Worksheet 1

Group 22 Eaton Heidegger and Shannon Schröder

In [1]:
```
import numpy as np
import os
import time
import matplotlib.pyplot as plt
import scipy
import matplotlib
import sympy
import numba
from numba import njit, prange , set_num_threads, get_num_threads
from typing import Callable
```

Exercise 1 Computing Gravitational Acceleration using a Naïve Algorithm

In [2]:
```
def get_acceleration(X: np.ndarray) -> np.ndarray:
    N = X.shape[0]
    a = np.zeros_like(X)  # Initialize acceleration array

    for i in range(N):
        for j in range(N):
            if i != j:
                r_ij = X[j] - X[i]  # Vector from i to j
                distance = np.linalg.norm(r_ij)
                if distance > 0:
                    a[i] += r_ij / distance**3  # Gravitational acceleration for

    return a
```

We implement a function `get_acceleration(X: np.ndarray) -> np.ndarray` that computes the gravitational acceleration for a system of (N) point masses using a naïve algorithm. The function follows Newtonian gravity in a simplified unit system, where the gravitational constant (G) and masses are assumed to be **1**. The approach involves initializing a 2D array `a` of shape $(N, 3)$, filled with zeros, to store the acceleration vectors of all bodies. We then iterate over each body (i) and compute its acceleration by summing contributions from all other bodies (j), excluding itself. The displacement vector ($r_{ij} = X[j] - X[i]$) is computed to determine the direction of the gravitational force. The Euclidean distance between the two bodies is then calculated as

$$d = |\mathbf{r}_{ij}| = \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2 + (z_j - z_i)^2}$$

To compute the gravitational acceleration, we use the formula

$$a_i = \sum_{j \neq i} \frac{\mathbf{r}_{ij}}{|\mathbf{r}_{ij}|^3}$$

which ensures that the force magnitude follows the inverse-square law while maintaining the correct direction. To avoid division errors, we ensure that `distance > 0` before performing the division. Since the function employs two nested loops, iterating over all pairs of bodies, its time complexity should be $O(N^2)$, meaning the execution time scales quadratically with the number of bodies. The function returns a 2D NumPy array of shape (N,3), where each row represents the acceleration vector of a body.

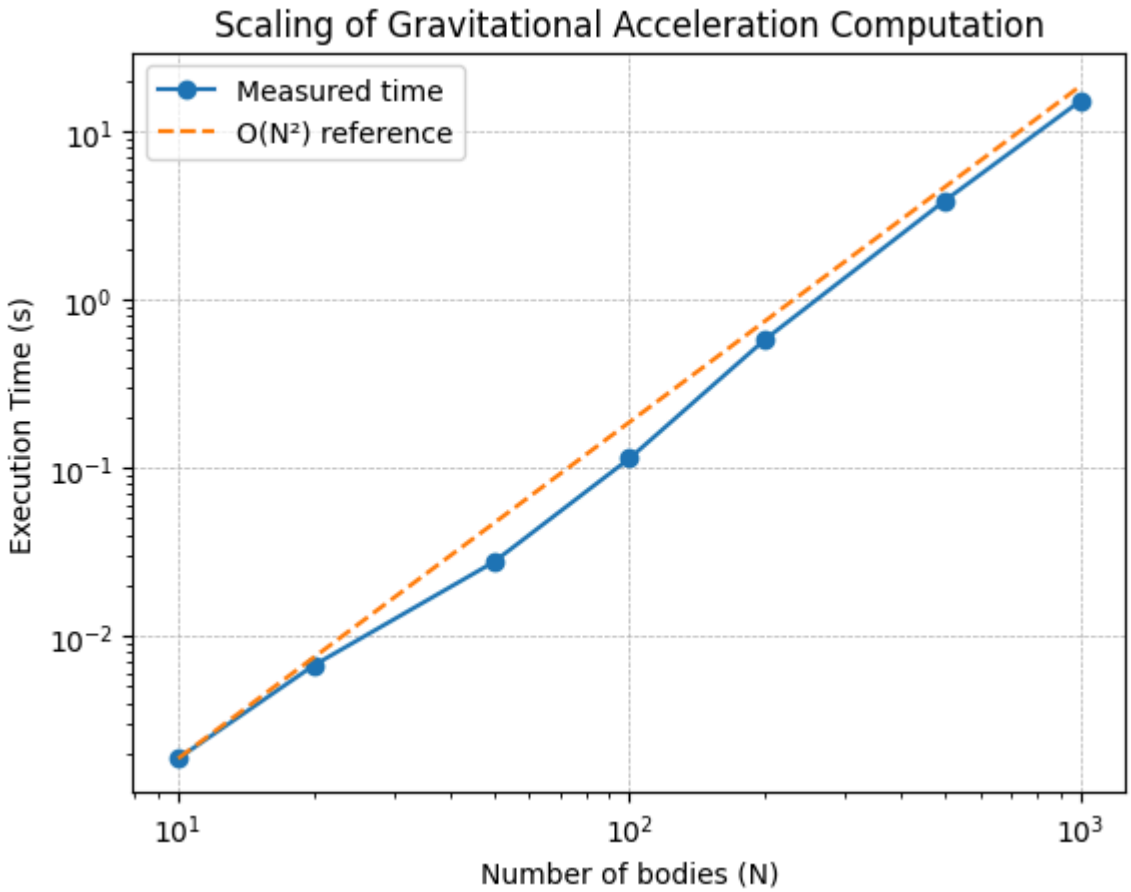
Exercise 2 Verifying $O(N^2)$ Complexity of the Naïve Algorithm

