Parallel JPEG2000 Image Coding on Multiprocessors

Peter Meerwald Roland Norcen Andreas Uhl
RIST++ & Department of Scientific Computing
University of Salzburg, AUSTRIA
e-mail: {pmeerw,rnorcen,uhl}@cosy.sbg.ac.at

Abstract

In this paper, we discuss how the two reference implementations of the upcoming JPEG2000 image coding standard can be parallelized for the execution on shared-memory multiprocessors. By runtime analysis, we identify two major stages in the coding process of JPEG2000 where parallelism can be exploited. We present techniques to exploit the parallelism within these two stages, and speedup results obtained on several hardware platforms. We focus on OpenMP as well as JAVA threads for programming within shared-memory environments.

1. Introduction

In recent years there has been a tremendous increase in the demand for digital imagery. Applications include consumer electronics (Kodak's Photo-CD, HDTV, SHDTV, Video-on-Demand, and Sega's CD-ROM video game), medical imaging (digital radiography), video-conferencing, surveillance applications, and scientific visualization. The problem inherent to any digital image or digital video system is the large bandwidth required for transmission or storage.

Unfortunately, many applications demand execution times that are not possible using a single serial microprocessor, which leads to the use of high performance computers for such tasks [19] (beside the use of DSP chips, FPGAs, media processors, or application specific VLSI designs). In this context, several papers have been published describing image coding on general purpose parallel architectures – see for example JPEG [3, 6, 8], vector quantization [13, 14], and fractal compression [9, 10, 11, 23].

Image and video coding methods that use wavelet transforms [22] 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. With the finalization of the wavelet based JPEG2000 standard [1, 4, 5, 2] and the inclusion of a wavelet algorithm for synthetic/natural hybrid coding in MPEG-4 [20] there is no doubt left that wavelet image compression has to be considered state of the art nowadays. Therefore, a thorough investigation of parallel versions of these algorithms seems mandatory.

In this work, we discuss the parallelization of two JPEG2000 reference implementations: the JJ2000 codec (see `http://jj2000.epfl.ch`) using JAVA threads and the Jasper C codec (see `http://www.ece.ubc.ca/~madams`) using OpenMP [7] (see `http://www.openmp.org`). Section 2 is devoted to a short introduction to JPEG2000 which highlights algorithmic properties and improvements over JPEG. The following sections discuss the parallelization approaches and present the corresponding experimental results.

2. JPEG2000

The JPEG2000 image coding standard is based on a scheme originally proposed by Taubman and known as EBCOT ("Embedded Block Coding with Optimized Truncation" [21]). The major difference between previously proposed wavelet-based image compression algorithms such as EZW [18] or SPIHT [16] is that EBCOT as well as JPEG2000 operate on independent, non-overlapping blocks which are coded in several bit layers to create an embedded, scalable bitstream. Instead of zerotrees, the JPEG2000 scheme depends on a per-block quad-tree structure since the strictly independent block coding strategy precludes structures across subbands or even code-blocks. These independent code-blocks are passed down the "coding pipeline" shown in Fig.1 and generate separate bitstreams. Transmitting each bit layer corresponds to a certain distortion level. The partitioning of the available bit budget between the code-blocks and layers ("truncation points") is determined using a sophisticated optimization strategy for optimal rate/distortion performance.

The main design goals behind EBCOT and JPEG2000 are versatility and flexibility which are achieved to a large

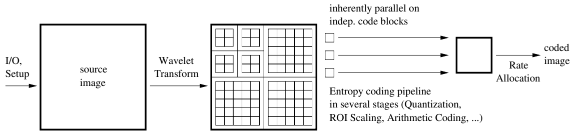

Figure 1. JPEG2000 coding pipeline

extent by the independent processing and coding of image blocks [4], and of course to provide a codec with a better rate-distortion performance than the widely used JPEG, especially at lower bitrates [17]. The default for JPEG2000 is to perform a five-level wavelet decomposition with 7/9-biorthogonal filters and then segment the transformed image into non-overlapping code-blocks of no more than 4096 coefficients which are passed down the coding pipeline.

