King Saud University

Computer Science Department

Parallel Solution for Radix Sort

CSC 453

Meshal Alanazi (443101341)
Mishari Albuhairi (443102188)

Instructor:
Sofien Ezzine GANNOUNI

First Semester 1446
Fall 2024

Contents

List of Figures

Listings

1 Introduction

Radix Sort is a non-comparative efficient sorting algorithm that sorts numbers by processing individual digits. It works by sorting the input numbers based on each digit, starting from either the least significant digit (LSD) or the most significant digit (MSD). This algorithm is particularly useful for sorting large sets of numbers where the range of values is significantly larger than the number of elements. Radix Sort is often used in scenarios where stability is required, and it is efficient when the number of digits in the data is not excessively large.

Radix Sort is widely used in applications that require stable sorting, such as sorting strings, integers, or floating-point numbers. It is also used in areas like digital signal processing and computer graphics where efficient sorting is crucial.

2 Explanation of Radix Sort Algorithm

Radix Sort is a digit-based sorting algorithm that works by sorting numbers one digit at a time. The sorting can be performed starting from the Least Significant Digit (LSD) or the Most Significant Digit (MSD). Radix Sort uses a stable subroutine, such as Counting Sort, to sort the elements based on each digit.

2.1 Least Significant Digit (LSD) Approach

In the LSD approach, the sorting process starts from the rightmost digit and moves to the leftmost digit. This means that numbers are first sorted based on the least significant digit, then by the next significant digit, and so on, until all digits have been considered. LSD sorts are generally stable sorts.

2.2 Most Significant Digit (MSD) Approach

In the MSD approach, the sorting process starts from the leftmost digit and moves to the rightmost digit. This means that numbers are first sorted based on the most significant digit, then by the next digit, and so on. MSD sorts are suitable for sorting strings or fixed-length integer representations.

3 Counting Sort and Stability in Radix Sort

Radix Sort is composed of Counting Sort as a subroutine, which is used to sort the elements based on individual digits in a stable manner. Counting Sort is a non-comparative algorithm, and its stability helps maintain the relative order of elements with the same value.

Aspect	**Radix Sort**	**Counting Sort**
Use Case	Digit-wise sorting	Counts occurrences to sort elements
Stability	Stable due to Counting Sort	Stable
Complexity	$O(n \cdot d)$	$O(n + k)$
Approach	Sorts by processing each digit	Uses counting and accumulation

Table 1: Comparison between Radix Sort and Counting Sort

4 Complexity Analysis

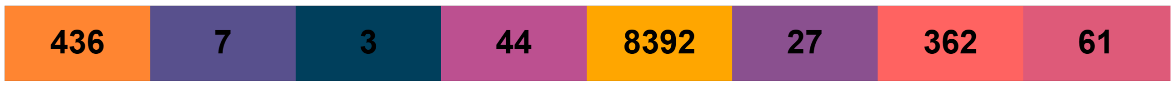
The time complexity of Radix Sort is $O(n \cdot d)$, where n is the number of elements and d is the number of digits in the largest number. The space complexity is $O(n + k)$, where k is the range of digit values (typically 10 for decimal representation).

Aspect	**Complexity**
Time Complexity	$O(n \cdot d)$, n: number of elements, d: number of digits in the largest number.
Space Complexity	$O(n + b)$, b: where b is the base (typically 10 for decimal representation).
Stability	Stable (preserves the relative order of equal elements).
Efficiency	Performs well when d is small compared to n.

Table 2: Radix Sort: Complexities and Characteristics

5 Example of Radix Sort on an 8-Element Array

Consider the following array of 8 elements:



We will use a stable sorting technique, counting sort, as detailed in 3.
Same keys can be in a different order than the input array, Solution is detailed in 8.1.

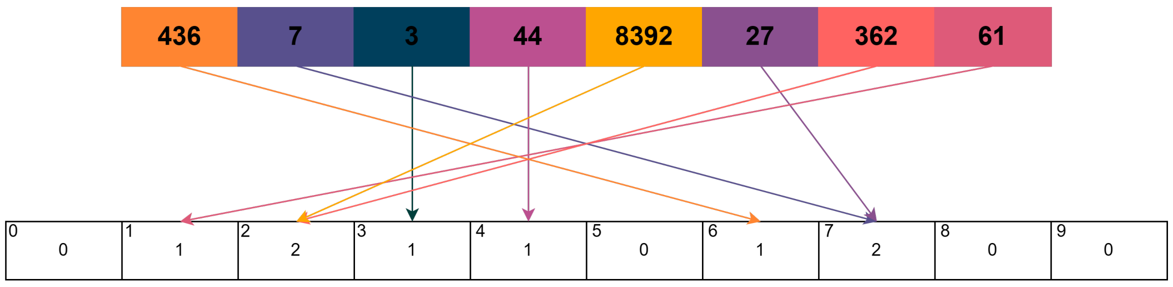
5.1 Steps to slove

5.1.1 Find the largest element in the array

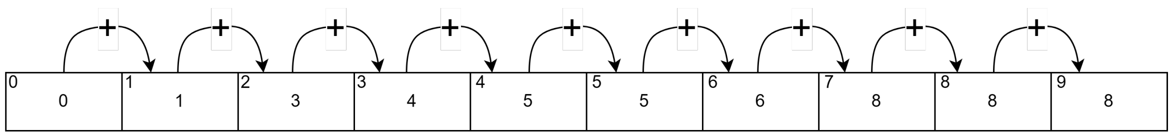
In this example, **8392** is the largest element, It has 4 digits, so we will iterate **4** times, once for each significant place.

