Computer and Information Science; Vol. 10, No. 1; 2017
ISSN 1913-8989 E-ISSN 1913-8997
Published by Canadian Center of Science and Education

Exploiting Data-Parallelism on Multicore and SMT Systems for Implementing the Fractal Image Compressing Problem

Rodrigo da Rosa Righi[1], Vinicius F. Rodrigues[1], Cristiano A. Costa[1] & Roberto Q. Gomes[1]

[1] Applied Computing Graduate Program, Universidade do Vale do Rio dos Sinos, Brazil

Correspondence: Rodrigo R. Righi, Applied Computing Graduate Program, Universidade do Vale do Rio dos Sinos, Unisinos Av. 950, São Leopoldo, Rio Grande do Sul, Brazil. E-mail: rrrighi@unisinos.br

Correspondence: Rodrigo da Rosa Righi, Applied Computing Graduate Program, Universidade do Vale do Rio dos Sinos, Brazil. E-mail: rrrighi@unisinos.br

Received: November 21, 2015 Accepted: December 8, 2015 Online Published: December 25, 2016

doi:10.5539/cis.v10n1p34 URL: http://dx.doi.org/10.5539/cis.v10n1p34

Abstract

This paper presents a parallel modeling of a lossy image compression method based on the fractal theory and its evaluation over two versions of dual-core processors: with and without simultaneous multithreading (SMT) support. The idea is to observe the speedup on both configurations when changing application parameters and the number of threads at operating system level. Our target application is particularly relevant in the Big Data era. Huge amounts of data often need to be sent over low/medium bandwidth networks, and/or to be saved on devices with limited store capacity, motivating efficient image compression. Especially, the fractal compression presents a CPU-bound coding method known for offering higher indexes of file reduction through highly time-consuming calculus. The structure of the problem allowed us to explore data-parallelism by implementing an embarrassingly parallel version of the algorithm. Despite its simplicity, our modeling is useful for fully exploiting and evaluating the considered architectures. When comparing performance in both processors, the results demonstrated that the SMT-based one presented gains up to 29%. Moreover, they emphasized that a large number of threads does not always represent a reduction in application time. In average, the results showed a curve in which a strong time reduction is achieved when working with 4 and 8 threads when evaluating pure and SMT dual-core processors, respectively. The trend concerns a slow growing of the execution time when enlarging the number of threads due to both task granularity and threads management.

Keywords: image compression, fractal compression, simultaneous multithreading, big data.

1. Introduction

Considering the era of Big Data, the thematic of image compression becomes more and more relevant (Chen et al., 2012; Revathy & Jayamohan, 2012; Sundaresan & Devika, 2012). The main objective consists in reducing the irrelevance and redundancy of the image data to store or transmit data in an efficient way. For instance, images obtained by experiments in the fields of astronomy, medicine and geology may present several gigabytes in memory, emphasizing the use of image compression properly (Pinto & Gawande, 2012). In this context, a technique called Fractal Image Compression (FIC) appears as one of most efficient solutions for reducing the size of files (Jeng et al., 2009; Khan & Akhtar, 2013). An expensive encoding phase characterizes the FIC method, since the search used in the algorithm to find self-similarities is time-consuming. A square image with 1024 pixels as dimension may take more than an hour to be compressed in a single processing system. This elucidates why this technique is not so widespread among the traditional operating systems. However, at high compression ratios, fractal compression may offer superior quality than JPEG and Discrete-cosine- transform (DCT)-based algorithms (George & Al-Hilo, 2009). Unlike the coding phase, the decoding one occurs quickly, for instance, enabling users to download compressed images or videos from Web servers and visualize them in their hosts in a reasonable time interval.

Considering a lower encoding phase of FIC method, some alternatives are considered to minimize this process. Basically, the most alternatives try to reduce the coding time by reducing the search for the best-match block in a large domain pool (Fu & Zhu, 2009; Jeng et al., 2009; Mitra et al., 1998; Qin et al., 2009; Revathy & Jayamohan, 2012; Rowshanbin et al., 2006; Sun & Wun, 2009; Vahdati et al., 2010). Other possibilities consist in exploring the power of parallel architectures like nCUBE (Jackson & Blom, 1995), SIMD (Single Instruction Multiple

Data) (Khan & Akhtar, 2013; Wakatani, 2012) processors and clusters (Righi, 2012; Qureshi & Hussain, 2008). The use of multitasking on recent computing systems is a possibility not deeply explored for solving the FIC problem (Cao & Gu, 2010; Cao & Gu, 2011). The authors of these last initiatives presented an OpenMP solution that was tested over a quad-core processor. Besides multicore, we are focusing our attention on SMT (Simultaneous Multithreading) (Raasch & Reinhardt, 2003) capability, since both technologies are common on off-the-shelf computers. Some researchers affirm that we will have tens or hundreds of cores, each one with multiple execution threads (Note 1), inside a processor in the next years (Diamond et al., 2011; Rai et al., 2010). This emphasizes the significance of modeling applications for such architectures.

The improvement in performance obtained by using multicore and SMT technologies depends on the software algorithms and their implementations. Task granularity, threads synchronization and scheduling, memory allocation, conditional variables and mutual exclusion are parameters under user control that must be carefully analyzed for extracting the power of these technologies in a better way. In this context, the present paper describes the FIC technique and its threads-based implementation. The FIC problem allows a program organization without data dependencies among the threads, which is special useful for observing key performance factors on parallel machines. Therefore, we modeled an embarrassingly parallel application by exploiting data-parallelism on the aforemesaid problem. Contrary to (Cao & Gu, 2010; Cao & Gu, 2011), we obtained the results by varying the input image, the application parameters as well as the target machine. Particularly, we used two dual-core machines, one with and another without SMT capacity. In this case, SMT doubles the number of execution threads from 1 per core to 2, increasing processor throughput by multiplexing the execution threads onto a common set of pipeline resources. Our evaluation confirmed gains up to 29% when enabling SMT. Besides computer architecture information, this paper also discusses the impact of the number of threads and task granularity on the obtained results.

This paper is organized as follows. Section 2 describes the two traditional approaches for image compression. The FIC method is presented in Section 3 in details. Section 4 shows the parallel modeling proposed for the FIC problem, while Section 5 describes its implementation. The tests and the discussion of the results are presented in Section 6. Section 7 presents some related works. Finally, Section 8 points out the concluding remarks, future works and emphasizes the main contribution of the work.

2. Image Compression

A Pixel is the minimum unit to define an image. A digital image is a bi-dimensional matrix composed by a set of pixels whose spatial resolution is $I \times J$, where both I and $J \in \mathrm{N}$ and corresponding matrix element value identifies a set of discretized attributes (ex. gray level, color, transparency, and so on). Consequently, the larger the size of the image, greater will be the number of its pixels and attribute discretization, where each pixel is represented by a collection of bits, normally 16, 24 or 32 bits. In case, 16 Mbytes of memory are required to store a single image of 2048 × 2048, with 32 bits/pixel. In addition, some square images obtained by researchers can present dimensions up to 106, which turns clear the importance of the image compression field. We could classify the compression process in two subprocesses: (i) lossless compression and; (ii) lossy compression.

