Approaches to Zerotree Image and Video Coding on MIMD Architectures

Rade Kutil
Dept. of Scientific Computing, University of Salzburg, Austria

Abstract

The wavelet transform is more and more widely used in image and video compression. One of the best known algorithms in image compression is the SPIHT algorithm which involves the wavelet transform. As today the parallelisation of the wavelet transform is sufficiently investigated, this work deals with the parallelisation of the compression algorithm itself as a next step. Two competitive approaches are presented: One is a direct parallelisation and the other uses an altered algorithm which suits better to the parallel architecture.

keywords: MIMD, zerotree, video coding, wavelets

1 Introduction

Image and video coding methods that use wavelet transforms have been successful in providing high rates of compression while maintaining good image quality and have generated much interest in the scientific community as competitors to DCT based compression schemes in the context of the MPEG-4 and JPEG2000 standardisation processes.

Most video compression algorithms rely on 2-D based schemes employing motion compensation techniques. On the other hand, rate-distortion efficient 3-D algorithms exist which are able to capture temporal redundancies in a more natural way [10, 5, 4, 16, 1]. Unfortunately, these 3-D algorithms often show prohibitive computational and memory demands (especially for real-time applications). Therefore, MIMD architectures seem to be an interesting choice for such an algorithm.

A significant amount of work has already been done on parallel wavelet transform algorithms for all sorts of high performance computers. We find various kinds of suggestions for 1-D, 2-D and 3-D algorithms on MIMD computers for decomposition only [8, 19, 15, 13, 3, 6] as well as in connection with image compression schemes [9, 2].

Zero-tree based coding algorithms efficiently encode approximations of wavelet coefficients by encoding collections of neglectable (insignificant) coefficients through single symbols. These collections are called zero-trees because of the tree-like arrangement of wavelet coefficients which exploits the wavelet transform's self-similarity property.

A parallelisation of the EZW algorithm – an important zero-tree coding scheme – is presented in [2] where two approaches are proposed: One is a straight-forward parallelisation which performs the EZW algorithm locally on each processing element (PE) for distinct blocks. This makes the resulting bit-stream (BS) incompatible with the sequential algorithm. The other approach reserves one PE for the collection of the symbols that have to be encoded. This PE performs a reordering of the symbols before it encodes them. This approach is similar to the approaches presented in this work and (as far as the author understands) is compatible with the sequential EZW.

An improvement of the EZW is the SPIHT algorithm (Set Partitioning In Hierarchical Trees [14]). It is a well known, fast and efficient algorithm which can also be used as 3D-variant in video compression [5]. However, this algorithm makes use of lists of coefficients, which makes it hard to parallelise. This means that although consecutive list entries initially point to neighbouring wavelet transform coefficients, the algorithm jumbles the entries after a while. Thus, there is no easy data driven parallelisation.

An approach to *packetise* the SPIHT algorithm is presented in [17]. This is similar to the first approach in [2]. The output of several executions of SPIHT on spacial blocks is multiplexed based on a rate allocation technique that approximates distortion reduction by fast SPIHT-specific statistics. However, the resulting bit-stream (BS) is not compatible with sequential SPIHT.

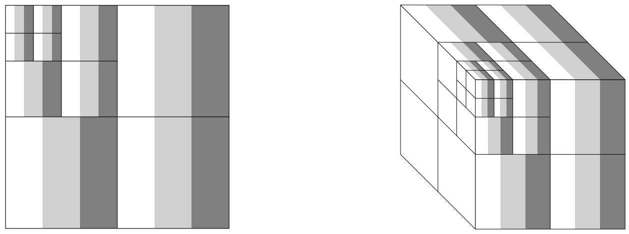
This work concentrates on the parallelisation of the SPIHT algorithm and presents two competitive approaches. The first is a direct parallelisation, i.e. the sequential algorithm is mapped to the parallel architecture without alteration of the *sequential* algorithm. Several "tricks" have to be found to overcome involved parallelisation difficulties. The second approach introduces a variant of SPIHT that involves a more spacially oriented coefficient scan order and, thus, avoids the problems of the first approach. Similar algorithms are proposed in [18, 11, 12]. Although a breadth-first scan order through the trees of coefficients [18] is PSNR-optimal, a depth-first scan order [11, 12] is preferred here because of better spacial separability.

1.1 Parallel Wavelet Transform

The fast wavelet transform can be efficiently implemented by a pair of appropriately designed Quadrature Mirror Filters (QMF) consisting of a low-pass and a high-pass filter which decompose the original data set into two frequency-bands. These sub-bands are down-sampled by 2 and the same procedure is recursively applied to the coarse scale (low-pass filtered) sub-band. In the 2-D case consecutive filtering of rows and columns produces four sub-bands (eight in the 3-D case). Only the one which is filtered with the low-pass filter in each dimension is decomposed further. This is called the pyramidal wavelet transform.

To perform the wavelet transform in parallel, the data has to be distributed among the PEs in some way. In this work data is splitted into slices (in the time domain). The filtering is performed in parallel on local data. Border data has to be exchanged before each decomposition step between neighbouring PEs due to the filter length. After that, transformed data is found distributed as shown in Figure 1.

