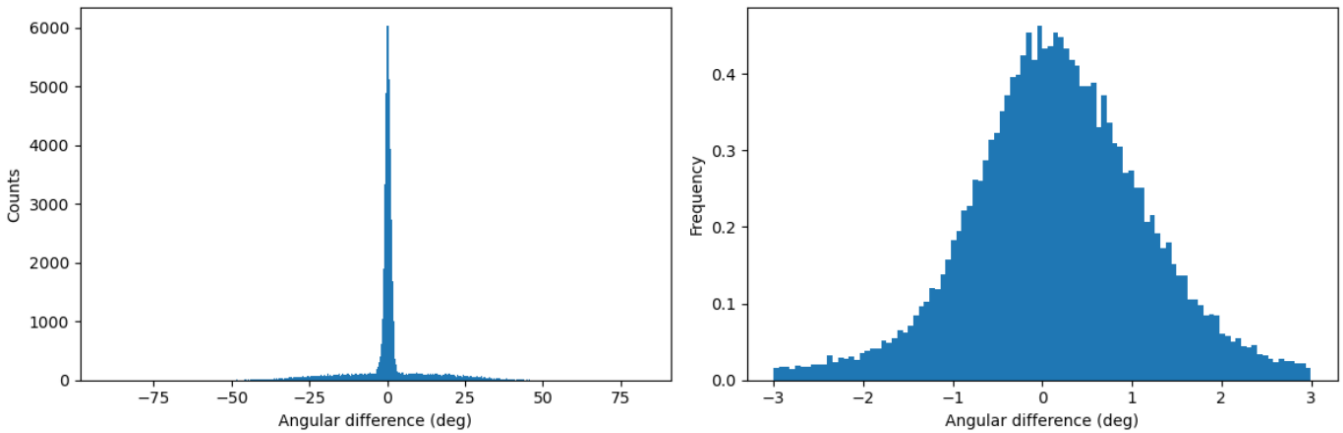
Table of Contents

Angular difference

- We cleansed the dataset by excluding all rows with $\frac{\chi^2}{ndof} > 0.7$
- We computed the angular difference between the global and the local fit slope to get an estimate of the detector resolution

Angular difference

- We cleansed the dataset by excluding all rows with $\frac{\chi^2}{ndof} > 0.7$
- We computed the angular difference between the global and the local fit slope to get an estimate of the detector resolution