2.1 Lossless Compression

Situations in which the information needs to be kept intact after uncompressing usually employ Lossless compression. Medical images, technical drawings or texts are examples of using the lossless approach (Chen & Chuang, 2010). First, this process consists in transforming an input image $f(x)$ in $f'(x)$. According to Fu and Zhu (2009), this transformation can include differential and predictive mapping, unitary transforms, sub-band decomposing and color spacing conversion. After that, the data-to-mapping stage converts the $f'(x)$ in symbols, using the partitioning or run-length coding (RLC).

Lossless symbol coding stage generates a bit-stream by assigning binary codewords to symbols that were already mapped. Lossless compression is usually achieved by using variable-length codewords. This variable-length codeword assignment is known as variable-length coding (VLC) and also as entropy coding. Figure 1 depicts the process for obtaining a compressed image through the lossless method. Such method is used on algorithms for producing BMP, TGA, TIFF and PNG-typed images.

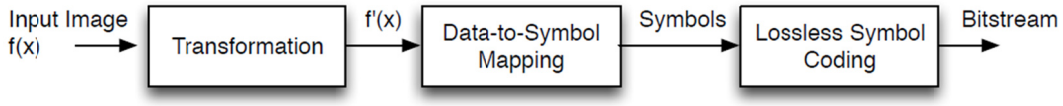

Figure 1. Common steps for compressing an image when using a lossless encoder

2.2 Lossy Compression

Lossy compression is an irreversible method, but yields better compact ratio results. After the uncompressing process, the resultant image will be almost identical to the original one, however never will be the same again (Jeng et al., 2009; Khan & Akhtar, 2013). Analogous to lossless compression, there are three stages on any lossy compression: (i) transform, (ii) quantization and; (iii) coding. The transformation stage reduces the correlation among pixels that results in a matrix of values. The quantizer stage is used to reduce quantity of bits per pixel, for example to transform color images into gray-scale ones. This process is irreversible and defines the loss level of the image quality. The stage coding process utilizes some methods that avoid more losses in the entire coding. Figure 2 depicts the functioning of the lossy compression methods. Among lossy techniques often used are Predictive coding, JPEG coding, and Fractal coding.

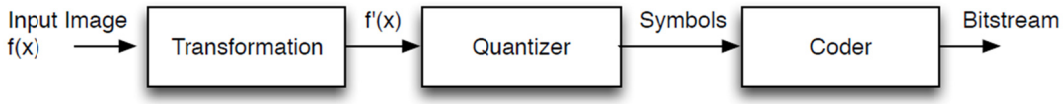

Figure 2. Common steps for image compression for a lossy-based encoder

3. Fractal Image Compression

Mandelbrot and Fisher described the concept of fractal, which are infinitely self- similar, iterated and described by mathematical formalisms (Chaurasia & Somkuwar, 2009; Wakatani, 2012). To better understand the feature of ”self-similarity”, for instance, we could make an analogy to zooming in with a lens or other device, which uncovers finer, previously invisible, new structure in digital images. Therefore, fractals are structures that have irregularities and fragmentations in a large range such as clouds, smokes, mountains and other nature elements. Fractal image compression is an example of asymmetrical methods. They take more time/effort compressing an image than decompressing it. The idea is to do most of the work during the compression, thus creating an output file that can be decompressed very quickly.

Coding phase of the Fractal Image Compression (FIC) method uses the self-similarity concept to represent image blocks through the transformation of coefficients (Jeng et al., 2009; George & Al-Hilo, 2009). This technique does not store or send blocks of pixels, but rather only functions that represent their transformations. This is because a fractal represents a shape that contains parts that are replicas among them- selves under some transformation aspect, as shown Figure 3 (a). The fractal compression is a technique based in the iterated function system (IFS) theory. To understand, consider the following formalism. Let F a gray-scale image with size $m \times m$, then F is divided in squares non-overlapped of size $r \times r$ called *Range Blocks*. The set of range blocks is R. Also, F is partitioned in squares $d \times d$ called *Domain Blocks*. The set of domain blocks is D. Usually $d = 2 \times r$ as shown in Figure 3 (b). In dark blue it is a subset of D and in light blue a subset of R. Then $R = \{R_0, R_1, ..., R_M\}$, with $M = (m \div r)^2 - 1$, and $D = \{D_0, D_1, ..., D_N\}$ with $N = (m \div d)^2 - 1$.

Contractive Mapping Fixed-Point theorem is the main idea behind a fractal encoding (Chaurasia & Somkuwar, 2009; Sharabayko & Markov, 2012). The theorem affirms “if a transformation is contractive then when applied repeatedly starting with any initial point, we converge to a unique fixed point”. Let X a complete metric space and f a function. If $f : X \rightarrow X$ is contractive, then f has a unique fixed point $|f|$. The Eq. (1) defines a contractive function, where s is a contractive factor with $0 < s < 1$ and o the offset value (in this case o represents brightness level).

$$f(x) = s(x) + o \tag{1}$$

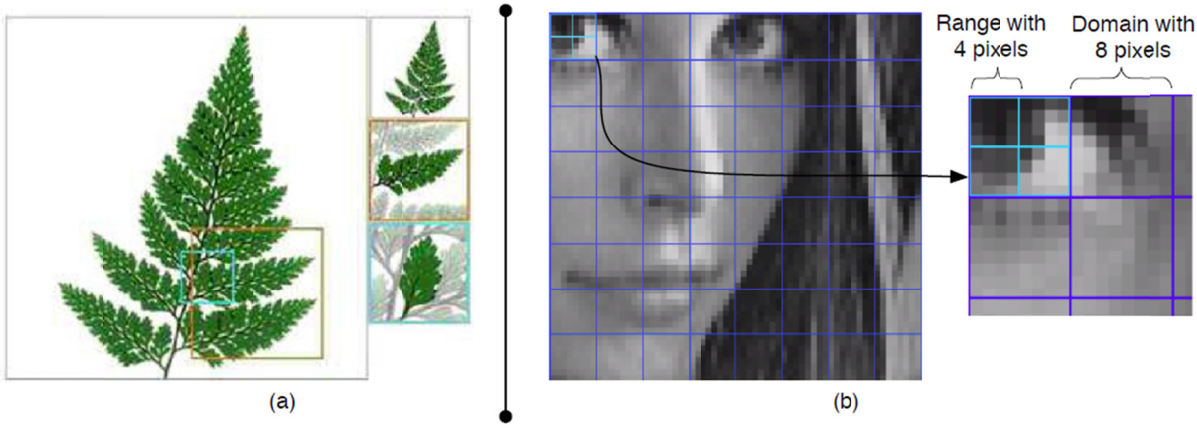

Figure 3. (a) Observing the recurrent pattern of fractals in the leaves of a plant; (b) Analyzing the relation of a subset R with $r = 4$ and a subset D with $d = 8$ in a sample image

Besides the fixed-point theorem, Affine Transformations (Sharabayko & Markov, 2012) are useful to adapt D_j to match to R_i. These transformations are performed in the following characteristics of the block D_j: (i) isometric (usually eight); (ii) scale and; (iii) contrast. An Affine Transformation is given by Eq. (2), where T() is an isometric transformation, s is a contractive scale and o means a specific brightness level. $S(D_j)$ represents a scale factor that contracts D_j size to R_i size. The values of s and o are defined in Eq. (3) and (4), respectively. The following works (Cao & Gu, 2011; Sharabayko & Markov, 2012; Wakatani, 2012) provide detailed discussion regarding these equations.

