Computable Bitstring-compressed Matrices

Crysttian Arantes Paixão[1*], Flávio Codeço Coelho[2],
1 Crysttian Arantes Paixão Applied Mathematics School, Getulio Vargas Foundation, Rio de Janeiro, RJ, Brazil
2 Flávio Codeço Coelho Applied Mathematics School, Getulio Vargas Foundation, Rio de Janeiro, RJ, Brazil
∗ **E-mail: Corresponding crysttian.paixao@fgv.br**

Abstract

The biggest cost of computing with large matrices in any modern computer is related to memory latency and bandwidth. The average latency of modern RAM reads is 150 times greater than a clock step of the processor [1]. Throughput is a little better but still 25 times slower than the CPU can consume. The application of bitstring compression allows for larger matrices to be moved entirely to the cache memory of the computer, which has much better latency and bandwidth (average latency of L1 cache is 3 to 4 clock steps). This allows for massive performance gains as well as the ability to simulate much larger models efficiently. In this work, we propose a methodology to compress matrices in such a way that they retain their mathematical properties. Considerable compression of the data is also achieved in the process Thus allowing for the computation of much larger linear problems within the same memory constraints when compared with the traditional representation of matrices.

Author Summary

Introduction

Data compression is traditionally used to reduce storage resources usage and/or transmission costs [2]. Compression techniques can be classified into lossy and lossless. Examples of lossy data compression are MP3 (audio), JPEG (image) and MPEG (video). In this paper, we discuss the use of lossless compression for numerical data structures such as numerical arrays to achieve compression without losing the mathematical properties of the original data.

Lossless compression methods usually exploit redundancies present in the data in order to find a shorter form of describing the same information content. For example, a dictionary-based compression, only stores the positions in which a given word occurs in a document, thus saving the space required to store all its repetitions [3].

Any kind of compression incurs some computational cost. Such costs often have to be paid twice since the data needs to be decompressed to be used for its original purpose. Sometimes computational costs are irrelevant, but the need to decompress for usage, can signify that the space saved with compression must be available when data is decompressed for usage, thus partially negating the advantages of compression.

Most if not all existing lossless compression methods were developed under the following usage paradigm: *produce* → *compress* → *store* → *uncompress* → *use*. The focus of the present work is to allow a slightly different usage: *produce* → *compress* → *perform mathematical manipulations* and *decompress* (only for human reading).

With the growth of data volumes and analytical demands, creative solutions are needed to efficiently store as well as consume it on demand. This issue is present in many areas of application, ranging from business to science [4], and is being called the *Big Data* phenomenon. In the world of Big Data, the need to analyze data immediately after its coming into existence became the norm. And this analysis must take place, efficiently, within the confines of (RAM) memory. This kind of analyses are what is now known as streaming data analysis [5]. Given a sufficiently dense stream of data, compression an decompression

costs may become prohibitive. So having a way to compress data and keeping it compressed for the entire course of the analytical pipeline, is very desirable.

This paper will focus solely on numerical data which for the purpose of the applications is organized as matrices. This is a most common data structure found in computational data analysis environments. The matrices compressed according to the methodologies proposed here should be able to undergo the same mathematical operations as the original uncompressed matrices, e.g. linear algebra manipulations. This way, the cost of compression is reduced to a single event of compression and no need of decompression, except when displaying the results for human reading. The idea of operating with compressed arrays is relatively new [6], and it has yet to find mainstream applications to the field of numerical computations. One application which employs a form of compression is the sparse matrix linear algebra algorithms [7], in this case there is no alteration in the standard encoding of the data, but only the non-zero elements of the matrices are stored and operated upon.

Larger than RAM data structures can render traditional analytical algorithms impracticable. Currently, the technique most commonly used when dealing with large matrices for numerical computations, is memory mapping [8, 9]. In memory, mapping the matrix is allocated in a virtual contiguous address space which extends from memory into disk. Thus, larger than memory data structures can be manipulated as if they were in memory. This technique has a big performance penalty due to lower access speeds of disk when compared to RAM.

In this paper, we present two methods for the lossless compression of (numerical) arrays. The methods involve the encoding of the numbers as strings of bits of variable length. The methods resemble the arithmetic coding [10] algorithm, but is cheaper to compute. We describe the process of compression and decompression, and study their efficiency under different applications. We also discuss the efficiency of the compression as a function of the distribution of the elements of the matrix, some results comparing our method with the traditional method of matrix allocation, and arithmetic operations as a benchmark measure the efficiency of operating with bitstring-compressed matrices.

Methods

Matrix compression

To maintain mathematical equivalence with the original data for any arithmetic operations, we need to maintain the structure of the matrix, ie, the ability to access any element given its row i and column j and also the numeric nature of its elements. In order to achieve compression, we decided to exploit inefficiencies in the conventional way matrices are allocated in memory. The analyses presented in this article concern arrays of positive integers, but can be applied to signed integers and real numbers with few adaptations.

The compression method is as follows. Let $M_{r\times c}$ be a matrix, in which r is the number of rows and c the number of columns. Each element of this matrix, called m_{ij}, is a positive integer. In digital computers, all information is stored as binary code (base 2 numbers). However, the conventional way to store arrays of integers is on a memory block sequence of fixed size (power of 2 number of bits), one for each element. The maximum size of a block is equal to the word size of the processor, which for most current CPUs is 64 bits. Some special number such as complex number may be encoded as two blocks instead of one. The size of the chunk of memory allocated to each number will determine their maximum size (for integers) or their precision (for floating-point numbers). So for matrix M, the total memory allocated, assuming chunks of 64 bits, is given by $\mathcal{B} = r \times c \times 64$.

The number of bits allocated $\mathcal{B}$, is larger than the absolute minimum number of bits required to represent all the elements of M, since smaller integers, when converted to base 2, require less digits. From now on, when the numerical base will be explicitly notated when necessary to avoid confusion between binary and decimal integers.

Let's consider an extreme example: a matrix composed exclusively of 0s and 1s (base 10). If the matrix type is set to 64-bit integers, 63 bits will be wasted per element of the matrix, since the minimum number of bits needed to store such a matrix is $\mathfrak{b} = r \times c \times 1$. The potential economy of bits ξ can be represented by $\xi = \mathcal{B} - \mathfrak{b} = r \times c \times 63$.

So it is evident that for any matrix whose greatest element requires less than 64 bits (or the fixed type of the matrix) to be represented, potential memory savings will grow linearly with the size of the matrix.

Method 1: The Supreme Minimum (SM)

The SM method consists in determining the value of the greatest element of matrix M, which coincides with its supremum, $maxM == \sup M$ and determine the minimum number of bits, $b(\sup M)$(Equation 1), required to store it. We will use capital roman letters to denote uncompressed matrices and the corresponding lower case letter for the compressed version.

$$b(\sup M) = \begin{cases} 1, & \text{if } \sup M \in \{0, 1\} \\ \lfloor \log_2(\sup M) \rfloor + 1, & \text{if } \sup M > 1 \end{cases} \tag{1}$$

The allocation of memory still happens in the usual way, ie, in fixed size 64-bit chunks, only that now, in the space required for a single 64 bit integer, we can store for example, an entire 8×8 matrix of 0_{10} and 1_{10}.

Let's look at a concrete example: suppose that the greatest value to be stored in a matrix M is $\max M = 1023$. Therefore, the number of bits required to represent it is 10 (1111111111). Let the first 8 elements of M be:

$$M = \begin{bmatrix} 900 & 1023 & 721 & 256 & 1 & 10 & 700 & 20 & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \tag{2}$$

These elements of M, in binary, are shown in Table 1. It is evident that the number of bits required to represent any other element must be lower or equal to 10. From now on the minimum number of bits required to represent a base 10 integer will be refered to as its bit-length.

To store matrix M it first has to be converted to base 2 (M_2). Then it will be unraveled by column (column major, e.g. in Fortran) or by row (row major, e.g. in C) and its elements will be written as fixed size adjacent chunks of memory. The size of each chunk is determined by the type associated with the matrix (typically 64 bits, but always a power of 2).

According to the SM method, having determined that each element will require at most 10 bits, we can divide the memory block corresponding to a single 64 bit integer into six 10-bit chunks, which can each hold a single element of M. These 64-bit blocks will be called a bitstring. The remaining 4 bits will be used later. The number of bitstrings needed will be $\lfloor \frac{\dim(M) * b(\sup M)}{64} \rfloor + 1$, where $dim(M)$ is the dimension of the matrix or its number of elements.

The final layout of the first 6 elements of m in the first bitstring can be seen in 3.

$$bitstring_1 = 0000\underbrace{0000001010}_{10}\underbrace{0000000001}_{1}\underbrace{0100000000}_{256}\underbrace{1011010001}_{721}\underbrace{1111111111}_{1023}\underbrace{1110000100}_{900} \tag{3}$$

Here is a step-by-step description of the application of the SM method to matrix M:

1. Element $M_{1,1} = 900 = m_{1,1} = 1110000100$ is stored in the first 10-bit chunk of the element strip $bitstring[1]$, which corresponds to bits 0 to 9 (read from right to left).

$$bitstring_1 = 0000000000000000000000000000000000000000000000000000\underbrace{1110000100}_{900}$$

Ser que no devemos trocar ≈ por = aqui? Note que o nmero de bits no aproximado e sim o exato.

2. Element $M_{1,2} = 1023$ is allocated in the second chunk, from bit 10 to bit 19.

$$bitstring_1 = 00000000000000000000000000000000000000000000\underbrace{1111111111}_{1023}\underbrace{1110000100}_{900}$$

3. Repeat for elements $M_{1,i}$ with $i = 1, \ldots, 6$ which are stored on the remaining chunks.

$$bitstring_1 = 0000\underbrace{0000001010}_{10}\underbrace{0000000001}_{1}\underbrace{0100000000}_{256}\underbrace{1011010001}_{721}\underbrace{1111111111}_{1023}\underbrace{1110000100}_{900}$$

4. Element $M_{1,7} = 700 = 1010111100$ does not fit on the remaining 4 bits of the first bitstring. So it will straddle two bitstrings, ie, it is divided in two segments a and b, a is written on the first bitstring and b on the second.

$$\underbrace{\underbrace{0000010100}_{20}\overbrace{101011}^{b}}_{bitstring_2} \,|\, \underbrace{\overbrace{1100}^{a}\underbrace{0000001010}_{10}\underbrace{0000000001}_{1}\ldots}_{bitstring_1}$$

Please, notice that bitstrings are written from right to left.

$$\underbrace{0000010100}_{20}\underbrace{\overbrace{101011}^{a}\,|\,\overbrace{1100}^{b}}_{700}\underbrace{0000001010}_{10}\ldots$$

Thus the compressed matrix $m = M_2$ requires less memory than the conventional storage of M as a 64-bit integer array. An illustration of the compression process and the allocation of the elements of matrix M in the matrix m is shown in Figure 1. Note that the matrix m has fewer columns that the matrix M, then it demands less memory to be represented. We also emphasize that the bits which represent the value 700 is divided in two parts and stored in different strips in the matrix m.

Method 2: Variable Length Blocks (VLB)

In the SM method, there is still waste of space since for elements smaller than the supremum, a number of bits remain unused.

