Zero-tree Based 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.

Keywords: Wavelets, Zero-tree, MIMD, Compression, Video coding

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.[1–5] 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[6–11] as well as in connection with image compression schemes.[12,13]

As the next step – the parallelisation of the coding algorithm – has not catched any attention yet as far as the author knows this work is dedicated to this topic. The SPIHT algorithm[14] was chosen for this purpose – or to be more concise: a 3-D variant.[2] However, this algorithm makes use of lists of coefficients which makes it hard to parallelise. This means that although initially consecutive list entries point to neighbouring wavelet transform coefficients, the algorithm jumbles the entries after a while. Thus, there is no easy data driven parallelisation.

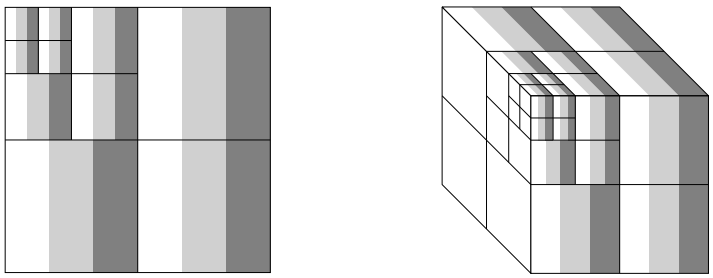
A method is needed to distribute lists among PEs (processing elements) and to handle specific list operations which guarantee that the collection of PE-local lists can be assembled correctly at any point in the execution of the algorithm. In this context separators will be introduced, i.e. dummy elements of a list which if read by the algorithm duplicate themselves into another or the same list. The result is a collection of PE-local bit-streams which contain separators as well. These separators split the lists and bit-streams into parts which form the global list if arranged in alternating order. By this means, they can be used to finally assemble the bit-stream properly. This will be the only sequential part of the algorithm.

2. 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.

Author contact information: University of Salzburg, Dept. of Scientific Computing, Jakob Haringer Str. 2, A-5020 Salzburg, Austria; E-mail: rkutil@cosy.sbg.ac.at

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



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

Figure 1. Distribution of coefficients or list entries. Different colours indicate different PEs.

In contrast to the parallelisation of the wavelet transform as presented in a previous paper[15] 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 the data are 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 5.

3. ZERO-TREES

Zero-tree based algorithms arrange the coefficients of a wavelet transform in a tree-like manner, 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). Here the following notations are used:

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

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

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 imported role in the SPIHT algorithm:

$$\mathrm{sig}(c) \Leftrightarrow |c| \geq \mathrm{threshold}$$

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 coefficients significances are statistically related which is exploited by the SPIHT algorithm, this also means that zero-trees are local objects (see Figure 2). This can be exploited by the parallelisation of the SPIHT algorithm (see Section 5).

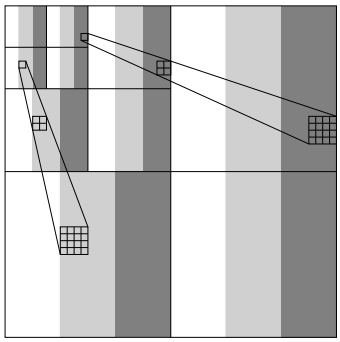


Figure 2. After parallel decomposition data is distributed in a way so that each zero-tree resides on a single PE.

4. SPIHT ALGORITHM

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

In the beginning the threshold is bigger than all the coefficients. Thus, all coefficients are insignificant. Then the threshold is repeatedly divided by 2 and the changes in significance have to be coded. Before and after each step (refinement step) the significance of the coefficients is represented by three lists:

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

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

LSP List of significant pixels
$c \in \mathrm{LSP} \Leftrightarrow \mathrm{sig}(c)$

When processing a refinement step each entry of each list has to be tested for a change of significance. For instance, if a type A entry of the LIS turns out not to to be insignificant any more (to be precise: its descendants) it 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.

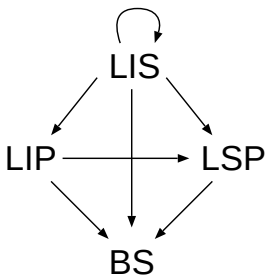


Figure 3. Which list entries cause new entries in other lists (or in the bit-stream BS)

Figure 3 shows all possible list entry transitions. Here BS stands for "bit-stream" which means that each evaluation of a list entry has to be coded into the bit-stream (so the decoding process is able to reproduce the decision). Furthermore, when processing the LSP the value of the corresponding coefficient is refined by one bit (which is written into the bit-stream).