$$f(x) = s_i T_i\left(S(D_j)\right) + o \tag{2}$$

$$s_i = \frac{\sum S(D_j)\sum R_i - N^2 \sum S(D_j)R_i}{(\sum S(D_j))^2 - N^2 \sum S(D_j)^2} \tag{3}$$

$$o_i = \frac{\sum S(D_j)\sum R_i - \sum S(D_j)}{(\sum S(D_j))^2 - N^2 \sum S(D_j)^2} \tag{4}$$

$$MSE(R_i, R'_i) = \frac{1}{r^2}\sum_{k=0,l=0}^{r-1}(R_i(k,l) - R'_i(k,l))^2 \tag{5}$$

Considering the aforementioned context, IFS represents the operations performed over Affine Transformations. For any initial value of $D_j(x, y)$, there is a finite value n that will be the amount of iterative operations needed to get the fixed-point near to the original value $D_j(x, y)$. The main goal is to find a $D_j \in D$ which presents a high similarity to $R_i \in R$. In this context, the metric used to discover a best matching is a mean square error (MSE). Eq. (5) shows how the MSE is found. In this equation, $R_i(x, y)$ is an original pixel from F. R'_i is D_j modified by Affine transformations. Before evaluating MSE, each D_j is contracted in factor $r \div d$. This contraction can be used as the average among a group of nearest pixels as the absolute value of defined position in D_j. $S(D_j)$ will suffer all the eight standard isometric transformations: (i) rotation 0°; (ii) rotation 90°; (iii) rotation 180°; (iv) rotation 270°; (v) flip H; (vi) flip V; (vii) flip HV and; (viii) flip HV inverted. The next step is to define which D_j yields the minor MSE value. Finally, for each R_i will be found a D_j , an isometry, a scale and a brightness. According to Fu and Zhu (Fu & Zhu, 2009), this matching operation has complexity of $O[N^4]$.

4. Parallel Program Modelling

Commonly, we need to rewrite sequential programs to take the advantages of parallel architectures. Basically, a parallel program can follow one or a combination of the following paradigms: (i) message-passing in a multicomputer environment; (ii) multithreading programming by exploring multiprocessors (or multicore) systems; (iii) GPU (Graphical Processing Unit) programming on vectorial-based machines. This article presents a modeling approach in accordance with the second paradigm. In this context, the threads may communicate among themselves through a common shared-memory space in which they can both read from and write to. A multithreaded program can either define the launching of threads by using function calls explicitly or use library-assisted mechanisms for creating them implicitly. In the same way, the operating system is in charge of

scheduling each created thread to a specific processor (or core) without user intervention or he/she can aid the operating system with scheduling instructions.

In particular, gains are limited by the fraction of the software that can be run in parallel simultaneously on multiple cores. This effect is described by Amdahl's law. In the best case, so-called embarrassingly parallel problems may realize speedup factors near the number of cores, or even more if the problem is split up enough to fit within each core's cache(s), avoiding use of much slower main system memory. As already presented by Kim and Choi (Kim & Choi, 2011), FIC has a natural parallelism. Each comparison between ranges and domains is independent. Therefore, we modeled an application for exploiting data-parallelism upon this feature. The main idea is to start more than one FIC threads at the same time by not defining data dependencies among them. Each one will be responsible for a subset of the original image. Following Garcia and Gao (2013), embarrassingly parallel applications are ideal for parallel computers. Their basic argument concerns that it is possible to achieve higher speedups if the interprocess communication is either lower or non-existent. These authors affirm that this performance level is hardly matched by another application model. Thus, Figure 4 illustrates the proposed model when employing 4 threads.

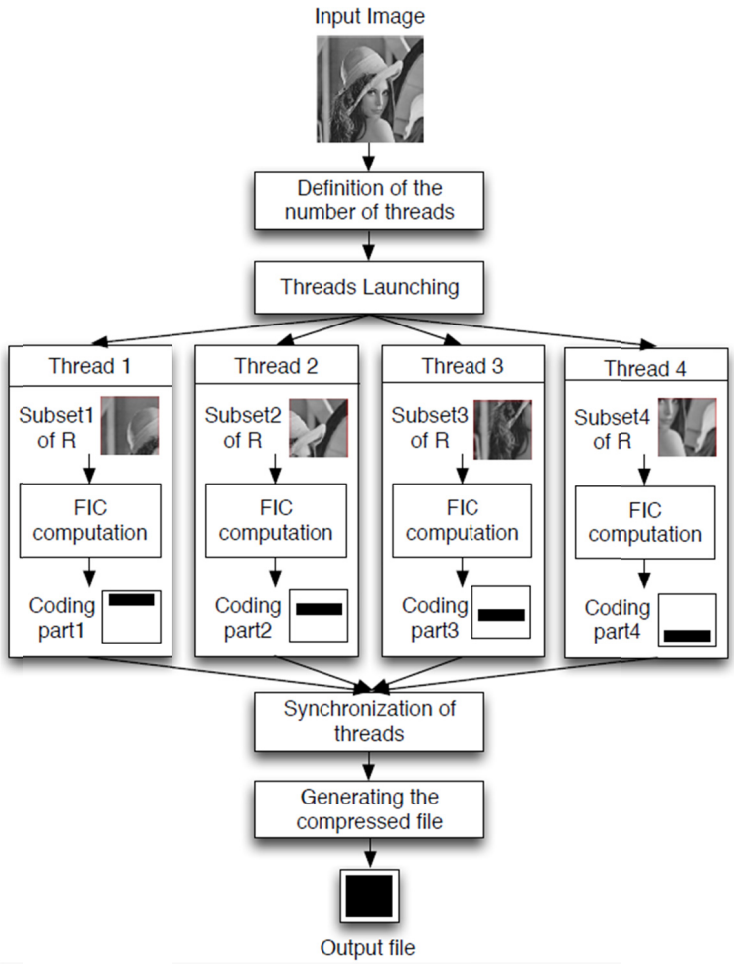

Figure 4. Example of the Parallel Model for the FIC problem with 4 threads

The first step of the model consists in splitting the original image in *y* equal subsets of *R* (see Section 3 for details). The value of y indicates the number of threads employed on compression coding. Each thread works with a whole set of *D* and must test all elements of this set against each range element received previously. Thus, each thread computes the FIC algorithm for its own block, generating both *s* and *o* sets. The final step concerns the appending of all blocks for generating the final compressed image. This task only occur after a synchronization point, which waits for the ending of all threads. Naturally, the performance depends on the number of cores or processors on a multiprocessing machine, as well as the granularity of the work. Commonly, the execution curve presents a performance peak when the number of threads is close to the processing elements in the system. A modeling without data dependencies is useful to concentrate the discussion in the following

issues: (i) what is the performance in terms of application time when employing different number of cores; (ii) which is the impact of r (dimension of the range) and m (dimension of the image) on multicore and SMT systems.

