In [33]:
```python
import numpy as np
import matplotlib.pyplot as plt
import time
import pandas as pd
from scipy import stats

# Import the hawkes package
import Hawkes as hk
```

Simulate one path of a Hawkes process

Functions for thinning and branching algorithm :

In [34]:
```python
# 1. Thinning Algorithm for Hawkes Process Simulation
def simulate_hawkes_thinning(lambda_0, kernel_func, horizon, max_intensity=None):
    """
    Simulate a Hawkes process using the thinning algorithm

    Parameters:
    -----------
    lambda_0 : float
        Baseline intensity
    kernel_func : function
        Function that takes a time difference and returns the kernel value
    horizon : float
        Time horizon for simulation
    max_intensity : float, optional
        Maximum intensity estimation. If None, will be estimated

    Returns:
    --------
    numpy.ndarray
        Array of event times
    """
    # Determine maximum intensity if not provided
    if max_intensity is None:
        # A rough estimation of the maximum intensity
        # This assumes kernel is decreasing and integrates to less than 1
        max_intensity = lambda_0 * 3

    events = []
    t = 0

    while t < horizon:
        # Generate next candidate event time using homogeneous Poisson process
        t = t + np.random.exponential(scale=1.0 / max_intensity)

        if t >= horizon:
            break

        # Calculate current intensity
        lambda_t = lambda_0
        for event_time in events:
            lambda_t += kernel_func(t - event_time)
            #if t - event_time > 0:  # Only consider past events


        # Accept with probability lambda_t / max_intensity (thinning)
        if np.random.random() <= lambda_t / max_intensity:
            events.append(t)

            # Update max_intensity if needed
            current_max = lambda_0 + sum(kernel_func(0) for _ in events)
            if current_max > max_intensity:
                max_intensity = current_max

    return np.array(events)
```

In [35]:
```python
# Helper functions for the branching algorithm
def estimate_kernel_integral(kernel_func, horizon, num_points=1000):
    """
```

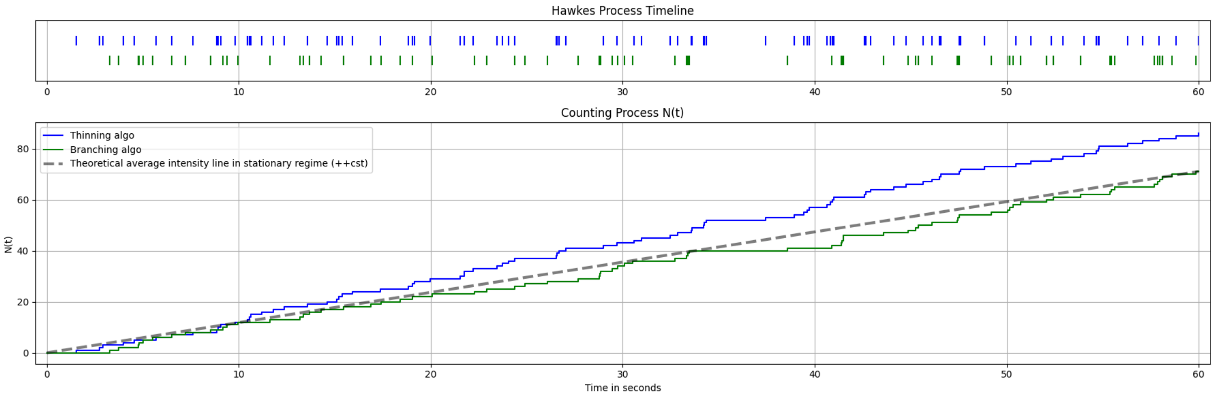
```python
    Estimate the integral of the kernel function over [0, horizon]
    """
    t = np.linspace(0, horizon, num_points)
    kernel_values = np.array([kernel_func(ti) for ti in t])
    return np.trapezoid(kernel_values, t)

def sample_from_kernel(kernel_func, horizon, num_points=1000):
    """
    Sample a time according to the normalized kernel function
    This is a simple rejection sampling approach
    """
    # Estimate the maximum of the kernel
    t = np.linspace(0, horizon, num_points)
    kernel_values = np.array([kernel_func(ti) for ti in t])
    max_kernel = np.max(kernel_values)

    while True:
        # Propose a time
        proposed_time = np.random.uniform(0, horizon)
        kernel_value = kernel_func(proposed_time)

        # Accept with probability kernel_value / max_kernel
        if np.random.random() <= kernel_value / max_kernel:
            return proposed_time

# 2. Branching Algorithm for Hawkes Process Simulation
def simulate_hawkes_branching(lambda_0, kernel_func, horizon):
    """
    Simulate a Hawkes process using the branching algorithm

    Parameters:
    -----------
    lambda_0 : float
        Baseline intensity
    kernel_func : function
        Function that takes a time difference and returns the kernel value
    horizon : float
        Time horizon for simulation

    Returns:
    --------
    numpy.ndarray
        Array of event times
    """
    # First generate immigrant events from homogeneous Poisson process
    immigrant_times = []
    t = 0
    while t < horizon:
        t += np.random.exponential(1.0 / lambda_0)
        if t < horizon:
            immigrant_times.append(t)

    # For each immigrant, generate its offspring
    all_events = immigrant_times.copy()
    generation = 0
    current_gen_events = immigrant_times.copy()

    while current_gen_events:
        generation += 1
        next_gen_events = []

        for parent_time in current_gen_events:
            # Determine the branching factor (number of direct offspring)
            # This depends on the kernel
            kernel_integral = estimate_kernel_integral(kernel_func, horizon)
            num_offspring = np.random.poisson(kernel_integral)

            for _ in range(num_offspring):
                # Sample offspring time
                while True:
                    # Sample a time according to the normalized kernel
                    # This is an approximation - for specific kernels, more efficient methods exist
                    offspring_time = sample_from_kernel(kernel_func, horizon)
                    event_time = parent_time + offspring_time
```

```
                    if event_time < horizon:
                        next_gen_events.append(event_time)
                        all_events.append(event_time)
                        break
                    else:
                        break  # Skip if beyond horizon

        current_gen_events = next_gen_events

    return np.sort(np.array(all_events))
```