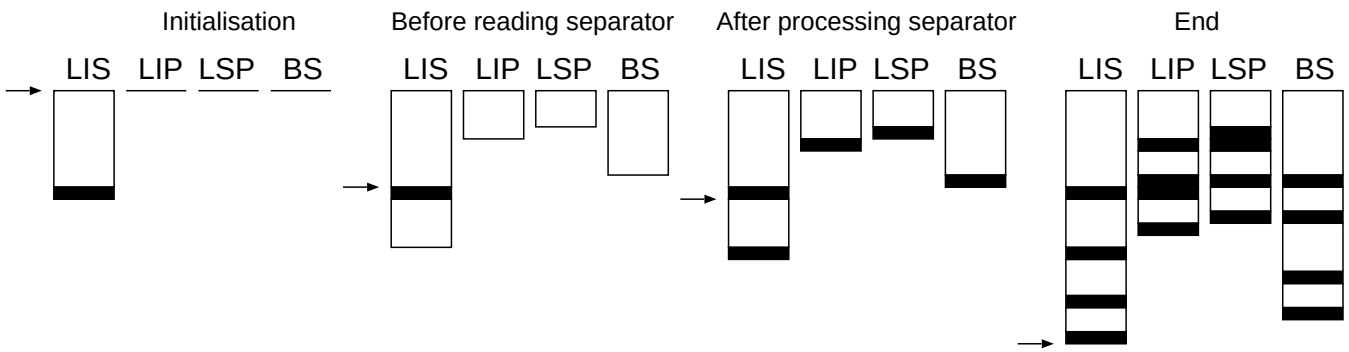


Figure 4. Functionality of separators. Four states of the three lists and the bit-stream while processing the LIS.

The decoding process performs the same algorithm but it does not evaluate the significance of the list entries but simply reads this information from the bit-stream and corrects the coefficients value as good as it can.

5. 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 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. So 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.

For initial distribution this is a simple task but 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 PEs view: where a sequence of local coefficients ends and parts of another PEs 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. So the initial distribution is to split the approximation sub-band into equal parts, assigning each part to a list on a single PE and appending a separator to the end of the 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 an entry could potentially have been inserted while processing previous iterator positions (this is the relation shown in Figure 3). Applying this principle the lists L_i on PE_i are split by separators into part L_{ij} such that the assembled list $L_{11}L_{21}L_{31}\ldots L_{12}L_{22}\ldots$ is identical to the list the sequential algorithm would produce. The same is true for the bit-stream.

An important question is when the processing of a list 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. But, 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 continuing with the next list/refinement step.

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.

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).

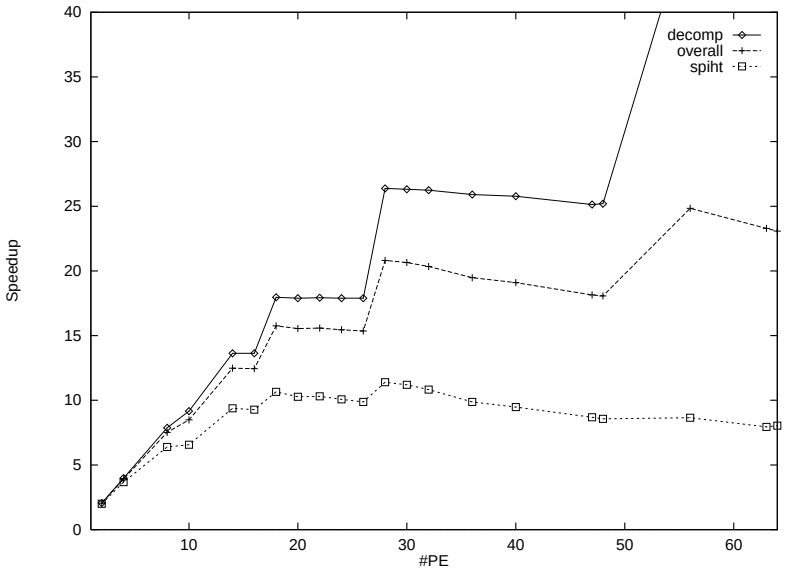


Figure 5. Speedups for varying #PE and fixed compression rate (0.14 bpp).