5. Application Development and Evaluation Methodology

Concerning the model explained earlier, we developed an application written in C programming language that uses the routines from Pthreads for enabling the threads facilities. Our implementation uses the following directive groups from this library: (i) thread management for creating, detaching, and joining threads; (ii) mutexes management for creating, destroying, locking, and unlocking mutexes; (iii) synchronization barriers. Figure 5 delineates the steps executed in the main program. Concerning the input image, the most references found in the computing graphics literature (Chen & Chuang, 2010; Garg, 2011; George & Al-Hilo, 2009; Jeng et al., 2009; Sundaresan & Devika, 2012) employed square-shaped gray-scale images. We have decided to keep this configuration and used two 24bpp BMP-typed input files. We are testing the application with different range of dimensions ($r \times r$): 2x2, 4x4, 8x8, 16x16 and 32x32. Basically, the shorter the dimension of the ranges, the larger the computational time to solve the FIC problem. This parameter has an impact on application execution, since the threads management overhead is the same when maintaining the number of them. Our application creates one, two, four, eight, sixteen, or thirty two threads. Each thread has the task of operating on a quadrant of the image. We evaluated each range belonging to the quadrant against all domains. In addition, tests are multiplied by 8 since each domain has 8 isometries.

```
Input: The number of threads "y" and a target image.
Output: Compressed Image.
1. Open the image and collects its dimensions.
2. Clones the image in order to compute the PSNR index after compressing.
3. Allocates a memory array to be send for each thread (range subset, domain set D)
4. Initialize a timer t1.
5. Launching of "y" threads for coding subparts of the image.
6. Synchronization procedure for receiving data from all threads.
7. Initialize a timer t2 and take the elapsed time between t2 and t1.
8. Collecting the results for generating the compressed image.
9. Decompress the resultant image in order to compare it with the original one.
```

Figure 5. The algorithm executed by the main program.

The peak signal-to-noise ratio (PSNR) (Sharabayko & Markov, 2012) measures the error or distortion between original image f and a decoded one f'. At the end of the decompress algorithm, the gerated image is compared to the original one, pixel-by-pixel, in order to compute the PSNR value. The Eq. (6) defines PSNR. MSE indicates the mean error and was defined in Eq. (5) previously. Several works use PSNR metric to qualify the reconstruction of a lossy compression method (Sharabayko & Markov, 2012; Kim & Choi, 2011; Sun & Wun, 2009; Qin et al., 2009). The parallel environment, in its turn, has two nodes, each one with a dual- core processor. Although they present the same number of cores, they present different compositions: (i) Intel E7500, 2.93 GHz, with 3 Mbytes of Cache L2; (ii) Intel i5-460M, SMT capacity with 4 execution threads, 2.53 GHz, 512 Kbytes Cache L2, 3 Mbytes Cache L3. The first configuration presents two execution threads, one per each processing core.

$$PSNR = 10 \times log_{10}\left(\frac{255^2}{MSE(f,f')}\right) \quad (6)$$

Simultaneous Multithreading is a way to virtualize one or more cores on a single one. At operating system level, a SMT-assisted dual-core processor will be reported as four logical processors. The main strength of SMT is that it allows for flexible scheduling of all available execution slots, which increases efficiency by keeping the execution core as busy as possible. For accomplishing this, the main function of SMT technology is to decrease the number of dependent instructions on the pipeline by taking advantage of a superscalar architecture (multiple instructions operating on separate data in parallel) (Diamond et al., 2011). Thus, SMT enables each core to handle multiple tasks by allowing one task to work while the other is waiting for a result, or allowing both instructions to be completed simultaneously because they use non-conflicting resources (Rai et al., 2010). Especially applications like FIC with data-parallelism, where multiple execution threads execute the same code on different sets of data, SMT can improve their performance in approximately 30% when compared with non-SMT solutions (Raasch & Reinhardt, 2003).

6. Experimental Results and Discussion

We have used two input images for performing our evaluation. The first refers to the Lenna (Note 2) picture and presents 256x256 pixels, while the second is a Coliseum photo with 512x512 pixels. Each experiment was run 30 times and we got the mean value and the standard deviation. Considering all the tests, the highest standard deviation for the 256x256 image as 2.78% from the average, while 1.51% was the index obtained for the 512x512 input. We started the time counter before launching the first thread and stopped it after finalizing the execution of all threads. This method discarded sequential code in the measures. Table 1 presents the obtained PSNR when varying the number of ranges. The number of threads does not matter for evaluating this index since the output image is always the same. The 2x2-sized range achieved the best results resulting from its better entropy when compared to larger ranges. Visually, images with PSNR greater than 21 have a good visualization capacity for human beings (Türkan et al., 2012). We achieved a compression rate of 2:1 in both images when employing a range with dimensions 2x2. However, 234:1 and 250:1 compression ratios were observed for 32x32-sized ranges when manipulating Lenna and Coliseum images, respectively.

Table 1. Analyzing the obtained PSNR (measured in decibels) for both evaluated images.

Input Image	Dimension of ranges				
	2x2	4x4	8x8	16x16	32x32
Lenna	35	31	26	22	19
Coliseum	38	27	22	19	18

Tables 2 and 3 present the evaluation of both input images when using a dual-core machine without SMT facility. As expected, the best results appear when testing 2 or 4 threads. For example, when testing only one thread with a range dimension equal to 4 the result was 6.57 seconds. This configuration does not take profit of the parallel machine. However, the execution with 32 threads presented the highest execution time when comparing executions of multiple threads. This behavior is explained by the overhead of mutex, synchronization and thread management primitives. The larger the number of threads, the higher this overhead. This elucidates a common behavior on evaluating threads on dual-core processors, where the application time decreases abruptly with 2 and 4 threads and grows up slowly when enlarging the number of threads. Figure 6 illustrates the speedup (*sequential_time* ÷ *parallel_time*) and the parallel efficiency (*Speedup* ÷ *processors*) for the tests with 2x2 range. Our application presents a poor speedup because the number of threads is greater than the number of execution cores. This statement becomes clear in the efficiency graph. Considering that we have only 2 physical cores, the execution with two threads presented the highest efficiency (92%). The execution with 4 up to 32 threads expresses the dilemma of concurrence, since each pair of threads competes for a single processor.

Figure 7 depicts the speedup evaluation results of the Coliseum image over a dual-core machine. This image presents a larger computation grain if compared with the Lenna one. In other words, the overhead associated with threads are better amortized when testing the Coliseum image since each thread has more work to compute in comparison with the other image. In this way, the execution with 2 threads reaches indexes up to 1.97 of speedup which is considered a good measure since the ideal speedup for this configuration is 2. Besides this analysis, it is possible to observe other two behaviors in the graph of Figure 7. Firstly, the larger the dimension of the ranges, the lower the captured speedup. For example, the execution with a range of 32x32 presents a lowest computation grain per each thread. Secondly, we can observe an execution pattern among the threads. Independent of the number of threads, the speedup curve presents the same aspect.

Table 2. Evaluating a dual-core processor without SMT support with a 256x256-sized image (Lenna) - Time in seconds.