In Fig.2 we compare the time required for encoding differently sized images using four image codecs: DCT-based JPEG, wavelet-based SPIHT, Jasper, and JJ2000 (Jasper and JJ2000 both implement the JPEG2000 standard). Note, that JPEG, SPIHT, and Jasper are C/C++ based whereas JJ2000 is written in JAVA.

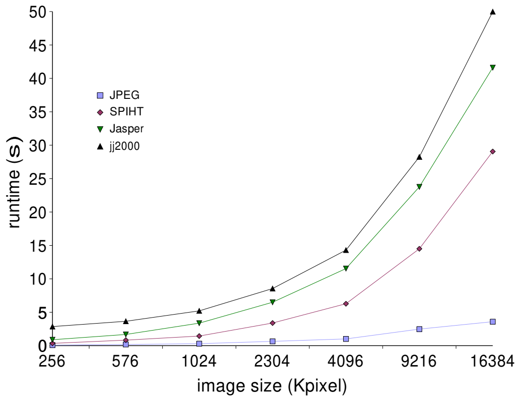

Figure 2. Compression timings

Evidently, JPEG is the by far fastest algorithm, whereas both JPEG2000 implementations are slowest. Interestingly, there is not much difference between the C and JAVA implementations (the IBM JDK 1.1.8 just-in-time compiler is used for JJ2000). Fig.3 shows a runtime analysis of the sequential execution of JJ2000 and Jasper. The wavelet transform part (intra-component transform) is clearly the most demanding part of the algorithm, followed by the encoding stage (tier-1 coding). Fortunately, both stages can be parallelized with little effort. Intrinsically sequential parts of the algorithm are image and bitstream I/O and R/D allocation which all show relatively low complexity. Obviously, high parallelization potential was one of the design goals of JPEG2000.

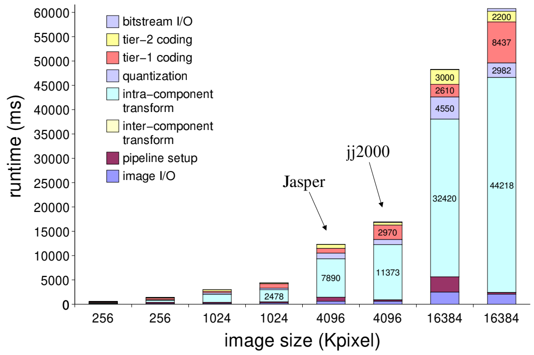

Figure 3. Serial Runtime Analysis of JJ2000 and Jasper on Intel Pentium II Xeon, 500 MHz

3. Parallel JPEG2000

The multiprocessor architecture (i.e. shared memory and virtual shared memory MIMD) – often also denoted SMP – is an interesting alternative to multicomputers for image processing tasks due to the high memory requirements of these applications. Also, the availability of comfortable programming environments for parallel processing on such architectures (e.g. OpenMP, JAVA Threads) is an important aspect. Finally, the excellent prize-performance ratio of Intel-based SMPs makes such systems very popular for many applications involving visual data processing [15]. In this section, we describe a "straightforward" SMP parallelization of two JPEG2000 reference implementations: the JJ2000 codec using JAVA threads and the Jasper C codec using OpenMP.

3.1. Parallelization using image tiling

Traditional parallelization approaches for JPEG such as [3, 6] and [8] include tiling the image and distributing the tiles among separate CPUs. As JPEG performs the DCT on 8×8 image blocks, this straightforward tile-based parallelization approach does not impair image quality because tiles are generally much larger than the transform blocks.

JPEG2000 employs the wavelet transform for image decorrelation. The wavelet decomposition is usually computed on the entire image, which inhibits annoying compression block artifacts that occur at low bit rate coding. However, in spite of the quality impact, JPEG2000 also supports the concept of image tiling for operation in low memory environments. In this case, the wavelet transform is performed on each image tile independently. Figure 4 illustrates the subjective image degradation due to tiling. In

(a) JPEG (b) JPEG2000 (c) JPEG2000 with tiling

Figure 4. The center part of the Lena image coded with a bitrate of 0.125 bpp, on the left using JPEG, in the middle employing JPEG2000 without tiling, and on the right using JPEG2000 with a tile size of 64×64.

figure 5, we show the impact of parallelizing JPEG2000 using this simple image tiling approach: Obviously, the processing of independent image tiles in parallel leads to a significant rate-distortion loss and severe blocking artifacts as the number of tiles and processors is increased.