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.

Note that for the reverse algorithm – the reconstruction of video data from the bit-stream – the methods described above are not applicable. Although this 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 list have to be kept at each PE. The only speedup potentials are:

- Not adjusting non-local coefficients which involves floating point operations.
- 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 the necessity of process synchronisation and it contains no sequential part.

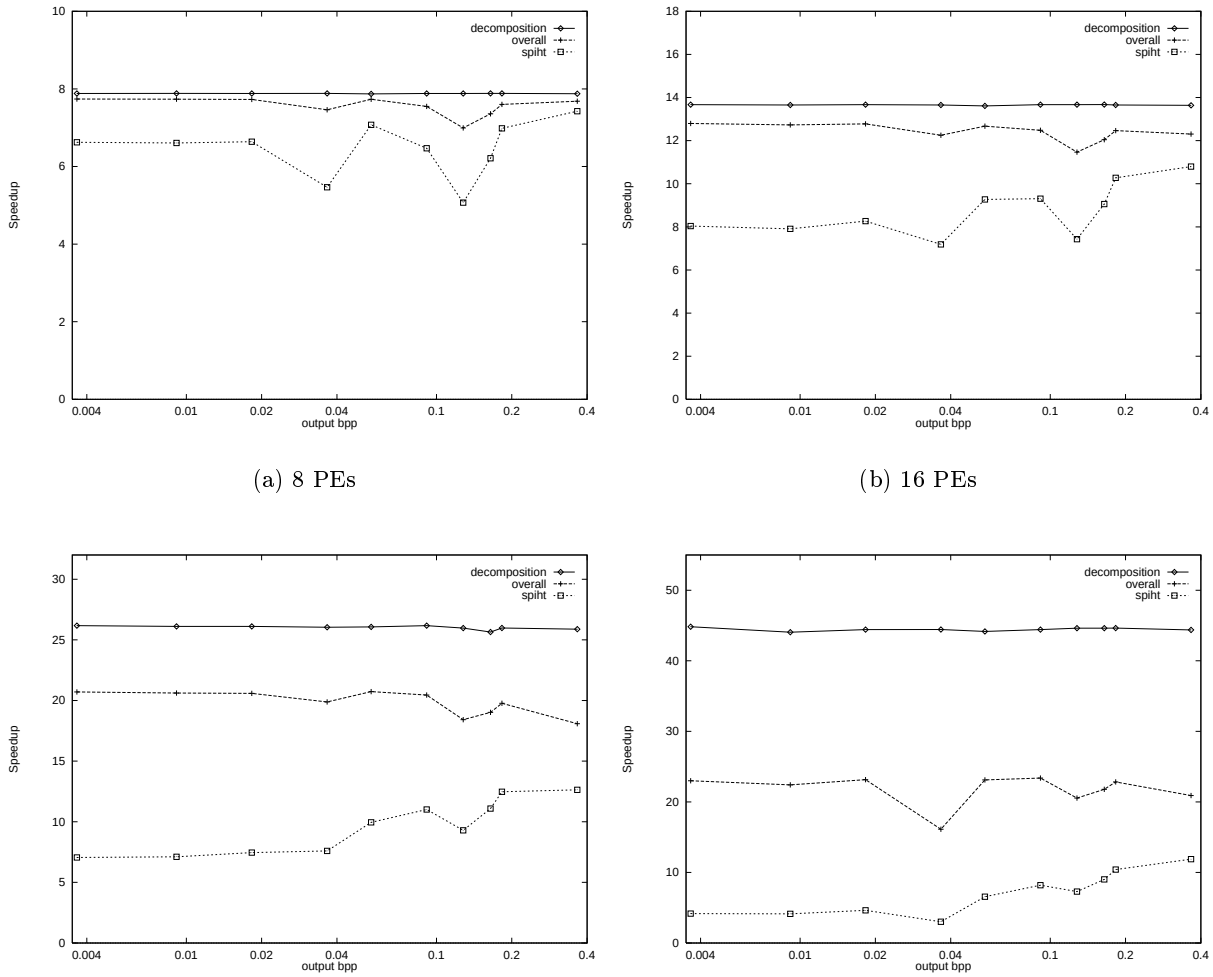
6. EXPERIMENTAL RESULTS

Experimental results were conducted on a Cray T3E-900/LC at the Edinburgh Parallel Computing Centre. 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.