In the VBL method, the absolute minimal number of bits are used to store each value. However, if we are going to divide the biststrings into variable length chunks, we also need to reserve some extra bits to represent the size of each chunk, otherwise the elements cannot be recovered once they are stored.

Lets use again the matrix described in Equation 2, where the largest element is number 1023. Now instead of assigning one chunk of the bitstring to each element of m, we will assign two chunks: the first will store the number of bits required to store the element and the second will store the actual element. The first chunk will have a fixed size, in this case, 4 bits. These 4 bits are the required space to store the bit-length of $\sup M$, in this case, 10.

Lets go through VLB compression step-by-step. The largest element of M is 1023. Its bit-length is 10 which in turn is 4 bits long in base 2 (1010). Thus the fixed size chunk is 4 bits long for every element.

1. The first element $M_{1,1} = 900$ requires 10 bits to store, so we write 10 in the first chunk and 900 in the second.

$$bitstring_1 = 00000000000000000000000000000000000000000000\underbrace{\underbrace{1110000100}_{\text{element=900}}\quad\underbrace{1010}_{\text{bit-length=10}}}_{M_{1,1}}$$

2. Do the same for the next element, $M_{1,2} = 1023$.

$$bitstring_1 = \ldots 00000000000000000000000000\underbrace{\underbrace{1111111111}_{\text{element=1023}}\ \underbrace{1010}_{\text{bit-length=10}}}_{M_{1,2}}\ \underbrace{\underbrace{1110000100}_{\text{element=900}}\ \underbrace{1010}_{\text{bit-length=10}}}_{M_{1,1}}$$

3. Element $M_{1,3} = 721$ is also added taking the bitstring to the state.

$$bitstring_1 = 00000000000000000000\underbrace{1011010001}_{721}\underbrace{1010}_{10}\underbrace{1111111111}_{1023}\underbrace{1010}_{10}\underbrace{1110000100}_{900}\underbrace{1010}_{10}$$

So far the VLB method is more wasteful than the SM, but when we add $M_{1,4} = 256$, we start to save some space.

4. Element $M_{1,4} = 256$ is added.

5. Elements $M_{1,5} = 1$ and $M_{1,6} = 10$ are added requiring a total of 13 bits instead of 20 with the SM method. With the addition of these elements we require a second bitstring.

$$bitstring_1 = \underbrace{0100}_{4}\underbrace{1}_{1}\underbrace{0001}_{1}\underbrace{100000000}_{256}\underbrace{1001}_{9}\underbrace{1011010001}_{721}\underbrace{1010}_{10}\underbrace{1111111111}_{1023}\underbrace{1010}_{10}\underbrace{1110000100}_{900}\underbrace{1010}_{10}$$
$$bitstring_2 = 00000000000000000000000000000000000000000000000000000000000000\underbrace{1010}_{10}$$

6. The remaining two elements are added $M_{1,7} = 700$ and $M_{1,8} = 20$ in the second bit strip.

$$bitstring_2 = 00000000000000000000000000000000\underbrace{10100}_{20}\underbrace{0101}_{5}\underbrace{1010111100}_{700}\underbrace{1010}_{10}\underbrace{1010}_{10}$$

We used a total of 87 bits to store matrix m with the VLB method instead of 80 bits using the SM method. However, as shall be seen later, the VLB method will be the most efficient for most matrices. With the VLB method the resulting matrix is smaller than the original matrix.

Compression Efficiency

Compression efficiency depends of the data being compressed. Below, a formula for calculating compression efficiency is derived for both methods. They will be based on the following ratio:

$$\eta = \frac{bits\,alocated - bits\,used}{bits\,alocated} \tag{4}$$

Where *bits alocated* above mean total bits required for standard storage of the matrix, without compression, while bits used mean total bits requires to store the matrix after compression. From now on the efficiencies are denoted by η_1 for the SM method and by η_2 for the VLB method.

SM Method

Let $M_{r\times c}$ be the matrix we wish to compress. In comparison with a conventional allocation (64-bit integers), we can apply Equation 4 to calculate the efficiency of the SM method:

$$\begin{aligned}\eta_1 &= \frac{64 \times rc - b(maxM) \times rc}{64 \times rc} \\ &= \frac{64 - b(maxM)}{64}\end{aligned} \tag{5}$$

As we see in 5, η_1 does not depend on size of the matrix, only on the bit-length of $\max M$. If $b(\max M) = 64$, η_1 is 0, ie, no compression is possible. On the other extreme, if the matrix is composed exclusively of 0s and 1s, maximal compression is achievable, $\eta_1 = 0.9843$.

VLB Method

For the VLB method, compression depends on the value of each element of the matrix. In this method bit-lentgh variability affects the compression ratio, so the formula will have to include this information.

Let the rc elements of the matrix $M_{r\times c}$ be divided into g groups, each with f_i numbers of bit-length $b_i = b(m_i)$. Thus f_i is the frequency of each bit-length present in M. Let $k = b(b(\max M))$, ie, the bit length of the bit-length of $\max M$. The efficiency η_2 is shown below.

$$\eta_2 = \frac{64 \times rc - \sum_{i=1}^{g}(b_i + k) \times f_i}{64 \times rc} \tag{6}$$

We can further simplify Equation 6 to get at shorter expression for the compression ratio.

$$\begin{aligned}\eta_2 &= \frac{64 \times rc - \sum_{i=1}^{g}(b_i \times f_i + k \times f_i)}{64 \times rc} \\ &= \frac{64 \times rc - \sum_{i=1}^{g} b_i \times f_i - \sum_{i=1}^{g} k \times f_i}{64 \times rc} \\ &= \frac{64 \times rc - \sum_{i=1}^{g} b_i \times f_i - k \times \sum_{i=1}^{g} f_i}{64 \times rc}\end{aligned}$$

Knowing that

$$\sum_{i=1}^{g} f_i = rc,$$

we can simplify the equation above, obtaining (7).

$$\begin{aligned}\eta_2 &= \frac{64 \times rc - \sum_{i=1}^{g} b_i \times f_i - k \times rc}{64 \times rc} \\ &= 1 - \frac{\sum_{i=1}^{g} b_i \times f_i}{64 \times rc} - \frac{k}{64}\end{aligned} \tag{7}$$

For this method, the highest value obtained for the data compression is 0.9687, when b = 1 and k = 1. However, the lower value is defined by a range of values, such that the value of $k = \frac{\sum_{i=1}^{g} b_i \times f_i}{rc} - 64$.

Results

Random Matrix Generation. In both methods, compression efficiency depends on the distribution of the bit-lengths $b(m_{i,j})$. Thus, in this section, a method to generate a variety of random bit-length distributions is proposed.

For simplicity we will model the distribution of as a mixture X of two Beta distributions, $B_1 \sim Beta(\alpha_1, \beta_1)$ and $B_2 \sim Beta(\alpha_2, \beta_2)$, whose probability function is shown in Equation 8. Since the Beta distribution is defined only in the interval $[0, 1] \subset \mathbb{R}$, we applied a simple transformation $(\lfloor 64 \times x \rfloor + 1)$ to the mixture in order to map it to the interval of $[1, 64] \subset \mathbb{Z}$.

$$f(x) = w\, Beta(\alpha_1, \beta_1) + (1 - w)\, Beta(\alpha_2, \beta_2) \tag{8}$$

The intention of using this mixture was to find a simple way to represent a large variety of bit length distributions. The first two central moments of this mixture are given in 9 and it will be used later to summarize our numerical results.

$$\begin{aligned} E(X) &= wE(B_1) + (1 - w)E(B_2) \\ &= w\frac{\alpha_1}{\alpha_1 + \beta_1} + (1 - w)\frac{\alpha_2}{\alpha_2 + \beta_2} \\ Var(X) &= wVar(B_1) + (1 - w)Var(B_2) + w(1 - w)(E(B_1)^2 - E(B_2)^2) \end{aligned} \tag{9}$$

In order to explore the compression efficiency of both methods, we generated samples from the mixture defined above, varying its parameters. From now on, when we mention Beta distribution we will mean the transformed version defined above.

From now on, we will apply Equations 5 and 7, to determine the compression efficiency of SM and VLB methods for random matrices generated as describe above.

With $w = 0$, a single Beta distribution is used. In Figure 2, we show some distributions of bit-lengths for some combinations of α_1 and β_1. From the figure it can be seen that a large variety of unimodal distributions can be generated in the interval $[1, 64]$.

As we are sampling from a large set distributions of bit-length, represented by the mixture of betas presented above, in order to make our results more general, we will base our analysis on the expected bit-length of a sample, since the efficiency of both methods depends on it. So, from Equations 5 and 7, the expected efficiencies become:

$$E(\eta_1) = 1 - \frac{k}{64} \tag{10}$$

$$E(\eta_2) = 1 - \frac{E(b)}{64} - \frac{k}{64} \tag{11}$$

where k, in (11), is set to 7 (the bit-length required to represent the largest possible bit-length: 64). In (10), k is the bit-length of the greatest element, or in the worst case, 64.

We will use the difference $D = E(\eta_1) - E(\eta_2)$ to compare the efficiency of the two methods. Thus a positive D will favor SM method while a negative D favors VLB method.

The expected compression efficiency in the following numeric experiments, will be calculated from 3 matrices of dimension 10000, generated as described, and presented in tables and figures below.

In Figure 3, we can see the distribution of efficiencies and their difference for a sample generated from a single Beta distribution of bit-lengths. Note that both methods can achieve efficiencies greater than 80% for matrices with very small numbers. Also note that the VLB method is more efficient in the majority of cases.

Now let $w = 0.5$, ie, matrices will have elements with bit-lengths coming from a mixture of beta distributions, $B_1 \sim Beta(\alpha_1, \beta_1)$ and $B_2 \sim Beta(\alpha_2, \beta_2)$. The expected value for this mixture is shown in Equation 12.

$$E(B) = 0.5E(B_1) + 0.5E(B_2) \tag{12}$$

For bit-lengths coming from a mixture ($w > 0$), let the expected efficiencies for the SM and VLB methods be as given by Equations 13 and 14. So now, instead of having the efficiency be a function of greatest bit-length in the sample (denoted as k in 5 and 7), it will be a function of $max\{E(B_1), E(B_2)\}$.

$$E(\eta_1) = 1 - \frac{max\{E(B_1), E(B_2)\}}{64} \tag{13}$$

$$E(\eta_2) = 1 - 0.5\frac{E(B_1)}{64} - 0.5\frac{E(B_2)}{64} - \frac{max\{E(B_1), E(B_2)\}}{64} \tag{14}$$

As before, we generate 3 matrices of dimension $10,000$ for each parameterization, calculate the average efficiencies (Equations 13 and 14) and their difference D.

Before moving on to efficiency results and analyses, let's first inspect samples from the mixture of transformed Beta distributions. Figures 4 and 5, show a few parameterizations and their resulting sample distributions. It is important to note that from the mixture, we can now generate bimodal distributions as well as the unimodal types tested before. Since we are making statements about efficiency as a function of the expected bit-length, it is important to verify if these statements hold for bimodal distributions as well.

After sampling uniformly ($[1, 5, 9, \ldots, 64]$, $n = 65,536$) the bit-length space and comparing efficiencies, we summarized the results on Table 2. In it we see how many parameterizations (from our sample) favor each method. We can also look at the distribution of efficiencies on our samples for each method (Figure 6), which clearly demonstrate the greater expected efficiency of method VLB (Figure 6(b)).

