DOI 10.12694/scpe.v16i2.1090
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 16, Number 2, pp. 205–217. http://www.scpe.org

PARALLEL WATERMARKING OF IMAGES IN THE FREQUENCY DOMAIN*

DOROTHY BOLLMAN, ALCIBIADES BUSTILLO, EINSTEIN MORALES†

Abstract. While the internet has made it possible for the consumer to easily obtain images, audio, video, etc. in digital form, it has also made it easier to illegally obtain copyrighted material. Digital watermarking is a partial solution to this problem. Embedding a watermark in a legal version of material can help the copyright owner to identify who has an illegal copy. Because of the ever increasing enormity of the flow of information, it becomes necessary to watermark files in the least amount of time possible. For this reason it is natural to turn to parallel computing. In this work we compare the performance of three different implementations on a cluster of SMPs, in OpenMP, MPI, and CUDA, of a simple algorithm for watermarking digital images. Our experiments show that CUDA with one gpu is almost 300 times faster than the sequential version and many times faster than OpenMP and MPI using 1 up to 8 nodes.

Key words: parallel computing, digital watermarking, discrete cosine transform, frequency domain, OpenMP, MPI, CUDA.

AMS subject classifications. 68W10, 94A60, 68P25

1. Introduction. Watermarking is the process of embedding data into multimedia, including text, still images, video, or audio, that is typically used in order to show ownership. A watermark can be either perceptionally visible or invisible to the human eye. Visible watermarks are used to protect copyright or to simply identify a source of material such as a library or organization. A visible watermark identifies the owner of material, but does not necessarily prevent other uses. On the other hand, invisible watermarks are usually used in order to detect fraudulent use of material. For example, a seller might want to identify a person who has used his/her material without paying royalties or the government might want to detect the identity of a person who released classified material.

In order for a watermark to be useful, it must be either detectable or extractable by the owner. It must also be "robust" or resistant to attacks, either intentional or non-intentional. That is, the watermark must remain intact after attacks.

Although many different techniques for embedding watermarks in digital images have appeared in the literature for at least the last twenty years, only a few have considered the possibility of applying parallel computing and those that do consider only the use of GPUs. In this paper we give an embarrassingly parallel algorithm for a certain family of watermarking algorithms in the frequency domain and we compare performance of sequential, OpenMP, MPI, and CUDA implementations of a simple representative of this family, with particular emphasis on OpenMP and MPI.

In the following section, we briefly review several digital image watermarking algorithms that have been considered in the literature. In Sect. 3. we define the discrete cosine transform ("DCT") and describe the symmetries that we take advantage of in our implementations. In Sect. 4. we describe and compare our implementations of a watermarking procedure in OpenMP, MPI, and CUDA. In Sect. 5. we discuss experimental results of the three implementations. In Sect. 6. we make some concluding remarks.

2. Digital Image Watermarking. Digital image watermarking can be done either in the spatial domain or the frequency domain (or perhaps, as in [5], in both). Spatial domain techniques involve direct manipulation of the pixel values. For example, colors of certain pixels could be changed. Another very simple spatial domain technique is the "least significant bit" method in which a given number of least significant bits of the host image are replaced by the most significant bits of the watermark image. A frequency domain method consists of embedding the watermark in the "frequency domain", i.e., in the functional image of a discrete transform such as a Fourier, cosine, or wavelet transform.

Here we are interested in frequency domain methods and we briefly mention only a representative sampling of some of the work that has been done in this area. Although there are several works in which grid (GPU)

*This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1053575.

†Department of Mathematical Sciences, University of Puerto Rico at Mayagüez (dorothy.bollman, alcibiades.bustillo, einstein.morales)@upr.edu

computing has been applied to image watermarking, we know of no previous work involving the application of OpenMP and MPI.

A frequency domain method typically consists of three stages: (1) convert the host image I from the spatial domain to the frequency domain, i.e., compute the discrete transform $\mathcal{T}$, such as the Fourier, cosine, or wavelet transform, of I; (2) apply an embedding algorithm E to $\mathcal{T}(I)$ and $\mathcal{S}(W)$, where $\mathcal{S}$ is some function defined on the watermark W, to obtain a new image array $E = E(\mathcal{T}(I), \mathcal{S}(W))$; (3) Convert E back to the spatial domain to obtain the watermarked image E'. An extraction process consists of a procedure X which takes the watermarked image E' and possibly the original image I and produces the original watermark $W = X(E', I)$.

The DCT and IDCT (inverse DCT) of images used in frequency domain methods are applied either to blocks of the image or to the complete image. Shieh *et al* [10] develop a method in which a genetic algorithm is used to insert 8×8 blocks of the binary watermark into corresponding blocks of the DCT of the host image. The watermarked image then consists of the concatenation of the IDCT of each resulting block. Using this algorithm, the watermark can be extracted without the need for the original host image.

Garcia-Cano *et al* [2] implement the Shieh algorithm on a GPU and compare performance results with those of a sequential version. Other works in which GPUs have been used in watermarking are, for example, those of Lin, Zhoa, and Yang [6] and Vihari and Mishra [16]. Lin *et al* extract features from the low and middle frequency domain of the DCT and embed them in the high frequency domain. Vihari and Mishra use Huffman coding to encode copyright data that is then embedded using the "Modified Auxiliary Carry Watermarking" method.

