International Journal of Pattern Recognition
and Artificial Intelligence
Vol. 24, No. 6 (2010) 897–915

DOI: 10.1142/S0218001410008202

World Scientific
www.worldscientific.com

A HIGH PERFORMANCE 3D EXACT EUCLIDEAN DISTANCE TRANSFORM ALGORITHM FOR DISTRIBUTED COMPUTING

JULIO CESAR TORELLI

Instituto de Ciências Matemáticas e de Computação — ICMC
Universidade de São Paulo — USP, São Carlos, Brazil
julio@icmc.usp.br

RICARDO FABBRI

Brown University
182 Hope St. Box D
Providence, RI 02912, USA
rfabbri@lems.brown.edu

GONZALO TRAVIESO* and ODEMIR MARTINEZ BRUNO†

Instituto de Física de São Carlos — IFSC
Universidade de São Paulo — USP, São Carlos, Brazil
**gonzalo@ifsc.usp.br*
†*bruno@ifsc.usp.br*

The Euclidean distance transform (EDT) is used in various methods in pattern recognition, computer vision, image analysis, physics, applied mathematics and robotics. Until now, several sequential EDT algorithms have been described in the literature, however they are time- and memory-consuming for images with large resolutions. Therefore, parallel implementations of the EDT are required specially for 3D images. This paper presents a parallel implementation based on domain decomposition of a well-known 3D Euclidean distance transform algorithm, and analyzes its performance on a cluster of workstations. The use of a data compression tool to reduce communication time is investigated and discussed. Among the obtained performance results, this work shows that data compression is an essential tool for clusters with low-bandwidth networks.

Keywords: Euclidean distance transform; parallel computing; distributed computing.

1. Introduction

The *distance transform* (DT) is the operation that converts a binary matrix, representing an image made of 1-points and 0-points, into another, the *distance map*, in which each element has a value corresponding to the distance from the corresponding image point to the nearest 0-point by given a distance function.[24,29] Various distance

functions can be used in the algorithm, but the Euclidean distance, which is a particular case of the Minkowski distances, is often adopted, mainly because of its rotational invariance property. The distance transform based on the Euclidean distance is called the *Euclidean distance transform* (EDT). It has been extensively used in numerical methods in pattern recognition, computer vision, image analysis, physics and applied mathematics. Some examples of EDT applications are:

(i) For use in general numerical methods: Voronoi tessellations[21,26,30]; Delaunay triangulation[30]; Bouligand–Minkowski fractal dimension[1,2]; Level Set Method[23,27,31]; multi-phase flow simulations.[4]
(ii) Applied to image analysis and computer vision: successive dilatation and erosion in mathematical morphology operations[25]; curve smoothing[33]; skeletonization[28]; Watershed segmentation[14,15,30]; path planning[29] in applications such as robotics.

When presenting Euclidean distance transform algorithms, most authors adopt the point of view of image processing, which is also the case in this paper. An image is a matrix whose elements are called pixels. The first published EDT algorithm was by Danielsson.[10] It generates the Euclidean distance map of a binary matrix/image by assuming that for any pixel there is another one in its neighborhood with the same nearest background pixel. Since this assumption does not always hold, as described in Ref. 9, it does not produce an *exact* distance map, in the sense that the results are not completely error free. Other non-exact EDT algorithms were reported later.[19,33] While non-exact EDTs are good enough for some applications, there are applications where an exact (error free) Euclidean distance transform is required. For example, the mathematical morphology dilatation operator can be implemented as a threshold of the Euclidean distance map. If a non-exact distance map is used, the errors in the distance map can lead to pixels missing from the dilated object.[8]

Various exact Euclidean distance transform algorithms have been described in the literature[7,8,11,12,20,22,26] however, unless the image is small, they are time- and memory-consuming. Therefore, applications involving large images, mainly 3D images, require parallel implementations of the EDT.

This paper presents a novel parallel algorithm based on a well-known 3D sequential EDT algorithm (by Saito and Toriwaki[26]) on a cluster of workstations. This algorithm was chosen because it is one of the fastest and simplest exact EDT algorithms.[12] This sequential algorithm produces the distance map of a 3D binary image by making three one-dimensional transformations, one for each coordinate direction.

We parallelized these transformations using domain decomposition partitioning.[13] Initially the planes of the image are distributed by a master processor among all the processors, including the master. Then each processor executes the first two transformations on the planes assigned to it. Next, each processor exchanges the results from these transformations with each other. This data exchange carries out a new data division of the image. Then each processor executes the third transformation on its

new image part. Finally, the results from the third transformation are returned to the master processor to be combined into the final distance map.

This paper starts by describing the sequential EDT we adopted. Next, our parallel EDT is presented in an abstract way, so as to make it more independent of hardware or implementation issues. This approach to algorithm parallelization supplies the basis enabling it to be implemented in different distributed memory MIMD machines. In the following section, implementation details on a cluster of workstations are presented and discussed. The use of a data compression tool to reduce communication time is also presented and discussed. Finally, the performance of the parallel EDT is quantified in terms of the speed-up obtained using different images.

2. A Short Review on Parallel EDT Algorithms

Some sequential EDT algorithms have been described in the literature. However, unless the matrix is small, they are time- and memory-consuming. Therefore, applications involving a large matrix, mainly a 3D matrix, require parallel implementations of the EDT. Although there are some reports of parallel EDT algorithms, most of them are developed to dedicated hardware or theoretical models, which limits the real use of the technique for applications using general computers and clusters.

Although Parallel exact EDT algorithms have been reported, most of them are designed for SIMD architectures[5,32] or specific architectures, such as systolic and reconfigurable architectures[6,17] or MIMD theoretical shared memory models.[16] SIMD machines are especially designed to perform parallel computations for vector or matrix types of data. In such machines, a program has a single instruction stream, and each instruction operates on multiple data elements simultaneously. This parallel programming model is simpler but also more limited. Another problem with SIMD machines is their relative high cost. Special architectures, such as systolic and reconfigurable architectures, are machines specifically built to carry out a single task. MIMD machines, on the other hand, are more general purpose and can be obtained at relatively low cost, for example, a cluster of workstations.