In contrast to the parallelisation of the wavelet transform as presented in a previous paper[7], the parallel wavelet transform used here dispenses with video data distribution as well as collection of transformed data. Initial data distribution is not necessary because input is performed in parallel (i.e. each PE reads its own part of the video data). Note that the speedups reported in this work do not include I/O operations as I/O is not viewed as a part of the algorithm. Another advantage of this is that we can drop the host-node paradigm because there is no extra single PE responsible for data distribution. Likewise, the collection of transformed data is not necessary because data is passed on to the coding part of the algorithm which is also performed in parallel. There is no redistribution of data required as we will see in section 2.2 and 3.2.



(a) 2-D case (b) 3-D case

Figure 1: Distribution of coefficients or list entries after parallel wavelet transform. Different colours indicate different PEs

1.2 Zero-Trees

Zero-tree based algorithms arrange the coefficients of a wavelet transform in a tree-like manner (as in Figure 2), i.e. each coefficient has a certain number of child coefficients in another sub-band (mostly 4 in the 2-D, 8 in the 3-D case). We will use the following notations:

off(p) The direct offspring of a coefficient p, i.e. all coefficients whose parent coefficient is p.

desc(p) All descendants of a coefficient p. This includes $\text{off}(p)$, $\text{off}(\text{off}(p))$ and so on.

parent(p) The parent coefficient of p. $p \in \text{off}(\text{parent}(p))$.

Furthermore, a zero-tree is a sub-tree which entirely consists of insignificant coefficients. The significance of a coefficient is relative to a threshold which plays an important role in the SPIHT algorithm:

$$\text{sig}(p) \Leftrightarrow |p| \geq \text{threshold} .$$

The statistical properties of transformed image or video data (self-similarity) ensures the existence of many zero-trees. Sets of insignificant coefficients can be encoded efficiently with the help of zero-trees. We will see that sometimes the root coefficient of the subtree (or even its direct offspring) does not have to be insignificant.

Zero-trees can be viewed as a collection of coefficients with approximately equal spacial position. While this fact implies that the coefficient's significances are statistically related, which is exploited by the SPIHT algorithm, this also means that zero-trees are local objects, corresponding to the data distribution produced by the parallel wavelet transform (see Figure 2). This can be exploited by the parallelisation of the zero-tree algorithms (see Section 2.2 and 3.2).

2 Parallelisation without Algorithm Alteration

2.1 The SPIHT Algorithm

Although the SPIHT algorithm is sufficiently explained in the original paper[14], it is helpful in this context to reformulate the algorithm.

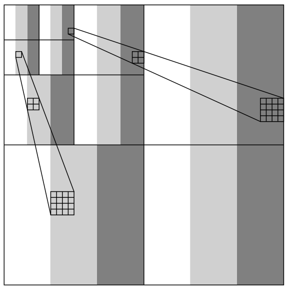
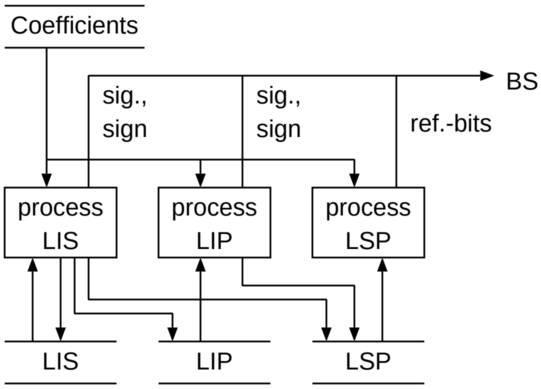


Figure 2: After parallel decomposition, data is distributed in a way so that each zero-tree resides on a single PE

threshold $\leftarrow \max_{p \in \text{allcoefficients}} \mid p \mid$
fill LIS, LIP with approximation subband
set LSP empty
for each refinement step
    threshold $\leftarrow$ threshold $/2$
    process LIS
    process LIP
    process LSP

(a) Pseudo code



(b) Data flow graph

Figure 3: The SPIHT algorithm

Significance information is represented by three lists:

LIS List of insignificant set of pixels
An entry in this list can be of two types:
Type A: $p \in \text{LIS}_A \Leftrightarrow \neg\,\text{sig}(\text{desc}(p)) \wedge p \notin \text{desc}(\text{LIS})$
Type B: $p \in \text{LIS}_B \Leftrightarrow \neg\,\text{sig}(\text{desc}(\text{off}(p))) \wedge p \notin \text{desc}(\text{LIS})$

LIP List of insignificant pixels
$p \in \text{LIP} \Leftrightarrow \neg\,\text{sig}(p) \wedge p \notin \text{desc}(\text{LIS}_A) \wedge p \notin \text{desc}(\text{off}(\text{LIS}_B))$

LSP List of significant pixels
$p \in \text{LSP} \Leftrightarrow \text{sig}(p)$

The LIS basically contains all zero-tree roots. An entry of type A corresponds to an insignificant sub-tree without its root. An entry of type B corresponds to an insignificant sub-tree without its root and the root's direct offspring. The LIP contains all insignificant coefficients that are not part of any zero-tree in the LIS. The LSP contains all significant coefficients.