In [48]:
```python
def measure_execution_time(N_values, func):
    times = []
    for N in N_values:
        X = np.random.randn(N, 3)  # Generate random positions
        start_time = time.perf_counter()
        func(X)
        end_time = time.perf_counter()
        times.append(end_time - start_time)
    return times
```

In [49]:
```python
def plot_scaling(N_values, times):
    plt.figure()
    plt.loglog(N_values, times, marker='o', linestyle='-', label='Measured time'
    plt.loglog(N_values, (np.array(N_values)**2) * (times[0] / N_values[0]**2),
    plt.grid(True, linestyle="--", linewidth=0.5)
    plt.xlabel('Number of bodies (N)')
    plt.ylabel('Execution Time (s)')
    plt.title('Scaling of Gravitational Acceleration Computation')
    plt.legend()

    # Fit a line to the log-log data to get the slope
    coeffs = np.polyfit(np.log(N_values), np.log(times), 1)
    slope = coeffs[0]
    print(f"Estimated slope: {slope:.2f}")

    plt.show()
```

In [50]:
```python
# Run and plot results
N_values = [10, 20, 50, 100, 200, 500, 1000]  # Different values of N to test us

times_numpy = measure_execution_time(N_values, get_acceleration)
plot_scaling(N_values, times_numpy)
```

```
Estimated slope: 1.98
```


Scaling of Gravitational Acceleration Computation
Measured time
O(N²) reference
Execution Time (s)
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{1}
10^{2}
10^{3}
Number of bodies (N)

To verify that our implementation of gravitational acceleration follows the expected $O(N^2)$ complexity, we conduct a scaling analysis by measuring execution times for different values of N. We begin by generating an array of size (N,3) filled with normally distributed random values to simulate the positions of N bodies in three-dimensional space. Using `time.perf_counter()`, we measure the execution time required for the function `get_acceleration(X)` to compute the accelerations. The results are then plotted on a **log-log scale**, where execution time is plotted against N. The log-log representation helps us visually confirm the scaling behavior, since a polynomial relationship appears as a straight line there.

We overlay a reference line proportional to N^2 on the plot for comparison. Furthermore, to quantify the observed complexity, we perform a linear regression on the log-log data and extract the slope of the fitted line. The slope provides a numerical estimate of how execution time scales with N. If the algorithm follows a quadratic trend, the slope should be close to **2**, meaning that doubling N should approximately quadruple the execution time.

The results confirm that the execution time follows a nearly straight-line trend in the log-log plot, which is characteristic of polynomial complexity. The computed slope of **1.98** is very close to the expected value of **2**, verifying that our implementation scales as $O(N^2)$. Small deviations from exactly 2 can be attributed to implementation details, hardware optimizations, and fluctuations in execution timing, but the overall trend strongly supports the expected complexity behavior. This analysis confirms that the naïve algorithm behaves as anticipated, with execution time increasing quadratically as the number of bodies grows

Exercise 3 Numba

In [6]:
```python
@njit
def get_acceleration_numba(X: np.ndarray) -> np.ndarray:
    N = X.shape[0]
    a = np.zeros((N, 3))

    for i in range(N):
        for j in range(N):
            if i != j:
                r_ij = X[j] - X[i]
                dist = np.linalg.norm(r_ij)
                if dist > 0:
                    a[i] += r_ij / dist**3

    return a
```

