# Low-Complexity Lossless Compression on High-Speed Networks

Sergio De Agostino
*Computer Science Department*
*Sapienza University*
*Rome, Italy*
*Email: deagostino@di.uniroma1.it*

***Abstract*—We present a survey of results on how to implement low-complexity lossless data compression on a high speed network, so that the computational phase requires no interprocessor communication. It follows that the computation in between the input and output phases has a linear speed-up when the network size increases, regardless of the bandwidth and latency of the network. Depending on the type of data, the performance of the compression method changes in terms of scalability. Images are more suitable than strings, since text compression is scalable only on very large size files.**

***Keywords-high speed network application; lossless compression; distributed algorithm; scalability.***

## I. INTRODUCTION

Arithmetic encoders enable the best lossless compressors by means of the model driven method [1]. The *model driven method* consists of two distinct and independent phases: *modeling* [2] and *coding* [3]. Arithmetic encoders are the best model driven compressors, but they are often ruled out because they are too complex. Low-complexity compression avoids arithmetic encoders.

Sliding window compression [4] is the most effective low-complexity text compression method (SW compression). When applied in parallel to data blocks on a large scale high speed network, the approach is practical only when the file size is large because of its adaptiveness [5].

Storer [6] extended SW compression to binary images by means of a square greedy matching technique (BLOCK MATCHING). The technique is suitable for high speed applications. Rectangle matching improves the compression performance, but it is slower since it requires $O(M \log M)$ time for a single match, where $M$ is the size of the match [7]. Therefore, the sequential time to compress an image of size $n$ by rectangle matching is $\Omega(n \log M)$. A variant of this method, called monochromatic pattern substitution (MP-SUB), compresses only monochromatic rectangles with a variable length code [8]. Such monochromatic rectangles are detected by means of a *raster* scan (row by row). If the 4 x 4 subarray in position $(i, j)$ of the image is monochromatic, then we compute the largest monochromatic rectangle in that position else we leave it uncompressed. The encoding scheme is to precede each item with a flag field indicating whether there is a monochromatic rectangle or raw data. The procedure for computing the largest monochromatic rectangle with left upper corner in position $(i, j)$ takes $O(M \log M)$ time, where $M$ is the size of the rectangle. The positions covered by the detected rectangles are skipped in the linear scan of the image. The analysis of the running time of this algorithm involves a *waste factor*, defined as the average number of matches covering the same pixel. We experimented that the waste factor is less than 2 on realistic image data. Therefore, the heuristic takes $O(n \log M)$ time in practice. On the other hand, the decoding algorithm is linear. The compression effectiveness of this technique is about the same as the one of the rectangular block matching technique [7]. Moreover, compression via monochromatic pattern substitution (MP-SUB compression) has no relevant loss of effectiveness if the image is partitioned into up to a thousand blocks and each block is compressed independently. Therefore, the computational phase can be implemented on both small and large scale distributed systems with no interprocessor communication. BLOCK MATCHING, instead, does not work locally since it applies the generalized SW-type method with an unrestricted window. Finally, MP-SUB compression has a speed-up if applied sequentially to the partitioned image [9]. Experimental results suggest that the speed-up happens if the image is partitioned into up to 256 blocks and sequentially each block is compressed independently. It follows that the speed-up can also be applied to a parallel implementation on a small scale system. Such speed-up depends on the fact that monochromatic rectangles crossing boundaries between blocks are not computed and, consequently, the waste factor decreases when the number of blocks increases. If we refine the partition by splitting the blocks horizontally and vertically, after four refinements experimentations show that no further improvement is obtained.

The extension of Storer's method to grey scale and color images was left as an open problem, but it seems not feasible since the high cardinality of the alphabet causes an unpractical subexponential blow-up of the hash table used in the implementation. A low-complexity application compressing 8x8 blocks of a grey-scale or color image by

 ISBN: 978-1-61208-231-8 

means of a header and a fixed-length code is presented in [10], which can be implemented on an arbitrarily large scale system with no interprocessor communication during the computational phase. A first step toward a good low-complexity compression scheme was FELICS (Fast Efficient Lossless Image Compression System) [11], which involves Golomb-Rice codes [12], [13]. With the same complexity level for compression (but with a 10 percent slower decompressor) LOCO-I (Low Complexity Lossless Compression for Images) [14] attains significantly better compression than FELICS. As explained in [10], parallel implementations of FELICS and LOCO-I require more sophisticated architectures than a simple array of processors.

