Parallelization strategy for elementary morphological operators on graphs: distance-based algorithms and implementation on multicore shared-memory architecture

Imane Youkana, Jean Cousty, Rachida Saouli, Mohamed Akil

Laboratoire d'informatique Gaspard-Monge (preprint)

Parallelization strategy for elementary morphological operators on graphs: distance-based algorithms and implementation on multicore shared-memory architecture

Imane Youkana · Jean Cousty · Rachida Saouli · Mohamed Akil

May 2017

Abstract This article focuses on the (unweighted) graph-based mathematical morphology operators presented in *[J. Cousty* et al*, "Morphological filtering on graphs", CVIU 2013]*. These operators depend on a size parameter that specifies the number of iterations of elementary dilations/erosions. Thus, the associated running times increase with the size parameter, the algorithms running in $O(\lambda.n)$ time, where n is the size of the underlying graph and λ is the size parameter. In this article, we present distance maps that allow us to recover (by thresholding) all considered dilations and erosions. The algorithms based on distance maps allow the operators to be computed with a single linear $O(n)$ time iteration, without any dependence to the size parameter. Then, we investigate a parallelization strategy to compute these distance maps. The idea is to build iteratively the successive level-sets of the distance maps, each level set being traversed in parallel. Under some reasonable assumptions about the graph and sets to be dilated, our parallel algorithm runs in $O(n/p + K \log_2 p)$ where n, p, and K are the size of the graph, the number of available processors, and the number of distinct level-sets of the distance map, respectively. Then, implementations of the proposed algorithm on a shared-memory multicore architecture are described and assessed on datasets of 45 images and 6 textured 3-dimensional meshes, showing a reduction of the processing time by a factor up to 55 over the previously available implementations on a 8 core architecture.

Keywords Graph-based mathematical morphology · distance maps · parallel algorithm · multicores/multithreaded architectures.

1 Introduction

Mathematical morphology provides a set of filtering and segmenting tools that are very useful in applications to image analysis. From a historical point of view, the first field of applications of mathematical morphology was digital images, *i.e.*, sets of pixels aligned on a 2D or 3D grid and equipped with binary, grayscale, or vectorial values [29]. However, the theoretical basis of mathematical morphology relies on the abstract algebraic structure of a complete lattice [15] allowing us to consider the processing of a very broad class of data with mathematical morphology operators.

Imane Youkana, Jean Cousty, Rachida Saouli, and Mohamed Akil
Université Paris-Est, Laboratoire d'Informatique Gaspard-Monge, A3SI, ESIEE Paris, CNRS
E-mail: {imane.youkan, jean.cousty, rachida.saouli, mohamed.akil}@esiee.fr,

Imane Youkana and Rachida Saouli
Université de Biskra, Département d'Informatique, Biskra, Algérie.

On the other hand, there is a growing interest for considering digital objects not only composed of points but also composed of elements lying between them and carrying structural information about how the points are glued together (see *e.g.*, [24], [7,8,12,11], and [2] for recent mathematical morphology operators on graphs, on cubical/simplicial complexes, and on hypergraphs respectively). The simplest of these representations are the graphs. The domain of an image is considered as a graph (which can be planar or not) whose vertex set is made of the pixels and whose edge set is given by an adjacency relation on these pixels. Note that this adjacency relation can be either spatially invariant or spatially variant leading to operators that are either spatially invariant or spatially variant (see [18], for early applications of spatially variant mathematical morphology). Graphs are also useful to process other kinds of discrete structures defined for instance on 3-dimensional meshes. In this context, it becomes relevant to consider morphological transformations acting on the subsets of vertices, the subsets of edges and the subgraphs of a graphs and not only those acting on the set of all subsets of pixels.

Mathematical morphology on graphs was pioneered by Vincent [33] who proposes operators relying on a dilation (and its adjunct erosion) that act on the vertices of a graph. More recently, [9] investigates four basic dilatations and erosions that map a set of vertices to a set of edges and a set of edges to a set of vertices. It was shown in [9] that these operators can be combined in order to obtain operators acting on the subsets of edges, on the subsets of vertices and on the subgraphs of a given graph. In particular, interesting openings and closings (and then the associated alternate sequential filters) are obtained by iteration of the basic operators. The number of iterations constitutes a filtering parameter related to the size of the features to be preserved or removed. Therefore, based on the straightforward definition, the time-complexity of the associated algorithms increases with the size parameter. More precisely, for a parameter value of λ the algorithms run in $O(\lambda.n)$ time, where n is the size of the underlying graph.

In this article, our main contributions are threefold: i) we introduce new distance maps that lead to original characterizations of the operators of [9]; ii) based on these distance maps, we propose the first $O(n)$ time sequential algorithms for all erosions/dilations of [9], avoiding the dependence to the size parameter λ; and iii) we propose a parallelization strategy leading to fast computation, in particular, for multicore shared memory architectures.

After presenting background notions about morphology and graphs in Section 2, we study in Section 3 some distance maps that lead to characterizations of the dilations and erosions presented in [9]. In particular, we introduce edge-vertex and vertex-edge distance maps. Given a set of edges (resp. vertices), the edge-vertex (resp. vertex-edge) distance map provides for each vertex (resp. edge) a distance to the closest edge (resp. vertex) of the input set. In order to compute these distance maps, Section 4 presents adaptations of linear-time algorithms for distance maps in unweighted graphs. These algorithms derive from breadth first search (see, *e.g.*, [6] for an introduction to breadth first search). The dilations and erosions of [9] can then be obtained by thresholding these distance maps, leading to linear-time algorithms for the operators of [9]. Section 5 introduces a parallel algorithm to compute the proposed distance maps, hence the morphological operators of [9]. Parallel and/or separable algorithms for morphological operators and distance maps on images have been widely studied [31, 28,5,20,32,30,4,25]. Based on the regular structure of the space, such computations use a static partitioning of the image into rows, columns or blocks processed in parallel. In order to handle the non-regular structure of a graph, our parallelization strategy is based on a dynamic partitioning of the space which depends on the input set and which is iteratively computed during the execution. The time complexity of our parallel algorithm is analyzed. Under some reasonable assumptions about the graph and set under consideration, our algorithm runs in $O(n/p + K \log_2 p)$ time, where n, p, and K are the size of the underlying graph, the number of available processors and the number of distinct level sets of the distance map, respectively. Finally, the three last sections of the article, namely Sections 6, 7, and 8, present an implementation of the proposed parallel algorithm on a shared memory multicore architecture and assess the proposed implementation on three datasets containing images commonly used in the literature as well as textured meshes. The assessment shows, in particular, that the proposed parallel implementation runs up to 55 times faster than the implementations of the operators of [9] that were available before the present article.

This article extends an article [34] published in a conference. In particular, it contains the proof of the properties presented in [34]. The three last sections dedicated to the implementation of the proposed parallel algorithm and its assessment are also completely original.

2 Background notions for morphology on graphs

In this section, we recall background notions for mathematical morphology on graphs. After providing basic definition for graphs, we present in Section 2.1 four operators studied in [10,9] and whose grayscale extension was first introduced in [23]. They constitute a set of basic building blocks for morphology on graphs. Section 2.2 presents operators obtained by composition and iterations of these building blocks. These iterated operators are known to be efficient for processing binary images or more generally graphs [9]. Finally, in Section 2.3, we remind a notion of a distance map in a (unweighted) graph and a link, established in [33], with two of the operators of Section 2.2. In the remainder of the article, this link is extended to all operators of Section 2.2 and used to propose efficient sequential and parallel algorithms for the morphological operators of [9].

2.1 Elementary morphological operators on graphs

A *(undirected) graph* is a pair $X = (X^\bullet, X^\times)$ where $X^\bullet$ is a set and $X^\times$ is composed of unordered pairs of distinct elements in $X^\bullet$, *i.e.*, $X^\times$ is a subset of $\{\{x, y\} \subseteq X^\bullet \mid x \neq y\}$. Each element of $X^\bullet$ is called a *vertex or a point (of X)*, and each element of $X^\times$ is called an *edge (of X)*.

Important notation. Hereafter, the workspace is a graph $\mathbb{G} = (\mathbb{G}^\bullet, \mathbb{G}^\times)$ and we consider the sets $\mathcal{G}^\bullet$, $\mathcal{G}^\times$ and $\mathcal{G}$ of respectively all subsets of $\mathbb{G}^\bullet$, all subsets of $\mathbb{G}^\times$ and all subgraphs of $\mathbb{G}$.

Mathematical morphology filtering on graphs, as introduced in [9], relies on four basic operators which are used to derive a set of edges from a set of vertices and a set of vertices from a set of edges.

Definition 1 *We define the operators $\delta^\bullet$ and $\epsilon^\bullet$ from $\mathcal{G}^\times$ to $\mathcal{G}^\bullet$ and the operators $\epsilon^\times$, and $\delta^\times$ from $\mathcal{G}^\bullet$ to $\mathcal{G}^\times$ as follows:*

$$\delta^\bullet(X^\times) = \{x \in \mathbb{G}^\bullet \mid \exists \{x, y\} \in X^\times\}, \text{ for any } X^\times \subseteq \mathbb{G}^\times; \tag{1}$$

$$\epsilon^\bullet(X^\times) = \{x \in \mathbb{G}^\bullet \mid \forall \{x, y\} \in \mathbb{G}^\times, \{x, y\} \in X^\times\}, \text{ for any } X^\times \subseteq \mathbb{G}^\times; \tag{2}$$

$$\epsilon^\times(X^\bullet) = \{\{x, y\} \in \mathbb{G}^\times \mid x \in X^\bullet \text{ and } y \in X^\bullet\}, \text{ for any } X^\bullet \subseteq \mathbb{G}^\bullet; \text{ and} \tag{3}$$

$$\delta^\times(X^\bullet) = \{\{x, y\} \in \mathbb{G}^\times \mid x \in X^\bullet \text{ or } y \in X^\bullet\}, \text{ for any } X^\bullet \subseteq \mathbb{G}^\bullet. \tag{4}$$

In other words, the operator $\delta^\bullet$ maps to any edge set $X^\times$ the set of all vertices that belong to an edge in $X^\times$. The operator $\epsilon^\bullet$ maps to any edge set $X^\times$ the set of vertices that are "completely covered" by edges in $X^\times$, *i.e.* the vertices that do not belong to any edge of the complement $\mathbb{G}^\times \setminus X^\times$ of $X^\times$. The operator $\epsilon^\times$ maps to any vertex set $X^\bullet$ the set of all edges whose two extremities are in $X^\bullet$. The operator $\delta^\times$ maps to any vertex set $X^\bullet$ the set of all edges that have at least one extremity in $X^\bullet$.

The operators $\epsilon^\times$, $\delta^\times$, $\epsilon^\bullet$ and $\delta^\bullet$ are illustrated in Figures 1 (first three columns), 2 (first two columns), 3 (first two columns), and 4 (first two columns), respectively.

Many mathematical morphology operators rely on the algebraic structure of a lattice [15,14,26]. In particular, any operator that commutes with the union (resp. intersection) is called a *dilation* (resp. an *erosion*). It is established in [9] that the operators $\delta^\bullet$ and $\delta^\times$ are indeed morphological dilations and that the operators $\epsilon^\bullet$ and $\epsilon^\times$ are erosions since they commute with union and intersection respectively, *i.e.* we have $\delta^\bullet(X^\times \cup Y^\times) = \delta^\bullet(X^\times) \cup \delta^\bullet(Y^\times)$, $\epsilon^\bullet(X^\times \cap Y^\times) = \epsilon^\bullet(X^\times) \cap \epsilon^\bullet(Y^\times)$, $\delta^\times(X^\bullet \cup Y^\bullet) = \delta^\times(X^\bullet) \cup \delta^\times(Y^\bullet)$, and $\epsilon^\times(X^\bullet \cap Y^\bullet) = \epsilon^\times(X^\bullet) \cap \epsilon^\times(Y^\bullet)$, for any $X^\bullet, Y^\bullet \subseteq \mathbb{G}^\bullet$ and $X^\times, Y^\times \subseteq \mathbb{G}^\times$. Therefore, in the following, the operators $\delta^\times$ and

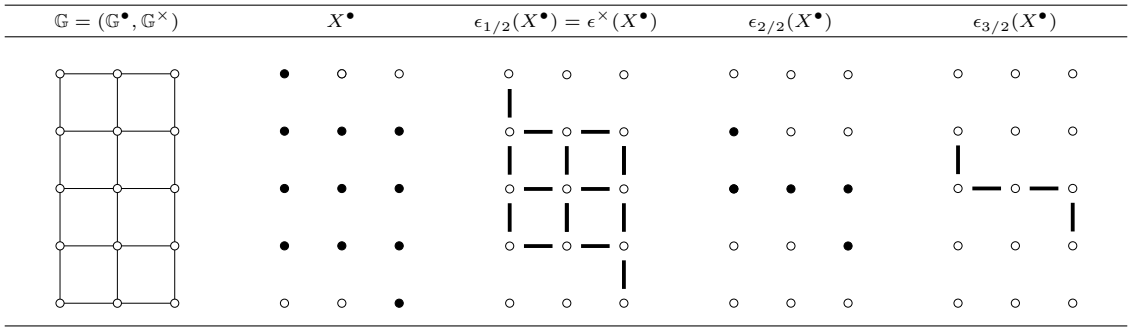


Fig. 1 Illustration of the operator $\epsilon_{\lambda/2}$ with $\lambda \in \{1, 2, 3\}$. The considered sets are depicted by black dots or line segments.

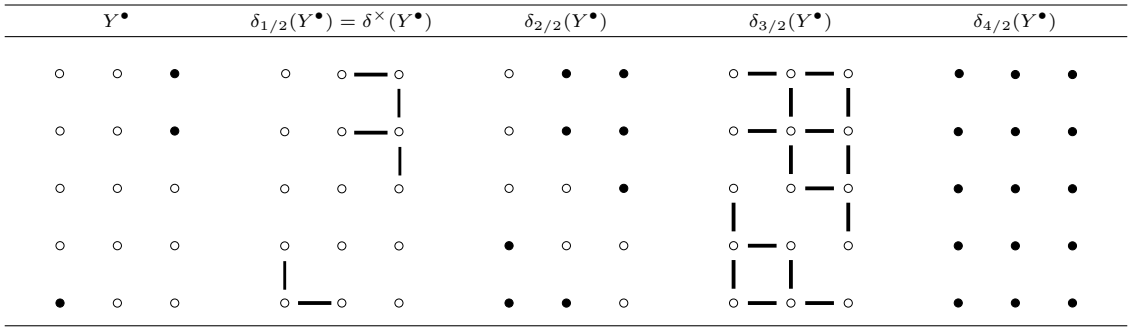


Fig. 2 Illustration of the operator $\delta_{\lambda/2}$ with $\lambda \in \{1, 2, 3, 4\}$, the workspace being the graph $\mathbb{G}$ depicted in Figure 1. The considered sets are depicted by black dots or line segments.

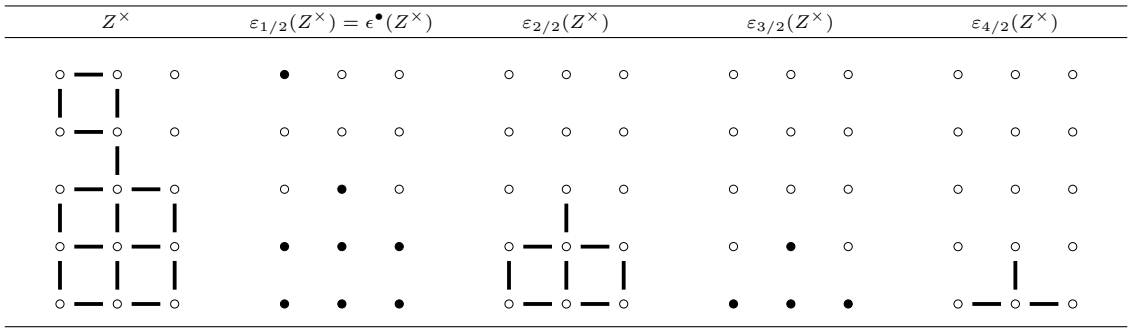


Fig. 3 Illustration of the operator $\varepsilon_{\lambda/2}$ with $\lambda \in \{1, 2, 3, 4\}$, the workspace being the graph $\mathbb{G}$ depicted in Figure 1. The considered sets are depicted by black dots or line segments.

$\delta^\bullet$ (resp. $\epsilon^\times$ and $\epsilon^\bullet$) are referred to as the *vertex-edge dilation* and the *edge-vertex dilation* (resp. the *vertex-edge erosion* and the *edge-vertex erosion*).

Observe that the operators introduced in Definition 1 are given for arbitrary (unweighted, undirected) graphs. In particular, they can be applied on spatially invariant and spatially variant pixel adjacency graphs as well as any other kind of graphs such as, *e.g.*, the graphs derived from three dimensional meshes in Section 7.

