Individual Activity: Exploring Parallel Sorting Algorithms

Subject: ELEC3A – Parallel and Distributed Computing

Week 12 Topic: Understanding Different Parallel Sorting Algorithms

Prepared by: [Deocareza, Laurence Jade A.]

Based on Material by: Celine Dianne Montan

Objective:

This activity aims to deepen your understanding of parallel sorting algorithms by analyzing their strategies, strengths, weaknesses, and performance through practical comparison using a common dataset.

Instructions:

- Answer all questions in Parts A, B, and C.
- Use the provided dataset to compare the efficiency of three parallel sorting algorithms:
  - Parallel Merge Sort
  - Parallel Quicksort
  - Parallel Radix Sort
- You may code your solutions using a programming language of your choice, use online sorting visualizers, or simulate manually.
- Submit your answers in a typed document (PDF or Word). Include screenshots if you code or simulate.
- File name format: Lastname_Firstname_ParallelSortingActivity
- Deadline: [5/10/2025]

Part A: Concept Understanding

1. Define parallel sorting.
   In your own words, explain what makes parallel sorting different from sequential sorting.

Answer: Parallel sorting is the process where the task of sorting data is distributed on different multiple threads or cores so that they sort the data faster. This differs from sequential sorting, where it only utilises one thread or corestep by step; parallel sorting splits the work among the processors, which yields faster results.

2. List three real-world applications where parallel sorting plays a crucial role. Briefly describe how it contributes to performance in each case.

Answer:

Image and Video Processing: In high-resolution image or frame sorting (e.g., for compression, analysis, or rendering), parallel sorting helps manage pixel or frame data efficiently across GPU or CPU cores.

Simulations: Simulations in physics, climate modeling, or genomics often involve sorting large arrays of data. Parallel sorting speeds up data organization, which is critical for time-sensitive simulations and for handling results across thousands of variables.

Machine Learning: During data preprocessing stages, sorting large datasets (e.g., for feature selection or batching) is common. Parallel sorting shortens training preparation time, improving the overall speed of model development.

Part B: Algorithm Analysis

3. Compare the following algorithms:

Algorithm	Main Strategy	Strengths	Weaknesses
Parallel Merge Sort	Divide-and-conquer with parallel splitting, concurrent sorting and merging	Efficient and stable; good load balancing during merging; suitable for large data sets	Merging step can be complex and memory-intensive; may involve significant synchronization overhead
Parallel Quicksort	Parallel partitioning and optimized recursive division	Fast average-case performance; effective use of multiple threads for partitions	Sensitive to data distribution; imbalanced partitions can cause load imbalance and performance drops
Parallel Radix Sort	Digit-based bucketing and parallel digit processing	Excellent for large datasets with fixed-length keys; avoids comparisons; highly efficient for uniform data	Complex implementation; high memory usage; not suitable for variable-length or floating-point data

Part C: Performance Activity

Step 1: Use the Dataset Below

You will use this dataset of 100 random integers:

895, 123, 456, 981, 342, 5, 745, 601, 230, 84,
512, 900, 77, 33, 695, 288, 821, 149, 603, 788,
109, 344, 912, 305, 666, 731, 118, 53, 477, 819,
225, 411, 602, 702, 309, 199, 844, 37, 369, 740,
643, 88, 19, 333, 721, 555, 486, 231, 890, 3,
112, 888, 467, 188, 139, 728, 282, 914, 750, 489,
14, 92, 634, 780, 297, 122, 817, 376, 205, 900,
198, 65, 342, 478, 119, 655, 982, 40, 505, 316,
324, 101, 920, 366, 888, 754, 963, 710, 337, 802,
300, 60, 490, 802, 456, 78, 400, 672, 234, 105

Step 2: Compare Sorting Algorithms

Implement or simulate each of the following algorithms:

- Parallel Merge Sort
- Parallel Quicksort
- Parallel Radix Sort

Record the following:

Algorithm	Time Taken / Steps Counted	Observations (e.g., memory use, CPU efficiency, load balance)
Parallel Merge Sort	0.000999928 seconds	Memory usage is a bit high due to temp arrays, and the CPU efficiency isn't that noticeable with such a small dataset.
Parallel Quicksort	0 seconds	Very fast for small datasets, but task imbalance might show up with larger ones.
Parallel Radix Sort	0.00100017 seconds	Memory usage is higher due to buckets, and parallelizing can add overhead from locking mechanisms.

Step 3: Answer Reflection Questions

4. Which algorithm sorted the dataset the fastest? Provide reasoning based on your results.

Answer: Parallel Quicksort seems to be the fastest, specifically by its in-place sorting in which sorts the array in place, meaning it doesn't require extra memory for temporary arrays or structures.

5. Which one required the most processing steps or passes? Explain how the structure of the algorithm contributes to this.

Answer: Merge Sort requires the most processing steps or passes because of its divide-and-conquer approach. The algorithm recursively splits the array into smaller subarrays until each subarray contains a single element, and then it merges them back together.

6. If you were to sort a dataset of 10,000+ integers, which algorithm would you choose? Justify your choice considering scalability, memory usage, and processor distribution.

Answer: If I were to sort a dataset of 10,000+ integers, I'd go with Parallel Radix Sort because it scales better for large numeric datasets by avoiding comparisons and instead sorting based on digit positions. It spreads the workload efficiently across threads during the bucketing phase, making processor usage more balanced. While it does use more memory for temporary buckets, the trade-off is worth it since it finishes faster and handles uniform data types like integers really well.

7. Discuss one performance challenge (e.g., synchronization overhead, load imbalance) you observed and suggest a possible solution or improvement.

Answer: One performance challenge I noticed was load imbalance in Parallel Quicksort — if the pivot choice results in uneven partitions, some threads finish early while others do more work. A possible improvement is using a better pivot selection strategy like median-of-three by choosing the median value among first element, middle element, and last element or even switching to a different algorithm like Merge Sort when partitions get too unbalanced.

Bonus (Optional):

- Try sorting a larger dataset (e.g., 1,000 or 10,000 numbers) and describe how performance scaled for each algorithm.
- Share your code in the appendix section of your submission.