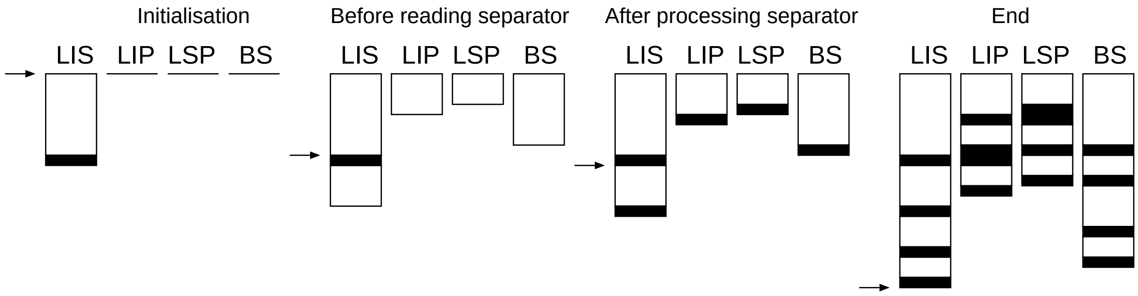


Figure 4: Functionality of separators. Four states of the three lists and the bit-stream while processing the LIS.

The algorithm is shown at a coarse level in Figure 3. Initially, the threshold is greater than all coefficients. Thus, the LIS and the LIP are filled with the approximation sub-band's coefficients, and the LSP is empty. After that, each entry of each list has to be tested for a change of significance, and the result of the test has to be encoded as a bit in the bit-stream (BS). If, for instance, a type A entry of the LIS turns out not to to be insignificant any more (to be precise: its descendants), a zero bit has to be written into the bit-stream, and the entry has to be deleted from the LIS, inserted as a type B entry at the end of the LIS and its direct offspring has to be inserted at the end of the LIP. All entries inserted at the end of a list are also processed in the same refinement step until no more entries are left. Figure 3(b) shows a data flow graph for a refinement step.

The decoding process performs the same algorithm. It does not, however, evaluate the significance of the list entries, but simply reads this information from the bit-stream and approximates the value of the corresponding coefficient as good as it can.

2.2 SPIHT Parallelisation

When parallelising the SPIHT algorithm, we have to face the problem that it uses lists of coefficient positions and is, therefore, inherently sequential. The reason is that it is hard to perform general list operations on distributed lists. Nevertheless, the set of distinct list operations involved in the SPIHT algorithm is limited. This enables us to develop an efficient way to manage distributed lists of coefficients.

2.2.1 Separators

The basic operations of the algorithm are: Moving an iterator all through a list, deleting elements at iterator position and appending elements at the end of a list. The aim is to distribute the list so that each PE-local entry corresponds to a local coefficient where coefficients are distributed among the PEs as shown in Figure 1.

This is a simple task for initial distribution. However, as coefficients are appended to the end of lists, one has to provide a mechanism to indicate which parts of a list belong to which PE – or from a PE's view: where a sequence of local coefficients ends and parts of another PE's list should be inserted. This work is done by separators (see Figure 4).

The idea is to insert a separator at the end of each part of the list which entirely belongs to a single PE. Initially, the approximation sub-band is split into equal slices. Each slice is assigned to a single PE. On each PE, the (local) lists LIS and LIP are filled with all coefficients from the PE's (local) slice, and a separator is appended to the end of each list. From here on, the sequential algorithm is performed locally with one exception: Each time the iterator meets a separator, the separator is copied to the end of each destination list. A destination list is a list into which entries are potentially inserted during the current list processing (see Figure 3(b)). Applying this principle, the lists L_i on PE$_i$ are split by separators into parts L_{ij} such that the assembled list $L_{11}L_{21}L_{31}\dots L_{12}L_{22}\dots$ is identical to the list the sequential algorithm would produce. The same is true for the bit-stream. This enables the parallel algorithm to assemble the bit-stream correctly after each PE has encoded its part of the wavelet coefficients.

An important question is when the processing of a list (i.e. a refinement step) is completed. Essentially, the procedure can stop if it has processed the last non-separator entry in the list. Unfortunately, this does not guarantee that each PE produces the same number of separators. However, this is a necessary condition for the correctness of the parallel algorithm because, otherwise, the correct order of the list-parts would be lost. Therefore, the global maximum number of separators has to be calculated (which unfortunately synchronises the PEs), and the lists have to be filled up with separators before the algorithm continues with the next refinement step (in fact even before each processing of a list).

As a matter of fact, the number of separators grows exponentially with the number of refinement steps, and very often separators appear in a row together in the list. To avoid unnecessary memory demands, consecutive separators should be kept together in a single entry associated with a counter. This means, if a separator entry (containing a counter) is inserted at the end of a list where a separator entry (also containing a counter) is already sitting, their counters can simply be added.

2.2.2 Algorithm Termination

Another problem is the termination of the whole algorithm. In the sequential case, the process terminates when the required number of bits have been written to the bit-stream. In the parallel case, this test (as it is a global test) can only be executed at the end of a refinement step. Thus, the parallel algorithm potentially generates too much bits which of course decreases the speedup in inconvenient cases. If necessary, superfluous bits can simply be cut off after assembling the bit-stream (due to the nature of the SPIHT algorithm).

The procedure of assembling the bit-stream (after collecting the PE-local bit-streams) is the only sequential part of the algorithm. Unfortunately, it gets more complicated and, therefore, consumes more calculation time when the number of PEs is increased. The result will be a significant decrease in speedup.

