Parallelization of Pigeonhole Sort for Efficient Data Sorting

Pasupuleti Rohith Sai Datta [1], Chinmaya D Kamath[2], N Gopalakrishna Kini[3]and Ashwath Rao B[4]
[1-4]Department of Computer Science and Engineering, Manipal Institute of Technology
Manipal Academy of Higher Education, Manipal-576 104, Karnataka, India
[3] ng.kini@manipal.edu
[1, 2, 4]{pasupueltirohithsaidatta03, ckamath2000, ashwath.rao.b}@gmail.com

***Abstract*—The need for parallel sorting algorithms have been driven by the increasing need for large-scale datasets to be processed efficiently. Pigeonhole sorting is one of the sorting algorithms that carries sorting in linear time. This study focuses on enhancing the efficacy of the Pigeonhole Sorting method to improve the performance of the algorithm by employing parallel programming techniques specifically Message Passing Interface (MPI) and Compute Unified Device Architecture (CUDA). The primary objective is to develop and assess parallel solutions for Pigeonhole Sorting, with the aim of optimizing sorting efficiency in data-intensive applications. Commencing with a comprehensive analysis of the sequential design of the Pigeonhole Sorting algorithm, this work proceeds to create parallel implementations using CUDA for Graphics Processing Unit (GPU) acceleration and MPI for distributed memory parallelism. This work contributes valuable insights into adapting the Pigeonhole Sorting algorithm to parallel contexts. The findings emphasize the potential advantages of parallelization in reducing the overall computation time.**

***Index Terms*— Pigeonhole sorting, Parallel programming, Message passing interface, Compute unified device architecture, Graphics processing unit, Speedup**

I. INTRODUCTION

The rapid growth of data-intensive applications and the escalating volumes of information processed in contemporary computing environments have necessitated the development of efficient parallel algorithms. Parallel programming, a paradigm wherein multiple computational tasks are executed simultaneously, offers a promising solution to meet the escalating demands for enhanced computational speed and efficiency. In the realm of sorting algorithms, the focus of this research is on Pigeonhole Sorting, a technique known for its simplicity and effectiveness in sequentially organizing data.
Growing datasets challenge traditional sequential sorting algorithms, necessitating parallel programming to improve efficiency. This section emphasizes the fundamental principles of parallel programming to address modern computational demands. Parallel sorting algorithms are crucial when data scale surpasses traditional sequential capabilities. Pigeonhole Sorting, despite its sequential nature, presents an opportunity for efficient parallel sorting.
The demand for efficient parallel algorithms is driven by data-intensive applications in various domains. This research contributes by investigating MPI and CUDA integration into Pigeonhole Sorting, enhancing its applicability to contemporary computing challenges. Data-intensive challenges extend beyond computational speed, requiring parallel algorithms to address complexities in storage, communication, and energy consumption.
Originally designed for sequential execution, Pigeonhole Sorting is a compelling candidate for parallelization due to its simplicity. This research explores harnessing its parallelization potential to meet demands for efficient sorting in contemporary computing. Advances in parallel architectures, like multi-core processors

and GPUs, enhance parallel algorithm efficiency. This section explores key features aligning with contemporary parallel computing requirements. Integration of MPI and CUDA into Pigeonhole Sorting is motivated by their strengths. MPI facilitates communication in distributed memory systems, while CUDA leverages GPU parallel processing, enhancing sorting efficiency. This section outlines their rationale and integration into the research framework.

The main contribution of this paper is:

- To present how a novel MPI and CUDA based parallel approach is investigated to Pigeonhole sorting algorithm to improve overall computational performance.
- To analyze the speedup performance of parallel algorithms with classical implementation on single CPU-core.

II. RELATED WORK

Sorting algorithms play a crucial role in data processing, significantly influencing diverse fields ranging from database management to scientific computing. As the volume of data continues to grow exponentially, there is an increasing demand for sorting algorithms capable of efficiently handling large datasets without compromising performance. The relevance of parallelized sorting algorithms in the context of data processing has been extensively explored in a multitude of studies, each contributing unique insights and methodologies to enhance the efficiency of sorting techniques within parallel computing environments.

A comprehensive A comprehensive survey of GPU-based sorting algorithms, categorizing various approaches and highlighting the performance improvements over time is addressed in [1]. Their foundational work emphasizes the importance of adapting sorting algorithms to meet the challenges posed by increasingly large datasets. The survey covers notable algorithms like Bitonic Sort and Radix Sort, emphasizing their efficiency in leveraging GPU parallelism to achieve significant speedups compared to CPU-based implementations.