As we have shown, the VLB method is more effective compressing most integer datasets up to 64 bits in size. This is due to its ability to exploit the variance in the data set and reduce the waste of bits in the representation of some numbers. In specific cases, where the variance in the data null or too small, method I will be more efficient. As a matter of fact, for matrices where all elements have the same bit-length, SM method will always be better, regardless of bit-length (Figures 7(a) and 7(b)). There are only one exception: for bit-length 64, where neither method is able to compress the data.

Discussion

Calculating Efficiencies

To determine the best compression method to apply, it's necessary to inspect the distribution of bit-lengths of matrix elements. When matrix elements are small or have nearly-constant bit-length, the SM method is better, otherwise, the VLB method should be chosen.

As an example, let $M_{r\times c}$ be a integer matrix, such that the half of its elements have bit-length 1 and the other half 64. Recalling Equation 7, now we have two groups of elements (by bit-length), $b_1 = 1$, $b_2 = 64$ and $f_i = \frac{rc}{2}$ for $i = 1$ and 2. As the greatest bit-length is 64, then $k = 7$. Compression efficiency η_2 can be calculated using Equation 7. After plugging in our numbers, we obtain a compression of 38.29%.

$$\eta_2 = 1 - \frac{\sum_{i=1}^{2} b_i \times f_i}{64 \times rc} - \frac{7}{64}$$

$$\eta_2 = 1 - \frac{1 \times \frac{rc}{2} + 64 \times \frac{rc}{2}}{64 \times rc} - \frac{7}{64}$$

$$\eta_2 = 1 - \frac{32.5 \times rc}{64 \times rc} - \frac{7}{64}$$

$$\eta_2 \approx 1 - 0.5078 - 0.1093$$

$$\eta_2 \approx 38.29\%$$

The efficiency of the VLB method is influenced by the relative size of the bit-length groups. In this first example, we considered only two groups, each comprised of half the matrix elements. Let's now vary the relative frequency of the groups, $\frac{f_i}{rc}$, while sticking to two groups. Here, we assume that $\frac{f_i}{rc}$ is a good approximation to the probability of a given bit-length in a matrix, which we will denote by p_i.

With this definition, we can rewrite the Equation 7, which becomes 15. In Equation 15, the $\frac{f_i}{rc}$ is replaced by p_i, representing the probability of elements from group i in matrix M.

$$\eta_2 = 1 - \frac{\sum_{i=1}^{g} b_i \times p_i}{64} - \frac{k}{64} \tag{15}$$

with

$$p_i = \frac{f_i}{rc} \tag{16}$$

With the Equation 15 can analyze the influence of bit-length probability in compression efficiency. In this example, p_1 and p_2 represent the probability of elements of bit-lengths 1 and 64, respectively. Thus, efficiency is defined in Equation 17.

$$\eta_2 = 1 - \frac{1 \times p_1 + 64 \times p_2}{64} - \frac{7}{64} \tag{17}$$

Now, we can determine which probabilities give us the best and worst compression levels. When $\eta_2 = 1$, then the efficiency is maximal and if $\eta_2 = 0$, a efficiency is minimal. To calculate the values of p_1 and p_2 for both extreme values of η_2, we must solve the linear systems shown in Equations 18 and 19. The first equation on both systems come from the law of total probability. The second comes from 17 after setting η_2 to 1 and 0, respectively.

$$\eta_2 = 1 : \begin{cases} p_1 + p_2 = 1 \\ p_1 + 64p_2 = 7 \end{cases} \tag{18}$$

Solving the system above, we find that when $p_1 = 0.9047$ and $p_2 = 0.0953$, efficiency is maximal, and in this particular case is equal to 87.5%.

$$\eta_2 = 0 : \begin{cases} p_1 + p_2 = 1 \\ p_1 + 64p_2 = 57 \end{cases} \tag{19}$$

Thus, when $p_1 = 0.1111$ and $p_2 = 0.8889$ the efficiency is minimal for the VLB method. For other combinations see Table 3. Looking at this table, one can see two negative efficiencies, when (p_1,p_2) assume the values (0,1) and (0.1,0.9). This correspond th cases, when the method increases, the memory requirements instead of decreasing it.

So far, we have examined only two groups (hence two probabilities) of bit-length for the sake of simplicity. Before, we generalize to probability distributions let's take a quick look at the efficiencies for more groups, with uniform probability:

- 3 groups with bit-lengths 1, 32 and 64 bits, efficiency $\eta_2 = 0.3854$,
- 5 groups with bit-lengths 1, 16, 32, 48 and 64 bits, efficiency $\eta_2 = 0.3875$,
- 8 groups with bit-lengths 1, 8, 16, 24, 32, 40, 48, 56 and 64 bits, efficiency $\eta_2 = 0.3888$

When the distribution of the group probabilities is uniform, ie, the groups have approximately the same size, efficiency is basically the same, regardless of the number of groups.

Now, we can leverage the notion of bit-length probabilities, and study efficiency when bit-lengths follow some commonly used discrete probability distributions: Discrete Uniform, Binomial and Poisson. For all the experiments, we assume $k = 7$, that is, the maximum possible bit-length is 64 bits. Thus, efficiency obtained will not be the best possible, since for that we would need assume small values of k (Equation 15).

Discrete Uniform

Let the bit-lengths of the matrices be distribute according to the Uniform distribution $U(a = 1, b = 64)$, which means bit-lengths may take values in the set $\{1, 2, 3, \ldots, 64\}$ with equal probability, ie, $\frac{1}{64}$.

Theoretical Efficiency: Let the random variable $B \sim U(a = 1, b = 64)$ represent the bit-length of the elements of matrix M. Then $E(b_i) = \sum_i b_i \times p(b_i) = \frac{a+b}{2}$. Applying this result to the expected compression efficiency of VLB method (Equation 15), we have

$$E(\eta_2) = 1 - \frac{E(B)}{64} - \frac{k}{64} \tag{20}$$

assuming all bit-lengths are possible, i.e., $a = 1$ and $b = 64$, and hence $k = 7$, we can calculate η_2:

$$E(\eta_2) = 1 - \frac{\frac{1+64}{2}}{64} - \frac{7}{64} \approx 38.28\% \tag{21}$$

This result agrees with the numerical estimates presented in Table 4.

Numerical Estimates: To calculate the VLB efficiency, we generated a matrices with 100 ($M_{10\times10}$), 10,000 ($M_{100\times100}$) and 1,000,000 ($M_{1,000\times1,000}$) elements with 1, 8, 16, 32 and 64 number of bits. The average efficiency (Table 3) is calculated from a 1,000 replicates of each matrix size. As expected the compression efficiency gets better with lower expected bit-length.

Binomial Distribution

For the binomial distribution, we will use $Bin(n, p)$, with the number of trials n representing the greatest possible bit-length in the matrix, and np giving us the expected bit-length.

Theoretical Efficiency: Let bit-length (B) be a random variable with Binomial distribution, $B \sim B(n = 64, p = 0.5)$, $E(b_i) = \sum_i b_i \times p(b_i) = n \times p$ and the efficiency becomes (with $k = 7$):

$$\begin{aligned} E(\eta_2) &= 1 - \frac{64 \times p}{64} - \frac{k}{64} \\ &= 1 - \frac{64 \times 0.5}{64} - \frac{7}{64} \approx 39.05\% \end{aligned} \tag{22}$$

Which again agrees with estimates in Table 5.

Numerical Estimates: For these experiments, the parameter n represents the maximum bit-length of matrix elements and takes values in $\{1, 8, 16, 32, 64\}$. In this case, we evaluate the efficiency as a function of the parameter n, and matrix size. Even though efficiency does not depend on matrix size, we tried different sizes to test the stability of the compression algorithm. Results are shown in Table 5. As expected, smaller bit-lengths lead to higher compression efficiencies.

Poisson Distribution

With bit-length derived from a Poisson(λ), the parameter λ corresponds to the expected bit-length. For the purpose of this analysis this Poisson distribution is truncated at 64.

Theoretical Efficiency: Let bit-length $B \sim Poisson(\lambda = 32)$. In this case, $E(b_i) = \lambda$, with $k = 7$, the efficiency becomes:

$$E(\eta_2) = 1 - \frac{\lambda}{64} - \frac{k}{64} \tag{23}$$

$$= 1 - \frac{32}{64} - \frac{7}{64} \approx 39.06\% \tag{24}$$

This result is in accordance to Table 6.

Numerical Estimates: The results for this simulation can be seen in table 6. Note that increasing the number of bits to represent numbers increases, there is a loss of efficiency in the compression process. In this case, we did not simulate for $\lambda = 64$ since a large portion of the samples would fall above the maximum bit-length.

These results show that a good compression is guaranteed when bit-lengths are distributed according to the tested distributions regardless of sample size.

Implementation

acho que esta seo toda deve ir para o material suplementar.

colocar o link para o repositrio aqui

This paper proposes the bitstring-compression methodologies. The SM method was implemented as a Fortran library which is available under an open-source license. The implementation extends the standard Matrix type of Fortran, overloading operators such as assignment, addition, subtraction, multiplication, transpose, and maximum of a matrix. Although only few significant matrix operations have been ovearloaded in this example implementation, any other operation can be performed, since the methods for inserting and collecting an element in a matrix of bitstring were also implemented. With these two methods, as long as they are properly adapted into algorithms, it is possible to implement any other desired operation.

The SM method was also extended to allow for the allocation of real and signed integer matrices, besides positive integer ones. Some adjustments were made so that the other types of numbers could be represented. In order to represent the signed integers, it was necessary to use two's complement, as it is applied to thein accordance to traditional methods.

acho que esta explicao ficou muito solta. No precisamos explica a implementao no artigo, podemos colocar na documentao do cdigo

The SM method uses a fixed number of bits to store the values in such a way that the size or number of bits used is important to represent the highest value in the bitstring matrix. The adaptation consists in determining the largest value to be stored considering the absolute value of the number. That done, when the number is negative, it is necessary to rewrite it since, due to the sign, this will be written in the form of two's complement. The purpose of this conversion is to optimize library processing. Take the following as an example: the number 5, when represented in a 32-bit integer, is defined as:

$$5 = 00000000000000000000000000000101$$

However, if the number is negative, this is represented in the form of two's complement. Therefore, the number -5 is represented as:

$$-5 = 11111111111111111111111111111011$$

Note that the bit positioned more to the left should be used to inform that the number is -5. For the purpose of optimization of the library, it was chosen to conversion of positive numbers to negative numbers and use of 1 bit to represent the sign. (Dvidas aqui)

In turn, in order to represent real numbers, a conversion procedure of real numbers to integer numbers was used. The real number is multiplied by a power of ten and is then immediately truncated. The real numbers in a binary system of 64 bits are represented using a bit to represent the sign of the number, 11 bits to the exponent, and 52 bits to the mantissa. Note that 1 bit is still used implicitly, according to the IEEE-754 standard. In summary, the mantissa is represented in the base 2^{-1}, 2^{-2}, 2^{-3}, ..., 2^{-52}. Depending on the value to be represented, loss of information can take place, which is related to the accuracy of the type used. In this case, considering a real number with double precision, the demand of 64 bits to represent each one occurs, and in some cases, the number to be represented is an approximation. As an example, consider that the number to be stored is 1.109. Following the traditional system, the number 1,109 is represented by

0011111111110001101111100111011011001000101101000011100101011000

We consider that the power used is 10^3, that is, the number which needs to be stored would be 1109. In binary, that number is represented by 10001010101. Note that the number of bits required is smaller, but depending on numerical accuracy, the compression methods suggested are ineffective. When the power of 10 is used, it must be applied across the matrix. This conversion allows the proposed methods to be applied to a set of real numbers. It should be taken into account that the greater the power of 10 used, which is associated with the conversion of the real number, the larger the set of bits will be required to represent the converted number. This affects the efficiency of data compression. The use of the library to represent real numbers is conditioned to a study of the necessary level of precision to solve the problem.

