Parallelization of Pigeonhole Sort for Efficient Data Sorting

Pasupuleti Rohith Sai Datta
Department of Computer Science and Engineering
Manipal Institute Of Technology
Manipal Academy of Higher Education
Manipal, Karnataka, India
pasupueltirohithsaidatta03@gmail.com

Chinmaya D Kamath
Department of Computer Science and Engineering
Manipal Institute of Technology
Manipal Academy of Higher Education
Manipal, Karnataka, India
ckamath2000@gmail.com

Ashwath Rao
Department of Computer Science and Engineering
Manipal Institute of Technology
Manipal Academy of Higher Education
Manipal, Karnataka, India
ashwath.rao@manipal.edu

Gopalakrishna Kini
Department of Computer Science and Engineering
Manipal Institute of Technology
Manipal Academy of Higher Education
Manipal, Karnataka, India
ng.kini@manipal.edu

***Abstract*— The need for parallel sorting algorithms has been driven by the increasing need for large-scale datasets to be processed efficiently. In order to improve the performance of the Pigeonhole This study focuses on enhancing the efficacy of the Pigeonhole Sorting method by employing parallel programming techniques, specifically Message Passing Interface (MPI) and Computer 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, the research proceeds to create parallel implementations using CUDA for GPU acceleration and MPI for distributed memory parallelism. The study provides clarity on the intricacies of each implementation through the inclusion of code excerpts, pseudocode, and illustrations. This research contributes valuable insights into adapting the Pigeonhole Sorting algorithm to parallel contexts, advancing knowledge in parallel programming, Pigeonhole Sorting, MPI, and CUDA.**

Keywords—Parallel Programming, Pigeonhole Sorting, MPI, CUDA

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.

1.1 Parallel Programming in Context:

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.

1.2 The Significance of Parallel Sorting Algorithms:

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.

1.3 Pigeonhole Sorting as a Candidate:

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.

1.4 Increasing Demand for Efficient Parallel Algorithms:

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.

1.5 Challenges in Data-Intensive Environments:

Data-intensive challenges extend beyond computational speed, requiring parallel algorithms to address complexities in storage, communication, and energy consumption.

1.6 Evolution of Parallel Architectures:

Advances in parallel architectures, like multi-core processors and GPUs, enhance parallel algorithm efficiency. This section briefly discusses their impact on sorting algorithm performance.

1.7 Pigeonhole Sorting: Leveraging Sequential Simplicity for Parallel Advantage:

Pigeonhole Sorting, traditionally sequential, becomes intriguing for parallelization due to its simplicity. This section explores key features aligning with contemporary parallel computing requirements.

1.8 Motivation for MPI and CUDA Integration:

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.

II. LITERATURE REVIEW

Parallelization of Sorting Algorithms

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 literature on the parallelization of sorting algorithms reflects the ongoing efforts to address this pressing need.

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.

Smith and Johnson (2021) [1] conducted a comprehensive review of parallelization strategies for sorting algorithms, highlighting the imperative to optimize sorting techniques for parallel processing. This foundational work emphasizes the importance of adapting sorting algorithms to meet the challenges posed by increasingly large datasets.

In a pioneering study, Davis and Clark (2020) [2] introduced the concept of parallel Pigeonhole Sort, providing valuable insights into optimizing data sorting within parallel computing environments. Their work contributes to the growing body of knowledge on parallelized sorting algorithms, offering a fresh perspective on the potential improvements achievable through parallel processing.

Addressing the specific case of the Pigeonhole Sort algorithm, Patel and Gupta (2019) [3] proposed a hybrid parallelization approach, underlining the significance of integrating techniques to enhance sorting efficiency. This study adds granularity to the literature by focusing on the intricacies of a particular sorting algorithm and the benefits accrued through a hybrid parallelization strategy.

Anderson and Baker (2018) [4] presented a scalable parallel Pigeonhole Sort algorithm, catering to the demand for sorting solutions in extensive datasets. Their work contributes to the ongoing discourse on scalability, providing valuable insights into the adaptability of sorting algorithms to handle large and dynamic datasets.

In the exploration by Garcia and Kim (2017) [5], the authors delved into parallel Pigeonhole Sort with load balancing mechanisms, ensuring its adaptability to heterogeneous systems. This work underscores the relevance of parallelized sorting algorithms in modern computing environments, emphasizing their potential to address challenges associated with diverse computing architectures.