Tests :

In [36]:
```
#parameters

# Define exponential kernel function
# kernel functions
def exponential_kernel(alpha, beta):
    """
    Returns an exponential kernel function: alpha * exp(-beta * t)
    alpha: scaling factor
    beta: decay rate
    """
    return lambda t: alpha * beta * np.exp(-beta * t)

alpha_test = 0.5 / 7
beta_test = 7
lambda0_test = 1.1
horizon_test = 60

# Average intensity in stationnary regime
av_intensity_SR = lambda0_test / (1 - alpha_test)
```

In [37]:
```
# Run simulation
events_thinning = simulate_hawkes_thinning(lambda_0=lambda0_test, kernel_func=exponential_kernel(alpha_t
events_branching = simulate_hawkes_branching(lambda_0=lambda0_test, kernel_func=exponential_kernel(alpha

# Create figure with two subplots with different heights (1:3 ratio)
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(18, 6), gridspec_kw={'height_ratios': [1, 4]})

# Create timeline
ax1.hlines(y=0, xmin=0, xmax=horizon_test, color='black', linewidth=0.5)
ax1.scatter(events_thinning, np.ones_like(events_thinning) * 2/3, color='blue', marker='|', s=100)
ax1.scatter(events_branching, np.ones_like(events_branching) * 1/3, color='green', marker='|', s=100)
ax1.set_ylim(0, 1)
ax1.set_xlim(-horizon_test*0.01, horizon_test*1.01)
ax1.set_title('Hawkes Process Timeline')
ax1.set_yticks([])
ax1.grid(True, axis='x')

# Plot counting process (keep the same)
times_thinning = np.concatenate([[0], events_thinning, [horizon_test]])
counts_thinning = np.array( [0] + [i+1 for i in range(len(events_thinning))] + [len(events_thinning)] )
ax2.step(times_thinning, counts_thinning, where='post', color="blue", label="Thinning algo")
times_branching = np.concatenate([[0], events_branching, [horizon_test]])
counts_branching = np.array( [0] + [i+1 for i in range(len(events_branching))] + [len(events_branching)]
ax2.step(times_branching, counts_branching, where='post', color="green", label="Branching algo")

ax2.plot([0, horizon_test], [0, av_intensity_SR * horizon_test], color="black", label = "Theoretical ave
ax2.set_xlim(-horizon_test*0.01, horizon_test*1.01)
ax2.set_xlabel('Time in seconds')
ax2.set_ylabel('N(t)')
ax2.set_title('Counting Process N(t)')
ax2.legend()
ax2.grid(True)

plt.tight_layout()
plt.show()

print(f"Number of events thinning: {len(events_thinning)}")
print(f"Number of events branching: {len(events_branching)}")
```



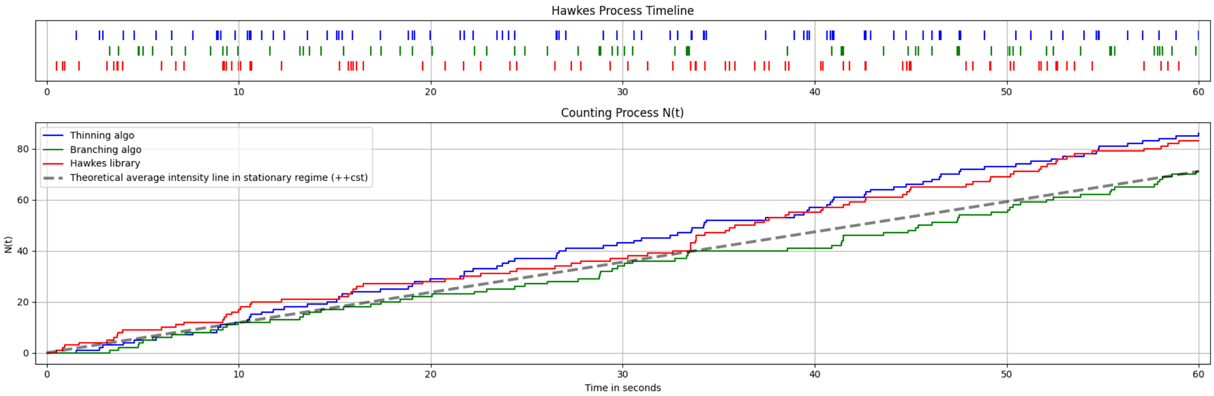

```
Number of events thinning: 86
Number of events branching: 71
```