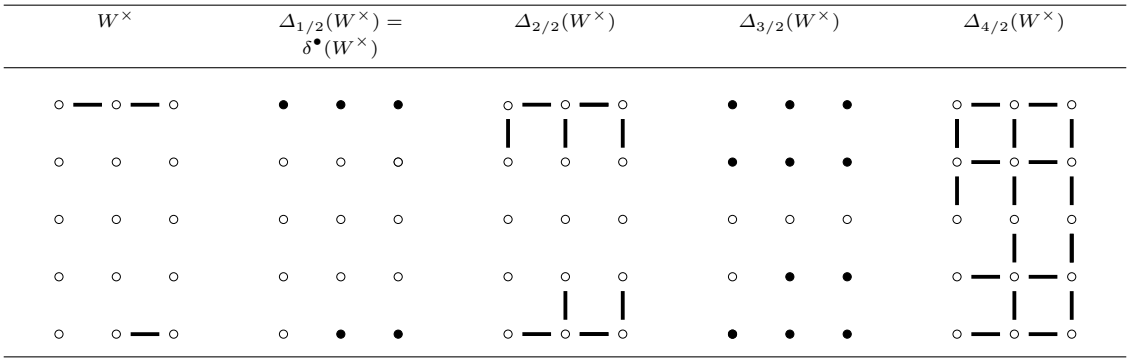


Fig. 4 Illustration of the operator $\Delta_{\lambda/2}$ with $\lambda \in \{1, 2, 3, 4\}$, the workspace being the graph $\mathbb{G}$ depicted in Figure 1. The considered sets are depicted by black dots or line segments.

The erosions and dilations introduced in Definition 1 allow for defining operators such as opening, closing and alternate sequential filters that are useful in applications [9, 22]. To this end, one needs to consider compositions and iterations of these building blocks.

2.2 Iterated dilations and erosions on graphs

Let α be an operator acting on $\mathcal{G}^\bullet$ or on $\mathcal{G}^\times$ and let i be a non negative integer. The operator α^i is defined by the identity when $i = 0$ and by $\alpha \circ \alpha^{i-1}$ otherwise.

In other words, when i is greater than 1, the result of the operator α^i applied to a set X can be obtained by applying i iterations of α from X.

The elementary operators presented in Definition 1 map the elements of $\mathcal{G}^\bullet$ (*i.e.*, subsets of vertices) to those of $\mathcal{G}^\times$ (*i.e.*, subsets of edges) or the elements of $\mathcal{G}^\times$ to those of $\mathcal{G}^\bullet$. Thus, since the input and output sets of these operators are distinct, they cannot be directly iterated. However, any composition of an operator acting from $\mathcal{G}^\bullet$ to $\mathcal{G}^\times$ (resp. from $\mathcal{G}^\times$ to $\mathcal{G}^\bullet$) with an operator from $\mathcal{G}^\times$ to $\mathcal{G}^\bullet$ (resp.from $\mathcal{G}^\bullet$ to $\mathcal{G}^\times$) leads to an operator on $\mathcal{G}^\bullet$ (resp. on $\mathcal{G}^\times$). Then, such composition can be iterated and eventually followed again by an operator from $\mathcal{G}^\bullet$ to $\mathcal{G}^\times$ (resp. from $\mathcal{G}^\times$ to $\mathcal{G}^\bullet$). Therefore, to define iterated operators on graphs, we can distinguish two cases depending whether a final composition with an operator from $\mathcal{G}^\bullet$ to $\mathcal{G}^\times$ (resp. from $\mathcal{G}^\times$ to $\mathcal{G}^\bullet$) is considered or not.

Definition 2 (Iterated dilations/erosions) *Let λ be a nonnegative integer.*

Case 1 (even values of λ). *If λ is even, we define the operators $\delta_{\lambda/2}$ and $\epsilon_{\lambda/2}$ on $\mathcal{G}^\bullet$ and the operators $\Delta_{\lambda/2}$ and $\varepsilon_{\lambda/2}$ on $\mathcal{G}^\times$ by:*

$$\delta_{\lambda/2}(X^\bullet) = (\delta^\bullet \circ \delta^\times)^{\lambda/2}(X^\bullet), \text{ for any } X^\bullet \subseteq \mathbb{G}^\bullet; \tag{5}$$

$$\epsilon_{\lambda/2}(X^\bullet) = (\epsilon^\bullet \circ \epsilon^\times)^{\lambda/2}(X^\bullet), \text{ for any } X^\bullet \subseteq \mathbb{G}^\bullet; \tag{6}$$

$$\Delta_{\lambda/2}(X^\times) = (\delta^\times \circ \delta^\bullet)^{\lambda/2}(X^\times), \text{ for any } X^\times \subseteq \mathbb{G}^\times; \text{ and} \tag{7}$$

$$\varepsilon_{\lambda/2}(X^\times) = (\epsilon^\times \circ \epsilon^\bullet)^{\lambda/2}(X^\times), \text{ for any } X^\times \subseteq \mathbb{G}^\times. \tag{8}$$

Case 2 (odd values of λ). *If λ is odd, we define the operators $\delta_{\lambda/2}$ and $\epsilon_{\lambda/2}$ from $\mathcal{G}^\bullet$ to $\mathcal{G}^\times$ and the operators $\Delta_{\lambda/2}$ and $\varepsilon_{\lambda/2}$ from $\mathcal{G}^\times$ to $\mathcal{G}^\bullet$ by:*

$$\delta_{\lambda/2}(X^\bullet) = \delta^\times \circ \delta_{(\lambda-1)/2}(X^\bullet), \text{ for any } X^\bullet \subseteq \mathbb{G}^\bullet; \tag{9}$$

$$\epsilon_{\lambda/2}(X^\bullet) = \epsilon^\times \circ \epsilon_{(\lambda-1)/2}(X^\bullet), \text{ for any } X^\bullet \subseteq \mathbb{G}^\bullet; \tag{10}$$

$$\Delta_{\lambda/2}(X^\times) = \delta^\bullet \circ \Delta_{(\lambda-1)/2}(X^\times), \text{ for any } X^\times \subseteq \mathbb{G}^\times; \text{ and} \tag{11}$$

$$\varepsilon_{\lambda/2}(X^\times) = \epsilon^\bullet \circ \varepsilon_{(\lambda-1)/2}(X^\times), \text{ for any } X^\times \subseteq \mathbb{G}^\times. \tag{12}$$

The operators $\epsilon_{\lambda/2}$, $\delta_{\lambda/2}$, $\varepsilon_{\lambda/2}$, and $\Delta_{\lambda/2}$ are illustrated for various (even and odd) values of λ in Figures 1, 2, 3, and 4, respectively. In particular, it can be observed that, when λ is even, the operators $\epsilon_{\lambda/2}$ and $\delta_{\lambda/2}$ map a set vertices to a set of vertices and that the operators $\varepsilon_{\lambda/2}$ and $\Delta_{\lambda/2}$ map a set of edges to a set of edges. On the other hand, when λ is odd, the operators $\epsilon_{\lambda/2}$ and $\delta_{\lambda/2}$ map a set of edges to a set of vertices and the operators $\varepsilon_{\lambda/2}$ and $\Delta_{\lambda/2}$ map a set of vertices to a set of edges.

It is known in morphology that interesting filters, called openings and closings, are obtained by composition of corresponding (adjunct) dilations and erosions [15,14,26]. Such openings and closings are useful in applications in order to filter out noise and to regularize the contours of images. The size of the structures which are preserved/suppressed with openings and closings depends on the number of iterations of the basic dilations and erosions. Larger structures are removed/preserved when higher numbers of iterations are considered. Therefore, the number of iterations involved in a morphological operator is often referred to as its *size parameter*. The openings and closings resulting from the compositions of the dilations and erosions of Definition 2 act either on sets of vertices or on sets of edges as recalled in Table 1 (see [9] for an extensive study). Furthermore, the simultaneous application of these filters (for a same size parameter) on the vertices and on the edges of a subgraph of $\mathbb{G}$ leads to a subgraph of $\mathbb{G}$, hence morphological filtering on subgraphs.

Domain	**Parity of λ**	**Openings**	**Closings**
Vertex sets	even	$\delta_{\lambda/2} \circ \epsilon_{\lambda/2}$	$\epsilon_{\lambda/2} \circ \delta_{\lambda/2}$
-	odd	$\Delta_{\lambda/2} \circ \epsilon_{\lambda/2}$	$\varepsilon_{\lambda/2} \circ \delta_{\lambda/2}$
Edge sets	even	$\Delta_{\lambda/2} \circ \varepsilon_{\lambda/2}$	$\varepsilon_{\lambda/2} \circ \Delta_{\lambda/2}$
-	odd	$\delta_{\lambda/2} \circ \varepsilon_{\lambda/2}$	$\epsilon_{\lambda/2} \circ \Delta_{\lambda/2}$

Table 1 Summary of the openings and closings resulting from the dilations and erosions of Definition 2.

By further composing openings and closings with increasing size parameter, alternate sequential filters are obtained. They generally outperform the results of elementary openings and closings in applications to image regularization. Describing in details the filters that can be obtained thanks to the operator of Definition 2 was done in [9] and is beyond the scope of this article.

As seen above, when the value of λ is even, the input and output sets of the operators $\delta_{\lambda/2}$ and $\epsilon_{\lambda/2}$ are subsets of vertices of the graph $\mathbb{G}$. From a mathematical morphology point of view, these operators (*i.e.* $\{\delta_{\lambda/2}, \epsilon_{\lambda/2} \mid \lambda \text{ is even }\}$) were first studied in [33], where a link with the notion of a distance map was established. This link is reminded in Section 2.3 before being extended to all operators of Definition 2 in Section 3.

2.3 Vertex-vertex distance maps on graphs

Let x and y be two vertices in $\mathbb{G}^\bullet$. A *(vertex-vertex) path from x to y* is a sequence $(x_0, u_0, \ldots, x_{\ell-1}, u_{\ell-1}, x_\ell)$ such that $x_0 = x$, $x_\ell = y$, and, for any i in $\{0, \ldots, \ell - 1\}$, and we have $u_i = \{x_i, x_{i+1}\}$ where u_i is an edge

of $\mathbb{G}$. The *length of a path* $(x_0, u_0, \ldots, x_{\ell-1}, u_{\ell-1}, x_\ell)$ is the number of its elements minus one, *i.e.*, the integer value 2ℓ. A *shortest path from x to y* is a path of minimal length from x to y. We denote by $L(x, y)$ the length of a shortest path from x to y.

Observe that the length of any vertex-vertex path is even, which is not usual in standard graph textbooks. Indeed, the length of a path is often considered as the number of edges along the path, whereas, in this article, both the numbers of vertices and of edges along the path are considered. It can be remarked that the two notions of length are equivalent up to a factor 2. The choice of multiplying by 2 the usual length of paths was driven by Property 4 (presented hereafter) that directly links the length of paths to the operators of Section 2.2.

In order to establish such link, an elementary use of path is considered, namely the computation of distance maps [27]: given a set $X^\bullet$ of vertices, for any vertex of $\mathbb{G}$, one can compute the graph-distance (*i.e.*, the length of a shortest path) to the closest vertex in $X^\bullet$.

Definition 3 (vertex-vertex distance map) *Let $X^\bullet$ be a subset of $\mathbb{G}$. The* (vertex-vertex) distance map to $X^\bullet$ *is the map $D^{(\bullet;\bullet)}_{X^\bullet}$ from $\mathbb{G}^\bullet$ to the set of integers such that:*

$$D^{(\bullet;\bullet)}_{X^\bullet}(x) = \min\{L(x, y) \mid y \in X^\bullet\}, \text{ for any } x \in X^\bullet. \tag{13}$$

An illustration of vertex-vertex distance map is provided in Figure 5 (leftmost subfigure).

The computation of shortest paths is a well studied topic in the field of graph algorithms. In particular, when the considered graph is unweighted, the lengths of shortest paths to a given vertex can be obtained with a breadth-first search algorithm. As precisely studied in Section 4, such algorithm runs in linear $O(|\mathbb{G}^\bullet| + |\mathbb{G}^\times|)$ time complexity.

When λ is an even integer, distance maps can be used to compute the dilation or erosion of size parameter $\lambda/2$ of a subset of vertices. Indeed, as stated by the following property due to L. Vincent [33], one can compute a distance map and deduce the result of the dilation or erosion by considering thresholds of that distance map at value λ.

Property 4 (from [33]) *Let $X^\bullet$ be a subset of $\mathbb{G}$. Then, the following relations hold true:*

$$\delta_{\lambda/2}(X^\bullet) = \{x \in \mathbb{G}^\bullet \mid D^{(\bullet;\bullet)}_{X^\bullet}(x) \leq \lambda\}, \text{ for any } X^\bullet \in \mathcal{G}^\bullet; \tag{14}$$

$$\epsilon_{\lambda/2}(X^\bullet) = \{x \in \mathbb{G}^\bullet \mid D^{(\bullet;\bullet)}_{\overline{X^\bullet}}(x) > \lambda\}, \text{ for any } X^\bullet \in \mathcal{G}^\bullet, \tag{15}$$

where $\overline{X^\bullet}$ is the complement of $X^\bullet$ in $\mathbb{G}^\bullet$, that is $\overline{X^\bullet} = \mathbb{G}^\bullet \setminus X^\bullet$.

In other words, the set $\delta_{\lambda/2}(X^\bullet)$ contains any vertex with a value not greater than λ for the distance map $D^{(\bullet;\bullet)}_{X^\bullet}$. The set $\epsilon_{\lambda/2}(X^\bullet)$ contains any vertex with a value greater than λ for the distance map $D^{(\bullet;\bullet)}_{\overline{X^\bullet}}$ to the complementary set $\overline{X^\bullet}$ of $X^\bullet$.

When λ is even, a naive approach based on Definition 2 to compute the dilation $\delta_{\lambda/2}(X^\bullet)$ from the set $X^\bullet$ consists of performing $\lambda/2$ iterations of the operator $(\delta^\bullet \circ \delta^\times)$. A linear-time algorithm for $\delta^\bullet$ and for $\delta^\times$ can be easily designed. Thus, this naive algorithm to compute $\delta_{\lambda/2}(X^\bullet)$ runs in $O(\lambda(|\mathbb{G}^\bullet| + |\mathbb{G}^\times|))$ time complexity. On the other hand, based on Property 4, a $O(|\mathbb{G}^\bullet| + |\mathbb{G}^\times|)$ time-complexity algorithm can be obtained for performing the same task. To this end, one needs to compute a distance map and to threshold it at λ. Using the algorithm presented in Section 4, such distance map can be obtained in linear-time with respect to the size of the graph $\mathbb{G}$ and the simple thresholding operation can be performed in linear time with respect to the number of vertices of the graph. Hence, the iterated dilation of size $\lambda/2$ can be computed with a single iteration instead of iterating elementary dilation λ times. This allows to avoid the dependence to the size parameter λ in the algorithm time-complexity.

In this section, the link, established in [33], between distance map and two of the height operators presented in Definition 2 was reminded. This link leads to a sequential linear-time algorithm for computing the results of the corresponding dilation and erosion. In the next section, a similar approach is developed to obtain linear-time algorithms for the six remaining operators of Definition 2.

3 Distance maps for morphological operators on graphs

Following the morphological approach based on distance maps recalled in Section 2.3, we introduce in this section three original notions of distance maps on graphs called edge-edge, edge-vertex, and vertex-edge distance maps. Given a set of edges, the edge-edge (resp. edge-vertex) distance map provides for each edge (resp. each vertex) of the graph a distance to the closest edge in the input set. Given a set of vertices, the vertex-edge distance map provides for each edge a distance to the closest vertex in the input set. Then, we show that all dilations and erosions on graphs presented in [9] can be characterized with distance maps. These characterizations lead to linear-time sequential algorithms for computing all these dilations and erosions.

Let us start by presenting notions of shortest edge-edge, vertex-edge and edge-vertex paths that are necessary for defining the edge-edge, vertex-edge and edge-vertex distance maps.

Let u and v be two edges in $\mathbb{G}^\times$ and let x be a vertex in $\mathbb{G}^\bullet$.

- A *(edge-edge) path from u to v* is a sequence $(u_0, x_0, \ldots, u_{\ell-1}, x_{\ell-1}, u_\ell)$ such that $u_0 = u$, $u_\ell = v$, $x_0 \in u_0$, $x_{\ell-1} \in u_\ell$, and $(x_0, u_1, \ldots, u_{\ell-1}, x_{\ell-1})$ is a vertex-vertex path from x_0 to $x_{\ell-1}$.
- A *(vertex-edge) path from x to u* is a sequence $(x_0, u_0, \ldots, x_\ell, u_\ell)$ such that $x_0 = x$, $u_\ell = u$, $x_\ell \in u_\ell$, and $(x_0, u_0, \ldots, x_\ell)$ is a vertex-vertex path from x_0 to x_ℓ.
- A *(edge-vertex) path from u to x* is a sequence $(u_0, x_0, \ldots, u_\ell, x_\ell)$ such that $(x_\ell, u_\ell, \ldots, x_0, u_0)$ is a vertex-edge path from x to u .