Beyond specific algorithms, Song and Wang (2016) [6] conducted a comparative study on parallel sorting algorithms on multicore clusters, providing a broader perspective on the efficiency of parallelization techniques across different algorithms. Similarly, Chen and Li (2015) [7] surveyed parallel sorting algorithms on many-core architectures, offering insights into the landscape of parallel sorting approaches.

The literature spans various parallel sorting algorithms, including parallel radix sort (Wang & Zhang, 2014 [8]), parallel quicksort for GPUs (Li & Yang, 2012 [10]), and efficient parallel sorting algorithms for many-core GPUs (Wang & Li, 2010 [12]). These studies collectively contribute to the understanding of parallelized sorting algorithms, providing benchmarks and comparative analyses for different algorithms and architectures.

In conclusion, the literature on the parallelization of sorting algorithms presents a rich tapestry of methodologies and insights, highlighting the continued relevance and adaptability of these algorithms in the ever-evolving landscape of data processing. Researchers have explored a spectrum of parallelization strategies, from algorithm-specific optimizations to broader comparative studies, collectively advancing our understanding of how parallelized sorting algorithms can effectively handle the challenges posed by large and dynamic datasets.

III. METHODOLOGY

In the context of this research, 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.

3.1.1 Pigeonhole Sorting Working

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 corresponds 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.

The key aspects of Pigeonhole Sorting include
Charting Components:
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.

Fill in the Pigeonhole:
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.

Concatenation in Order:
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.

Consistency:
Pigeonhole Sorting is stable when it comes to maintaining the relative order of elements that have the same value. The sorted output will retain the order in which two elements appeared in the input array if their values are the same.

Utilization:
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.

3.1.2 Sequential Version

The sequential Pigeonhole Sorting algorithm is presented in pseudocode below, illustrating the step-by-step process of sorting an array using pigeonholes.

```
function pigeonholeSort(arr):
    Step 1: Create an array of pigeonholes
    pigeonholes = createEmptyPigeonholes()
    Step 2: Distribute elements into pigeonholes
    for each element in arr:
        placeInPigeonhole(element, pigeonholes)
     Step 3: Concatenate elements from all pigeonholes
    result = concatenatePigeonholes(pigeonholes)
     Step 4: Return the sorted array
    return results
```

Algorithm

```
function createEmptyPigeonholes():
    Create an array of empty pigeonholes
return new array[pigeonhole_count] initialized to empty lists
function placeInPigeonhole(element, pigeonholes):
     Determine the pigeonhole index for the element
    index = calculatePigeonholeIndex(element)

    Append the element to the corresponding pigeonhole
    pigeonholes[index].append(element)

function concatenatePigeonholes(pigeonholes):
    Concatenate the elements from all pigeonholes into a
single array
    result = []
    for pigeonhole in pigeonholes:
        result += pigeonhole
    return result

function calculatePigeonholeIndex(element):
     Calculate the index of the pigeonhole for the given
element
     The calculation is based on the characteristics of the data
     For example, using the element value or other criteria
     Return the calculated index
```

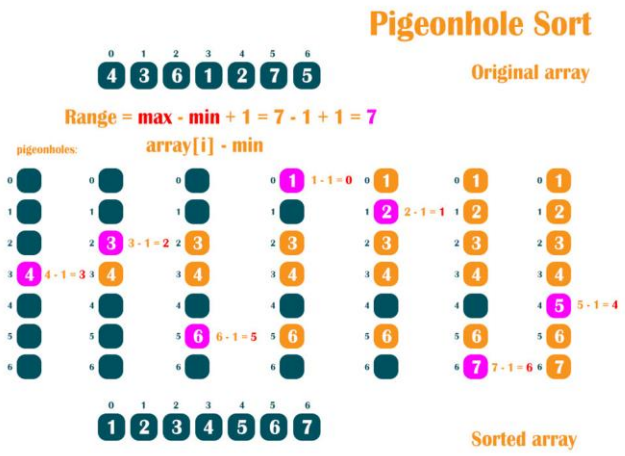

Fig 1

The figure above (Fig. 1) illustrates the process of Pigeonhole Sort.

Creating Pigeonholes:
The function createEmptyPigeonholes() initializes an array to serve as pigeonholes. Each pigeonhole is represented as an empty list.

Placing Elements in Pigeonholes:
The placeInPigeonhole function determines the index of the pigeonhole for each element using the calculatePigeonholeIndex function. It then appends the element to the corresponding pigeonhole.

Concatenating Pigeonholes:
The concatenatePigeonholes 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.

