INF236
Assignment 1
Alina Artemiuk

Problem 1

The function sampleSort takes as input a vector arr of unsigned long long integers, the size of the vector n, and the number of threads p. This function returns a sorted vector.

First, we split the input vector into p equal-sized subsequences, then sort each subsequence and store the sorted subsequences in the subseqVector vector.

Then, using the fillArrayWithDividers function, we fill the bucketDel vector with p-1 dividers of each of the p buckets.

fillArrayWithDividers is implemented according to the algorithm that was discussed earlier in the class. The function searches for bucket dividers as follows:

First, the function calculates the remainder of dividing the number of non-dividers by p - the number of buckets. Then we iterate from 1 to p - 1 inclusive and select elements from the array at positions increased by the step size. If there is a remainder, the step is increased by one until the remainder is completely used up. So, from each sorted sequence, we get p - 1 evenly spaced elements. Therefore, the length of the array with dividers that we will sort sequentially is p * (p - 1).

Next, using the fillArrayWithDividers function again, we fill the array, now bucketDel, with dividers using sorted vector divVector (found in the previous step), which contains p * (p - 1) elements with p - 1 dividers from each of the p buckets.

Now, having a vector of separators, we calculate the size of each bucket by counting the number of elements in each sub-sequence that fall into each bucket. For this, two loops are used, which work as follows:
The first for loop iterates over each row of sizeMat and each divider in bucketDel. It then iterates over each element in the subsequence corresponding to that row and increments the corresponding element in sizeMat if the element is less than the current divider.

The second loop iterates over each row of sizeMat and each divider except for the first one. It subtracts the sum of all previous elements in that row from the current element, effectively computing the number of elements in that subsequence that are less than the current divider but greater than or equal to the previous divider. Finally, it sets the last

element in each row to the number of elements in the subsequence that are greater than or equal to the last divider.
Thus, we get a sizeMat matrix of size p * p in which each row denotes the number of items to be placed in each of the p buckets. With this matrix, we can find the size of each bucket simply by calculating the sum of each column.

The next step is to create a Boolean matrix of flags to keep track of the items that have been placed in each bucket. The dimensions of this matrix are the same as the buckets.

Next, we fill the buckets with the subsequence elements that belong to each bucket. First, we iterate through each subsequence and check if element belongs to any bucket. If an element does not belong to any bucket and it is less than the bucket delimiter, it is added to the bucket and its flag is set to true to mark it as added. The variable bucketElemIndex keeps track of the index where the element should be inserted in the bucket. Once all the elements are added to the buckets, the last loop iterates through the subsequence elements again to find the ones that belong to the last bucket and adds them to it. Overall, subsequence elements are grouped into buckets based on their value with respect to the dividers. The flags matrix is used to ensure that each element is added to only one bucket.

Following the bucket sort algorithm, the next step after placing the items in their buckets is to sort each of them. To do this, we use the built-in function of the algorithm library – sort(). And finally, having completed all the steps, we concatenate all the buckets into one array.

Problem 2

n	p	part 1	part 2	part 3	part 4	in total
25000000	1					3.23457
25000000	2	2.38707	0.00002	0.45096	3.05984	6.18413
25000000	5	2.34191	0.00003	0.90844	0.80002	4.51284
25000000	10	2.32493	0.00002	1.13306	0.81522	4.75471
25000000	20	2.32817	0.00005	1.89831	0.78976	5.50506
25000000	40	2.33090	0.00021	3.48517	0.81475	7.11247
25000000	50	2.30519	0.00028	4.25442	0.82309	7.80459
25000000	80	2.26842	0.00063	6.61413	0.86149	10.21820
50000000	1					4.93581
50000000	2	4.84494	0.00001	0.89018	6.09349	12.39220
50000000	5	4.76704	0.00002	2.05248	1.85138	9.57476
50000000	10	4.68336	0.00002	3.80835	2.78211	12.32030
50000000	20	4.65744	0.00005	3.86643	1.54867	11.05890
50000000	40	4.66989	0.00016	6.96318	1.52858	14.15160