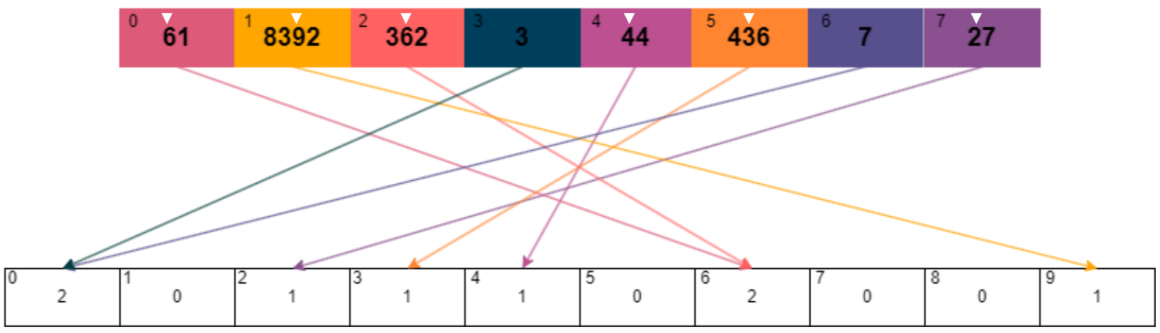
5.1.2 First Iteration: Sorting by ones place

Starting from the ones place, count the occurrence of each digit and store it in an array of size 10.



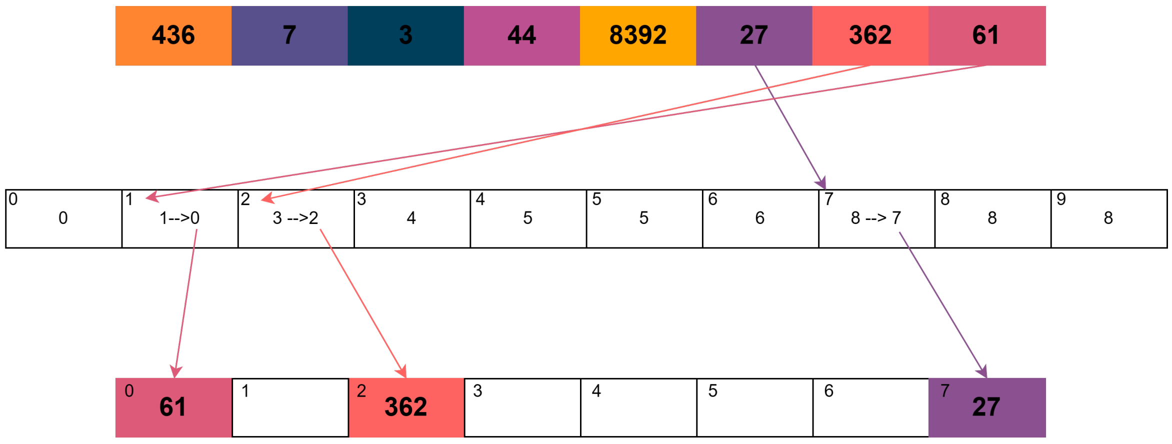
Then compute cumulative counts in this array to determine the position of each number in the sorted array.