2.2.3 Parallel SPIHT Decompression

Note that for the reverse algorithm – the reconstruction of video data from the bit-stream – the methods described above are not applicable. Although it is not part of this work, we can shortly outline the ideas how to implement a parallel SPIHT decoder: First of all, the whole bit-stream has to be copied to all PEs. Also, each PE has to process all bits of the bit-stream, independent of whether they belong to local coefficients. Therefore, the global lists have to be kept at each PE. The only speedup potentials are:

- Not adjusting non-local coefficients which involves floating point operations.

$$
\begin{aligned}
A(p) &\Leftrightarrow \mid p \mid \geq \text{threshold} \\
B(p) &\Leftrightarrow \bigvee_{q \in \mathrm{desc}(p)} A(q) \\
C(p) &\Leftrightarrow \bigvee_{q \in \mathrm{off}(p)} B(q)
\end{aligned}
$$

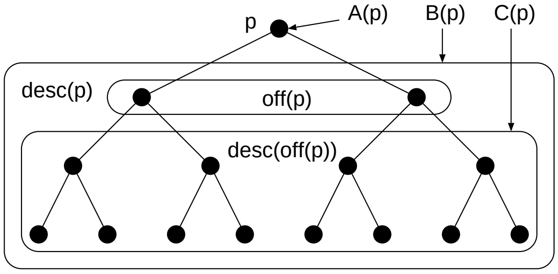


Figure 5: Predicates used in the algorithm

- Keeping consecutive non-local list entries (entries belonging to non-local coefficients) together in a single entry associated with a counter (similar to separators). This is possible because position information is not needed for non-local entries.

Although this seems to be a very simple approach, it does not imply the necessity of process synchronisation and it does not contain a sequential part.

3 Parallelisation with Algorithm Alteration

The approach described above reveals some drawbacks as e.g. complicated bit-stream handling, additional communication needs and non-neglectable sequential code parts. This is a direct consequence of the fact that the SPIHT algorithm is inherently sequential.

Therefore, we will modify the sequential algorithm itself. Although the resulting bit-stream will not be compatible with SPIHT, the parallelisation of the altered algorithm will, of course, be compatible with its sequential version. The basic idea is to substitute the lists of coefficient positions involved in the algorithm by bitmaps indicating the membership of each coefficient to a certain list. As a result, list iteration, which is used frequently to process the list entries, is turned into a normal scan of coefficients that follows a certain spacial direction. Thus, the data driven parallelisation can be performed more easily by a loop parallelisation of the coefficient scan.

3.1 Zero-Tree Compression with Significance Maps (SM)

In the following, we will use three logical predicates $A(p)$, $B(p)$ and $C(p)$ which are defined as in Figure 5. $A(p)$ simply denotes the significance of the coefficient p. $B(p)$ is true if and only if at least one of p's descendants is significant, while $C(p)$ denotes the same but does not include the direct offspring of p. This is visualised in Figure 5. The state of significance of a given set of coefficients can be described by these predicates in terms of zero-trees. Corresponding to these predicates, we will use the mappings a, b and c which essentially represent the same as A, B and C. The difference is that A, B and C immediately change their values if the threshold is changed and are, therefore, implemented as a function/procedure in the used programming language. a, b and c have to be updated explicitly and are, therefore, implemented as array of Boolean values. We will call a, b and c "significance maps" (SM). They substitute the lists of coefficients (see section 2.1).

The algorithm is responsible for the equality of a, b, c and A, B, C respectively while the

```
ProcessAll :=
    threshold ← max_{p∈allcoefficients} | p |
    set a, b and c to all false
    for each refinement step
        threshold ← threshold /2
        for p in approximation-subband
            ProcessCoeff(p, true)
```

```
ProcessCoeff(p, ĉ) :=
    if a_p then Refine(p) else a_p ← A(p)
    if ¬b_p ∧ ĉ then b_p ← B(p)
    if b_p ∧ ¬c_p then c_p ← C(p)
    if b_p then
        for q in off(p)
            ProcessCoeff(q, c_p)
```

Figure 6: SM based Zero-tree coding algorithm

threshold is successively decreased by a factor of $\frac{1}{2}$. This should be done by avoiding the evaluation of A, B and C as far as possible because – following the idea of the SPIHT algorithm – the result of each evaluation will be encoded into the bit-stream as one bit to allow the decoder to reproduce the decisions the encoder has made.

The algorithm that obeys these rules is shown in Figure 6. The outer loop is the refinement loop which divides the threshold by 2 in each iteration. This is exactly the same as in the original SPIHT algorithm. Within this loop, the algorithm navigates through the set of coefficients along trees of coefficients in a depth-first manner (this is the major difference between the SM based algorithm and SPIHT) starting at the set of coefficients contained in the approximation sub-band. This is accomplished by the recursive procedure "ProcessCoeff".

For each coefficient p, the state of a, b and c is checked one after another: $a_p \leftarrow A(p)$ has to be evaluated only if a_p is false because the transition true $\mapsto$ false is not possible for a_p. If $A(p)$ is true then the sign of p has to be encoded as well. If a_p is already true the procedure "Refine" is called which en/decodes another bit of the coefficients value to refine the decoded approximation. $b_p \leftarrow B(p)$ has to be evaluated only if, again, b_p is false *and* $\hat{c} = C(\text{parent}(p))$ is true because $B(p) \Rightarrow C(\text{parent}(p))$. $c_p \leftarrow C(p)$ has to be evaluated only if c_p is false *and* b_p is true because $C(p) \Rightarrow B(p)$. At last, the recursion to the child coefficients only has to be performed if b_p is true (for an obvious reason).