The *length* of a path is the number of its elements minus one. Hence, the length of an edge-edge path $(u_0, x_1, \ldots, u_{\ell-1}, x_{\ell-1}, u_\ell)$ is equal to the integer value 2ℓ. The lengths of a vertex-edge path $(x_0, u_0, \ldots, x_\ell, u_\ell)$ and of an edge-vertex path $(u_0, x_0, \ldots, u_\ell, x_\ell)$ are both equal to the integer value $2\ell + 1$. Let e_1 and e_2 be two elements in $\mathbb{G}^\bullet \cup \mathbb{G}^\times$. A *shortest path from e_1 to e_2* is a path of minimal length from e_1 to e_2. We denote by $L(e_1, e_2)$ the length of a shortest path from e_1 to e_2.

Let us now present the main notions of this section, namely edge-edge, edge-vertex, and vertex-edge distance maps.

Definition 5 *Let $X^\times$ and $X^\bullet$ be two subsets of $\mathbb{G}$.*

- **Edge-edge distance map.** *The (edge-edge) distance map to $X^\times$ is the map $D^{(\times,\times)}_{X^\times}$ from $\mathbb{G}^\times$ to the set of integers such that:*

$$D^{(\times,\times)}_{X^\times}(u) = \min\{L(v, u) \mid v \in X^\times\}, \text{ for any } u \in X^\times. \tag{16}$$

- **Vertex-edge distance map.** *The (vertex-edge) distance map to $X^\bullet$ is the map $D^{(\bullet,\times)}_{X^\bullet}$ from $\mathbb{G}^\times$ to the set of integers such that:*

$$D^{(\bullet,\times)}_{X^\bullet}(u) = \min\{L(x, u) \mid x \in X^\bullet\}, \text{for any } u \in X^\times. \tag{17}$$

- **Edge-vertex distance map.** *The (edge-vertex) distance map to $X^\times$ is the map $D^{(\times,\bullet)}_{X^\times}$ from $\mathbb{G}^\bullet$ to the set of integers such that:*

$$D^{(\times,\bullet)}_{X^\times}(x) = \min\{L(u, x) \mid u \in X^\times\}, \text{ for any } x \in X^\bullet. \tag{18}$$

In other words, given a subset $X^\times$ of edges, the edge-edge (resp. edge-vertex) distance map to $X^\times$ provides for each edge u (resp. vertex x) of $\mathbb{G}$ the minimal length of a path from u (resp. x) to an edge of $X^\times$ and, given a subset $X^\bullet$ of vertices, the vertex-edge distance map to $X^\bullet$ provides for each edge u of $\mathbb{G}$ the minimal length of a path from u to a vertex in $X^\bullet$.

The vertex-edge, edge-edge and edge-vertex distance maps are illustrated in Figures 5.

The distance maps introduced above are directly linked to the morphological operators presented in Section 2. Indeed, as stated by the following characterization theorem, the result of every dilation and of every erosion

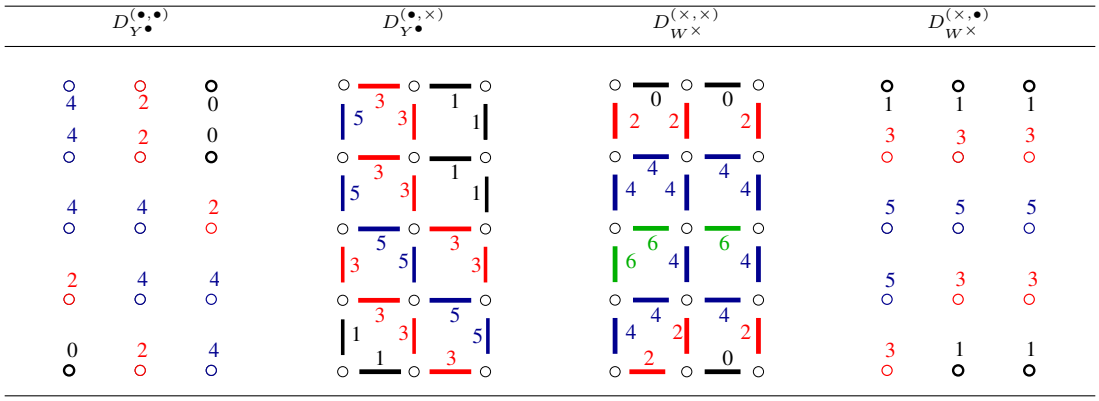


Fig. 5 Illustration of distance maps on graphs. The two first subfigures show the vertex-vertex and vertex-edge distance maps to the set $Y^\bullet$ of Figure 2 (in the graph $\mathbb{G}$ of Figure 1). The two last subfigures show the edge-edge and edge-vertex distance maps to the set $W^\times$ of Figure 4.

presented in Definition 2 can be obtained from distance maps. More precisely, any dilation or erosion of a set, for any given size parameter λ, can be obtained by thresholding a distance map at value λ according to the six relations established by Theorem 6 and to the two relations of Property 4.

Theorem 6 *Let λ be any positive integer.*

- *If λ is even, then the two following relations hold true:*

$$\Delta_{\lambda/2}(X^\times) = \{u \in \mathbb{G}^\times \mid D^{(\times,\times)}_{X^\times}(u) \leq \lambda\}, \text{ for any } X^\times \in \mathcal{G}^\times; \text{ and} \tag{19}$$
$$\varepsilon_{\lambda/2}(X^\times) = \{u \in \mathbb{G}^\times \mid D^{(\times,\times)}_{\overline{X^\times}}(u) > \lambda\}, \text{ for any } X^\times \in \mathcal{G}^\times, \tag{20}$$

  where $\overline{X^\times}$ is the complement of $X^\times$ in $\mathbb{G}^\times$, that is $\overline{X^\times} = \mathbb{G}^\times \setminus X^\times$.
- *If λ is odd, then the four following relations hold true:*

$$\delta_{\lambda/2}(X^\bullet) = \{u \in \mathbb{G}^\times \mid D^{(\bullet,\times)}_{X^\bullet}(u) \leq \lambda\}, \text{ for any } X^\bullet \in \mathcal{G}^\bullet; \tag{21}$$
$$\epsilon_{\lambda/2}(X^\bullet) = \{u \in \mathbb{G}^\times \mid D^{(\bullet,\times)}_{\overline{X^\bullet}}(u) > \lambda\}, \text{ for any } X^\bullet \in \mathcal{G}^\bullet; \tag{22}$$
$$\Delta_{\lambda/2}(X^\times) = \{x \in \mathbb{G}^\bullet \mid D^{(\times,\bullet)}_{X^\times}(x) \leq \lambda\}, \text{ for any } X^\times \in \mathcal{G}^\times; \text{ and} \tag{23}$$
$$\varepsilon_{\lambda/2}(X^\times) = \{x \in \mathbb{G}^\bullet \mid D^{(\times,\bullet)}_{\overline{X^\times}}(x) > \lambda\}, \text{ for any } X^\times \in \mathcal{G}^\times, \tag{24}$$

  where $\overline{X^\times}$ is the complement of $X^\times$ in $\mathbb{G}^\times$, that is $\overline{X^\times} = \mathbb{G}^\times \setminus X^\times$ and $\overline{X^\bullet}$ is the complement of $X^\bullet$ in $\mathbb{G}^\bullet$, that is $\overline{X^\bullet} = \mathbb{G}^\bullet \setminus X^\bullet$.

Proof

– Let us first prove that relation (21) holds true. Let λ be odd and let $X^\bullet \subseteq \mathbb{G}^\bullet$. The following statements are equivalent.

$$\delta_{\lambda/2}(X^\bullet) = \delta^\times \circ \delta_{(\lambda-1)/2}(X^\bullet), \quad \text{(by Equation 9)}$$
$$\delta_{\lambda/2}(X^\bullet) = \delta^\times(\{x \in \mathbb{G}^\bullet \mid D^{(\bullet,\bullet)}_{X^\bullet}(x) \leq \lambda - 1\}), \quad \text{(by Equation 14, since } \lambda - 1 \text{ is even)}$$
$$\delta_{\lambda/2}(X^\bullet) = \delta^\times(\{x \in \mathbb{G}^\bullet \mid \exists y \in X^\bullet, \exists \text{ a path from } y \text{ to } x \text{ of length not greater than } \lambda - 1\}), \quad \text{(by Definition 3)}$$
$$\delta_{\lambda/2}(X^\bullet) = \{\{x, z\} \in \mathbb{G}^\times \mid \exists y \in X^\bullet, \exists \text{ a path from } y \text{ to } x \text{ of length not greater than } \lambda - 1\}), \quad \text{(by Equation 4)}$$
$$\delta_{\lambda/2}(X^\bullet) = \{u \in \mathbb{G}^\times \mid \exists y \in X^\bullet, \exists \text{ a path from } y \text{ to } u \text{ of length not greater than } \lambda\}), \quad \text{(by definition of a path)}$$
$$\delta_{\lambda/2}(X^\bullet) = \{u \in \mathbb{G}^\times \mid D^{(\bullet,\times)}_{X^\bullet}(u) \leq \lambda\}, \quad \text{(by Equation 17)}$$

– Let us now prove that relation (19) holds true. Let λ be even and let $X^\times \subseteq \mathbb{G}^\times$. The following statements are equivalent.

$$\Delta_{\lambda/2}(X^\times) = (\delta^\times \circ \delta^\bullet)^{\lambda/2}(X^\times), \quad \text{(by Equation 7)}$$
$$\Delta_{\lambda/2}(X^\times) = \delta^\times \circ (\delta^\bullet \circ \delta^\times)^{(\lambda-2)/2} \circ \delta^\bullet(X^\times)$$
$$\Delta_{\lambda/2}(X^\times) = \delta^\times \circ \delta_{(\lambda-2)/2} \circ \delta^\bullet(X^\times), \quad \text{(by Equation 5)}$$
$$\Delta_{\lambda/2}(X^\times) = \delta_{(\lambda-1)/2} \circ \delta^\bullet(X^\times), \quad \text{(by Equation 9)}$$
$$\Delta_{\lambda/2}(X^\times) = \{u \in \mathbb{G}^\times \mid D^{(\bullet,\times)}_{\delta^\bullet(X^\times)}(u) \leq \lambda - 1\}, \quad \text{(by Equation 21)}$$
$$\Delta_{\lambda/2}(X^\times) = \{u \in \mathbb{G}^\times \mid \exists x \in \delta^\bullet(X^\times), \exists \text{ a path from } x \text{ to } u \text{ of length not greater than } \lambda - 1\}, \quad \text{(by Equation 17)}$$
$$\Delta_{\lambda/2}(X^\times) = \{u \in \mathbb{G}^\times \mid \exists v \in X^\times, \exists \text{ a path from } v \text{ to } u \text{ of length not greater than } \lambda\}, \quad \text{(by Equation 1)}$$
$$\Delta_{\lambda/2}(X^\times) = \{u \in \mathbb{G}^\times \mid D^{(\times,\times)}_{X^\times}(u) \leq \lambda\}, \quad \text{(by Equation 16)}$$

– We are now going to prove that relation (23) holds true. Let λ be odd and let $X^\times \subseteq \mathbb{G}^\bullet$. The following statements are equivalent.

$$\Delta_{\lambda/2}(X^\times) = \delta^\bullet \circ \Delta_{(\lambda-1)/2}(X^\times), \text{, by Equation 11}$$
$$\Delta_{\lambda/2}(X^\times) = \delta^\bullet \circ (\delta^\times \circ \delta^\bullet)^{(\lambda-1)/2}(X^\times), \quad \text{(by Equation 7)}$$
$$\Delta_{\lambda/2}(X^\times) = (\delta^\bullet \circ \delta^\times)^{(\lambda-1)/2} \circ \delta^\bullet(X^\times)$$
$$\Delta_{\lambda/2}(X^\times) = \delta_{(\lambda-1)/2} \circ \delta^\bullet(X^\times), \quad \text{(byEquation 5)}$$
$$\Delta_{\lambda/2}(X^\times) = \{x \in \mathbb{G}^\bullet \mid D^{(\bullet,\bullet)}_{\delta^\bullet(X^\times)}(x) \leq \lambda - 1\}, \text{(byEquation 14)}$$
$$\Delta_{\lambda/2}(X^\times) = \{x \in \mathbb{G}^\bullet \mid \exists y \in \delta^\bullet(X^\times), \exists \text{ a path from } y \text{ to } x \text{ of length not greater than } \lambda - 1\}, \quad \text{(by Definition 3)}$$
$$\Delta_{\lambda/2}(X^\times) = \{x \in \mathbb{G}^\bullet \mid \exists u \in X^\times, \exists \text{ a path from } u \text{ to } x \text{ of length not greater than } \lambda\} \quad \text{(by Equation 1)}$$
$$\Delta_{\lambda/2}(X^\times) = \{x \in \mathbb{G}^\bullet \mid D^{(\times,\bullet)}_{X^\times}(x) \leq \lambda\} \quad \text{(by Equation 18)}$$

– Relations (20), (22) and (22) can be deduced by duality from relations (19), (21) and (23), respectively. □

In other words, Theorem 6 states that, when λ is even, the set $\Delta_{\lambda/2}(X^\times)$ (resp. $\varepsilon_{\lambda/2}(X^\times)$) contains any edge with a value not greater than λ (resp. greater than λ) for the distance map $D_{X^\times}^{(\times,\times)}$ (resp. $D_{\overline{X^\times}}^{(\times,\times)}$). When λ is odd, the set $\delta_{\lambda/2}(X^\bullet)$ (resp. $\epsilon_{\lambda/2}(X^\bullet)$) contains any edge of value not greater than λ (resp. greater than λ) for the distance map $D_{X^\bullet}^{(\bullet,\times)}$ (resp. $D_{\overline{X^\bullet}}^{(\bullet,\times)}$) and the set $\Delta_{\lambda/2}(X^\times)$ (resp. $\varepsilon_{\lambda/2}(X^\times)$) contains any vertex of value not greater than λ (resp. greater than λ) for the distance map $D_{X^\times}^{(\times,\bullet)}$ (resp. $D_{\overline{X^\times}}^{(\times,\bullet)}$).

For example, the dilations $\{\delta_{\lambda/2}(Y^\bullet)\}$ for odd values of λ in $\{1,3\}$, which are shown in Figures 2, can be obtained by thresholding the vertex-edge distance map $D_{Y^\bullet}^{(\bullet,\times)}$ presented in the second column of Figures 5. The dilations $\{\Delta_{\lambda/2}(W^\times)\}$ with even (resp. odd) value of λ in $\{2,4\}$ (resp. $\{1,3\}$), which are shown in Figures 4, can be obtained by thresholding the edge-edge distance map $D_{W^\times}^{(\times,\times)}$ (resp. the edge-vertex distance map $D_{W^\times}^{(\times,\bullet)}$) presented in Figures 5.

From Property 4 and Theorem 6, we deduce that the result of any operator of Definition 2 can be obtained by thresholding a distance map at value λ, where λ is the size parameter of the operator. Thresholding, a (distance) map that weights the vertices (resp. edges) of a graph can be done in linear-time with respect to the number of vertices (resp. edges) of the graph with a sequential algorithm. Furthermore, a parallel algorithm that performs this computation in $O(n/p)$ time can be easily designed, where n is the number of vertices (resp. edges) of the graph and where p is the number of available processors. In both sequential and parallel cases, the overall complexity of the algorithms based on distance maps to compute the results of the operators of Definition 2 depend on the efficiency of distance maps algorithms. The next sections are devoted to sequential and parallel algorithms for distance maps on graphs.

4 Linear-time sequential algorithms for morphological operators on graphs

In this section, sequential algorithms to compute vertex-vertex, edge-edge, vertex-edge and edge-vertex distance maps in linear time with respect to the size of the graph $\mathbb{G}$ are presented. These algorithms are variations on breadth-first search. As stated at the end of the previous section, due to Property 4 and Theorem 6, these algorithms allow us for computing the results of all operators of Definition 2 in linear time with respect to the size of the graph, without any dependence to the size parameter of the operators.

Algorithm 1, presented below, allows us to compute both the vertex-vertex distance map $D_{X^\bullet}^{(\bullet,\bullet)}$ and the vertex-edge distance map $D_{X^\bullet}^{(\bullet,\times)}$ to a given subset $X^\bullet$ of vertices. The basic idea is to perform a breadth-first exploration of the edges and vertices of $\mathbb{G}$ from the elements of $X^\bullet$. The distance map value of each vertex (resp. edge) is computed when the vertex is first encountered during exploration. The breadth-first exploration, which ensures correct distance values, is made possible thanks to the auxiliary queue $\mathcal{Q}$ that is managed with a FIFO (First-In-First-Out) property.

In order to establish the correctness of Algorithm 1, let us analyze four invariants of Algorithm 1, *i.e.* four properties that hold true at every iterations of the main loop (line 6): i) every finite value associated to a point or an edge is the correct distance map value to the input set $X^\bullet$; ii) every vertex of $\mathbb{G}$ with a finite value that has an adjacent edge or vertex with an infinite value belongs to $\mathcal{Q}$; iii) the vertices of $\mathcal{Q}$ are mapped to finite values and they are stored in increasing order, the value of the first element of $\mathcal{Q}$ being $\mathcal{Q}_{\min}$ and the one of the last element being not greater than $\mathcal{Q}_{\min}+2$; and iv) any element e of $\mathbb{G}^\bullet$ or of $\mathbb{G}^\times$ such that there is a vertex x in $X^\bullet$ with $L(x,e) < \mathcal{Q}_{\min}$ is mapped to a finite value.