In [2], authors have focused on the GPU-based parallelization of topological sorting. Their research demonstrates how GPUs can be utilized to parallelize sorting processes, reducing computational time. A radix sorting parallel algorithm tailored for GPU computing is dealt in [3]. Here, authors study the details the algorithm's scalability and efficiency in handling large datasets, discussing implementation challenges and optimization techniques like memory management and parallel workload distribution. These insights are crucial for developing an efficient parallelized pigeonhole sorting algorithm, demonstrating the significant performance gains possible with GPU parallelism.

In [4], authors have compared various parallel sorting algorithms, providing performance analysis across different hardware configurations. The study highlights the importance of selecting appropriate algorithms based on data characteristics and computing environments, which is pertinent for optimizing pigeonhole sorting algorithms for GPU platforms. Their work underscores the necessity of ongoing research to enhance sorting algorithm efficiency, particularly for complex data.

Performance analysis of several parallel sorting algorithms using GPU computing are conducted in the literature [5]. Their findings indicate significant performance gains for traditional sorting algorithms when parallelized, though the extent of these gains varies based on the algorithm's inherent parallelism and the GPU architecture. This benchmark is valuable for evaluating the performance of new parallel sorting algorithms, including pigeonhole sorting. In [6], authors have explored the Pigeonhole Principles, providing a detailed explanation of its theoretical foundations. Understanding these fundamentals is essential for effectively applying the principle to sorting algorithms. The theoretical insights from this study will inform the development of a parallelized pigeonhole sorting algorithm.

The effect of cardinality in the Pigeonhole Principle, providing insights into optimizing algorithms based on data distribution is addressed in [7]. The work in this paper suggests that the principle's effectiveness in sorting is influenced by the distribution of data elements, a critical consideration for developing a parallelized pigeonhole sorting algorithm. The efficiency and performance of sorting and searching algorithms for array elements are examined in [8]. The study highlights recent advancements and emphasizes the importance of optimization for specific data characteristics. This recent perspective underscores the necessity of ongoing research to enhance sorting algorithm efficiency, particularly for complex data.

In [9] and [10], it is discussed on how to parallelize the sorting algorithms using MPI and CUDA platforms. The time taken by the sequential sort is compared with the with MPI and CUDA platform implementation.

Speedup of these sorting algorithms achieved using MPI and CUDA implementation have showcased the enhancing sorting efficiency.

In conclusion, the reviewed literature reveals significant advancements in GPU-based sorting algorithms and the theoretical applications of the Pigeonhole Principle. However, there remains a gap in applying the Pigeonhole Principle to parallelized sorting algorithms optimized for GPU architectures. The proposed research on parallelized pigeonhole sorting aims to address this gap by leveraging GPU parallelism and the organizational advantages of the Pigeonhole Principle, potentially leading to substantial performance improvements for sorting large datasets.

The main contribution of this paper is:

- To present how a novel MPI and CUDA based parallel approach is investigated to Pigeonhole sorting algorithm to improve overall computational performance.
- To analyze the speedup performance of parallel algorithms with classical implementation on single CPU-core.

III. METHODOLOGY

In the context of this work, the sequential Pigeonhole Sorting algorithm serves as the foundational sorting method. It operates by iteratively distributing elements into their respective Pigeonholes based on their values. The sequential nature ensures a step-by-step organization of the entire dataset, making it conducive to parallelization for improved efficiency.

The MPI implementation introduces a distributed memory approach to enhance the scalability of the Pigeonhole Sorting algorithm. This involves partitioning the dataset across multiple nodes, with each node independently performing sorting on its designated portion. The MPI framework facilitates communication and coordination among nodes, enabling a collaborative effort to achieve a faster overall sorting process.

Conversely, the CUDA implementation targets the acceleration of sorting tasks through parallel processing on GPUs. By harnessing the computational power of GPUs, the algorithm can simultaneously process multiple elements, significantly reducing the sorting time. This methodology capitalizes on the parallel architecture of GPUs, allowing for a substantial boost in sorting performance compared to traditional sequential implementations.

Throughout the methodology, careful consideration is given to optimizing load balancing, data distribution, and communication overhead in both the MPI and CUDA implementations. The goal is to harness the full potential of parallel processing while minimizing bottlenecks, ensuring a comprehensive and effective approach to sorting large datasets in distributed and GPU-accelerated environments.