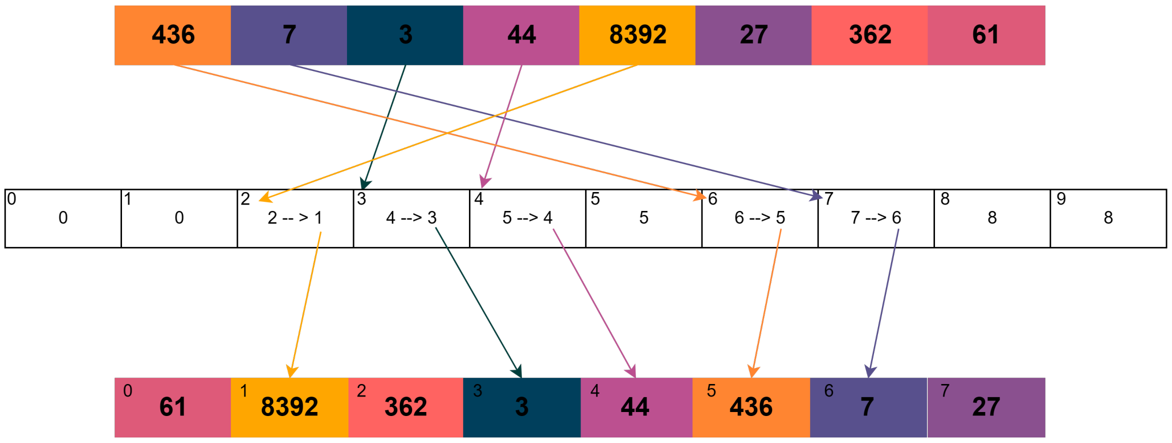


Then traverse the original array from the end, using the cumulative count array to place each number in its correct position in an auxiliary array. And decrement the value in the cumulative count array.

Part 1



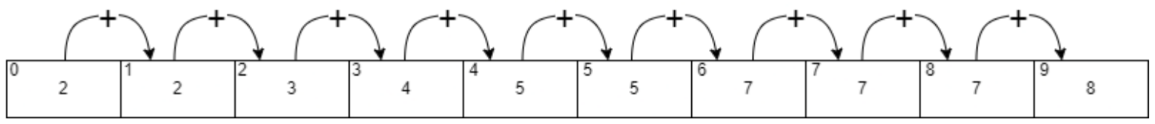
Part 2



5.1.3 Second Iteration: Sorting by Tens Digit

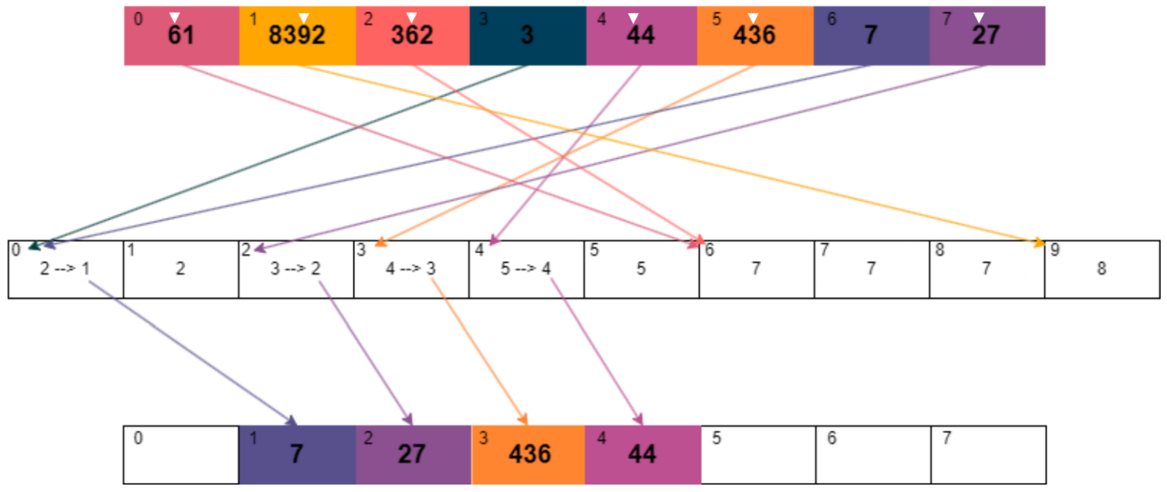
Now we do the same with the tens place now, count the occurrence of each digit and store it in an array of size 10

Then compute cumulative counts in this array to determine the position of each number in the sorted array.

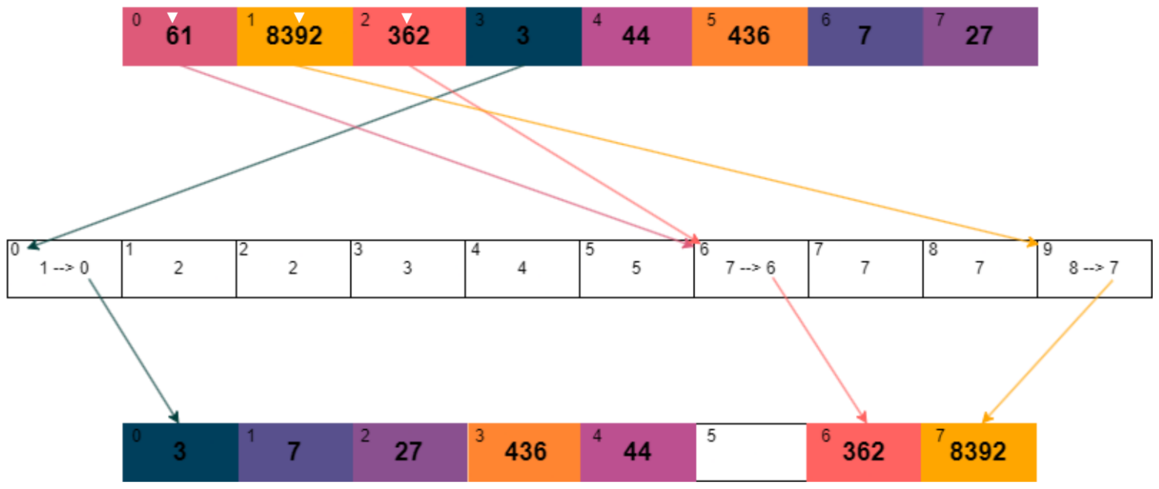


Then traverse the original array from the end, using the cumulative count array to place each number in its correct position in an auxiliary array. And decrement the value in the cumulative count array.