These invariants trivially hold true after the initialization step (lines 1-5). At every iteration of the main loop, a vertex x is popped from $\mathcal{Q}$ (line 6). Let λ be the distance map value of this vertex. In order to preserve properties i)-iv) above, all vertices and edges adjacent to x are analyzed thanks to the foreach loop at line 8. If an adjacent edge (resp. vertex) with an infinite value is found, then we can easily deduced that this element can be reach from $X^\bullet$ (through x) with a path of length $\lambda+1$ (resp. $\lambda+2$). Furthermore, such path is a shortest one (otherwise, due to invariant ii) and iv)), there would be an adjacent vertex in $\mathcal{Q}$ with a value smaller than λ, a contradiction

Algorithm 1: Vertex-vertex and vertex-edge distance maps.

Data: a connected graph $\mathbb{G} = (\mathbb{G}^\bullet, \mathbb{G}^\times)$, and subset $X^\bullet$ of $\mathbb{G}^\bullet$.
Result: the distance maps $D_{X^\bullet}^{(\bullet;\bullet)}$ and $D_{X^\bullet}^{(\bullet;\times)}$ to the set $X^\bullet$.

1 $\mathcal{Q}$:= an empty queue with FIFO property;
2 **foreach** *vertex* x *in* $\mathbb{G}^\bullet$ **do**
3 **if** $x \in X^\bullet$ **then** $\mathcal{Q}$.push(x); $D_{X^\bullet}^{(\bullet;\bullet)}(x) := 0$;
4 **else** $D_{X^\bullet}^{(\bullet;\bullet)}(x) := \infty$;
5 **foreach** *edge* $\{x, y\}$ *in* $\mathbb{G}^\times$ **do** $D_{X^\bullet}^{(\bullet;\times)}(\{x, y\}) := \infty$;
6 **while** $\mathcal{Q}$*.isNotEmpty()* **do**
7 $x := \mathcal{Q}$.pop();
8 **foreach** *vertex* y *adjacent to* x *in* $\mathbb{G}$ **do** // *i.e.,* when $\{x, y\} \in \mathbb{G}^\times$
9 **if** $D_{X^\bullet}^{(\bullet;\times)}(\{x, y\}) = \infty$ **then** $D_{X^\bullet}^{(\bullet;\times)}(\{x, y\}) := D_{X^\bullet}^{(\bullet;\bullet)}(x) + 1$;
10 **if** $D_{X^\bullet}^{(\bullet;\bullet)}(y) = \infty$ **then** $\mathcal{Q}$.push(y); $D_{X^\bullet}^{(\bullet;\bullet)}(y) := D_{X^\bullet}^{(\bullet;\bullet)}(x) + 1$;

with invariant iii)). Thus, the new value is correct and invariant i) is preserved. The newly discovered vertex is pushed in $\mathcal{Q}$ at line 10 in order to preserve invariant ii). Note that invariant iii) is trivially preserved since the newly inserted vertex has a value of $\lambda + 2$. Let us finally establish that invariant iv) holds true at the end of the while loop iteration. It can be seen that any element e in $\mathbb{G}^\bullet \cup \mathbb{G}^\times$ such that there is a vertex y in $X^\bullet$ with $L(y, e) < \mathcal{Q}_{\min}$ is adjacent to a vertex z such that $L(y, z) < Q_{\min} - 2$. But it can be observed that we have $\mathcal{Q}_{\min} \in \{\lambda, \lambda + 2\}$. Thus, we deduce that $L(y, z) < \lambda$. Hence, since invariant iv) holds true at the beginning of the while loop iteration, we deduce that z is mapped to a finite value. Furthermore, since $L(y, z) < \lambda$, we deduce that z is not in $\mathcal{Q}$. Therefore, by (the contraposition of) ii), we may affirm that e is mapped to a finite value, which proves that invariant iv) holds true at the end of the iteration of the while loop. Thus, invariants i)-iv) hold true when the foreach loop at line 8 terminates. Hence, we deduce that when the algorithm halts the produced maps $D_{X^\bullet}^{(\bullet;\bullet)}$ and $D_{X^\bullet}^{(\bullet;\times)}$ are indeed the vertex-vertex distance and the the vertex-edge distance map to $X^\bullet$.

Let us now analyze the time complexity of Algorithm 1. The initialization of the queue $\mathcal{Q}$ (line 1) is done in constant time. Since a push operation on a queue can be done in constant time, the overall complexity of the initialization loop at lines 1-4 is done in $O(|\mathbb{G}^\bullet|)$ and the edge distance map initialization is done in $O(|\mathbb{G}^\times|)$ by the loop at line 5. In order to analyze the complexity of the main loop of Algorithm 1, it is important to note that every vertex of the graph is pushed at most once in $\mathcal{Q}$. This is ensured by immediately setting to a finite value a point which is inserted in $\mathcal{Q}$ and by only inserting in $\mathcal{Q}$ points with an infinite value (see lines 3 and 10). Thus, since a vertex is popped from $\mathcal{Q}$ at every iteration of the while loop, we deduce that there are at most $|V|$ iterations of this loop. At each iteration of the while loop the vertices and edges adjacent to the vertex which is popped are explored (foreach loop at line 8). Thus, if the graph $\mathbb{G}$ is represented as an array of lists associating to each vertex the list of its adjacent edges, the complexity of line 8 is $O(|V| + |E|)$. The instructions, inside this foreach loop are executed at most $2 \times |E|$ times, each of them being individually performed in constant time. Therefore, the overall complexity of these instructions is $O(|E|)$. Thus, we deduce that the overall complexity of the algorithm is $O(|V| + |E|)$.

Algorithm 2, presented hereafter, is similar to Algorithm 1 but allows us to compute both the edge-edge and edge-vertex distance maps to a given subset of edges instead of the vertex-vertex and vertex-edge distance maps to a subset of vertices. The initialization steps (lines 2 to 8), which has to be made from a set of edges rather than from a set of vertices, is the main difference with Algorithm 1. However, the same arguments can be used to establish the correctness and complexity of Algorithm 2. Thus, the following property can be deduced.

Algorithm 2: Edge-edge and edge-vertex distance maps.

Data: a connected graph $\mathbb{G} = (\mathbb{G}^\bullet, \mathbb{G}^\times)$ and a subset $X^\times$ of $\mathbb{G}^\times$.
Result: the distance maps $D_{X^\times}^{(\times,\times)}$ and $D_{X^\times}^{(\times,\bullet)}$ to the set $X^\times$.

$\mathcal{Q}$:= an empty queue with FIFO property;
foreach *vertex* x *in* $\mathbb{G}^\bullet$ **do** $D_{X^\times}^{(\times,\bullet)}(x) := \infty$;
foreach *edge* $\{x, y\}$ *in* $\mathbb{G}^\times$ **do**
  if $\{x, y\} \in X^\times$ **then**
    if $D_{X^\times}^{(\times,\bullet)}(x) == \infty$ **then** $\mathcal{Q}$.push(x); $D_{X^\times}^{(\times,\bullet)}(x) := 1$;
    if $D_{X^\times}^{(\times,\bullet)}(y) == \infty$ **then** $\mathcal{Q}$.push(y); $D_{X^\times}^{(\times,\bullet)}(y) := 1$;
    $D_{X^\times}^{(\times,\times)}(\{x, y\}) := 0$;
  else $D_{X^\times}^{(\times,\times)}(\{x, y\}) := \infty$;
while $\mathcal{Q}$*.isNotEmpty()* **do**
  $x := \mathcal{Q}$.pop();
  foreach *vertex* y *adjacent to* x *in* $\mathbb{G}$ **do** // *i.e.,* when $\{x, y\} \in \mathbb{G}^\times$
    if $D_{X^\times}^{(\times,\times)}(\{x, y\}) == \infty$ **then** $D_{X^\times}^{(\times,\times)}(\{x, y\}) = D_{X^\times}^{(\times,\bullet)}(x)+1$;
    if $D_{X^\times}^{(\times,\bullet)}(y) == \infty$ **then** $\mathcal{Q}$.push(y); $D_{X^\times}^{(\times,\bullet)}(y) := D_{X^\times}^{(\times,\bullet)} + 2$;

Property 7 *Algorithm 1 outputs two maps* $D_{X^\bullet}^{(\bullet,\bullet)}$ *and* $D_{X^\bullet}^{(\bullet,\times)}$ *that are the vertex-vertex and vertex-edge distance maps, respectively, to the input subset* $X^\bullet$ *of* $\mathbb{G}^\bullet$*. Algorithm 2 outputs two maps* $D_{X^\times}^{(\times,\times)}$ *and* $D_{X^\times}^{(\times,\bullet)}$ *that are the edge-edge and edge-vertex distance maps, respectively, to the input subset* $X^\times$ *of* $\mathbb{G}^\times$*. Furthermore Algorithms 1 and 2 both run in linear time with respect to* $|V| + |E|$*.*

5 Parallel strategy for distance maps on graphs

This section presents a parallel strategy to compute the distance maps introduced in Section 3, hence the mathematical morphology operators on graphs of [9]. To this end, related parallelization strategies of distance map algorithms are first presented (Section 5.1). Then, the proposed parallel algorithm is introduced (Section 5.2) and necessary auxiliary functions are described (Section 5.3). Finally, the time complexity of the proposed algorithm is analyzed (Section 5.4).

5.1 Related work on parallel algorithms for distance maps

Several parallel and/or separable algorithms for computing distance maps on images have been proposed. In particular, based on the regular structure of the space such computations use a static partitioning of the input image into rows, columns, or blocks processed in parallel. For example, Shyu *et al.*[30] proposed an efficient parallel algorithm to compute the Chamfer distance transformation of a binary image. Shyu *et al.*'s parallel implementation requires the decomposition of the input image into bands, each band is distributed to a processor. In each processor, the distance transformation can be computed by preforming two passes over the image : a forward pass to propagate the distance transform from the causal neighbors, followed by a backward pass to propagate the distance transform from anti-causal neighbors. This method computes the distance transformation on a distributed system. Therefore, the intermediate results across several processors must be synchronized using Message Passing Interface (MPI). Pham *et al.*[25] presented a parallel implementation of distance transformation

using OpenMP. In fact, this implementation is based on the parallel execution of the sequential Chamfer distance transformation proposed in [30] using the shared memory model on multicore CPUs. The parallel implementation of this distance transformation requires more than one iteration of forward and backward passes unlike the Chamfer distance transformation. As a result, the distance transformation can be propagated from one band to the next one in following iterations contrary to [30]. Man *et al.*[20] [21] presented a parallel algorithm for computing Euclidean distance map on both multicore processors and GPU system. The idea of this algorithm consists of performing two steps, in each step both a forward and backward scan is performed. In fact, The input image is partitioned into rows and columns where in the first step columns are scanned and in the second rows are scanned. The scanning of a particular column (resp. row) is independent to the scanning of the others columns (resp. rows). The work of [28] described a separable algorithm to efficiently compute the distance transformation, the separability means that the computations are performed dimension by dimension.

5.2 Parallel algorithm for distance maps on graphs

Contrary to the parallel computation of distance maps on an image, which, as seen in the previous subsection, is often based on a static partitioning of the image into rows, columns or blocks processed in parallel, our parallelization strategy on graphs is based on dynamic partitioning. The partition depends on the input set and is iteratively computed during the execution. More precisely, our strategy iteratively considers the successive level-sets of the distance maps, each level set being partitioned and then traversed in parallel. In this section, this parallel strategy is presented and a precise description of a parallel algorithm for vertex-vertex and vertex-edge distance maps (*i.e.*, a parallelization of Algorithm 1) is given. For the sake of simplicity, we only give a precise description of a parallelization of Algorithm 1, but the proposed strategy can also be adapted to edge-edge and and edge-vertex distance maps computations to obtain a parallelization of Algorithm 2.

Let us first present our strategy from a high level point of view. To this end, we recall the notion of a level set. Given an integer λ and a (distance) map D from $\mathbb{G}^\bullet$ in the set of integers, the λ-*level set* of D is the set of all elements of value λ for D (*i.e.*, the set $\{x \in \mathbb{G}^\bullet \mid D(x) = \lambda\}$).

We are now ready for providing the overview of Algorithm 3. Given a subset $X^\bullet$ of $\mathbb{G}^\bullet$, after an initialization step where an integer variable λ is set to 0 and where the elements of $X^\bullet$ are inserted in a variable set E (hence E is the $(\lambda = 0)$-level-set of $D_{X^\bullet}^{(\bullet;\bullet)}$), our algorithm can be sketched as follows:

1. Partition E (*i.e.*, the λ-level set of $D_{X^\bullet}^{(\bullet;\bullet)}$) into p balanced subsets $E_1, \ldots, E_p$ (line 10 of Algorithm 3).
2. Assign each of the p subsets $E_1, \ldots, E_p$ to one of the p processors (line 11 of Algorithm 3).
3. Let, in parallel, each processor browse on the neighbors of the elements in its assigned subset E_i, insert those which are not yet traversed into a private variable set S_i, and set their distance map values to $\lambda + 2$ (lines 13 to 18 of Algorithm 3).
4. Merge the private sets $\{S_i \mid i \in \{1, \ldots, p\}\}$ and store the result in E so that E becomes the $(\lambda + 2)$-level set of $D_{X^\bullet}^{(\bullet;\bullet)}$ (line 19 of Algorithm 3).
5. Increment λ and repeat steps 1-4 until E becomes empty (line 20).

In Step 3, in order to concurrently check if a vertex has been already traversed, we need to equip each vertex with a synchronization Boolean variable that is handled with an atomic *test-and-set* instruction. The test-and-set instruction sets a given variable to true and returns its old value as a single atomic (*i.e.*, non-interruptible) instruction.

Algorithm 3 provides the precise description of our parallel strategy. It uses two auxiliary functions called *Partition* and *Union*. The function *Partition* takes two arguments. The first one is the set E to be partitioned and the second one is an integer p corresponding to the number of classes of the returned partition. The function *Partition* returns a balanced partition of E into p classes, the partition being balanced in the sense that any of its classes contains either $|E|/p$ or $|E|/p + 1$ elements. The function *Union* is simply computing the union of its arguments.

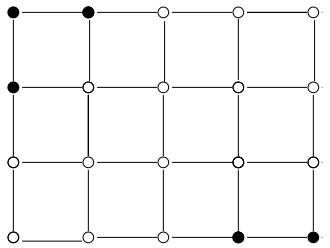
Algorithm 3: Parallel vertex-vertex and vertex-edge distance maps.

Data: A connected graph $(\mathbb{G}^\bullet, \mathbb{G}^\times)$, a subset $X^\bullet$ of $\mathbb{G}^\bullet$, the number p of processors.
Result: The distance maps $D_{X^\bullet}^{(\bullet;\bullet)}$ and $D_{X^\bullet}^{(\bullet;\times)}$ to the set $X^\bullet$.