As far as the model driven method for grey scale and color image compression is concerned, the modeling phase consists of three components: the determination of the context of the next pixel, the prediction of the next pixel and a probabilistic model for the *prediction residual*, which is the value difference between the actual pixel and the predicted one. In the coding phase, the prediction residuals are encoded. The use of prediction residuals for grey scale and color image compression relies on the fact that most of the times there are minimal variations of color in the neighborood of one pixel. Therefore, in [10] we were able to implement an extremely local procedure which is able to achieve a satisfying degree of compression by working independently on very small blocks. We presented the heuristic for grey scale images, but it can also be applied to color images by working on the different components. The main advantage is that it provides a highly parallelizable compressor and decompressor since it can be applied independently to each block of 8x8 pixels, achieving 80 percent of the compression obtained with LOCO-I (JPEG-LS), the current lossless standard in low-complexity applications. We called such procedure PALIC (Parallelizable Lossless Image Compression). The compressed form of each block employs a header and a fixed length code. Two different techniques might be applied to compress the block. One is the simple idea of reducing the alphabet size by looking at the values occurring in the block. The other one is to encode the difference between the pixel value and the smallest one in the block. This second technique can be interpreted in terms of the model driven method, where the block is the context, the smallest value is the prediction and the fixed length code encodes the prediction residual.

In Sections 2, 3 and 4, we explain the SW, MP-SUB and PALIC heuristics, respectively. The computational phase for these heuristics requires no interprocessor communication when implemented on a distributed system as, for example, a high speed network. It follows that the computation in between the input and output phases has a linear speed-up when the network size increases, regardless of the bandwidth and latency of the network. Quantitative results of such speed-up are provided in [5], [8], [9]. Conclusions and future work are given in Section 5.

## II. Text Compression

Sliding window (SW) compression [4] is based on string factorization. Each factor extends by one character the longest match with a substring to its left in the input string. SW compression is a dictionary-based technique and is also called the sliding dictionary method. In fact, the factors of the string are substituted by *pointers* to copies stored in a dictionary. Distributed algorithms for SW compression approximating in practice its compression effectiveness have been realized in [5] on an array of processor with no interprocessor communication. However, the scalability of a parallel implementation of SW compression on a distributed system with low communication cost guarantees robustness only on very large size files.

### A. SW Compression

Given an alphabet $A$ and a string $S$ in $A^*$ the factorization of $S$ is $S = f_1 f_2 \cdots f_i \cdots f_k$ where $f_i$ is the shortest substring, which does not occur previously in the prefix $f_1 f_2 \cdots f_i$ for $1 \leq i \leq k$. With such factorization, the encoding of each factor leaves one character uncompressed. To avoid this, a different factorization was introduced where $f_i$ is the longest match with a substring occurring in the prefix $f_1 f_2 \cdots f_i$ if $f_i \neq \lambda$, otherwise $f_i$ is the alphabet character next to $f_1 f_2 \cdots f_{i-1}$ [15]. $f_i$ is encoded by the pointer $q_i = (d_i, \ell_i)$, where $d_i$ is the displacement back to the copy of the factor and $\ell_i$ is the length of the factor. If $d_i = 0$, $l_i$ is the alphabet character. In other words a dictionary of factors is defined by a window sliding its right end over the input string, that is, it comprises all the substrings of the prefix read so far in the computation. The factorization processes just described are such that the number of different factors (that is, the dictionary size) grows with the string length. In practical implementations instead the dictionary size is bounded by a constant and the pointers have equal size. This can be simply obtained by bounding the match and window lengths (therefore, the left end of the window slides as well).

### B. Compression with Finite Windows

A real-time implementation of compression with finite window is possible using a suffix tree data structure [16], [17]. Much simpler real-time implementations are realized by means of hashing techniques providing a specific position in the window where a good approximation of the longest match is found on realistic data. In [18], the three current characters are hashed to yield a pointer into the already compressed text. In [19], hashing of strings of all lengths is used to find a match. In both methods, collisions are resolved by overwriting. In [20], the two current characters are hashed and collisions are chained via an offset array. Also the Unix gzip compressor chains collisions, but hashes three characters [21].