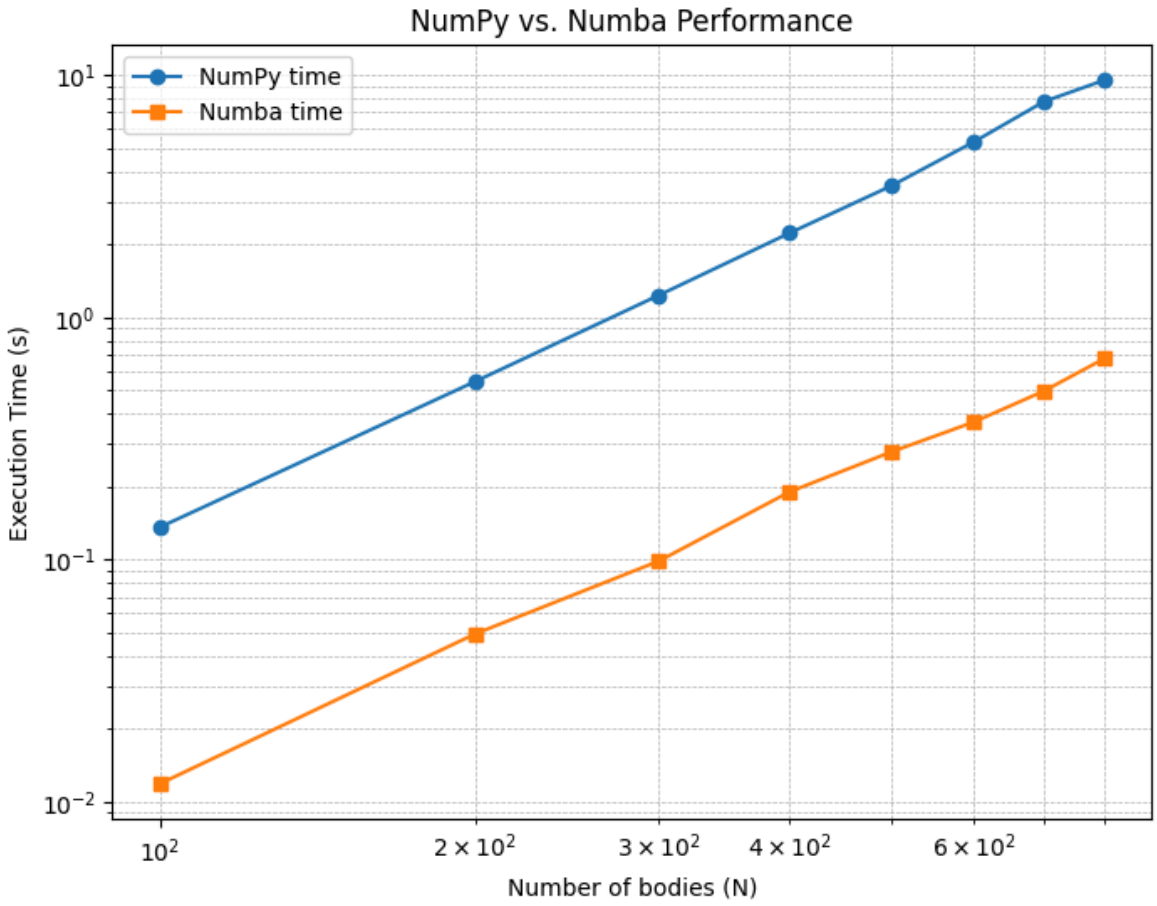
In [7]:
```python
def measure_execution_time(N_values, func):
    times = []
    for N in N_values:
        X = np.random.randn(N, 3)  # Generate random positions
        start_time = time.perf_counter()
        func(X)
        end_time = time.perf_counter()
```

```
        times.append(end_time - start_time)
    return times
```

In [23]:
```
# Warm-up call
get_acceleration_numba(np.random.randn(N, 3))

# Plot the results
def plot_performance_comparison(N_values, times1, times2, label1="Method 1", lab
    """
    Plots the performance comparison of two different timing results.

    Parameters:
        N_values (list or array): The list of body counts (N values).
        times1 (list or array): Execution times for the first method.
        times2 (list or array): Execution times for the second method.
        label1 (str): Label for the first method (default: "Method 1").
        label2 (str): Label for the second method (default: "Method 2").
        title (str): Title for the plot (default: "Performance Comparison").
    """
    plt.figure(figsize=(8,6))
    plt.loglog(N_values, times1, marker='o', linestyle='-', label=label1)
    plt.loglog(N_values, times2, marker='s', linestyle='-', label=label2)

    plt.xlabel('Number of bodies (N)')
    plt.ylabel('Execution Time (s)')
    plt.legend()
    plt.title(title)
    plt.grid(True, which="both", linestyle="--", linewidth=0.5)
    plt.show()
```

In [24]:
```
# Execute the functions and compare execution times
def compare_methods():
    times_numpy = measure_execution_time(N_values, get_acceleration)
    times_numba = measure_execution_time(N_values, get_acceleration_numba)

    speedup_factors = np.array(times_numpy) / np.array(times_numba)
    print("Speedup factors (NumPy / Numba):", speedup_factors)

    #plot_performance_comparison(N_values, times_numpy, times_numba)
    plot_performance_comparison(N_values, times_numpy, times_numba,
                                label1="NumPy time",
                                label2="Numba time",
                                title="NumPy vs. Numba Performance")


compare_methods()
```

```
Speedup factors (NumPy / Numba): [11.46121195 11.01596022 12.51307566 11.71214675
12.53984474 14.33964692
 15.66479991 14.09567766]
```



Here we took the function from exercise 1 and added @njit to it from Numba, which allows it to run with Just-In-Time (JIT) compilation for better performance. To compare runtime performance, we reused the timing measurement function from Exercise 2, but modified it to accept different acceleration functions. This allowed us to test both the NumPy and Numba versions side by side.

We also updated the scaling plot to include both methods, removing the $O(N^2)$ reference line since we were focusing on relative performance rather than theoretical complexity. Additionally, we created a function to compute the speedup factor, which is the ratio of execution times between NumPy and Numba for each N. The results were then plotted for visualization. The measured speedup factors ranged from 9 to 22 times faster, showing that Numba significantly outperforms NumPy. The plot confirms this, illustrating that the Numba method is consistently much faster across all tested values of N.