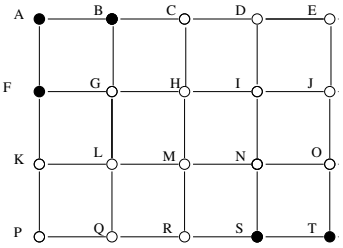
```
1  E := ∅; λ :=0;
2  Set to False all elements of a shared Boolean array Traversed of size |𝔾•|
3  (E_1, ...,E_p):= Partition(X•, p);
4  (F_1, ...,F_p):= Partition(𝔾×, p);
5  foreach processor i in {1, ...,p} do in parallel
6      foreach vertex x ∈ E_i do D_{X•}^{(•;•)}(x) := λ; Traversed[x] := True ;
7      foreach edge e ∈ F_i do D_{X•}^{(•;×)}(e) := ∞ ;
8  E:=Union(E_1, ..., E_p);
9  while E ≠ ∅ do
10     (E_1, ...,E_p):= Partition(E, p);
11     foreach processor i in {1, ..., p} do in parallel
12         S_i:=∅;
13         foreach x in E_i do
14             foreach vertex y adjacent to x in 𝔾 do // i.e., when {x, y} ∈ 𝔾×
15                 if D_{X•}^{(•;×)}({x, y}) = ∞ then D_{X•}^{(•;×)}({x, y}) := λ + 1;
16                 if test-and-set(Traversed[y]) = False then
17                     S_i:=S_i ∪ {y};
18                     D_{X•}^{(•;•)}(y) := λ;
19     E:=Union(S_1, ..., S_p);
20     λ:=λ + 2;
```

Before further studying Algorithm 3, let us analyze an execution example. To this end, the graph $\mathbb{G} = (\mathbb{G}^\bullet, \mathbb{G}^\times)$ and the set $X^\bullet = \{A, B, F, S, T\}$ shown Figure 6(a,b) are considered. For the sake of simplicity, in this example, let us focus only on the distance map $D_{X^\bullet}^{(\bullet;\bullet)}$ and let us discard the instructions concerning $D_{X^\bullet}^{(\bullet;\times)}$. Figure 6(c) illustrates the iterations of the main loop of Algorithm 3 with three processors, *i.e.*, with $p = 3$. After the initialization, we have $E = X^\bullet = \{A, B, F, S, T\}$, $\lambda = 0$, and $D_{X^\bullet}^{(\bullet;\bullet)}(A) = D_{X^\bullet}^{(\bullet;\bullet)}(B) = D_{X^\bullet}^{(\bullet;\bullet)}(F) = D_{X^\bullet}^{(\bullet;\bullet)}(S) = D_{X^\bullet}^{(\bullet;\bullet)}(T) = 0$. In the first iteration, the first step consists of partitioning the input set $E = \{A, B, F, S, T\}$ into 3 balanced subsets $E_1 = \{A, B\}$, $E_2 = \{F, S\}$, $E_3 = \{T\}$. Then, the sets E_1, E_2, and E_3 are assigned to the processors 1, 2, and 3, respectively. Each processor p_i ($i \in \{1, 2, 3\}$), in parallel, explores the non-already traversed neighbors of the vertices in its subset E_i and inserts these neighbors into a private set. As shown in Figure 6(c), the resulting private sets S_1, S_2, and S_3 are such that $S_1 = \{C, G\}$, $S_2 = \{K, R, N\}$, $S_3 = \{O\}$. Observe that the vertex G (written in bold in the figure) was concurrently found by two distinct processors (numbered 1 and 2) since it is a successor of B, which belongs to E_1, and a successor of F, which belongs to E_2. But thanks to the test on the synchronized Boolean array *Traversed*, it was actually inserted only into S_1 and not into S_2. When an element is inserted into a set S_i, its distance map value is updated with the current value of λ plus 2 which here is equal to 2 since we are at the first iteration. Thus, we now have $D_{X^\bullet}^{(\bullet;\bullet)}(C) = D_{X^\bullet}^{(\bullet;\bullet)}(G) = D_{X^\bullet}^{(\bullet;\bullet)}(K) = D_{X^\bullet}^{(\bullet;\bullet)}(R) D_{X^\bullet}^{(\bullet;\bullet)}(N) = D_{X^\bullet}^{(\bullet;\bullet)}(0) = 2$. Then, once the parallel neighbor search on E_1, E_2, and E_3 is over, the sets S_1, S_2, and S_3 are merged thanks to a call to the *Union* function resulting into the updated set $E = \{C, G, K, R, N, O\}$. This updated set E, the 2-level set of $D_{X^\bullet}^{(\bullet;\bullet)}$, is considered for the next iteration of the main while loop of the algorithm. Before starting a new iteration the

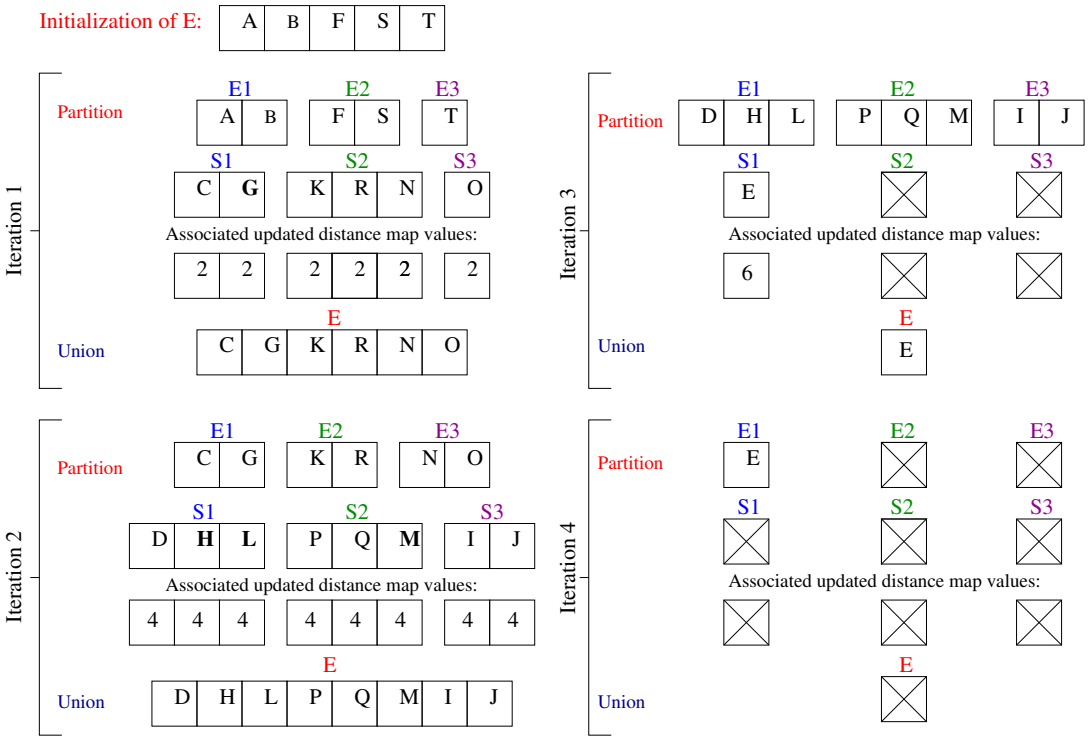
value of λ is updated and therefore set to 2. After this first iteration, since E is nonempty, a second iteration of the main while loop is considered. In fact, on this example, four iterations of the main loop are necessary to compute the resulting distance maps, as shown on Figure 6(c).



(a) A subset $X^{\bullet}$ of vertices (bold)



(b) Labeling of the vertices of $\mathbb{G}$.



(c) The different steps of our parallel algorithm.

Fig. 6 Illustration of the proposed parallel algorithm (Algorithm 3): a step-by-step execution example with $p = 3$ processors.

5.3 Parallel partition and disjoint union algorithms

Let us now present the parallel algorithms for the *Partition* and *Union* functions used in Algorithm 3.

The parallel partition algorithm (see Algorithm 4) consists of computing in parallel, with p processors, a *balanced partition* $\{E_1, \ldots, E_p\}$ of a set E. The partition is balanced in the sense that the k first sets of the partition contain $|E|/p+1$ elements whereas the following ones contain $|E|/p$ elements, where k is the remainder in the integer division of $|E|$ by p and where a/b denotes the quotient of the integer division of a by b. The elements of E, stored in an array of size $|E|$, are moved to arrays previously allocated for the subsets $E_1, \ldots, E_p$ in the order of their indices: the first set receives the first elements of the array E and so on (see Figure 7). Thus, each processor computes the index of the first and of the last element that must be copied (lines 2 to 6) before actually copying the elements of E located between the computed indices (line 7). The computation of the first and of the last indices can be done in constant time and the copying step is done in linear time with respect to $|E|/p$ (each processor moves at most $|E|/p+1$ elements according to its number of elements). For these steps, there is no dependence between processors. Thus, all processors can perform these steps in parallel.

Algorithm 4: Partition.

Data: An array E of $n = |E|$ elements, the number p of processors.
Result: A balanced partition (E_1, ...,E_p) of E.

```
1 foreach processor i in {1, ...,p} do in parallel
2     if i ≤ (n mod p) then
3         start[i] := (i − 1) ∗ (n/p + 1); end[i] := start[i] + n/p;
4     else
5         start[i] := (n mod p) ∗ (n/p + 1) + (i − 1 − (n mod p)) ∗ (n/p);
6         end[i] := start[i] + n/p - 1;
7     foreach j_i in {start[i], ..., end[i]} do E_i[j_i − start[i]] := E[j_i] ;
```

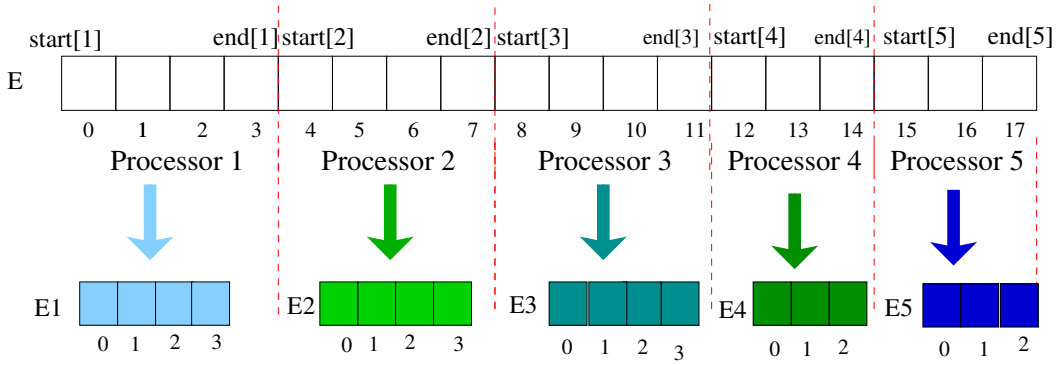


Fig. 7 Illustration of the *Partition* algorithm with $p = 5$ processors.

The parallel *Union* algorithm (see Algorithm 5) computes the union of p disjoint sets $\{S_1, \ldots, S_p\}$ with p processors. The elements of each set are stored in an array and each processor p_i ($i \in \{1, \ldots, p\}$) copies the

elements of the array S_i in the array E. The elements of S_i are stored consecutively in the resulting array E from the index $start[i]$, where $start[i]$ is the sum of the cardinalities of the sets $S_1, \ldots, S_{i-1}$ (see Figure 8): $start[i] = \sum_{j \in \{1,\ldots,i-1\}} |S_j|$. Thus, our algorithm first computes the values $\{start[i]$ for any i in $\{1, \ldots, p\}\}$ (line 1) before actually copying the elements into E (line 3). Given the cardinalities $|S_1|, \ldots, |S_p|$, computing the values $start[i]$ for any i in $\{1, \ldots, p\}$ is known as the prefix-sum problem. It can be solved in parallel with p processors with a $O(\log_2 p)$ running-time algorithm [17]. Then, each processor i copies in parallel (line 3) the elements of S_i into E at the correct position.

Algorithm 5: Union.

Data: A series $S_1, \ldots, S_p$ of p sets.
Result: An array E whose elements constitutes the union of $\{S_1, \ldots, S_p\}$.
1 start = ParallelPrefixSum($|S_1|, \ldots, |S_p|$);
2 **foreach** *processor* i *in* $\{1, \ldots, p\}$ **do in parallel**
3 ⌊ **foreach** j_i *in* $\{0, \ldots, |S_i| - 1\}$ **do** $E[start[i] + j_i] := S_i[j_i]$;

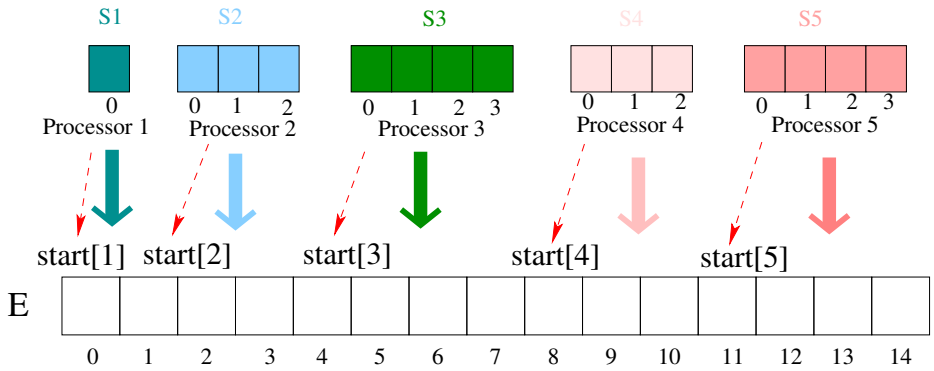


Fig. 8 Illustration of the *Union* algorithm with $p = 5$ processors.

5.4 Complexity analysis

In this subsection, the worst-case time complexity of Algorithm 3 is analyzed (Theorem 11). This complexity depends of the ratio, called the unbalancing balancing factor of the graph (Definition 9), of the maximum degree of a vertex of the graph over the minimum one. The complexity of Algorithm 3 is analyzed under an assumption about the graph and set under scrutiny. Indeed, in some "degenerated" cases, which we seek to avoid by this assumption, the distance map computation problem becomes sequential. This is the case, in particular, when the graph depicts a "linear shape", *e.g.*, when each vertex is adjacent to at most two other vertices. An example of such graph is shown in Figure 9a. In order to handle such problematic situations, the following notion of a regular set is proposed.

Definition 8 (regular set) *Let p be a positive integer and let $X^\bullet$ be a subset of $\mathbb{G}^\bullet$. We say that $X^\bullet$ is* p-regular *(for $\mathbb{G}$) if every nonempty level-set of $D_{X^\bullet}^{(\bullet,\bullet)}$ contains at least p elements.*

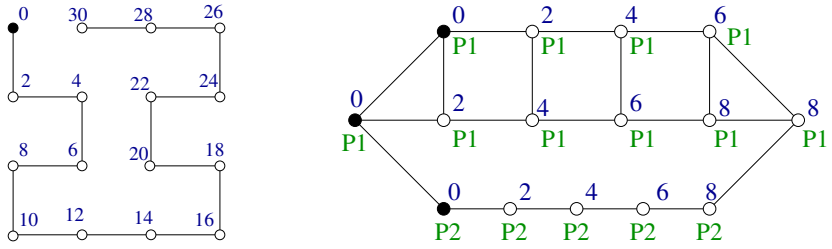


Fig. 9 (a) A graph $\mathbb{G}$ and a subset $X^\bullet$ (in black) of vertices that is 1-regular but not 2-regular. The distance map $D_{X^\bullet}^{(\bullet;\bullet)}$ is given by the blue numbers. (b) A graph whose unbalancing factor is $\frac{3}{2}$ used to illustrate a situation of unbalanced workload obtained with Algorithm 3 (see text).

For instance, if we consider the graph $\mathbb{G}$ of Figure 9(a), the set $X^\bullet$ (black vertices in Figure 9(a)) is 1-regular but not 2-regular since every level set of $D_{X^\bullet}^{(\bullet;\bullet)}$ (represented by blue numbers in the figure) contains a single vertex. It can be seen that when Algorithm 3 is applied to the graph $\mathbb{G}$ and the set $X^\bullet$ of this example, then, at each iteration, there is a single nonempty set among the sets $E_1, \ldots, E_p$ obtained after the call to the *Partition* function. Thus, it can be observed, that, at each iteration, there is no parallel calculus done in the parallel loop line 11. Hence, in order to study the complexity of Algorithm 3, it is assumed in the following that the input set $X^\bullet$ is p-regular, p-being the number of available processors. This ensures that any of the sets $E_1, \ldots, E_p$, obtained after the call to the *Partition* function, is nonempty. Hence, every processor processes a nonempty set E_i at every iteration of the parallel loop line 11. In Section 7, this assumption is practically assessed in the context of morphological processing of images and textured meshes.

Another important issue for studying the time complexity of Algorithm 3 is related to the degrees of the vertices of $\mathbb{G}$. Indeed, after line 10 of Algorithm 3, the sets $\{E_i \mid i \in \{1, \ldots, p\}\}$ are balanced and each processor must traverse all the neighbors of exactly one of these sets. Thus, if all vertices of the graph have the same degree (*i.e.*, if they all belong to the same number of edges), the workload is well balanced over the processors. On the other hand, if the degree varies from one vertex to another one, then, one may easily end up in a situation which is unfair for one of the processors. An example of such situation is provided in Figure 9(b) which illustrates a possible execution of Algorithm 3 on the set $X^\bullet$ of black vertices with two processors called P1 and P2. It can be seen that every level set of $D_{X^\bullet}^{(\bullet;\bullet)}$ contains three vertices which are partitioned at line 10 of Algorithm 3 into two sets E_1 and E_2, handled respectively by P1 and P2. In the figure, we indicate in green the processor which traverse every vertex. It can be seen that the obtained partition is, at each iteration, such that E_1 contains two vertices and E_2 contains one vertex. Furthermore, each vertex in E_1 belongs to three edges, whereas each vertex in E_2 belongs to a single edge. Overall, it can thus be checked that P1 must browse over 30 edges (10 vertices contained in 3 edges) in the loop line 14 whereas P2 browses over only 10 edges (5 vertices contained in two edges), leading to an unbalanced workload between P1 and P2. The occurrence of such situation and its amplitude can be bounded thanks to the notion of unbalancing factor of a graph, that is introduced hereafter (Definition 9). Then, the time-complexity of Algorithm 3, established in Theorem 11, can be expressed as a function of this unbalancing factor.

We recall that, if x is a vertex of $\mathbb{G}$, the *degree of* x, denoted by $d(x)$, is the number of edges that contain x, *i.e.*, $d(x) = |\{\{x, y\} \in \mathbb{G}^\times\}|$. Furthermore, we denote by $d_{\min(\mathbb{G})}$ (resp. $d_{\max(\mathbb{G})}$) the minimum (resp. maximum) of the degrees of the vertices of $\mathbb{G}$: $d_{\min}(\mathbb{G}) = \min\{d(x) \mid x \in \mathbb{G}^\bullet\}$ (resp. $d_{\max}(\mathbb{G}) = \max\{d(x) \mid x \in \mathbb{G}^\bullet\}$).

