# OPTIMIZATION OF BITSTREAM ASSEMBLY IN PARALLEL MULTIMEDIA COMPRESSION

RADE KUTIL *

**Abstract.** Multimedia compression algorithms usually take DCT or wavelet transformed input data and produce a stream of bits as output. It has been shown that compression algorithms can be parallelized in a data driven way so that a sequence of bit-stream parts is produced by each processing element (PE) corresponding to the PE-local part of the partitioned and distributed input data. Unfortunately, the collection and assembly of these bit-stream parts by a single PE turns out to be a major bottleneck because it is sequential. This work addresses this problem by sequential optimizations and parallelization of the assembly process itself.

**Key words.** video coding, transform coding, parallel processing

**1. Introduction.** The most successful method for image and video compression is transform coding. The input data is usually given as two- or three-dimensional arrays of pixels. This data is transformed by a blocked discrete cosine transform (DCT) (as in JPEG [15] or MPEG [2, 9]) or by a discrete wavelet transform (as in JPEG2000 [7] or SPIHT [14]). The transformed data is then quantized and the resulting integer values are encoded bit by bit.

Although these algorithms are efficient in terms of rate-distortion performance as well as with respect to computational complexity, the amount of data that has to be processed causes prohibitive computational and memory demands (especially for real-time applications). Therefore, MIMD architectures seem to be an interesting choice for such algorithms.

The parallelization of a multimedia compression scheme is divided into at least two phases: Parallel transform and parallel compression. Parallel transform is easy for blocked DCT, since DCT blocks do not interfere. A significant amount of work has been done on parallel wavelet transform [12, 17, 6]. The parallelization of the compression algorithm did not catch so much attention, although there exists parallel implementations of MPEG [1], JPEG [3, 5], JPEG2000 [13], EZW [4, 16] and SPIHT [10, 11].

The author's own investigations showed that the SPIHT algorithm can be fully parallelized except for the bit-stream assembly as the only sequential part. Although bit-stream assembly seems to be a simple and non-complex process, it turned out to be a major bottleneck in parallel SPIHT compression. This work is dedicated to the comparison of several approaches to efficient bit-stream assembly.

For obtaining experimental results, MPI implementations on a Cray T3E were employed which is situated at the High-Performance Computing-Center in Stuttgart and consists of 512 DEC Alpha EV5 processors (900 MFLOPS each) interconnected by a 3-D torus with 3 GB/s bandwidth in each network node. Results were also conducted on a shared memory machine (SGI Powerchallenge GR at RIST++, Salzburg Univ.) with 20 MIPS R10000 processors. However, these results are very similar to the Cray T3E results. Therefore, they are not shown in this paper. The parallel SPIHT algorithm is applied to 3-D monochrome video data with $864 \times 88 \times 72$ pixels. In all examples, a bit-rate of 0.09 bpp is chosen.

*Dept. of Scientific Computing, University of Salzburg, Jakob Haringer-Str. 2, A-5020 Salzburg, Austria (rkutil@cosy.sbg.ac.at).

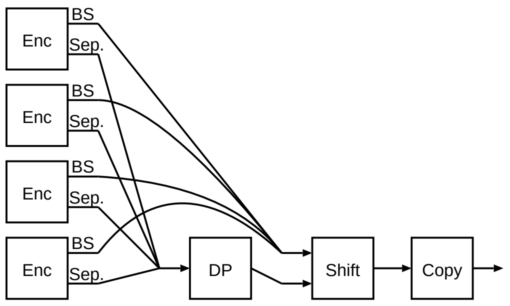

FIG. 3.1. *Method A. Enc = Encoding process (produces bit-stream parts and separators), BS = bit-stream, Sep. = separators, DP = calculation of destination positions, Shift = bit-alignment of bit-stream parts, Copy = merge bit-stream parts to a single bit-stream.*

## 2. Separators.

**2. Separators.** Most multimedia compression algorithms can be viewed as an abstract procedure which processes a set of primitive tasks in a certain order. For each task, a certain amount of bits is written into a bit-stream. Moreover, each task is associated with a certain approximate spatial position (i.e. a DCT-block or a wavelet). Hence, there exists a natural approach to data-driven parallelization, that is to distribute the source data among the PE's and to distribute the primitive tasks corresponding to the data distribution.