Range	Sequential	Threads				
		2	4	8	16	32
2x2	45.379	24.899	25.352	25.583	25.490	25.548
4x4	6.576	3.523	3.459	3.520	3.540	3.562
8x8	1.249	0.834	0.779	0.790	0.796	0.825
16x16	0.268	0.219	0.213	0.219	0.226	0.245
32x32	0.058	0.059	0.062	0.069	0.081	0.117

Table 3. Evaluating a dual-core processor without SMT support with a 512x512-sized image (Coliseum) - Time in seconds.

Range	Sequential	Threads				
		2	4	8	16	32
2x2	682.097	346.828	343.326	344.427	347.849	350.961
4x4	114.911	59.910	58.306	58.450	59.042	59.792
8x8	21.523	10.948	11.028	11.076	11.234	11.309
16x16	4.563	2.476	2.442	2.492	2.525	2.593
32x32	1.023	0.661	0.623	0.637	0.665	0.725

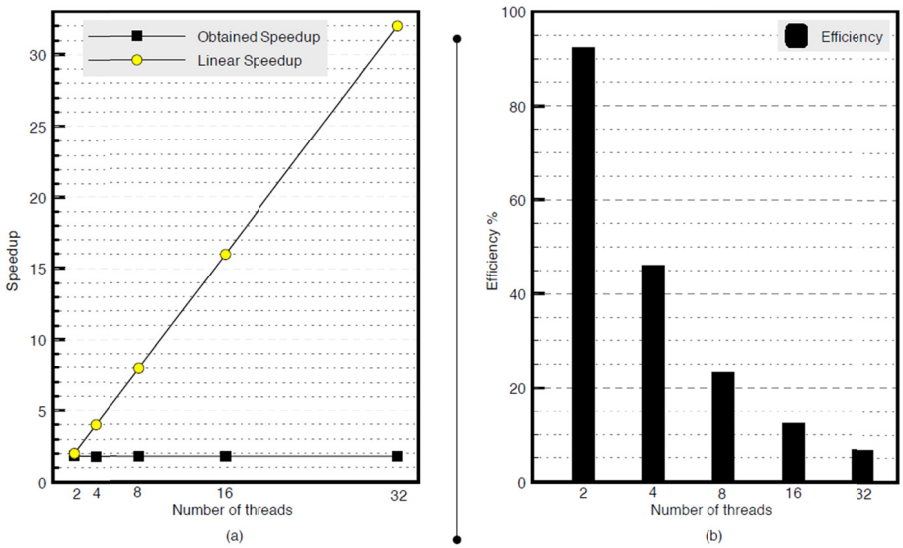

Figure 6. (a) Speedup and (b) parallel efficiency when using a 256x256-sized image, 2x2 range and a dual-core machine without SMT support

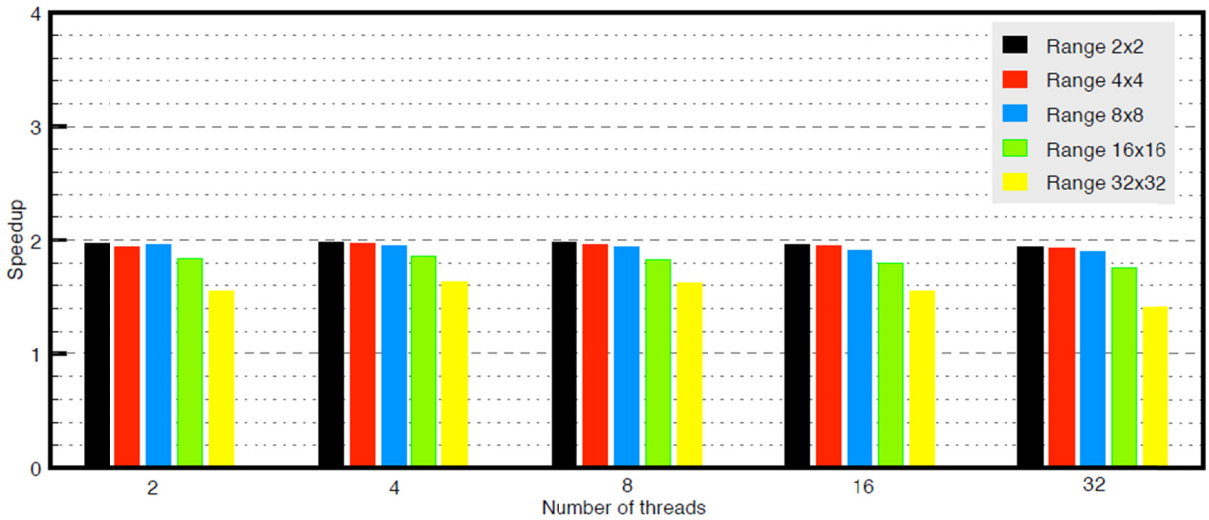

Figure 7. Speedup with a 512x512-sized image and a dual-core machine without SMT support

Both Tables 4 and 5 present the results when changing the infrastructure to the processor with SMT support. Different from the evaluation on Tables 2 and 3, the employment of 4 execution threads favors the execution time when using 4 threads significantly. Figure 8 shows the gain measured by *sequential_time* ÷ *parallel_time* when analyzing the coliseum picture with SMT-assisted dual-core processor. The performance of two threads

obtained gains up to 1.98, which are considered a good measure for this number of threads. However, we can observe that the use of 2 threads does not take profit from the entire power of the parallel architecture, since an execution thread remains allocated per core. The execution with 4 up to 32 cores took profit from the SMT solution. Particularly, we obtained a gain of 3.05 when testing 4 threads and ranges with dimension 16x16, which represents more than 75% of usage considering the execution threads inside the cores. The most relevant verifications concern the execution with 2x2 ranges. As we can see in Figure 8, the performance of this configuration does not scale well when treating for 4 or more threads. The calculus with this dimension of ranges is more computationally intensive than others. Furthermore, interactions require more memory since the subset of ranges belonging to each thread is larger than other range configurations. Clearly, any of the following observations causes a system bottleneck (Diamond et al., 2011): (i) memory contention; (ii) cache miss; (iii) concurrent access to components in the superscalar pipeline of the SMT core.

Table 4. Evaluating a SMT dual-core processor with a 256x256-sized image (Lenna) - Time in seconds.

Range	Sequential	Threads				
		2	4	8	16	32
2x2	39.872	23.804	20.735	20.493	20.518	20.717
4x4	6.241	3.190	2.700	2.704	2.726	2.763
8x8	1.476	0.768	0.612	0.557	0.562	0.597
16x16	0.345	0.191	0.159	0.157	0.163	0.213
32x32	0.077	0.046	0.045	0.050	0.068	0.112

Figure 9 illustrates a comparison graph considering both configuration of dual-core processors and the Lenna image. Although the SMT processor operates with 4 execution threads, our evaluation showed that the best results were obtained with 8 user threads. This combination was the best one for enlarging the efficiency regarding the cores utilization. Despite a large number of threads rises the operating system time for both managing and scheduling them efficiently, the threads are useful for exploiting superscalar and preemption facilities found on SMT processors. Logically, the number of threads must be analyzed with the thread granularity. In out case, 8 threads and 8x8 ranges compose the set with better performance. Finally, Figure 10 depicts the tests in which a range of 32x32 pixels and the Coliseum image were employed. This configuration points out the traditional curve when working with threads. We have a perceptible reduction in time when enabling threads and the time grows up when enlarging the number of threads as well. This is explained by computational work grain. The larger the number of threads, the lower the grain to be calculated by each thread (each thread receives a subset of ranges uniformly). In addition, more threads implies in a higher cost on synchronization and mutex primitives.