Definition 9 (unbalancing factor) *The* unbalancing factor of $\mathbb{G}$, *denoted by* $\beta(\mathbb{G})$, *is the fraction:*

$$\beta(\mathbb{G}) = \frac{d_{\max}(\mathbb{G})}{d_{\min}(\mathbb{G})}. \tag{25}$$

For instance, the unbalancing factor of the graph $\mathbb{G}$ shown in Figure 9(b) is equal to 3/2 since $d_{\min}(\mathbb{G}) = 2$ and $d_{\max}(\mathbb{G}) = 3$. The unbalancing factor of the 4-adjacency graph of Figure 6(a) is 2. Note that if we modify this graph by identifying the vertices of the first and last lines and those of the first and last columns (so that a torus is obtained), then the degree of every vertex in the obtained graph is equal to 4, leading to an unbalancing factor of 1.

The time complexity of Algorithm 3 depends of the unbalancing factor of $\mathbb{G}$. The next lemma is an important result in order to establish this time complexity.

Lemma 10 *Let p be a positive integer and let $X^\bullet$ be a p-regular subset of $\mathbb{G}^\bullet$. When Algorithm 3 is executed on $\mathbb{G}$, $X^\bullet$, and p, then the following statements hold true.*

1. *The instruction lines 15 and 16 are executed less than $\frac{4\beta(\mathbb{G})}{2\beta(\mathbb{G})+p+1}|\mathbb{G}^\times|$ times by each processor;*
2. *Using an array of linked lists to represent the graph $\mathbb{G}$ (*i.e.*, one linked list for any vertex to store its adjacent edges), the continuation condition in the for each loop at line 14 is tested less than $\frac{2}{p+1}|\mathbb{G}^\bullet| + \frac{4\beta(\mathbb{G})}{2\beta(\mathbb{G})+p+1}|\mathbb{G}^\times|$ times by each processor.*

Proof At every iteration of the main loop line 9 the set E is the λ-level set of $D_{X^\bullet}^{(\bullet,\bullet)}$. Let n_λ be the cardinality of E. Then, by definition of the *Partition* function, for any i in $\{1,\ldots,p\}$, the number $n_{\lambda,i}$ of elements in the set E_i obtained at line 10 is either equal to n_λ/p or to $n_\lambda/p+1$, where a/b denotes the quotient in the integer division of a by b. Let $\delta^\times(E)$ (resp. $\delta^\times(E_i)$) be the set that contains any edge with a vertex in E (resp. in E_i). Let us also denote by m_λ and $m_{\lambda,i}$ the cardinalities of $\delta^\times(E)$ and $\delta^\times(E_i)$, respectively. Note that for a given value of λ, the instruction lines 15 and 16 are executed exactly $m_{\lambda,i}$ times by the processor i. In order to establish Lemma 10.1, we are going to prove that $m_{\lambda,i} \leq \frac{2\beta(\mathbb{G})}{2\beta(\mathbb{G})+p-1} m_\lambda$. To this, end we are going to distinguish two cases.

– Let us first assume that $n_\lambda > p$. Thus, we have $q \leq n_{\lambda,i} \leq 2q$, where q is the quotient in the integer division of n_λ by $(p+1)$. Since $n_\lambda > p$, the quotient q is nonnull. By definition of $d_{\min}(\mathbb{G})$ and $d_{\max}(\mathbb{G})$, we have $d_{\max}(\mathbb{G}).n_{\lambda,i} \leq m_{\lambda,i} \leq d_{\max}(\mathbb{G}).n_{\lambda,i}$. Hence, we deduce that $d_{\min}(\mathbb{G})q \leq m_{\lambda,i} \leq 2d_{\max}(\mathbb{G})q$. By definition of m_λ, we have:

$$\frac{m_{\lambda,i}}{m_\lambda} = \frac{m_{\lambda,i}}{m_{\lambda,i} + \sum_{j\in\{1,\ldots,p\}\setminus\{i\}} m_{\lambda,j}} = 1 - \frac{\sum_{j\in\{1,\ldots,p\}\setminus\{i\}} m_{\lambda,j}}{m_{\lambda,i} + \sum_{j\in\{1,\ldots,p\}\setminus\{i\}} m_{\lambda,j}}. \tag{26}$$

Hence, since $m_{\lambda,i} \leq 2d_{\max}(\mathbb{G})q$, we deduce that:

$$\frac{m_{\lambda,i}}{m_\lambda} \leq 1 - \frac{\sum_{j\in\{1,\ldots,p\}\setminus\{i\}} m_{\lambda,j}}{2d_{\max}(\mathbb{G})q + \sum_{j\in\{1,\ldots,p\}\setminus\{i\}} m_{\lambda,j}} \quad \text{, which can be rewritten as:} \tag{27}$$

$$\frac{m_{\lambda,i}}{m_\lambda} \leq \frac{2d_{\max}(\mathbb{G})q}{2d_{\max}(\mathbb{G})q + \sum_{j\in\{1,\ldots,p\}\setminus\{i\}} m_{\lambda,j}} \tag{28}$$

Since, $d_{\min}(\mathbb{G})q \leq m_{\lambda,j}$, for any $j \in \{1,\ldots,p\}$ we deduce that:

$$\frac{m_{\lambda,i}}{m_\lambda} \leq \frac{2d_{\max}(\mathbb{G})q}{2d_{\max}(\mathbb{G})q + (p-1)d_{\min}(\mathbb{G})q} \quad \text{, which can be simply rewritten as:} \tag{29}$$

$$\frac{m_{\lambda,i}}{m_\lambda} \leq \frac{2d_{\max}(\mathbb{G})}{2d_{\max}(\mathbb{G}) + (p-1)d_{\min}(\mathbb{G})} \quad \text{, which can again be rewritten as:} \tag{30}$$

$$\frac{m_{\lambda,i}}{m_\lambda} \leq \frac{2\frac{d_{\max}(\mathbb{G})}{d_{\min}(\mathbb{G})}}{2\frac{d_{\max}(\mathbb{G})}{d_{\min}(\mathbb{G})} + (p-1)\frac{d_{\min}(\mathbb{G})}{d_{\min}(\mathbb{G})}} \quad \text{, which can be simplified as:} \tag{31}$$

$$\frac{m_{\lambda,i}}{m_\lambda} \leq \frac{2\beta(\mathbb{G})}{2\beta(\mathbb{G}) + p - 1}. \tag{32}$$

Thus, since m_λ is positive, we deduce that:

$$m_{\lambda,i} \leq \frac{2\beta(\mathbb{G})}{2\beta(\mathbb{G}) + p - 1} m_\lambda. \tag{33}$$

– Let us now assume that $n_\lambda = p$. In this case, for any $i \in \{1, \ldots, p\}$, we have $n_{\lambda,i} = 1$ and $d_{\min}(\mathbb{G}) \leq m_{\lambda,i} \leq d_{\max}(\mathbb{G})$. Using similar arguments as in the previous case, we deduce that

$$m_{\lambda,i} \leq \frac{\beta(\mathbb{G})}{\beta(\mathbb{G}) + p - 1} m_\lambda \tag{34}$$

Hence, since $2\beta(\mathbb{G}) \geq \beta(\mathbb{G}) \geq 0$, we also have in this second case:

$$m_{\lambda,i} \leq \frac{2\beta(\mathbb{G})}{2\beta(\mathbb{G}) + p - 1} m_\lambda. \tag{35}$$

Let us set $m_i = \sum_{\lambda \in \mathbb{N}} m_{\lambda,i}$. It can be seen that m_i is the overall number of executions of the instruction lines 15 and 16 by the processor i. Since the set of all level-sets of $D_{X_\bullet^\bullet}^{(\bullet;\bullet)}$ partitions $\mathbb{G}^\bullet$ and since every edge contains exactly two vertices of $\mathbb{G}$, we deduce that $\sum_{\lambda \in \mathbb{N}} m_\lambda = 2.|\mathbb{G}^\times|$. Hence, from Equations 33 and 35, we deduce the following inequation that establishes Lemma 10.1:

$$m_i \leq \frac{4\beta(\mathbb{G})}{2\beta(\mathbb{G}) + p - 1} |\mathbb{G}^\times|. \tag{36}$$

Let us now establish Lemma 10.2. To this end, it can be seen that there are, for each processor $i \in \{1, \ldots, p\}$, m_i positive continuation tests at line 14 of Algorithm 3. It can also be seen that, for every value of λ, there is one negative test for each element in E_i. Hence, for any processor i and any value of λ, there is at most $\frac{2n_\lambda}{p+1}$ negative tests. Hence, since the level-set of $D_{X_\bullet^\bullet}^{(\bullet;\bullet)}$ partitions $\mathbb{G}^\bullet$, and since n_λ is the number of elements in the λ-level set of $D_{X_\bullet^\bullet}^{(\bullet;\bullet)}$, we deduce that, for any processor, there are at most $\frac{2|\mathbb{G}^\bullet|}{p+1}$ negative tests at line 14 of Algorithm 3. Hence, the condition in the for each loop at line 14 is tested less than $\frac{2}{p+1}|\mathbb{G}^\bullet| + \frac{4\beta(\mathbb{G})}{p+1+2.\beta(\mathbb{G})}|\mathbb{G}^\times|$ times by each processor. □

Let us now analyze the complexity of Algorithm 3. At every iteration of the main loop (line 9), the set E is the λ-level set of $D_{X_\bullet^\bullet}^{(\bullet;\bullet)}$. Since the *Partition* function (Algorithm 4) presented in the previous section runs in linear time with respect to $|E|/p$ and since the level sets of $D_{X_\bullet^\bullet}^{(\bullet;\bullet)}$ partitions $\mathbb{G}^\bullet$, the time complexity of line 10 is $O(|\mathbb{G}|/p)$. Furthermore, after the execution of line 10, we always have $|E_i| \leq |E|/p + 1$, for any $i \in \{1, \ldots, p\}$. Thus, the complexity of line 13 is $O(|\mathbb{G}|/p)$. From Lemma 10.2, we deduce that the time complexity of line 14 is $O(\frac{\beta(\mathbb{G})}{\beta(\mathbb{G})+p}|\mathbb{G}^\times| + \frac{1}{p}|\mathbb{G}^\bullet|)$ and, from Lemma 10.1, we deduce that the complexity of lines 15 and 16 is $O(\frac{\beta(\mathbb{G})}{\beta(\mathbb{G})+p}|\mathbb{G}^\times|)$. Let us finally analyze the complexity of line 19. By Lemma 10.1, the overall number of insertions in a set S_i is less than $\frac{4\beta(\mathbb{G})}{2\beta(\mathbb{G})+p+1}|\mathbb{G}^\times|$, for any i in $\{1, \ldots, p\}$. In the function *Union* (Algorithm 5), each element of S_i is copied once in E (line 3) by the processor numbered i. Thus, when *Union* is called by Algorithm 3, the overall complexity of line 3 in Algorithm 5 is $O(\frac{\beta(\mathbb{G})}{\beta(\mathbb{G})+p}|\mathbb{G}^\times|)$. Furthermore, since there is one call to *ParallelPrefixSum* for each call to *Union*, there is one call of *ParallelPrefixSum* for each level set of $D_{X_\bullet^\bullet}^{(\bullet;\bullet)}$. Therefore, if K is the number of level sets of $D_{X_\bullet^\bullet}^{(\bullet;\bullet)}$, the time complexity of line 19 is $O(\frac{\beta(\mathbb{G})}{\beta(\mathbb{G})+p}|\mathbb{G}^\times| + K \log_2 p)$ since the complexity of *ParallelPrefixSum* is $O(\log_2 p)$. It can be also seen that the worst-case time complexity of the initialization steps (lines 1 to 8) is $O(\frac{\beta(\mathbb{G})}{\beta(\mathbb{G})+p}|\mathbb{G}^\times| + \log_2 p)$. Hence, the following theorem can be stated.

Theorem 11 *Let p be a positive integer and let $X^\bullet$ be a p-regular subset of $\mathbb{G}^\bullet$. Then, Algorithm 3 terminates in $O(\frac{1}{p}|\mathbb{G}^\bullet| + \frac{\beta(\mathbb{G})}{p+\beta(\mathbb{G})}|\mathbb{G}^\times| + K \log_2 p)$ time, where K is the number of distinct level sets of $D_{X_\bullet^\bullet}^{(\bullet;\bullet)}$.*

Observe that, when the unbalancing factor of $\mathbb{G}$ is less than a small constant, the complexity of Algorithm 3 reduces to $O(\frac{|\mathbb{G}^\bullet|+|\mathbb{G}^\times|}{p} + K \log_2 p)$. Considering the complexity of the sequential distance map algorithm (Algorithm 1), the best possible theoretical complexity for a parallelization of Algorithm 1 with p processors would be $O(\frac{|\mathbb{G}^\bullet|+|\mathbb{G}^\times|}{p})$. This complexity is reached with Algorithm 3 (in the case of small unbalancing factor) up to a $K \log_2 p$ term, which is due to the computation of the parallel dynamic partition. On the other hand, it can be observed that, when the unbalancing factor of $\mathbb{G}$ is high compared to the number p of processors, the complexity of Algorithm 3 tends to $O(\frac{1}{p}|\mathbb{G}^\bullet| + |\mathbb{G}^\times| + K \log_2 p)$. Hence, in this case the complexity gain over the sequential algorithm is less interesting. Fortunately, in practical cases, the considered graphs have a small unbalancing factor. For instance, as mentioned before in this section, when an image is structured with the 4-adjacency relation, the unbalancing factor of the resulting graph is two (or even one if the image borders are identified). Hence, as confirmed in the following experimental sections, in practical cases, the proposed parallel algorithm offers an interesting speedup over the sequential version.

6 Target platform and implementation for experimental results

In this section, we first present a short overview of popular parallel-programming paradigms. Then, we describe the target platform (shared-memory multicore architecture) that is used for the experimental assessment of Algorithm 3 presented in Section 8 . We also present some details of the associated implementation of Algorithm 3 devised for this platform. In an effort to promote reproducible results in image processing, a C implementation of Algorithm 3 is available online at the following web address `perso.esiee.fr/~dpt-it/MorphoPar/ParDMaps.tgz`.

Multicore and multithreaded CPUs architectures have become an increasingly popular way to implement dynamic, highly asynchronous, concurrent programs. They both exploit concurrency by executing multiple threads at the more fine-grained instruction level. In fact, there are two dominant parallel programming paradigms for multicores and multithreaded CPUs environments : the shared memory and the distributed memory (Message passing). In distributed memory architecture, each processor has its own local memory where all variables of a program are private. While, in shared memory architecture every processor has access to all of the memory. The distributed memory paradigms assume that the computing infrastructure is composed of multiple nodes with distinct memory address spaces. Each compute node can only directly reference its own memory. The communication of data occurs through discrete messages sent from process to process. The Message Passing Interface (MPI) is the standard language for parallel programs on distributed memory systems. It is a specification for message passing operations which is provided as a library for C, C++ and Fortran [16]. It is based on explicit message passing and collective communications that have non-trivial consequences for performance.

The shared memory programming model is often achieved with some form of multithreading, where multiple threads on the same node share the same address space. Threads can easily communicate because they operate in the same address space, so explicit synchronization must be used to synchronize the simultaneously access on shared variables. OpenMP (Open Multi-Processing) and POSIX Threads (Pthreads) are two of the most widely used solutions for shared memory programming.

OpenMP consists of a set of compiler directives. It is a directive-based extension to C, C++ and Fortran languages commonly used in high performance computing. Moreover, OpenMP is becoming more important when the number of cores per system increases. Due to the higher level specification of parallelism with OpenMP, the gain in productivity is that OpenMP programmers do not have to concern themselves with such details as when or how to create new threads, how to distribute their data between these threads and how to synchronize the computation itself between the threads [16, 13].

The POSIX Threads is a set of C programming language types and procedure calls. Various benefits exist to using Pthreads such as the simplicity, the flexibility, and the portability. Pthreads library provides various primitives for synchronizing concurrent computations and memory accesses of threads. In particular, it provides implementations of mutexes and semaphores [3] for Linux. A mutex is a variable which is either locked or unlocked

and can be handled with atomic (non-interruptible) instructions, ensuring that, at a given time, a single thread is accessing to the variable with the atomic instruction. It is mainly used to ensure exclusive access to data shared between threads, whereas semaphores generalize mutexes by allowing a limited number, but possibly greater than one, to access concurrently to critical data [16, 13].

Today, most modern computer systems include multicore chips that support shared memory in hardware. Therefore, parallel computing on shared memory multicores architecture is now very popular. Because of the high availability of this architecture, the large amount of available computing paradigms for these architectures and the parallelism in our proposed strategy, we decided to carry out experiments and developments of Algorithm 3 on a two quadcore Intel(R) Xeon(R) E5630 processors. With such multicore-multithreaded shared memory architecture, up to 8 threads can run in parallel, emulating the p processors (with $p \leq 8$) of the theoretical parallel model used in the previous section.

In this article, a multithreaded implementation of Algorithm 3 is devised for the above described target architecture. POSIX Threads (Pthreads) library on a Linux system is considered for this implementation since it allows to handle low-level synchronization between threads. We remind that such synchronization is required to implement the proposed parallelization strategy which controls precisely the load balancing between the available processors. It can also be noticed that the use of MPI is not adapted to our target architecture (shared memory) and that the use of OpenMP does not allow one to control precisely the load balancing between processors as proposed in Algorithm 3.