The problem is now to assemble the bit-stream parts produced by each PE both correctly and efficiently. A good way to do this is to use separators. Separators are dummy entries in the list of tasks which indicate a change of the PE the tasks belong to. So, each PE only has local tasks in its local part of the task list and inserts a separator wherever the global list continues with other PE's tasks. While tasks are processed, new tasks can be created and appended at the end of the task list. If the creation of these tasks depends only on local information (i.e. other tasks that belong to the same PE), then it is possible that each PE handles its separators correctly without any additional communication. When separators are processed as part of the task list, two things have to be done: First, a separator has to be inserted into the bit-stream. This is necessary to identify the bit-stream parts that have to be assembled in an alternating way at the end of the encoding process. Second, a separator has to be inserted at the end of the task list to separate newly created tasks correctly from newly created tasks of other PEs.

Experiments show that, very often, there are no actual tasks between separators. This is the case when many tasks do not initiate new tasks, so only separators are appended at the end of the task list. To reduce memory demands, it is necessary to keep such groups of separators together in a single entry – associated with a counter. For the bit-stream, separators are implemented as arrays of pointers to exact bit positions. Accordingly, equal pointers are kept together and are associated with a counter.

In this work, a parallel implementation [10, 11] of the well-known SPIHT algorithm [14, 8] is used. In this algorithm, the task list is split into three lists. Each task is associated with a single wavelet coefficient. Additionally, each list is reread when its end is reached so as to encode another bit-plane for higher precision. Entries that may not be processed anymore have to be removed.

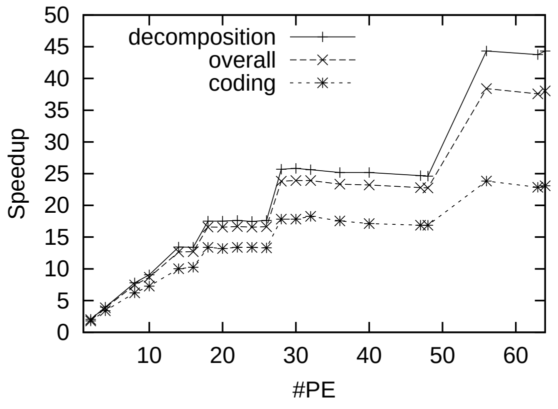

Fig. 3.2. *Speedups for method A*

**3. Method A: Sequential Assembly.** The first and most easy-to-implement way of bit-stream assembly is to collect all bit-stream parts together with their separators and to put the parts together sequentially on a single PE. This process is visualized in Figure 3.1. The assembly itself is split into three blocks: The first one determines the destination position of the bit-stream parts in the output bit-stream. The second part shifts the bit-stream parts to the correct bit-position, since the bit-stream parts are not byte-aligned. The last block simply copies the parts together.

The first block - the calculation of the destination positions - can be implemented as follows:

```
$p \leftarrow 0$
while still separators left
    for $k$ in $1 \ldots$ #PE
        select new separator from PE$_k$
        $l \leftarrow$ difference to old separator from PE$_k$
        if $l > 0$
            output $(p, p + l)$ as destination positions
            $p \leftarrow p + l$
```

A first improvement of this algorithm is to reduce the number of iterations of the "while still separators left" loop. This can be done because, very often, subsequent separators are equal (and kept together in a single entry). If this is the case for all PE's separator lists at the beginning of the loop body, the minimum equal-separator count can be removed from all lists at once without corrupting the output. This modification reduces the time for bit-stream assembly (without collection) by about 40%.

Speedup results for the wavelet transform, the SPIHT encoding and the overall encoding process are shown in Figure 3.2. Although the speedups for the coding part are not as good as those of the parallel wavelet decomposition, the overall speedups are quite reasonable.

**4. Method B: Parallel Shift.** The next approach is to parallelize the shift operation because it is the most time consuming operation in the process of bit-stream assembly. To do this, the destination positions have to be distributed to enable the PEs to determine the correct bit-alignments of the bit-stream parts (see Figure 4.1). This additional communication step is the main disadvantage of this approach. Thus,