In this paper, we do not follow this simple tiled-based parallelization idea but propose to distribute the global wavelet transform, as well as the code-block processing, among several CPUs.

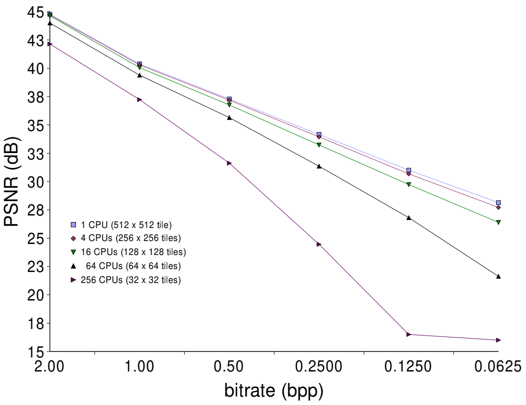

Figure 5. The impact of tile-based parallelization on JPEG2000 image quality.

3.2. Parallelization of JJ2000 using JAVA Threads

The approach followed in this work is to change as little as possible in the original JJ2000 code for parallelization. JAVA multi-threading is employed in the wavelet transform and encoding stage. For a multi-threaded wavelet transform, different parts of the data are assigned to different threads, the deterministic workload allows a static load allocation. However, synchronization is required at each decomposition level between vertical and horizontal filtering. In the encoding stage, on the other hand, no synchronization is necessary due to the processing of independent code-blocks. The load balance problem caused by the different runtime for each code-block is solved by using a pool of worker threads and a staggered round robin assignment of the code-blocks to these threads. Whereas the JJ2000 code already contains the necessary thread invocation calls for a parallel encoding stage, the transform part is covered in this work.

Fig.6 displays the runtime analysis of a multi-threaded execution on a 4 processor SMP system (a Compaq server with Intel Pentium II Xeon processors running at 500 MHz which is used for all subsequent experiments in this section). An overall speedup of 1.75 is achieved only. When analyzing the chart in more detail, we find that the speedup corresponding to the encoding stage is about 3.6 whereas the wavelet transform speedup is 1.6 at most. Therefore, we investigate the wavelet transform part in more detail.

Fig.7 shows the timings for the filtering procedures, broken down into the vertical and horizontal parts, respectively. The vertical filtering step requires more than 10 times the

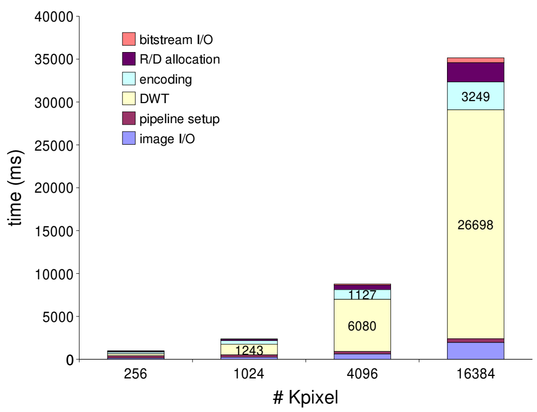

Figure 6. Parallel Runtime Analysis of JJ2000 on SMP (four processor Compaq, Intel Pentium II Xenon, 500 MHz)

execution time of the horizontal counterpart. Surprisingly, also the speedup for the vertical filtering is significantly lower than this for the horizontal case (compare Fig.8).