Cox *et al* [1] develop an invisible watermark consisting of real numbers $x_1, x_2, \cdots, x_n$ that are chosen according to a normal distribution with mean 0 and variance 1. The DCT of the host image, as a single block, is computed and the most perceptually significant components, determined by the largest DCT coefficients, are replaced by $v_i(1 + \alpha x_i)$ where v_i is a frequency component of the host image and α is a scalar factor. The watermarked image then consists of the inverse DCT applied to this result.

A typical frequency domain procedure uses the DCT and partitions the image I and a logo watermark image W into 8×8 blocks of pixels The DCT of each block I_{ij} of the host image I is replaced $\alpha_{ij} I_{ij} + \beta_{ij} W_{ij}$ where $W_{ij}(n)$ is the DCT of the corresponding block of the watermark image and where α_{ij} and β_{ij} are values that are chosen in accordance with properties of block I_{ij}. The watermarked image $I(W)$ then results from applying the inverse DCT to each of the resulting blocks and concatenating the resulting blocks. In symbols

$$I(W) = \bigcup_{ij} IDCT(\alpha_{ij} DCT(I_{ij}) + \beta_{ij} DCT(W_{ij}))$$

Kankanhali *et al* [4] choose α_{ij} and β_{ij} according to perceptual methods developed by Tao and Dickinson [15]. Mohanty *et al* [8] claim to improve this latter procedure by taking texture into consideration. We call this type of watermarking procedure, an "$\alpha - \beta$" method. In an $\alpha - \beta$ method each pair of blocks (I_{ij}, W_{ij}) can be processed independently, i.e., such a method is "embarrassingly parallel".

Frequency domain watermarking is generally regarded as being more robust than spatial domain methods, mainly because of its resistance to lossy compression attacks. Indeed, the steps involved in frequency domain watermarking are very similar to those of image compression such as JPEG compression. Typically, these steps consist of the following: (1) partition the image into 8×8 blocks of pixels; (2) apply the two dimensional DCT to each such block; (3) apply "quantization" to each block of resulting DCT coefficients; (4) apply the inverse DCT to each of the blocks; (5) apply entropy coding to the quantized data. Quantization is a process of reducing the number of possible values of a quantity and entropy coding is a method for representing the quantized data in a compact form.

3. The Discrete Cosine Transform. The one-dimensional discrete cosine transform is a linear function $F : \mathbb{R}^N \to \mathbb{R}^N$ defined by

$$C(u) = \alpha(u) \sum_{x=0}^{N-1} f(x) \cos\left(\frac{\pi(2x+1)u}{2N}\right), u = 0, \ldots, N-1 \tag{3.1}$$

The inverse transform exists and is defined by

$$f(x) = \sum_{x=0}^{N-1} \alpha(u)C(u)\cos\left(\frac{\pi(2x+1)u}{2N}\right), x = 0, \ldots, N-1 \tag{3.2}$$

where

$$\alpha(u) = \begin{cases} \sqrt{\frac{1}{N}} & x = 0 \\ \sqrt{\frac{2}{N}} & x \neq 0 \end{cases} \tag{3.3}$$

The one-dimensional DCT is a linear function and can thus be represented as a matrix, i.e.,

$$[C_{ij}] = \left[\alpha(i)\cos\left(\frac{(2j+1)\pi i}{2N}\right)\right] \qquad i, j = 0, 1, \cdots, N-1$$

and similarly for the DCT inverse. The two-dimensional DCT is a function $F : \mathbb{R}^{N^2} \to \mathbb{R}^{N^2}$ which when applied to a matrix can be computed by first computing the one-dimensional DCT of the rows and then using the result to compute the one-dimensional DCT of the columns.

Since an $\alpha - \beta$ method applies the DCT to 8×8 blocks of images multiple times, it is of interest to minimize the number of operations in its computation. For this we use an idea of Obukhov and Kharlamov [9] which is described in the following.

The matrix form of the one-dimensional DCT exhibits various symmetries that can be taken advantage of. For $N = 8$ the representation is as follows:

$$F = \frac{1}{\sqrt{8}} \left[\begin{array}{cccc:cccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ a & c & d & f & -f & -d & -c & -a \\ b & e & -e & -b & -b & -e & e & b \\ c & -f & -a & -a & d & a & f & -c \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ d & -a & f & c & -c & -f & a & -d \\ e & -b & b & -e & -e & b & -b & e \\ f & -d & c & -a & a & -c & d & -f \end{array}\right]$$

$$a = \sqrt{2}\cos\left(\frac{\pi}{16}\right)$$

$$b = \sqrt{2}\cos\left(\frac{\pi}{8}\right)$$

$$c = \sqrt{2}\cos\left(\frac{3\pi}{16}\right)$$

$$d = \sqrt{2}\cos\left(\frac{5\pi}{16}\right)$$

$$e = \sqrt{2}\cos\left(\frac{3\pi}{8}\right)$$

$$f = \sqrt{2}\cos\left(\frac{7\pi}{16}\right)$$

Separating even and odd numbered rows we have

$$\begin{bmatrix} Y(0) \\ Y(2) \\ Y(4) \\ Y(6) \end{bmatrix} = \frac{1}{\sqrt{8}} \begin{bmatrix} 1 & 1 & 1 & 1 \\ b & e & -e & -b \\ 1 & -1 & -1 & 1 \\ e & -b & b & -e \end{bmatrix} \begin{bmatrix} X(0)+X(7) \\ X(1)+X(6) \\ X(2)+X(5) \\ X(3)+X(4) \end{bmatrix}$$