Figure 5 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 speedup of the SPIHT parallelisation (without wavelet transform) seems to be limited by approximately 10. The reason for this is firstly the increased influence of the sequential part for higher #PE and secondly bigger load balancing problems for higher #PE due to the unevenly distributed complexity (see below).

Figure 6 shows speedup curves for fixed #PE and varying compression rate. Of course the wavelet decomposition is not dependent on the compression rate. The reason for the speedup breakdowns at certain compression rates is



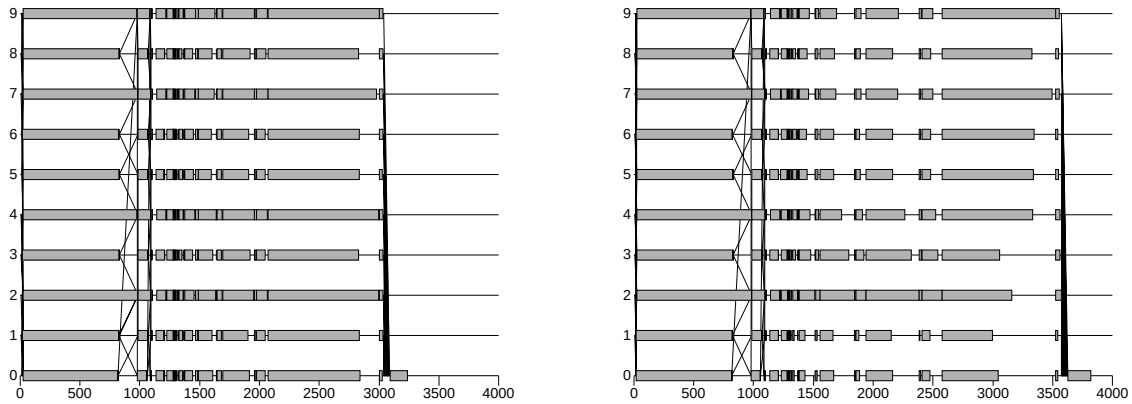
(a) 8 PEs

(b) 16 PEs

(c) 32 PEs

(d) 64 PEs

Figure 6. Speedup for decomposition, SPIHT coding and overall speedup for varying compression rate.



(a) evenly distributed complexity

(b) unevenly distributed complexity

Figure 7. Execution scheme of decomposition and 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.

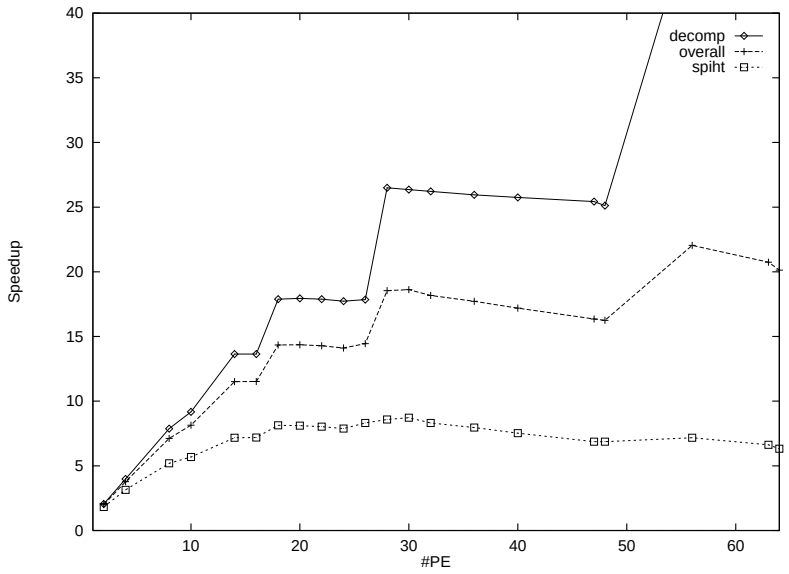


Figure 8. Speedups for video with uneven distributed complexity (at fixed compression rate (0.14 bpp)).

the problem of terminating the parallel algorithm (see Section 5). Only at compression rates achieved at the end of a refinement step the parallel algorithm is optimal.