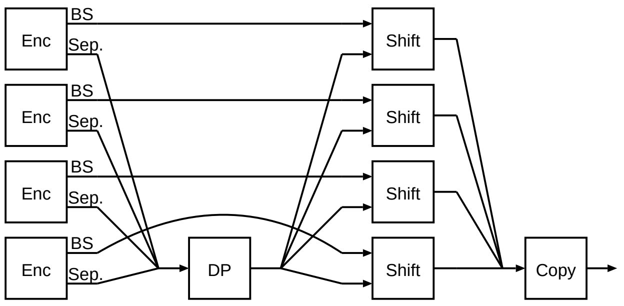

Fig. 4.1. *Method B*

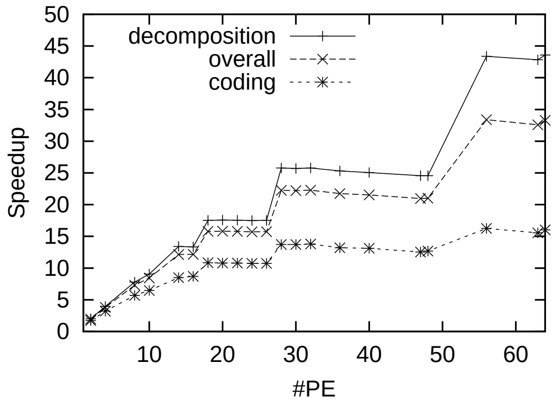

Fig. 4.2. *Speedups for method B*

the speedups are worse than for method A (see Figure 4.2).

**5. Method C: Redundant Calculation of Destination Positions.** To avoid the additional communication step of distributing the destination positions, this method calculates the destination positions redundantly on every PE. Unfortunately, this involves $n$-to-$n$ communication because each PE needs the full separator lists of each other PE. Therefore, the speedups of this approach are also degraded because of communication overhead as can be seen in Figure 5.2.

**6. Method D: Parallel Calculation of Destination Positions.** To calculate the destination positions for its own bit-stream parts, a PE does not really have to know all other's separator lists in detail. It would be sufficient to have a separator list of a fictive bit-stream in which all other PE's bit-stream parts are assembled. Such a separator list can easily be constructed by adding corresponding separators as in Figure 6.1.

With the help of this construction, it is possible to gather enough information on each PE by an **all-reduce** operation as shown in Figure 6.1. If the PE's are hypercube-connected, the communication can be performed in parallel as depicted in Figure 6.1. In this case, this operation has logarithmic complexity.

The speedup results are shown in Figure 6.2. Comparing the results to those of method A (Figure 3.2), we see that they are approximately equal. This means that although we have parallelized the bit-stream assembly almost completely, the

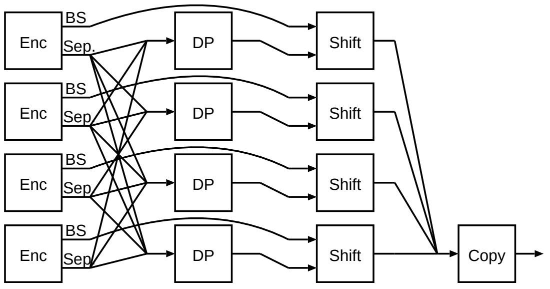

Fig. 5.1. *Method C*

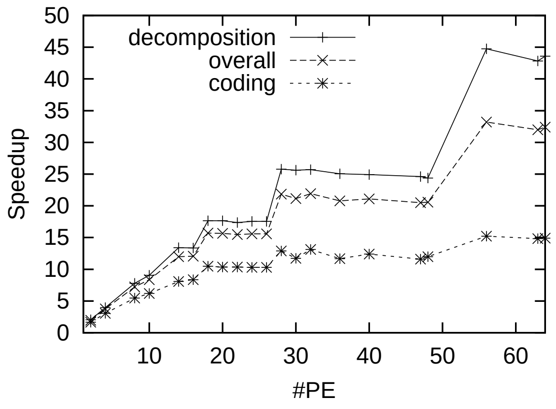

Fig. 5.2. *Speedups for method C*

increased communication overhead makes the additional speedup potential vanish. However, given a better communication bandwidth and/or lower computational power of processor elements, method D could outperform method A.

**7. Conclusions.** Four different approaches to efficient bit-stream assembly in parallel multimedia coding were presented. While method B and C showed major drawbacks in performance due to significant communication overhead, methods A and D show similar performance results. The advantage of method A is that it is easy to implement. However, method D could outperform method A given better communication bandwidth and/or lower computational power of processor elements. This is because method D manages to parallelize the bit-stream assembly almost completely (speedups are primarily degraded by communication overheads) while method A is a purely sequential approach.

The presented methods can also be used for parallel post-processing of bit-streams such as adding error correcting codes. Method D is expected to be the optimal choice for such operations because greater computational demands reduce the share of the communication overhead.

**Acknowledgments.** The author was supported by the Austrian Science Fund FWF, project no. P13903. The author also wants to thank the High-Performance Computing-Center (HLRS) in Stuttgart for granting access to their computing facilities.