FIG. 4.1. *(a) Image (Lena), (b) Watermark (Barbara), (c) Watermarked image*

$$\begin{bmatrix} Y(1) \\ Y(3) \\ Y(5) \\ Y(7) \end{bmatrix} = \frac{1}{\sqrt{8}} \begin{bmatrix} a & -c & d & -f \\ c & f & -a & d \\ d & a & f & -c \\ f & d & c & a \end{bmatrix} \begin{bmatrix} X(0)-X(7) \\ X(2)-X(1) \\ X(4)-X(5) \\ X(5)-X(3) \end{bmatrix}$$

Thus, the one-dimensional DCT applied to a vector of length 8 can be computed using only 28 multiplications and 56 additions. The usual product of an 8×8 matrix times a vector of length 8 requires 64 multiplications and 56 additions.

4. Parallel Implementations of an $\alpha - \beta$ Frequency Domain Watermarking Algorithm. Given grayscale images of a host I and a watermark W, each of the same size, which we assume to be a power of 2, an $\alpha - \beta$ algorithm embeds W in I by (1) partitioning both I and W into blocks of 8×8 pixels and computing the DCT of each of the corresponding blocks, I_{ij} and W_{ij}; (2) computing $\alpha_{ij} = \alpha(I_{ij}, W_{ij})$ and $\beta_{ij} = \beta(I_{ij}, W_{ij})$; (3) replacing each value $DCT(I_{ij})$ in the host image by $\alpha_{ij}DCT(I_{ij}) + \beta_{ij}DCT(W_{ij})$; (4) computing the inverse DCT of each of the blocks $\alpha_{ij}DCT(I_{ij}) + \beta_{ij}DCT(W_{ij})$ and concatenating the results.

Following is an example where $\alpha = \alpha_{ij} = 0.9$ and $\beta = \beta_{ij} = 0.14$ for all i, j. The degree of visibility of the watermark is determined by the values of α and β.

Let us begin with a sequential version of this algorithm for an image of size $dim \times dim$ (Algorithm 1).

Algorithm 1 Sequential

```
1: procedure WATERMARKING(I, W, dim, α, β)
2:     m = dim/8
3:     for i = 0 : m − 1 do
4:         for j = 0 : m − 1 do
5:             X_ij = DCT(I_ij)
6:             Y_ij = DCT(W_ij)
7:             Compute α = α(I_ij, W_ij) and β = β(I_ij, W_ij)
8:             Z_ij = αX_ij + βY_ij
9:             Z_ij = IDCT(Z_ij)
       return Z
```

Each pair (I_{ij}, W_{ij}) of image blocks in the above algorithm can be processed independently of the others and hence in parallel and an efficient implementation consists of determining how the available resources can be used to most effectively achieve this. We shall see how this can be done in OpenMP, MPI, and CUDA.

4.1. OpenMP. OpenMP ("Open Multi-Processing") is a shared memory programming model in which the programmer can insert compiler directives or "pragmas" into ordinary sequential C, C++, or FORTRAN programs in order to partition tasks into parallel threads, the smallest unit of processing that can be scheduled by an operating system. The most common and easiest way to parallelize code in OpenMP is by parallelizing **for** loops.

In the above procedure WATERMARKING, each of the (i, j) iterations is independent of the others and so ideally, it would be convenient to use just one parallel **for** loop. However, because of the double indexing, we must use two nested **for** loops and in OpenMP it is possible to parallelize only the exterior **for** loop. However, recent versions of OpenMP allow one to parallelize multiple loops in a nest without introducing nested parallelism by making use of the collapse pragma. Thus, we have Algorithm 2.

The number of threads used is specified at run time.

Algorithm 2 OpenMP-Version

1: **procedure** WATERMARKING(I, W, dim, α, β)
2: $m = \frac{dim}{8}$
3: #**pragma omp parallel for collapse**(2)
4: **for** $i = 0 : m - 1$ **do**
5: **for** $j = 0 : m - 1$ **do**
6: $X_{ij} = DCT(I_{ij})$
7: $Y_{ij} = DCT(W_{ij})$
8: Compute $\alpha = \alpha(I_{ij}, W_{ij})$ and $\beta = \beta(I_{ij}, W_{ij})$
9: $Z_{ij} = \alpha X_{ij} + \beta Y_{ij}$
10: $Z_{ij} = IDCT(Z_{ij})$
return Z

4.2. MPI. MPI ("Message Passing Interface"), originally designed for distributed memory architectures, is a library of functions that can be used in conjunction with C, C++, or FORTRAN, as well as other languages, in order to effect communications between processors. In the MPI implementation of our watermarking algorithm we partition the grid of 8×8 blocks of both the host image as well as the watermark into n equal size strips of rows of blocks, where n is the number of processors. The processor, say p_0, that initially contains both images, I and W, sends to each of the n processors a pair of strips (one from the host image and one from the watermark). Each processor then processes its share of the two images as in the sequential version and then sends its result to p_0 which assembles the result into the complete watermarked image (Algorithm 3).

