Advanced Data Management for Data Analysis - Assignment 4

Group 2
Lucas van Rooij (s1695185), Nick Radunovic (s2072724),
Srishankar Sundar (s3151298), Luiza Plaiu (s3340228)

December 6, 2021

1 Introduction

For this assignment, a comparison was made between two different implementations of a comparison-based sorting algorithm: Bubble Sort. Both programs read a sequence of n non-negative 4-byte integer numbers (in the range $[0, 2^{31} - 1]$, i.e., $[0, 2147483647]$) from a given file into an in-memory array, sort that array in memory, and then output the sorted array to the console.

2 Methods

For both programs, the Bubble Sort algorithm was implemented using C and runs at a time complexity of $O(n^2)$. The "original" implementation of bubble sort incorporates branching, whereas the "predicated" bubble sort is modified such that it isn't prone to branch mis-predictions by avoiding branching.

2.1 Original Bubble Sort

Note, that this version of the bubble sort algorithm was implemented as usual (i.e. no modification that prevents branch mispredictions). Listing 1 depicts the pseudo-code of the Bubble Sort sorting algorithm.

```
BubbleSort(Array, n)
{
    for i = 0 to n
    {
        for j = 0 to n-i-1
        {
            if Array[j] > Array[j+1]
            {
                    temp = Array[j]
                    Array[j] = Array[j+1]
                    Array[j+1] = temp
            }
        }
    }
}
```

Listing 1: Pseudo-code for Branched implementation of Bubble Sort.

Note that the implementation provided in the code differs in terms of identifiers used but the mechanism remains the same. The same holds for the following sub-section.

2.2 "Predicated" Bubble Sort

The crux of the bubble sort algorithm is the (conditional) swap of two consecutive values. Therefore, the predicated implementation aims to implement the swapping without enclosing it in an if-else construct.

The 'swap' of two variables is implemented as the difference between the two variables being added to

the smaller variable and subtracted from the larger variable. To avoid the use of the if-else construct, the addition and subtraction of the differences is multiplied with 1 or 0 based on the condition being used. If the condition evaluates to 0, then the value being added/subtracted is 0 and thus the values remain the same. The exact implementation is given below:

```
BubbleSort(Array, n)
{
    for i = 0 to n
    {
        for j = 0 to n-i-1
        {
                x = Array[j]
                y = Array[j+1]
                Array[j] = Array[j] - ((x > y) * (x - y))
                Array[j+1] = Array[j+1] + ((x > y) * (x - y))
        }
    }
}
```

Listing 2: Pseudo-code for predicated implementation of Bubble Sort.

2.3 Experimental Setup

Two different datasets were used as input for both programs: **the first ∼1M rows** of file *l_orderkey-int32.csv* provided in the third assignment (**Dataset 1**), and **∼1M rows randomly selected** from the same file but permuted (**Dataset 2**). Note that dataset 1 was already sorted, whereas dataset 2 was shuffled. Both implementations ran a total of three times on each dataset. The implementations were run multiple times so as to check/eliminate the effect of varying CPU loads during the execution.

The templates provided with the assignment was used, therefore the `make` command needs to be run first. Following this, the files **ADM-2021-A4-sort-branched-inplace** and **ADM-2021-A4-sort-predicated-inplace** can be run using `./(filename)`. The former file refers to the branched implementation and the latter file refers to the predicated implementation. The same details can be found in a README file provided as part of the submission.

3 Results

Table 1 depicts the execution times for each performed run with both Bubble Sort implementations on each of both datasets.

The runs were carried out on a system with a Ryzen 5 3600H CPU clocked at 3.3GHz and RAM of 8GB.

Table 1: Execution times for each implementation of Bubble Sort (i.e., Branched, Predicated) on datasets 1 and 2.

		Execution time			
		Branched	Averaged	Predicated	Averaged
Dataset 1 (sorted)	Run 1	4m25s	4m16s	7m54s	7m
	Run 2	4m18s		6m28s	
	Run 3	4m5s		6m39s	
Dataset 2 (shuffled)	Run 1	9m55s	9m49s	6m31s	6m31s
	Run 2	9m51s		6m31s	
	Run 3	9m42s		6m31s	

4 Discussion & Conclusion

From the results shown above, it can be seen that the **branched implementation performs far better in best-case scenarios, i.e the sorted datasets**. This is expected, since the branching is eliminating the need for the swap operations, which are expensive. This operation takes place regardless in the predicated implementation, leading to considerably higher execution time in the latter case.

This can also indicate that there are very few, if any, branch mis-predictions happening in the case of the branched implementation.

On the other hand, the **predicated implementation significantly out-performs the branched implementation on shuffled datasets**, which are arguably the more typical loads for sorting algorithms. This can be attributed to branch mis-predictions significantly affecting the execution time in case of the branched implementation, whereas there are no mis-predictions in the predicated implementation.

It can also be seen that the effect of branch mis-predictions is an approximate doubling of execution time in the branched implementation for the dataset used, thus making a strong case for the predicated implementation.