The decoding algorithm looks exactly the same. The only difference is that instead of encoding the results of the evaluation of A, B and C, this information is read from the bit-stream. Together with the sign- and refinement-bits, this is enough information to enable the decoder to perform the same steps as the encoder and approximate the coefficients with an error below the threshold.

Note that this algorithm encodes the same information (in fact the same bits) as the SPIHT algorithm. The order of the bits is the only difference. This means that at the end of each refinement step, the compression performance is equal to that of SPIHT. In between, the order of the bits that are written to the bit-stream is crucial because the bits can have different effect on the decoded image. Thus, it is important to encode bits with greater effect first. Figure 7 shows the comparison of the PSNR-performance for the well-known "Lena"-image (2-D case). The algorithm shows major drawbacks with respect to the original SPIHT which, nevertheless, can almost be overcome by scanning the set of coefficients in several passes. The first pass should only process those coefficients that are not part of zero-trees but not significant. Subsequent passes check the state of zero-tree roots and process those coefficients that emerge from decomposed zero-trees. This method is denoted "sophisticated" in Figure 7. Nevertheless, this improvement of the algorithm is not used in this work in parallelisation investigations.

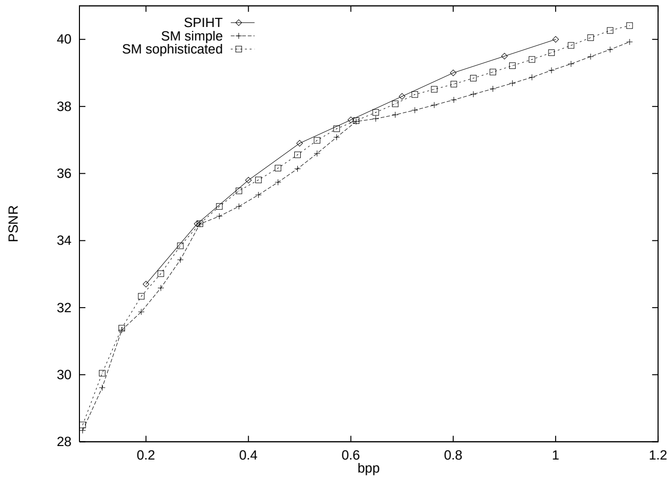


Figure 7: PSNR of the SM based algorithm compared to the original SPIHT

3.2 Parallelisation

In contrast to the original SPIHT algorithm, the parallelisation of the SM based algorithm is easy. Again, it is based on the fact that after the parallel wavelet transform, data is distributed in a way so that zero-trees are local objects (see Figure 2).

All we have to do is to parallelise the inner loop in the procedure ProcessAll (which reads "for p in approximation-subband") according to the data distribution of the approximation sub-band (see Figure 1). All other computations within a refinement step are localised, i.e. PE-local computations do not depend on data of neighbouring PEs. So, no communication is required within a refinement step.

Each PE produces one continuous part of the bit-stream for each refinement step. At the end, these parts have to be collected by a single PE and assembled properly (i.e. in an alternating way). As in the direct SPIHT parallelisation, this is a major bottleneck. However, the number of bit-stream parts is reduced significantly, which should speed up the bit-stream assembly.

Because of the termination problem (see section 2.2) the PEs again have to synchronise at the end of each refinement step to determine if the global number of bits produced so far is sufficient. However, this synchronisation can be dropped if the termination condition is not the bit-stream size but a fixed number of refinement steps.

4 Experimental Results

Experimental results were conducted on a Cray T3E-900/LC at the Edinburgh Parallel Computing Centre using MPI. Video data size is always 864 frames with 88 by 72 pixels. The video sequence used here is the U-part of "grandma". The wavelet transform is performed up to a level of 3.

Note that the data size is limited by memory constraints in the case when the number of PEs is 1 and a single PE has to hold all video data. The number of frames has to be high to enable uniform data distribution for parallelisation as well as down-scaling for the wavelet transform.

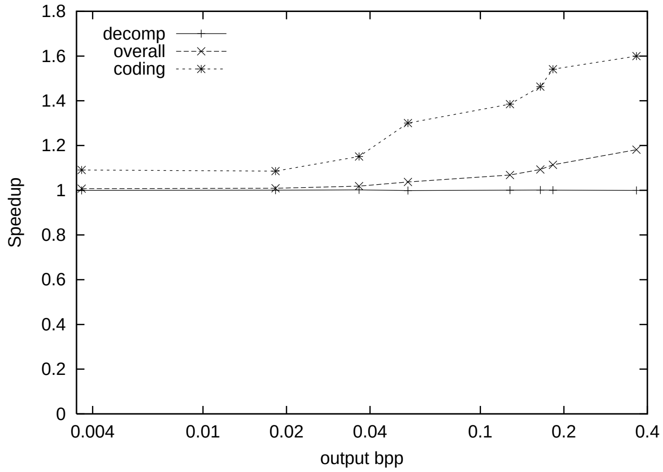


Figure 8: Sequential speedup of the SM based algorithm with respect to SPIHT

Thus, the frame size is small. In a real-world scenario, however, the frame size can be bigger. For the same reason, frame size scalability is difficult to measure. Nevertheless, linear scalability is assumed due to the authors experiences and the fact that the execution time of the coding part does not depend on the video data size but on the number of output bits only.

