A Work-Optimal Parallel Implementation of Lossless Image Compression by Block Matching

Sergio De Agostino

School of Computing
Armstrong Atlantic State University
11935 Abercorn Street
Savannah, Georgia 31419
USA

e-mail: agos@armstrong.edu

Abstract. Storer suggested that fast encoders are possible for two-dimensional lossless compression by showing a square greedy matching LZ1 heuristic for bi-level images, which can be implemented by a simple hashing scheme [S96]. In this paper, we show a work-optimal parallel algorithm using a rectangle greedy matching technique requiring $O(\log M \log n)$ time on the PRAM EREW model, where n is the size of the image and M the maximum size of a rectangle.

Key words: lossless image compression, sliding dictionary, parallel algorithm, PRAM EREW

1 Introduction

Textual substitution compression methods (often called "LZ" methods due to the work of Lempel and Ziv [LZ76]) have been designed by Lempel and Ziv [LZ77, ZL78] and Storer and Szymanski [SS82]. These methods parse a string in *phrases* and replace them with *pointers* to copies, called *targets* of the pointers, that are stored in a *dictionary*. The encoded string is a sequence of pointers (some of which may represent single characters). *Static* methods are when the dictionary is known in advance. By contrast, with *dynamic* methods (LZ1 [LZ77] and LZ2 [ZL78]) the dictionary may be constantly changing as the data is processed (see [BCW90, St88] for references).

Storer [S96] and Storer and Helfgott [SH97] generalized the LZ1 method to lossless image compression and suggested that very fast encoders are possible by showing a square greedy matching LZ1 compression heuristic, which can be implemented by a simple hashing scheme and achieves 60 to 70 percent of the compression of JBIG1 on the CCITT bi-level image test set.

With LZ1 text compression, one simply proceeds from left to right making matches in greedy fashion between a substring in the current position and a copy in the part of the string already seen. A key advantage of LZ1 compression is that decoding is always simple and fast. Another advantage is that it is relatively easy to implement. The two key issues for practical implementations are how the encoder searches for matches and how pointers are encoded.

An image has to be scanned in some linear order. In order to achieve a good compression performance, bidimensional matches have to be computed. In [SH97], a square-match encoding algorithm is proposed using a simple hashing scheme directed to bi-level images. A 64K table with one position for each possible 4x4 subarray is the only data structure used. All-zero and all-one squares are handled differently. The encoding scheme is to precede each item with a flag field indicating whether there is a monochromatic square, a match or raw data. When there is a match, the 4 x 4 subarray in the current position is hashed to yield a pointer to a copy. This pointer is used for the current square greedy match and then replaced in the hash table by a pointer to the current position.

To improve the compression performance, it was also introduced a slower rectangle greedy matching technique requiring $O(M \log M)$ time where M is the size of the match [SH97]. Therefore, $O(n \log M)$ is the best sequential time for an image of size n if we compress by rectangle matching with M the maximum size of a rectangle.

Both heuristics work with an unrestricted window. In [CDG01] a rectangle greedy matching heuristic using a finite window and a bound to the match size was presented. The heuristic is suitable for a fast implementation similar to the one in [S96] and achieves 75 to 90 percent of the compression of JBIG1 on the CCITT bi-level image test set. In this paper, we show a work-optimal PRAM EREW implementation of lossless image compression by block matching requiring $O(\log M \log n)$ time which uses a rectangle greedy matching technique similar to the one in [CDG01]. The parallel heuristic achieves 95 to 97 percent of the compression of the sequential heuristic mentioned above [CDL02]. In section 2, we show how the sequential heuristic works. In section 3, we explain the parallel algorithm. In section 4, conclusions and future work are given.

2 The Rectangle Greedy Matching Technique

The compression heuristic scans an image n x m row by row (*raster* scan) (the greedy matching technique could work with any other scan described in [SH97]). 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) that matches a rectangle with left upper corner (k, h) is described in Fig. 1.

At the first step, the procedure computes the longest possible width for a rectangle match in (i, j) with respect to the position (k, h). The rectangle 1 x ℓ computed at the first step is the current rectangle match 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 one-dimensional match on the next row and column j, not exceeding the current width, is computed with respect to the row next to the current copy and to column h. 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 match, the current match is replaced by R. We iterate this operation on each row until the area of the current match is greater or equal to the area of the longest feasible $width$-wide rectangle, since no further improvement would be possible at that point. For example, in Fig. 2 we apply the procedure to find the largest rectangle match between position $(0, 0)$ and $(6, 6)$.

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

Figure 1: Computing the largest rectangle match in (i, j) and (k, h).