Properties of Hawkes MLE estimates

Tests library hawkes

In [38]:

```
# Run simulation with hawkes library
para = {'mu': lambda0_test, 'alpha': alpha_test, 'beta': beta_test}
hk_simulator_for_comparaison = hk.simulator()
hk_simulator_for_comparaison.set_kernel('exp')
hk_simulator_for_comparaison.set_baseline('const')
hk_simulator_for_comparaison.set_parameter(para)
# Simulate process
itv = [0, horizon_test]
events_hk = hk_simulator_for_comparaison.simulate(itv)

# Create figure with two subplots with different heights (1:3 ratio)
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(18, 6), gridspec_kw={'height_ratios': [1, 4]})

# Create timeline
ax1.hlines(y=0, xmin=0, xmax=horizon_test, color='black', linewidth=0.5)
ax1.scatter(events_thinning, np.ones_like(events_thinning) * 3/4, color='blue', marker='|', s=100)
ax1.scatter(events_branching, np.ones_like(events_branching) * 2/4, color='green', marker='|', s=100)
ax1.scatter(events_hk, np.ones_like(events_hk) * 1/4, color='red', marker='|', s=100)
ax1.set_ylim(0, 1)
ax1.set_xlim(-horizon_test*0.01, horizon_test*1.01)
ax1.set_title('Hawkes Process Timeline')
ax1.set_yticks([])
ax1.grid(True, axis='x')

# Plot counting process (keep the same)
times_thinning = np.concatenate([[0], events_thinning, [horizon_test]])
counts_thinning = np.array( [0] + [i+1 for i in range(len(events_thinning))] + [len(events_thinning)] )
ax2.step(times_thinning, counts_thinning, where='post', color="blue", label="Thinning algo")

times_branching = np.concatenate([[0], events_branching, [horizon_test]])
counts_branching = np.array( [0] + [i+1 for i in range(len(events_branching))] + [len(events_branching)]
ax2.step(times_branching, counts_branching, where='post', color="green", label="Branching algo")

times_hk = np.concatenate([[0], events_hk, [horizon_test]])
counts_hk = np.array( [0] + [i+1 for i in range(len(events_hk))] + [len(events_hk)] )
ax2.step(times_hk, counts_hk, where='post', color="red", label="Hawkes library")

ax2.plot([0, horizon_test], [0, av_intensity_SR * horizon_test], color="black", label = "Theoretical ave
ax2.set_xlim(-horizon_test*0.01, horizon_test*1.01)
ax2.set_xlabel('Time in seconds')
ax2.set_ylabel('N(t)')
ax2.set_title('Counting Process N(t)')
ax2.legend()
ax2.grid(True)

plt.tight_layout()
plt.show()

print(f"Number of events (thinning function): {len(events_thinning)}")
print(f"Number of events (branching function): {len(events_branching)}")
print(f"Number of events (hawkes library): {len(events_hk)}")
```




```
Number of events (thinning function): 86
Number of events (branching function): 71
Number of events (hawkes library): 83
```

Fitting library hawkes model to data generated by thinning and branching methods

In [39]:
```
#New parameters :
horizon_test2 = 500
itv2 = [0, horizon_test2]

# Run simulation
events_thinning2 = simulate_hawkes_thinning(lambda_0=lambda0_test, kernel_func=exponential_kernel(alpha_

model_fitted_thinning = hk.estimator()
model_fitted_thinning.set_kernel('exp')
model_fitted_thinning.set_baseline('const')
model_fitted_thinning.fit(events_thinning2, itv2)  # Fixed: using itv2 instead of itv
params_fitted_thinning = model_fitted_thinning.para


print("Hawkes Library fitted on thinning algorithm")
print(f"Expected mu: {lambda0_test} VS {float(params_fitted_thinning['mu'])} : mu calibrated")
print(f"Expected alpha: {alpha_test} VS {float(params_fitted_thinning['alpha'])} : alpha calibrated")
print(f"Expected beta: {beta_test} VS {float(params_fitted_thinning['beta'])} : beta calibrated")
```

```
Hawkes Library fitted on thinning algorithm
Expected mu: 1.1 VS 1.0328789766986386 : mu calibrated
Expected alpha: 0.07142857142857142 VS 0.16704594180849058 : alpha calibrated
Expected beta: 7 VS 3.7950881603405096 : beta calibrated
```

In [40]:
```
# Run simulation
horizon_test2 = 1500
itv2 = [0, horizon_test2]
events_branching2 = simulate_hawkes_branching(lambda_0=lambda0_test, kernel_func=exponential_kernel(alph

model_fitted_branching = hk.estimator()
model_fitted_branching.set_kernel('exp')
model_fitted_branching.set_baseline('const')
model_fitted_branching.fit(events_branching2, itv2)  # Using itv2 instead of itv
params_fitted_branching = model_fitted_branching.para

print("Hawkes Library fitted on branching algorithm")
print(f"Expected mu: {lambda0_test} VS {float(params_fitted_branching['mu'])} : mu calibrated")
print(f"Expected alpha: {alpha_test} VS {float(params_fitted_branching['alpha'])} : alpha calibrated")
print(f"Expected beta: {beta_test} VS {float(params_fitted_branching['beta'])} : beta calibrated")
```

```
Hawkes Library fitted on branching algorithm
Expected mu: 1.1 VS 1.1020948615419157 : mu calibrated
Expected alpha: 0.07142857142857142 VS 0.38798327469919536 : alpha calibrated
Expected beta: 7 VS 6.637334555895608 : beta calibrated
```

Lorsqu'on fait tendre l'hozizon vers l'infini, les paramètres calibrés tendent vers ce qui est attendu

In [41]:
```
# Function to analyze MLE properties
def compare_library_to_algo(lambda0_test, alpha_test, beta_test, simulate_hawkes_function, num_simulatio

    mu_true, alpha_true, beta_true = lambda0_test, alpha_test, beta_test

    # Create parameter dictionary as expected by the package
    para = {'mu': mu_true, 'alpha': alpha_true, 'beta': beta_true}
```

```
    mu_estimates = []
    alpha_estimates = []
    beta_estimates = []

    # Use the thinning simulator directly from Hawkes package
    for i in range(num_simulations):
        itv = [0, horizon]
        events = simulate_hawkes_function( lambda_0=mu_true, kernel_func=exponential_kernel(alpha_true,

        # Fit the Hawkes process
        model = hk.estimator()
        model.set_kernel('exp')
        model.set_baseline('const')
        model.fit([events], itv)

        # Extract estimated parameters
        params = model.para

        mu_est = params['mu']
        alpha_est = params['alpha']
        beta_est = params['beta']

        mu_estimates.append(mu_est)
        alpha_estimates.append(alpha_est)
        beta_estimates.append(beta_est)

    # Compute statistics
    # Convert the statistics to a DataFrame
    df_results = pd.DataFrame({
        'mu': [mu_true, np.mean(mu_estimates), np.std(mu_estimates),
               np.mean(mu_estimates) - mu_true, np.mean((np.array(mu_estimates) - mu_true) ** 2)],
        'alpha': [alpha_true, np.mean(alpha_estimates), np.std(alpha_estimates),
                  np.mean(alpha_estimates) - alpha_true, np.mean((np.array(alpha_estimates) - alpha_true)
        'beta': [beta_true, np.mean(beta_estimates), np.std(beta_estimates),
                 np.mean(beta_estimates) - beta_true, np.mean((np.array(beta_estimates) - beta_true) ** 2
    }, index=['true', 'mean', 'std', 'bias', 'mse'])

    return df_results
```