Table 5. Evaluating a SMT dual-core processor with a 512x512-sized image (Coliseum) - Time in seconds.

Range	Sequential	Threads				
		2	4	8	16	32
2x2	627.877	329.060	281.961	273.816	271.517	272.759
4x4	126.557	64.227	43.187	43.158	43.340	43.657
8x8	24.872	13.404	8.513	8.527	8.604	8.751
16x16	5.869	2.977	1.946	1.967	1.993	2.103
32x32	1.381	0.711	0.523	0.512	0.536	0.613

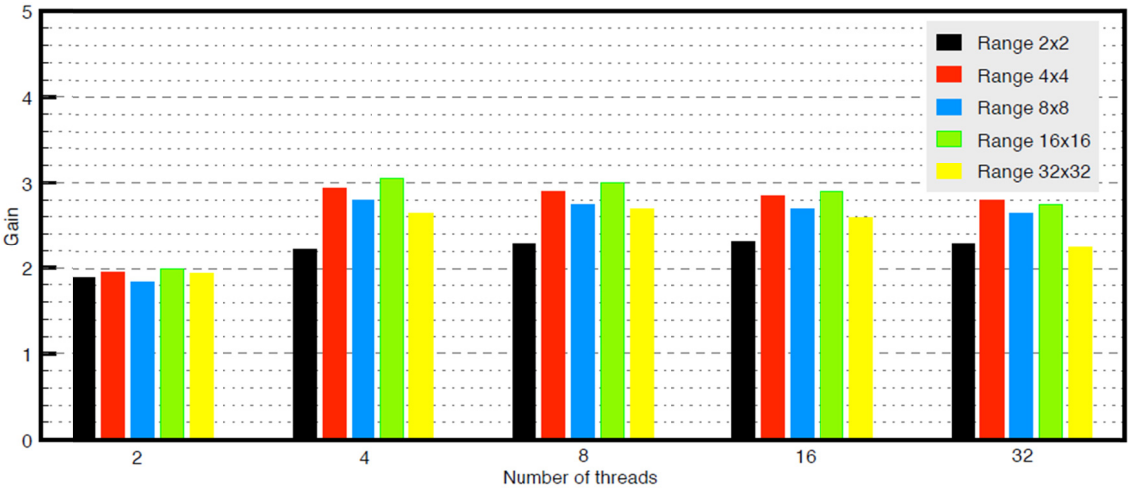

Figure 8. Evaluating the Coliseum figure with the SMT-assisted dual-core. The gain in *y* axis is equal to *sequential time* ÷ *parallel time*

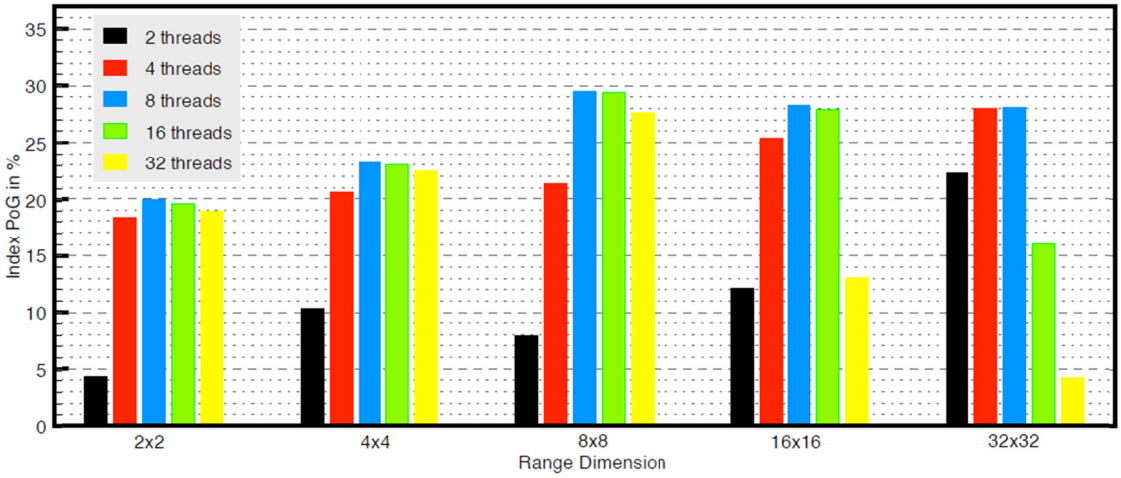

Figure 9. Evaluating both dual-core configuration with the Lenna image. Percentage of Gain (PoG) is computed as (1 - *SMT_dual-core* ÷ *Non_SMT_dual-core*).100

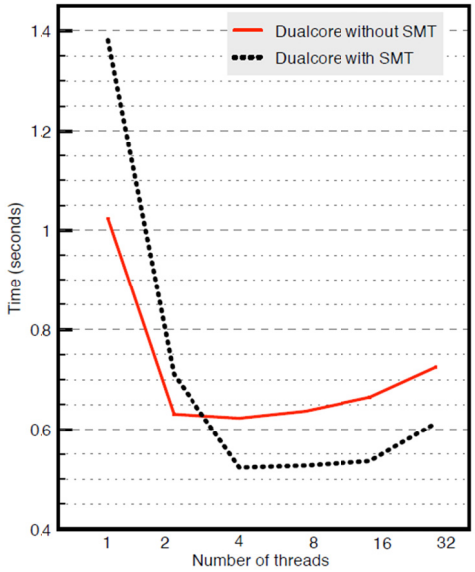

Figure 10. Evaluating the compression of Coliseum picture with 32 threads on both non-SMT and SMT-assisted dual-core systems.

7. Related Work

FIC technique has grabbed much attention in recent years because of manifold advantages, very high compression ratio, high decompression speed, high bit-rate and resolution independence. There have been many techniques, and improvements published in this field since 1990. Most of them are focused on some algorithm improvements for a smart search, which both reduce the size of search pool for range-domain matching and yield a significant speedup in execution time (Fu & Zhu, 2009; Jeng et al., 2009; Mitra et al., 1998; Qin et al., 2009; Revathy & Jayamohan, 2012; Rowshanbin et al., 2006; Sun & Wun, 2009; Vahdati et al., 2010). In particular, Revathy and Jayamohan (2012) proposed a dynamic preparation of a domain pool for each range block, instead of working with a set of static domains from the beginning of the execution (Revathy & Jayamohan, 2012). Vahdati et al. (2010) presented a Chaotic particle swarm optimization (CPSO) based on the characteristics of fractal and partitioned iterated function system. In addition, Ant Colony (Li et al., 2008), Neural Networks (Sun et al., 2001) and Genetic Algorithm (Mitra et al., 2000; Mitra et al., 1998; Wu & Lin, 2010) techniques were proposed to greatly decreases the search space for finding the self similarities in the given image. Contrary of exploring a reduction in the application time, Selim et al. focused on procuring a high compression index by maintaining a peak signal to noise ratio (PSNR) larger than 30 (Selim et al., 2008).