A one-dimensional match of width 6 is found at step 1. Then, at step 2 a better match is obtained which is 2 x 4. At step 3 and step 4 the current match is still 2 x 4 since the longest match on row 3 and 9 has width 2. At step 5, another match of width 2 provides a better rectangle match which is 5 x 2. At step 6, the procedure stops since the longest match has width 1 and the rectangle match can cover at most 7 rows. It follows that 5 x 2 is the greedy match since a rectangle of width 1 cannot have a larger area. Obviously, this procedure can be used for computing the largest monochromatic rectangle in a given position (i, j) as well. If the 4 x 4 subarray in position (i, j) is monochromatic, then we compute the largest monochromatic rectangle in that position. Otherwise, we compute the largest rectangle match in the position provided by the hash table and update the table with the current position. If the subarray is not hashed to a pointer, then it is left uncompressed and added to the hash table with its current position. The positions covered by matches are skipped in the linear scan of the image.

As pointed out in [SH97], for typical bi-level images this scheme is extremely fast for square matches and there is no significant slowdown over simply reading and writing the image. As mentioned in the introduction, in [SH97] it is shown that the rectangle greedy matching technique requires $O(M \log M)$ time where M is the size of the match. Therefore, $O(n \log M)$ is the best sequential time for an image of size n if we compress by rectangle matching with M the maximum size of a rectangle. The encoding scheme is to precede each item with a flag field indicating whether there is a monochromatic square (0 for white, 10 for black), a match (110) or raw data (111). Pointers are encoded with the straightforward encoding with three integers for x, y and size while a simple variable-length code is used to specify the size of a monochromatic square. We also mentioned in the introduction that a key issue for

0	0	1	0	1	1	0	1	0	0	0	0	1	1	1	step 1
0	0	1	1	1	0	0	1	0	0	0	0	0	1	0	step 2
1	0	1	1	1	0	0	1	0	1	0	0	1	1	1	step 3
0	1	1	0	1	1	0	0	0	0	0	0	0	1	1	step 4
0	1	1	0	1	0	0	1	0	0	0	0	0	0	1	step 5
0	0	0	0	1	1	0	1	1	0	0	0	1	1	1	step 6
0	0	1	0	1	1	0	0	1	0	1	1	1	1	1	step 1
0	0	1	0	1	1	0	0	1	1	0	0	1	1	1	step 2
0	0	1	0	1	1	1	0	0	0	0	0	1	1	1	step 3
0	0	1	0	1	1	0	1	0	0	0	0	1	1	1	step 4
0	0	1	0	1	1	0	1	0	0	0	0	1	1	1	step 5
0	0	1	0	1	1	0	1	0	0	0	0	1	1	1	step 6
0	0	0	0	1	1	0	1	1	0	0	0	1	1	1	

Figure 2: The largest match in (0,0) and (6,6) is computed at step 5.

practical implementations is how pointers are encoded. As pointed out in [SH97], good pointer coding schemes are important for text compression and become even more important for images since the number of matches that are used is typically less than the number found and the straightforward coding uses many more bits per pointer. With rectangular matches this issue becomes even more significant. The encoding of monochromatic rectangles is a dominant factor of the compression performance and the efficiency of the method increases with large images.

In [CDG01] we experimented our rectangle greedy matching algorithm with a bounded size dictionary defined by a *window* comprising the last 64K pixels read. We bounded by twelve the number of bits to encode either the width or the length of a rectangle match. We use either four or eight or twelve bits to encode one rectangle side. Therefore, nine different kinds of rectangle are defined. A pointer is encoded in the following way:

- the flag field indicating the type of item;
- if the item is not monochromatic, sixteen bits which are raw or indicating the position of the match in the window;
- three or four bits encoding one of the nine kinds of rectangle;
- bits for the length and the width.

Larger rectangles are less frequent but still relevant for the compression performance. Then, 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. On

the CCITT bi-level image test set, we achieved 75 to 90 percent of the compression of JBIG1.

3 The Parallel Algorithm

To achieve logarithmic time we partition an m x n image I in x x y rectangular areas where x and y are $\Theta(\log^{1/2} mn)$. In parallel for each area, one processor applies the sequential parsing algorithm so that in logarithmic time each area will be parsed in rectangles, some of which are monochromatic. We do not allow overlapping of the monochromatic rectangles when we apply the sequential algorithm to each area. Each processor could work with a sliding window of size 64K and bounded matches, using the same pointer encoding scheme described in the previous section. However, before encoding we wish to compute larger monochromatic rectangles. If we compute unbounded monochromatic rectangles, the coding for them could be the flag field, log m bits for the length and log n bits for the width.