Note that although the speedup of the coding part (SPIHT) increases with the bpp-value the overall speedup remains constant of drops slightly because the share in execution time of the coding part increases with the bpp-value.

The problem of unevenly distributed complexity is illustrated in Figure 7. Here approximately the first half of the video sequence is substituted by the "car-phone" sequence which contains much more motion as 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 8 shows the speedups that are – compared to Figure 5 – reduced because of this problem.

7. CONCLUSIONS

We have seen how an inherently sequential zero-tree coding algorithm can be parallelised using so called separators. 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.

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

ACKNOWLEDGEMENTS

The author wants to thank the EPCC (Edinburgh Parallel Computing Centre) and the TRACS programme (Training and Research on Advanced Computing Systems) for providing equipment, infrastructure and support for parallel computing (especially on Cray T3E). The author was also supported by the Austrian Science Fund FWF, project no. P13903.

REFERENCES

1. A. Lewis and G. Knowles, "Video compression using 3D wavelet transforms," *Electronics Letters* **26**(6), pp. 396–398, 1990.
2. B. Kim and W. Pearlman, "An embedded wavelet video coder using three-dimensional set partitioning in hierarchical trees (SPIHT)," in *Proceedings Data Compression Conference (DCC'97)*, pp. 251–259, IEEE Computer Society Press, Mar. 1997.
3. K. Goh, J. Soraghan, and T. Durrani, "New 3-D wavelet transform coding algorithm for image sequences," *Electronics Letters* **29**(4), pp. 401–402, 1993.
4. D. Taubman and A. Zakhor, "Multirate 3-D subband coding of video," *IEEE Transactions on Image Processing* **5**, pp. 572–588, Sept. 1993.
5. S. Choi and J. Woods, "Three-dimensional subband/wavelet coding of video with motion compensation," in *Visual Communications and Image Processing '97*, J. Biemond and E. Delp, eds., vol. 3024 of *SPIE Proceedings*, pp. 96–104, (San Jose), Feb. 1997.
6. 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), pp. 31–40, 2000.
7. M.-L. Woo, "Parallel discrete wavelet transform on the Paragon MIMD machine," in *Proceedings of the seventh SIAM conference on parallel processing for scientific computing*, R. S. et al., ed., pp. 3–8, 1995.
8. S. Sullivan, "Vector and parallel implementations of the wavelet transform," tech. rep., Center for Supercomputing Research and Development, University of Illinois, Urbana, 1991.
9. J. Patel, A. Khokhar, and L. 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*, pp. 376–385, 1996.
10. J. Fridman and E. Manolakos, "On the scalability of 2D discrete wavelet transform algorithms," *Multidimensional Systems and Signal Processing* **8**(1–2), pp. 185–217, 1997.
11. D. Krishnaswamy and M. Orchard, "Parallel algorithm for the two-dimensional discrete wavelet transform," in *Proceedings of the 1994 International Conference on Parallel Processing*, pp. III:47–54, 1994.
12. G. Lafruit and J. Cornelius, "Parallelization of the 2D fast wavelet transform with a space-filling curve image scan," in *Applications of Digital Image Processing XVIII*, A. Tescher, ed., vol. 2564 of *SPIE Proceedings*, pp. 470–482, 1995.
13. C. Creusere, "Image coding using parallel implementations of the embedded zerotree wavelet algorithm," in *Digital Video Compression: Algorithms and Technologies 1996*, B. Vasudev, S. Frans, and S. Panchanathan, eds., vol. 2668 of *SPIE Proceedings*, pp. 82–92, 1996.
14. A. Said and W. 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), pp. 243–249, 1996.
15. R. Kutil and A. Uhl, "Hardware and software aspects for 3-D wavelet decomposition on shared memory MIMD computers," in *Parallel Computation. Proceedings of ACPC'99*, P. Zinterhof, M. Vajtersic, and A. Uhl, eds., vol. 1557 of *Lecture Notes on Computer Science*, pp. 347–356, Springer-Verlag, 1999.