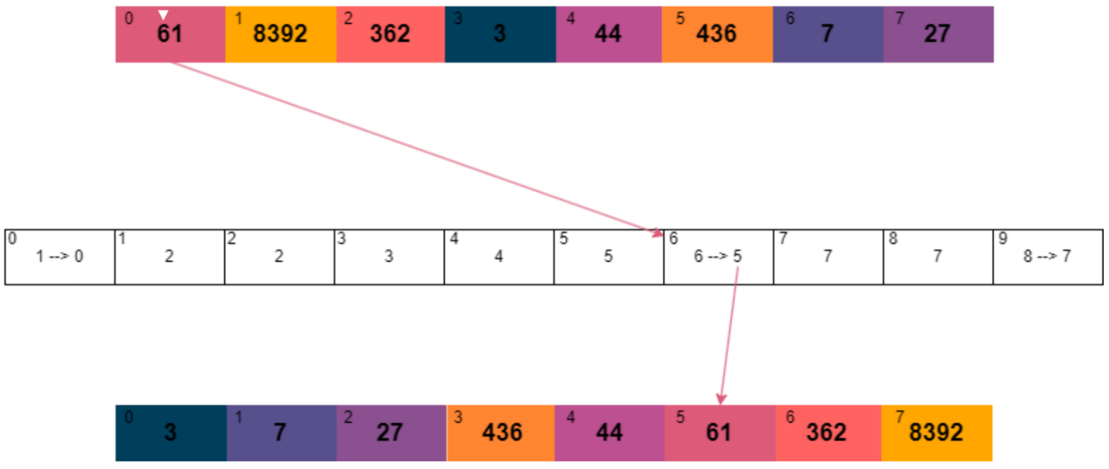
Part 1



Part 2

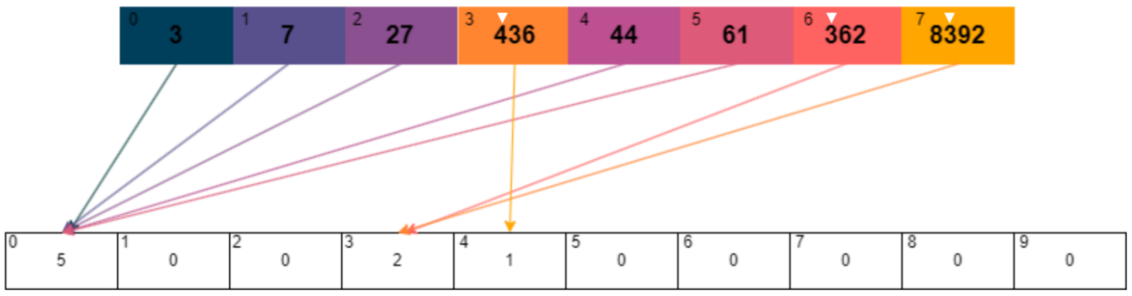


`Part 3`

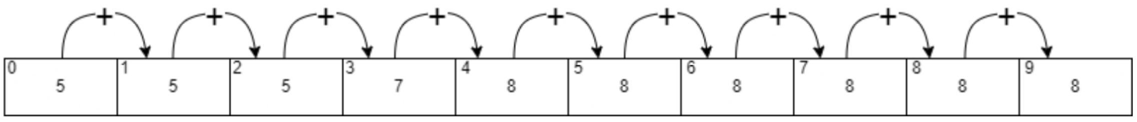


5.1.4 Third Iteration: Sorting by Hundreds Digit

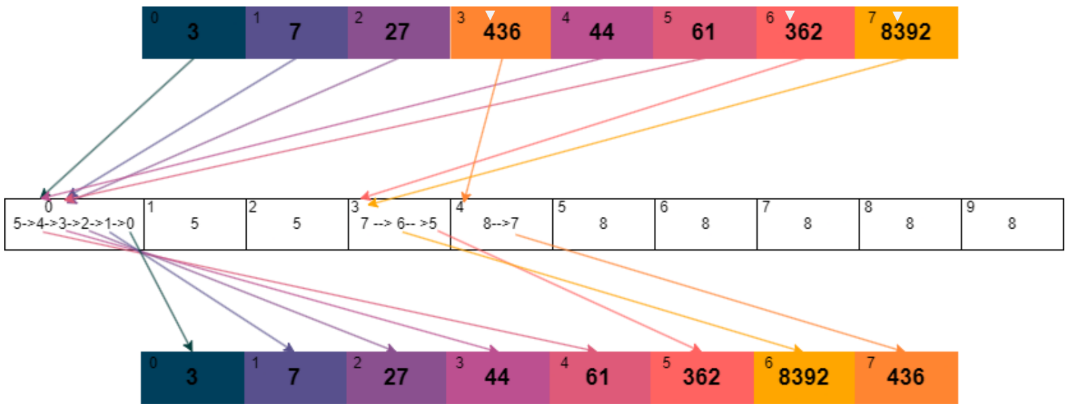
We do the same with the hundreds place now, count the occurrence of each digit and store it in an array of size 10.