In the description of the algorithm, we use four m x n matrices RC, CC, W and L which are determined by the parsing procedure on each area. $RC[i,j]$ and $CC[i,j]$ are equal to zero if $I[i,j]$ is not covered by a monochromatic rectangle, otherwise they are equal to the row and column coordinate of the left upper corner of the monochromatic rectangle. $W[i,j]$ and $L[i,j]$ are equal to zero if $I[i,j]$ is not covered by a monochromatic rectangle, otherwise they are equal to the width and length of the monochromatic rectangle. We also use four matrices TRC, TCC, TW and TR to store temporary values needed for the computation of the final parsing, which are initially set to RC, CC, W and L. The procedure to compute larger monochromatic rectangles works as in Fig. 3.

Basically, we try to merge monochromatic rectangles adjacent on the horizontal boundaries and then on the vertical boundaries, doubling in this way the length and width of each area at each step. It is always the rectangle of an area in odd position with respect either to the vertical or horizontal order which tries to merge with the adjacent rectangle in the next area. Generally, this merging operation causes that the rectangles split into two or three subrectangles. The rectangle from which we start the merging is split in at most two subrectangles since we want to preserve the upper left corner. The merging is realized by updating the temporary matrices storing the information on the monochromatic rectangles computed on the image. If we obtain a larger rectangle then we update the original matrices, otherwise we continue merging by working with the temporary values to see if we can get a larger rectangle later.

We describe the procedure more in details. At the first line internal to the main loop (Figure 3), we consider in parallel the left lower corners of monochromatic rectangles of the areas in odd positions which are adjacent to a monochromatic rectangle with the same color in the next area below. Then, at line 3 we change the width and length of the rectangle considered, where the length is the sum of the lengths of the two adjacent rectangles and the width changes if the right corners of the rectangle in the next area are to the left of the right corners of the other rectangle. At line 4 and 5 the values in the temporary matrices are changed also for the pixels in the next area since they merged. Obviously, these changes can be made with optimal work in logarithmic time. As mentioned above, the merging causes a splitting of rectangles into subrectangles and the values in the temporary matrices must be redefined also

until $x > m/2$ or $y > n/2$ do {
1 **in parallel for** i and j: $i = ax$, a odd, $j = TCC[i,j]$, $TRC[i+1,j] <> 0$ and $I[i,j] = I[i+1,j]$ {
2 **in parallel for** $TRC[i,j] \le k \le i$, $j \le h \le \min\{TCC[i+1,j] + TW[i+1,j] - 1, j + TW[i,j] - 1\}$ {
3 $TW[k,h] = \min\{TCC[i+1,j] + TW[i+1,j] - 1, j + TW[i,j] - 1\} - j + 1$;
3 $TL[k,h] = TL[k,h] + TL[i+1,j]$; }
4 **in parallel for** $i+1 \le k \le i+TL[i+1,j]$, $j \le h \le \min\{TCC[i+1,j]+TW[i+1,j]-1, j+TW[i,j]-1\}$
5 { $TCC[k,h] = j; TRC[k,h] = TRC[i,j]; TW[k,h] = TW[i,j]; TL[k,h] = TL[i,j]$; } }
6 **in parallel for** i and j: $TCC[i,j-1] + TW[i,j-1] = j, TCC[i,j] <> j, TCC[i,j] <>= 0$ or
7 $TCC[i,j] = j$ {
8 compute the smallest k with $k \ge j$: $TCC(i,k) + TW[i,k] > k+1, TCC[i,k+1] = k+1$ or
9 $TCC(i,k) + TW[i,k] = k+1$
10 **in parallel for** $j \le h \le k$ { $TCC[i,h] = j$; $TW[i,h] = k - j + 1$; } }
11 **in parallel for** i, j: $TRC[i,j] = i$ and $TCC[i,j] = j$
12 **if** $TW[i,j] * TL[i,j] \ge W[i,j] * L[i,j]$ **in parallel for** $i \le k < i + TL[i,j]$, $j \le h < j + TW[i,j]$ {
13 $RC[i,j] = TRC[i,j]$; $CC[i,j] = TCC[i,j]$; $W[i,j] = TW[i,j]$; $L[i,j] = TL[i,j]$; }
14 **in parallel for** i and j: $j = ay$, a odd, $i = TRC[i,j]$, $TRC[i,j+1] <> 0$ and $I[i,j] = I[i,j+1]$ {
15 **in parallel for** $TCC[i,j] \le h \le j$, $i \le k \le \min\{TRC[i,j+1] + TL[i,j+1] - 1, i + TL[i,j] - 1)\}$ {
16 $TL[k,h] = \min\{TRC[i,j+1] + TL[i,j+1] - 1, i + TL[i,j] - 1))\}$;
16 $TL[k,h] = TL[k,h] + TL[i+1,j]$; }
17 **in parallel for** $j+1 \le h \le j+TW[i,j+1]$, $i \le k \le \min\{TRC[i,j+1]+TL[i,j+1]-1, i+TL[i,j]-1)\}$
18 { $TRC[k,h] = i; TLC[k,h] = TLC[i,j]; TW[k,h] = TW[i,j]; TL[k,h] = TL = [i,j]$; } }
19 **in parallel for** i and j: $TRC[i-1,j] + TL[i-1,j] = i-1, TRC[i,j] <> i, TRC[i,j] <>= 0$ or
20 $TRC[i,j] = i$ {
21 compute the smallest k with $k \ge i$: $TRC(k,j) + TL[k,j] > k+1, TRC[k+1,j] = k+1$ or
22 $TRC(k,j) + TL[k,j] = k+1$
23 **in parallel for** $i \le h \le k$ { $TRC[h,j] = i$; $TW[h,j] = k - i + 1$; } }
24 **in parallel for** i, j: $i = TRC[i,j]$ and $j = TRC[i,j]$
25 **if** $TW[i,j] * TL[i,j] \ge W[i,j] * L[i,j]$ **in parallel for** $i \le k < i + TL[i,j]$, $j \le h < j + TW[i,j]$ {
26 $RC[i,j] = TRC[i,j]$; $CC[i,j] = TCC[i,j]$; $W[i,j] = TW[i,j]$; $L[i,j] = TL[i,j]$; }
27 $x = 2x$; $y = 2y$; }