4.3. CUDA. CUDA ("Compute Unified Device Architecture"), created by Nvidia, is a parallel computing platform and programming model in which GPUs (Graphics Programming Units) can be accessed by programmers for general purpose computing through the use of CUDA-accelerated libraries, compiler directives, and extensions of C, C++, and FORTRAN, as well as other languages. A CUDA application can involve hundreds of cores and thousands of parallel threads. Parallel portions of an application can be defined as functions that are executed on the GPU. Such functions are called kernels. Threads are grouped into blocks of up to 512 threads, which in turn are organized into grids. All threads in a grid execute the same kernel. A grid can be one-, two-, or three-dimensional.

With CUDA we achieve our original goal of assigning a thread to each iteration of the nested **for** loop of the sequential version of our algorithm presented in Sect. 4. For this we make use of dct8×8_kernel2 of the Nvidia code [9] for computing the DCT and IDCT in 8×8 blocks, in which the grid constructed is 3-dimensional with 64 threads per block (Algorithm 4).

Algorithm 3 MPI-Version

procedure WATERMARKING(I, W, dim, α, β)
  n = number of processors
  $m1 = \frac{dim}{8}$
  $m2 = \frac{m1}{n}$
  send a section $I^{(p)}$ of $m2$ rows of 8×8 blocks of I to each processor p
  send a section $W^{(p)}$ of $m2$ rows of 8×8 blocks of W to each processor p
  for $i = 0 : m2 - 1$ **do**
    for $j = 0 : m1 - 1$ **do**
      $X_{ij}^{(p)} = DCT(I_{ij}^{(p)})$
      $Y_{ij}^{(p)} = DCT(W_{ij}^{(p)})$
      Compute $\alpha = \alpha(I_{ij}, W_{ij})$ and $\beta = \beta(I_{ij}, W_{ij})$
      $Z_{ij}^{(p)} = \alpha X_{ij}^{(p)} + \beta Y_{ij}^{(p)}$
      $Z_{ij}^{(p)} = IDCT(Z_{ij}^{(p)})$
  receive each $Z^{(p)}$ in Z

Algorithm 4 CUDA-Version

procedure WATERMARKING(I, W)
  for each thread-block in *block-grid* **do** in parallel
    $X = kernel\ DCT(I_{thread})$
    $Y = kernelDCT(W_{thread})$
    $\alpha = kernel\ \alpha(I_{ij}, W_{ij})$
    $\beta = kernel\ \beta(I_{ij}, W_{ij})$
    $Z_{thread} = kernel\ linearcomb(\alpha, X, \beta, Y)$
    $Z_{thread} = kernel\ IDCT(Z_{thread})$
  return Z

5. Experimental Results. For our actual implementations we chose a simple representative from the $\alpha - \beta$ family of algorithms in which the values of α and β are the same for all blocks (as for example in cf. Fig. 4.1), thus replacing the computation step for α and β in the above algorithms by inputs. Experiments were conducted on the Stampede supercomputer [11], located at the Texas Advanced Computer Center (TACC) and sponsored by the Extreme Science and Engineering Environment (XSEDE) with funding by the National Science Foundation. The Stampede system is a 10 PFLOPS (PF) Dell Linux Cluster consisting of 6,400 + Dell PowerEdge server nodes, each one of which has 2 Intel Xeon E5 (Sandy Bridge) processors and an Intel Xeon Phi Coprocessor (MICZ Architecture). For the OpenMP and MPI programs the authors used the 16 compute node "development queue", each node of which consists of 16 cores with 32 GB of shared memory. For the CUDA experiments, the authors used the "gpudev queue" consisting of 4 compute nodes each one of which is equipped with a NVIDIA K20GPU with 8GB of GDDR5 memory.

We tested our OpenMP, MPI and CUDA programs with greyscale images of 512, 1024, 2048, 4096, and 8192 square pixels. For OpenMP and MPI we used a maximum of all 16 cores per node. For CUDA we used one compute node with a GPU.

It will be noted that the behavior for 16 cores can sometimes be erratic, especially for small images. This is because some of a node's resources are necessarily dedicated to other instrinsic processes of the system.

5.1. OpenMP Results. The times for OpenMP are depicted in Fig. 5.1. Speedups for OpenMP are given in cf. Fig. 5.2.

As can be seen from Table 5.1 the OpenMP implementation has almost linear speedup for 2 up to 12 threads.