 ISBN: 978-1-61208-231-8 

### C. The High Speed Network Implementation

For every integer $k$ greater than 1 an O($kw$) time, O($n/kw$) processors distributed algorithm factorizing an input string $S$ was presented on an array of processors with no interconnections in [5], whose cost approximates the cost of the string factorization within the multiplicative factor $(k+m-1)/k$, where $n$, $m$ and $w$ are the lengths of the input string, the longest factor and the window respectively. The approach provides an approximation scheme for such string factorization problem since the multiplicative approximation factor converges to 1 when $kw$ converges to $n$.

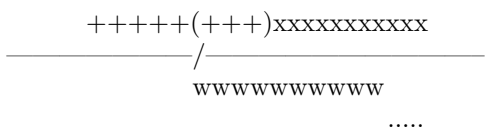


Figure 1. The making of the surplus factors.

We simply apply in parallel sliding window compression to blocks of length $kw$. It follows that the algorithm requires O($kw$) time with $n/kw$ processors and the multiplicative approximation factor is $(k+m-1))/k$ with respect to any parsing. In fact, the number of factors of a factorization on a block is at least $kw/m$ while the number of factors of the factorization produced by the scheme is at most $(k-1)w/m+w$. As shown in Figure 1, the boundary might cut a factor (sequence of plus signs) and the length $w$ of the initial full size window of the block (sequence of w's) is the upper bound to the factors produced by the scheme in it. Yet, the factor cut by the boundary might be followed by another factor (sequence of x's) which covers the remaining part of the initial window. If this second factor has a suffix to the right of the window, this suffix must be a factor of the sliding dictionary defined by it (dotted line) and the multiplicative approximation factor follows.

The approximation scheme is suitable for a small scale system but due to its adaptiveness it works on a large scale parallel system when the file size is large. From a practical point of view, we can apply something like the gzip procedure to a small number of input data blocks achieving a satisfying degree of compression effectiveness and obtaining the expected speed-up on a high speed network. Making the order of magnitude of the block length greater than the one of the window length largely beats the worst case bound on realistic data. The window length is usually several thousands of kilobytes. The compression tools of the Zip family, as the Unix command "gzip" for example, use a window size of at least 32K. It follows that the block length in our parallel implementation should be about 300K **at least. It follows that the file size should be at least about one third of the number of processors in megabytes.**

To decode the compressed files on the network, it is enough to use a special mark occurring in the sequence of pointers each time the coding of a block ends. The input phase distributes the subsequences of pointers coding each block among the processors.

## III. Binary Image Compression

Monochromatic pattern substitution (MP-SUB) is so far the only low-complexity lossless binary image compression technique implementable on a high speed network with no interprocessor communication during the computational phase and no scalability issues [8]. We describe sequential and parallel implementations of this technique in the following subsections.

### A. Monochromatic Pattern Substitution

The MP-SUB technique scans an image row by row. If the 4 x 4 subarray in position $(i,j)$ of the image is monochromatic, then we compute the largest monochromatic rectangle in that position. We denote with $p_{i,j}$ the pixel in position $(i,j)$. The procedure for finding the largest rectangle with left upper corner $(i,j)$ is described in Figure 2. At the first step, the procedure computes the longest possible width for a monochromatic rectangle in $(i,j)$ and stores the color in $c$. The rectangle 1 x $\ell$ computed at the first step is the current detected rectangle and the sizes of its sides are stored in $side1$ and $side2$. In order to check whether there is a better match than the current one, the longest sequence of consecutive pixels with color $c$ is computed on the next row starting from column $j$. Its length is stored in the temporary variable $width$ and the temporary variable $length$ is increased by one. If the rectangle $R$ whose sides have size $width$ and $length$ is greater than the current one, the current one is replaced by $R$. We iterate this operation on each row until the area of the current rectangle is greater or equal to the area of the longest feasible $width$-wide rectangle, since no further improvement would be possible at that point. Such procedure for computing the largest monochromatic rectangle in position $(i,j)$ takes O($M \log M$) time, where $M$ is the rectangle size. In fact, in the worst case a rectangle of size $M$ could be detected on row $i$, a rectangle of size $M/2$ on row $i+1$, a rectangle of size $M/3$ on row $i+2$ and so on.