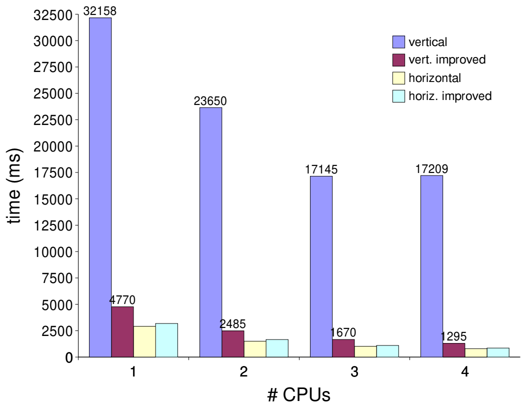

Figure 7. Original and improved filtering

This unexpected behaviour suggests the existence of a severe cache-miss problem (see also [12] for similar effects in an MPI implementation for a 3-D wavelet decomposition). In fact, it turns out that when using large images with width equal to a power-of-two and the filter length is longer than 4 (this corresponds to the 4-way associative cache), an entire image column is mapped onto a single cache-set. Consequently, during the execution of vertical wavelet filtering an enormous amount of cache misses occur. We have considered two approaches to improve the cache hit rate. First, the image width is forced to be not a power-of-two (e.g. by inserting dummy data, compare [12]). This technique does not require any modification in the filter code and results in the use of more cache sets and consequently allows cache hits on vertically adjacent pixels. Second, several adjacent columns are filtered concurrently within a single processor. When loading the first data points of an image column into the cache, the corresponding data of adjacent columns are situated within the same cache line. Therefore, computing the products of pixels and filter coefficients of all these columns can be performed without any cache misses (except the initial access which triggers the cache load). Here, a modification of the filter code is required – the results of the different columns have to be buffered. The second approach has turned out to be more effective.

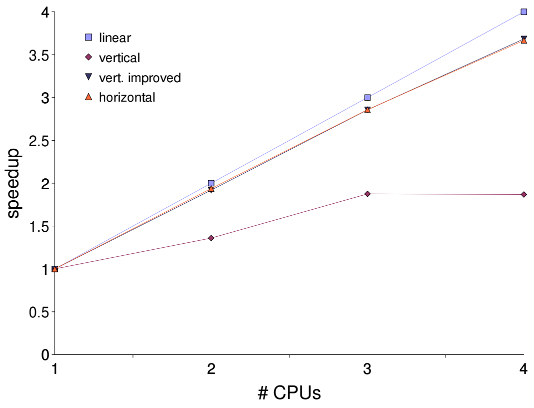

Figure 8. Speedup of filtering routines

A significant improvement is observed in Fig.7 – almost factor 10 is gained by our technique, horizontal and vertical filtering are now almost identical with respect to runtime. Additionally, the speedup of the improved vertical filtering routine is significantly higher (Fig.8) and now equals that of horizontal filtering. Note that the constrained speedup of the original filtering routine is due to the congestion of the bus caused by the high number of cache misses.

Finally, Fig.9 shows the runtime analysis of JJ2000 with the improved filtering routine. We notice an overall speedup of 5.39 with respect to the original JJ2000 implementation (see Fig.3). Of course, the superlinearity is due to the improved filtering routine. A further significant increase of parallel efficiency can not be expected, since the intrinsically sequential stages contribute already about 40% to the overall execution time and the efficiency of the parallel parts can hardly be improved without massively changing the code, which is not the scope of this work.

3.3. Parallel Jasper using OpenMP

With OpenMP we have another tool for programming within shared-memory environments. Basically, OpenMP

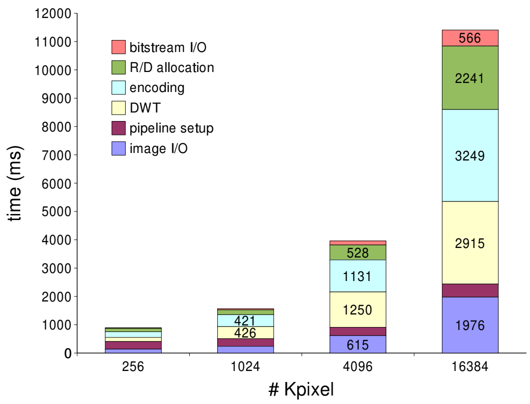

Figure 9. Parallel Runtime Analysis of JJ2000 with improved filtering (Intel SMP)

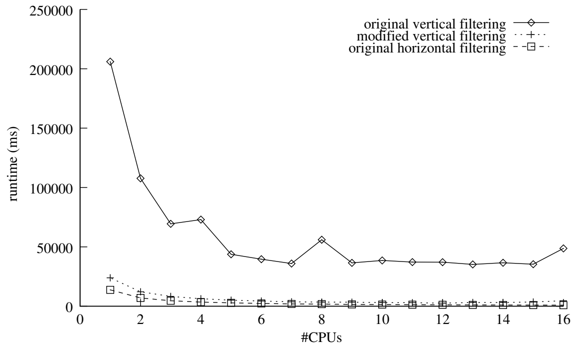

Figure 10. Parallel Runtime Analysis of Jasper (16384 Kpixel image, SGI): Original and improved filtering