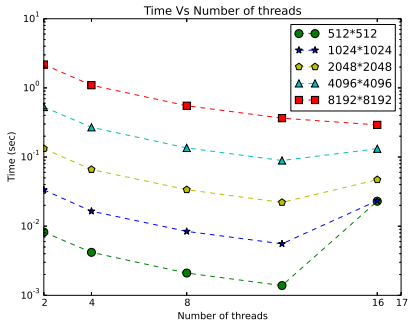

FIG. 5.1. *OpenMP times*

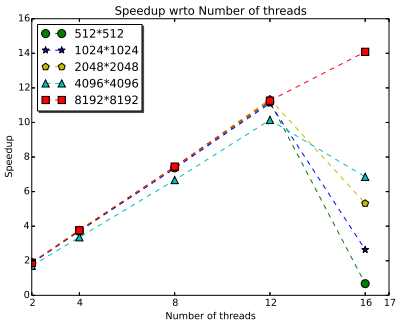

FIG. 5.2. *OpenMP Speedup*

TABLE 5.1
Speedup

threads	Speedup				
	512×512	1024×1024	2048×2048	4096×4096	8192×8192
2	1.884852	1.843737	1.893246	1.708771	1.885878
4	3.699033	3.747434	3.790393	3.357762	3.7578
8	7.372234	7.338334	7.427529	6.668273	7.44652
12	11.18421	11.10579	11.34707	10.14434	11.25846
16	0.674747	2.642157	5.317085	6.853785	14.08695

5.2. MPI Results. Times and speedups with respect to MPI tasks on 1, 2, 4, and 8 nodes are given in cf. Fig. 5.3 through 5.10.

Using all 16 cores on each node, speedups with respect to the number of nodes are given in cf. Fig. 5.11 and 5.12.