In [42]:
```
# tests on thinning algo

results_thinning = compare_library_to_algo(lambda0_test, alpha_test, beta_test, simulate_hawkes_thinning
print(results_thinning)
```

```
            mu     alpha       beta
true  1.100000  0.071429   7.000000
mean  1.033333  0.112574   5.201010
std   0.081990  0.051907   4.719797
bias -0.066667  0.041145  -1.798990
mse   0.011167  0.004387  25.512846
```

In [43]:
```
# tests on branching algo

results_branching = compare_library_to_algo(lambda0_test, alpha_test, beta_test, simulate_hawkes_branchi
print(results_branching)
```

```
            mu     alpha       beta
true  1.100000  0.071429   7.000000
mean  1.075118  0.109469   9.196943
std   0.092198  0.050758   7.564006
bias -0.024882  0.038040   2.196943
mse   0.009120  0.004023  62.040751
```

Computational cost of Hawkes simulators

In [44]:
```
horizon_values = [20, 40, 80, 100, 150, 200, 300, 500]
execution_times = {'thinning': [], 'branching': [], 'Library hawkes' : []}

para = {'mu': lambda0_test, 'alpha': alpha_test, 'beta': beta_test}
hk_simulator = hk.simulator()
hk_simulator.set_kernel('exp')
hk_simulator.set_baseline('const')
hk_simulator.set_parameter(para)
```

```python
for horizon in horizon_values:
    # Thinning algorithm
    start_time = time.time()
    _ = simulate_hawkes_thinning(lambda_0=lambda0_test,
                                kernel_func=exponential_kernel(alpha_test, beta_test),
                                horizon=horizon)
    execution_times['thinning'].append(time.time() - start_time)

    # Branching algorithm
    start_time = time.time()
    _ = simulate_hawkes_branching(lambda_0=lambda0_test,
                                 kernel_func=exponential_kernel(alpha_test, beta_test),
                                 horizon=horizon)
    execution_times['branching'].append(time.time() - start_time)

    # Library hawkes
    itv = [0, horizon]
    start_time = time.time()
    # Simulate process
    _ = hk_simulator.simulate(itv)
    execution_times['Library hawkes'].append(time.time() - start_time)

# Create DataFrame for results
df_times = pd.DataFrame({
    'horizon': horizon_values,
    'thinning_time': execution_times['thinning'],
    'branching_time': execution_times['branching'],
    'Library_time': execution_times['Library hawkes'],
})

print("Execution times (seconds):")
print(df_times)
```

```
Execution times (seconds):
   horizon  thinning_time  branching_time  Library_time
0       20       0.003504        0.038398      0.000000
1       40       0.014863        0.077568      0.001004
2       80       0.179169        0.133995      0.001005
3      100       0.287549        0.172974      0.002504
4      150       1.016500        0.350556      0.003570
5      200       2.507216        0.525138      0.004014
6      300       6.371891        1.097911      0.007566
7      500      39.111996        2.305502      0.011716
```

In [45]:
```python
# Plot execution times
plt.figure(figsize=(17, 5))
plt.plot(df_times['horizon'], df_times['thinning_time'], 'b-o', label='Thinning')
plt.plot(df_times['horizon'], df_times['branching_time'], 'g-o', label='Branching')
plt.plot(df_times[df_times['Library_time'] > 0]['horizon'], df_times[df_times['Library_time'] > 0]['Libr
plt.xlabel('Horizon')
plt.ylabel('Execution Time (seconds)')
plt.yscale("log")
plt.title('Execution Time vs Horizon')
plt.legend()
plt.grid(True)
plt.show()
```

La complexité des algorithmes branching et de la library semble être en log(horizon). On s'attend à avoir une complexité en horizon**(x) pour l'algorithme thinning

In [46]:

```python
def compare_computational_cost(horizons=[ i for i in range(50,300, 20)]):
    """
    Compare the computational cost of different Hawkes simulation methods
    and analyze their complexity with respect to the horizon
    """
    # Parameters for the process
    lambda_0 = 0.5
    alpha = 0.8
    beta = 1.0
    kernel_func = exponential_kernel(alpha, beta)

    # Store times for each method
    thinning_times = []
    branching_times = []
    library_times = []
    num_events = []

    # Parameter dictionary for the Hawkes library
    para = {'mu': lambda_0, 'alpha': alpha, 'beta': beta}

    # Setup library simulator once
    library_simulator = hk.simulator()
    library_simulator.set_kernel('exp')
    library_simulator.set_baseline('const')
    library_simulator.set_parameter(para)

    # Run simulations for various horizons
    for horizon in horizons:
        print(f"Testing with horizon = {horizon}")

        # Measure time for thinning algorithm
        start_time = time.time()
        events_thinning = simulate_hawkes_thinning(lambda_0, kernel_func, horizon)
        thinning_time = time.time() - start_time
        thinning_times.append(thinning_time)

        # Measure time for branching algorithm
        start_time = time.time()
        events_branching = simulate_hawkes_branching(lambda_0, kernel_func, horizon)
        branching_time = time.time() - start_time
        branching_times.append(branching_time)

        # Measure time for Hawkes library
        start_time = time.time()
        events_library = library_simulator.simulate([0, horizon])
        library_time = time.time() - start_time
        library_times.append(library_time)

        # Store the average number of events to relate to complexity
        num_events.append(len(events_library))

    # Fit polynomial to understand complexity
    # log(time) = log(a) + b*log(horizon) => time ~ horizon^b
    log_horizons = np.log(horizons)

    # For thinning
    log_thinning = np.log(thinning_times)
    thinning_poly = np.polyfit(log_horizons, log_thinning, 1)
    thinning_exponent = thinning_poly[0]

    # For branching
    log_branching = np.log(branching_times)
    branching_poly = np.polyfit(log_horizons, log_branching, 1)
    branching_exponent = branching_poly[0]

    # For library
    log_library = np.log(library_times)
    library_poly = np.polyfit(log_horizons, log_library, 1)
    library_exponent = library_poly[0]

    # Store results
    results = {
        'horizons': horizons,
        'thinning_times': thinning_times,
```