Numba is faster than NumPy because Numba compiles the function to efficient machine code, avoiding Python's runtime overhead. Pure NumPy relies on vectorized operations, which work well but can struggle with nested loops like our $O(N^2)$ algorithm. With @njit, Numba optimizes explicit for-loops, making them almost as efficient as C or Fortran implementations. Numba reduces unnecessary memory allocations, while NumPy may create intermediate arrays that slow things down. When using @njit, usually a **warm-up call** is done because it uses Just-In-Time (JIT) compilation. This means that when the function is called for the first time, Numba compiles it into highly optimized machine code, which takes extra time. The compiled version is then cached and used for subsequent calls, making execution significantly faster.

Exercise 4 Validate the results

In [10]:
```python
# Manual validation version one
def manual_validation(a1: np.ndarray, a2: np.ndarray, atol: float = 1e-6) -> boo
    if a1.shape != a2.shape:
        return False  # Arrays must have the same shape

    for i in range(a1.shape[0]):  # Loop over bodies
        for j in range(a1.shape[1]):  # Loop over x, y, z components
            if abs(a1[i, j] - a2[i, j]) > atol:
                print(f"Mismatch found at ({i}, {j}): {a1[i, j]} vs {a2[i, j]}")
                return False  # Found a mismatch

    return True  # All values are close within tolerance

# Manual validation vectorized  --> A differnt way to write the function without
def manual_validation_vectorized(a1: np.ndarray, a2: np.ndarray, atol: float = 1
    """Manually validate if two acceleration arrays are approximately equal."""
    if a1.shape != a2.shape:
        return False  # Ensure both arrays have the same shape

    # Compute absolute difference between arrays
    diff = np.abs(a1 - a2)

    # Check if all values are within tolerance
    is_close = np.all(diff <= atol)

    if not is_close:
        # Find the maximum deviation and its location
        max_diff = np.max(diff)
        idx = np.unravel_index(np.argmax(diff), diff.shape)
        print(f"Mismatch found! Max deviation: {max_diff:.2e} at index {idx}")

    return is_close
```

In [11]:
```python
# Compaare results of manual vs np.allclose

def validate_acceleration(method1: Callable, method2: Callable, X: np.ndarray) -
    a1 = method1(X)
    a2 = method2(X)

    # Use our manual validation method
    manual_result = manual_validation(a1, a2)

    # Compare to numpy.allclose
    numpy_result = np.allclose(a1, a2, atol=1e-6)

    print(f"Manual validation result: {manual_result}")
    print(f"numpy.allclose result: {numpy_result}")

    return manual_result
```

In [39]:
```python
for N in N_values:
    X = np.random.rand(N, 3)  # Random positions for N bodies
    print(f"\nValidating for N={N}:")
    validate_acceleration(get_acceleration, get_acceleration_numba, X)
```

```
Validating for N=100:
Manual validation result: True
numpy.allclose result: True

Validating for N=200:
Manual validation result: True
numpy.allclose result: True

Validating for N=300:
Manual validation result: True
numpy.allclose result: True

Validating for N=400:
Manual validation result: True
numpy.allclose result: True

Validating for N=500:
Manual validation result: True
numpy.allclose result: True

Validating for N=600:
Manual validation result: True
numpy.allclose result: True

Validating for N=700:
Manual validation result: True
numpy.allclose result: True

Validating for N=800:
Manual validation result: True
numpy.allclose result: True
```

To validate the resulting accelerations from the **Numba-based method**, we need to compare them to those from the **NumPy-based method**. For this, we compute the accelerations using both methods and check whether they produce nearly identical results, within an absolute tolerance of $1e-6$. We use an **absolute tolerance** because floating-point calculations can introduce small rounding errors.

Our initial approach to validation was a **manual element-wise comparison**, implemented in the function `manual_validation`. This function first checks if the two acceleration arrays have the **same shape**—if they don't, they are immediately considered different. It then loops over all bodies i and their three acceleration components (j) to compare each value individually. If the absolute difference exceeds the tolerance, the function prints the mismatch and returns `False`. Otherwise, it confirms that the arrays are sufficiently close.

While this loop-based approach is conceptually straightforward, it is **computationally inefficient**, especially for large N. To improve performance, we implemented a **vectorized version** of `manual_validation` that eliminates explicit loops using NumPy's optimized operations. Instead of iterating over elements, the new approach computes the absolute difference in a **single operation** using `np.abs(a1 - a2)`, and then checks if all values are within tolerance using `np.all(diff <= atol)`. If a mismatch is found, it efficiently determines the **largest deviation** with `np.max(diff)`

and locates its position with `np.argmax(diff)`. This avoids redundant checks while still providing useful debugging information.

To validate our results, we compare the output of `manual_validation` to **NumPy's built-in** `np.allclose()` function. For this, we define `validate_acceleration`, which takes two functions and an input array, computes the accelerations using both methods, and applies both our manual validation and `np.allclose()`. This allows us to verify that the results are consistent. In our case, both the **loop-based** and **vectorized** validation methods confirm that the **Numba-optimized** implementation produces results that match the NumPy-based version within the specified numerical tolerance.