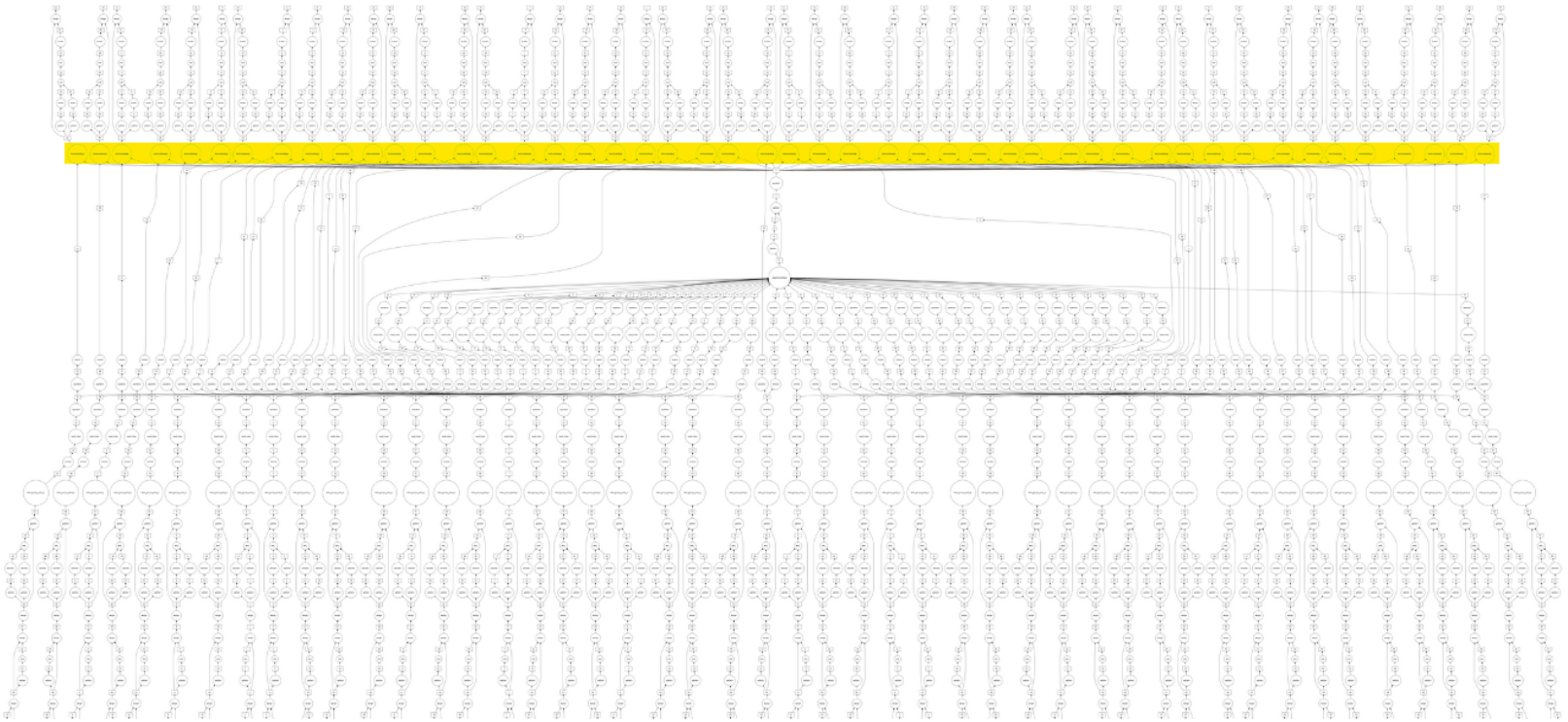


- Approximately Gaussian distribution with $\mu = 0.16^o$, $\sigma = 0.99^o$

Execution


```
df_ang_diff_all = global_fit_df_rn_all.merge(regression_results_df_ch2_rn_all, on = ['ORB_CNT'])

df_ang_diff_all['Angular difference'] = (np.arctan(df_ang_diff_all['Reg_Coefficient_glo']) - np.arctan(df_ang_diff_all['Reg_Coefficient_loc']))*180/np.pi
```