There are also papers that deal with EDT MIMD implementations, but most of them report just theoretical algorithms designed for abstract machines such as PRAM, that models shared memory architecture.[16] Considering MIMD on distributed systems (cluster of workstation), few works are reported in the literature. Perhaps this happens because of the nature of the EDT algorithm, since it has a strong demand for memory accesses, implying low parallel performance when clusters are used. We found only two works[3,18] that deal directly with cluster architecture. In Ref. 3, a non-optimal optimized algorithm is reported, which is several times slower than the one presented here; in Ref. 18, an optimized algorithm projected for EREW PRAM computer model is presented, but the reported experiment on a SP2 (MIMD

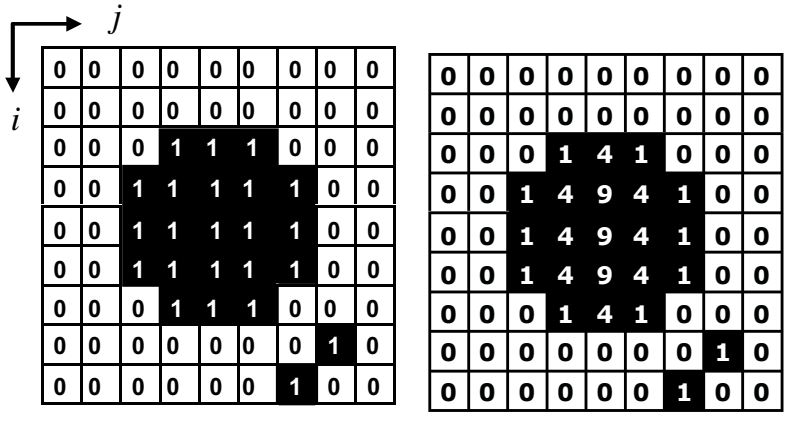
distributed memory machine) failed to achieve speed-up for more than three processors.

The estimation of the algorithm performance of the EDT is a hard task, since it is strongly dependent on input data. Fabbri *et al.*[12] proposed a framework to compare EDT performance. Most of the parallel EDT papers report new parallel algorithms and achieved speed-up with respect to the sequential version. Such comparison is not of much use, as optimized sequential EDT algorithms have huge variations of execution time. Considering this, the algorithm proposed in this paper is a parallel version of the sequential algorithm by Saito,[26] which is one of the fastest EDT algorithms.[12]

3. The Sequential EDT

The algorithm by Saito and Toriwaki[26] is one of the most famous EDT algorithms, both for its efficiency and reasonably easy implementation. It computes the exact Euclidean distance map of any n-dimensional (n-D) binary image by making n one-dimensional transformations, one for each coordinate direction. For 3D images, distance values are first computed along each row (first transformation). These values are then used to compute the distances into each plane (second transformation). Finally, by using the values from the planes, the distance from each voxel to the nearest background voxel in the three-dimensional space is computed (third transformation). Formally:

Transformation 1. Given an input binary image $F = \{f_{ijk}\}$ having R rows, C columns and P planes, $(1 \leq i \leq R,\ 1 \leq j \leq C,\ 1 \leq k \leq P)$, *transformation 1*



(a) (b)

Fig. 1. Example of *transformation 1*. (a) The first plane of a 3D binary image, and (b) this plane after *transformation 1*.

generates an image $G = \{g_{ijk}\}$ defined as follows:

$$g_{ijk} = \min\{(j-y)^2; f_{iyk} = 0, 1 \le y \le C\} \tag{1}$$

This corresponds to computing the distance from each voxel (i, j, k) to the nearest 0-voxel in the same row as (i, j, k). The algorithm for *transformation 1* consists of one scan from left to right (forward scan) plus one scan from right to left (backward scan) on each row of the input image F. It is summarized in the following (assume every g_{ijk} is initialized as ∞, which is chosen to be larger than the maximum possible distance for the input image):

```
for k = 1 to P do
    for i = 1 to R do {
        for j = 1 to C do {      // forward scan
            if f_ijk ≠ 0 then    g_ijk = (√g_i(j−1)k + 1)^2
            else                 g_ijk = 0
        }
        for j = C − 1 to 1 do    // backward scan
            g_ijk = min{(√g_i(j+1)k + 1)^2, g_ijk}
    }
```

Transformation 2. From G, *transformation 2* generates an image $H = \{h_{ijk}\}$ defined as follows (Fig. 2):

$$h_{ijk} = \min\{g_{xjk} + (i-x)^2; 1 \le x \le R\} \tag{2}$$

Actually, to calculate h_{ijk}, it is usually not necessary to check the value of all voxels in the column including (i, j, k), as suggested in Fig. 2. Instead, the algorithm acts by scanning each column of G once from top to bottom (forward scan) and once in the opposite direction (backward scan).

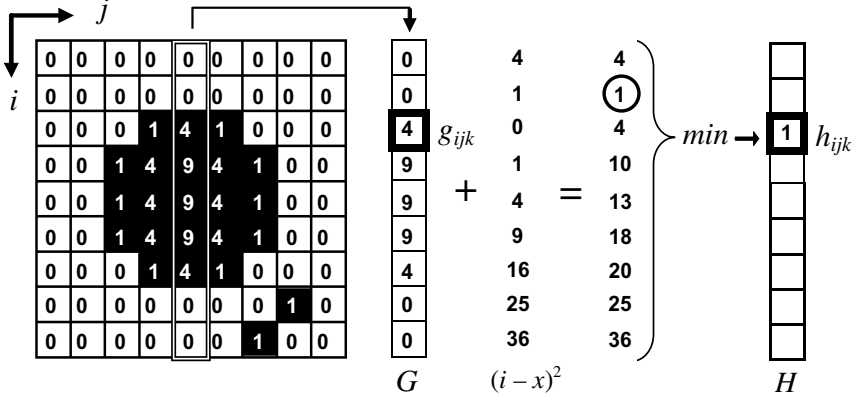


Fig. 2. Example of *transformation 2* process. In order to calculate h_{ijk}, add to the value of each voxel (x, j, k), i.e. the voxels in the column including (i, j, k), the distance from that voxel to (i, j, k), i.e. $(i-x)^2$. The value h_{ijk} is then defined as the minimum of the additions.

In the forward scan, the following procedure is performed:

for each row $i = 1$ to R do

  if $g_{ijk} > g_{(i-1)jk} + 1$ then

    // Attempt to propagate the distance of the voxel at $(i-1)jk$
    // to further rows $i+n$, *with* n *from* 0 *to an upper bound.*
    for $n = 0$ to $(g_{ijk} - g_{(i-1)jk} - 1)/2$ do

      if $g_{(i-1)jk} + (n+1)^2 < g_{(i+n)jk}$

        // The distance at $(i+n)jk$ *is greater than the propagated value*
        // from $(i-1)jk$, *thus propagate the distance to* $(i+n)jk$
        // taking into account the vertical distance $(n+1)$
        $h_{(i+n)jk} = g_{(i-1)jk} + (n+1)^2$

      else

        break *// go to the next* i

  else $h_{ijk} = g_{ijk}$

Remember that after *transformation 1* each voxel has a value equal to the distance from this voxel to the nearest 0-voxel in its row. The tests above are then carried out to check if the voxel $(i+n, j, k)$, for $0 \leq n \leq (g_{ijk} - g_{(i-1)jk} - 1)/2$, is nearer to the 0-voxel in the row $i-1$ than to the 0-voxel in its row. If this is true, its value is redefined to indicate its distance in relation to that 0-voxel (i.e. $g_{(i-1)jk} + (n+1)^2$). The maximum value of n is found by noting that $g_{(i-1)jk} + (n+1)^2 > g_{ijk} + n^2$ for any $n > (g_{ijk} - g_{(i-1)jk} - 1)/2$). Note that the second transformation will take more or less time to execute depending on the values of the first transformation, which in turn depends on the contents of the image.

