Parallel Sorting Performance Evaluation

This document reflects the performance evaluation and analysis of two parallel sorting implementations: MPI-Only and Hybrid MPI + OpenCL, using the Data Sorting Algorithm (Quicksort strategy).

1. System Configuration

Parameter	Value
Algorithm Used	Data Sorting (Quicksort Decomposition)
Local Sort Technique (MPI-Only)	std::sort (optimized C++ Quicksort variant)
Local Sort Technique (MPI + OpenCL)	Bitonic Sort (GPU-optimized)
Processor (CPU)	Apple M3 Pro
Operating System	macOS ARM64
OpenCL Device (GPU/CPU)	Integrated GPU
Total MPI Processes Tested	4

2. Performance Data

The baseline sequential time from a comparable module 2 task is assumed to be roughly **10,000 ms** for to provide context for the MPI-Only model's speedup. The primary comparison is between **MPI-Only** (the current baseline) and the **Hybrid MPI + OpenCL** model.

Data Size (N)	MPI-Only Time (ms)	Hybrid MPI + OpenCL Time (ms)	Speedup (MPI-Only / Hybrid OCL)
10,000,000	1108	1232	**0.90x** (Slower)
20,000,000	2195	2468	**0.89x** (Slower)
30,000,000	3209	3317	**0.97x** (Slower)
40,000,000	4290	4878	**0.88x** (Slower)
50,000,000	5379	5861	**0.92x** (Slower)

3. Analysis and Decomposition

3.1. Decomposition Strategy

- **Strategy:** The array was divided into 4 chunks (equal to the number of processes) using .
- **Parallelism:** The *Local Sort* phase is fully parallel. Each process sorts its local chunk independently, either on the CPU (MPI-Only) or the GPU (MPI+OpenCL).
- **Communication:** is used for initial distribution, and is used for final collection.
- **Global Result:** The global sorted result requires a final **Merge** step on the root process

(Rank 0) to combine the 4 sorted segments.

3.2. Performance Reflection (MPI-Only vs. Hybrid MPI + OpenCL)

The results show that the **MPI-Only baseline is superior** for this distributed sorting task, being consistently faster than the Hybrid MPI + OpenCL approach by 3% to 12% across all tested sizes.

- **GPU Overhead is Dominant:** Unlike matrix multiplication where the compute intensity is extremely high, local sorting involves significant **data movement**. The model introduces two major overheads that cripple its performance:
    1. **Host-to-Device Transfer:** Copying the local data chunk from the 's to the 's memory.
    2. **Device-to-Host Transfer:** Copying the sorted data back to the 's so the process can participate in the and final merge.
- **CPU Optimization:** The -Only model avoids this transfer cost entirely. The local sorting is done by the standard library's std::sort, which is a highly optimized function that executes on data already in the 's cache. The time saved by the 's parallel execution is completely outweighed by the or shared memory transfer latency.
- **Scalability:** Both methods show excellent linear scaling with array size, meaning the local sorting time doubles when the array size doubles. However, the constant overhead of data transfer keeps the version operating at a slower offset.

3.3. OpenCL Local Sort Choice

- **Local Sort:** Bitonic Sort was chosen for the kernel instead of Quicksort.
- **Reasoning:** architectures (like the one in your) require algorithms with minimal branching and predictable memory access patterns. Quicksort is highly recursive and branch-heavy, making it inefficient on . Bitonic Sort is a naturally parallel, branch-free, **data-parallel algorithm** that is far more suitable for execution, even though in this case, its benefits were negated by data transfer costs.

4. Conclusion

For the distributed data sorting task using a **"Divide, Local Sort, Merge"** decomposition, the **MPI-Only model is the superior choice** on this hardware.
The -Only model maximizes efficiency by utilizing the 's highly optimized sorting routines (std::sort) on local data, eliminating the substantial -to- data transfer overhead that made the hybrid slower. The parallel efficiency of the Bitonic Sort was not sufficient to overcome the latency of moving the large data chunks across the host-device boundary.