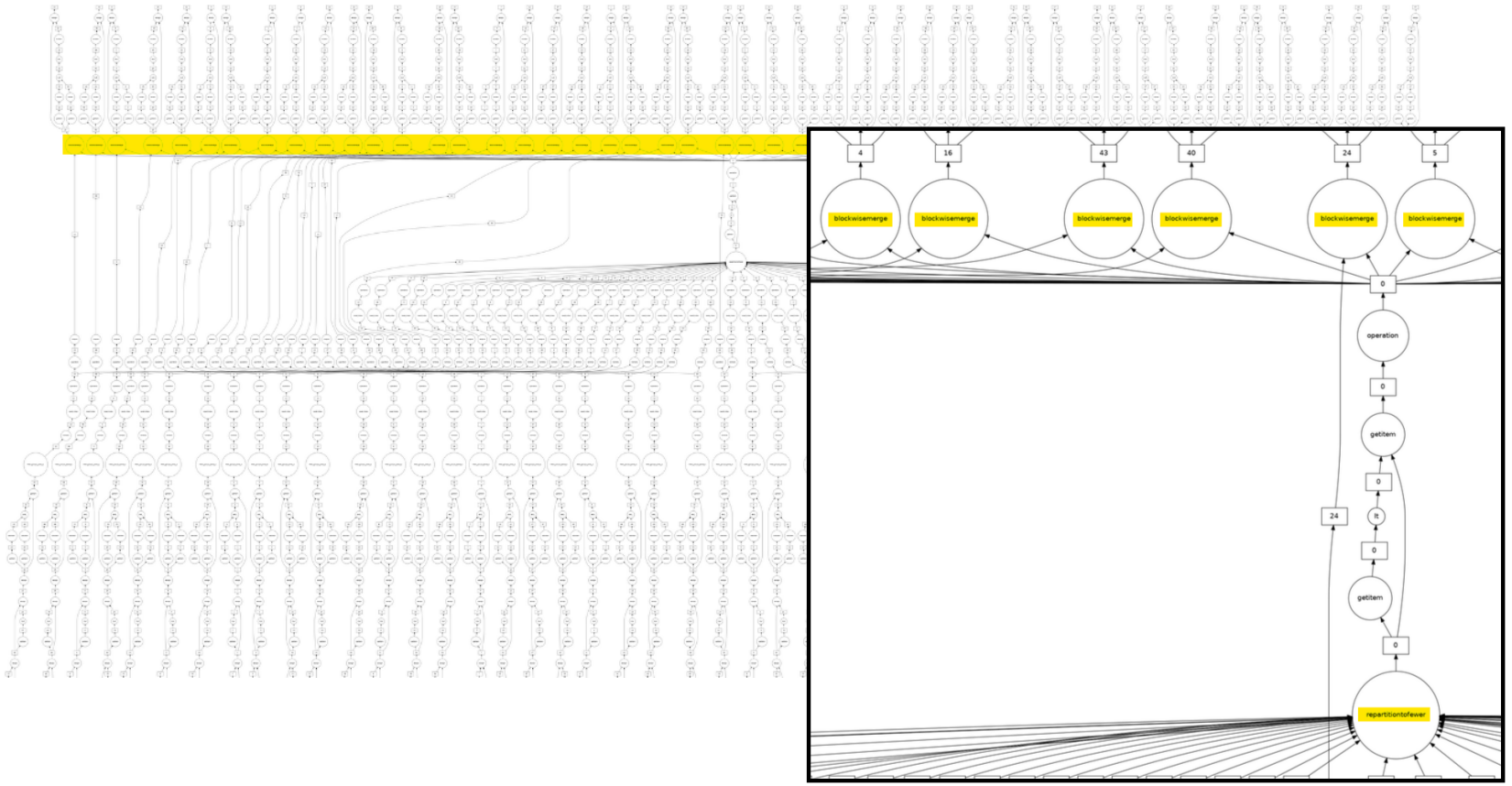
```
df_ang_diff_all.visualize(True)
```



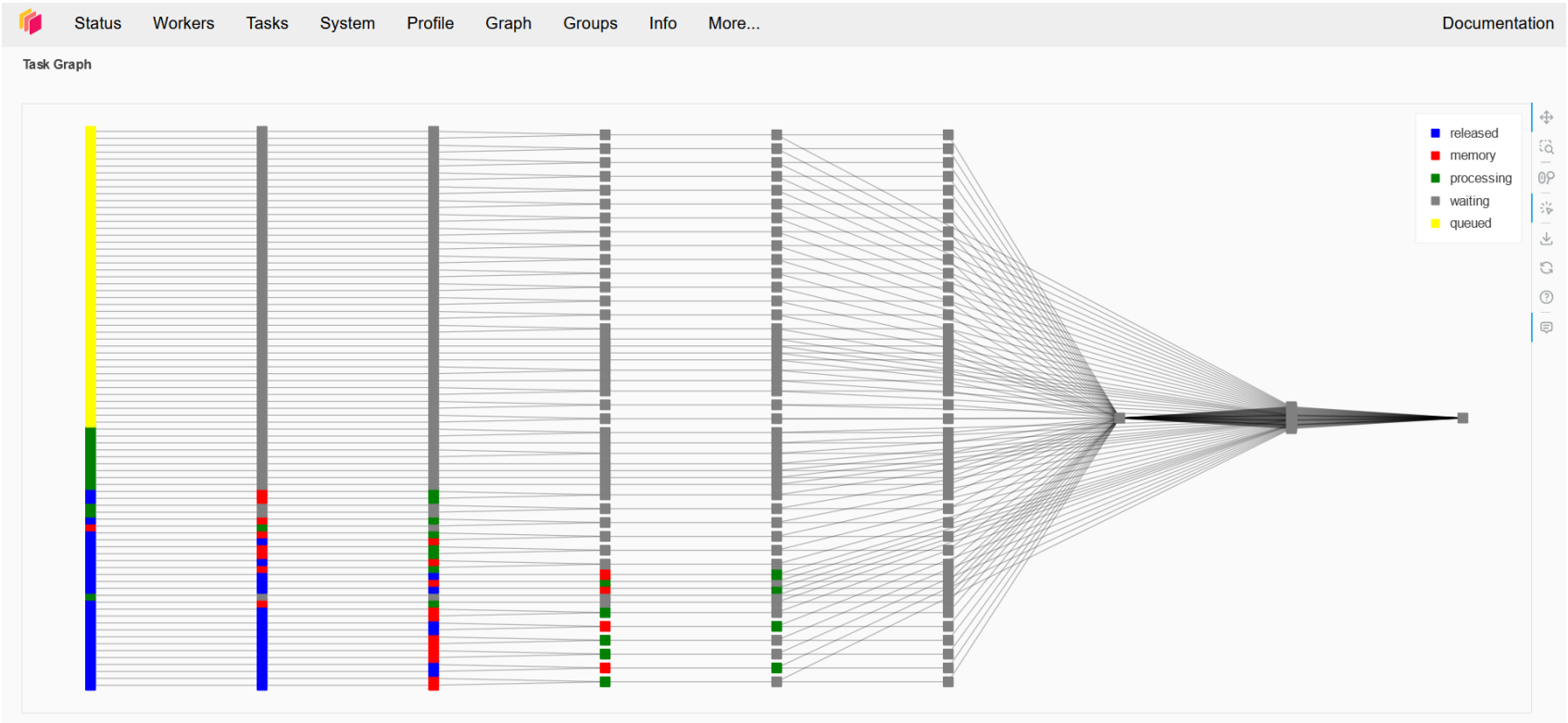
Execution

```
df_ang_diff_all = global_fit_df_rn_all.merge(regression_results_df_ch2_rn_all, on = ['ORB_CNT'])

df_ang_diff_all['Angular difference'] = (np.arctan(df_ang_diff_all['Reg_Coefficient_glo']) - np.arctan(df_ang_diff_all['Reg_Coefficient_loc']))*180/np.pi
```