If the 4 x 4 subarray in position $(i,j)$ of the image is not monochromatic, we do not expand it. The positions covered by the detected rectangles are skipped in the linear scan of the image. The encoding scheme for such rectangles uses a flag field indicating whether there is a monochromatic match (0 for the white ones and 10 for the black ones) or not (11). If the flag field is 11, it is followed by the sixteen bits of the 4 x 4 subarray (raw data). Otherwise, we bound by twelve the number of bits to encode either the width or the length of the monochromatic rectangle. We use either four or eight or twelve bits to encode one rectangle side. Therefore, nine

 ISBN: 978-1-61208-231-8 

$c = p_{r,j}$;
$r = i$;
$width = m'$;
$length = 0$;
$side1 = side2 = area = 0$;
**repeat**
    Let $p_{r,j} \cdots p_{r,j+\ell-1}$ be the longest string in $(r, j)$ with color $c$ and $\ell \leq width$;
    $length = length + 1$;
    $width = \ell$;
    $r = r + 1$;
    **if** $(length * width > area)$ {
        $area = length * width$;
        $side1 = length$;
        $side2 = width$;
    }
**until** $area \geq width * (i - k + 1)$ or $p_{r,j} <> c$

Figure 2. Computing the largest monochromatic rectangle match in $(i, j)$.

different kinds of rectangle are defined. A monochromatic rectangle is encoded in the following way:

- the flag field indicating the color;
- three or four bits encoding one of the nine kinds of rectangle;
- bits for the length and the width.

Four bits are used to indicate when twelve bits or eight and twelve bits are needed for the length and the width. This way of encoding rectangles plays a relevant role for the compression performance. In fact, it wastes four bits when twelve bits are required for the sides but saves four to twelve bits when four or eight bits suffice.

### B. The High Speed Network Implementation

The MP-SUB technique has been applied to the CCITT test set (Figure 3) and has provided a compression ratio equal to 0.13 in average. The images of the CCITT test set are 1728 x 2376 pixels. If these images are partitioned into $4^k$ sub-images and the compression heuristic is applied independently to each sub-image, the compression effectiveness remains about the same for $1 \leq k \leq 5$ with a 1 percent loss for $k = 5$. Raw data are associated with the flag field 110, so that we can indicate with 111 the end of the encoding of a sub-image. For $k = 6$, the compression ratio is still just a few percentage points of the sequential one. This is because the sub-image is 27 x 37 pixels and it still captures the monochromatic rectangles which belong to the class encoded with four bits for each dimension. These rectangles are the most frequent and give the main contribution to the compression effectiveness. The compression effectiveness of the variable-length coding employed by the technique depends on the sub-image size rather than on the whole image. In fact, if we apply the parallel procedure to the test set of larger binary images as the 4096 x 4096 pixels half-tone

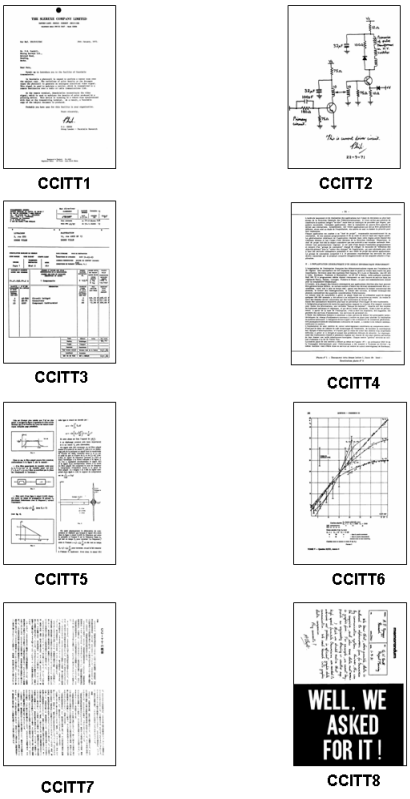


Figure 3. The CCITT image test set.

Figure 4. Images 1-5 from left to right.

topographic images of Figure 4, we obtain about the same compression effectiveness for $1 \leq k \leq 5$. The compression ratio is 0.28 with a 2 percent loss for $k = 6$. This means that actually the approach without interprocessor communication works in the context of unbounded parallelism as long as the elements of the image partition are large enough to capture the monochromatic rectangles encoded with four bits for each dimension.

