Write a program to implement parallel Bubble sort and Merge sort using OpenMP. Use existing algorithms and measure the perfoemance of sequential and parallel algorithms.

```
from numba import njit, prange
import numpy as np
import time

@njit(parallel=True)
def parallel_bubble_sort(arr):
    n = len(arr)
    for i in prange(n-1):
        for j in prange(n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]

def sequential_bubble_sort(arr):
    n = len(arr)
    for i in range(n-1):
        for j in range(n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]

arr = np.random.randint(0, 100000, size=10)

start_time = time.time()
sequential_bubble_sort(arr)
end_time = time.time()
print("Time taken by sequential bubble sort:", end_time - start_time)

start_time = time.time()
parallel_bubble_sort(arr)
end_time = time.time()
print("Time taken by parallel bubble sort:", end_time - start_time)
```

```
Time taken by sequential bubble sort: 8.749961853027344e-05
Time taken by parallel bubble sort: 0.8377890586853027
```

```
import numpy as np
import time
import concurrent.futures as cf

# Sequential merge sort function
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        L = arr[:mid]
        R = arr[mid:]

        merge_sort(L)
        merge_sort(R)

        i = j = k = 0

        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1

# Parallel merge sort function
def parallel_merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        L = arr[:mid]
        R = arr[mid:]
        with cf.ThreadPoolExecutor(max_workers=2) as executor:
            futures = [executor.submit(parallel_merge_sort, L), executor.submit(parallel_merge_sort, R)]

        L, R = futures[0].result(), futures[1].result()

        i = j = k = 0

        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1

    return arr

if __name__ == '__main__':
    n = 10
    arr = np.random.randint(0, 1000, n)

    start = time.time()
    merge_sort(arr)
    end = time.time()
    sequential_time = end - start
    print("Sequential merge sort time: {:.6f} s".format(sequential_time))

    start = time.time()
    sorted_arr = parallel_merge_sort(arr)
    end = time.time()
    parallel_time = end - start
    print("Parallel merge sort time with 2 threads: {:.6f} s".format(parallel_time))

    print("Speedup: {:.2f}x".format(sequential_time/parallel_time))
```

```
Sequential merge sort time: 0.000104 s
Parallel merge sort time with 2 threads: 0.017494 s
Speedup: 0.01x
```