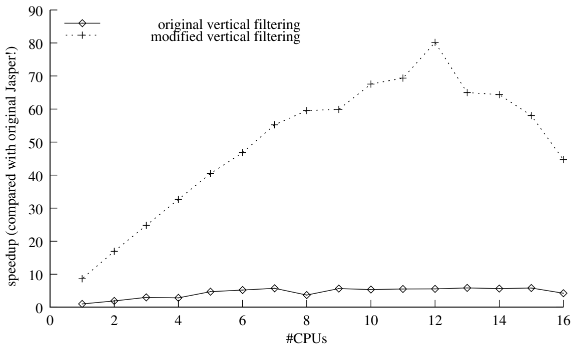

Figure 11. Parallel Runtime Analysis of Jasper (16384 Kpixel image, SGI): Speedup for original and improved filtering with respect to the original Jasper runtime

and threads have a similar programming flexibility and variety, since the first mentioned technique is based upon threads. OpenMP can be seen as a programming interface generalizing the usage of threads, hiding the pure thread and its appliance, respectively the synchronization between threads under macro constructs, so called *pragmas*. These *pragmas* provide more general constructs for performing sections of a sequential program (i.e. loops) in parallel.

When analyzing the single coding stages of Jasper, we see a very similar load distribution as the JJ2000 coder (figure 3). Zooming into the intra-component transform of the Jasper coder, the part in which the wavelet transform is performed, we also face the analogous problem with cache-misses. This cache problem increases with the dimensions of the image. Thus it is very convenient and straightforward to apply a similar parallelization for the Jasper codec as proposed in section 3.2. We enhance the vertical filtering in the same fashion as done in the JJ2000 coder: Filtering of vertical columns is done in a 'parallel' fashion, several neighbouring image columns are filtered concurrently within a single processor. Additionally, we employ OpenMP to parallelize the intra-component phase, where the wavelet transform is applied, as well as the tier-1 coding stage, where the independent code-block processing is done. Both stages of the coding phase can be parallelized easily and efficiently as in the JJ2000 case.

We have analyzed the parallel Jasper performance for different architectures. On an Intel SMP architecture (4 SMP Intel Pentium II Xeon running at 500 MHz), our results are similar to the JJ2000 thread-based parallelization. Generally, the Jasper C code saves about 20 percent of the JJ2000 computation time. The percentage of the parallel parts with respect to the total execution times are very similar. We want to give also results for a SGI Power Challenge (20 IP25 RISC processing units running at 194 MHz). The proportions of execution times do not change significantly, although we face very poor computation times when compared with the fast Intel processors, running at high frequencies.

Figure 10 shows the runtimes for the vertical wavelet filtering part of the Jasper coder on the Power Challenge. We clearly see the big gap between horizontal and vertical filtering. Applying the described improved vertical filtering, we close this gap significantly. Distributing the load of the modified wavelet decomposition with the aid of OpenMP to a number of processors, we can increase the vertical filtering over all resolution levels by a factor of 80 (see figure 11). Considering the overall runtime, including improved filtering, as well as parallelizing the wavelet transform and the code-block processing, we reduce the processing time by a factor of about 5 (figure 12). We must note, of course, that our comparisons are made with respect to the runtimes

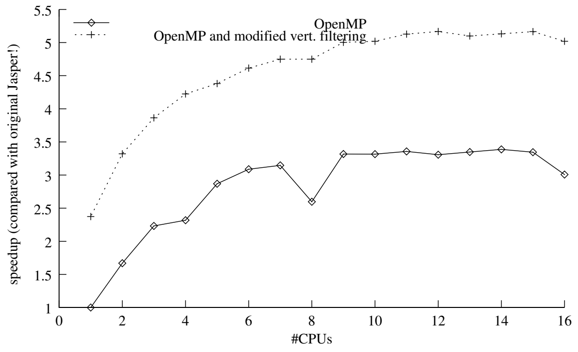

Figure 12. Parallel Runtime Analysis of Jasper (16384 Kpixel image, SGI): Speedup for the entire coding time with respect to the original Jasper runtime