### C. A Sequential Speed-Up

We experimented that if we partition an image into $4^k$ sub-images and apply compression via monochromatic pattern substitution to each sub-image independently the waste factor decreases with the increasing of $k$ [9]. A speed-up of the sequential algorithm follows from this fact. As mentioned in the introduction, the waste factor is less than 2 on realistic image data for $k = 0$ and decreases to about

 ISBN: 978-1-61208-231-8 

1 when $k = 4$. It follows that if we refine the partition by splitting the blocks horizontally and vertically, after four refinements no further relevant speed-up is obtained. The same happens when we partition the set of 4096 x 4096 pixels images of Figure 4, that is, the waste factor seems to be determined by the number of refinements independently from the image size on realistic data. Obviously, there is a similar speed-up for the decompressor.

Since the sequential speed-up happens for an image partitioned into less than 256 blocks, it can be applied to a parallel implementation on a small scale network. Obviously, a similar experiment could be run using two refinemets or one refinement of the partition on a network of 16 or 64 nodes respectively.

## IV. Grey Scale and Color Image Compression

We explain the PALIC heuristic which compresses grey scale and color images [10]. PALIC works independently on blocks of 8x8 pixels. The heuristic is described for grey scale images, but it can be trivially extended to RGB color images by working separately on each of the three components of the image. As previously mentioned, the compressed form of each block employs a header and a fixed length code.

### A. The Heuristic

We still assume to read the image with a raster scan on each block. The heuristic applies at most three different ways of compressing the block and chooses the best one. The first one is the following.

The smallest pixel value is computed on the block. The header consists of three fields of 1 bit, 3 bits and 8 bits, respectively. The first bit is set to 1 to indicate that we compress a block of 64 pixels. This is because one of the three ways partitions the block in four sub-blocks of 16 pixels and compresses each of these smaller areas. The 3-bits field stores the minimum number of bits required to encode in binary the distance between the smallest pixel value and every other pixel value in the block. The 8-bits field stores the smallest pixel value. If the number of bits required to encode the distance, say $k$, is at most 5, then a code of fixed length $k$ is used to encode the 64 pixels, by giving the difference between the pixel value and the smallest one in the block. To speed up the procedure, if $k$ is less or equal to 2 the other ways are not tried because we reach a satisfying compression ratio on the block. The second and third ways are the following.

The second way is to detect all the different pixel values in the 8x8 block, to create a reduced alphabet and to encode each pixel in the block using a fixed length code for this alphabet. The employment of this technique is declared by setting the 1-bit field to 1 and the 3-bits field to 110. Then, an additional three bits field stores the reduced alphabet size $d$ with an adjusted binary code in the range $2 \leq d \leq 9$. The last component of the header is the alphabet itself, a concatenation of $d$ bytes. Then, a fixed length code is used for the 64 pixels.

The third way compresses the four 4x4 pixel sub-blocks. The 1-bit field is set to 0. Four fields follow the flag bit, one for each 4x4 block. The two previous techniques are applied to the blocks and the best one is chosen. If the first technique is applied to a block, the corresponding field stores values from 0 to 7 rather than from 0 to 5 as for the 8x8 block. If such value is in between 0 and 6, the field stores three bits. Otherwise, the three bits (111) are followed by three more. This is because 111 is used to denote the application of the second way to the block as well, which is less frequent to happen. In this case, the reduced alphabet size stored in these three additional bits has range from 2 to 7, it is encoded with an adjusted binary code from 000 to 101 and the alphabet follows. 110 denotes the application of the first technique with distances expressed in seven bits and 111 denotes that the block is not compressed. After the four fields, the compressed forms of the blocks follow, which are similar to the ones described for the 8x8 block. When the 8x8 block is not compressed, 111 follows the flag bit set to 1. How the heuristic works on an example is shown in [10].

### B. The High Speed Network Implementation

The heuristic is obviously implementable on a large scale high speed network since an 8x8 pixels block is compressed independently. There is no issue in scaling down the network since one node can process more blocks sequentially [5]. On many images, we experimented positively the effectiveness of a less robust approach employing only the first way of compressing data, which shortens the coding and speeds up the process improving the compression efficiency.

### C. Decompression

Parallel decoding of compressed gray scale images is trivial. As mentioned at the beginning of this section, color images are compressed by applying the method to each of the three components. It is obviously better to apply the method to each component of a block rather than coding each component of the whole image. In this way, besides producing on on-line decodable compressed form we simplify the input phase of the high speed network implementation.

