Performance Enhancement of Quicksort through

Parallelization using MPI

Introduction:

Quicksort is a widely known sorting algorithm due to its efficient average-case performance. However, as datasets grow larger, the time taken by the algorithm on a single processor can become significant. The key to addressing this challenge lies in the parallel processing capabilities of MPI (Message Passing Interface). This report delves into how the quicksort algorithm was sped up using MPI and how it was ensured that every worker handled N/P elements throughout the process in qs_mpi.c.

1. Performance Boost through Parallelism:

a. Data Decomposition: Instead of a single processor sorting an array of N elements, the dataset was divided into P segments with roughly N/P elements each. Each worker (or process) was then responsible for sorting its own segment. By distributing the work, the time taken for sorting was significantly reduced as multiple processes executed the task simultaneously.

b. Parallel Pivot Selection: In standard quicksort, selecting a pivot is a sequential task. However, in our parallelized version, each worker chose a local median as a pivot candidate from its segment. These were then collated to choose the median of medians as the global pivot. This approach ensured a more representative pivot for the entire dataset, leading to more balanced partitions, and hence, faster sorting.

c. Reduced Communication Overhead: By leveraging `MPI_Comm_split`, the processes were grouped based on data partitions (less than or greater than the pivot). This allowed these groups to work independently in a recursive manner, reducing the need for frequent inter-process communications.

Comparison of Performance based amount of worker processes and data set sizes:

		Size of dataset	
	1000000	10000000	100000000
1	1.1378s	13.2898s	149.8805s
2	0.3742s	4.3803s	49.9427s
Number of threads 4	0.2304s	2.5909s	29.6497s
6	0.2325s	1.7807s	19.4221s
8	0.1498s	1.4244s	15.3998s
16	0.1022s	0.7265s	7.8802s

2. Ensuring Each Worker Managed N/P Elements:

a. Consistent Data Segmentation: The initial division of data ensured that each worker started with roughly N/P elements.

b. Preserving Partition Balance: After each round of partitioning based on the global pivot, the total number of elements across workers remained consistent. Some workers might have had more elements less than the pivot and others more elements greater than the pivot. However, the combined data after partitioning remained roughly N/P elements per worker.

c. Recursive Balance: As the algorithm recursively partitioned the data, processes were split based on their rank, not data size. This ensured a balanced number of workers in each recursive call. As a result, every recursive iteration guaranteed each worker managed approximately N/P elements.

3. Concluding Observations:

Parallelizing quicksort using MPI provided a substantial speed-up, primarily due to simultaneous processing of data segments and an optimized pivot selection mechanism. The strategy ensured that the workload remained balanced across all workers, with each consistently managing N/P elements. This balance not only optimized performance but also ensured predictability in execution, making the algorithm more reliable for large datasets in distributed environments.The key takeaway is that even traditional algorithms, when adapted effectively to parallel paradigms, can realize significant performance improvements, ensuring their relevance and utility in the age of big data and distributed computing.