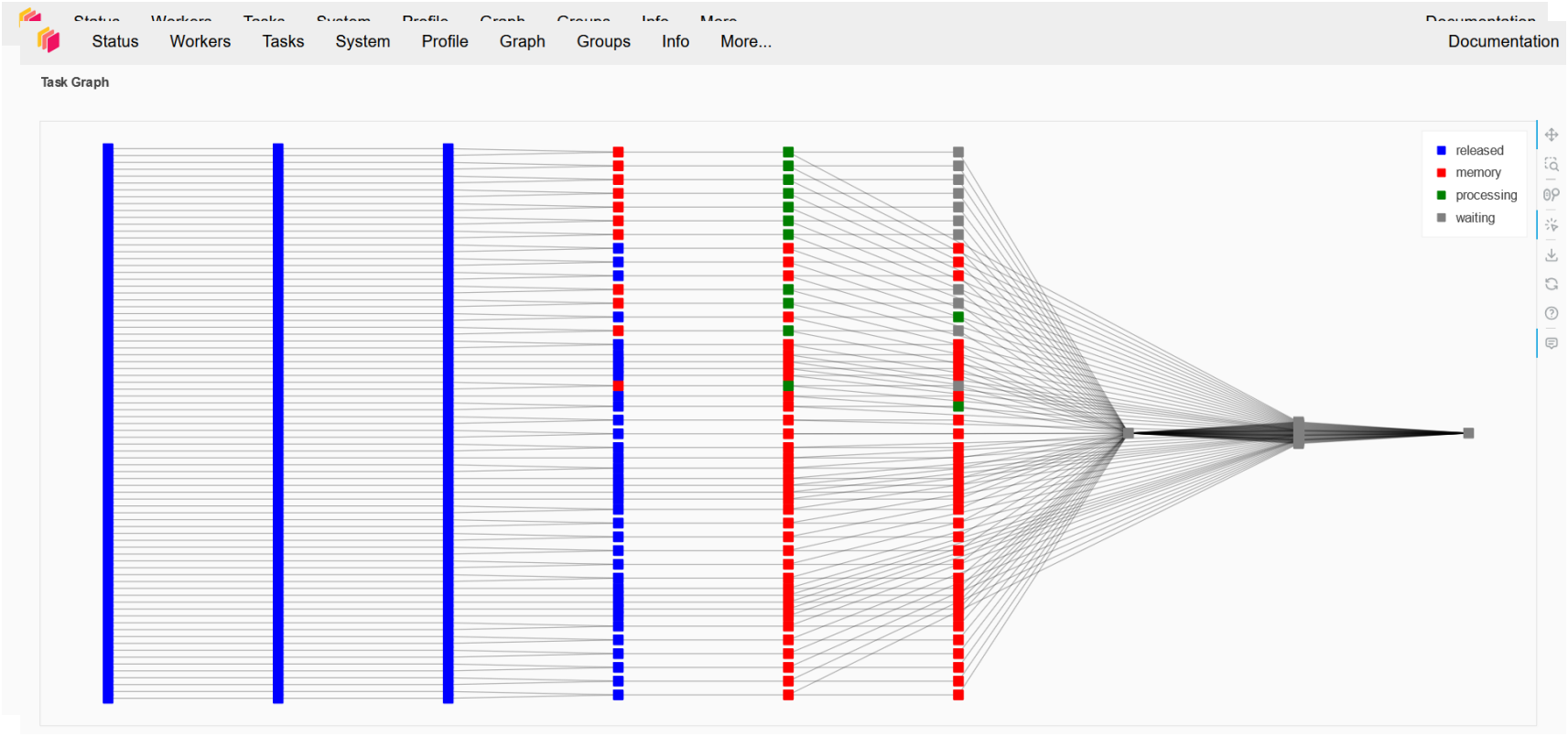
```
df_ang_diff_all.visualize(True)
```



Dask Scheduler Graph



Dask Scheduler Graph



Dask Scheduler Graph

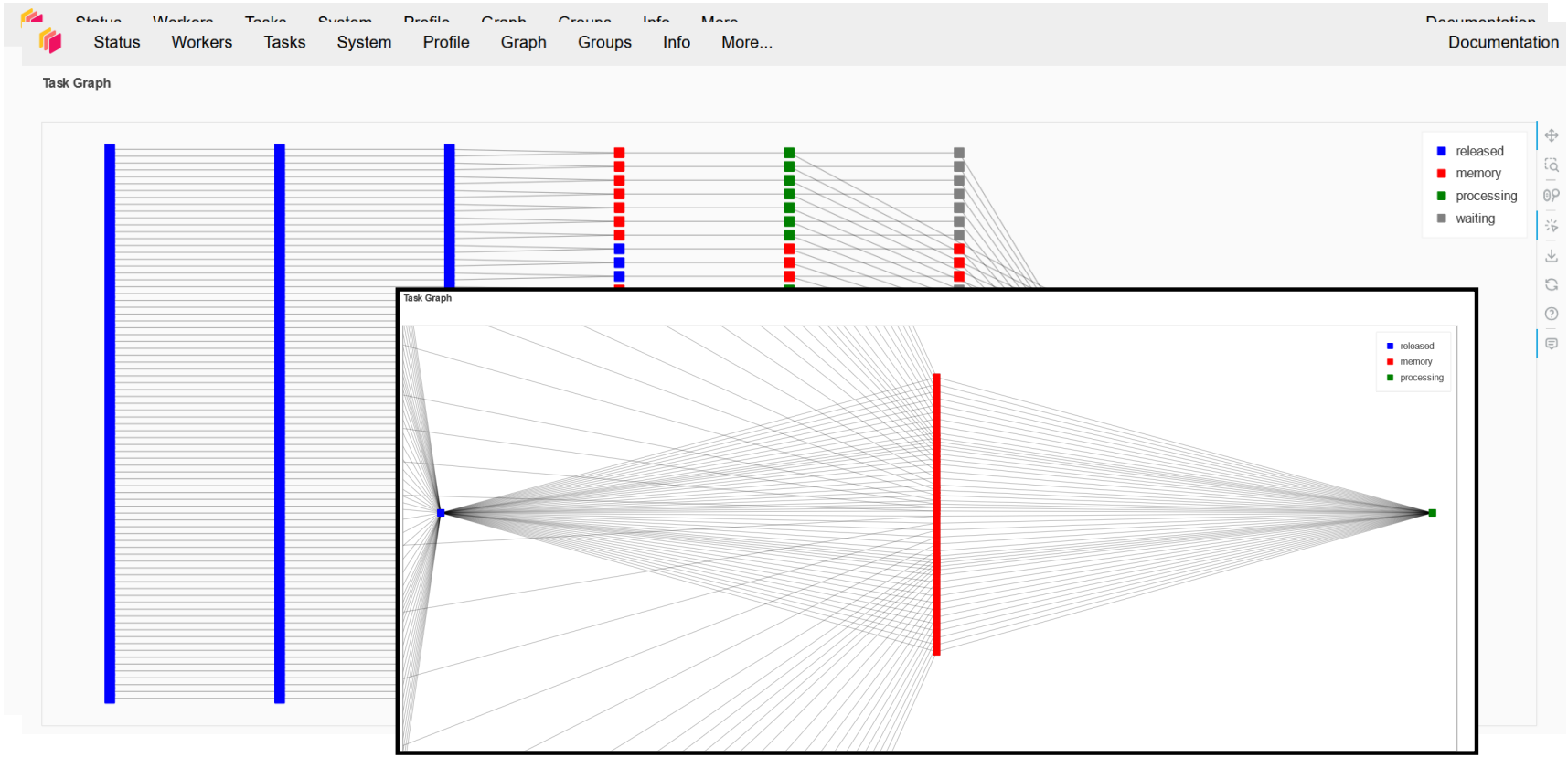


Table of Contents

What do we want to measure?