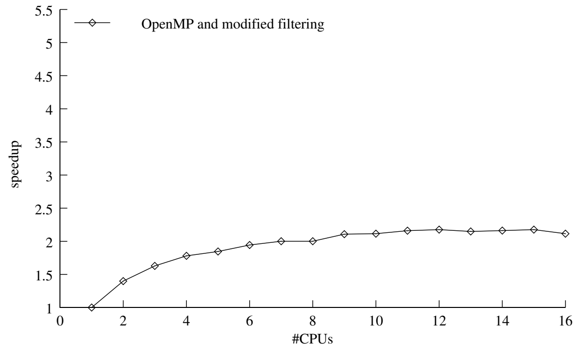

Figure 13. Parallel Runtime Analysis of Jasper (16384 Kpixel image, SGI): Speedup for the entire coding time with respect to the filtering optimized Jasper

of the original Jasper source. This is the reason, why we see these superlinear speedups. Anyhow, we think that these figures give a good feeling of how much the Jasper reference implementation can be improved, when having a look at both, sequential and parallel optimizations.

When taking the filtering optimized code as the reference for our speedup measurements, we can observe a total speedup of little more than 2 (figure 13). The values seen in this chart give the classical speedup for our parallelization, since we compare the parallel runtimes with the fastest available sequential code, which is - in this case - the Jasper code with improved vertical filtering. The modified vertical filtering technique accelerates the coding of large images by a factor of about 2.4.

Beside, we also parallelize the quantization step, which is only applied in the lossy case, since lossless compression doesn't use quantization. Quantization can be parallelized easily and very straightforward, since every processor may have a chunk of coefficients from the wavelet transform which it has to quantize. Extracting the needed time portion of the quantization and calculating the speedup with respect to this time slice, we see speedups of approximately 3.2 for performing the quantization stage in parallel.

Nevertheless, the contribution of this small computation slice to the whole coding time is too small to show a reasonable performance improvement for the whole image coder.

Applying OpenMP within the stages of the most computational effort, and optimizing the efficiency of vertical filtering, we proofed a speedup of more than 5 compared with the original Jasper reference implementation. This gain is reached with the aid of 10 processors and minimal implementation effort, meaning that only minor parts of the Jasper source code had to be changed to get this performance profit.

3.4. Theoretical versus Practical Speedup

Amdahl's law gives an upper bound on the achievable parallel speedup, assuming that for concurrent sections in the code perfect parallelism can be obtained. It can be written as

$$speedup = \frac{(s+p)}{(s+\frac{p}{N})},$$

where s is the runtime spent in inherently sequential code, p is the time spent in code which can be potentially executed in parallel and N is the number of processors available.

When we analyze the measured runtimes obtained on the Intel platform for the Jasper and JJ2000 code, we get an expected overall theoretical speedup of 2.60 and 3.19, respectively, for a 4 processor system. Our experimental results showed speedups of 1.85 and 1.75. When performing the improved filtering, the percentage of parallel code decreases obviously, and with it also the possible theoretical speedup. This is the reason for the restricted speedups of figure 13, where the maximum theoretical speedup would be around 2.4 for a 4 processor environment.

Wee clearly see, that the potential of our presented parallelizations is limited. Producing better speedups would require larger parts of the code to be run in parallel. The way JPEG2000 is designed, this could not be done without massively changing the code.

4. Conclusion

The runtime performance of the upcoming JPEG2000 reference implementations can be improved significantly, when operating in parallel and exploiting the features of threads and shared-memory. We could show, that all this can be done with minimal implementation effort, without changing major parts of the sources. Applying an optimized vertical filtering technique, we could additionally enhance the performance, especially for large image data.

Acknowledgements

This work has been partially supported by the Austrian Science Fund (project FWF-13903).

References

[1] JPEG2000 part 1 final committee draft version 1.0. Technical report, ISO/IEC FCD15444-1, March 2000.

[2] M. D. Adams, H. Man, F. Kossentini, and T. Ebrahimi. JPEG2000: The next generation still image compression standard. Technical report, Image Power Inc., Vancouver, BC, Canada, 2000.