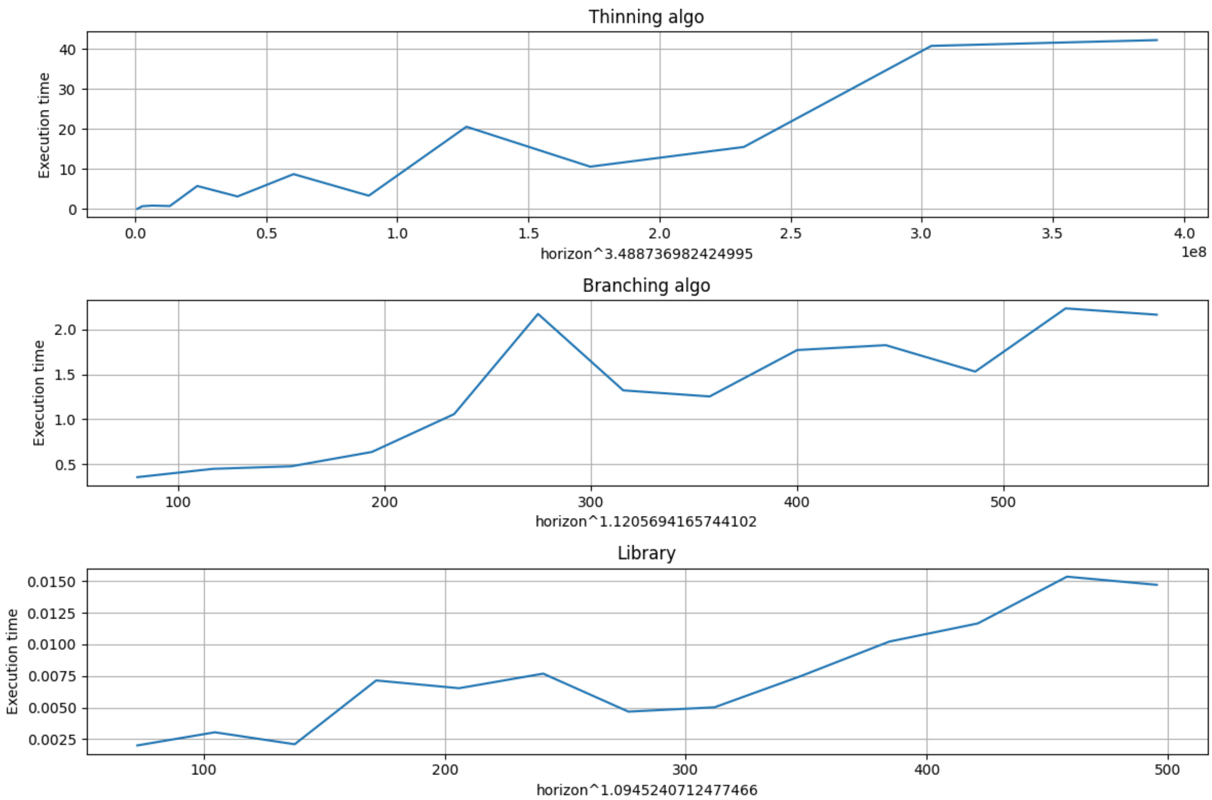
```python
        'branching_times': branching_times,
        'library_times': library_times,
        'num_events': num_events,
        'complexity': {
            'thinning_exponent': thinning_exponent,
            'branching_exponent': branching_exponent,
            'library_exponent': library_exponent
        }
    }

    return results
```

In [47]:
```python
results = compare_computational_cost()
complexity = results['complexity']
complexity
```

```
Testing with horizon = 50
Testing with horizon = 70
Testing with horizon = 90
Testing with horizon = 110
Testing with horizon = 130
Testing with horizon = 150
Testing with horizon = 170
Testing with horizon = 190
Testing with horizon = 210
Testing with horizon = 230
Testing with horizon = 250
Testing with horizon = 270
Testing with horizon = 290
```

Out[47]:
```
{'thinning_exponent': np.float64(3.488736982424995),
 'branching_exponent': np.float64(1.1205694165744102),
 'library_exponent': np.float64(1.0945240712477466)}
```

In [48]:
```python
fig, (ax1, ax2, ax3) = plt.subplots(3, 1, figsize=(12,8), gridspec_kw={'height_ratios': [1, 1, 1]})

# Plot thinning complexity
ax1.plot(results['horizons']**(results['complexity']['thinning_exponent']), results['thinning_times'])
ax1.set_xlabel(f"horizon^{results['complexity']['thinning_exponent']}")
ax1.set_ylabel('Execution time')
ax1.set_title('Thinning algo')
ax1.grid(True)

# Plot Branching complexity
ax2.plot(results['horizons']**(results['complexity']['branching_exponent']), results['branching_times'])
ax2.set_xlabel(f"horizon^{results['complexity']['branching_exponent']}")
ax2.set_ylabel('Execution time')
ax2.set_title('Branching algo')
ax2.grid(True)

# Plot Library complexity
ax3.plot(results['horizons']**(results['complexity']['library_exponent']), results['library_times'])
ax3.set_xlabel(f"horizon^{results['complexity']['library_exponent']}")
ax3.set_ylabel('Execution time')
ax3.set_title('Library')
ax3.grid(True)

plt.tight_layout()
plt.show()
```