Then traverse the original array from the end, using the cumulative count array to place each number in its correct position in an auxiliary array. And decrement the value in the cumulative count array.

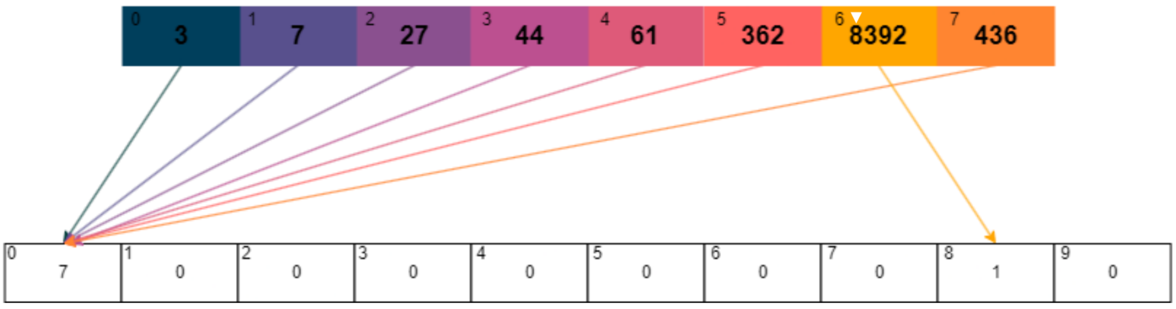


Then traverse the original array from the end, using the cumulative count array to place each number in its correct position in an auxiliary array. And decrement the value in the cumulative count array.

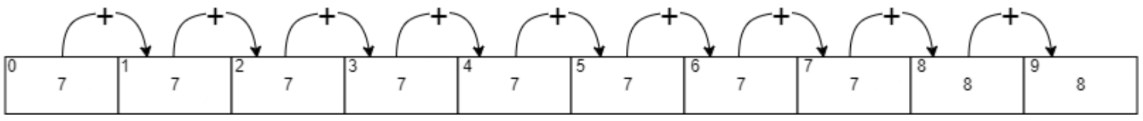


5.1.5 Fourth and last Iteration: Sorting by Thousands Digit

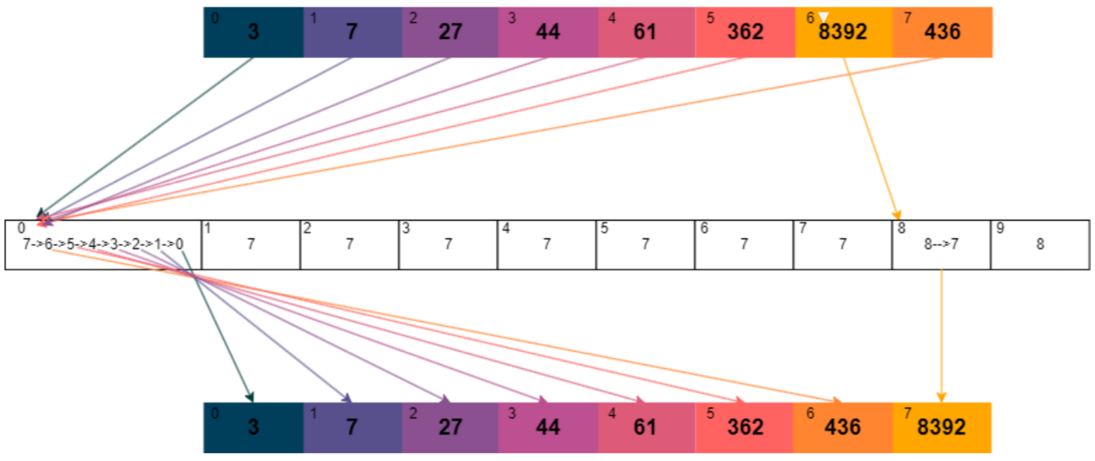
We do the same with the thousands place now, count the occurrence of each digit and store it in an array of size 10.