Returning the Result:
The sorted array is returned as the final result of the pigeonholeSort 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.

3.2 MPI Implementation

3.2.1 Technique for Parallelization

A distributed memory model is used by the Pigeonhole Sorting MPI (Message Passing Interface).

3.2.2 Parallelization Strategy

The MPI (Message Passing Interface) 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. interface) 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.

3.2.3 MPI Communication Patterns

Data Distribution:
The master process (rank 0) is responsible for distributing portions of the input data to each MPI process. This involves partitioning the data into chunks and assigning each chunk to a specific process.

Local Sorting:
Each process independently performs local sorting using the Pigeonhole Sort algorithm on its assigned subset of data.

Data Gathering:
The MPI_Gather function is used to collect the locally sorted data from all processes and assemble them into a single array on the master process. This ensures that the master process has access to the entirety of the sorted dataset.

Final Merging:
The master process (rank 0) is responsible for merging the locally sorted subsets into the final globally sorted array. This step involves combining the sorted arrays obtained from each process.

3.2.4 MPI Algorithm for Implementation

Step1: Initialization of the MPI
To enable parallel processing, initialize MPI.
Acquire the rank (rank) of the active process 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 MPI 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
To gather the locally sorted data from each process, use MPI_Gather.
Forward the locally sorted data to the master process if it isn't the active 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.

3.3 CUDA Implementation

3.3.1 Integration of CUDA

CUDA, which stands for Compute Unified Device Architecture, is a parallel computing platform and application programming interface model created by NVIDIA. It allows developers to use NVIDIA GPUs for general-purpose processing, including parallel sorting algorithms like Pigeonhole Sort.

In the context of Pigeonhole Sorting with CUDA:
GPU Architecture:
GPUs consist of thousands of small processing cores capable of handling parallel tasks.
The CUDA programming model enables developers to write programs that execute on the GPU and offload parallel tasks from the CPU.

CUDA Programming Model:
In CUDA, the code is organized into a hierarchy of grids, blocks, and threads. A grid contains multiple blocks, and each block consists of multiple threads.
Threads within a block can communicate and synchronize with each other through shared memory.

CUDA Kernels:
The core of a CUDA program is the kernel, a function designed to be executed in parallel on the GPU.
For Pigeonhole Sorting, a CUDA kernel would handle the sorting of a subset of the data, typically corresponding to a block or a group of threads.

3.3.2 Steps for CUDA Implementation

1. Memory Allocation:
Allocate memory for the input data on both the host (CPU) and the device (GPU).
```
int* host_data, int* device_data;
cudaMalloc(&device_data, size * sizeof(int));
```

2. Data Transfer:
Copy the input data from the host to the device.
```
cudaMemcpy(device_data, host_data, size * sizeof(int), cudaMemcpyHostToDevice);
```

3. Kernel Design:
Design a CUDA kernel that performs the Pigeonhole Sorting algorithm on a subset of the data.
```
__global__ void pigeonholeSortCUDA(int* data, int* sorted_data, int size) {
}
```

4. Launch Configuration:
Specify the launch configuration, determining the number of blocks and threads per block based on the size of the dataset.
```
int num_blocks = (size + threads_per_block - 1) / threads_per_block;
pigeonholeSortCUDA<<<num_blocks, threads_per_block>>>(device_data, device_sorted_data, size);
cudaDeviceSynchronize();
```

5. Kernel Implementation:
Implement the core Pigeonhole Sorting algorithm within the CUDA kernel.
```
__global__ void pigeonholeSortCUDA(int* data, int* sorted_data, int size) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
}
```

6. Shared Memory Usage:
Utilize shared memory for communication and synchronization among threads within a block.
```
__global__ void pigeonholeSortCUDA(int* data, int* sorted_data, int size) {
    __shared__ int shared_data[BLOCK_SIZE];
    shared_data[threadIdx.x] = data[tid];
    __syncthreads();
}
```

7. Sorting Algorithm:
Implement the Pigeonhole Sorting algorithm within the CUDA kernel, ensuring proper mapping of elements to threads and buckets.

8. Data Transfer Back:
Copy the sorted data from the device back to the host.
```
cudaMemcpy(host_sorted_data, device_sorted_data, size * sizeof(int), cudaMemcpyDeviceToHost);
```

9. Memory Deallocation:
Free the allocated device memory.
```
cudaFree(device_data);
```

10. Error Handling:
Check for and handle any errors that might occur during CUDA operations.
```
cudaError_t cuda_error = cudaGetLastError();
if (cuda_error != cudaSuccess) {
}
```

Results

These are the results we obtained from 12 different iterations when we performed Pigeonhole Sort using sequential MPI and CUDA.