After the forward scan the backward scan proceeds similarly. At the end of the backward scan, each voxel has a value equal to its distance to the nearest 0-voxel into its plane.

Transformation 3. From H, *transformation 3* generates an image $S = \{h_{ijk}\}$ (actually, the images F, G, H and S can be assigned to the same address in computer memory) given by the following equation:

$$s_{ijk} = \min\{h_{ijz} + (k-z)^2; 1 \leq z \leq P\} \tag{3}$$

The algorithm for *transformation 3* is the same for *transformation 2*. Only, suffix k is changed instead of the suffix i, i.e. both the forward scan and the backward scan proceed in the k-axis direction.

Note that Saito and Toriwaki's EDT uses the squared Euclidean distance, instead of the Euclidean distance. The square root operation must be used in the post processing to obtain the Euclidean distance map from the squared Euclidean distance map.

4. The Parallel EDT

This section presents the strategy adopted for the parallelization of the EDT algorithm. Such parallelization strategy is presented here in a more abstract way, so as to make it more independent of hardware or implementation issues. The aim of this approach to algorithm presentation is to supply the basis enabling it to be implemented in different distributed memory MIMD machines. Distributed memory MIMD machines are those having two or more independent processors that do not share a memory and communicate by passing messages over a network. The implementation of such parallelization strategy on a cluster of workstations is presented in the next sections.

Figure 3 graphically presents the overall organization of the parallelization strategy. For the sake of a clearer presentation, it is divided into stages. The first stage consists of loading and dividing the 3D image. On a parallel computer with p processors, one master and $p-1$ slaves, the master divides the image into p chunks, each of which contains a number of consecutive planes. The master then sends one chunk to each processor, including itself. Note that load balancing could be performed at this stage by including more or less planes into each chunk, proportional to processor speed. At the second stage, each processor executes *transformation 1* and then *transformation 2* on the planes at its image chunk; at this stage, each process executes independently in parallel, and no communication is needed among processors. The third processing stage involves executing *transformation 3*, but before such a transformation is to be performed some data exchange is necessary. As it was described in Sec. 3, during *transformation 3* the image is scanned in the k-axis direction. However, in the first stage of our parallel algorithm, the planes are distributed among the processors, which do not share a memory. Therefore, before *transformation 3* is to be performed, we need to join the planes. Afterwards, we need to divide the image into parts to be concurrently processed in the third

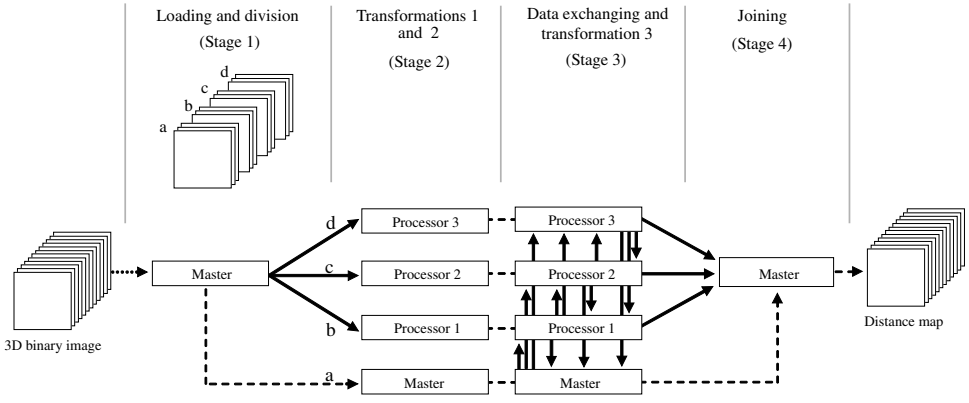


Fig. 3. Parallelization strategy (dashed rows indicate communications through memory).