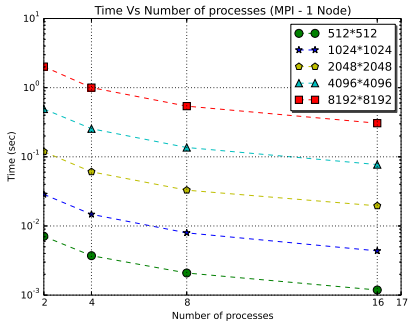

FIG. 5.3. *MPI 1 Node*

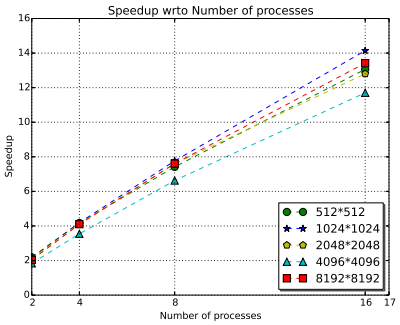

FIG. 5.4. *Speedup 1 Node*

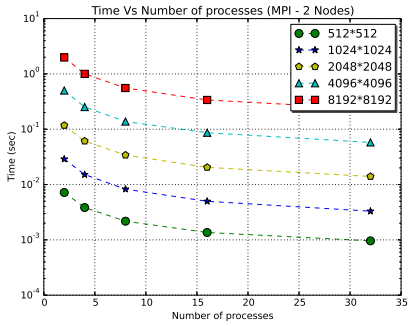

FIG. 5.5. *MPI 2 Nodes*

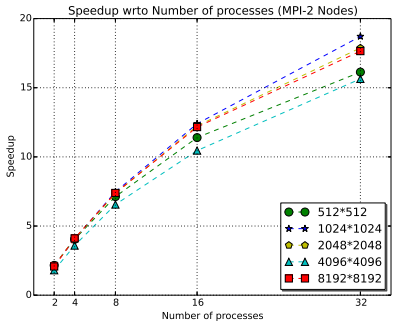

FIG. 5.6. *Speedup 2 Nodes*

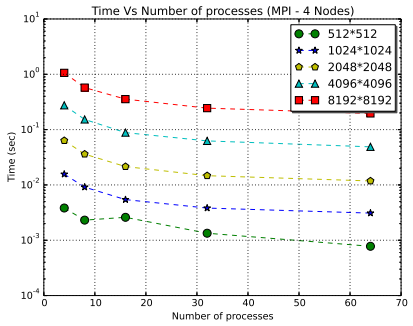

FIG. 5.7. *MPI 4 Nodes*

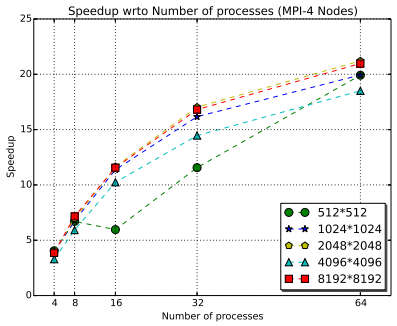

FIG. 5.8. *Speedup 4 Nodes*

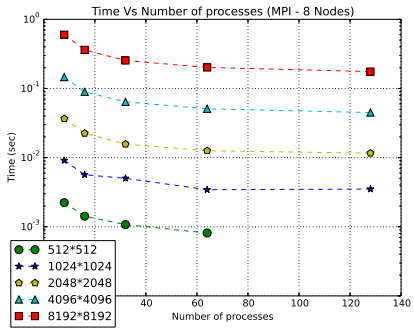

FIG. 5.9. *MPI 8 Nodes*

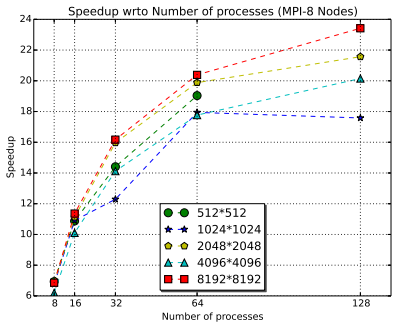

FIG. 5.10. *Speedup 8 Nodes*

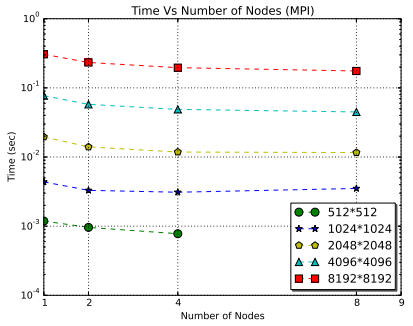

FIG. 5.11. *MPI times*

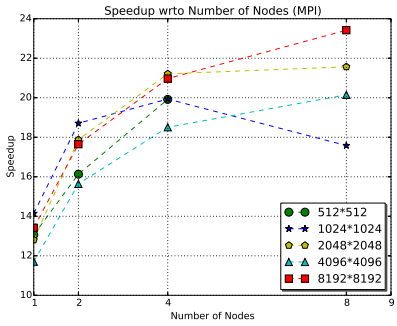

FIG. 5.12. *MPI Speedups*

As can be seen, the MPI implementation exhibits poor scalability with respect to the number of nodes (i.e., sequential time divided by the time for n nodes using all 16 cores), even when using all 16 cores. However, it is not true that the best time for a given number of nodes used corresponds to 16 processes (cores). But even using the best times, speedups with respect to the number of nodes is almost flat, as shown in Table 5.2 for an image of size 1024×1024.

TABLE 5.2
Speedup (1024×1024)

	Speedup (1024×1024)
1 node/ 16 processes	14.1429884
2 nodes/ 32 processes	18.7097302
4 nodes/ 64 processes	19.9331964
8 nodes/ 128 processes	17.5811589

On the other hand, when we compute speedups with respect to the number of processes (i.e., for a fixed number of nodes, the sequential time divided by the time for the number of processes), the best scalability occurs when we use just one node. In fact in this case as shown in the Table 5.3 we have superlinear speedup for 2 and 4 processes, except for size 4096×4096.

TABLE 5.3
Speedup (1 Node)

	Speedup				
processes	512×512	1024×1024	2048×2048	4096×4096	8192×8192
2	2.18684251	2.155103	2.110129	1.835898	2.037115
4	4.16952721	4.189678	4.1226	3.545565	4.100949
8	7.41785612	7.7622	7.600967	6.62734	7.605353
16	13.0625843	14.14299	12.80332	11.70284	13.42

5.3. OpenMP vs MPI. In cf. Fig. 5.13 through 5.17 we compare the performance of OpenMP for 2, 4, 8 and 16 threads with MPI on one node for 2, 4, 8, and 16 tasks, respectively and we see that the performance of the two are nearly the same up to 8 threads/tasks. For 16 threads/tasks, MPI wins for sizes up to 4096×4096, but the gap narrows for increasing image size and OpenMP wins by a very narrow margin for image size 8192×8192.

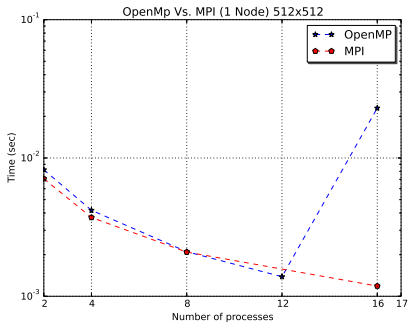

FIG. 5.13. *MPI vs. OpenMP (512×512)*

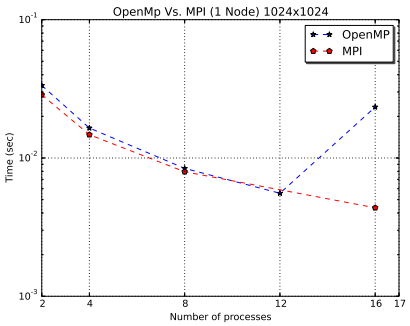

FIG. 5.14. *MPI vs. OpenMP (1024×1024)*

5.4. Hybrid. We developed a hybrid OpenMP MPI program in the standard way, by starting with the MPI version and distributing strips of 8×8 blocks of pixels among the nodes just as we did in the MPI version except

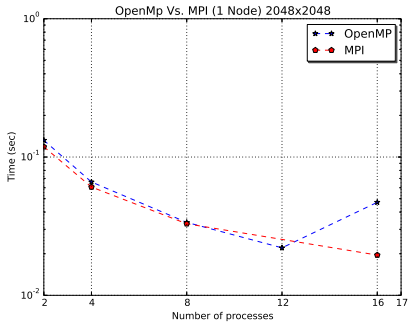

FIG. 5.15. *MPI vs. OpenMP (2048×2048)*

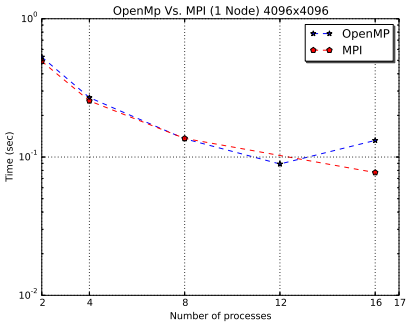

FIG. 5.16. *MPI vs. OpenMP (4096×4096)*

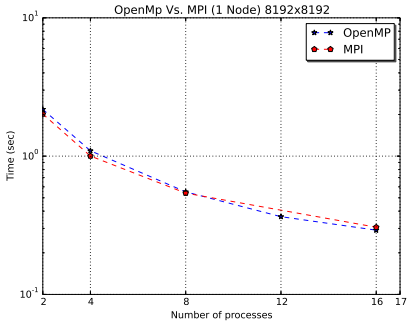

FIG. 5.17. *MPI vs. OpenMP (8192×8192)*

.