[3] S. Bevinakoppa and other. Implementation of JPEG algorithm on SHIVA parallel architecture. In V. Prasanna, V. Bhaktar, L. Patnaik, and S. Tripathi, editors, *Parallel Processing – Proceedings of the International Workshop on Parallel Processing*, pages 184–188. Tata-McGraw Hill, 1994.

[4] M. Charriera, D. S. Cruz, and M. Larsson. JPEG2000, the next millenium compression standard for still images. In *Proceedings of the IEEE ICMCS'99*, June 1999.

[5] C. Christopoulos, A. Skodras, and T. Ebrahimi. The JPEG2000 still image coding system: an overwiew. *IEEE Transactions on Consumer Electronics*, 46(4):1103–1127, 2000.

[6] G. Cook and E. Delp. An investigation of scalable SIMD I/O techniques with application to parallel JPEG compression. *Journal of Parallel and Distributed Computing*, 30:111–128, 1996.

[7] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[8] J. Falkemeier and G. Joubert. Parallel image compression with JPEG for multimedia applications. In J. Dongarra et al., editors, *High Performance Computing: Technologies, Methods & Applications*, number 10 in Advances in Parallel Computing, pages 379–394. North Holland, 1995.

[9] J. Hämmerle and A. Uhl. Fractal image compression on MIMD architectures II: Classification based speed-up methods. *Journal of Computing and Information Technology (Special Issue on Parallel Numerics and Parallel Computing in Image Processing, Video Processing, and Multimedia)*, 8(1):71–82, 2000.

[10] D. Jackson and W. Mahmoud. Parallel pipelined fractal image compression using quadtree recomposition. *The Computer Journal*, 39(1):1–13, 1996.

[11] D. Jackson and G. Tinney. Performance analysis of distributed implementations of a fractal image compression algorithm. *Concurrency: Practice and Experience*, 8(5):357–380, June 1996.

[12] 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.

[13] P. Lukowicz and R. Cober. A massively parallel implementation of the full search vector quantization algorithm. In W. Gentzsch and U. Harms, editors, *High Performance Computing and Networking. Proceedings of HPCN Europe 1994*, volume 796 of *Lecture Notes on Computer Science*, pages 420–421. Springer, 1994.

[14] M. Manohar and J. Tilton. Progressive vector quantization on a massively parallel SIMD machine with application to multispectral image data. *IEEE Trans. on Image Process.*, 5(1):142–146, 1996.

[15] C. Rothlübbers and R. Orglmeister. Parallel image processing using a Pentium based shared-memory multiprocessor system. In H. Shi and P. Coffield, editors, *Parallel and Distributed Methods for Image Processing*, volume 3166 of *SPIE Proceedings*, pages 46–54, 1997.

[16] 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):243–249, 1996.

[17] D. Santa-Cruz and T. Ebrahimi. A study of JPEG 2000 still image coding versus other standards. In *Proceedings of the 10th European Signal Processing Conference, EUSIPCO '00*, Tampere, Finland, September 2000.

[18] J. Shapiro. Embedded image coding using zerotrees of wavelet coefficients. *IEEE Trans. on Signal Process.*, 41(12):3445–3462, 1993.

[19] K. Shen, G. Cook, L. Jamieson, and E. Delp. An overview of parallel processing approaches to image and video compression. In M. Rabbani, editor, *Image and Video Compression*, volume 2186 of *SPIE Proceedings*, pages 197–208, 1994.

[20] I. Sodagar, H. Lee, P. Hatrack, and Y. Zhang. Scalable wavelet coding for synthetic/natural hybrid coding. *IEEE Transactions on Circuits and Systems for Video Technology*, 9(2):244–254, 1999.

[21] D. Taubman. High performance scalable image compression with EBCOT. *IEEE Transactions on Image Processing*, 9(7):1158 – 1170, 2000.

[22] P. Topiwala, editor. *Wavelet Image and Video Compression*. Kluwer Academic Publishers Group, Boston, 1998.

[23] A. Uhl and J. Hämmerle. Fractal image compression on MIMD architectures I: Basic algorithms. *Parallel Algorithms and Applications*, 11(3–4):187–204, 1997.