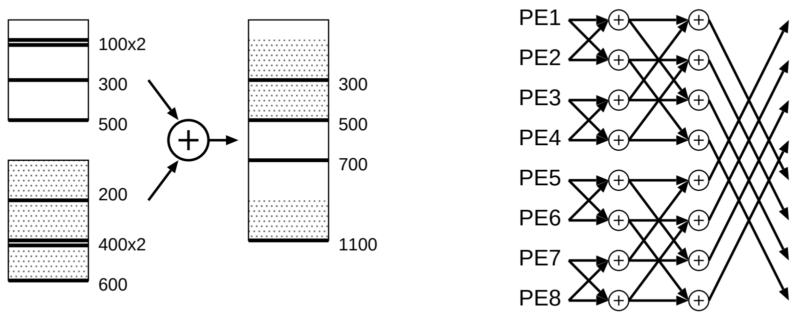

Fig. 6.1. *Calculation of destination positions in method D by an all-reduce operation*

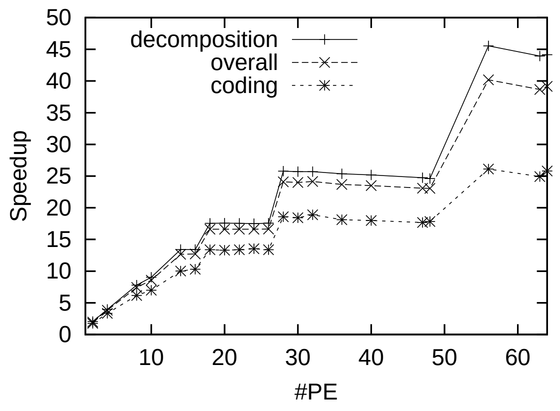

Fig. 6.2. *Speedups for method D*

## REFERENCES

[1] S. Akramullah, I. Ahmad, and M. Liou, *A data-parallel approach for real-time MPEG-2 video encoding*, Journal of Parallel and Distributed Computing, 30 (1995), pp. 129–146.

[2] L. Chiariglione-Convenor, *MPEG-2: Generic coding of moving pictures and associated audio information.* ISO/IEC JTC1/SC29/WG11, July 1996.

[3] G. W. Cook and E. J. Delp, *An investigation of JPEG image and video compression using parallel processing*, in Proceedings of the IEEE International Conference on Accoustics, Speech and Signal Processing, ICASSP '94, Adelaide, South Australia, Australia, Apr. 1994, pp. 437–440.

[4] 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, 1996, pp. 82–92.

[5] J. Falkemeier and G. Joubert, *Parallel image compression with JPEG for multimedisa applications*, in High Performance Computing: Technologies, Methods & Applications, J. Dongarra et al., eds., no. 10 in Advances in Parallel Computing, North Holland, 1995, pp. 379–394.

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

[7] ISO/IEC JPEG committee, *JPEG 2000 image coding system — ISO/IEC 15444-1:2000*, Dec. 2000.

[8] 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), IEEE Computer Society Press, Mar. 1997, pp. 251–259.

[9] R. Koenen, *Overview of the MPEG-4 standard.* ISO/IEC JTC1/SC29/WG11, July 1996.

[10] R. Kutil, *Zerotree based video coding on MIMD architectures*, in Media Processors 2001, S. Panchanathan, V. Bove, and S. Sudharsanan, eds., vol. 4313 of SPIE Proceedings, Jan. 2001, pp. 61–68.

[11] ———, *Approaches to zerotree image and video coding on MIMD architectures*, Parallel Computing, 28 (2002), pp. 1095–1109.

[12] 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 (2000), pp. 31–40.

[13] P. Meerwald, R. Norcen, and A. Uhl, *Parallel JPEG2000 image coding on multiprocessors*, in Proceedings of the International Parallel & Distributed Processing Symposium 2002 (IPDPS'02), Fort Lauderdale, FL, USA, Apr. 2002, IEEE Computer Society Press, p. 2.

[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 (1996), pp. 243–249.

[15] G. Wallace, *The JPEG still picture compression standard*, Communications of the ACM, 34 (1991), pp. 30–44.

[16] F. Wheeler and W. Pearlman, *Low-memory packetized SPIHT image compression*, in Proceedings of the Asilomar Conference on Signals, Systems, and Computers, Oct. 1999.

[17] 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. Schreiber et al., eds., 1995, pp. 3–8.