A. Working of Pigeonhole Sorting

Pigeonhole sorting is a straightforward sorting method that targets particular traits in the data distribution. The concept of categorizing stuff into pigeonholes according to specific attributes is where this algorithm gets its name. The basic idea is to assign items to distinct "pigeonholes" or buckets, each of which stands for a certain value range. The algorithm creates a sorted arrangement by sorting the elements inside these pigeonholes.

Pigeonhole sorting is particularly effective when dealing with a limited range of values within the dataset. The number of pigeonholes correspond to the range of values present in the data, and each pigeonhole represents a unique value or a specific range of values. This sorting technique is advantageous for datasets with a relatively small range compared to the number of elements, as it allows for a more efficient arrangement of items.

One notable characteristic of pigeonhole sorting is its linear time complexity in the best-case scenario. When the range of values is not significantly larger than the number of elements, this algorithm can achieve a sorting time proportional to the number of elements. However, its practicality diminishes when the range becomes much larger, potentially leading to inefficient use of memory due to the creation of numerous empty pigeonholes.

Despite its simplicity and efficiency in certain scenarios, pigeonhole sorting may not be the optimal choice for datasets with a wide range of values. The algorithm's performance heavily relies on the distribution of data and may exhibit suboptimal results when faced with skewed distributions. In such cases, other sorting algorithms with adaptive strategies may be more suitable for achieving better overall performance.

Pigeonholes are assigned elements from the input array according to a mapping function. Depending on the type of data, this function could be as simple as the element's value or it could be a more complicated criterion. The pigeonhole that corresponds to each element's mapped value is used. A localized grouping is formed when elements with comparable or identical values are placed to the same slot.
The sorted array is produced by the algorithm concatenating the elements from each pigeonhole after the first assignment. Concatenating these groups yields a globally sorted sequence as each pigeonhole's elements are already arranged in order.
Pigeonhole Sorting is very useful when the dataset has an unequal element distribution and a narrow range of values. It performs best in situations where some values or ranges are more common than others.

B. Sequential implementation of Pigeonhole Sorting Algorithm

The algorithmic step-by-step process of sorting an array using sequential Pigeonhole Sorting algorithm is presented in Figure 1. The illustration of the Pigeonhole Sort process can be seen in Figure 2.

```
----------------------------------------------------------------------------------------------------

Algorithm: The sequential Pigeonhole Sorting
        Step 1:  Create Empty Pigeonholes:
                 Initialize an array of empty pigeonholes.
                 Return this array.
        Step 2:  Place Elements in Pigeonholes:
                 For each element in the input array:
                 Determine the index of the pigeonhole for the element.
                 Append the element to the corresponding pigeonhole.
        Step 3:  Concatenate Pigeonholes:
                 Concatenate the elements from all pigeonholes into a single array.
                 Return the concatenated array.
        Step 4:  Calculate Pigeonhole Index:
                 Calculate the index of the pigeonhole for a given element based on specific characteristics of the data.
                 The calculation can be based on the element value or other criteria.
                 Return the calculated index.
        Step 5:  Pigeonhole Sort Function:
                 Perform the steps in the following sequence:
                 Create empty pigeonholes.
                 Distribute elements into pigeonholes.
                 Concatenate elements from all pigeonholes.
                 Return the sorted array.
----------------------------------------------------------------------------------------------------
```

Figure 1. Pigeonhole Sorting algorithm

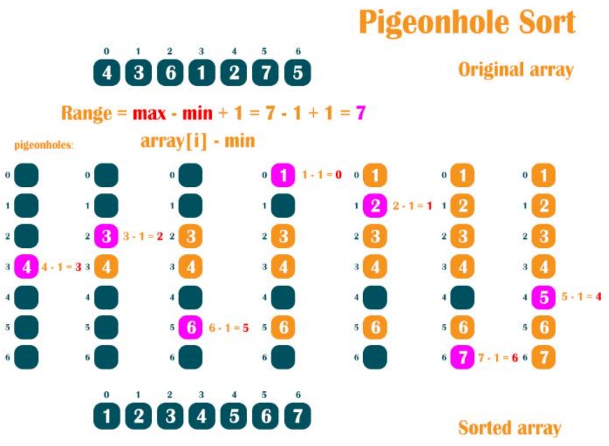

Figure 2. Illustration of Pigeonhole Sort process

A function initializes an array to serve as pigeonholes. Each pigeonhole is represented as an empty list. Another function determines the index of the pigeonhole for each element. It then appends the element to the corresponding pigeonhole. A function combines the elements from all pigeonholes into a single array. This step ensures that the elements are arranged in order within their respective pigeonholes. The sorted array is returned as the final result of the pigeonhole Sort algorithm. This sequential version establishes the baseline for understanding the logic of Pigeonhole Sorting, which will be further extended and adapted for parallel implementations using MPI and CUDA.