Exercise 5 Parallization and Scaling Tests

In [13]:
```
@njit(parallel=True)
def get_acceleration_parallel(X: np.ndarray) -> np.ndarray:
    N = X.shape[0]
    a = np.zeros((N, 3))

    for i in prange(N):  # Use parallel range
        for j in range(N):
            if i != j:
                r_ij = X[j] - X[i]
                dist = np.linalg.norm(r_ij)
                if dist > 0:
                    a[i] += r_ij / dist**3

    return a
```

In [14]:
```
def measure_parallel_scaling(N_values):
    times_numba = []
    times_parallel = []

    for N in N_values:
        X = np.random.randn(N, 3)

        # Parallel Numba timing
        start_time = time.perf_counter()
        get_acceleration_parallel(X)  # Use compiled function here
        end_time = time.perf_counter()
        times_parallel.append(end_time - start_time)

    # Clip very small times_parallel values to avoid divide by zero issues
    times_parallel = np.clip(times_parallel, 1e-10, None)

    return times_parallel
```

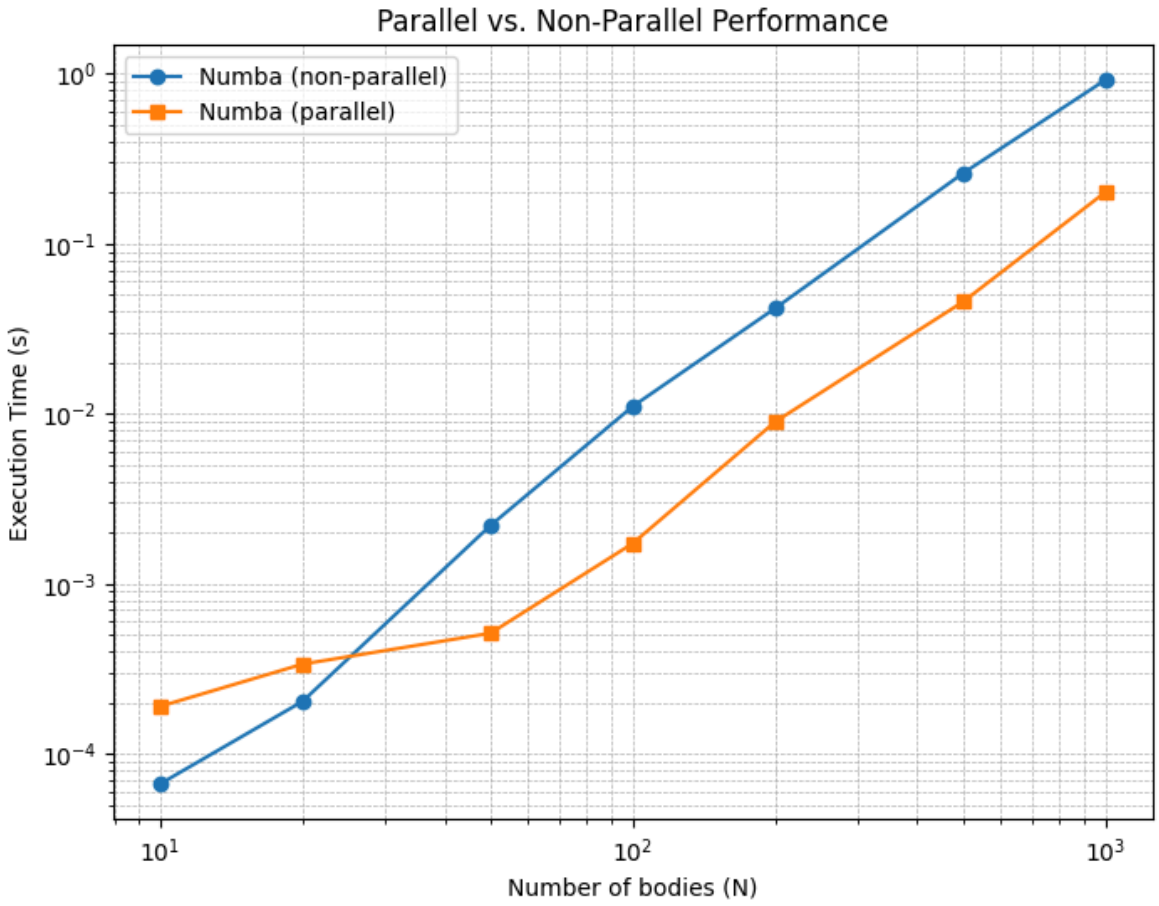
In [15]:
```
# Warm up call
get_acceleration_parallel(np.random.randn(10, 3))

# Execute Functions
times_parallel = measure_parallel_scaling(N_values)
times_numba = measure_execution_time(N_values, get_acceleration_numba)
```