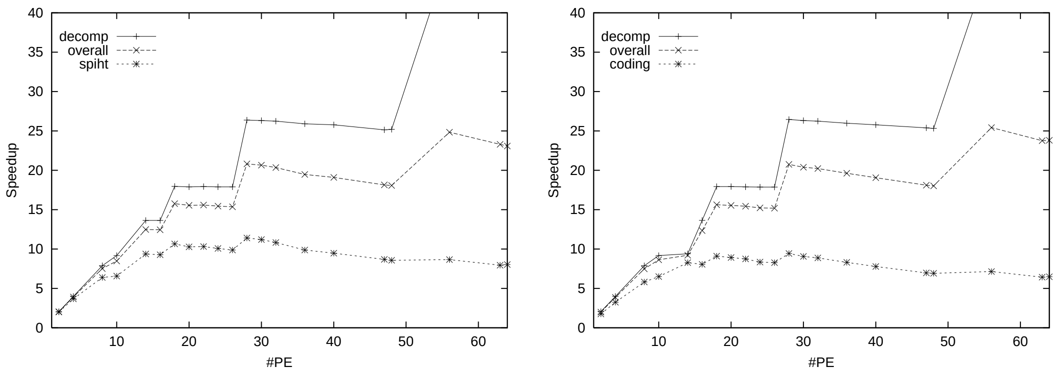
First of all, we have to look at the sequential performance of the SM based algorithm because if it was slower than the original SPIHT, its parallelisation would not make sense. However, Figure 8 shows that it outperforms the original SPIHT especially for higher bit-rates. On the other hand, this means that it is even harder to get reasonable speedups.

Figure 9 shows speedups for a fixed compression rate: 0.14 bpp (bits per pixel, pixels in different frames are counted as different pixels). The fact that the speedup curves are not smooth, i.e. have discontinuities at #PE = 9, 18, 27 and 54, is caused by the divisibility of 864 – the length of the video sequence which determines the size of the local data sub-sets. Note that due to the depth of the wavelet transform (3), this size is divided by 8 and the resulting number is then divided by the number of PEs which is not always possible without remainder.

The sequential bit-stream assembly takes more and more execution time for higher numbers of PEs. Its share in execution time gets higher than 50% of the coding part. Note at this point that in a particular hardware implementation, the bit-stream assembly can be integrated in the output module and separated from the actual coding.

The speedups of the two different algorithms are about the same. This shows that the complicated bit-stream assembly - which is the main problem of a direct SPIHT parallelisation - could be solved efficiently. Some dominant problems, such as PE synchronisation and sequential code parts, are present in both approaches. However, the SM based algorithm is expected to gain a lower parallelisation efficiency for two reasons:

- It owns the same communication overhead while the sequential algorithm is faster.
- There is less potential for positive caching effects in the parallelisation because optimal cache utilisation (due to the spacially oriented coefficient scan) is supposed to be the main reason that the SM based algorithm is faster.



(a) SPIHT

(b) SM based algorithm

Figure 9: Speedups for varying #PE and fixed compression rate (0.14 bpp).

The fact that the parallelisation efficiency is about the same strongly suggests that the bit-stream assembly in the parallel SM based algorithm is more efficient.

Figure 10 shows speedup curves for fixed #PE and varying compression rate. Of course, the execution time of the wavelet decomposition does not depend on the compression rate. The reason for the speedup breakdowns at certain compression rates is the termination problem of the parallel algorithm (see section 2.2.2). The parallel algorithm is optimal only at compression rates achieved at the end of a refinement step.

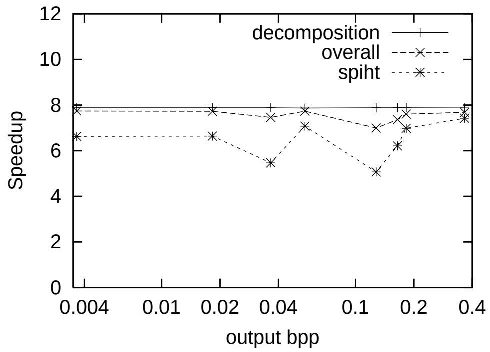
Note that although the speedup of the coding part increases with the bit-rate, the overall speedup remains constant or drops slightly because the share in execution time of the coding part increases with the bit-rate.

The problem of unevenly distributed complexity is illustrated in Figure 11. Here, approximately the first half of the video sequence is substituted by the "car-phone" sequence which contains much more motion than the "grandma" sequence. This causes bigger coefficient values at higher frequency sub-bands for the more complex video parts. Thus, more coefficients have to be processed within a refinement step, which makes the algorithm consume more computation time. A load balancing problem is the consequence. One can clearly see that the necessity of process synchronisation at several points in the algorithm leads to an increase of idle times of PEs waiting for other PEs. Figure 12 shows the speedups that are – compared to Figure 9 – slightly reduced because of this problem.