Library Application

Tambm deve ir para o material suplementar

In order to create an application, some elementary matrix operations. Different operations have been performed by measuring the execution time of each. We basically, considered the attribution of a constant to all the elements of a matrix; the addition, subtraction and multiplication of the elements of a matrix by a constant, the addition, subtraction and multiplication of matrices. In addition to that, the calculation of the transposed matrix and the maximum an array were also considered. This measure was used to evaluate the efficiency of the method implemented when compared to traditional methods.

One of the characteristics of bitstring matrices is that the operations per column, in the case of implementation in Fortran, can be parallelized. This reduces the execution time, compared to traditional methods. Therefore, in order to achieve this optimization, threads implementations were used for the implemented operations.

As a test methodology, all the described operations were also compared to traditional methods. The matrix dimensions n x n used were $n = 10,\ 100,\ 1,000,\ 10,000,\ 20,000$ to $100,000$, by $10,000$. We emphasize that the matrix, created with a 1,000 dimension has 10^6 elements, while the one with a 100,000 dimension has 1×10^{10} elements. With the increase of the size of the matrices, a larger amount of memory must be reserved to allocate the numbers. The compression effect of the matrices can be verified in the results since by the proposed method the matrix is directly loaded into the main memory, avoiding the need for access to the secondary memory.

In Figures 9 and 10 the results from the processing time (in milliseconds) for the operations that involved only integer (Z) and positive integer numbers (Z^+) are represented. The traditional method (Normal) and the Bitstring method (Bitstring Z, integer numbers and Bitstring Zp, positive integer numbers) were used. The operations were attribution ($A = 5$, A is a matrix with dimension n by n); sum

of the elements of a matrix with a constant ($A = A + 5$); sum of matrices ($A = A + A$); andsubtraction of the elements of a matrix with a constant ($A = A - 8$). These operations are represented in Figure 9. Figure 10, in turn, shows the results for the multiplicationoperations of matrices ($A = A \times A$); for the multiplication of the the elements of a matrix by a constant ($A = A \times 2$); and for the calculation of transpose and maximum of a matrix.

The results show that the bitstring method is faster than the traditional ones only for the attribution. This occurs because the element of the matrix is used more than once to store the numbers, differently from the traditional method. Due to the number of manipulations the bitstring method does, the other operations demand more time to be processed, but this can be optimized. When applied to positive integers, the bitstring model works faster than when applied to integers because it does not need to do operations related to the signals.

The results for the real numbers are represented in Figures 11 and 12. By repeating the operations tested in the integer numbers, it was possible to see that the bitstring method cannot overcome the traditional method. This happensbecause its application to the real numbers, as alternative to maintain the compression, asks for a conversion of real to integer numbers in order for them to be stored and when accessed, another conversion is made. This is a problem from the implementation process, but it can be optimized.

Another library application was the implementation of the algorithm in order to deal with the problem of arrays of collaborative filtering [11]. These arrays have large dimensions due to the number of users and products. Each element of this matrix stores the rating of a user with respect to a specific product. With this information, it is possible to specify a product for a determined customer.

The manipulation of these matrices, depending on the amount of memory to be allocated, are one of the possible applications of the methods proposed in this work, since it promotes the compression and data allocation in the main memory. The allocation of the matrix within the main memory makes the access time to the elements much smaller than when the matrix is stored in secondary memory. This latter type of storage is used in some already mentioned techniques. As an example, let us examine the manipulation of a CF matrix, evaluating the access time to the elements and especially the size of the memory required to run such operation. The algorithms applied to this matrix were the average per-use and the bias from mean [11], this last one of the simplest algorithms for predicting the rate of users.

An example of the application of the methodology presented is shown in Figure 8. In this figure, the matrix with normal and bitstring formats are displayed. In both matrices, the information is represented and can be accessed. Therefore, it becomes possible to apply the selected algorithms.

In order to test a possible model application, we considered a matrix with the following dimensions: 95,000 users (lines) by 3,000 movies (columns), with the ratings for Netflixs movies [1]. In order to evaluate a larger dataset, we modified the NetFlix data so we could increase the dimension of the matrix to be analyzed. The 95,000 users correspond to thelines of the matrix. In orderto gradually increase the number of lines, we performed consecutive re-samplings and added them to the original matrix . As a results, we obtained a matrix with 600,000 lines and 3,000 columns. The operations used with the matrix were the Per User Average (PUA) and the rating estimation of a movie i using the Bias From Mean (BFM) [11]. The analyses were made, considering the created matrix with different dimensions, with n from 40,000 to 600,000 by 20,000. It is important to emphasize that the number of columns is constant and equal to 3,000. The implementations of this problem were made in Fortran and both the developedlibrary based in the method SM and in the traditional form were used. Codes were inserted in the library in order to optimize the multicore processing using threads. No additional configuration was necessary to use the multicore processing. since the library detects the characteristics of the processor uses them in the simulations. In turn, for the traditional implementation, weused the optimization process and parallelization, which are made available by the processor,as the vectorization and multicore parallelization.

The results are represented in Figures 13, 14 and 15 for the PUA Method and in Figures 16, 17 and

[1] http://graphlab.org/downloads/datasets/

18 for the BFM Methods. A computer with 8 GB of RAM Memory, 16 GB of swap, processor core i7 model 870 and operational system Linux Debian 7.0 was used to perform this simulation. In Figure 13, it is possible to verify the approximate moment when the the storage in disk (swaap) begins to be used. This is indicated by the vertical green line. At this moment, the bitstring method becomes faster than the traditional method because it does not require disk storage due tocompressed memory representation. Note that, in the Figure 14, the traditional method is faster than the bitstring one before the beginning of the disk access. However, after it becomes necessary use the disk so that the matrix can be accessed and stored, , the traditional method becames slower (Figura 15). These results indicate that the bistring method can promote the optimization of the memory use because the access to the main memories (RAM and cache) are faster than the access to secondary memories (disk). In Figures 16, 17 and 18 the results of the processing time for the BFM method are represented. Once again, a vertical green line indicates the approximate moment when the disk access begins. There is a difference in relation to the PUA method since the BMF method uses the PUA values to perform its calculation. Therefore, the BMF method demands more memory to be processed, which reduces the matrix dimensions involved in the process, represented in the main memory. As indicated in Figure 15, for the PUA method, and Figure 18, for the BMF method, it is possible to verify that before the access to the disk, the traditional method is faster than the bitstring one, but once the access is initiated, the bitstring method becames faster.

confirmar modelo

In spite of the simplicity of the application, it is relevant since it allows for large-sized matrices to be allocated in memory and manipulated.

Conclusion

In this paper, we have focused in the compression of matrix data, since this is one of the most important application the authors foresee. However, the compression methodology presented can be applied to any numerical data structure, with gains to performance and memory footprint [12].

Further discussions about doing computation with such compressed data-structures will be the subject of another manuscript (in preparation) in which we will present details about the implementation of the compression algorithm, and benchmarks on classical linear algebra tasks such as those in Linpack [13].

For the compression calculations presented in this paper we limited bit-lenght of integers to 64 bits. However the compression would work in the same way as discussed for computer architectures with larger word sizes.

Representation of floating point numbers is also possible within the proposed compression framework, but at the expense of precision in their representation. Although this may sound like a limitation, when we take into consideration that most experimental data have fewer “significant” digits than the maximal precision available in modern computers, fairly good compression may still be achievable for floats.

Acknowledgments

We would like to thank Claudia Torres Codeço, Moacyr Silva and Paulo Cezar Carvalho for fruitful discussions and key ideas which helped improve the manuscript.

Adicionar a Carol

References

1. Alted F (2010) Why modern cpus are starving and what can be done about it. Computing in Science & Engineering 12: 68–71.

2. Salomon D, Motta G, Bryant D (2009) Handbook of Data Compression. London: Springer, 1359 pp.

3. Salomon D (2007) Data Compression: The Complete Reference. Number v. 10 in Data compression: the complete reference. London: Springer-Verlag New York Incorporated.

4. Lynch C (2008) Big data: How do your data grow? Nature 455: 28–29.

5. Gaber M, Zaslavsky A, Krishnaswamy S (2005) Mining data streams: a review. ACM Sigmod Record 34: 18–26.

6. Yemliha T, Chen G, Ozturk O, Kandemir M, Degalahal V (2007) Compiler-directed code restructuring for operating with compressed arrays. In: Proceedings of the 20th International Conference on VLSI Design held jointly with 6th International Conference: Embedded Systems. Citeseer, pp. 221–226.

7. Dodson D, Grimes R, Lewis J (1991) Sparse extensions to the fortran basic linear algebra subprograms. ACM Transactions on Mathematical Software (TOMS) 17: 253–263.

8. Van Der Walt S, Colbert S, Varoquaux G (2011) The numpy array: a structure for efficient numerical computation. Computing in Science & Engineering 13: 22–30.

9. Kane MJ, Emerson JW (2012) bigmemory: Manage massive matrices with shared memory and memory-mapped files. URL `http://CRAN.R-project.org/package=bigmemory`. R package version 4.3.0.

10. Bodden E, Clasen M, Kneis J (2007) Arithmetic coding revealed. In: Sable Technical Report 2007-5, Sable Research Group, School of Computer Science, (McGill University, Montréal.

11. Lemire D, Maclachlan A (2007) Slope one predictors for online rating-based collaborative filtering. CoRR abs/cs/0702144.

12. Paixão CA (2012) Bitstring model for study of the propagation Dengue fever. Phd thesis, Federal University of Lavras.

13. Dongarra JJ (1979) LINPACK users' guide. 8. Siam.

Figure Legends

Adicionei a ltima frase

A escala de cor a mesma em todas as figuras, no texto estava dizendo que era apenas na a, b e c. Eu alterei.