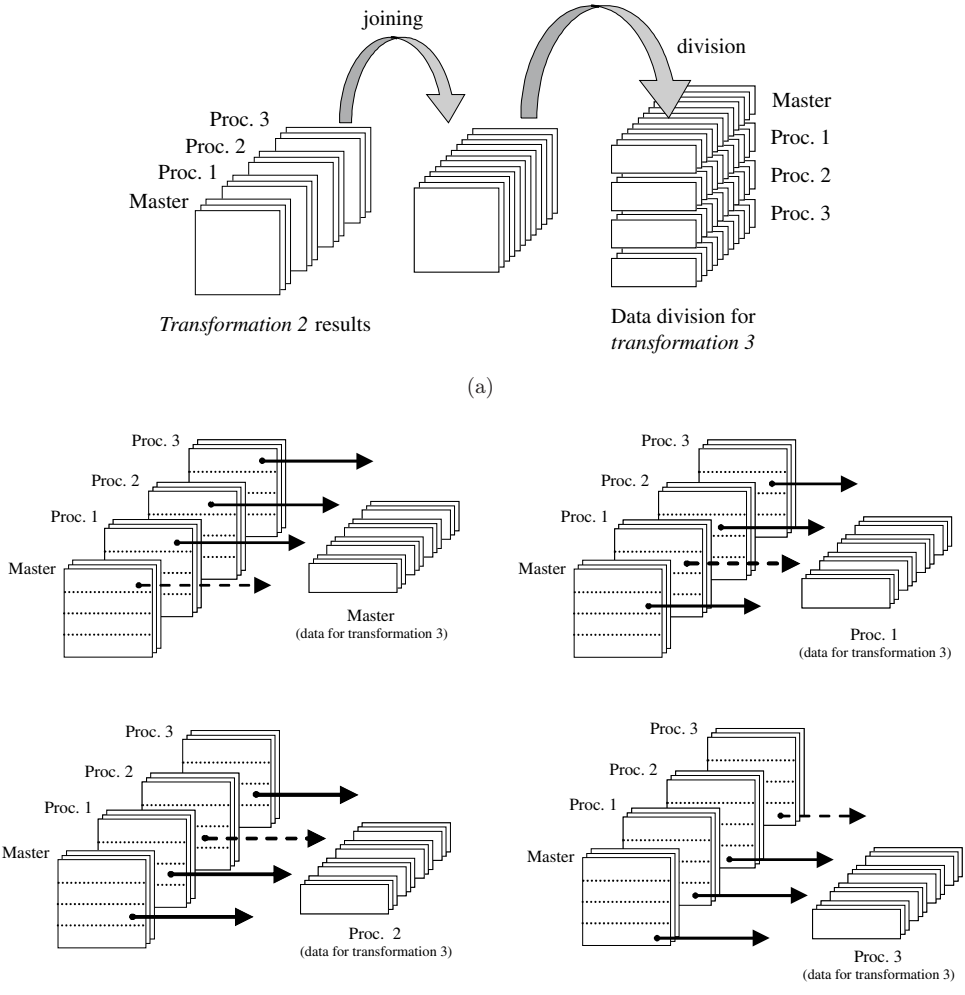


Fig. 4. Joining and division. (a) Logical joining and division; (b) real joining and division.

transformation. We choose to divide the image into p parts, each of which contain a number of consecutive rows, as in Fig. 4(a). There are at least two real ways to perform such operations: (i) each processor sends the planes with the results from the second transformation to the master processor; the master then joins all the planes, divides the image into those new p parts, and sends one part to each processor; or (ii) each processor directly sends the rows that the processor uses in *transformation 3* to each other (in such a case, it is necessary for each processor to previously know the rows to send to each other). We choose this second way, because the amount of data which is necessary to transmit through the network is smaller. Furthermore, we can achieve some parallelism, since different processors can communicate at the same

time. This data exchange scheme is illustrated in Fig. 4(b). Note that each processor sends one message to each other processor. Although the messages can be sent in any order, sending the messages in such a way that simultaneously there is no more than one processor sending a message to the same destination processor, can improve the parallelism in communication. Each processor, after receiving the messages with the rows from all the others, executes *transformation 3* on these rows (image part). Then each processor sends such data to the master processor. In the fourth processing stage, the master joins the parts from all the processors to make the final Euclidean distance map.

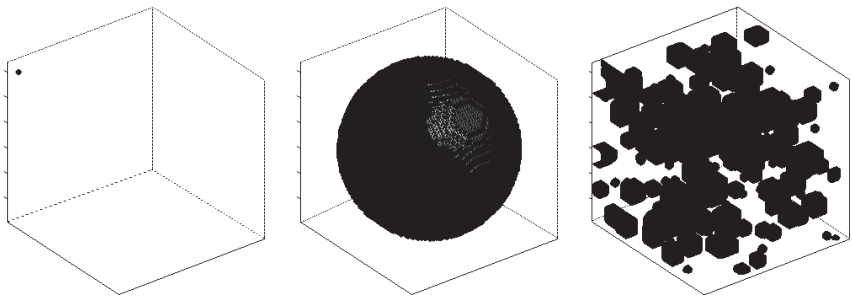
5. Performance Issues

Saito and Toriwaki's EDT, as almost all exact EDT algorithms, is image dependent, i.e. even for the same image size its execution time varies depending on both the number and the shape of the objects in the image. Therefore, the choice of the images on which the implementations are executed can significantly affect the results. In order to validate our implementations we used different images of different sizes. The content of such images are described below, together with an explanation of their properties useful for this task:

A single background voxel in a corner (Fig. 5(a)). In such cases, the EDT produces the largest possible distance for a given image size. Moreover, the number of voxels which receive a nonzero distance value is also the biggest.

An inscribed sphere (Fig. 5(b)). The Voronoi tiles of such an image are very irregular. Some EDT algorithms perform poorly in such cases.

Random cubes (Fig. 5(c)). They imitate a real image. The cubes represent objects with straight boundaries − the midpoint of each cube is randomly chosen; the image has about 50% of background voxels.



(a) (b) (c)

Fig. 5. Test images: (a) single voxel; (b) inscribed sphere; and (c) random cubes.

6. Implementation Issues

This section reports on the implementation of the parallel EDT on a cluster of workstations. Clusters of workstations are becoming increasingly popular as a parallel hardware platform, mainly because of their good cost-benefit relationship. In 3D digital image processing, a cluster can be used not only to speed-up the applications by parallel processing, but also to overcome memory constraints. Since single computers usually have low memory resources, using the memories of the computers of a cluster may overcome this problem.

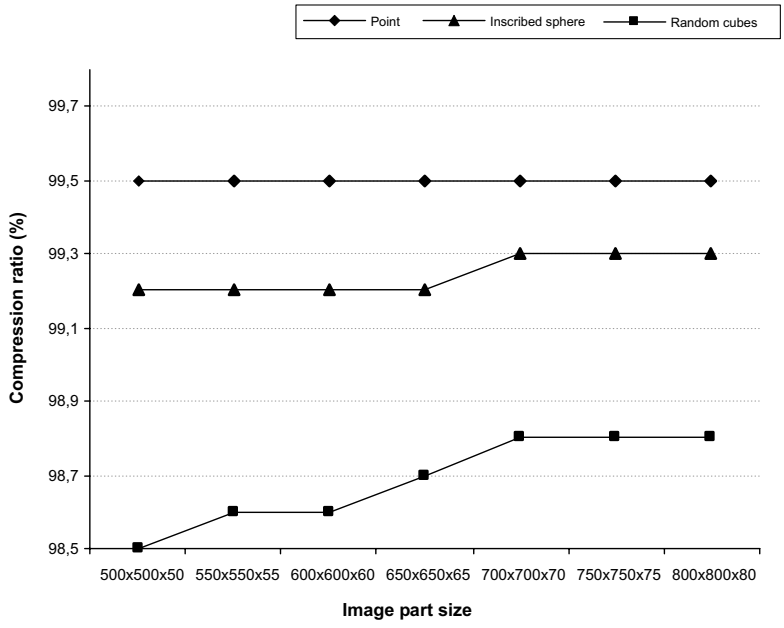
In this work we used a ten-node cluster. Each node had a 2.8 GHz Pentium 4, 1.5 GB of RAM and a Fast-Ethernet network interface connected to a 100 Mbps switch. We call "master" (or node 0) the node where the images are stored. We call "slaves" (or node 1, node 2,..., node 9) the other nodes. Although the framework presented in this section could be implemented with any message-passing tool, we decided for MPICH (http://www-unix.mcs.anl.gov/mpi/mpich/), a freely available implementation of Message Passing Interface (MPI) standard (http://www.mpi-forum.org). MPI is the *de facto* standard for message-passing parallel programing, and MPICH is the most commonly used implementation of MPI.