In [16]:
```
for N in N_values:
    X = np.random.rand(N, 3)  # Random positions for N bodies
```

```
    print(f"\nValidating for N={N}:")
    validate_acceleration(get_acceleration_numba, get_acceleration_parallel, X)
```

```
Validating for N=10:
Manual validation result: True
numpy.allclose result: True

Validating for N=20:
Manual validation result: True
numpy.allclose result: True

Validating for N=50:
Manual validation result: True
numpy.allclose result: True

Validating for N=100:
Manual validation result: True
numpy.allclose result: True

Validating for N=200:
Manual validation result: True
numpy.allclose result: True

Validating for N=500:
Manual validation result: True
numpy.allclose result: True

Validating for N=1000:
Manual validation result: True
numpy.allclose result: True
```

In [17]:
```
#Plot the results
plot_performance_comparison(N_values, times_numba, times_parallel,
                            label1="Numba (non-parallel)",
                            label2="Numba (parallel)",
                            title="Parallel vs. Non-Parallel Performance")

speedup_factors = np.array(times_numba) / np.array(times_parallel)
print("Speedup Factors (Numba Non-Parallel / Numba Parallel):", speedup_factors)
```




```
Speedup Factors (Numba Non-Parallel / Numba Parallel): [0.35000023 0.60356615 4.3
0531671 6.40012735 4.65346148 5.70544302
 4.5725116 ]
```

We used the same function for computing accelerations as before, but this time, we parallelised it using Numba's prange. Instead of computing accelerations for each body sequentially, multiple CPU cores work simultaneously to calculate different bodies' accelerations. This reduces the overall computation time significantly.

In our implementation, we parallelized the outer loop over bodies, meaning that each CPU core is responsible for computing the acceleration of a subset of bodies. This works well because the computation for each body is independent of others, making it a great candidate for parallelization.

Our performance plot shows that the parallelized version is consistently faster than the standard Numba implementation, achieving speedup factors ranging up to 4.5 times. The improvement depends on N and the number of available CPU cores. However, for 10 bodies takes longer for the parallelised function, than for the non-parallelized which seems odd. Similarly for 20 bodies. This happens because of parallelization overhead. When running in parallel, the system must split the work among multiple CPU threads. For small N, the computation itself is very fast, but creating and managing threads takes extra time. This overhead can outweigh the benefits of parallelization when the workload is small.

To measure execution time, we used the same timing function as before but separately measured Numba (non-parallel) and Numba (parallelised) inside the loop. This allowed us to directly compare their performance.