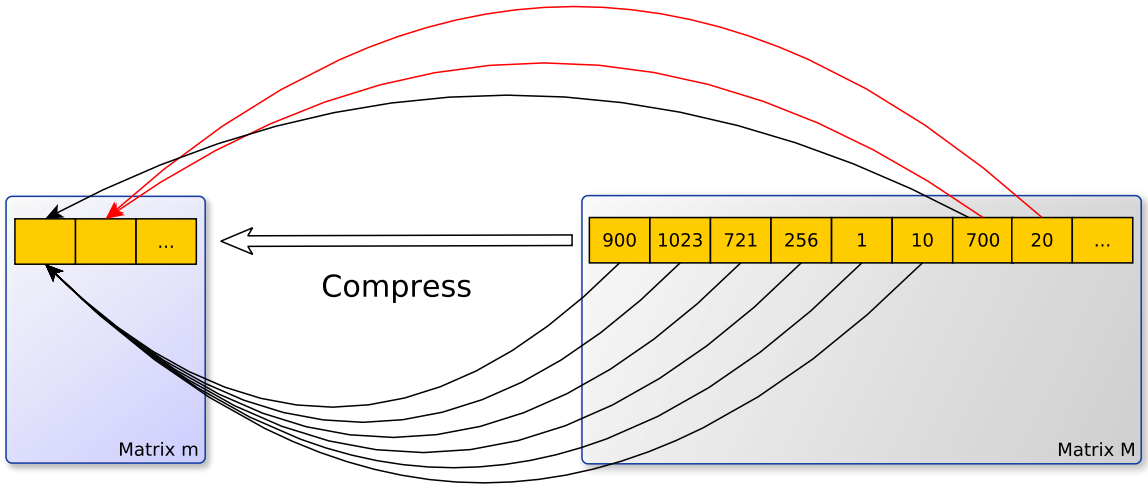
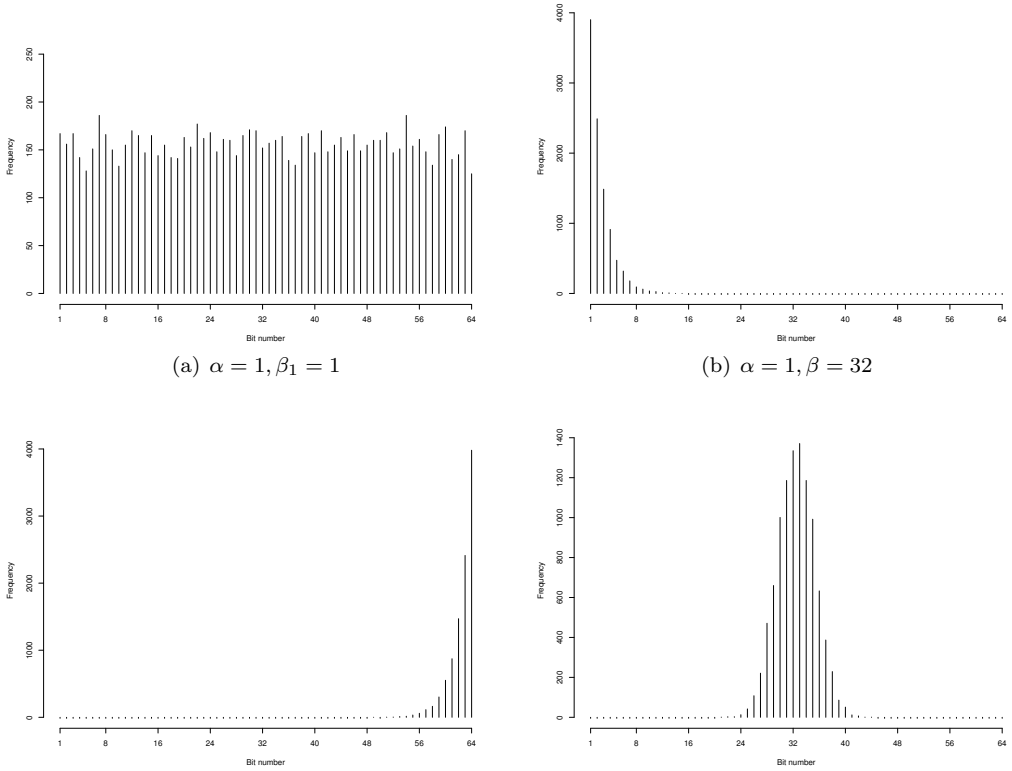


Figure 1. Representation of the compression process and allocation of elements of matrix M in the matrix m. The strip of value 700 is divided in two parts and stores in different strips in the matrix m.



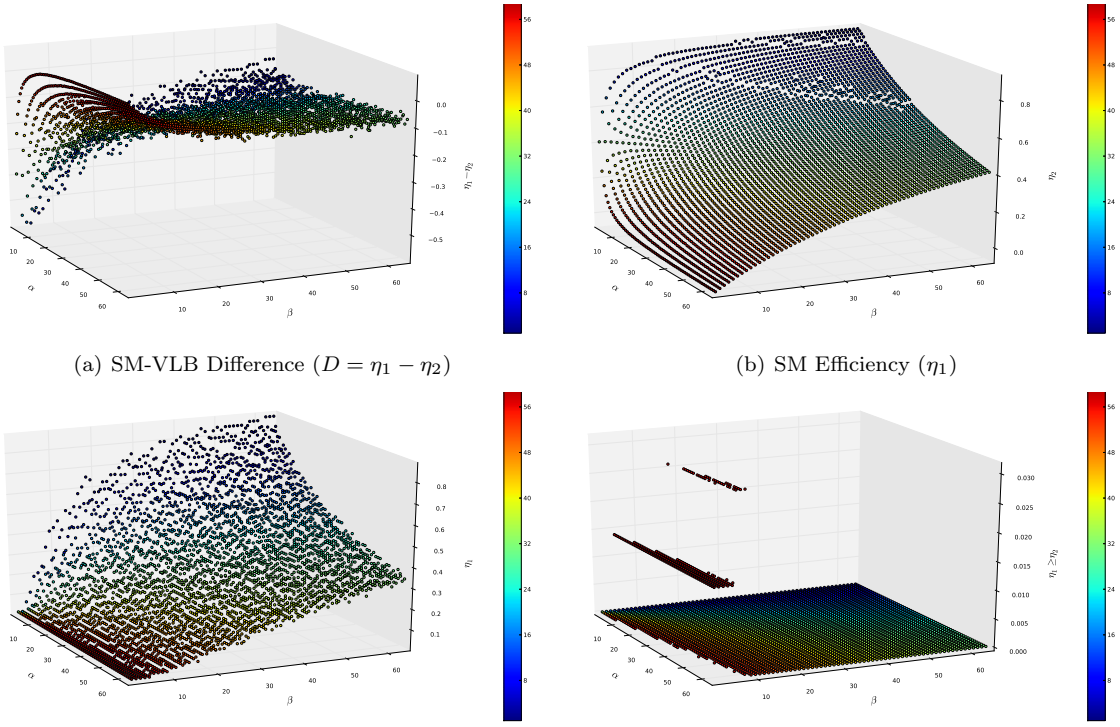
(a) $\alpha = 1, \beta_1 = 1$

(b) $\alpha = 1, \beta = 32$

(c) $\alpha = 32, \beta = 1$

(d) $\alpha = 64, \beta = 64$

Figure 2. Histograms constructed from samples with 10,000 elements, generated from a Beta distribution. Below each histogram is possible to verify the parameters used.With these two parameters it is possible to perform different combinations of numbers to fill the arrays to be compressed.



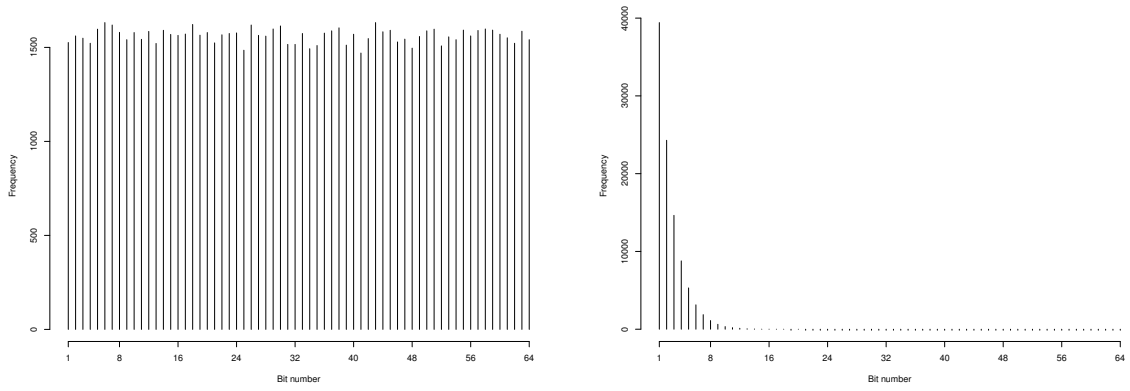
(a) SM-VLB Difference ($D = \eta_1 - \eta_2$)

(b) SM Efficiency (η_1)

(c) VBL Efficiency (η_2)

(d) SM≥VLB Efficiency ($\eta_1 \geq \eta_2$)

Figure 3. Comparing compression efficiency of methods 1 and 2. Color scale in represent average bit-length. In (a) we can see the difference $D = \eta_1 - \eta_2$. It can be seen that for most combination of α and β, $D < 0$, meaning the second method is more efficient to compress a sample of numbers with bit-lengths coming from a $Beta(\alpha, \beta)$ distribution. However, there is a small region in parameter space, which is shown in white on (d), where the SM method is more efficient. This region corresponds to the dots in red in (a), where the average bit-length is higher. In panels (b) and (c), we can see the efficiencies of SM and VLB methods, respectively.



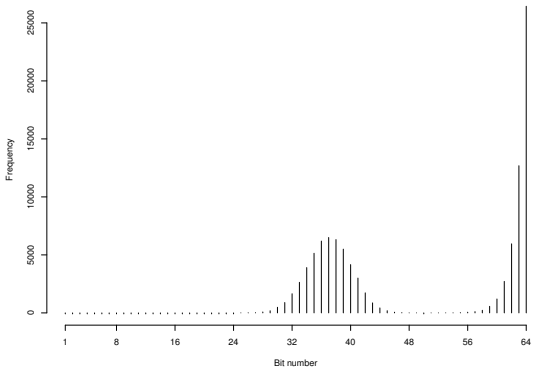
(a) $\alpha_1 = 1, \beta_1 = 1, \alpha_2 = 1, \beta_2 = 1$

(b) $\alpha_1 = 1, \beta_1 = 32, \alpha_2 = 32, \beta_2 = 1$

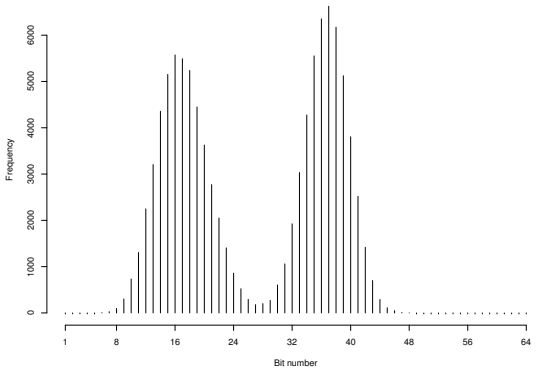
(c) $\alpha_1 = 32, \beta_1 = 32, \alpha_2 = 32, \beta_2 = 32$

(d) $\alpha_1 = 64, \beta_1 = 32, \alpha_2 = 32, \beta_2 = 64$

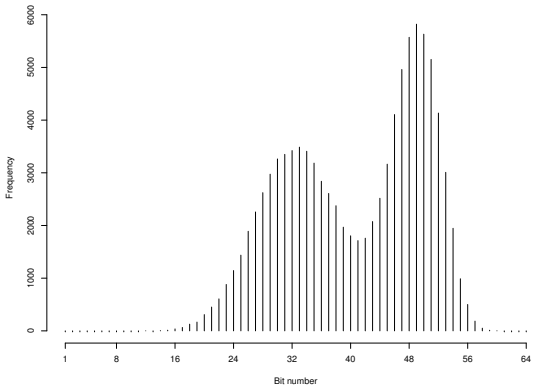
Figure 4. Histograms constructed from samples with 10,000 elements, generated from the mixture of two Beta distributions with $w = 0.5$. Below each histogram are the parameters of the mixture.