Regarding the exploration of parallel architectures, for the best of our knowledge there are the following initiatives for solving the FIC problem. Jackson and Blom (1995), based in a nCUBE multiprocessor, showed a parallel solution implementing a “host and nodes” solution, where a single processor was dedicated for distributing the workload to nodes and gathering results. Another message-passing solution were proposed by Qureshi and Hussein (2008), who implemented a three static master-worker MPI (Message Passing Interface) strategies for enabling load balancing on a Beowulf cluster of workstations. The authors measured both the speedup and the worker idle time of each implementation. Other features used in the context of FIC, considering a multicomputer environment, were Web Services (Fang et al., 2011) and process migration (Righi, 2012). Particular, this second work applies process rescheduling in grid environments for dealing with architecture heterogeneity and application dynamicity. Some works explore SIMD (Single Instruction Multiple Data) architectures, and more especially GPU (Graphical Processing Unit) (Wakatani, 2012; Khan & Akhtar, 2013). Kim and Choi (2011) combined both GPU and multithreading in their 2D DCT (discrete cosine transform) solution for the FIC problem (Kim & Choi, 2011). The article focused on the OpenCL parallel modeling. The authors just used an Intel core 2 Duo for the tests. Cao and Gu (Cao & Gu, 2010; Cao & Gu, 2011) presented a multithreading-based FIC implementation with OpenMP library by putting pragma codeword on iterative constructions simply. Albeit they pointed out a multicore implementation, the authors just presented tests with a quad-core system. Analyzing the contemplated related works, we observed a lack of studies on comparing the power of the recent multicore and SMT architectures for calculating the FIC problem. Hence, this opportunity of work was explored in this article.

8. Conclusion

With the help of recent development on semiconductor design, modern processors can provide a great opportunity to increase the performance on processing multimedia data by exploiting data-parallelism in multicore and SMT systems. Aiming to verify this statement, we employed in this article a parallel modeling of the so-called Fractal Image Compression (FIC) problem. Over the recent decades, FIC is a field of intensive research, applied not only in image processing but also in database indexing, texture mapping and pattern recognition problems. We designed a fork-join modeling to explore the fully potential of the parallel architecture, where each thread has a copy of the entire D (Domain) set and receives from the main program its own subset of ranges, which represents a subpart of the input image. The threads run without dependencies among themselves and are synchronized once for collecting the compressed image.

We confirmed the Garcia and Gao’s (2013) affirmation, that says applications with data-parallelism, where multiple threads execute the same code on different sets of data, can improve their performance dramatically when taking profit from SMT and multicore technologies. The results showed gains up to 68% (with SMT) and 48% (without SMT) when comparing multiple and single-thread scenarios in both configurations of dual-core processors. We can explain this rate by: (i) our modeling strategy and; (ii) fact that FIC is a CPU-bound problem. The benefits of data parallelism exploration were more evident in the SMT configuration. The use of 4 execution threads in SMT-assisted dual-core provided a performance gain up to 29% if compared to a non-SMT configuration. Particularly, we obtained this index with 8 user threads, which occupy each execution thread in a better way. In the best of our knowledge, this article is the first that presents a parallel FIC application focused on multicore and SMT systems, showing a detailed evaluation on them. Besides this, we can extend our contribution to operating systems. They can include the parallel FIC implementation proposed here as an

optional for compressing images, since multicore systems have become state-of-the-art in processor architecture field.

Finally, the tests allow us to conclude that the performance of a multithreading system depends on the computational grain on each thread, the number of processors in the target machine and the mutex/synchronization directives in the code. Future work comprises the execution of the FIC problem by modeling a message-passing application to execute over AMPI (Adaptive MPI) (Rodrigues et al., 2010). In this way, we intent to evaluate the problem with threads, with MPI solely and by combining both threads and MPI approaches.

Acknowledgments

The authors would like to thank to the following Brazilian agencies: CNPq, CAPES and FAPERGS.

References

Cao, H., & Gu, X. J. (2010). Openmp parallelization of jacquin fractal image encoding. In *E-Product E-Service and E- Entertainment (ICEEE), 2010 International Conference on*, pages 1-4. http://dx.doi.org/10.1109/ICEEE.2010.5661366

Cao, H., & Gu, X. Q. (2011). Implement research of fractal image encoding based on openmp parallelization model. In *Electric Information and Control Engineering (ICEICE), 2011 International Conference on*, 62–465. http://dx.doi.org/10.1109/ICEICE.2011.5777994

Chaurasia, V., & Somkuwar, A. (2009). Speed up technique for fractal image compression. In *Digital Image Processing, 2009 International Conference on*, pages 319–323. http://dx.doi.org/10.1109/ICDIP.2009.66

Chen, S., Cheng, X., & Xu, J. (2012). Research on image compression algorithm based on rectangle segmentation and storage with sparse matrix. In *Fuzzy Systems and Knowledge Discovery (FSKD), 2012 9th International Conference on*, pages 1904 –1908. http://dx.doi.org/10.1109/FSKD.2012.6233969

Chen, T. J., & Chuang, K.-S. (2010). A pseudo lossless image compression method. In *Image and Signal Processing (CISP), 2010 3rd International Congress on*, volume 2, pages 610 –615. http://dx.doi.org/10.1109/CISP.2010.5647247

Diamond, J., Burtscher, M., McCalpin, J. D., Kim, B. D., Keckler, S. W., & Browne, J. C. (2011). Evaluation and optimization of multicore performance bottlenecks in supercomputing applications. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '11, pages 32–43, Washington, DC, USA. IEEE Computer Society. http://dx.doi.org/10.1109/ISPASS.2011.5762713

Fang, Y., Cheng, H., & Wang, M. (2011). Parallel implementation of fractal image compression in web service environment. In *Distributed Computing and Applications to Business, Engineering and Science (DCABES), 2011 Tenth International Symposium on*, pages 59–63. http://dx.doi.org/10.1109/DCABES.2011.66

Fu, C., & Zhu, Z. L. (2009). A dct-based fractal image compression method. In *Chaos-Fractals Theories and Applications, 2009. IWCFTA '09. International Workshop on*, pages 439–443. http://dx.doi.org/10.1109/IWCFTA.2009.99

Garcia, E., & Gao, G. (2013). Strategies for improving performance and energy efficiency on a many-core. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13, pages 9:1–9:4, New York, NY, USA. ACM. https://doi.org/10.1145/2482767.2482779

Garg, A. (2011). Article: An improved algorithm of fractal image compression. *International Journal of Computer Applications*, 34(2):17–21. Published by Foundation of Computer Science, New York, USA.