The complexity of thinning algorithm is horizon^3

The complexity of Branching algorithm is horizon^1.1

The complexity of library algorithm is log(horizon)

A Hawkes process for trades

In [49]:
```python
# Function to analyze trade data using Hawkes processes
def analyze_trades_with_hawkes(trade_data, kernel='exp'):
    """Analyze if trade data follows a Hawkes process"""

    # Extract the timestamps and convert to seconds since start
    if isinstance(trade_data.iloc[0]['ets'], str):
        timestamps = pd.to_datetime(trade_data['ets'])
    else:
        timestamps = trade_data['ets']

    start_time = timestamps.min()
    times_in_seconds = [(t - start_time).total_seconds() for t in timestamps]
    times_in_seconds = np.array(times_in_seconds)

    # Define observation interval
    itv = [times_in_seconds.min(), times_in_seconds.max()]

    # Fit Hawkes process
    model = hk.estimator()
    model.set_kernel(kernel)
    model.set_baseline('const')

    # Fit the model using just the event times and interval
    model.fit([times_in_seconds], itv)

    # Get the parameters directly from model.para
    params = model.para

    # Get log-likelihood directly using model.L
    hawkes_loglikelihood = model.L

    # Compare with homogeneous Poisson process
    T = itv[1] - itv[0]
    n = len(times_in_seconds)
    lambda_mle = n / T
```

```python
    poisson_loglikelihood = n * np.log(lambda_mle) - lambda_mle * T

    # Calculate AIC
    if kernel == 'exp':
        hawkes_aic = -2 * hawkes_loglikelihood + 2 * 3  # mu, alpha, beta
    else:
        hawkes_aic = -2 * hawkes_loglikelihood + 2 * 3  # simplification

    poisson_aic = -2 * poisson_loglikelihood + 2 * 1

    # Use the built-in Kolmogorov-Smirnov test for Hawkes processes
    ks_result = model.plot_KS()  # This might return the KS statistic
    ks_stat = ks_result if ks_result is not None else None

    # For Poisson test - simple inter-arrival test
    inter_arrivals = np.diff(times_in_seconds)
    exp_stat, exp_pvalue = stats.kstest(inter_arrivals, 'expon', args=(0, 1 / lambda_mle))

    # Get branching ratio directly
    branching_ratio = model.br if hasattr(model, 'br') else 0

    # Return results in simplified format
    hawkes_params = {'mu': params.get('mu', 0)}
    if kernel == 'exp':
        hawkes_params.update({
            'alpha': params.get('alpha', 0),
            'beta': params.get('beta', 1),
            'branching_ratio': branching_ratio
        })

    # Calculate likelihood ratio test
    lr_statistic = 2 * (hawkes_loglikelihood - poisson_loglikelihood)
    lr_pvalue = 1 - stats.chi2.cdf(lr_statistic, df=2)  # df = 3-1 = 2 (difference in parameters)

    return {
        'hawkes_params': hawkes_params,
        'loglikelihood': {
            'hawkes': hawkes_loglikelihood,
            'poisson': poisson_loglikelihood,
        },
        'aic': {
            'hawkes': hawkes_aic,
            'poisson': poisson_aic,
            'better_model': 'Hawkes' if hawkes_aic < poisson_aic else 'Poisson'
        },
        'ks_test': {
            'poisson': {'statistic': exp_stat, 'pvalue': exp_pvalue}
        },
        'likelihood_ratio_test': {
            'statistic': lr_statistic,
            'pvalue': lr_pvalue,
            'reject_poisson': lr_pvalue < 0.05
        }
    }
```

In [50]:

```python
# Function to perform complete Hawkes process analysis

def analyze_mle_properties(true_params, num_simulations=100, horizon=100):
    """
    Analyze the statistical properties of the MLE estimator
    for Hawkes processes with an exponential kernel
    """
    mu_true, alpha_true, beta_true = true_params

    # Create parameter dictionary as expected by the package
    para = {'mu': mu_true, 'alpha': alpha_true, 'beta': beta_true}

    mu_estimates = []
    alpha_estimates = []
    beta_estimates = []

    # Use the thinning simulator directly from Hawkes package
    for i in range(num_simulations):
        # Create simulator with proper parameters
        simulator = hk.simulator()
```

```
        simulator.set_kernel('exp')
        simulator.set_baseline('const')
        simulator.set_parameter(para)

        # Simulate process
        itv = [0, horizon]
        events = simulator.simulate(itv)

        # Fit the Hawkes process
        model = hk.estimator()
        model.set_kernel('exp')
        model.set_baseline('const')
        model.fit([events], itv)

        # Extract estimated parameters
        params = model.para

        mu_est = params['mu']
        alpha_est = params['alpha']
        beta_est = params['beta']

        mu_estimates.append(mu_est)
        alpha_estimates.append(alpha_est)
        beta_estimates.append(beta_est)

    # Compute statistics
    results = {
        'mu': {
            'true': mu_true,
            'mean': np.mean(mu_estimates),
            'std': np.std(mu_estimates),
            'bias': np.mean(mu_estimates) - mu_true,
            'estimates': mu_estimates,
            'mse': np.mean((np.array(mu_estimates) - mu_true) ** 2)
        },
        'alpha': {
            'true': alpha_true,
            'mean': np.mean(alpha_estimates),
            'std': np.std(alpha_estimates),
            'bias': np.mean(alpha_estimates) - alpha_true,
            'estimates': alpha_estimates,
            'mse': np.mean((np.array(alpha_estimates) - alpha_true) ** 2)
        },
        'beta': {
            'true': beta_true,
            'mean': np.mean(beta_estimates),
            'std': np.std(beta_estimates),
            'bias': np.mean(beta_estimates) - beta_true,
            'estimates': beta_estimates,
            'mse': np.mean((np.array(beta_estimates) - beta_true) ** 2)
        }
    }

    return results

def run_hawkes_analysis(data_sg):
    """
    Run the complete Hawkes process analysis on the provided data

    Parameters:
    -----------
    data_sg : dict
        Dictionary containing SG trade data

    Returns:
    --------
    dict
        Dictionary with all analysis results
    """
    results = {}

    # Analyze trade data
    print("Analyzing trade data with Hawkes process...")
    hawkes_trade_results = {}
```