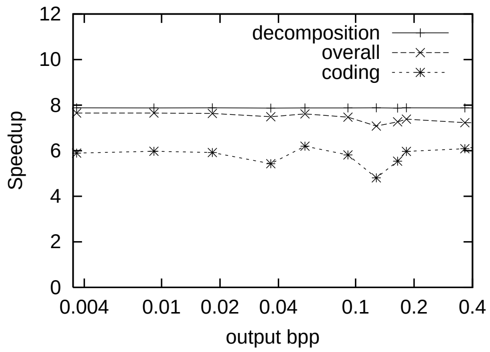
5 Conclusions

We have seen how an inherently sequential zero-tree coding algorithm can be parallelised. Although the speedups are not overwhelming, the presented way of parallelisation prevents the necessity to perform the coding sequentially. Thus, reasonable speedups are possible for higher numbers of processing elements and the whole range of compression rates.

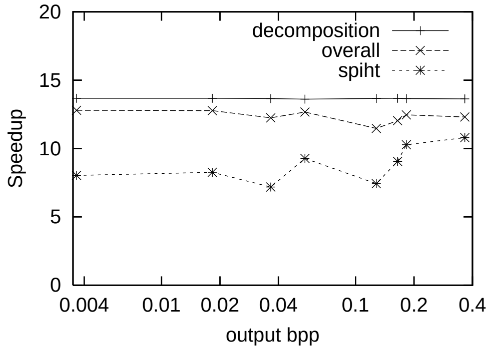
There are two methods for parallelisation: Either by the use of so called separators or by



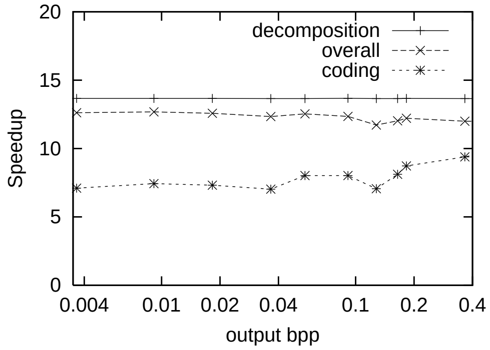
(a) SPIHT, 8 PEs



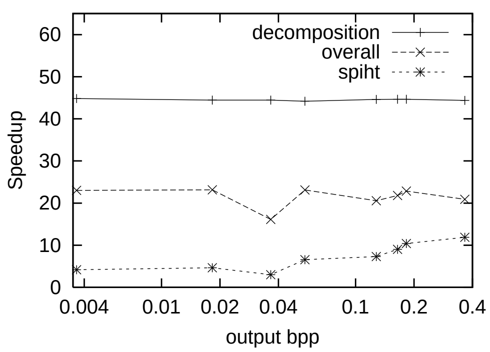
(b) SM based, 8 PEs



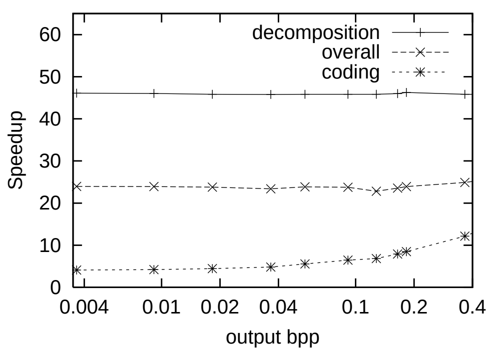
(c) SPIHT, 16 PEs



(d) SM based, 16 PEs

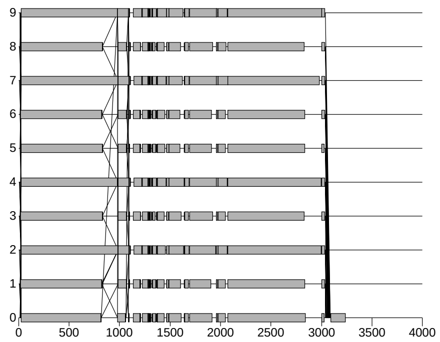


(e) SPIHT, 64 PEs

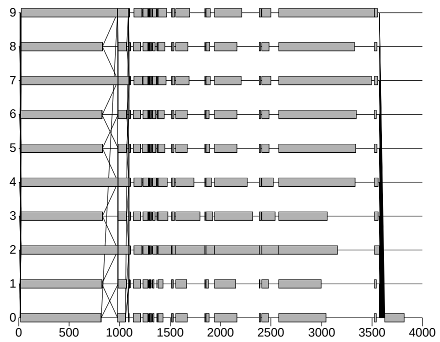


(f) SM based, 64 PEs

Figure 10: Speedup for decomposition, coding and overall speedup for varying compression rate.



(a) evenly distributed complexity



(b) unevenly distributed complexity

Figure 11: Execution scheme of decomposition and SPIHT coding for 10 PEs. Time (on horizontal axis) is measured in milliseconds. (Nearly) vertical black lines indicate data transfer. Horizontal grey bars indicate calculation phases.

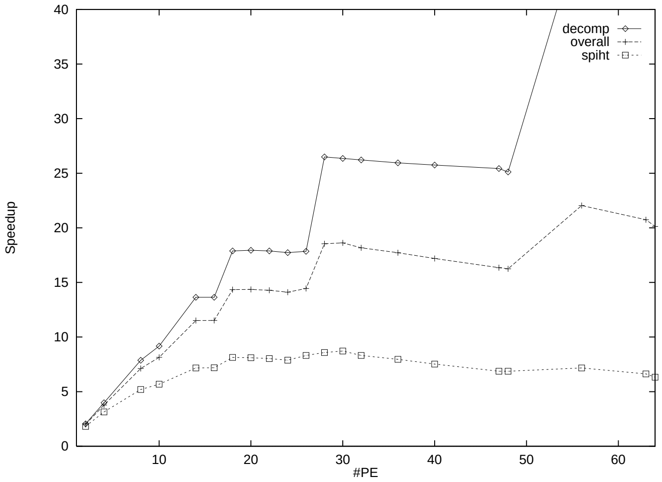


Figure 12: Speedups for video with unevenly distributed complexity (at fixed compression rate (0.14 bpp)).

rewriting the algorithm to fit better into the parallel architecture (significance map based algorithm). The first method is more complicated but guarantees compatibility with original SPIHT bit-streams. The second method shows very similar speedup results but better execution times.