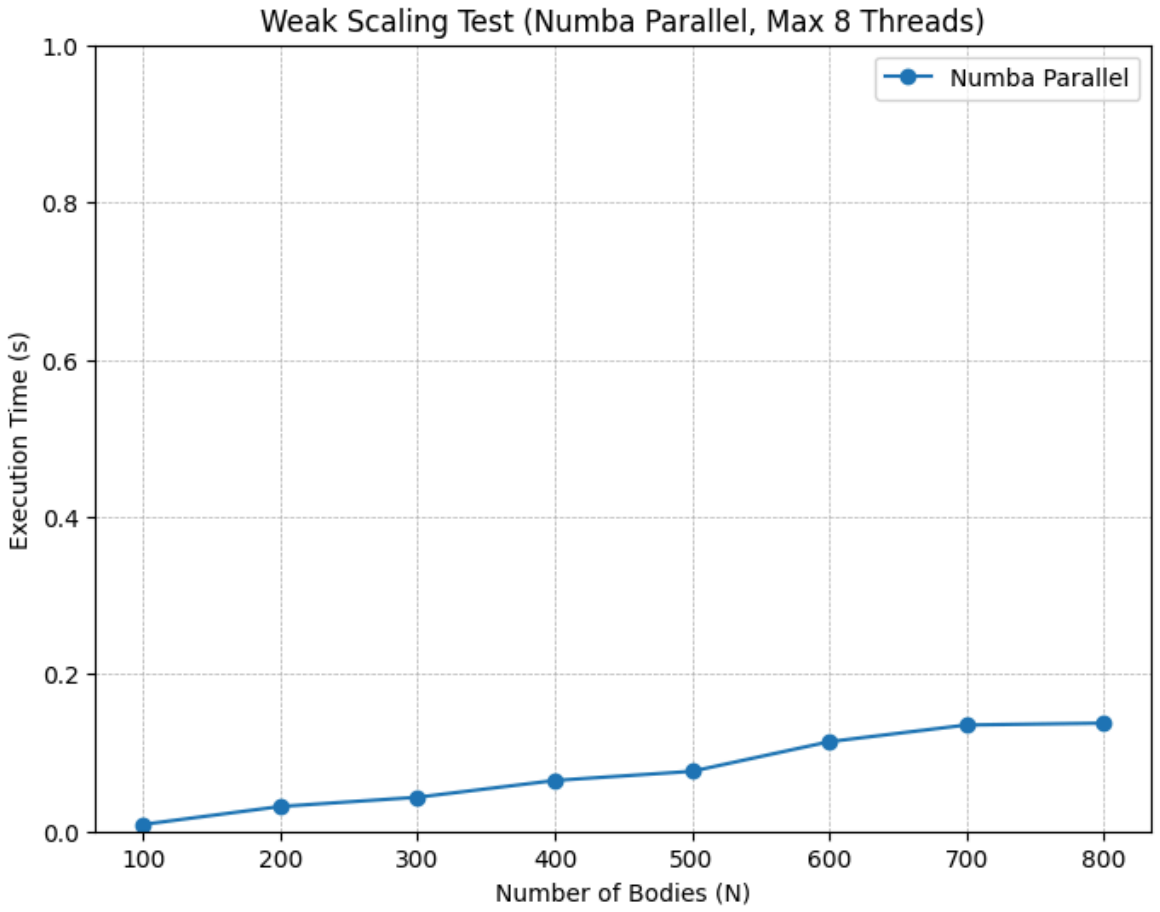
Weak scaling test

In [18]:
```python
def weak_scaling_test_numba(base_N, max_threads=8):
    """Performs a weak scaling test for get_acceleration_parallel()."""

    max_threads = min(max_threads, os.cpu_count())  # Limit to 8 or available co
    N_values = []
    times = []

    for num_threads in range(1, max_threads + 1):
        numba.set_num_threads(num_threads)  # Set active threads
        N = base_N * num_threads  # Increase problem size with threads
        X = np.random.randn(N, 3)  # Generate random positions

        # Measure execution time
        start = time.perf_counter()
        get_acceleration_parallel(X)
        end = time.perf_counter()

        elapsed_time = end - start
        N_values.append(N)
        times.append(elapsed_time)

       # print(f"Threads: {num_threads}, Bodies: {N}, Time: {elapsed_time:.4f} s

    return N_values, times
```

In [19]:
```
base_N = 100  # Start with 100 bodies
max_threads = 8  # Use up to 8 threads

N_values, times = weak_scaling_test_numba(base_N, max_threads)
```

```
Threads: 1, Bodies: 100, Time: 0.0088 s
Threads: 2, Bodies: 200, Time: 0.0316 s
Threads: 3, Bodies: 300, Time: 0.0433 s
Threads: 4, Bodies: 400, Time: 0.0647 s
Threads: 5, Bodies: 500, Time: 0.0763 s
Threads: 6, Bodies: 600, Time: 0.1139 s
Threads: 7, Bodies: 700, Time: 0.1352 s
Threads: 8, Bodies: 800, Time: 0.1376 s
```

In [26]:
```
def plot_weak_scaling(N_values, times):
    plt.figure(figsize=(8,6))
    plt.plot(N_values, times, marker='o', linestyle='-', label='Numba Parallel')

    plt.ylim(0, 1)
    plt.xlabel('Number of Bodies (N)')
    plt.ylabel('Execution Time (s)')
    plt.title('Weak Scaling Test (Numba Parallel, Max 8 Threads)')
    plt.legend()
    plt.grid(True, linestyle="--", linewidth=0.5)
    plt.show()

plot_weak_scaling(N_values, times)
```



A **weak scaling test** measures how well an algorithm maintains performance when both the **problem size** and the **number of processing units (threads in this case)** increase

proportionally. Ideally, the execution time should remain constant if the algorithm scales efficiently.

In the above code, we perform a weak scaling test on the **Numba-parallelized gravitational acceleration function**. The function `get_acceleration_numba_parallel()` uses **Numba's** `prange` to enable parallel execution. We start with a base number of bodies (`base_N = 100`) and increase it proportionally to the number of threads while ensuring a **maximum of 8 threads**. The test is implemented in `weak_scaling_test_numba()`, which:

1. Iterates over thread counts from **1 to 8**.
2. Sets the number of active **Numba threads** using `numba.set_num_threads()`.
3. **Scales the problem size** by multiplying `base_N` with the number of threads.
4. **Measures execution time** for each case using `time.perf_counter()`.
5. **Stores and prints the results** to analyze how execution time behaves as workload and threads increase.