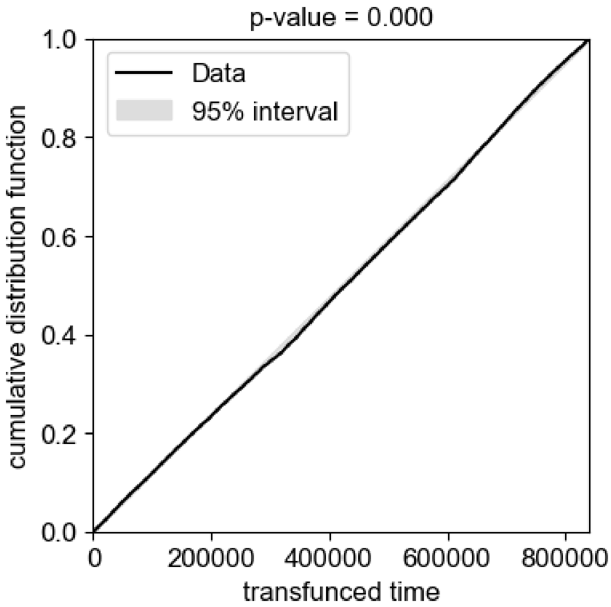
```python
    # Sample one day of data for demonstration
    sample_date = list(data_sg.keys())[0]
    sample_data = data_sg[sample_date]

    # Analyze the full day
    hawkes_trade_results['full_day'] = analyze_trades_with_hawkes(sample_data)

    # Analyze different time periods if data is large
    if len(sample_data) > 1000:
        # Morning session
        morning = sample_data[sample_data['ets'].dt.hour < 12]
        hawkes_trade_results['morning'] = analyze_trades_with_hawkes(morning)

        # Afternoon session
        afternoon = sample_data[sample_data['ets'].dt.hour >= 12]
        hawkes_trade_results['afternoon'] = analyze_trades_with_hawkes(afternoon)

    results['trade_analysis'] = hawkes_trade_results

    return results
```

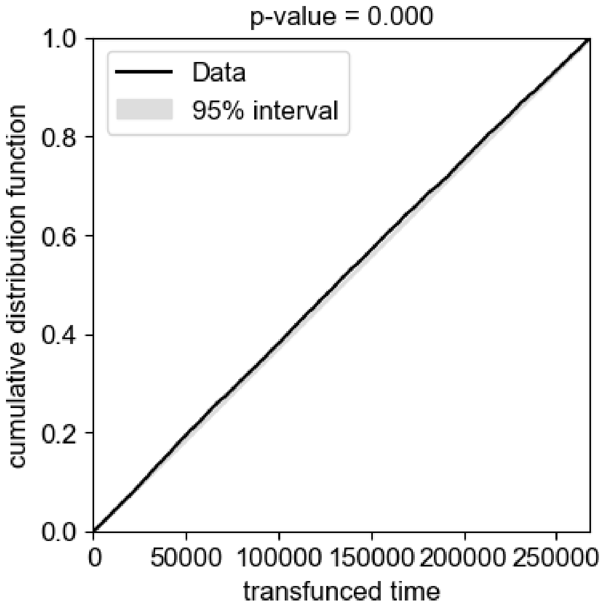
In [51]:
```python
def plot_results(results):

    # Trade Data Analysis
    plt.figure(figsize=(15, 10))
    for i, period in enumerate(['full_day', 'morning', 'afternoon']):
        if period in results['trade_analysis']:
            plt.subplot(2, 2, i + 1)
            trade_analysis = results['trade_analysis'][period]
            models = ['Hawkes', 'Poisson']
            aic_values = [trade_analysis['aic']['hawkes'], trade_analysis['aic']['poisson']]
            plt.bar(models, aic_values)
            plt.ylabel('AIC (lower is better)')
            plt.title(f'Model Comparison for Trade Data ({period})')

    # Add a subplot for parameters if available
    if 'full_day' in results['trade_analysis']:
        plt.subplot(2, 2, 4)
        params = results['trade_analysis']['full_day']['hawkes_params']
        plt.bar(['mu', 'alpha', 'beta'], [params['mu'], params['alpha'], params['beta']])
        plt.ylabel('Parameter Value')
        plt.title('Estimated Hawkes Parameters')

    plt.tight_layout()
    plt.show()
```

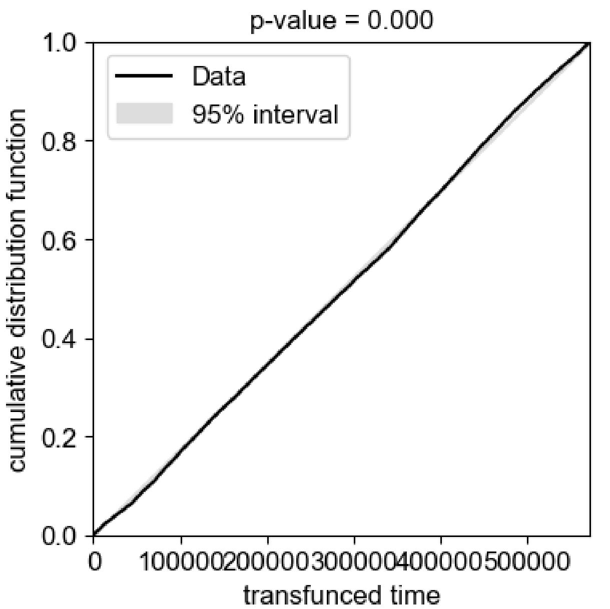
In [52]:
```python
# Try to load the real data
file_list = [
    'Data/SG/SG_20170117.csv.gz', 'Data/SG/SG_20170118.csv.gz',
    'Data/SG/SG_20170119.csv.gz', 'Data/SG/SG_20170120.csv.gz'
]

Data_sg = {}
for i, file in enumerate(file_list):
    Data_sg[file[8:-7]] = pd.read_csv(file, compression='gzip').drop(columns=['Unnamed: 0'])
    Data_sg[file[8:-7]]['ets'] = pd.to_datetime(Data_sg[file[8:-7]]['ets'], format='%Y%m%d:%H:%M:%S.%f')
```