The master node is responsible for loading and dividing the 3D image. Such operations are summarized in the following code:

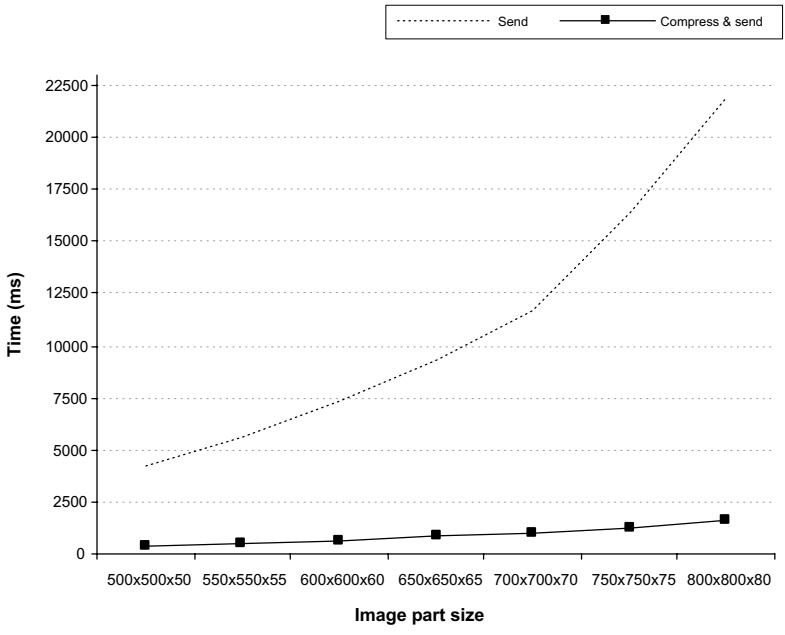
```
for i = 1 to p − 1 do {   // p is the number of nodes
    load   img_part i;
    compress   img_part i;
    send   img_part i to node i;
    free   img_part i;
}
```

In the above code we see that the master node loads, compresses and then sends to each node i one image part, which is represented by *img_part* i — such an image part contains a number of planes of the image. Data compression is used to reduce communication time.[3] It is performed with zlib (http://www.zlib.org), a lossless data compression tool. Since the voxels of a binary image can just be 0- or 1-valued, the image shows a good degree of redundancy favoring compression. Figure 6(a) shows the compression rates obtained while compressing the parts of the three test images. Note that high compression rates were obtained in all cases. The compression operation was, in turn, very fast. Figure 6(b) shows the average time necessary to transmit one image part with and without compression for the random cubes test image (when using compression, the times shown include compression, transmission and decompression times). Observe that, even for such an image, which was the one with the poorest compression rates, compressing and sending one image part was always significantly faster than sending that image part without compression.

The compressed image parts are sent through the *Ready mode MPI send function*. Such a function requires a matching received to be posted before send is called, but



(a)



(b)

Fig. 6. Graphs describing the compression operation for the parts of the test images. (a) Compressing ratio for the three test images; (b) sending time with and without compression for the random cube test image.

communication overhead is reduced because shorter protocols are used internally by MPI when it is known that a receipt has already been posted. Observe that the *img_part i* is loaded from the disk to the master node's memory right at the moment it is sent to node i (and it is removed from the memory after sending). Such a strategy avoids the physical memory of the master node to be completely full, when the operating system of such a node starts swapping data to and from the disk, increasing the program's execution time.

By possessing its image part, each node executes *transformation 1* and then *transformation 2* on it. Afterwards, it sends the image rows that the node uses in the third transformation to each other − such a data exchange scheme was explained in Sec. 4. Before sending, these rows are also compressed. The compressed rows are sent through the *nonblocking standard MPI send function*. Such a function returns immediately, with no information about whether the completion criteria have been satisfied. The advantage is that the nodes are free to compress the rows of other nodes while some communication proceeds "in the background."

After receiving the rows from all the other nodes, each node executes *transformation 3* on these rows. Then it compresses and sends them to the master node, which joins the rows from all the processors to build the final Euclidean distance map. In the distance transformation, each background voxel produces a series of equidistant sets that propagate along the matrix. Such a propagation proceeds until an obstacle (another propagating front) is found. Thus, the number and distribution of the background voxels become related to the compression ratio of the distance map, i.e. the compression ratio of the image parts with the results from the third transformation. Figure 8(a) shows the compression rates obtained while compressing the parts of three test images. Note that the poorest compression ratio occurred with the single background voxel image. Since there is only one background voxel in such an image, the number of voxels having the same distance value is smaller than in the other two images. However, compressing and sending the parts of the single background voxel image was still faster than sending those image parts without compression, as shown in Fig. 8(b).

7. Load Balance

Load imbalance occurs when a part of a parallel program takes more time on one processor than on the others and the processors have to wait for one another. On the parallel EDT, we noted that the image distribution is a source of load imbalance whether the image is divided into parts with the same number of planes. Figure 7 shows the activity plot of the execution of the parallel EDT while processing the $800 \times 800 \times 800$ inscribed sphere test image. Note that, although the nodes have the same processing power, bubbles appear in the execution as a consequence of the delay in loading, compressing and sending of each image part. Furthermore, we can see that the time each node spends processing the second transformation (and the third transformation) is different from the others, because the content of each image part is