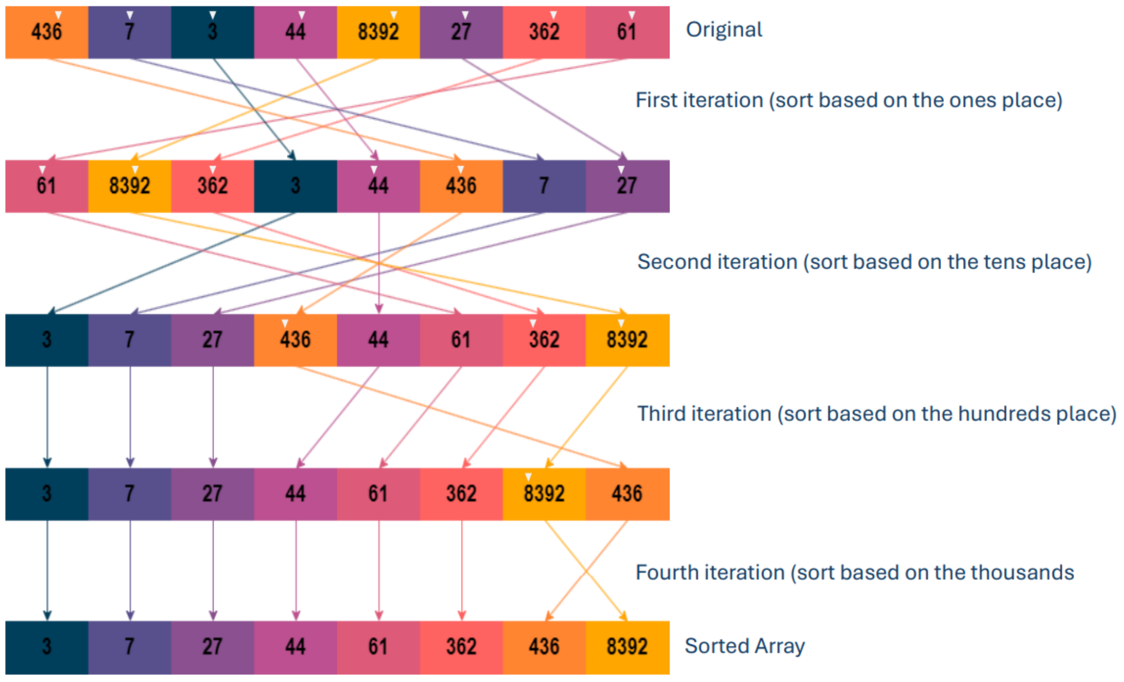
Then compute cumulative counts in this array to determine the position of each number in the sorted array



Then traverse the original array from the end, using the cumulative count array to place each number in its correct position in an auxiliary array. And decrement the value in the cumulative count array.


0 3
1 7
2 27
3 44
4 61
5 362
6 8392
7 436
0 7->6->5->4->3->2->1->0
1 7
2 7
3 7
4 7
5 7
6 7
7 7
8 8-->7
9 8
0 3
1 7
2 27
3 44
4 61
5 362
6 436
7 8392

5.1.6 Summary


436 7 3 44 8392 27 362 61
Original
First iteration (sort based on the ones place)
61 8392 362 3 44 436 7 27
Second iteration (sort based on the tens place)
3 7 27 436 44 61 362 8392
Third iteration (sort based on the hundreds place)
3 7 27 44 61 362 8392 436
Fourth iteration (sort based on the thousands
3 7 27 44 61 362 436 8392
Sorted Array

6 Sequential implementation

6.1 Code

```c
#include <stdio.h>
#define SIZE 8

// Print array
void printArr(int arr[], int n);

// Get max val in array (Step 1: Find the max number to determine the number of
    digits)
int getMax(int arr[], int n) {
    int mx = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > mx) mx = arr[i];
    return mx;
}

// Sort by digit (Step 2-4: Counting sort based on digit place value)
void countSort(int arr[], int n, int exp) {
    int out[n], cnt[10] = {0};

    // Step 1: Create an array to count occurrences of each digit
    for (int i = 0; i < n; i++) cnt[(arr[i] / exp) % 10]++;

    // Step 2: Compute cumulative counts
    for (int i = 1; i < 10; i++) cnt[i] += cnt[i - 1];

    // Step 3: Place the numbers in the correct position in the auxiliary array
    for (int i = n - 1; i >= 0; i--) out[--cnt[(arr[i] / exp) % 10]] = arr[i];

    // Step 4: Copy the sorted auxiliary array back to the original array
    for (int i = 0; i < n; i++) arr[i] = out[i];
}

// Radix Sort function
void radixSort(int arr[], int n) {
    // Step 1: Find the maximum number to determine the number of iterations
        (digits)
    for (int exp = 1, mx = getMax(arr, n); mx / exp > 0; exp *= 10) {
        countSort(arr, n, exp);
        // Print array after each digit sorting
        printf("After sorting with exp=%d: ", exp);
        printArr(arr, n);
    }
}

// Print array
void printArr(int arr[], int n) {
    printf("{");
    for (int i = 0; i < n; i++)
        printf("%s%d", i ? "," : "", arr[i]);
    printf("}\n");
}
```