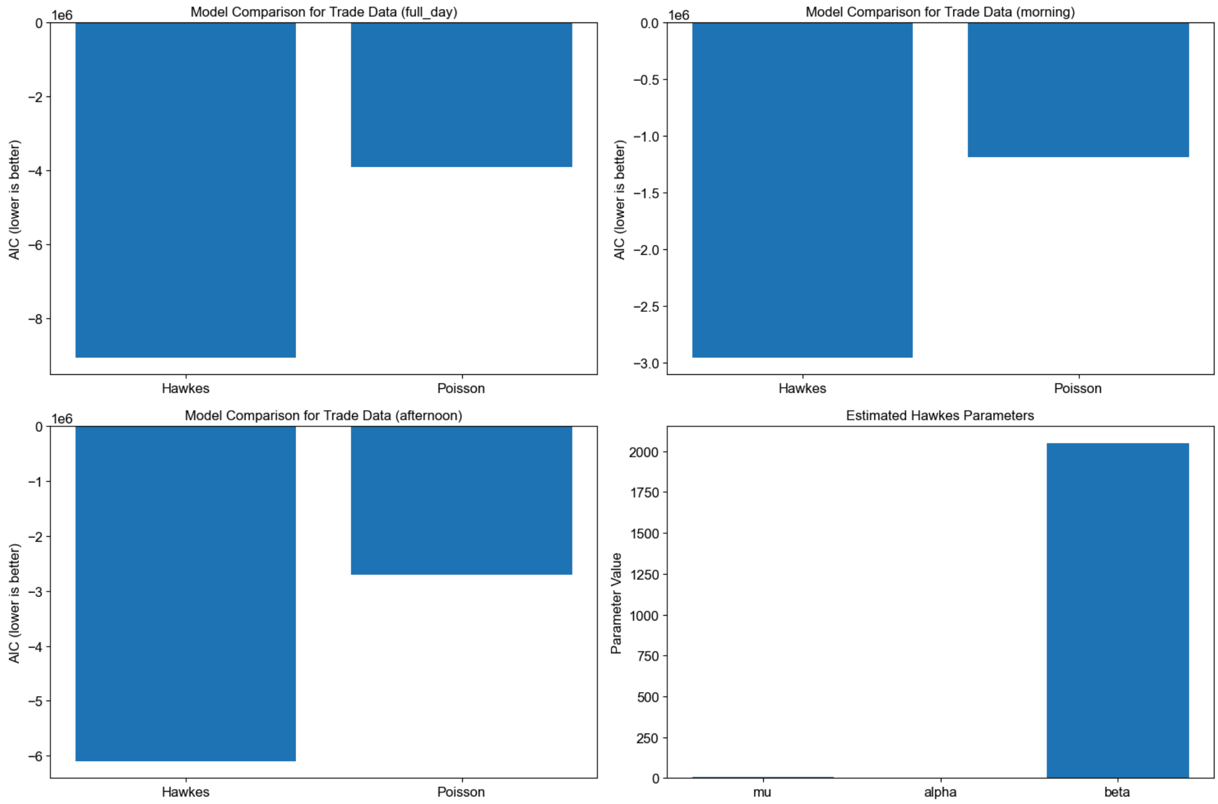
In [53]:
```python
# Create a dictionary containing only the data for 'SG_20170117'
specific_day_data = {"SG_20170117": Data_sg["SG_20170117"]}

# Run the complete analysis
results = run_hawkes_analysis(specific_day_data)

# Plot results
plot_results(results)
```

```
Analyzing trade data with Hawkes process...
```


p-value = 0.000
Data
95% interval
cumulative distribution function
transfunced time
0
200000
400000
600000
800000
0.0
0.2
0.4
0.6
0.8
1.0


p-value = 0.000
Data
95% interval
cumulative distribution function
transfunced time
0
50000
100000
150000
200000
250000
0.0
0.2
0.4
0.6
0.8
1.0


p-value = 0.000
Data
95% interval
cumulative distribution function
transfunced time
0
100000
200000
300000
400000
500000
0.0
0.2
0.4
0.6
0.8
1.0



In [54]:

```python
# Print summary of findings

print("\nTrade Data Analysis:")
trade_analysis = results['trade_analysis']['full_day']
print(f"  - Model Selection (AIC): {trade_analysis['aic']['better_model']} model is better")
print(f"  - Likelihood Ratio Test: p-value = {trade_analysis['likelihood_ratio_test']['pvalue']:.4f}")
if trade_analysis['likelihood_ratio_test']['reject_poisson']:
    print("    (Poisson model is rejected in favor of Hawkes)")
else:
    print("    (Cannot reject Poisson model)")

print(f"  - Estimated Hawkes Parameters: μ = {trade_analysis['hawkes_params']['mu']:.4f}, "
        f"α = {trade_analysis['hawkes_params']['alpha']:.4f}, "
        f"β = {trade_analysis['hawkes_params']['beta']:.4f}")
print(f"  - Branching Ratio: {trade_analysis['hawkes_params']['branching_ratio']:.4f}")
```

```
Trade Data Analysis:
  - Model Selection (AIC): Hawkes model is better
  - Likelihood Ratio Test: p-value = 0.0000
    (Poisson model is rejected in favor of Hawkes)
  - Estimated Hawkes Parameters: μ = 7.3244, α = 0.7341, β = 2050.2789
  - Branching Ratio: 0.7341
```