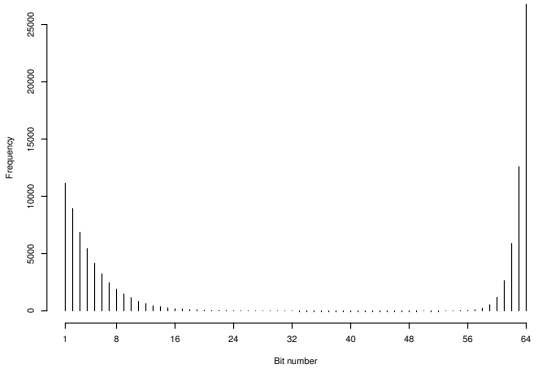
(a) $\alpha_1 = 64, \beta_1 = 48, \alpha_2 = 1, \beta_2 = 48$



(b) $\alpha_1 = 16, \beta_1 = 46, \alpha_2 = 49, \beta_2 = 64$

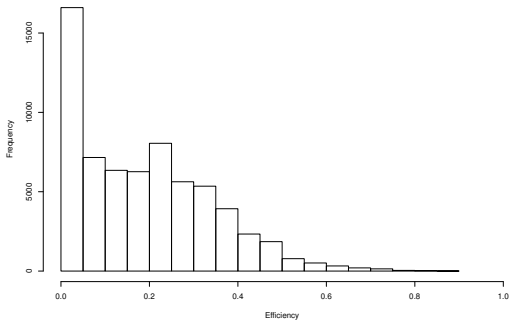


(c) $\alpha_1 = 16, \beta_1 = 16, \alpha_2 = 16, \beta_2 = 49$

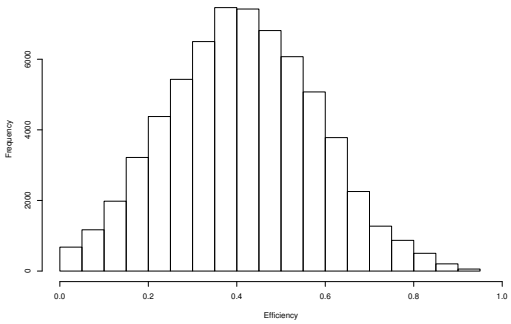


(d) $\alpha_1 = 1, \beta_1 = 16, \alpha_2 = 1, \beta_2 = 49$

Figure 5. Histograms constructed from samples with 10,000 elements, generated from the mixture of two Beta distributions with $w = 0.5$. Below each histogram are the parameters of the mixture.

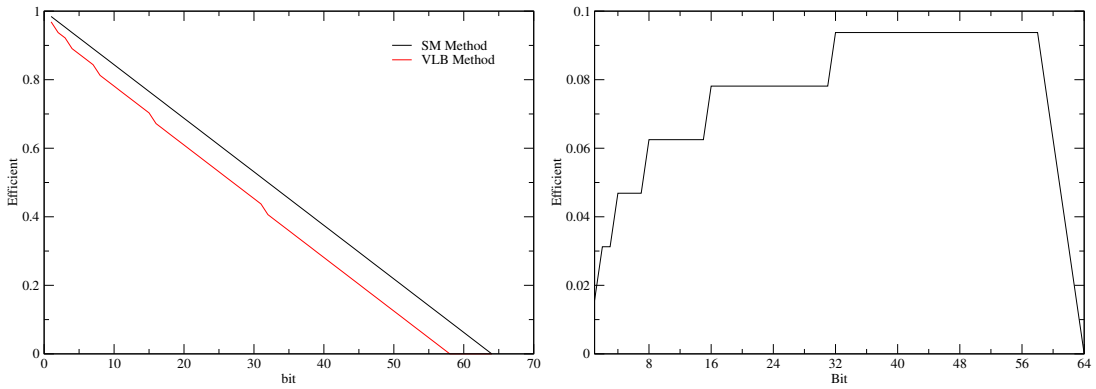


(a) Efficiency histogram of the SM method



(b) Efficiency histogram of the VLB method

Figure 6. Efficiency histograms of the SM and VLB methods. Note that the VLB method has a greater average efficiency than SM method.



(a) Efficiencies of the SM and VLB methods, for constant bit-length matrices.

(b) $\eta_1 - \eta_2$ for matrices of constant bit-length.

Figure 7. Compression efficiency of the SM and VLB methods for matrices of constant bit-length.

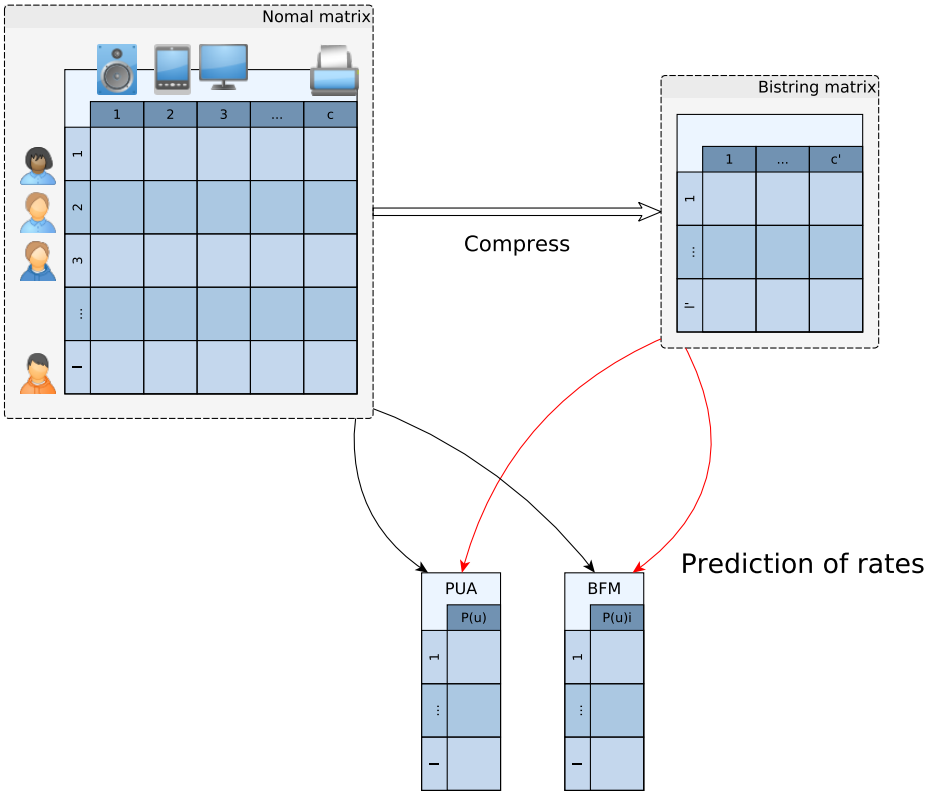
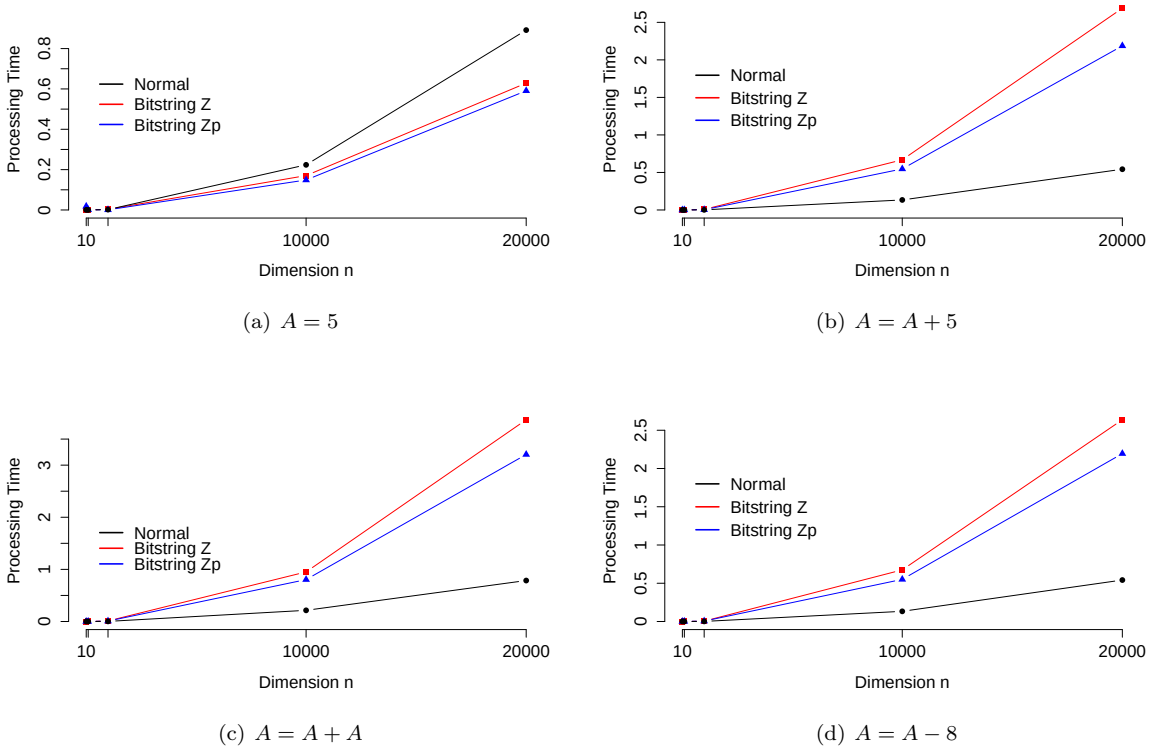


Figure 8. Illustration of the process of the bistring model application to matrices of collaborative filttering of the algorithms per use average (PUA) and of the bias from mean (BFM) in a matrix represented in the traditional and bitstring formats.



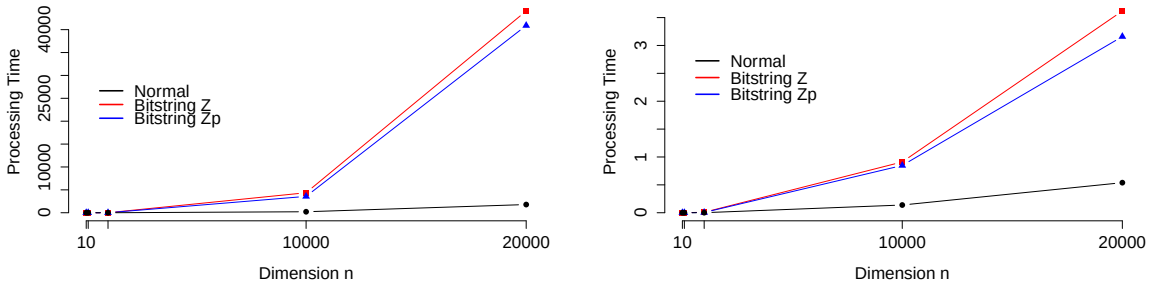
(a) $A = 5$

(b) $A = A + 5$

(c) $A = A + A$

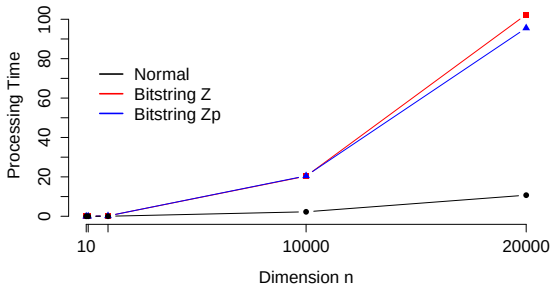
(d) $A = A - 8$

Figure 9. Processing time of the following operations: attribution ($A = 5$, A is the matrix with an n by n dimension); element sum of a matrix with a constant ($A = A + 5$);sum of matrices ($A = A + A$); and subtraction of the elements of a matrix by a constant ($A = A - 8$) involving integer numbers (Z) and positive integer numbers (Z^+). The bitstring method presented the best overall performance only for the attribution operations, because the storage of different numbers occurs in one element of a matrix, which reduces the accessing time. For the other operations, the bitstring method is slower due to the number ofoperations which are necessary to be executed in order to access the elements. This can still be optimized in the implemented library.