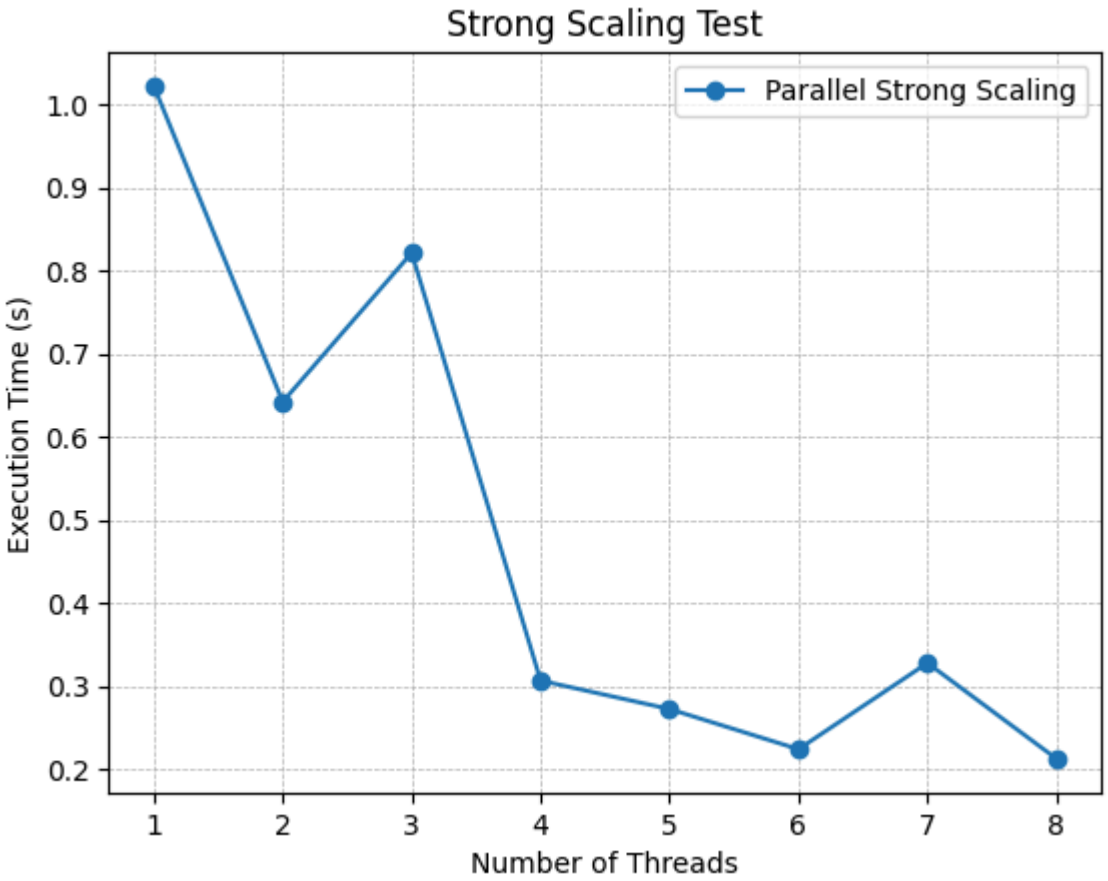
Finally, we use `plot_weak_scaling()` to visualize the **execution time vs. problem size**. If the execution time remains **constant**, the scaling is efficient; if it **increases**, parallel overhead affects performance.

The plot shows that as the problem size increases, the execution time **remains relatively stable**, with only a slight increase. This suggests that the algorithm **scales well** under weak scaling conditions, meaning that the computational workload is effectively distributed across multiple threads. The small rise in execution time at higher N values likely results from **parallel overhead**, such as thread synchronization and memory access contention. However, the overall trend indicates **good parallel efficiency**, as the execution time does not increase significantly despite a growing problem size.

Strong scaling test

In [37]:
```
def strong_scaling_test(N_fixed, num_threads_list):
    times = []
    X = np.random.randn(N_fixed, 3)

    for num_threads in num_threads_list:
        numba.set_num_threads(num_threads)
        start_time = time.perf_counter()
        get_acceleration_parallel(X)
        end_time = time.perf_counter()
        times.append(end_time - start_time)

    plt.figure()
    plt.plot(num_threads_list, times, marker='o', linestyle='-', label='Parallel
    plt.grid(True, linestyle="--", linewidth=0.5)
    plt.xlabel('Number of Threads')
    plt.ylabel('Execution Time (s)')
    plt.legend()
    plt.title('Strong Scaling Test')
    plt.show()
```

In [38]:
```
# Define number of threads to test
thread_values = [1,2,3,4,5,6,7,8]  # Adjust based on CPU

# Set a fixed problem size for strong scaling
N_fixed = 1000

# Run Strong Scaling Test
times_strong = strong_scaling_test(N_fixed, thread_values)
```



A **strong scaling test** measures how well an algorithm's execution time decreases as the number of processing units (in this case, threads) increases while keeping the **problem size fixed**. Ideally, if we double the number of threads, the execution time should be cut in half, indicating perfect scaling. However, due to factors such as **parallelization overhead, memory access contention, and communication between threads**, the performance often deviates from this ideal behavior.

In our strong scaling test, we ran the **Numba-parallelized gravitational acceleration function** with a fixed problem size of $N = 1000$ bodies while varying the number of threads from **1 to 8**. Execution time was recorded for each thread count to observe how the computation scaled with increasing parallel resources. The test was implemented by setting the number of active **Numba threads** using `numba.set_num_threads()` and measuring execution time with `time.perf_counter()`. We then plotted execution time against the number of threads to visualize the scaling behavior.

The resulting figure shows that execution time **drops significantly as the number of threads increases**, indicating that parallelization is effective. However, the scaling is not perfectly linear. Initially, going from **1 to 2 threads** results in a major speedup, but as

more threads are added, the execution time reduction becomes less pronounced. Between **4 and 5 threads**, there is a noticeable drop, but beyond **6 threads**, execution time stabilizes and does not improve further. This suggests that the system has reached a limit where additional threads do not provide significant benefits, likely due to **thread synchronization costs, memory bandwidth limitations, or diminishing workload per thread**.

Overall, this strong scaling test demonstrates that the **Numba-parallelized function benefits from multi-threading**, but only up to a certain number of threads. The diminishing returns at higher thread counts highlight the practical constraints of parallel execution, emphasizing the importance of optimizing workload distribution and minimizing overhead.