C. MPI implementation of Pigeonhole Sorting Algorithm

A distributed memory model is used by the Pigeonhole Sorting MPI. The MPI implementation of Pigeonhole Sorting employs a distributed-memory model, enabling multiple processes to collaborate in sorting a dataset. The primary strategy involves breaking down the dataset into smaller subsets, distributing them among MPI processes, independently sorting these subsets, and finally merging the sorted subsets to obtain the globally sorted array. MPI implementation, which allows several processes to work together to sort a dataset. First, the dataset is divided into smaller subsets, which are then distributed over MPI processes. Each of these subsets is then separately sorted. Lastly, the sorted subsets are merged to create the globally sorted array.
The MPI implementation steps are given in Figure 3.

Algorithm: MPI implementation of Pigeonhole Sorting Algorithm

```
Step 1: Initialization of the MPI to enable parallel processing.
        Acquire the rank (rank) of the active processes and the size (number) of all MPI processes.
Step 2: Distribution of Data
        If the master process (rank 0) is the one running at the moment:
            Divide the incoming data into segments.
            Each chunk should be assigned to the appropriate process.
Step 3: Local Sorting
            On the designated subset of data, each MPI process individually carries out local sorting using the Pigeonhole Sort
            algorithm.
Step 4: Data Gathering
            Gather the locally sorted data from each process.
            Forward the locally sorted data to the master process. Get the locally sorted data from every other process if the
            running process is the master process (rank 0).
Step 5: Complete Fusion
        If the master process (rank 0) is the one running at the moment:
            Combine the subgroups that were sorted locally and globally to create the final array. The arrays that have been
            sorted by each process are combined by the master process.
```

Figure 3. MPI implementation of Pigeonhole Sorting

D. CUDA Implementation of Pigeonhole Sorting Algorithm

CUDA is a parallel computing platform and a framework created by NVIDIA. It allows developers to use NVIDIA GPUs for general-purpose processing like Pigeon-hole Sorting. In the context of Pigeonhole Sorting with CUDA, the CUDA implementation steps are given in Figure 4.

Algorithm: CUDA implementation of Pigeonhole Sorting Algorithm

```
Step 1: Memory Allocation:
            Allocate memory for the input data on the host (CPU) and the device (GPU).
        Data Transfer:
            Copy the input data from the host memory to device memory.
Step 2: Kernel Design:
            Design a CUDA kernel that performs the Pigeonhole Sorting algorithm on subset of the data.
        Launch Configuration:
            Specify the launch configuration, determining the number of blocks per grid and number of threads per block
            based on the size of the dataset.
Step 3: Kernel Implementation:
            Implement the Pigeonhole Sorting algorithm within the CUDA kernel, ensuring proper mapping of elements to
            threads and buckets.
        Shared Memory Usage:
            Utilize shared memory for communication and synchronization among threads within a block.
```

```
Step 4:   Data Transfer Back:
              Copy the sorted data from the device memory back to the host memory.
          Memory Deallocation:
              Free the allocated device memory.
-----------------------------------------------------------------------------------------------
```
Figure 4. CUDA implementation of Pigeonhole Sorting

IV. RESULTS

The results obtained when Pigeonhole Sorting is performed using sequential MPI and CUDA for 12 different iterations are shown in Table I.

