1 Question 1

Performance Analysis

Breakdown

- Each process generates N random numbers (N).
- Generated numbers sorted using `qsort`, takes $N \log N$.
- Each process partitions its sorted array into splits, cost:

  $$p \log N,$$

  p is the number of processes. $p \ll N$, cost is negligible compared to $N \log N$.
- Each process sends counts (the number of elements to be sent to each process). cost

  $$p \cdot \text{latency}.$$

- sends parts of its sorted array to other process. The cost:

  $$T_{\text{comm-data}} \approx p \cdot \text{latency} + N \cdot t_{\text{comm}},$$

- Final Sort: After exchange, each process receives likely N numbers. Sorting numbers again takes:

  $$N \log N.$$

- Overall:

  $$T_p \approx N \log N + \Big(p \cdot \text{latency} + N \cdot t_{\text{comm}}\Big).$$

Serial

A serial algorithm would take:

$$T_s \approx pN \log(pN).$$

Ignore communication overhead, the parallel process would take:

$$\frac{T_s}{p} \approx N \log(pN \approx N(\log N + \log p).$$

So, ideal speedup S can be:

$$S \approx \frac{T_s}{T_p} \approx \frac{c_{\text{comp}}\, pN \log(pN)}{c_{\text{comp}}\, N \log N + (p \cdot \text{latency} + N \cdot t_{\text{comm}})},$$

close to p if we neglect communication overhead.

Speedup & Efficiency

Let c_{comp} is the cost per computational. Then:

$$T_p \approx c_{\text{comp}}\, N \log N + \Big(p \cdot \text{latency} + N \cdot t_{\text{comm}}\Big),$$

serial is:

$$T_s \approx c_{\text{comp}}\, pN \log(pN).$$

speedup is:

$$S = \frac{T_s}{T_p} \approx \frac{c_{\text{comp}}\, pN \log(pN)}{c_{\text{comp}}\, N \log N + (p \cdot \text{latency} + N \cdot t_{\text{comm}})}.$$

parallel efficiency E is:

$$E = \frac{S}{p} \approx \frac{c_{\text{comp}}\, N \log(pN)}{c_{\text{comp}}\, N \log N + (p \cdot \text{latency} + N \cdot t_{\text{comm}})}.$$

to achieve high efficiency, necessary that:

$$c_{\text{comp}}\, N \log N \gg p \cdot \text{latency} \quad \text{and} \quad c_{\text{comp}}\, N \log N \gg N \cdot t_{\text{comm}}.$$

Isoefficiency

To determine how the problem size N must scale with the number of processes p and have a constant efficiency.

$$N \log N \gtrsim K\Big(p \cdot \text{latency} + N \cdot t_{\text{comm}}\Big),$$

K is a constant.

- If latency is the dominant,

$$N \log N \gtrsim K\, p \cdot \text{latency}.$$

  N must grow with p.

- if communication cost dominates,

$$N \log N \gtrsim K\, N \cdot t_{\text{comm}},$$

  or equivalently,

$$\log N \gtrsim K\, t_{\text{comm}}.$$

  for large N, the communication cost is negligible.

N to get 30 seconds

Used `MPI_Wtime()` to measure the time to test. After different experiments approximately 1000000 runs in 30 seconds.

2 Question 2

To analyze and compare the performance of the three functions, Table 1 shows the execution times when varying numbers of threads with a fixed array size of 1000. Table 2 shows execution times when varying the array size with a fixed number of threads (8).

Analysis of Table 1 Results

In the **serial** code when varying the number of threads with a fixed array size 100, the execution time is stable, with average around 0.0037 seconds. Since the method is serial, it does not utilize multiple threads it just has small variations which is likely due to noise or system overhead. In **compute_tasks** the time is significantly higher. this function uses OpenMP tasks for parallelization, but the results suggest it has poor scalability. The execution time likely due to overhead in task creation and management or dependencies between tasks that which prevents effective parallel execution. In **compute_tasks_blocks** function the time decreases significantly, then stabilizes maybe due to limited parallelism or overhead. However, this method reduces execution time compared to the serial code.

Function	#Threads	Time (seconds)
compute_serial	2	0.003715
	4	0.003759
	6	0.003654
	8	0.003745
	10	0.003732
	12	0.003646
compute_tasks	2	0.940949
	4	0.926726
	6	0.956288
	8	0.959764
	10	0.937874
	12	0.944456
compute_tasks_blocks	2	0.004459
	4	0.001647
	6	0.001182
	8	0.001197
	10	0.001292
	12	0.001286

Table 1: Comparison of time using different number of threads with a fixed array size 1000.

Analysis of Table 2 Results

In the **serial** code, when varying the array size with a fixed number of threads (8), execution time increases with array size. As a serial function, it scales poorly with larger sizes. In **compute_tasks** times are higher than serial and it increases with array size. The performance is poor, likely due to the overhead of creating tasks and managing dependencies which outweighs parallelization, making it slower. In **compute tasks blocks**, The times increase with array size but it remain lower than serial. For small sizes (N=10), it's slower than serial. With larger arrays the method achieves speedup over serial. For small arrays, parallel overhead exceeds benefits.

Function	Array Size	Time (seconds)
compute_serial	2000	0.015522
	1000	0.003775
	600	0.001337
	500	0.000935
	100	0.000038
	10	0.000001
compute_tasks	2000	3.60650
	1000	0.924907
	600	0.305320
	500	0.247356
	100	0.012057
	10	0.000430
compute_tasks_blocks	2000	0.005182
	1000	0.001740
	600	0.000671
	500	0.000491
	100	0.000082
	10	0.000099

Table 2: Comparison of time using varying array sizes with a fixed number of threads 8.