that now each node does its work in parallel using OpenMP as in the OpenMP-only version. Unfortunately, this version under performed both the OpenMP and MPI versions. This is really not surprising. Indeed, there are well known cases (see *e.g.*, [3]) where hybrid is slower than OpenMP and MPI. Hybrid improves performance by reducing communications between nodes and increasing opportunites for parallelism and neither of these opportunities exist in the MPI version of our algorithm. Furthermore, it turns out that the scatter and gather operations are many times slower in the hybrid version than in the MPI version when applied to the same sets of data.

5.5. CUDA Results. For the CUDA code, we used the optimized code of kernel2 described in [9] for computing the 8×8 DCT on a NVIDIA GeForce 6800. We added kernels for embedding the watermark. Further experiments are needed in order to determine if performance on the NVDIA K20 can be improved even further by manipulating block sizes. Comparisons between the four implementations are depicted in cf. Fig. 5.18. Our CUDA implementation on just one GPU was many times faster than both OpenMP and MPI on any number of nodes up to 8.

Table 5.4 gives the times for 4096×4096 images for the four different versions on just one node, where OpenMP uses 16 threads and MPI uses 16 processes.

Thus, on just one node, CUDA is 287 times faster than the sequential version, 42 times faster than OpenMP, and 24 times faster than MPI. The fastest time, .0448525 sec, for MPI occurred on 8 nodes using 128 processes. Thus CUDA on just one node is 14 times faster than MPI on 8 nodes.

5.6. Watermark Quality. There are several metrics that are commonly used to measure the the quality of watermark algorithms. The Peak Signal to Noise Ratio (PSNR) evaluates image degradation or reconstruction fidelity. It is defined for two images I and Z of size $M \times N$ as:

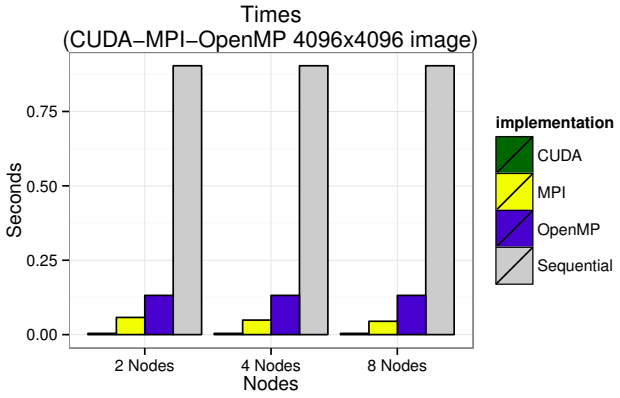

FIG. 5.18. *Cuda-MPI-OpenMP 4096×4096 image*

TABLE 5.4
Times for 4096×4096 images for the four different versions on one node (OpenMP uses 16 threads, MPI uses 16 processes)

Version	**Time (Seconds)**
Sequential	0.903651
OpenMP	0.131847
MPI	0.0772164
CUDA	0.0031519

$$PSNR(I,Z) = 20\log_{10}\frac{\max(I)}{\sqrt{MSE(I,Z)}} \tag{5.1}$$

where I is the original image and Z is the reconstructed image, $\max(I)$ is the maximum pixel value in I, and MSE is the mean square error between I and Z.

$$MSE(I,Z) = \frac{1}{M}\frac{1}{N}\sum_{i=0}^{M-1}\sum_{j=0}^{N-1}\|I(i,j) - Z(i,j)\|^2 \tag{5.2}$$

In image reconstruction the $PSNR$ values vary between $[30, 50]$. A $PSNR$ value of 50 or more indicates that the images are almost identical.

Robustness represents the resistance of a watermark against attacks, such as compression, scaling, cropping, rotation, smoothing, etc. The Normalized Correlation (NC) measures the correlation between the original watermark and the extracted watermark after attack. It is defined by

$$NC = \frac{\sum_{i=0}^{N-1}\sum_{j=0}^{N-1}[W(i,j)W'(i,j)]}{\sum_{i=0}^{N-1}\sum_{j=0}^{N-1}[W(i,j)]^2} \tag{5.3}$$

A value of NC equal to 1 indicates that the original and extracted watermark are exactly the same.

The quality of an $\alpha - \beta$ algorithm depends on the method for computing α and β. Our interest in this work has been focused on the computational aspects of $\alpha - \beta$ algorithms and we chose for our experiments the simplest representative, in which α and β are constant over all DCT blocks. Although we would not expect the quality for this representative to compare the most favorably with other members of the $\alpha - \beta$ family, it is

nevertheless of interest to determine values of PSNR and NC for this case. To this end, we used the software StirMark to determine the robustness for our algorithm using the values $\alpha = 0.9$ and $\beta = 0.2$.