TABLE I. EXECUTION TIME AND SPEEDUP OF PIGEONHOLE SORT OBTAINED FROM MPI AND CUDA

Sl. No.	Sequential Execution Time (ms)	MPI Execution Time (ms)	Speedup achieved with MPI	CUDA Execution Time (ms)	Speedup achieved with CUDA
1	17.472	14.449	1.209	0.045	388.26
2	21.752	16.521	1.316	0.09	241.68
3	17.607	16.621	1.059	0.05	352.14
4	16.441	16.508	0.995	0.08	205.51
5	16.732	14.579	1.147	0.067	249.73
6	16.991	14.322	1.186	0.055	308.92
7	18.878	17.541	1.076	0.075	251.70
8	18.221	14.781	1.232	0.048	379.60
9	21.499	13.829	1.554	0.085	252.92
10	16.515	14.355	1.150	0.062	266.37
11	19.226	13.281	1.447	0.07	274.65
12	18.134	13.006	1.394	0.052	348.73
AVG	***18.289***	***14.98***	***1.231***	***0.065***	***293.36***

The average sequential time for Pigeonhole Sort across 12 iterations is 18.289 milliseconds. When implementing the algorithm with MPI, the average time decreased to approximately 14.98 milliseconds, resulting in an MPI speedup of about 1.231. On the other hand, utilizing CUDA for parallel processing yielded a significant reduction in average time to approximately 0.065 milliseconds, achieving a substantial CUDA speedup of approximately 293.36.

V. CONCLUSIONS

The primary goal of this work was to enhance the efficacy of the Pigeonhole Sorting algorithm's performance by employing parallel programming techniques specifically MPI and CUDA. The Pigeonhole Sort algorithm demonstrates diverse time complexities in sequential, MPI, and CUDA implementations. These time complexities underscore efficiency gains in both MPI and CUDA parallel contexts.
The MPI implementation enables multiple processes to collaborate in sorting and managing distinct data subsets, leveraging the capabilities of several computing nodes. This distributed approach enhances the algorithm's ability to handle large datasets effectively.
Through GPU is used to inherit parallel processing with CUDA, it yielded several benefits for large datasets in terms of speed and efficiency. The algorithm's performance improves with increased processing power, facilitated by parallelization techniques that enable concurrent execution.

REFERENCES

[1] Singh, D.P., Joshi, I. & Choudhary, J, "Survey of GPU Based Sorting Algorithms," *Int J Parallel Prog*, vol. 46, pp. 1017–1034, 2018.
[2] Saxena, R., Jain, M., Sharma, D.P., "GPU-Based Parallelization of Topological Sorting," *Proceedings of First International Conference on Smart System, Innovations and Computing: SSIC 2017*, pp. 411-421, Springer, Singapore 2018.

[3] Xiao, Shi-yang, Cai-lin Li, Bao-yun Guo, and Han Xiao, "A radix sorting parallel algorithm suitable for graphic processing unit computing," *Concurrency and Computation: Practice and Experience*, vol. 33(6), e5818, 2021.

[4] Bozidar, Darko, and Tomaz Dobravec, "Comparison of parallel sorting algorithms," *Proceedings of the 2015 International Conference on Parallel and Distributed Systems*, vol. 33, pp. 411-421, 2015.

[5] Faujdar, Neetu, and S. P. Ghrera, "Performance Analysis of Parallel Sorting Algorithms using GPU Computing," *International Journal of Computer Applications*, vol.12(2), pp. 0975-8887, 2016.

[6] Díaz-Barrero, José Luis, "The Pigeonhole Principle," *Antoine Mhanna 23 The best constant in some geometric inequalities by Marius Dragan and Mihály Bencze 32*, p.56, 2022.

[7] Jacquet, Baptiste, and Jean Baratgin, "The effect of cardinality in the pigeonhole principle," *Thinking and Reasoning*, vol. 30(1), pp. 218-234, 2024.

[8] Yo'ldashev, Bilolkhon, and Sharobidinov Mukhriddin, "Investigating the Efficiency and Performance of Sorting and Searching Algorithms for Array Elements," *Science Promotion*, vol. 1(1), pp. 31-34, Dec. 2023.

[9] Anaghashree, Sushmita Delcy Pereira, Rao B. Ashwath, Shwetha Rai , and N. Gopalakrishna Kini, "Super Sort Algorithm Using MPI and CUDA," *Intelligent Data Engineering and Analytics, Advances in Intelligent Systems and Computing (AISC),* vol. 1177, pp. 165-170, Springer, Singapore, 2021.

[10] Harshit Yadav, Shraddha Naik, Ashwath Rao B, Shwetha Rai , and N. Gopalakrishna Kini, "Comparison of cutshort: A hybrid sorting technique using MPI and CUDA," *Evolution in Computational Intelligence, Advances in Intelligent Computing (AISC)*, vol. 1176, pp. 421-428, Springer, Singapore, 2021.