50000000	50	4.65949	0.00025	8.50249	1.53284	15.57820
50000000	80	4.62795	0.00096	13.21470	1.57913	20.41090
100000000	1					9.92001
100000000	2	9.80062	0.00001	1.77910	13.55190	26.27310
100000000	5	10.30650	0.00010	4.38529	4.26053	20.78640
100000000	10	10.48540	0.00003	8.23080	4.26046	25.03620
100000000	20	9.38792	0.00005	13.07040	3.62282	28.02880
100000000	40	10.01760	0.00015	20.45010	2.98521	35.41320
100000000	50	9.38296	0.00024	16.94070	2.98181	31.05300
100000000	80	9.35312	0.00058	27.46260	3.15217	42.01460

If the number of buckets p is 1, we will not have any separators, and thus sorting such an array will consist of using a built-in sort() function of the entire array without separations.

Looking at the results, it is clear that the time taken for each of the four parts of the algorithm varies depending on the values of n and p. However, some general trends can be observed:

For part 1 (initial sorting of subsequences), the time taken decreases as the number of buckets (p) increases. This is because as the number of buckets increases, each bucket contains fewer elements, and therefore the sorting time is reduced.

For part 2 (gathering and sorting dividers), the time taken increases as the number of buckets (p) increases. This is because as the number of buckets increases, there are more dividers to gather and sort, leading to a longer execution time.

For part 3 (placing elements in correct bins), the time taken is relatively constant for different values of p, but increases significantly as the value of n increases. This is because the number of elements being placed in bins increases, leading to longer execution times.

For part 4 (sorting local bins), the time taken decreases as the number of buckets (p) increases. This is because as the number of buckets increases, each bucket contains fewer elements, and therefore the sorting time is reduced.

Overall, the total execution time of the algorithm increases as the value of n increases. This is due to the increased time required for part 3 (placing elements in correct bins), which becomes the bottleneck as the size of the input increases. Execution time of the algorithm can be optimized by carefully choosing the values of n and p. Specifically, choosing a larger value of p can help reduce the execution time for parts 1 and 4, while choosing a smaller value of n can help reduce the execution time for part 3.

The largest array length that we can sort in about 10 seconds using the sequential version of the sample sort is 50 million.

Problem 3

Let's start with parallelizing the code that divides the array into p subsequences of length n/p and sorts them. The #pragma omp parallel directive is used to create a team of threads that will execute the following block of code in parallel. omp_get_thread_num() returns the ID of the current thread, which is used to calculate the range of indices that this thread will operate on. Each thread copies a portion of the input array into its own subsequence vector subseqVector[tid].

For sorting of each subsequence, each thread retrieves its own subsequence vector subseqVector[tid] and sorts it using the sort(). Multiple threads are working concurrently to sort the subsequence vectors in parallel, reducing the overall sorting time.
The second part, for which we recorded the execution time from Problem 2, takes so little time that there is no urgent need to parallelize this part of the code.

For sizeMat matrix, by using the #pragma omp atomic update directive inside the innermost loop, the code ensures that the shared variable sizeMat[i][k] is updated atomically by each thread. This prevents race conditions and ensures that the correct value is written to sizeMat[i][k] when multiple threads are executing the loop concurrently.

Also, the code uses the summation operation twice, first to find the sizeMat matrix, and then to find the vector of bucket sizes. The reduction(+:s) clause is used to perform a reduction operation on the variable s. The + operator indicates that the reduction operation is the sum of the individual thread's contributions to the variable. The s variable is private to each thread and its value is initialized to zero. After each thread completes its iterations, the partial sum computed by the thread is added to the global s variable using the reduction operation. This ensures that each thread's contribution to the sum is added correctly and eliminates race conditions that would arise if multiple threads updated the shared variables concurrently.