Despite the distribution of data in Algorithm 3, each thread has access to an overlap data (the Boolean array $Traversed$) in order to check if a vertex has been already traversed or not. The only access to this shared memory is done with the *test-and-set* instruction at line 16 of Algorithm 3. As mentioned in Section 5, this instruction involves a synchronization between processors or threads. The *test-and-set* instruction can be implemented with Pthreads library using a mutex variable. To this end, one must consider a mutex variable instead of a Boolean one. Then, the *test-and-set* instruction is implemented due to the Pthreads function *pthread_ mutex_ trylock*. This is a non-interruptible function that locks the mutex and returns 0 if the mutex is unlocked at the time of the call or that returns a positive value otherwise (*i.e.*, if the mutex is already locked when the function is called). Thus, the implementation of the *test-and-set* instruction is as follows:

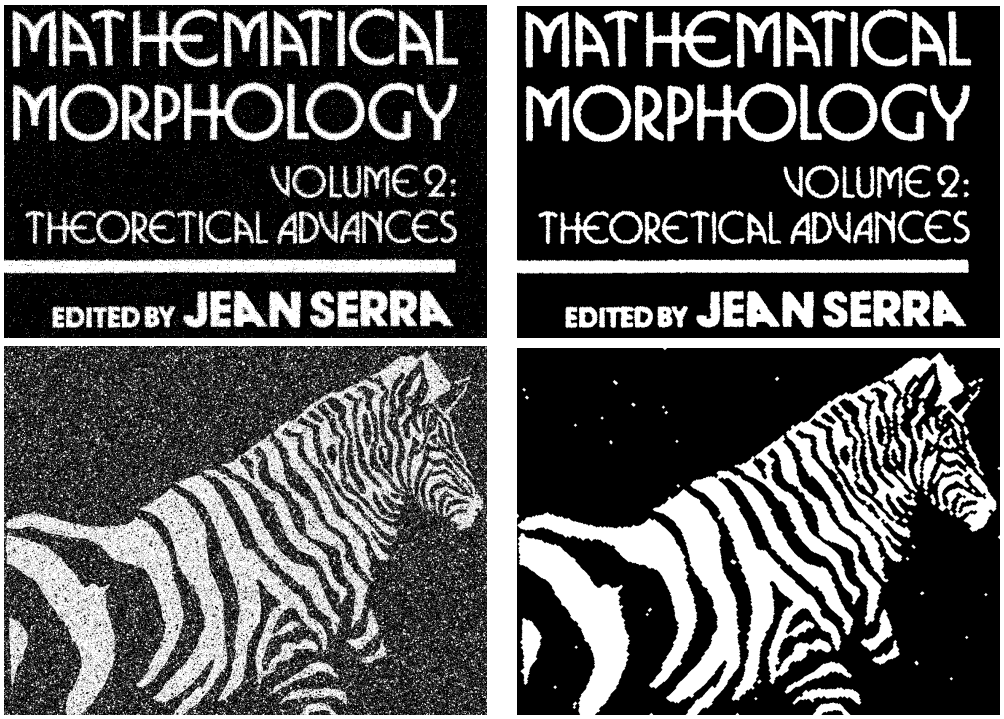
```
Boolean test-and-set(pthread_mutex_t m){
     if(pthread_mutex_trylock(m) == 0) return False;
     else return True;
}
```

With this test-and-set implementation, the Boolean array *Traversed* of Algorithm 3 is implemented with an array of mutexes that are initially unlocked.

7 Experimental-assesment dataset and regularity-assumption assesment

This section describes the image and textured-mesh datasets used for experimental evaluations. In particular, the cardinalities of the level-sets of the considered distance maps are analyzed, allowing us for assessing the regularity assumption of Theorem 11 on these experimental datasets.

For experimental evaluations, two image datasets and one textured mesh dataset are considered. The first image dataset is the one of [9]. It is a set of 33 binary images of different size (786 x 540, 800 x 600, 700 x 1285) and various contents (see 2 samples in Figure 10.a). The images of this dataset have been obtained by adding black and white random noise of different sizes and shapes to bitmap images of simple shapes (mainly letters). We also consider the *Standard* dataset which is a set of 12 images found frequently in the literature such as Lena, peppers, cameraman, lake, *etc.* All images of this dataset are grayscale and of the same size (512 x 512) (see 2 examples in Figure 11(a)). Finally, a dataset of 3 three-dimensional triangular meshes is also considered.



(a) initial binary images. (b) results of graph ASF.

Fig. 10 Illustrations of mathematical morphology ASF on graphs on two images of the dataset used in [9].

Each mesh is equipped with two binary black and white textures, one being obtained by adding random noise to another. Hence, overall, this dataset is made of 6 binary-textured meshes (see, *e.g.*, Figure 12).

Elementary mathematical morphology operators are often considered as a post-processing step after the binarization of images since it allows to reduce noise and regularize contours. Therefore, we apply Otsu thresholding method [19] to binarize the images of the standard dataset (see Figure 11(b)). Then, after this step, two databases of typical binary images are obtained. These binary images are processed with the mathematical morphological operators described in Section 2 (see examples of results in Figure 10(b) and Figure 11(c)). In order to filter an image with the operators studied in this article, the image domain is equipped with a graph. For each image, the vertex set of the considered graphs is made of the image pixels and the edge set is given by the well known 4-adjacency relation on these pixels. For each of these two image datasets, Table 2 provides the image sizes, the number of images of each size, the average number of white pixels (*i.e.*, the vertices $X^\bullet$), and the average number of level-sets in the considered vertex-vertex distance maps. In order to filter textured meshes with operators on graphs (see Figure 12), texture domains are equipped with graphs. More precisely, given a mesh, each corner of each triangle is a vertex of the associated graph and two vertices of the graph are linked by an edge if the associated triangle corners belong to a same triangle or if the associated triangle corners are at the same spatial position and belong to two adjacent triangles. Table 3 provides the number of vertices and of edges of the considered graphs, the average number of white vertices in the textures, and the average number of level-sets in the vertex-vertex distance maps associated with the textures.

Let us now analyze the datasets described above in order to assess the assumptions of Theorem 11. Indeed, Theorem 11 asserts that if i) the unbalancing factor of the graph $\mathbb{G}$ is lower than a small constant and if ii) the

(a) grayscale images (b) results of Otsu binarization (c) results of graph ASF

Fig. 11 Illustration of Otsu binarization followed by mathematical morphology filtering on graphs from two images of the *Standard* image dataset.

Image DSs:	Standard DS	DS of [9]		
Images size	512 × 512	786 × 540	800 × 600	700 × 1285
Number of images	12	2	29	2
Avg. number of vertices in $X^\bullet$	17 090	321 499	372 044	663 259
Avg. number of level-sets of $D_{X^\bullet}^{(\bullet,\bullet)}$	66	4	11	6

Table 2 Main features of the image datasets used for experimental evaluation. In the table, DS stands for dataset.

considered set of vertices is p-regular, then the time complexity of Algorithm 3 is $O((|\mathbb{G}^\bullet|+|G^\times|)/p+K\log_2 p)$, where K is the number of nonempty level-sets of the distance maps and where p is the number of available processors. Algorithm 3 is applied to sets of pixels of 2D binary images equipped with the 4-adjacency relation and to subsets of vertices of graphs associated to binary textured map. As mentioned in Section 5.4, the unbalancing factor of the graphs associated to the images is 2. Furthermore, by construction, the unbalancing factor of the graph associated to the meshes is 1 (each vertex is exactly linked to four other vertices). Thus, assumption i) is practically verified for the considered datasets. In order to assess assumption ii), we analyze the number of vertices in each level-set of the considered distance maps and compare these numbers with the numbers of processors used in the experiments presented in Section 8. Due to the chosen target architecture, this number is always lower than 8. For each dataset, the proportion of level-sets with less than k vertices is estimated for $k \in \{2, 4, 6, 8\}$.

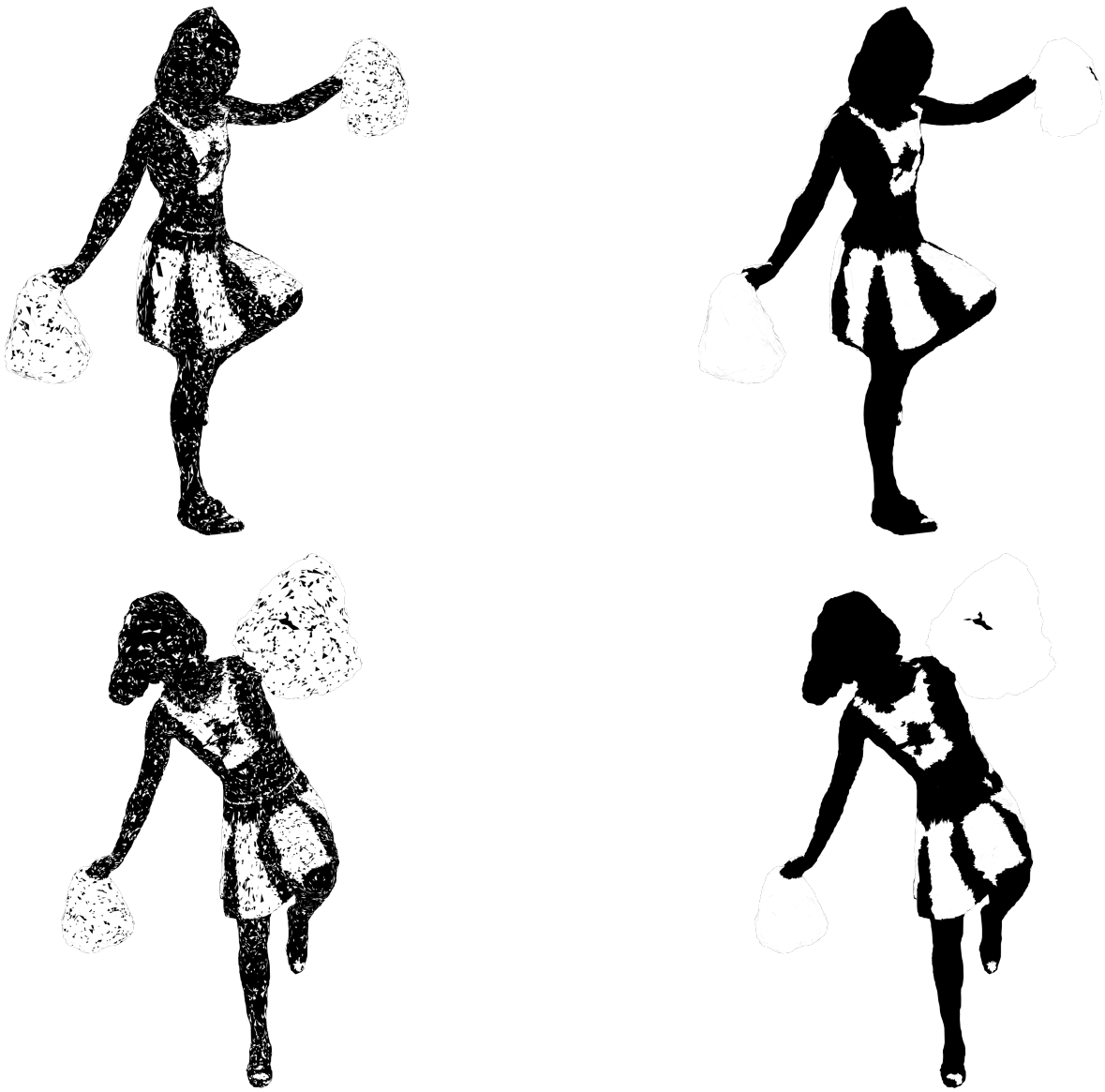


Fig. 12 Mathematical morphology filtering of a texture defined over a 3 dimensional mesh; (left) two renderings of a 3 dimensional mesh M equipped with a binary texture T; and (right) two renderings of the ASF-filtered version of the texture T on a graph associated with the mesh M.

The result of this analysis is presented in Table 4. To perform this analysis a total of 797 (resp. 716 and 187) level-sets are analyzed for the 12 (resp. 33 and 6) images of the standard dataset (resp. the dataset of [9] and the binary-textured mesh dataset). Observe that the average number of level set is significantly less for the dataset of [9] than for the standard dataset. This can be explained by the addition of random black and white noise in the later images which implies that every image pixel is close to white pixel and to a black one. For the standard

Number of textured meshes	6
Number of vertices/edges per graph	98 226 / 392 904
Average number of white vertices (vertices in $X^\bullet$)	45 102
Average number of level-sets in $D^\bullet_{X^\bullet}$	31

Table 3 Main features of the binary-textured mesh dataset used for experimental evaluations.

dataset, 3 level-sets contain less than 2 vertices (0.37%), 10 level-sets contain less than 4 vertices (1.25%), 16 level-sets contain less than 6 vertices (2 %), and 23 level-sets contain less than 8 vertices (2.88%). For the dataset of [9], 6 level-sets contain less than 2 vertices (0.83%), 11 level-sets contain less than 4 vertices (1.53%), 18 level-sets contain with less than 6 vertices (2.51%) and 20 level-sets contain less than 8 vertices (2.79%). Finally, for the binary-textured mesh dataset, no level-set contains less than 2 vertices (0%), 3 level-sets contain less than 4 vertices (1.6%), 5 level-sets contain with less than 6 vertices and less than 8 vertices (2.67%). Hence, the number and the proportion of level-sets with less than $k \in \{2, 4, 6, 8\}$ vertices is relatively small. It is also observed that the level-sets with few vertices tend to appear for the highest distance values. Hence, this analysis confirms that the assumptions in Theorem 11 are reasonable for the considered experiments.

k	2	4	6	8
Standard dataset	0.37%	1.25%	2%	2.88%
Image dataset used in [9]	0.83%	1.53%	2.51%	2.79%
Binary-textured mesh dataset	0%	1.6%	2.67%	2.67%

Table 4 Proportion of level-sets of the vertex-vertex distance maps with less than k vertices, the total number of analyzed level-sets being equal to 797 (resp. 716 and 187) for the 12 images (resp. 33 images and 6 textured meshes) of the standard dataset (resp. the dataset of [9] and the textured mesh dataset).

8 Performance evaluation and experimental results

This section analyzes the results obtained by applying the implementations of the proposed sequential and parallel algorithms on the target architecture (Section 6) to the datasets presented in Section 7. Section 8.1 first describes the measures assessing the performance of the proposed implementations (execution times and speedups). Then, Section 8.2 presents the evaluation results.

8.1 Performance Evaluation

The performance of the proposed sequential and parallel algorithms (Sections 3 and 5) is assessed through the execution times obtained with the implementation and target platform described in Section 6. For assessing the gain of the parallel implementation over the sequential one, the speedup measure is considered.

The reported execution times are obtained with Gprof, a profiling software available on GNU Linux. Gprof measures the time spent on each function for a particular execution of a program as well as the number of times each function is called. Each measurement reported in Section 8.2 is the average of the executions times of forty runs of the same program on the same data.

Speedup is an important measure of the quality of a parallel program. This measure expresses how many times a parallel program works faster than a sequential one, where both programs are solving the same problem.

For a given number p of processors, the speedup $S(p)$ is the ratio of the execution time $T(1)$ of a program on a single processor over the execution time $T(p)$ obtained with p processors: $S(P) = T(1)/T(p)$.

For p processors, the speedup is bounded by the the number of processors: $S(p) \leq p$. The maximum value of speedup that can be obtained is equal to the number of processors $S(p) = p$. This maximum speedup value could be achieved in an ideal multiprocessor system where there are no communication costs and the workload of processors is balanced (although in practice this is rarely achieved). In such a system, every processor needs $T(1)/p$ time units in order to complete its job [1].

One can consider speedup to analyze algorithms either theoretically using asymptotic time complexity or in practice by measuring the execution times of a program, as presented in the following section.

8.2 Experimental results

The results presented in this section are obtained by the application of the proposed sequential and parallel algorithms (namely, Algorithms 1 and 3) on the datasets presented in Section 7. We tested our implementations on two different types of sets: the sets of white vertices and their complements, which are the sets of black vertices. The distance map to a set of white vertices is used to obtain the dilations of this set whereas the distance map to its complementary set is used to obtain the erosions of the set (see Property 4 and Theorem 6).

Tables 5 and 6 and Figures 13 and 14 show the average execution times of the sequential and of the parallel implementations on 1, 2, 4, 6, and 8 cores for the different datasets. Clearly, as expected from the theoretical analysis of Algorithm 3 in Section 5.4, the obtained running times are decreasing with the number of cores, whatever the considered dataset.

		Execution times (in seconds)							
		original images				complemented images			
Images size		512 × 512	786 × 540	800 × 600	700 × 1285	512 × 512	786 × 540	800 × 600	700 × 1285
Sequential algorithm		0.026	0.055	0.057	0.060	0.035	0.04	0.044	0.07
Parallel algorithm	1 core	0.025	0.053	0.055	0.065	0.032	0.039	0.042	0.072
	2 cores	0.020	0.040	0.042	0.050	0.024	0.035	0.032	0.058
	4 cores	0.016	0.033	0.035	0.040	0.020	0.028	0.020	0.041
	6 cores	0.012	0.023	0.025	0.031	0.017	0.020	0.012	0.035
	8 cores	0.010	0.014	0.016	0.015	0.013	0.011	0.008	0.018

Table 5 Average execution times of the vertex-vertex distance map algorithms for the image datasets.