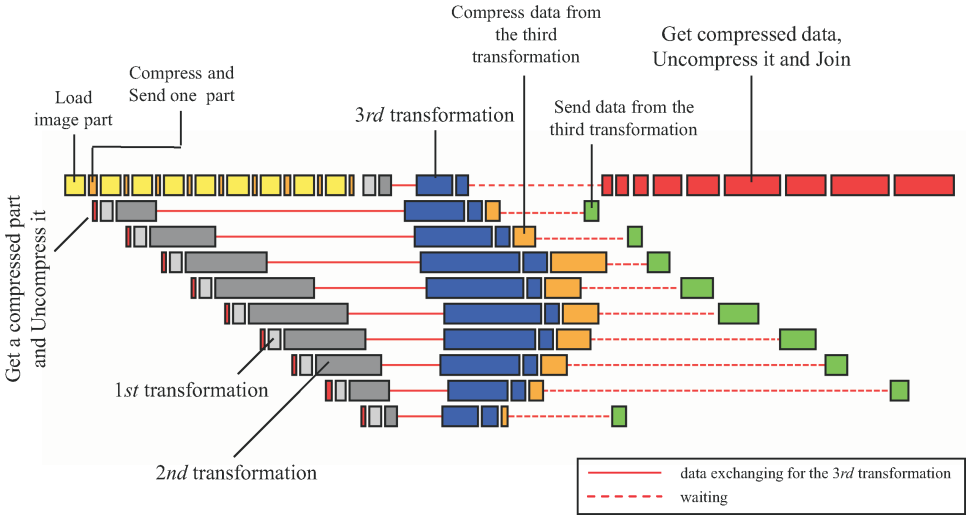
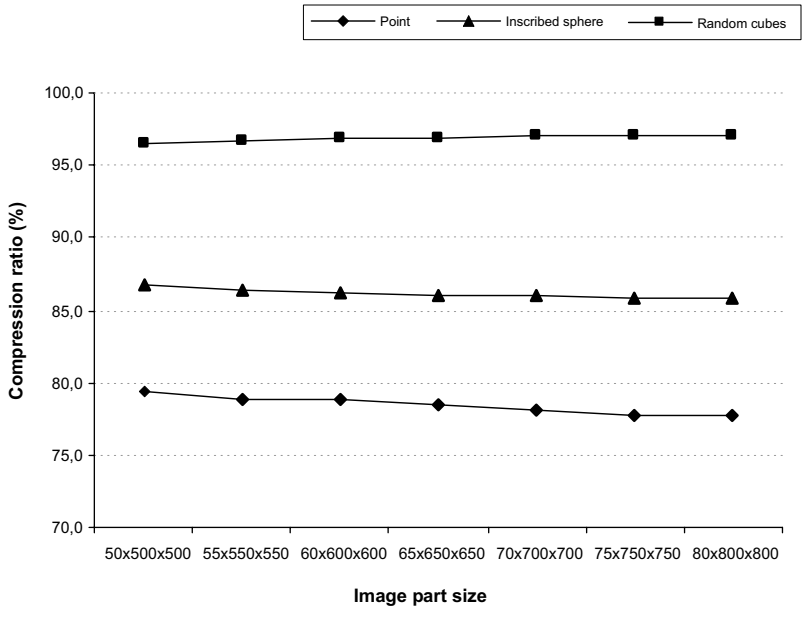


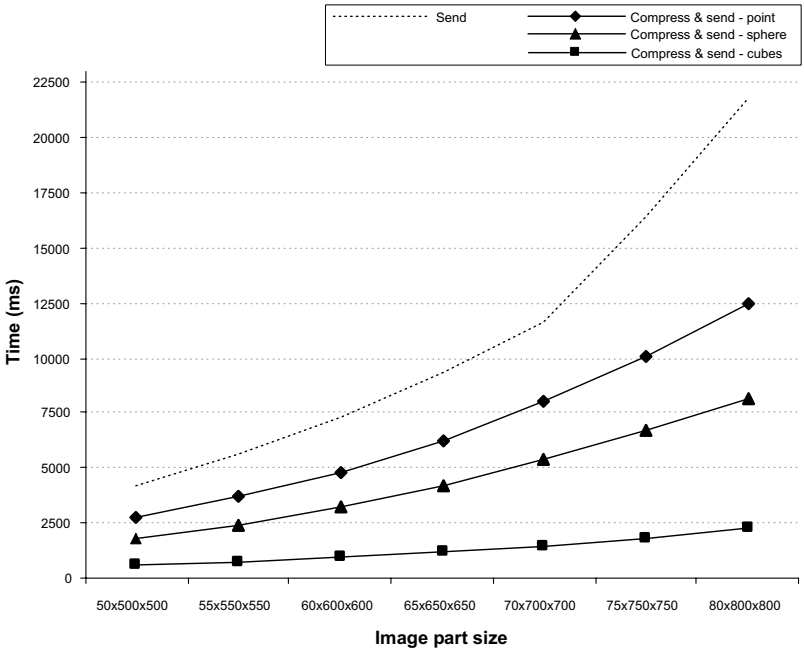
Fig. 7. Execution time diagram.

different from the others. That is why a perfect load balance scheme cannot possibly be reached. In order to reach a perfect load balance situation, an analysis of the content of the image would be necessary, but such a type of analysis would be more expensive than the delay caused by the load imbalance.



(a)

Fig. 8. Graphs describing the compression for the data of the third transformation. (a) Compressing ratio for the test images; (b) sending time with and without compression.



(b)

Fig. 8. (*Continued*)

8. Experimental Results

Speed-up is a metric which shows the increase in speed obtained from a parallel implementation of a problem. An indication of efficient use of the hardware resources is often considered. We calculate the speed-up by dividing the time spent with the sequential EDT execution by the time spent with the parallel EDT execution. Table 1 shows the average execution time of the sequential EDT while running on one node of the cluster.

Table 1. Execution time of the sequential EDT, in seconds, for the test images. The number in brackets represents the standard deviation around the average.

Image Size	Single Point	Inscribed Sphere	Random Cubes
$500 \times 500 \times 500$	36 (0)	53 (0)	33 (0)
$550 \times 550 \times 550$	48 (1)	74 (0)	44 (0)
$600 \times 600 \times 600$	64 (0)	100 (0)	58 (1)
$650 \times 650 \times 650$	80 (0)	133 (1)	73 (0)
$700 \times 700 \times 700$	103 (3)	172 (1)	91 (0)
$750 \times 750 \times 750$	535 (28)	598 (42)	486 (13)
$800 \times 800 \times 800$	974 (20)	1044 (32)	863 (41)

Note that the execution time of the sequential EDT drastically increases when images with $750 \times 750 \times 750$ or more voxels are used. Since a single node in the cluster has 1.5 GB of RAM, an image with $750 \times 750 \times 750$ voxels, which is close to 1.57 GB in size, cannot be completely loaded into its physical memory. In such a case, the operating system starts swapping parts of the image to and from the disk drastically increasing the execution time of the program.

In the currently considered parallel hardware (a cluster with a 100 Mbps network) the speed-up is mostly adversely affected by the overheads in communication. Thus, this was the motivation for using data compression, which allowed good speed-ups for large images. In order to characterize the performance of the parallel EDT, it was executed considering the three test images. Figure 9 presents the speed-ups obtained for such an experiment — note that ten processing elements were used.

As far as scalability is concerned, the cluster can virtually incorporate any number of nodes. Although the number of messages grows together with the number of nodes of the cluster, the amount of data transmitted through the network will always be the same for a given image size. Figure 10 presents the speed-ups obtained while processing the $650 \times 650 \times 650$ voxel test images using two, four, six, eight and ten processing elements. Note that the execution time was smaller when more nodes were used.