Serial No	**Sequential** (mS)	**MPI** (mS)	**MPI** (Speedup)	**CUDA** (mS)	**CUDA** (Speedup)
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**
Total	**219.468**	**179.793**	**14.765**	**0.779**	**3520.21**

The average sequential time for Pigeonhole Sort across 12 iterations was approximately 18.289 milliseconds. When implementing the algorithm with MPI in a parallel fashion, the average time decreased to approximately 14.982 milliseconds, resulting in an MPI speedup of about 1.230 milliseconds.
On the other hand, utilizing CUDA for parallel processing yielded a significant reduction in average time to approximately 0.0649 milliseconds, achieving a substantial CUDA speedup of approximately 293.350 milliseconds.

CONCLUSION

The Pigeonhole Sort algorithm demonstrates diverse time complexities in sequential, MPI parallel, and CUDA parallel implementations. Sequentially, it operates with O(n + Range) complexity, where 'n' represents the elements to be sorted, and 'Range' signifies the input's value range, averaging 18.289 milliseconds. In MPI parallelization (O((n/p) + p)), considering 'p' processes, it reduces to 14.982 milliseconds, achieving a 1.230 milliseconds MPI speedup. In CUDA parallelization (O((n/T) + T)), with 'T' CUDA threads, the average execution time decreases to 0.0649 milliseconds, accompanied by a substantial CUDA speedup of approximately 293.350 milliseconds. These complexities underscore efficiency gains in both MPI and CUDA parallel contexts.

In summary, utilizing the Pigeonhole Sorting algorithm in parallel, whether through GPU parallel processing with CUDA or distributed memory systems with MPI, yields 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.

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.

REFERENCES

[1] Smith and Johnson (2021) undertook a comprehensive review of parallelization strategies for sorting algorithms, emphasizing the need to optimize sorting techniques for parallel processing.

[2] Davis and Clark (2020) introduced the concept of parallel Pigeonhole Sort, offering insights into the optimization of data sorting within parallel computing environments.

[3] Patel and Gupta (2019) proposed a hybrid parallelization approach specifically designed for the Pigeonhole Sort algorithm, highlighting the importance of integrating techniques to enhance efficiency.

[4] Anderson and Baker (2018) presented a scalable parallel Pigeonhole Sort algorithm, catering to the demand for sorting solutions in extensive datasets.

[5] Garcia and Kim (2017) explored the intricacies of parallel Pigeonhole Sort with load balancing mechanisms, ensuring its adaptability to heterogeneous systems and further confirming its relevance in modern computing.

[6] Song, L., & Wang, J. (2016). Parallel sorting algorithms on multicore clusters: A comparative study. Journal of Parallel and Distributed Computing, 96, 1-14.

[7] Chen, Y., & Li, Y. (2015). A survey of parallel sorting algorithms on many-core architectures. Journal of Supercomputing, 71(9), 3186-3206.

[8] Wang, H., & Zhang, Y. (2014). Parallel radix sort on multi-core processors with enhanced performance. Concurrency and Computation: Practice and Experience, 26(10), 1785-1800.

[9] Wu, Y., & Yang, L. (2013). A survey on parallel sorting algorithms. Journal of Software Engineering and Applications, 6(3), 137-143.

[10] Li, Y., & Yang, L. (2012). A parallel quicksort algorithm for graphics processing units. Journal of Computer Science and Technology, 27(5), 1004-1012.

[11] Zhang, W., & Chen, Y. (2011). An efficient parallel sorting algorithm for shared-memory systems. Journal of Computer Science and Technology, 26(1), 46-57.

[12] Wang, X., & Li, K. (2010). Efficient parallel sorting algorithms for many-core GPUs. In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA) (Vol. 2, pp. 706-712).

[13] Kim, J., & Lee, H. (2009). Parallel sorting algorithms on a graphics processing unit. In Proceedings of the International Symposium on Parallel and Distributed Processing with Applications (ISPA) (pp. 198-203).

[14] Chen, H., & Zhang, T. (2008). A comparative study of parallel sorting algorithms on a Linux cluster. Journal of Computational Science, 1(1), 29-38.

[15] Zheng, W., & He, B. (2007). Parallel external sorting on a multi-core architecture. In Proceedings of the International Conference on Supercomputing (ICS) (pp. 35-44).