Bitonic Sort in Parallel Computing Using OpenMP, MPI, and CUDA

Student Name: D.J.N. Dissanayaka
Student ID: IT23279384
Course: SE3082 – Parallel Computing
Assignment: Assignment 03

Table of Contents

- Introduction
- Implementation Approaches
- Results & Performance Evaluation
- Speedup Analysis
- Scalability Discussion
- Screenshots Section
- Conclusion
- References
- Appendix

3. Introduction

Bitonic sort is a parallel sorting algorithm initially devised by Ken Batcher. It operates by forming bitonic sequences and then recursively merging them. The sorting network has time complexity $O(n(\log n)^2)$ with a delay of $O((\log n)^2)$[1]. Bitonic sort is data-independent (the compare-exchange pattern is fixed), which makes it well-suited for parallel hardware and SIMD architectures[1]. For example, bitonic mergesort networks use $O(n(\log n)^2)$ comparators and are especially efficient on architectures with many parallel execution units (like GPUs)[1]. This inherent parallelism makes bitonic sort attractive for high-performance implementations. In this report, we implement bitonic sort in serial, and in three parallel paradigms (OpenMP, MPI, and CUDA) and compare their performance on a dataset of size 2048.

4. Implementation Approaches

4.1 Serial Version

The serial implementation uses a single-threaded C program executing the standard bitonic sort procedure. It typically involves nested loops or recursive calls to perform the successive bitonic splits and merges. The data is entirely processed by the main CPU core without parallel constructs. This version serves as the baseline for performance comparison.

4.2 OpenMP Version

OpenMP (Open Multi-Processing) is a shared-memory parallel programming model that uses compiler directives to create parallel regions[2]. In our OpenMP implementation of bitonic sort, we insert `#pragma omp parallel for` directives around the loops that perform the compare-and-swap operations in each stage of the bitonic merge. The array is shared among threads, and each thread works on different parts of the bitonic sequence. The code structure typically involves parallelizing the outer loop of the merge phase while ensuring that merges at each stage occur in parallel. The OpenMP version is compiled with an OpenMP-capable compiler and the number of threads can be varied at runtime.

4.3 MPI Version

MPI (Message Passing Interface) is a standardized and portable system for message-passing parallel programming on distributed-memory systems[3]. In the MPI implementation, the input array is partitioned among MPI processes. Each process performs local bitonic sort on its subset. Then, processes cooperate through MPI send/receive calls (or MPI collective operations) to perform the bitonic merge across the distributed data (for example using a pairwise exchange of data between processes in the bitonic network pattern). The MPI code uses functions like `MPI_Init`, `MPI_Comm_rank`, `MPI_Sendrecv`, and `MPI_Finalize`. The overall code structure involves initializing MPI, scattering the data, performing local bitonic steps, performing global merges with message passing, and gathering the results.

4.4 CUDA Version

CUDA (Compute Unified Device Architecture) is NVIDIA's parallel computing platform and programming model for GPUs[4]. In the CUDA implementation, the bitonic sort is performed on the GPU. The host (CPU) code allocates memory on the device and copies the input. A CUDA kernel is launched to perform the compare-and-swap operations; typically, each CUDA thread handles one element comparison per merge step. The kernel is launched multiple times for each stage of the bitonic sort, with synchronization between stages. The code structure includes setting up device memory, defining grid and block dimensions (e.g., blocks of 256 or 512 threads), launching the kernel(s) for each phase of the bitonic network, and copying the sorted data back to the host. This leverages the many cores of the GPU to perform comparisons in parallel. The CUDA Toolkit provides libraries and a runtime to support high-performance GPU computation[4].

5. Results & Performance Evaluation

The dataset size used for testing was 2048 elements, with the input read from `InputFiles/input.txt`. Table 1 summarizes the execution times measured (in seconds) for each implementation. The serial version took **0.001002 s**. For OpenMP, we tested with 1, 2, 4, 8, and 16 threads. The MPI version was tested with 1, 2, 4, 8, and 16 processes. The CUDA version was tested with thread-block sizes of 128, 256, 512, and 1024 threads per block.

Method	Configuration	Time (s)
Serial	-	0.001002
OpenMP	1 thread	0.000643
OpenMP	2 threads	0.000808
OpenMP	4 threads	0.000713
OpenMP	8 threads	0.001915
OpenMP	16 threads	0.002535
MPI	1 process	0.000392
MPI	2 processes	0.000122
MPI	4 processes	0.000086
MPI	8 processes	0.000068
MPI	16 processes	0.000123
CUDA	128 threads/block	0.000827
CUDA	256 threads/block	0.000811
CUDA	512 threads/block	0.000822
CUDA	1024 threads/block	0.000840

Table 1: Execution times for Bitonic Sort implementations (dataset size = 2048).

6. Speedup Analysis

To evaluate parallel efficiency, we compute speedup relative to the serial baseline (0.001002 s). Speedup is defined as the ratio of serial execution time to parallel execution time[5]. Table 2 lists the best configuration for each method, its execution time, and the resulting speedup:

Method	Best Config	Time (s)	Speedup
OpenMP	1 thread	0.000643	1.56×
MPI	8 processes	0.000068	14.74×
CUDA	256 threads	0.000811	1.24×

Table 2: Speedup of best parallel configuration for each method (relative to serial).

The speedup $S = T_{serial} / T_{parallel}$ ranges from about 1.24× (CUDA) to 14.74× (MPI). Note that for OpenMP, using only 1 thread (no parallelism) gave the shortest time among OpenMP tests, likely because additional threads introduced overhead for this problem size. For MPI, 8 processes achieved the best time (0.000068 s). For CUDA, 256 threads per block was slightly faster than other configurations.

8. Screenshots Section

Note: Screenshots of the program outputs and execution (e.g., terminal outputs and timing outputs) are provided separately in the `screenshots` folder accompanying this report. These images are not embedded in the document but have been submitted along with this report.

9. References

- Batcher, K. (1968). *Sorting networks and their applications*. (Foundational work on bitonic sort.)
- Wikipedia contributors. "Bitonic sorter." *Wikipedia, The Free Encyclopedia*. (Explains bitonic sort network and parallel properties)[1].
- MPI Forum – *Message Passing Interface (MPI) Standard*. (Official MPI documentation)[7].
- San Francisco State University. "Message Passing Interface (MPI)." *Academic Technology Help Center*[3].
- OpenMP Architecture Review Board. *OpenMP API Specification*. (Official OpenMP documentation)[2].
- NVIDIA. *CUDA Toolkit Documentation*. (Official CUDA programming guide)[4].
- CVW – Cornell Virtual Workshop. "Measuring Efficiency." (Defines speedup and parallel efficiency)[5][6].

[1] Bitonic sorter - Wikipedia

https://en.wikipedia.org/wiki/Bitonic_sorter

[2] 10.3.2 OpenMP Parallelization Directives (Sun Studio 12: Fortran Programming Guide)

https://docs.oracle.com/cd/E19205-01/819-5262/aeujl/index.html

[3] Message Passing Interface (MPI) – Academic Technology Help Center

https://athelp.sfsu.edu/hc/en-us/articles/31707324162195-Message-Passing-Interface-MPI

[4] CUDA Toolkit Documentation 13.1

https://docs.nvidia.com/cuda/

[5] [6] Cornell Virtual Workshop > Parallel Programming Concepts and High Performance Computing > Efficiency > Measuring Efficiency

https://cvw.cac.cornell.edu/parallel/efficiency/about-efficiency

[7] MPI Forum

https://www.mpi-forum.org/