We would like to understand how much time is spent on the "milestone" code chunks of the project, namely:

1. Reading and filtering
2. Performing the local fits
3. Performing the global fits
4. Computing the angular differences

What do we want to measure?

We would like to understand how much time is spent on the "milestone" code chunks of the project, namely:

1. Reading and filtering
2. Performing the local fits
3. Performing the global fits
4. Computing the angular differences

Technical question

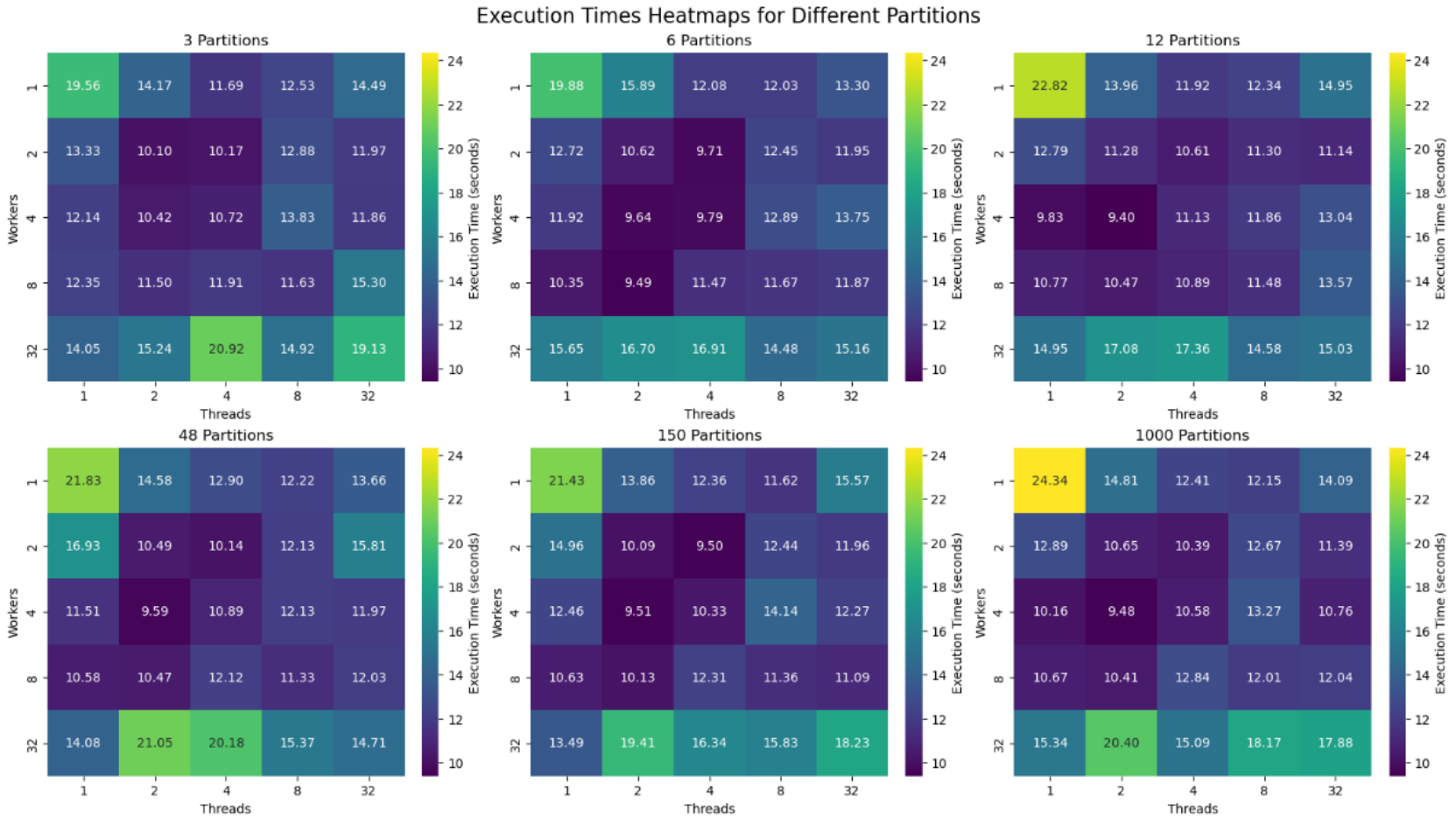
Our implementation relies on delayed operations... How can we measure the time in the intermediate stages of execution?