George, L., & Al-Hilo, E. (2009). Fractal color image compression by adaptive zero-mean method. In *Computer Technology and Development, 2009. ICCTD '09. International Conference on*, volume 1, pages 525 –529. http://dx.doi.org/10.1109/ICCTD.2009.150

Jackson, D. J., & Blom, T. (1995). A parallel fractal image compression algorithm for hypercube multiprocessors. In *Proceedings of the 27th Southeastern Symposium on System Theory (SSST'95)*, SSST '95, pages 274–, Washington, DC, USA. IEEE Computer Society. http://dx.doi.org/10.1109/SSST.1995.390570

Jeng, J. H., Tseng, C. C., & Hsieh, J. G. (2009). Study on huber fractal image compression. *Image Processing, IEEE Transactions on*, 18(5):995 –1003. http://dx.doi.org/10.1109/TIP.2009.2013080

Khan, S., & Akhtar, N. (2013). Parallelization of fractal image compression over cuda. In Das, V. V., editor, *Proceedings of the Third International Conference on Trends in Information, Telecommunication and*

Computing, volume 150 of Lecture Notes in Electrical Engineering, pages 375–382. Springer New York. http://dx.doi.org/10.1007/978-1-4614-3363-7_42

Kim, C. G., & Choi, Y. S. (2011). Exploiting multi- and many-core parallelism for accelerating image compression. In *Multimedia and Ubiquitous Engineering (MUE), 2011 5th FTRA International Conference on*, pages 12–17. http://dx.doi.org/10.1109/MUE.2011.13

Li, J., Yuan, D., Xie, Q., & Zhang, C. (2008). Fractal image compression by ant colony algorithm. In *Young Computer Scientists, 2008. ICYCS 2008. The 9th International Conference for*, pages 1890–1894. http://dx.doi.org/10.1109/ICYCS.2008.222

Mitra, S., Murthy, C., & Kundu, M. (1998). Technique for fractal image compression using genetic algorithm. *Image Processing, IEEE Transactions on*, 7(4):586–593. http://dx.doi.org/10.1109/83.663505

Mitra, S., Murthy, C., & Kundu, M. (2000). Image compression and edge extraction using fractal technique and genetic algorithm. In Pal, S., Ghosh, A., and Kundu, M., editors, *Soft Computing for Image Processing*, volume 42 of *Studies in Fuzziness and Soft Computing*, pages 79–100. Physica-Verlag HD. https://doi.org/10.1007/978-3-7908-1858-1_4

Pinto, S., & Gawande, J. (2012). Performance analysis of medical image compression techniques. In *Internet (AH-ICI), 2012 Third Asian Himalayas International Conference on*, 1-4. http://dx.doi.org/10.1109/AHICI.2012.6408455

Qin, F. Q., Min, J., Guo, H. R., & Yin, D. H. (2009). A fractal image compression method based on block classification and quadtree partition. In *Computer Science and Information Engineering, 2009 WRI World Congress on, 1*, 716–719. http://dx.doi.org/10.1109/CSIE.2009.230

Qureshi, K., & Hussain, S. S. (2008). A comparative study of parallelization strategies for fractal image compression on a cluster of workstations. *International Journal of Computational Methods, 5*(3), 463–482. http://dx.doi.org/10.1142/S0219876208001534

Raasch, S. E., & Reinhardt, S. K. (2003). The impact of resource partitioning on smt processors. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, PACT '03, 15, Washington, DC, USA. IEEE Computer Society. http://dx.doi.org/10.1109/PACT.2003.1237998

Rai, J. K., Negi, A., Wankar, R., & Nayak, K. D. (2010). Performance prediction on multi-core processors. In *Proceedings of the 2010 International Conference on Computational Intelligence and Communication Networks*, CICN '10, pages 633–637, Washington, DC, USA. IEEE Computer Society. http://dx.doi.org/10.1109/CICN.2010.125

Revathy, K., & Jayamohan, M. (2012). Dynamic domain classification for fractal image compression. CoRR, abs/1206.4880. http://doi.org/10.5121/ijcsit.2012.4208

Righi, R. R. (2012). *Process Migration in Grid Computing: Combining Multiple Metrics to Control Process Rescheduling in Response to Resource and Application Dynamics*. Lambert Academic Publishing.

Rodrigues, E. R., Navaux, P. O. A., Panetta, J., Fazenda, A., Mendes, C. L., & Kale, L. V. (2010). A Comparative Analysis of Load Balancing Algorithms Applied to a Weather Forecast Model. In *Proceedings of 22nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Itaipava, Brazil. http://dx.doi.org/10.1109/SBAC-PAD.2010.18

Rowshanbin, N., Samavi, S., & Shirani, S. (2006). Acceleration of fractal image compression using characteristic vector classification. In *Electrical and Computer Engineering, 2006. CCECE '06. Canadian Conference on*, pages 2057–2060. http://dx.doi.org/10.1109/CCECE.2006.277529

Selim, A., Hadhoud, M., Dessouky, M., & El-Samie, F. (2008). A simplified fractal image compression algorithm. In *Computer Engineering Systems, 2008. ICCES 2008. International Conference on*, 53–58. http://dx.doi.org/10.1109/ICCES.2008.4772965

Sharabayko, M. P., & Markov, N. G. (2012). Fractal compression of grayscale and color images: Tools and results. In *Strategic Technology (IFOST), 2012 7th International Forum on*, 1–5. http://dx.doi.org/10.1109/IFOST.2012.6357622

Sun, K., Lee, S., & Wu, P. (2001). Neural network approaches to fractal image compression and decompression. *Neurocomputing, 41*(14), 91-107. http://dx.doi.org/10.1016/S0925-2312(00)00349-0

Sun, Z., & Wun, Y. (2009). Multispectral image compression based on fractal and k-means clustering. In *Proceedings of the 2009 First IEEE International Conference on Information Science and Engineering*,

ICISE '09, pages 1341–1344, Washington, DC, USA. IEEE Computer Society. http://dx.doi.org/10.1109/ICISE.2009.772

Sundaresan, M., & Devika, E. (2012). Image compression using h.264 and deflate algorithm. In *Pattern Recognition, Informatics and Medical Engineering (PRIME), 2012 International Conference on*, 242-245. http://dx.doi.org/10.1109/ICPRIME.2012.6208351

Türkan, M., Thoreau, D., & Guillotel, P. (2012). Self-content super-resolution for ultra-hd up-sampling. In *Proceedings of the 9th European Conference on Visual Media Production*, CVMP '12, 49-58, New York, NY, USA. ACM. https://doi.org/10.1145/2414688.2414695

Vahdati, G., Yaghoobi, M., & Akbarzadeh-T, M. (2010). Fractal image compression based on particle swarm optimization and chaos searching. In *Computational Intelligence and Communication Networks (CICN), 2010 International Conference on*, 62-67. http://dx.doi.org/10.1109/CICN.2010.23

Wakatani, A. (2012). Implementation of fractal image coding for gpu systems and its power-aware evaluation. In *Systems Conference (SysCon), 2012 IEEE International*, 1-5. http://dx.doi.org/10.1109/SysCon.2012.6189434

Wu, M. S., & Lin, Y. L. (2010). Genetic algorithm with a hybrid select mechanism for fractal image compression. *Digit. Signal Process., 20*(4), 1150–1161. http://dx.doi.org/10.1016/j.dsp.2009.12.009

Notes

Note 1. We used the term "execution threads" in the remaining of this document for treating SMT technology, while we employed just "threads" for denoting multiple execution entities created by a parallel application.

Note 2. Standard test image which has been in use since 1973 in the computer graphics area. http://www.cs.cmu.edu/~chuck/lennapg/editor.html