Next, two loops are used to place the elements in their buckets - one of them is used to place elements that are larger or equal to the last divider. By using the #pragma omp atomic capture directive inside the innermost loop, the code ensures that the shared variable bucketIndices is updated atomically by each thread. This prevents race conditions and ensures that the correct value is written to bucketIndices when multiple threads are executing the loop concurrently. The atomic capture clause captures the old value of bucketIndices and stores it in the variable bucketElemIndex, then updates by incrementing it by 1.

And the last part of the code is to parallelize the sorting of each of the buckets formed, using the same principle as the sorting of the p subsequences of length n/p that were formed at the beginning. Here, we use omp_get_thread_num() again.

Table with the results of the parallelized version of the sample sort with the same set of n and p as for the sequential program experiments:

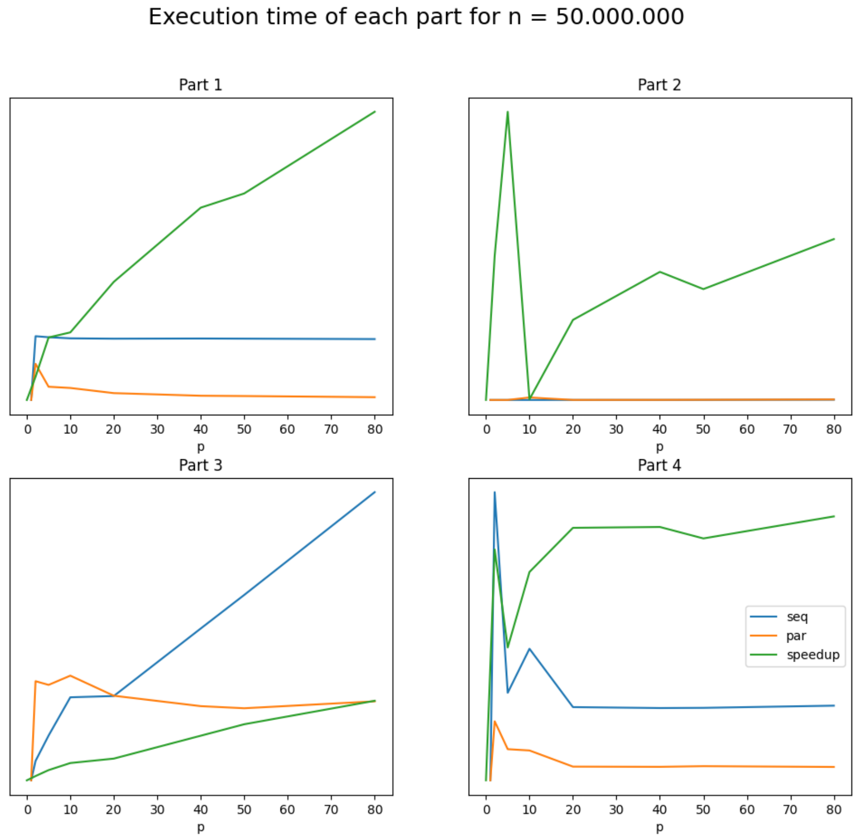
n	p	part 1	part 2	part 3	part 4	in total
25000000	1					3.14173
25000000	2	2.57627	0.00001	2.22304	0.93104	6.2137
25000000	5	0.85412	0.00001	2.69213	0.42431	4.47011
25000000	10	0.51132	0.00003	2.92929	0.31139	4.48914
25000000	20	0.26423	0.00009	2.97987	0.18267	4.1751
25000000	40	0.15839	0.00025	1.95424	0.14011	2.81692
25000000	50	0.13049	0.00057	2.04264	0.18703	2.97613
25000000	80	0.08657	0.00149	1.87446	0.13283	2.62952
50000000	1					6.5978
50000000	2	2.74889	0.00002	4.54695	1.24859	9.13033
50000000	5	1.00376	0.00002	4.3766	0.65901	6.94732
50000000	10	0.91189	0.00866	4.80002	0.63153	7.40242
50000000	20	0.51873	0.00018	3.87995	0.29004	5.73918
50000000	40	0.31946	0.00036	3.40273	0.28532	5.09978
50000000	50	0.29681	0.00065	3.30577	0.29967	4.82395
50000000	80	0.21126	0.00172	3.62254	0.28296	5.1895
100000000	1					9.89572
100000000	2	6.81773	0.00006	10.5314	3.67834	22.2434
100000000	5	2.53756	0.00003	10.5594	1.92409	16.993
100000000	10	1.66518	0.00006	9.22306	1.24581	14.2302
100000000	20	0.91109	0.0001	6.82863	0.61007	11.4529
100000000	40	0.60064	0.00051	6.38348	0.64747	10.0593
100000000	50	0.58256	0.00555	6.50997	0.51832	10.0533
100000000	80	0.39258	0.00163	6.48237	0.58389	10.4984