Listing 1: Radix Sort Sequential implementation

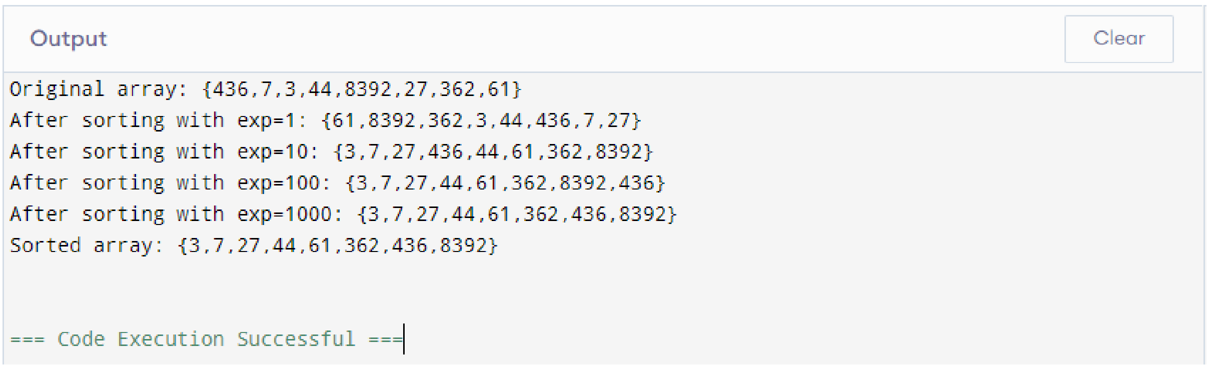
Main

```c
int main() {
    int arr[SIZE] = {436, 7, 3, 44, 8392, 27, 362, 61};
    printf("Original array: ");
    printArr(arr, SIZE);
    radixSort(arr, SIZE);
    printf("Sorted array: ");
    printArr(arr, SIZE);
    return 0;
}
```

Listing 2: Main Function Sequential implementation

6.2 Output

Screenshot of the output:



```
Original array: {436,7,3,44,8392,27,362,61}
After sorting with exp=1: {61,8392,362,3,44,436,7,27}
After sorting with exp=10: {3,7,27,436,44,61,362,8392}
After sorting with exp=100: {3,7,27,44,61,362,8392,436}
After sorting with exp=1000: {3,7,27,44,61,362,436,8392}
Sorted array: {3,7,27,44,61,362,436,8392}


=== Code Execution Successful ===
```

Figure 1: Screenshot of the Sequential Code

7 Parallel implementation

7.1 Code

```
#include <stdio.h>
#include <stdlib.h>

// #define SIZE 8
#define BASE 10


// get max element in array (host)
int getMaxCPU(int arr[], int n) {
    int mx = arr[0];
    for (int i = 1; i < n; i++) {
        if (arr[i] > mx)
            mx = arr[i];
    }
    return mx;
}

// kernel to count occurrences of each digit
__global__ void countDigitsKernel(const int *d_arr, int *d_count, int n, int exp) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) {
        int digit = (d_arr[idx] / exp) % BASE;
        atomicAdd(&d_count[digit], 1);
    }
}

// kernel to perform an inclusive prefix sum on the digit count array
__global__ void prefixSumKernel(int *d_count) {
    __shared__ int temp[BASE];
    int tid = threadIdx.x;

    temp[tid] = d_count[tid];
    __syncthreads();

    // inclusive scan
    for (int offset = 1; offset < BASE; offset <<= 1) {
        int val = 0;
        if (tid >= offset) val = temp[tid - offset];
        __syncthreads();
        temp[tid] += val;
        __syncthreads();
    }

    d_count[tid] = temp[tid];
}

// reorder inclusive kernel
__global__ void reorderKernel(const int *d_in, int *d_out, int *d_count, int n,
    int exp) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int revIdx = n - 1 - idx;
    if (revIdx < 0) return;

    int val = d_in[revIdx];
    int digit = (val / exp) % BASE;

    int pos = atomicSub(&d_count[digit], 1) - 1;
    d_out[pos] = val;
}
```

```



// Radix sort function using GPU kernels
void radixSortGPU(int arr[], int n) {
    int *d_arrIn, *d_arrOut, *d_count;
    cudaMalloc((void**)&d_arrIn, n * sizeof(int));
    cudaMalloc((void**)&d_arrOut, n * sizeof(int));
    cudaMalloc((void**)&d_count, BASE * sizeof(int));

    cudaMemcpy(d_arrIn, arr, n * sizeof(int), cudaMemcpyHostToDevice);

    int mx = getMaxCPU(arr, n);

    int threadsPerBlock = 256;
    int blocks = (n + threadsPerBlock - 1) / threadsPerBlock;

    for (int exp = 1; mx/exp > 0; exp *= 10) {

        // Step 1: count digits
        cudaMemset(d_count, 0, BASE * sizeof(int));
        countDigitsKernel<<<blocks, threadsPerBlock>>>(d_arrIn, d_count, n, exp);
        cudaDeviceSynchronize();

        // Step 2: prefix sum (inclusive)
        prefixSumKernel<<<1, BASE>>>(d_count);
        cudaDeviceSynchronize();

        // Step 3: reorder using counts (stable)
        reorderKernel<<<blocks, threadsPerBlock>>>(d_arrIn, d_arrOut, d_count, n,
            exp);
        cudaDeviceSynchronize();

        // Prepare for next digit iteration
        cudaMemcpy(d_arrIn, d_arrOut, n*sizeof(int), cudaMemcpyDeviceToDevice);
    }

    // copy final result back to host
    cudaMemcpy(arr, d_arrIn, n*sizeof(int), cudaMemcpyDeviceToHost);

    cudaFree(d_arrIn);
    cudaFree(d_arrOut);
    cudaFree(d_count);
}

// print array on host
void printArr(int arr[], int n) {
    printf("{");
    for (int i = 0; i < n; i++)
        printf("%s%d", i ? "," : "", arr[i]);
    printf("}\n");
}
```