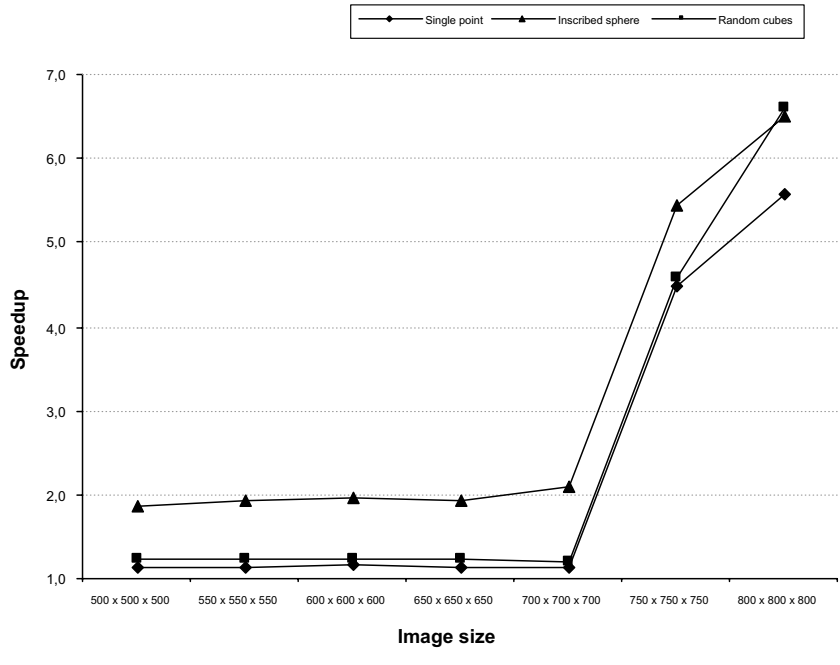


Fig. 9. Speed-up results when ten processing elements were used.

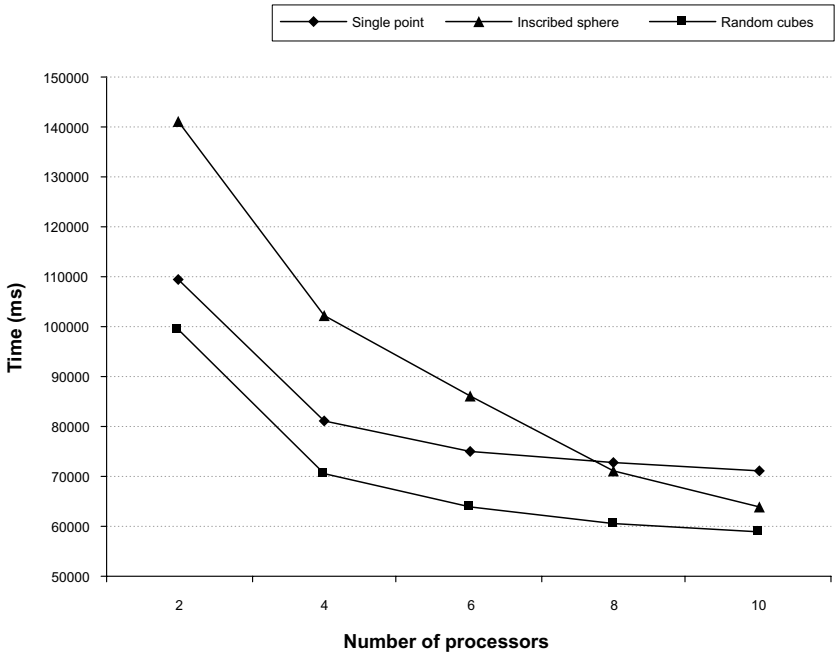


Fig. 10. Speed-up results using two, four, six, eight and ten processing elements.

9. Concluding Remarks

This article has reported a parallel implementation of a 3D exact Euclidean distance transform algorithm. The considered hardware was a cluster of workstations. The considered message-passing tool was MPICH. The parallelization strategy consists of (i) dividing the planes of the 3D binary image among the nodes of the cluster, which execute two first transformations on the planes in their part; (ii) joining these planes; (iii) redividing the 3D image among the nodes, which then execute a third transformation on their new part; and (iv) joining these parts to build the final Euclidean distance map. Data compression was used in a much successful attempt to reduce communication time.

Acknowledgments

The authors are grateful to CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico) from Brazil for financial support (Grant #132003/2004-0 and Grant #303746/2004-1). They are also grateful to Dr. Luciano da Fontoura Costa from the Cybernetic Vision Research Group of the Physics Institute (IFSC) at the University of São Paulo for letting them use his cluster.

References

1. A. R. Backes, D. Casanova and O. M. Bruno, Plant leaf identification based on volumetric fractal dimension, *Int. J. Patt. Recogn. Artif. Intell.* **23** (2009) 1145−1160.

