Big Data Analytics Programming: Assignment 1 Report

Fotios Kalioras

1 Blocked Matrix Rotation

1.1 Experiment Setup

For this experiment the rotmatrix.py function was appropriately modified in order to perform the block matrix rotation 100 times for each strategy. Also, big integer matrices of different sizes ($1500 \leq n \leq 10000$) were constructed using a helper python script. Different block sizes were considered with an emphasis on the ones that are a multiple of 2. Since timing is important in this part, all the experiments were conducted on the Stavelot computer during a night with no traffic.

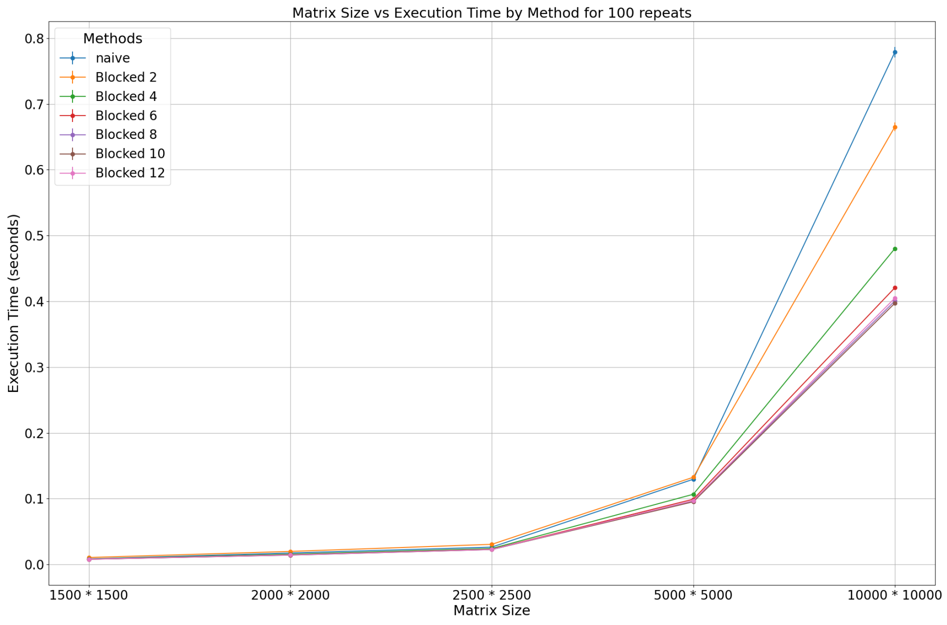

Figure 1: Execution times for different B and N

1.2 Experiment results and discussion

Figure 1 depicts the execution times for every algorithm and matrix size combination. It seems that blocks of size 10 × 10 deliver the best speedup for the selected matrices. Blocking becomes more beneficial as matrix size increases, as smaller matrices suffer from loop overhead that slows execution [1]. The ideal block size depends on the CPU's cache architecture, as varying cache sizes and structures affect performance. Tuning the block size is crucial for minimizing cache misses and maximizing efficiency, especially for larger matrices.

2 Bloom Filter

Following the assignment instructions the bloom filter functions were implemented. For the *getIndex* function special care were taken in order to adjust the bit range of the hash function output to the desired one. Specifically, the hash output was first converted to binary, stored as a temporary string and then trimmed to retain only the least significant *num_bits*, which was finally converted back to an integer. For generating multiple hashes, the Murmurhash function was used with a seed of $i * i$, where i represents the loop index.

2.1 Experiment Setup

Since timing is not important for this part of this assignment, experiments were conducted on a home PC. To evaluate theoretical and actual false positive rates (FPR) of the Bloom filter, helper functions were developed. A Python script read from ‘users.txt‘ to generate ‘n‘ fake usernames via random replacements and shuffles. The main Java function and Makefile were modified to accept ‘k‘ and ‘lognumBits‘ as arguments, with the main function adapted to process fake usernames and count false positives. Drawing inspiration from the first part, a Python driver script iterated over all ‘m‘ and ‘k‘ combinations, plotting theoretical and actual FPR curves on a single graph. The experiment used 9999 true and 9999 false usernames.