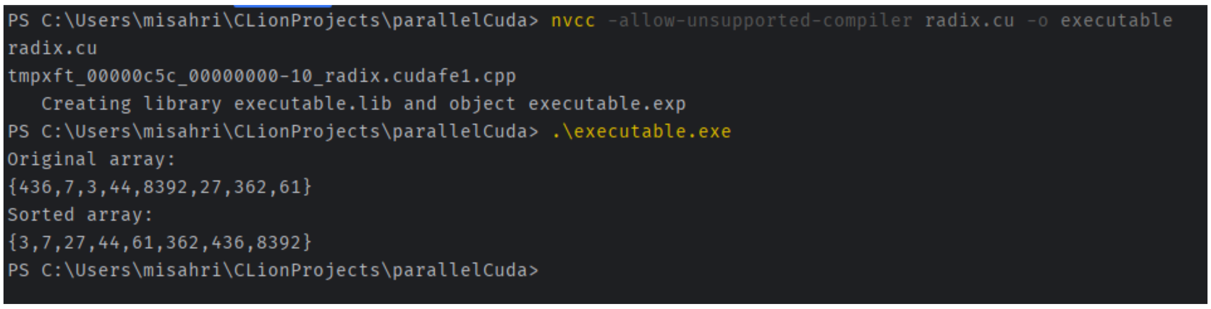
Listing 3: Radix Code Parallel implementation

Main

```c
int main() {
    int arr[] = {436, 7, 3, 44, 8392, 27, 362, 61, 88, 999999};
    int size = sizeof(arr) / sizeof(arr[0]); // Automatically calculate the array
        size

    printf("Original array:\n");
    printArr(arr, size);

    radixSortGPU(arr, size);

    printf("Sorted array:\n");
    printArr(arr, size);

    return 0;
}
```

Listing 4: Main Function Parallel implementation

7.2 Output



```
PS C:\Users\misahri\CLionProjects\parallelCuda> nvcc -allow-unsupported-compiler radix.cu -o executable
radix.cu
tmpxft_00000c5c_00000000-10_radix.cudafe1.cpp
   Creating library executable.lib and object executable.exp
PS C:\Users\misahri\CLionProjects\parallelCuda> .\executable.exe
Original array:
{436,7,3,44,8392,27,362,61}
Sorted array:
{3,7,27,44,61,362,436,8392}
PS C:\Users\misahri\CLionProjects\parallelCuda>
```

Figure 2: Screenshot of the Output

8 Challenges

On this algorithm we faced some challenges that will be detailed in this section.

8.1 Maintaining stability

Same keys can be in a different order than the input array. To solve this problem, We can iterate the input array in reverse order to build the output array. For example, consider the following input array of objects, each with a value and an identifier:

Given the array:

$$[5_{(0)}, 2_{(1)}, 5_{(2)}, 3_{(3)}]$$

where each element is shown with its index as a subscript.

After sorting by the value:

- In a stable sort, 5_0 will appear before 5_2 because they appear in that order in the input array.
- In an unstable sort, 5_2 might appear before 5_0, Changing the original order of elements with equal keys.

Solution:
count the occurrence of each value. Then you assign the position of each element in the output array.

But if you simply go through the input array from left to right while placing elements in the output, you might end up disrupting the order of keys that are equal.

To solve this problem, the solution is to iterate through the input array in reverse order when placing the elements into the output array.

In the sequential code it is implemented in the loop directly.
In the parallel code, atomic operations `(atomicSub)` are used to prevent race conditions while implementing reverse iteration.

9 Conclusion

Radix Sort is an efficient, stable sorting algorithm suitable for large datasets with limited digit counts. It uses Counting Sort as a stable subroutine and operates with a time complexity of $O(n \cdot d)$ and a space complexity of $O(n + k)$. This project demonstrated its implementation and analyzed its complexity.