Table 5.5 shows the results of applying the software StirMark [12] [13] to the watermarked image in Fig. 4.1 (c) of size 1024×1024 produced by our algorithm and subjected to the indicated attacks.

TABLE 5.5
The results of applying the software StirMark to the watermarked image

Image	**PSNR**	**NC**
Watermarked image	36.72	0.97
Compressed image	34.20	1.45
Smooth image	33.86	1.44
Rotated+scaling image	33.53	1.04
Randomly distorted image	30.21	1.09

The values of PSNR compare well with other watermarked image (e.g., [14]), in spite of the simplicity of our algorithm. On the other hand, we would expect values of NC closer to one for other more specialized $\alpha - \beta$ algorithms

5.7. Conclusions. We have shown how a commonly used family of watermarking algorithms in the frequency domain can be implemented in parallel and we compared the OpenMP, MPI, CUDA implementations of a simple representative of the $\alpha - \beta$ family of algorithms. We found that the CUDA implementation on just one GPU runs many times faster than the OpenMP version and the MPI version even when executed on 8 nodes. Thus the fastest implementation runs on just one node equipped with one GPU. For one node without a GPU, there are only slight differences between OpenMP and MPI, with OpenMP winning only for large images and MPI exhibiting a more regular speedup.

We observed that in spite of its simplicity, the chosen representative is resistant to attacks such as compression, distortion, rotation plus scaling, and smoothing.

The same method developed here for implementing this simple case could be applied to any $\alpha - \beta$ algorithm. Of course, the running times of all implementations will vary depending on the complexity of the calculations of the α_{ij} and β_{ij}.

REFERENCES

[1] I.J. Cox *et al* , *A secure robust watermarking for multimedia*, Proc. of First International Workshop on Information Hiding, Lecture Notes in Comp. Sc., Springer-Verlag, 1174 (1996), pp. 185–206.
[2] C. Garcia-Cano, B. Rabil, R. Sabourin, *A parallel watermarking application on a GPU*, Congreso Internacional de Investigacion en Nuevas Tecnologias Informáticas-CIINTI 2012, http://ciinti.info.memorias.
[3] Y. He, C. Ding, *MPI and OpenMP paradigms on cluster of SMP architectures: the vacancy tracking algorithm for multi-dimensional array transposition*, SCPE, 5 (2002), No. 2.
[4] M.S. Kankanhalli, *Adaptive visible watermarking of images*, Proc. ICMCS99, Florence, Italy, June (1999).
[5] Lenarczyk, Piotr and Piotrowski, Zbigniew, *Parallel blind digital image watermarking in spatial and frequency domains*, Springer US, Telecommunication Systems, 54 (2013), pp. 287-303, doi :10.1007/s11235-013-9734-x
[6] C. Lin, L. Zhao, and J. Yang, *A high performance image authentiction algorithm on GPU with CUDA*, I.J. Intelligent Systems and Applications, 2011, 2, pp. 55-59.
[7] S. Mohanty, *Digital watermarking: a tutorial review*, http://informatika.stei.itb.ac.id/ rinaldi.munir/Kriptografi/WMSurvey1999Mohanty.pdf.
[8] S. Mohanty, K. Ramakrishnan, M. Kankanhali, *A DCT domain visible watermarking technique for images*, International Conference on Multimedia Computing and Systems/International Conference on Multimedia and Expo-ICME(I Multimedia and Expo 2000, ICME(CMCS), pp. 1029–1032.
[9] A. Obukhov and A. Kharlamov, *Discrete cosine transform for 8×8 blocks with CUDA*, https://svn.inf.ufsc.br/luis.custodio/TCC-Dantas/CUDA/.../dct8x8.pdf
[10] C. Shieh, H. Huang, F. Wand, J. Pan, *Genetic Watermarking based on Transform-domain Techniques*, Pattern Recognition 37 (2004) pp. 555–565.
[11] Stampede User Guide, *https://portal.tacc.utexas.edu/user-guides/stampede*

[12] Petitcolas, FabienA.P. and Anderson, RossJ. and Kuhn, MarkusG. , *Attacks on Copyright Marking Systems*, Springer Berlin Heidelberg, Information Hiding, Lecture Notes in Computer Science, 1525 (1998), pp. 218-238
[13] Petitcolas, FabienA.P., *Watermarking schemes evaluation*, Signal Processing Magazine, IEEE, 17 Sep (2000), pp. 58-64
[14] Singh, A.K.; Dave, M.; Mohan, A., *A novel technique for digital image watermarking in frequency domain*, Parallel Distributed and Grid Computing (PDGC), 2012 2nd IEEE International Conference on, vol., no., pp.424,429, 6-8 Dec. 2012 doi: 10.1109/PDGC.2012.6449858
[15] B. Tao and B. Dickinson, *Adaptive watermarking in DCT domain*, Proc IEEE International Conf. on Acoustics, Speech and Signal Processing, ICASSP-97, 4 (1997), pp. 2985–2988.
[16] P. Vihari and M. Mishra, *Image authentication algorithm on GPU*, 2012 International Conference on Communication Systems and Network Technologies, pp. 874-878.

Edited by: Dana Petcu
Received: Apr 10, 2015
Accepted: Jun 24, 2015