Strategy

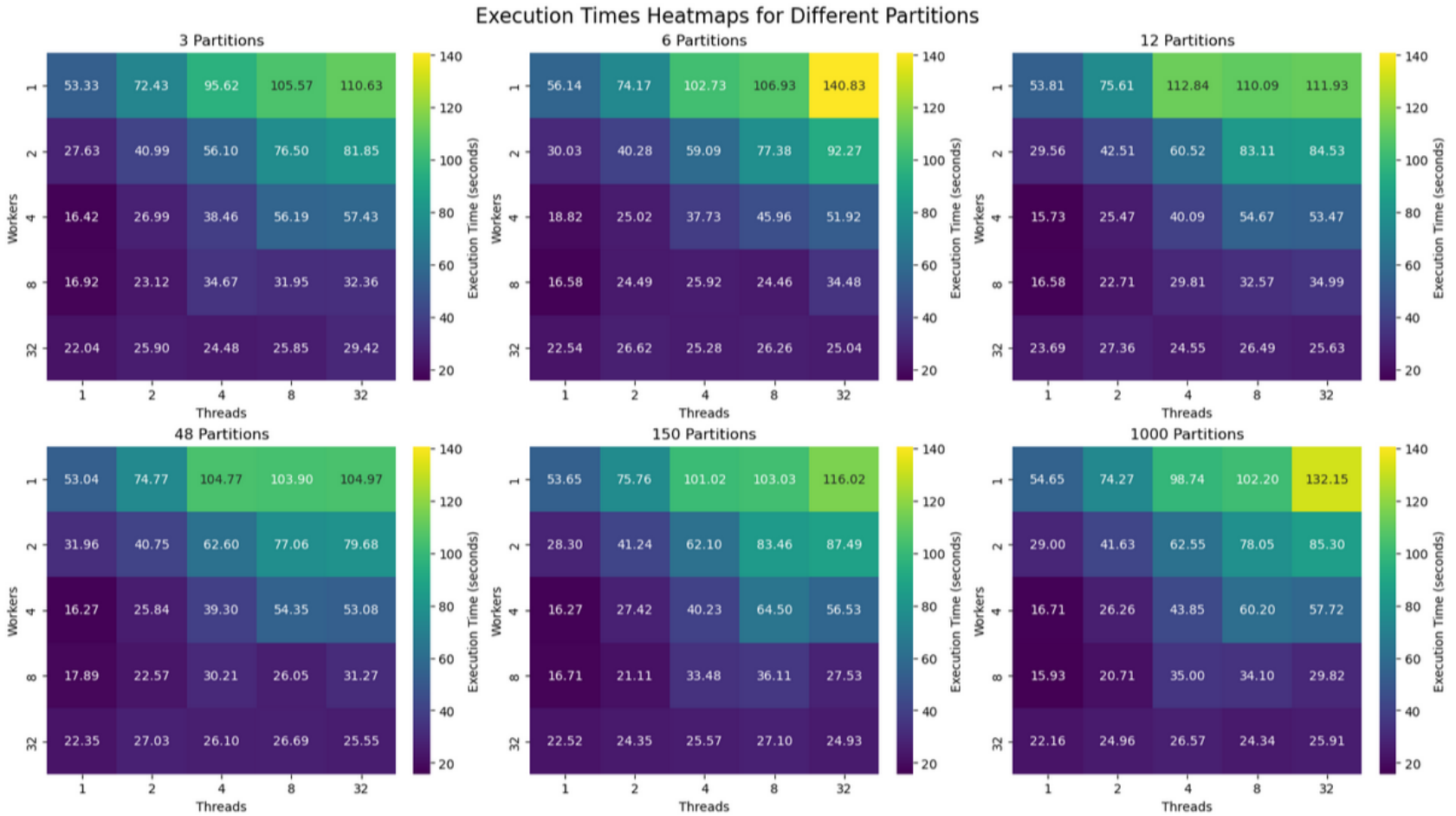
1. Define the operations.
2. Call persist() to trigger the execution of the tasks.
3. Wait for the computation to finish using the wait() function from the Dask.distributed library.

```
timings = {}

filename_all = 's3://project-bkp/data_0000*'
df_all = read_file(filename_all)

df_all = df_all.repartition(npartitions=n_partitions)
df_all = df_all.map_partitions(filter_groups_part1)

wall_start_time = time.time()
df_all = filter_groups_part2_a(df_all)
df_all = df_all.map_partitions(filter_groups_part2_b)
df_all = df_all.persist()
wait(df_all)
timings['filtering'] = time.time() - wall_start_time
```