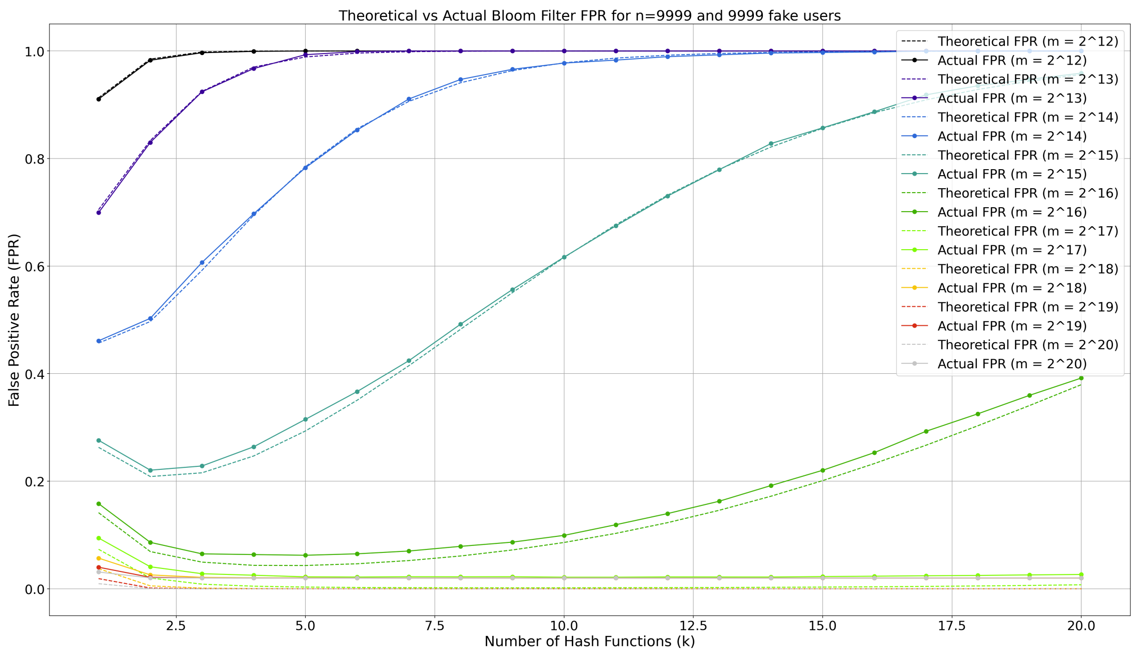

Figure 2: Theoretical and actual FPR for different number of m and k

2.2 Experiment results and discussion

Figure 2 depicts the results of the experiments.
For small m ($12 \leq m \leq 14$), the Bloom filter saturates quickly, causing the FPR to rise with additional hash functions. For $m \geq 15$, the FPR initially decreases with more hash functions but eventually rises as the bit array fills. Beyond $m \geq 18$, increasing the array size offers diminishing returns, as the FPR plateaus and memory efficiency decreases.
Overall, the results align with theory: the estimated FPR $((1-(1-\frac{1}{m})^{kn})^k)$ closely matches the actual one. The small underestimation lies in the fact that the formula assumes independence for the probabilities of each bit being set [2]. The **best k** is contingent on the formula $\frac{n}{m} * ln(2)$, which for this case equates to $\frac{9999}{m} * ln(2)$.

3 Radix Sort in Java

Following the assignment instructions the radixsort function has been implemented. Since in the lecture the LSD variant of the algorithm was presented, this algorithm was initially implemented. However, since the MSD implementation works better for variable length strings [3] it was also added as a function to be tested in tandem. For the MSD algorithm a cutoff value was also defined in order to switch to Insertion Sort when the bins become small and Radix Sort is inefficient. For both algorithms a version using a count Hashmap instead of count Array was tested but was dismissed due to poorer performance. It is also assumed that lexicographic ordering implies ASCII code ordering. Special care was taken for the fact that we are testing non-fixed length strings. More specifically, a function was created that returns either the ASCII value of the requested position of the string or -1 if the position is out of bounds. That means that an array of fixed length 129 was created, 128 for the ASCII plus 1 for the non-existent case. Also this is reflected on the code where every index derived from the ASCII value in order to use in the CountArray is shifted by 1.

3.1 Experiment Setup

In the experiment setup, the performance of custom LSD and MSD Radix Sort implementations was compared against Java's built-in sorting algorithm. A testing function was developed to evaluate the sorting algorithms on string arrays, with execution times measured using 'System.nanoTime()' for high-resolution timing. For the MSD algorithm, cutoff values of 0 and 5 were tested. Three test arrays were used: the first two were predefined arrays, while the third consisted of 1,000,000 randomly generated strings with variable lengths, where $1 \leq$ length < 26. Functions were added to repeat each sorting task 10 times and compute the average execution time for more reliable performance measurements.

3.2 Experiment results and discussion

The following table depicts the average time measured for each algorithm and each array.

Array	**LSD Radix Sort**	**MSD Radix Sort w/ cuttoff 0**	**MSD Radix Sort w/ cuttoff 5**	**Standard Java sort**
1^{st} array (banana, apple, ...)	51939 ns	11336 ns	794 ns	25401 ns
2^{nd} array (c3e, bfr, aAD, ...)	34944 ns	13231 ns	13328 ns	9237 ns
3^{nd} array (random)	4779757140 ns	245544066 ns	234659510 ns	50394849 ns

The results again seem to evaluate the theory; The LSD radix sort takes the most time since it suffers with variable length strings. This can be showcased with the second array, in which it performs better, probably due to the fact that the strings have a maximum length of 4, close to the average (3) so less padding is performed. In contrast, the MSD Radix Sort and Java's built-in sorting algorithm exhibit similar performance, with the MSD surpassing Java on 2 occasions. Surprisingly, it seems that cutoff makes little to no difference on performance, with differences almost equal to statistical error.

References

[1] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. 2nd. USA: Addison-Wesley Publishing Company, 2010. ISBN: 0136108040.

[2] Prosenjit Bose et al. "On the false-positive rate of Bloom filters". In: *Information Processing Letters* 108.4 (2008), pp. 210–213. ISSN: 0020-0190. DOI: `https://doi.org/10.1016/j.ipl.2008.05.018`. URL: `https://www.sciencedirect.com/science/article/pii/S0020019008001579`.

[3] Princeton University. *Radix Sorting*. `https://www.cs.princeton.edu/courses/archive/spr04/cos226/lectures/radix.4up.pdf`. Lecture slides, COS 226: Algorithms and Data Structures. 2004.