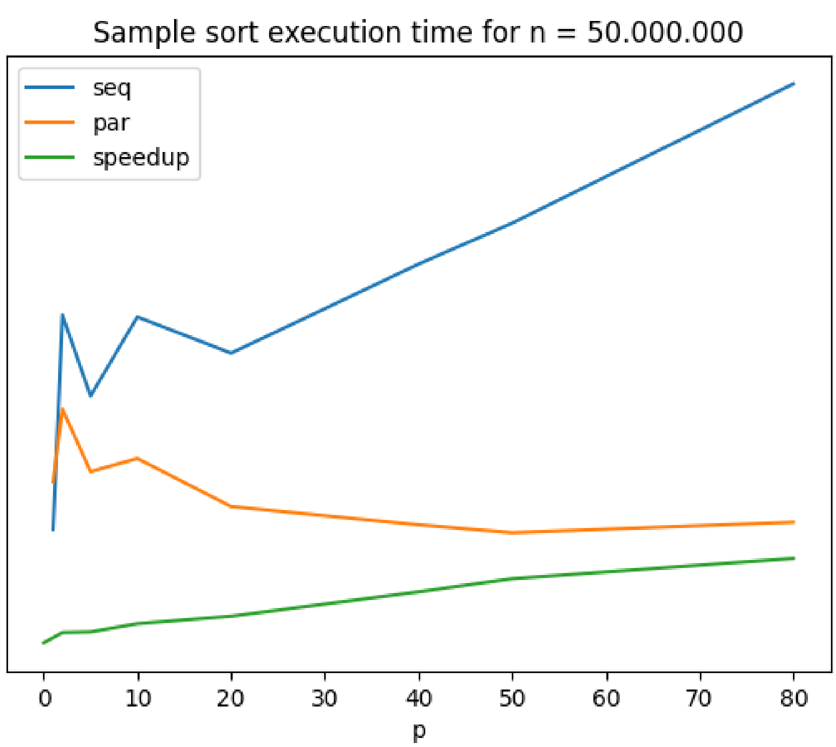
For n = 50 million, starting from p = 20, the sorting time is less than the sorting time using 1 thread and the built-in sort() function.

Problem 4

Strong scaling:

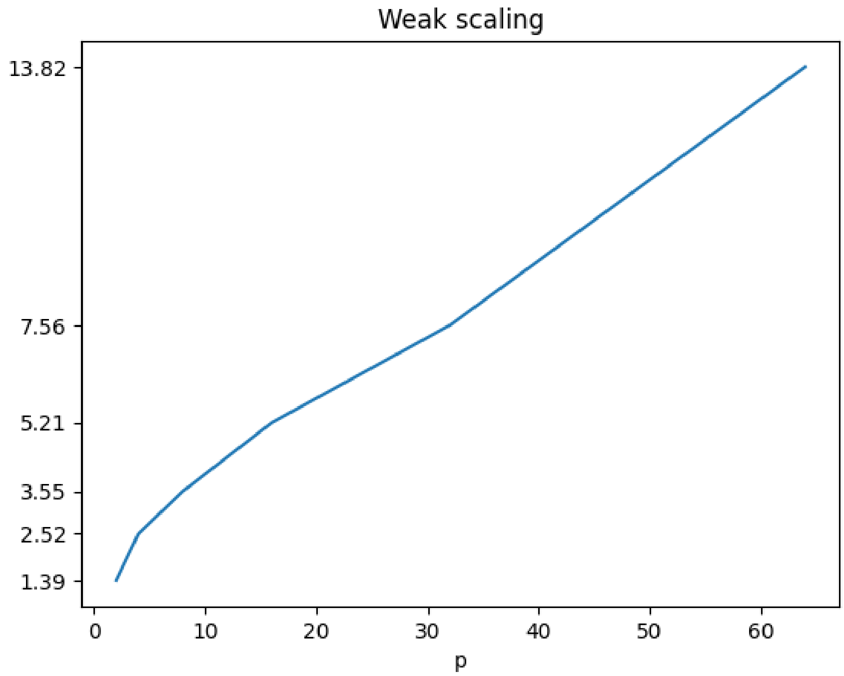
Here are the graphs built using the matplotlib library in Python:





Weak scaling results:

n	p	part 1	part 2	part 3	part 4	in total
5000000	2	0.52216	0.00001	0.60624	0.20717	1.39153
10000000	4	0.51870	0.00001	1.51485	0.33171	2.51869
20000000	8	0.49495	0.00002	2.40183	0.33470	3.54834
40000000	16	0.52649	0.00850	3.21323	0.39954	5.20903
80000000	32	0.57383	0.00027	5.05498	0.70746	7.56402
160000000	64	0.67820	0.01671	9.34030	1.22968	13.82340



Speedup of each part, as well as sorting time in general:

n	p	part 1	part 2	part 3	part 4	in total
25000000	1					1
25000000	2	0.92656	2.00000	0.20286	3.28648	0.99524
25000000	5	2.74190	3.00000	0.33744	1.88546	1.00956
25000000	10	4.54692	0.66667	0.38680	2.61800	1.05916
25000000	20	8.81115	0.55556	0.63704	4.32342	1.31855
25000000	40	14.71621	0.84000	1.78339	5.81507	2.52491
25000000	50	17.66564	0.49123	2.08280	4.40084	2.62240
25000000	80	26.20330	0.42282	3.52855	6.48566	3.88596
50000000	1					1
50000000	2	1.76251	0.50000	0.19578	4.88030	1.35726
50000000	5	4.74918	1.00000	0.46897	2.80934	1.37819
50000000	10	5.13588	0.00231	0.79340	4.40535	1.66436
50000000	20	8.97854	0.27778	0.99652	5.33950	1.92691
50000000	40	14.61807	0.44444	2.04635	5.35742	2.77494

50000000	50	15.69856	0.38462	2.57201	5.11509	3.22935
50000000	80	21.90642	0.55814	3.64791	5.58075	3.93311

100000000	1					1
100000000	2	1.43752	0.16667	0.16893	3.68424	1.18116
100000000	5	4.06158	3.33333	0.41530	2.21431	1.22323
100000000	10	6.29686	0.50000	0.89242	3.41983	1.75937
100000000	20	10.30405	0.50000	1.91406	5.93837	2.44731
100000000	40	16.67821	0.29412	3.20360	4.61058	3.52044
100000000	50	16.10643	0.04324	2.60227	5.75284	3.08884
100000000	80	23.82475	0.35583	4.23651	5.39857	4.00200

The best speedup results were obtained by sorting sequences of size n/p using p threads in Part 1 of Problem 2.