2. O. M. Bruno, R. de O. Plotze, M. Falvo and M. de Castro, Fractal dimension applied to plant identification, *Inform. Sci.* **178** (2008) 2722−2733.
3. O. M. Bruno and L. F. Costa, A parallel implementation of exact Euclidean distance transform based on exact dilations, *Microprocessors and Microsystems* **28**(3) (2004) 107−113.
4. H. D. Ceniceros and A. M. Roma, A multi-phase flow method with a fast, geometry-based fluid indicator, *J. Comput. Phys.* **205**(2) (2005) 391−400.
5. L. Chen and H. Y. H. Chuang, An efficient algorithm for complete Euclidean distance transform on mesh-connected simd, *Parallel Computing* **21**(5) (1995) 841−852.
6. L. Chen, Y. Pan, Y. Chen and X. Xu, Efficient parallel algorithms for Euclidean distance transform, *The Computing J.* **47**(6) (2004) 694−700.
7. O. Cuisenaire and B. Macq, Fast and exact signed Euclidean distance transformation with linear complexity, *Proc. IEEE Int. Conf. Acoustics, Speech and Signal Processing* (1999), pp. 3293−3296.
8. O. Cuisenaire and B. Macq, Fast Euclidean distance transformations by propagation using multiple neighborhoods, *Comput. Vis. Imag. Underst.* **76**(2) (1999) 163−172.
9. O. Cuisenaire, *Distance Transformations: Fast Algorithms and Applications to Medical Image Processing*, PhD thesis, Université Catholique de Louvain (1999).
10. P. E. Danielsson, Euclidean distance mapping, *Comput. Graph. Imag. Process.* **14** (1980) 227−248.
11. H. Eggers, Two fast Euclidean distance transformations in z2 based on sufficient propagation, *Comput. Vis. Imag. Underst.* **69**(1) (1998) 106−116.
12. R. Fabbri, L. Da F. Costa, J. C. Torelli and O. M. Bruno, 2d Euclidean distance transform algorithms: A comparative survey, *ACM Comput. Surv.* **40**(1) (2008) 1−44.
13. I. Foster, *Designing and Building Parallel Programs* (Addison-Wesley, Reading, MA, 1995).
14. M. Frucci, Oversegmentation reduction by flooding regions and digging watershed lines, *Int. J. Patt. Recogn. Artif. Intell.* **20**(1) (2006) 15−38.
15. W.-F. Kuo, C.-Y. Lin and Y.-N. Sun, Region similarity relationship between watershed and penalized fuzzy hopfield neural network algorithms for brain segmentation, *Int. J. Patt. Recogn. Artif. Intell.* **22**(7) (2008) 1403−1425.
16. Y. H. Lee, S. J. Horng, T. W. Kao, F. S. Jaung, Y. J. Chen and H. R. Tsai, Parallel computation of exact Euclidean distance transform, *Parallel Comput.* **22** (1996) 311−325.
17. Y. Lee, S. Horng, T. Kao and Y. Chen, Parallel computation of the Euclidean distance transform on the mesh of trees and the hypercube computer, *Comput. Vis. Imag. Underst.* **68**(1) (1997) 109−119.
18. Y. Lee, S. Horng and J. Seltzer, Parallel computation of the euclidean distance transform on a three-dimensional image array, *Parall. Distrib. Syst.* **14**(3) (2003) 203−212.
19. R. A. Lotufo, A. A. Falcão and F. A. Zampirolli, Fast Euclidean distance transform using a graph-search algorithm, *Proc. Brazilian Symp. Computer Graphics and Image Processing* (2000), pp. 269−275.
20. R. A. Lotufo and F. A. Zampirolli, Fast multi-dimensional parallel Euclidean distance transform based on mathematical morphology, *Proc. Brazilian Symp. Computer Graphics and Image Processing* (2001), pp. 100−105.
21. Y. Lu, C. Xiao and C. L. Tan, Constructing area voronoi diagram based on direct calculation of the freeman code of expanded contours, *Int. J. Patt. Recogn. Artif. Intell.* **21**(5) (2007) 947−960.
22. C. R. Maurer, R. Qi and V. Raghavan, A linear time algorithm for computing exact Euclidean distance transforms of binary images in arbitrary dimensions, *IEEE Trans. Patt. Anal. Mach. Intell.* **25**(2) (2003) 265−270.

23. S. Osher and J. A. Sethian, Fronts propagating with curvature-dependent speed, *J. Comput. Phys.* **79**(1) (1988) 12−49.
24. A. Rosenfeld and J. L. Pfaltz, Sequential operations in digital picture processing, *J. Assoc. Comp. Mach.* **13** (1966) 471−494.
25. J. C. Russ, *The Image Processing Handbook* (CRC Press, Boca Raton, FL, 2002).
26. T. Saito and J. I. Toriwaki, New algorithms for Euclidean distance transformation of an n-dimensional digitized picture with applications, *Patt. Recogn.* **27**(11) (1994) 1551−1565.
27. J. A. Sethian, *Level Set Methods: Evolving Interfaces in Geometry, Fluid Mechanics, Computer Vision, and Materials Science* (Cambridge University Press, Cambridge, 1996).
28. F. Y. Shih and C. C. Pu, A skeletonization algorithm by maxima tracking on Euclidean distance transform, *Patt. Recogn.* **28**(3) (1995) 331−341.
29. F. Y. Shih and Y. T. Wu, Three-dimensional Euclidean distance transformation and its application to shortest path planning, *Patt. Recogn.* **37**(1) (2004) 79−92.
30. L. Vicent and P. Soille, Watersheds in digital spaces: An efficient algorithm based on immersion simulations, *IEEE Trans. Patt. Anal. Mach. Intell.* **13**(6) (1991) 583−598.
31. R. Xia, W. Liu, Y. Wang and X. Wu, Fast initialization of level set method and an improvement to chan-vese model, *Int. Conf. Computer and Information Technology* (2004), pp. 18−23.
32. H. Yamada, Complete Euclidean distance operation by parallel operation, *Proc. Int. Conf. Patt. Recogn.* (1984), pp. 69−71.
33. Q. Z. Ye, The signed Euclidean distance transform and its applications, *Proc. Int. Conf. Patt. Recogn.* (1988), pp. 495−499.

Julio Cesar Torelli is a Software Architect working for COSAN – Brazil. He received a master's degree in computer sciences from the University of Sao Paulo in 2005.

Gonzalo Travieso is an electronic engineer from Escola de Engenharia de São Carlos, M.Sc. and obtained his Ph.D. in applied physics from the Instituto de Física de São Carlos, both from the Universidade de São Paulo.

His main research interests are parallel and distributed processing and complex systems.

Ricardo Fabbri began experimenting with C++ programming in middle and high school, inspired by his interests in video games. In 1999, he joined the ICMC institute from the University of São Paulo at São Carlos for his undergraduate degree in computer science. Since his freshman year, he has been an active computer vision researcher at the Cybernetic Vision Research Group from the Institute of Physics at São Carlos (IFSC), having published in prestigious conferences and an international journal as an undergraduate student. He graduated in 2004 amongst the top three students in his class, and obtained a master's degree in computer science also from ICMC in the same year. Since then he has been at the Laboratory for Engineering Man-Machine Systems (LEMS) at Brown University for his Ph.D. studies, where he has worked with 3D scene reconstruction and camera calibration from multiple views using general curve structures. In 2008 he interned at Google, Inc., where he worked on rapid 3D scanning systems for books. After the completion of his Ph.D. studies he will also be joining Google, Inc. He has published several papers in leading international journals.

His research interests include computer vision, computational geometry, differential geometry, multiview stereo, camera auto-calibration, structure from motion, and related open source software.

Odemir Martinez Bruno is an associate professor at the Institute of Physics of S. Carlos at the University of S. Paulo in Brazil. He received his B.Sc. in computer science in 1992, from the Piracicaba Engineering College (Brazil), his M.Sc. in applied physics (1995) and his Ph.D. in computational physics (2000) at the University of S. Paulo (Brazil).

His fields of interest include computer vision, image analysis, computational physics, pattern recognition and bioinformatics. He is an author of many papers (journal and proceedings) and several book chapters. He is co-author of two books (*Optical and Physiology of Vision: a multidisciplinary approach* (Portuguese only) *and Internet programming with PHP* (Portuguese only)) and an inventor of five patents.