Although unevenly distributed image/motion complexity can decrease the speedup potential, this effect seems to keep within limits.

Acknowledgments

The authors would like to acknowledge the support of the European Commission through TMR grant number ERB FMGE CT950051 (the TRACS Programme at EPCC). The author was also supported by the Austrian Science Fund FWF, project no. P13903.

References

[1] S.J. Choi and J.W. Woods. Three-dimensional subband/wavelet coding of video with motion compensation. In J. Biemond and E.J. Delp, editors, *Visual Communications and Image Processing '97*, volume 3024 of *SPIE Proceedings*, pages 96–104, San Jose, February 1997.

[2] C.D. Creusere. Image coding using parallel implementations of the embedded zerotree wavelet algorithm. In B. Vasudev, S. Frans, and S. Panchanathan, editors, *Digital Video Compression: Algorithms and Technologies 1996*, volume 2668 of *SPIE Proceedings*, pages 82–92, 1996.

[3] J. Fridman and E.S. Manolakos. On the scalability of 2D discrete wavelet transform algorithms. *Multidimensional Systems and Signal Processing*, 8(1–2):185–217, 1997.

[4] K.H. Goh, J.J. Soraghan, and T.S. Durrani. New 3-D wavelet transform coding algorithm for image sequences. *Electronics Letters*, 29(4):401–402, 1993.

[5] B.J. Kim and W.A. Pearlman. An embedded wavelet video coder using three-dimensional set partitioning in hierarchical trees (SPIHT). In *Proceedings Data Compression Conference (DCC'97)*, pages 251–259. IEEE Computer Society Press, March 1997.

[6] D. Krishnaswamy and M. Orchard. Parallel algorithm for the two-dimensional discrete wavelet transform. In *Proceedings of the 1994 International Conference on Parallel Processing*, pages III:47–54, 1994.

[7] R. Kutil and A. Uhl. Hardware and software aspects for 3-D wavelet decomposition on shared memory MIMD computers. In P. Zinterhof, M. Vajtersic, and A. Uhl, editors, *Parallel Computation. Proceedings of ACPC'99*, volume 1557 of *Lecture Notes on Computer Science*, pages 347–356. Springer-Verlag, 1999.

[8] R. Kutil and A. Uhl. Optimization of 3-d wavelet decomposition on multiprocessors. *Journal of Computing and Information Technology (Special Issue on Parallel Numerics and Parallel Computing in Image Processing, Video Processing, and Multimedia)*, 8(1):31–40, 2000.

[9] G. Lafruit and J. Cornelius. Parallelization of the 2D fast wavelet transform with a space-filling curve image scan. In A.G. Tescher, editor, *Applications of Digital Image Processing XVIII*, volume 2564 of *SPIE Proceedings*, pages 470–482, 1995.

[10] A.S. Lewis and G. Knowles. Video compression using 3D wavelet transforms. *Electronics Letters*, 26(6):396–398, 1990.

[11] W.-K. Lin and N. Burgess. Listless zerotree coding for color images. In *32nd Asilomar Conference on Signals, System and Computers*, CA, USA, November 1998.

[12] W.-K. Lin and N. Burgess. 3d listless zerotree coding for low bit rate video. In *IEEE International Conference on Image Processing*, Kobe, Japan, October 1999.

[13] J.N. Patel, A.A. Khokhar, and L.H. Jamieson. Scalability of 2-D wavelet transform algorithms: analytical and experimental results on coarse-grain parallel computers. In *Proceedings of the 1996 IEEE Workshop on VLSI Signal Processing*, pages 376–385, 1996.

[14] A. Said and W.A. Pearlman. A new, fast, and efficient image codec based on set partitioning in hierarchical trees. *IEEE Transactions on Circuits and Systems for Video Technology*, 6(3):243–249, 1996.

[15] S. Sullivan. Vector and parallel implementations of the wavelet transform. Technical report, Center for Supercomputing Research and Development, University of Illinois, Urbana, 1991.

[16] D. Taubman and A. Zakhor. Multirate 3-D subband coding of video. *IEEE Transactions on Image Processing*, 5(3):572–588, September 1993.

[17] F.W. Wheeler and W.A. Pearlman. Low-memory packetized SPIHT image compression. In *Proceedings of the Asilomar Conference on Signals, Systems, and Computers*, October 1999.

[18] F.W. Wheeler and W.A. Pearlman. SPIHT image compression without lists. In *Proceedings of the IEEE ICASSP 2000*, June 2000.

[19] M-L. Woo. Parallel discrete wavelet transform on the Paragon MIMD machine. In R.S. Schreiber et al., editor, *Proceedings of the seventh SIAM conference on parallel processing for scientific computing*, pages 3–8, 1995.