## V. Conclusion

We presented a survey describing three low-complexity lossless compression techniques for black and white images, color images and text respectively. These techniques can be implemented on a high speed network with no interprocessor communication. To garantee compression effectiveness and robustness, text compression requires each node of the network to store approximately 300 kilobytes of data while just 300 bytes suffice for black and white images. For color

 ISBN: 978-1-61208-231-8 

images, it is even enough that a node stores 64 bytes. It follows that, as far as text compression is concerned, scaling up the network is possible only for very large size files. As future work, the design of a more local low-complexity text compression technique is the main goal.

## REFERENCES


[1] T. C. Bell, J. G. Cleary, and I. H. Witten, *Text Compression,* Prentice Hall, 1990.

[2] J. Rissanen and G. G. Langdon, *Universal Modeling and Coding* IEEE Transactions on Information Theory 27, pp. 12-23, 1981.

[3] J. Rissanen, *Generalized Kraft Inequality and Arithmetic Coding* IBM Journal on Research and Development 20, pp. 198-203, 1976.

[4] A. Lempel and J. Ziv, *A Universal Algorithm for Sequential Data Compression,* IEEE Transactions on Information Theory 23, pp. 337-343, 1977.

[5] L. Cinque, S. De Agostino, and L. Lombardi, *Scalability and Communication in Parallel Low-Complexity Lossless Compression,* Mathematics in Computer Science 3, pp. 391-406, 2010.

[6] J. A. Storer, *Lossless Image Compression using Generalized LZ1-Type Methods* Proceedings IEEE Data Compression Conference, pp. 290-299, 1996.

[7] J. A. Storer and H. Helfgott H., *Lossless Image Compression by Block Matching* The Computer Journal 40, pp. 137-145, 1997.

[8] L. Cinque, S. De Agostino and L. Lombardi, *Binary Image Compression via Monochromatic Pattern Substitution: Effectiveness and Scalability* Proceedings Prague Stringology Conference, pp. 103-115, 2010.

[9] L. Cinque, S. De Agostino and L. Lombardi, *Binary Image Compression via Monochromatic Pattern Substitution: A Sequential Speed-Up* Proceedings Prague Stringology Conference, pp. 220-225, 2011.

[10] L. Cinque, S. De Agostino, F. Liberati and B. Westgeest, *A Simple Lossless Compression Heuristic for Grey Scale Images* International Journal of Foundations of Computer Science 16, pp. 1111-1119, 2005.

[11] P. G. Howard P. G. and J. S. Vitter, *Fast and Efficient Lossless Image Compression* IEEE Data Compression Conference, pp. 351-360, 1993.

[12] R. F. Rice, *Some Practical Universal Noiseless Coding Technique - part I* Technical Report JPL-79-22 Jet Propulsion Laboratory, Pasadena, California, USA, 1979.

[13] S. W. Golomb, *Run-Length Encodings* IEEE Transactions on Information Theory 12, pp. 399-401, 1966.

[14] M. J. Weimberger, G. Seroussi and G. Sapiro, *LOCO-I: A Low Complexity, Context Based, Lossless Image Compression Algorithm*, Proceedings IEEE Data Compression Conference, pp. 140-149, 1996.

[15] J. A. Storer and T. G. Szimansky, *Data Compression via Textual Substitution,* Journal of ACM 24, pp. 928-951, 1982.

[16] E. R. Fiala and D. H. Green, *Data Compression with Finite Windows,* Communications of ACM 32, pp. 490-505, 1988.

[17] E. M. Mc Creight, *A Space-Economical Suffix Tree Construction Algorithm,* Journal of ACM 23, pp. 262-272, 1976.

[18] J. R. Waterworth, *Data Compression System,* US Patent 4 701 745, 1987.

[19] R. P. Brent, *A Linear Algorithm for Data Compression,* Australian Computer Journal 19, pp. 64-68, 1987.

[20] D. A. Whiting, G. A. George, and G. E. Ivey, *Data Compression Apparatus and Method,* US Patent 5016009, 1991.

[21] J. Gailly and M. Adler, http://www.gzip.org, 1991 [retrieved: October, 2012].


 ISBN: 978-1-61208-231-8 