Figure 3: How to compute monochromatic rectangles in parallel.

for the pixels covered by the other rectangles produced by the merging. This is done from line 6 to 10. In parallel we consider all the pixels for which, according to the temporary values, either they are not on the leftmost column of a rectangle and the adjacent pixels in front of them result to be on the rightest column of a rectangle (line 6) or they are on the leftmost column (line 7). For each of them, we compute the closest pixel to the right for which, according to the temporary values, either it is not on the rightmost column of a rectangle and the next pixel results to be on the leftmost column of a rectangle (line 8) or it is on the rightmost column of a rectangle (line 9). Being this pixel the closest to the one computed in lines 6 and 7, they must be on the rightmost and leftmost column of the same monochromatic rectangle respectively. This is redefined in the temporary matrices at line 10. At this point, for each left upper corner of a monochromatic rectangle (line 11) if we obtained a larger rectangle after the merging (line 12) we can ovewrite the information on the

new rectangle on the original matrices (line 13). Observe that this way of updating the matrices may introduce overlapping of the monochromatic rectangles. Then, we repeat the same procedure trying to merge rectangles horizontally (line 14–26).

To analize the complexity of the algorithm, it is enough to consider that at each iteration of the main loop we double the sides of the areas and to recall the classical parallel prefix computation technique. All the statements inside the loop require logarithmic time with optimal parallel work (lines 8–9 and 21–22 by parallel prefix). Since no operation is executed if there is nothing to merge, the total running time with optimal parallel work is $O(\log n \log M)$, where M is the maximum size of a monochromatic rectangle. Then, from the matrices we can easily derive the sequence of pointers with optimal parallel work and logarithmic time by parallel prefix.

4 Conclusions

In this paper, we showed a work-optimal parallel algorithm for lossless image compression by block matching using a rectangle greedy matching technique which requires $O(\log M \log n)$ time. The algorithm is suitable for an implementation on practical parallel architectures as meshes of trees, multigrids and pyramids.

As future work, a detailed study on how the algorithm must be implemented on these architectures could be provided. Also, practical parallel algorithms for decompression should be designed.

References

[BCW90] Bell T.C., Cleary J.G., Witten I.H: Text Compression. Prentice Hall.

[CDG01] Cinque L., De Agostino S., Grande E: LZ1 Compression of Binary Images using a Simple Rectangle Greedy Matching Technique. IEEE Data Compression Conference, 492.

[CDL02] Cinque L., De Agostino S., Liberati F.: A Parallel Algorithm for Lossless Image Compression by Block Matching. IEEE Data Compression Conference, 450.

[LZ76] Lempel A., Ziv J: On the Complexity of Finite Sequences. IEEE Transactions on Information Theory, 22, 75-81.

[LZ77] Lempel A., Ziv J: A Universal Algorithm for Sequential Data Compression. IEEE Transactions on Information Theory, 23, 337-343.

[St88] Storer J.A.: Data Compression: Methods and Theory. Computer Science Press.

[S96] Storer J. A.: Lossless Image Compression using Generalized LZ1-Type Methods. IEEE Data Compression Conference, 290-299.

[SH97] Storer J. A., Helfgott H.: Lossless Image Compression by Block Matching. The Computer Journal, 40, 137-145.

[SS82] Storer J. A., Szymanski T. G.: Data Compression via Textual Substitution. Journal of ACM, 29, 928-951.

[ZL78] Ziv J., Lempel A.,: Compression of Individual Sequences via Variable Rate Coding. Transactions on Information Theory, 24, 530-536.