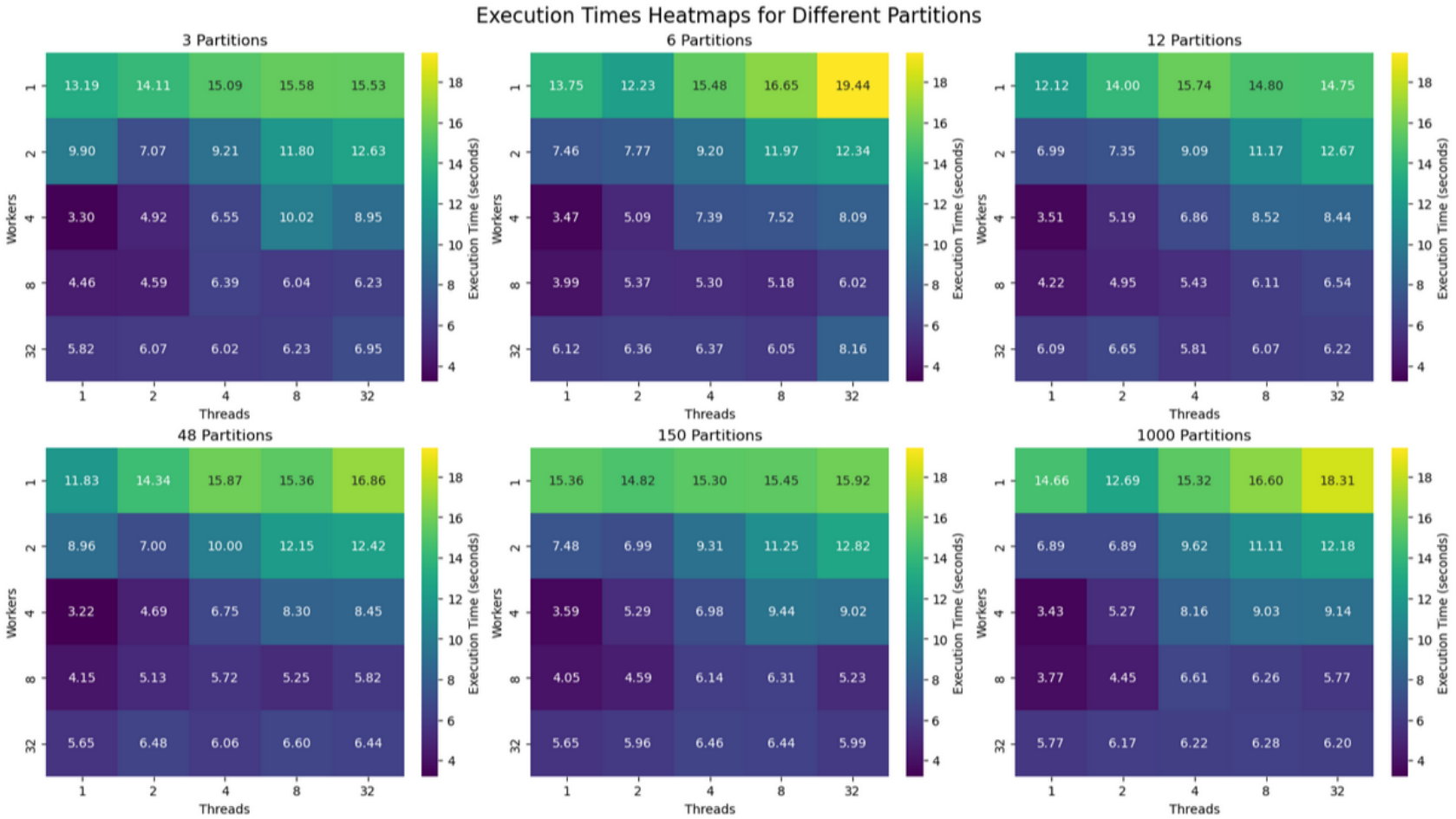
Reading and filtering



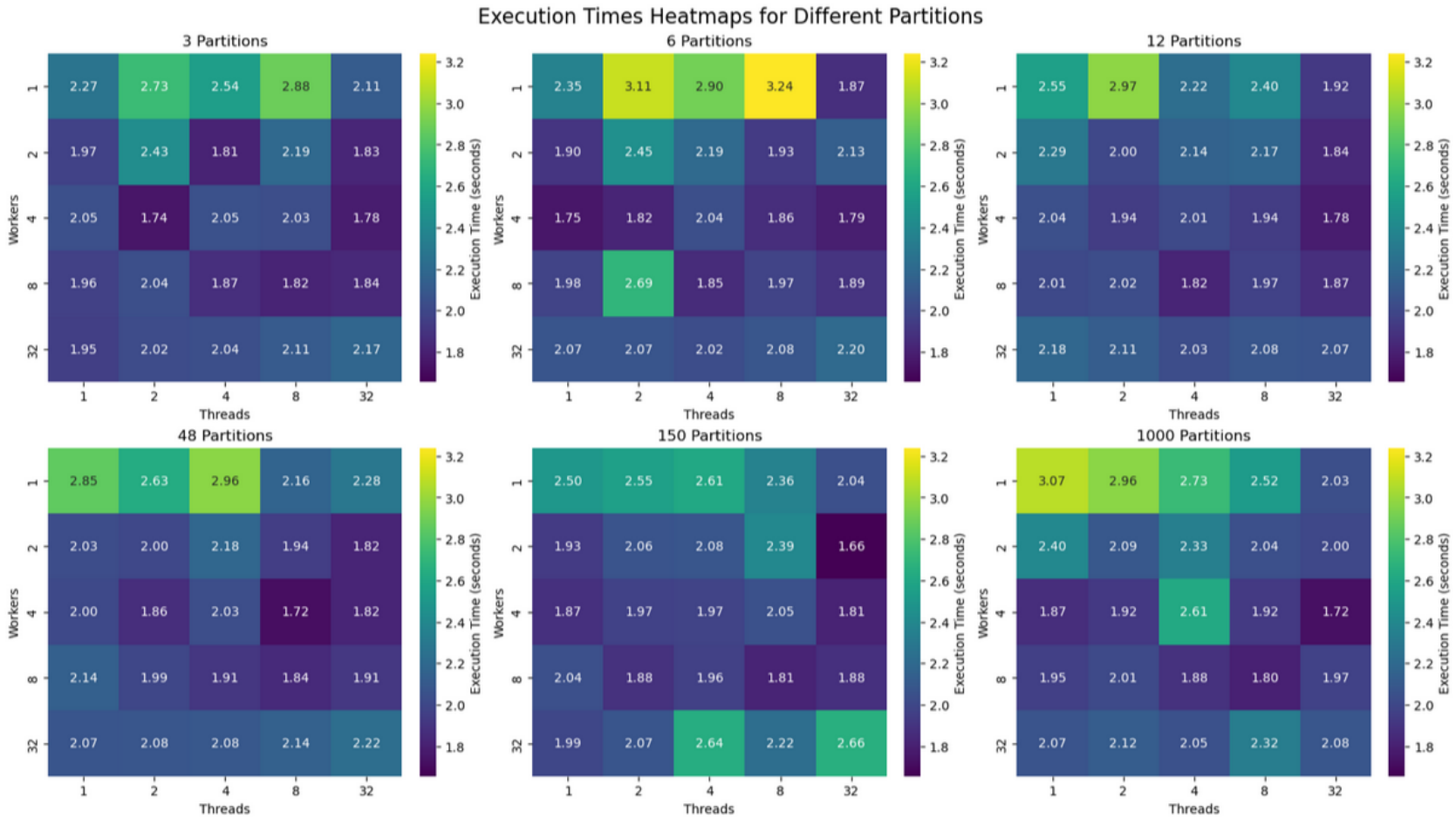
Fitting the local tracks



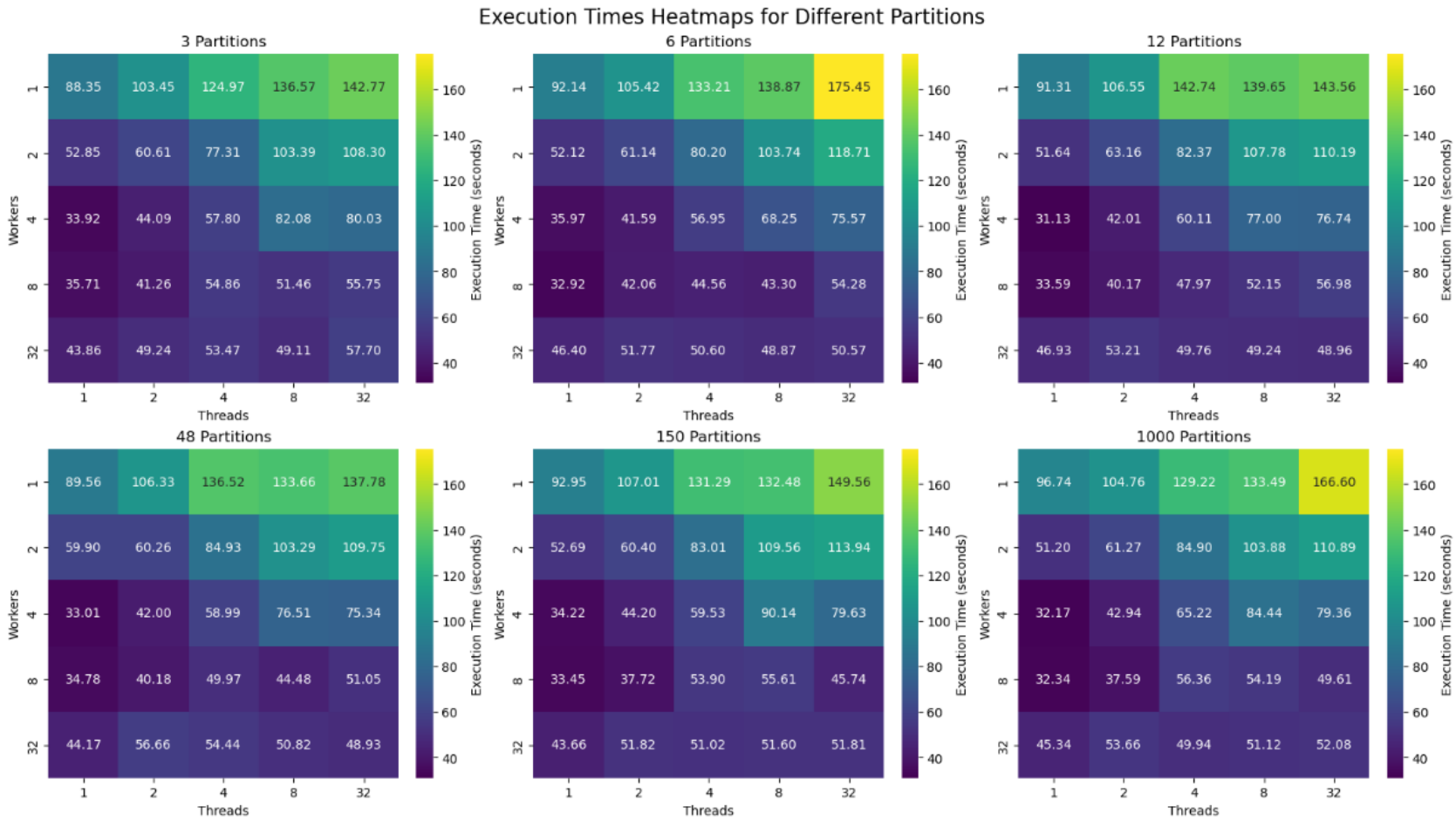
Fitting the global tracks



Calculate the angular differences



Wall time


Execution Times Heatmaps for Different Partitions
3 Partitions
88.35 103.45 124.97 136.57 142.77
52.85 60.61 77.31 103.39 108.30
33.92 44.09 57.80 82.08 80.03
35.71 41.26 54.86 51.46 55.75
43.86 49.24 53.47 49.11 57.70
6 Partitions
92.14 105.42 133.21 138.87 175.45
52.12 61.14 80.20 103.74 118.71
35.97 41.59 56.95 68.25 75.57
32.92 42.06 44.56 43.30 54.28
46.40 51.77 50.60 48.87 50.57
12 Partitions
91.31 106.55 142.74 139.65 143.56
51.64 63.16 82.37 107.78 110.19
31.13 42.01 60.11 77.00 76.74
33.59 40.17 47.97 52.15 56.98
46.93 53.21 49.76 49.24 48.96
48 Partitions
89.56 106.33 136.52 133.66 137.78
59.90 60.26 84.93 103.29 109.75
33.01 42.00 58.99 76.51 75.34
34.78 40.18 49.97 44.48 51.05
44.17 56.66 54.44 50.82 48.93
150 Partitions
92.95 107.01 131.29 132.48 149.56
52.69 60.40 83.01 109.56 113.94
34.22 44.20 59.53 90.14 79.63
33.45 37.72 53.90 55.61 45.74
43.66 51.82 51.02 51.60 51.81
1000 Partitions
96.74 104.76 129.22 133.49 166.60
51.20 61.27 84.90 103.88 110.89
32.17 42.94 65.22 84.44 79.36
32.34 37.59 56.36 54.19 49.61
45.34 53.66 49.94 51.12 52.08
Workers
Threads
Execution Time (seconds)

Overview with 4 workers per machine