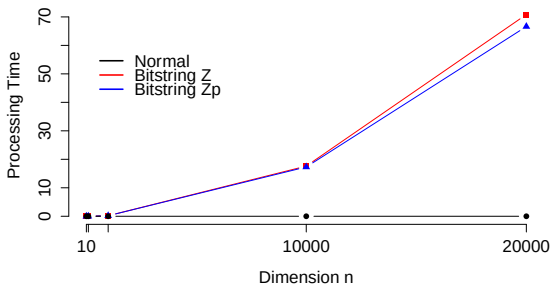


(a) $A = A \times A$

(b) $A = A \times 2$

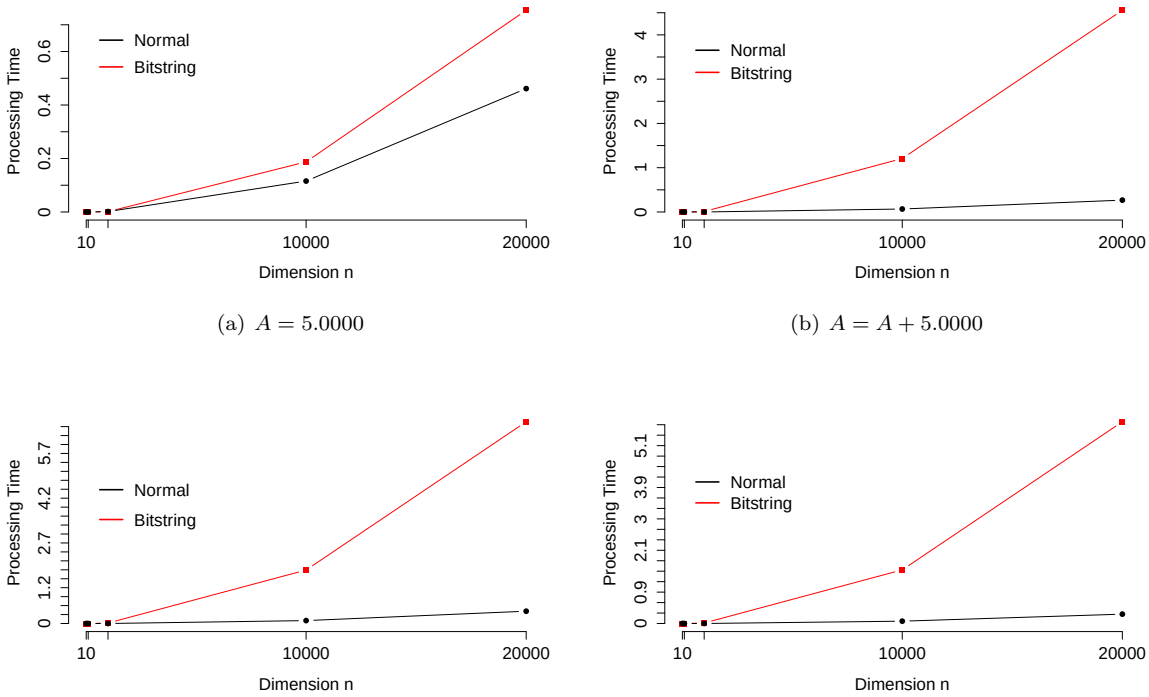


(c) $transpose(A)$



(d) $maximum(A)$

Figure 10. Processing time of matrix multiplication ($A = A \times A$, A is the matrix with a n by n dimension), element multiplication of matrix by constant ($A = A \times 2$), the calculation of transpose and maximum of a matrix involve integer numbers (Z) and positive integer numbers (Z^+). In this cases, the traditional method was faster than the bitstring model.



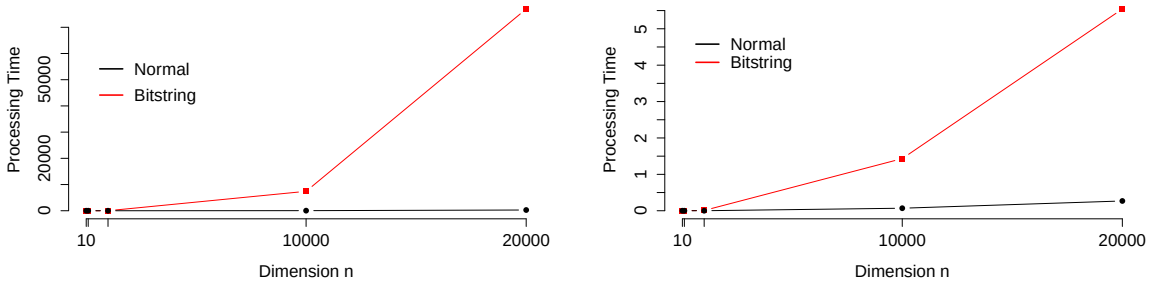
(a) $A = 5.0000$

(b) $A = A + 5.0000$

(c) $A = A + A$

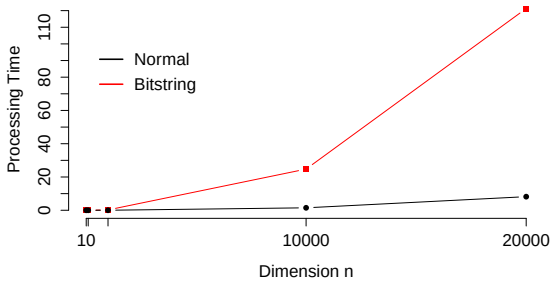
(d) $A = A - 8.0000$

Figure 11. Processing time of the following operations: attribution ($A = 5.0000$, A is the matrix with an n by n dimension); element sum of a matrix with a constant ($A = A + 5.000$);sum of matrices ($A = A + A$); and subtraction of the elements of a matrix by a constant ($A = A - 8.0000$) involving real numbers (R) and positive integer numbers (R^+). The bitstring method presented the best overall performance only for the attribution operations, because the storage of different numbers occurs in one element of a matrix, which reduces the accessing time. For the other operations, the bitstring method is slower due to the number ofoperations which are necessary to be executed in order to access the elements. This can still be optimized in the implemented library.

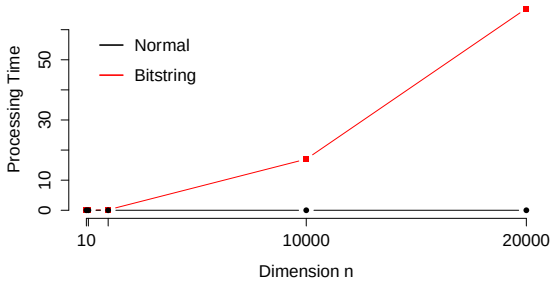


(a) $A = A \times A$

(b) $A = A \times 2$



(c) $transpose(A)$



(d) $maximum(A)$

Figure 12. Processing time of matrix multiplication ($A = A \times A$, A is the matrix with a n by n dimension), element multiplication of matrix by constant ($A = A \times 2.0000$), the calculation of transpose and maximum of a matrix involve integer numbers (R) and positive integer numbers (R^+). In this cases, the traditional method was faster than the bitstring model.

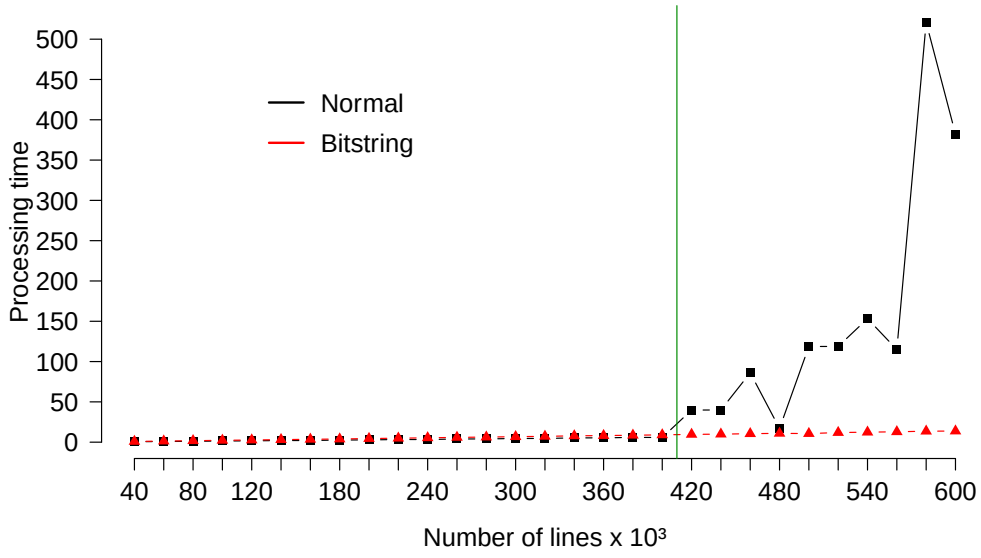


Figure 13. Processing time for the Per User Average method (PUA) in function of the numbers of linesof a matrix with 3,000 columns. The vertical green line indicates the approximate moment when the traditional method demands disk access in order to store the matrix to be analyzed, which overloads the process. The bitstring method fully represents a matrix in memory, without the necessity to access the disk.

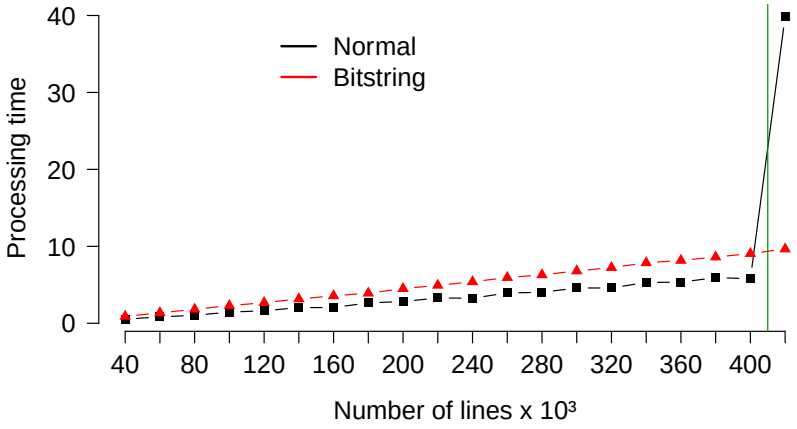


Figure 14. Processing time for the Per User Average method (PUA) in function of the numbers of lines of a matrix with 3,000 columns. The vertical green line indicates the approximate moment when the traditional method demands disk access in order to store the matrixto be analyzed, which overloads the process. For dimensions inferior to approximately 400 lines, the traditional method is faster than the bitstring one. It is important to emphasize that the computational cost of both methods is linear.

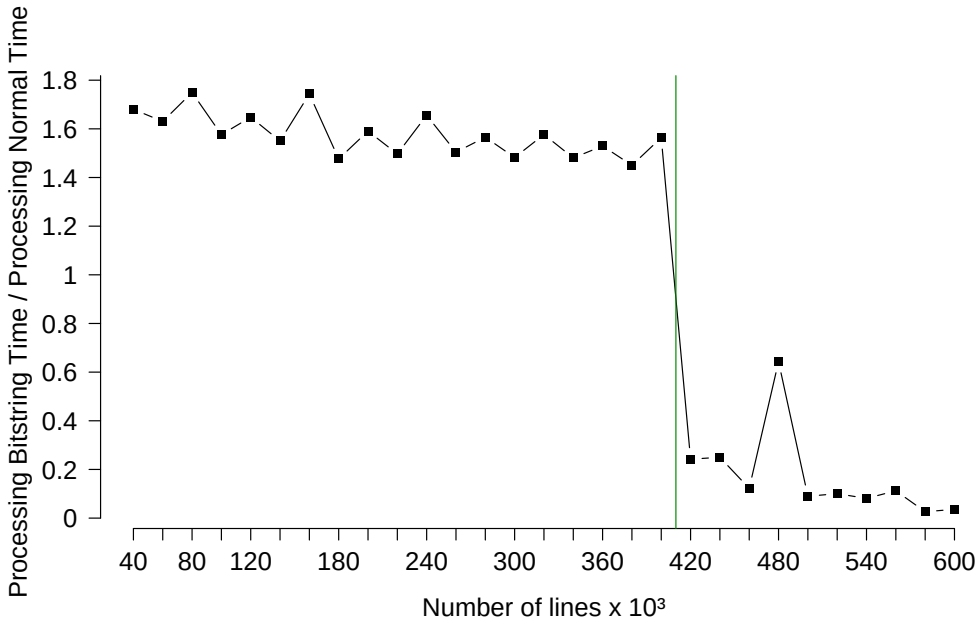


Figure 15. Relation between the processing time using the bitstring method and the traditional method in function of the line numbers of a matrix with 3,000 columns. It is possible to verify that for a matrix with less than 400 lines, the traditional method is faster. However, when the number of lines is greater than 400 lines, the best method is the bitstring because of the compressed representation in the matrix memory.

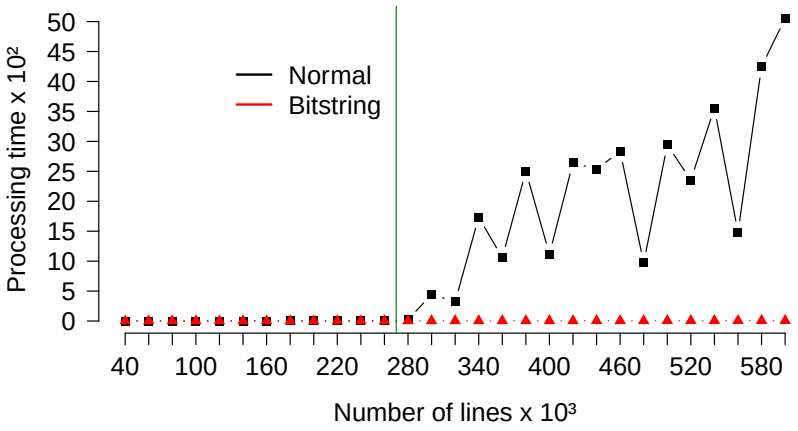


Figure 16. Processing time for the Bias From Mean method (BMF) in function of the numbers of linesof a matrix with 3,000 columns. The vertical green line indicates the approximate moment when the traditional method demands disk access in order to store the matrix to be analyzed, which overloads the process. The bitstring method fully represents a matrix in memory, without the necessity to access the disk.

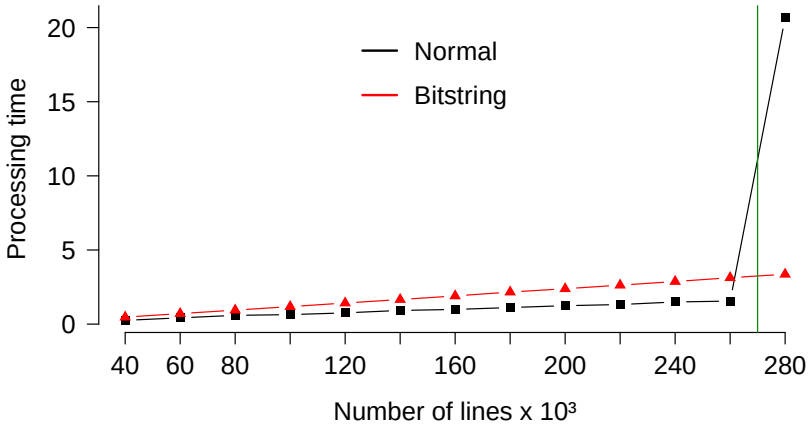


Figure 17. Processing time for the Bias From Mean method (BMF) in function of the numbers of lines of a matrix with 3,000 columns. The vertical green line indicates the approximate moment when the traditional method demands disk access in order to store the matrixto be analyzed, which overloads the process. For dimensions inferior to approximately 400 lines, the traditional method is faster than the bitstring one. It is important to emphasize that the computational cost of both methods is linear.

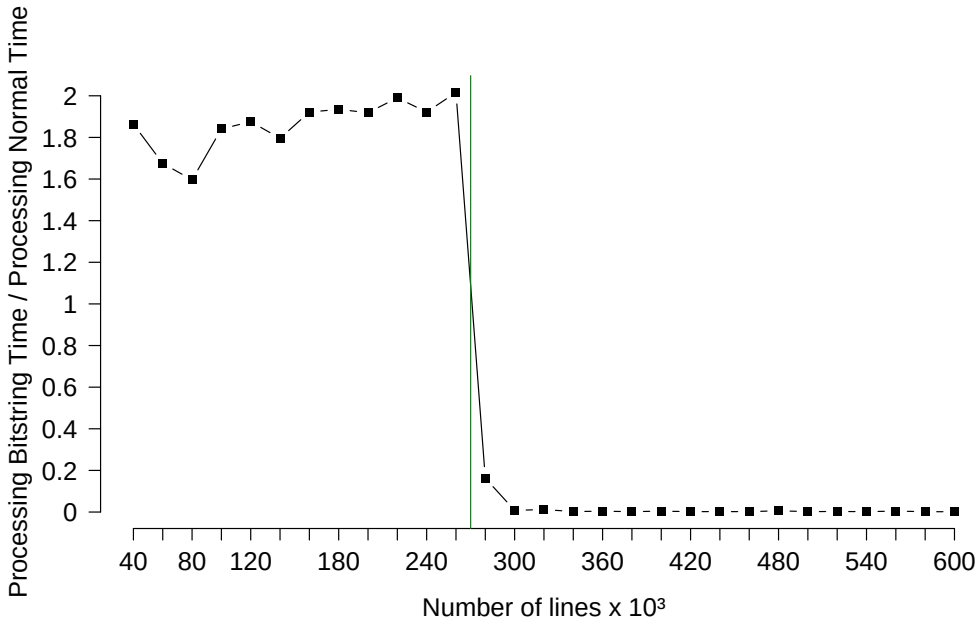


Figure 18. Relation between the processing time using the bitstring method and the traditional method in function of the line numbers of a matrix with 3,000 columns. It is possible to verify that for a matrix with less than 400 lines, the traditional method is faster. However, when the number of lines is greater than 400 lines, the best method is the bitstring because of the compressed representation in the matrix memory.

Tables

Table 1. The first 8 elements of M represented in binary base.

Element	Value	Binary	Bit length
$M_{1,1}$	900	1110000100	10
$M_{1,2}$	1023	1111111111	10
$M_{1,3}$	721	1011010001	10
$M_{1,4}$	256	100000000	9
$M_{1,5}$	1	1	1
$M_{1,6}$	10	1010	4
$M_{1,7}$	700	1010111100	10
$M_{1,8}$	20	10100	5

Table 2. Efficiency comparison of SM and VLB methods for parameters covering uniformly the support of B. Column n shows the number of parameter combinations with which each method has superior compression.

Methods	n	Percentage
SM	592	0.9034%
VLB	64944	99.0966%
Total	65536	100%

Table 3. Combinations p_1 and p_2 to calculate the efficiency.

p_1	p_2	η_2
0.0	1.0	-0.109
0.1	0.9	-0.010
0.2	0.8	0.087
0.3	0.7	0.185
0.4	0.6	0.284
0.5	0.5	0.382
0.6	0.4	0.481
0.7	0.3	0.579
0.8	0.2	0.678
0.9	0.1	0.776
1.0	0.0	0.875

Table 4. Compression efficiency of VLB method of samples with bit-lengths coming from a Discrete Uniform distribution $U(a = 1, b = 64)$. Average efficiency ($\overline{\eta_2} \pm \text{SD}$) were calculated over a 1,000 replicates.

	Matrix sizes		
	Sample size		
Expected bit-length	100	10,000	1,000,000
1	0.8750±0.0000	0.8750±0.0000	0.8750±0.0000
8	0.8202±0.0031	0.8203±0.0003	0.8203±0.0000
16	0.7580±0.0066	0.7578±0.0007	0.7578±0.0001
32	0.6330±0.0142	0.6329±0.0014	0.6328±0.0001
64	0.3826±0.0288	0.3828±0.0028	0.3828±0.0003

Table 5. Compression efficiency with bit-lengths distributed according to a binomial distribution $B(n, 0.5)$. Parameter $n \in \{1, 8, 16, 32, 64\}$ represents the maximum bit-length. Since $p = 0.5$ the expected bit-length is $n/2$ (first column)

	Efficiency		
	Sample size		
Expected bit-length (np)	100	1,000	1,000,000
1	0.8828±0.0004	0.8828±0.0004	0.8828±0.0000
8	0.8283±0.0022	0.8281±0.0002	0.8281±0.0000
16	0.7656±0.0032	0.7656±0.0003	0.7656±0.0000
32	0.6406±0.0045	0.6406±0.0004	0.6406±0.0000
64	0.3910±0.0065	0.3906±0.0006	0.3906±0.0001

Table 6. Compression efficiency with bit-lengths distributed according to a Poisson distribution (λ), where λ represents the expected bit-length.

	Efficiency η_2		
	Matrix Size		
Expected bit-length (λ)	100	1,000	1,000,000
1	0.8751±0.0015	0.8750±0.0004	0.8750±0.0000
8	0.7654±0.0046	0.7656±0.0005	0.7656±0.0000
16	0.6405±0.0064	0.6406±0.0006	0.6406±0.0001
32	0.3908±0.0087	0.3906±0.0009	0.3906±0.0001