The speedup values achieved by the parallel implementation of the distance map algorithm are presented in Tables 7 and 8. We observe that, in all the cases, a good speedup over the sequential implementation is reached. Figures 15 and 14 plot the obtained speedups when the number of threads varies from 2 to 8. On the image datasets (resp. textured-mesh dataset) the best speedup value, obtained with 8 cores, is equal to 5.5 (resp. 5.33). These values are obtained for the set of black pixels of the images of size 800 × 600 and for the set of black vertices of the textured meshes, respectively.

		Execution times (in seconds)	
		original textures	complemented textures
Sequential algorithm		0.015	0.016
Parallel algorithm	1 core	0.014	0.015
	2 cores	0.009	0.010
	4 cores	0.006	0.007
	6 cores	0.004	0.004
	8 cores	0.003	0.003

Table 6 Average execution times of the vertex-vertex distance map algorithms for the textured-mesh dataset.

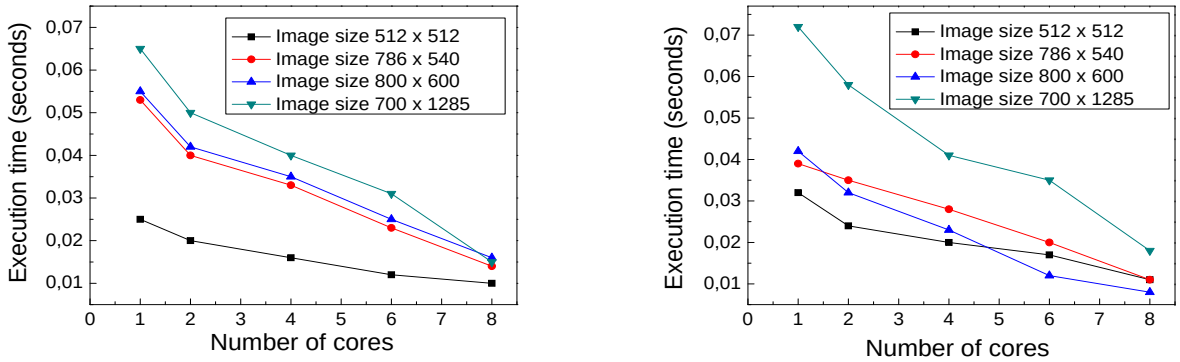
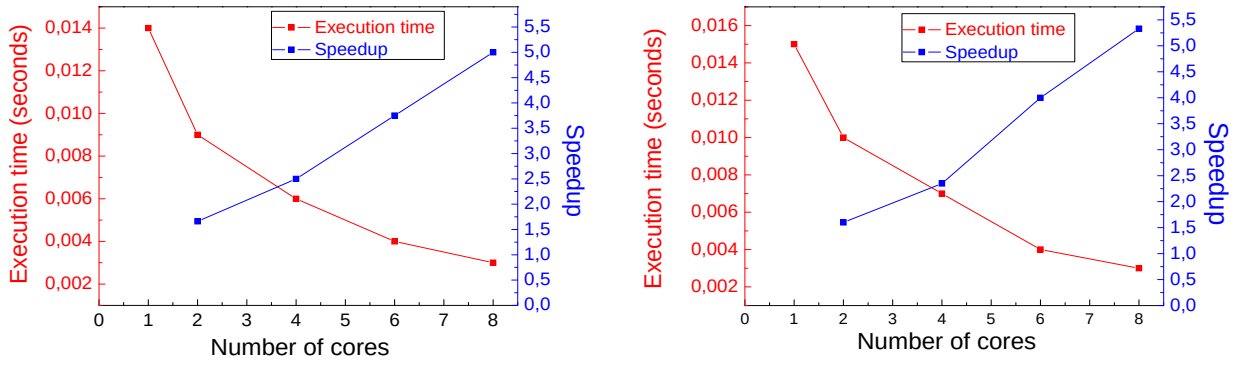


Fig. 13 Average execution times, for the image datasets, of the parallel vertex-vertex distance map algorithm, plotted as a function of the number of cores.

Furthermore, we tried to vary the number of threads from 1 to 30 on the 8 cores platform for the most favorable cases (*i.e.*, distance maps to sets of black pixels from the images of size 800 × 600). The results are presented in Figure 16 where it can be seen that for each number of threads the running time of the parallel algorithm decreases until 8 threads. Then, for each number of thread greater than 8, the running time increases. Therefore, we can deduce that the best performance (running time and speedup) is achieved when the number of thread is not greater than the number of cores(see Figure 16).

The previous experiments was designed to asses the gain of the parallel distance map implementation over the sequential one for processing binary 2d images and textured meshes. Let us now assess the results obtained using distance maps to compute the results of the morphological operators presented in Definition 2. For the sake of simplicity, we only consider the case of even size parameter λ. We recall that, due to the properties presented in Section 3, the results of the dilations $\delta_{\lambda/2}$ and erosions $\epsilon_{\lambda/2}$ for any size parameter λ can be computed with a single iteration by thresholding a distance map instead of applying λ iterations of elementary dilation as one would do by straightforwardly applying the definition. However, such straightforward implementation, was, as far as we know, up to the present article, the only available implementation of these operators.



(original textures) (complemented textures)

Fig. 14 Average execution times and speedups, on the textured-mesh dataset, of the parallel vertex-vertex distance map algorithm, plotted as a function of the number of cores.

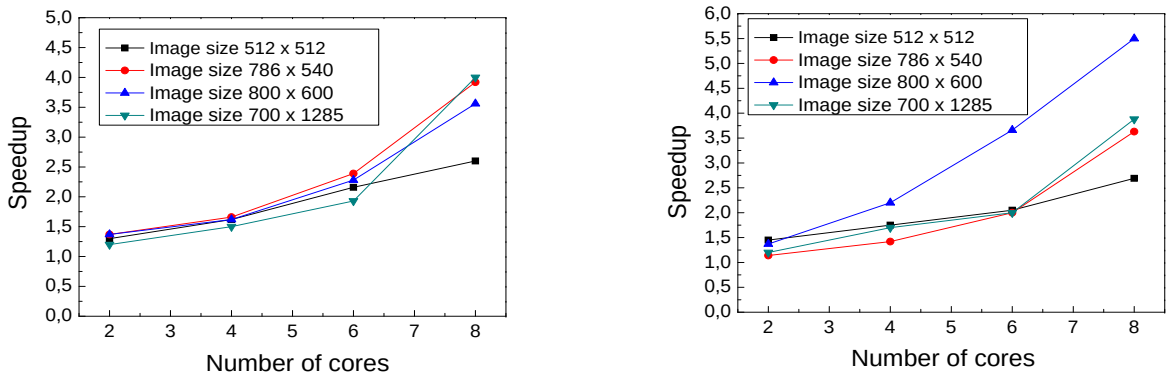
	Speedup							
	original images				complemented images			
Images size	2 cores	4 cores	6 cores	8 cores	2 cores	4 cores	6 cores	8 cores
512 × 512	1.3	1.62	2.16	2.6	1.45	1.75	2.05	2.69
786 × 540	1.37	1.66	2.39	3.92	1.14	1.42	2	3.63
800 × 600	1.35	1.62	2.28	3.56	1.37	2.2	3.66	5.5
700 × 1285	1.2	1.5	1.93	4	1.20	1.70	2	3.88

Table 7 Average speedups of the parallel vertex-vertex distance map algorithm on the image datasets.

	Speedup							
	original textures				complemented textures			
	2 cores	4 cores	6 cores	8 cores	2 cores	4 cores	6 cores	8 cores
Speedup	1.66	2.5	3.75	5	1.6	2.28	4	5.33

Table 8 Average speedups of the parallel vertex-vertex distance map algorithm on the textured-mesh dataset.

We compare the execution times of the parallel dilation implementation based on distance map on 8 cores with the sequential version also based on distance map and with the straightforward iterative implementation. Figure 17 displays the resulting execution times. It can be seen that the running times of the implementations based on distance maps remain constant while the size parameter increases whereas, with the straightforward implementation, the execution times increase linearly with the size parameter. In particular, when $\lambda = 2$, the sequential implementation based on distance maps and the straightforward implementation runs in about the



(original images) (complemented images)

Fig. 15 Average speedups, on the image datasets, of the parallel vertex-vertex distance map algorithm, plotted as a function of the number of cores.

same time (0.060 sec) but when λ is equal to 24 the implementation based on distance map is more than 10 times faster than the straightforward implementation. Observe also that, due to the speedup of 5.5 achieved by the parallel implementation for 8 cores machine, the result of the dilation of size 2/2 is obtained 5.5 times faster with the parallel implementation than with the straightforward implementation and more than 55 times faster when the size parameter is equal to 24/2.

9 Conclusion

This article proposes efficient sequential and parallel algorithms for the (binary) mathematical morphology operators on graphs defined in [9]. To reach this goal, new notions of distance maps in unweighted graphs are investigated and characterizations of the considered operators in terms of distance maps are provided (Section 3). Then, efficient sequential and parallel algorithms for distance maps are proposed in Sections 4 and 5, respectively. The sequential algorithms run in linear time with respect to the size of the graph whereas the parallel algorithm runs, for practical cases, in $O(n/p + K \log_2 p)$ time complexity, where n, p and k are the size of the graph, the number of available processors, and the number of distinct level sets of the distance map, respectively. Based on this study, a parallel implementation on shared memory multicore/multithreaded architecture is proposed (Section 6) and assessed on datasets of images and textured meshes (Sections 7 and 8). In terms of execution times, the sequential implementation based on the proposed sequential distance map algorithm shows a significant improvement over the previous existing implementation of the operators of [9] since it runs about ten times faster on the tested images for a dilation size of 24/2. Furthermore, the parallel implementation based on distance maps yields another significant improvement over the sequential one: a speedup factors up to 5.5 are achieved for 8 cores machine, leading to execution times 55 times smaller than with the previous existing implementation of the operators of [9].

This article studies the case of elementary mathematical morphology operators on graphs for processing binary data (*i.e.* subsets of vertices or of edges of a graph). Even if the definition of these operators can be straightforwardly extended to the case of grayscale data (*i.e.*, grayscale functions on the vertices or on the edges), an efficient extension of the proposed algorithms to grayscale case is not straightforward and is an interesting topic for future research. On the other hand, the use of distance maps in unweighted graphs opens doors towards

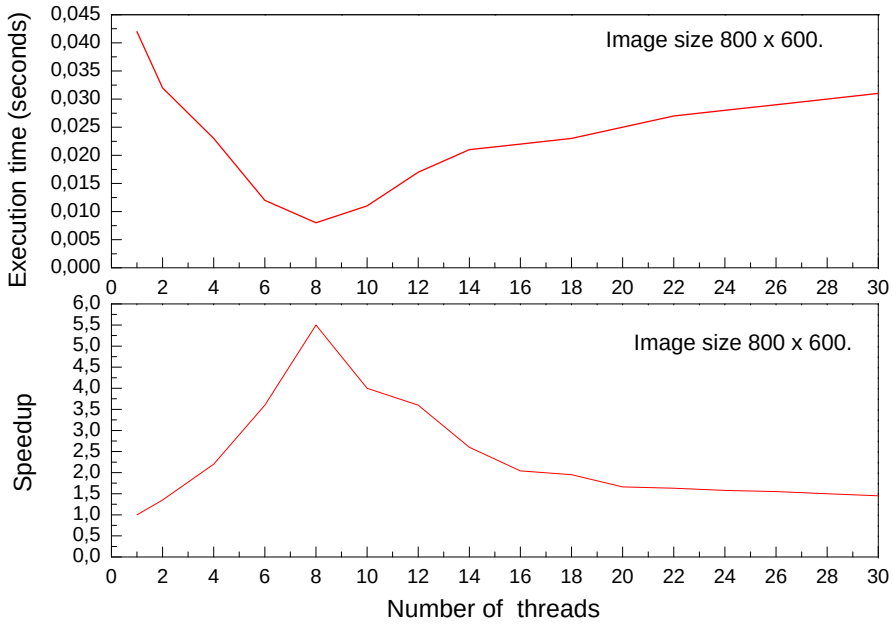


Fig. 16 Timing and speedup of parallel vertex-vertex distance map obtained with various number of threads on 8 cores (considering complement images size 800 × 600).

the investigation of morphological operators on graphs embedded in metric spaces (or more generally on weighted graphs) where the result of an operator depends on the "length" of the edges according to the metric. Such study could lead to significant quality improvements for the results of mathematical morphological operators on graphs. A last perspective of our work, on the computer architecture side, is performance optimization based on a better understanding of the effects of multithreading on cache memories and based on the use of massive parallelism available in the graphic programming architecture GPU. The later optimization could require new variant around the parallel strategy presented in this article.

Acknowledgments

We warmly thank the consortium of the RECOVER3D project funded by the French 1st *Programme d'Investissements d'Avenir (PIA)* including, in particular, Ludovic Blache and Laurent Lucas, for providing us with the textured meshes processed in the work presented in this article.

References

1. Alecu, F.: Performance analysis of parallel algorithms. Journal of Applied Quantitative Methods **129** (2007)
2. Bloch, I., Bretto, A.: Mathematical morphology on hypergraphs, application to similarity and positive kernel. Computer vision and image understanding **117**(4), 342–354 (2013)

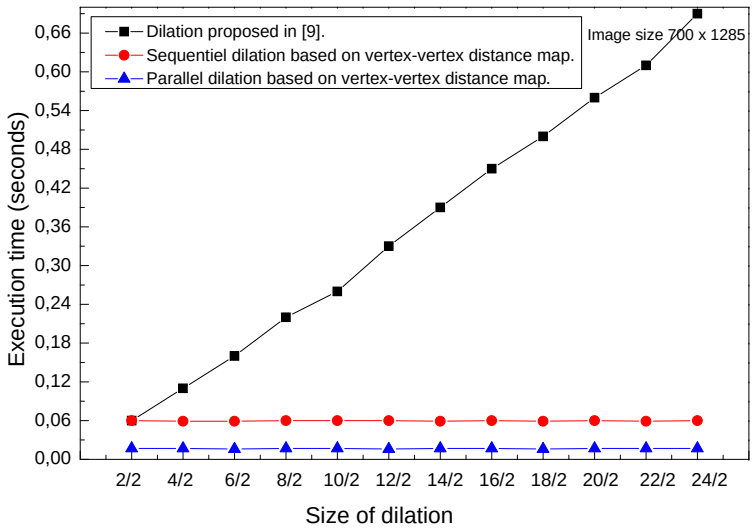


Fig. 17 Experimental evaluation of iterated dilations $\delta_{\lambda/2}$ when λ is even.

3. Butenhof, D.R.: Programming with POSIX threads. Addison-Wesley Professional (1997)
4. Chia, T.L., Wang, K.B., Chen, Z., Lou, D.C.: Parallel distance transforms on a linear array architecture. IPL **82**(2), 73–81 (2002)
5. Coeurjolly, D.: 2d subquadratic separable distance transformation for path-based norms. In: Discrete Geometry for Computer Imagery, pp. 75–87. Springer (2014)
6. Cormen, T.H.: Introduction to algorithms. MIT press (2009)
7. Couprie, M., Bertrand, G.: New characterizations of simple points in 2d, 3d, and 4d discrete spaces. IEEE Transactions on Pattern Analysis and Machine Intelligence **31**(4), 637–648 (2009)
8. Cousty, J., Bertrand, G., Couprie, M., Najman, L.: Collapses and watersheds in pseudomanifolds of arbitrary dimension. Journal of mathematical imaging and vision **50**(3), 261–285 (2014)
9. Cousty, J., Najman, L., Dias, F., Serra, J.: Morphological filtering on graphs. CVIU **117**(4), 370–385 (2013)
10. Cousty, J., Najman, L., Serra, J.: Some morphological operators in graph spaces. In: Mathematical Morphology and Its Application to Signal and Image Processing, pp. 149–160. Springer (2009)
11. Delgado-Friedrichs, O., Robins, V., Sheppard, A.: Skeletonization and partitioning of digital images using discrete morse theory. IEEE transactions on pattern analysis and machine intelligence **37**(3), 654–666 (2015)
12. Dias, F., Cousty, J., Najman, L.: Dimensional operators for mathematical morphology on simplicial complexes. Pattern Recognition Letters **47**, 111–119 (2014)
13. Grama, A.: Introduction to parallel computing. Pearson Education (2003)
14. Heijmans, H.J.: Morphological image operators. Advances in Electronics and Electron Physics Suppl., Boston: Academic Press,| c1994 **1** (1994)
15. Heijmans, H.J., Ronse, C.: The algebraic basis of mathematical morphology i. dilations and erosions. Computer Vision, Graphics, and Image Processing **50**(3), 245–295 (1990)
16. Kasim, H., March, V., Zhang, R., See, S.: Survey on parallel programming model. In: Network and Parallel Computing, pp. 266–275. Springer (2008)
17. Ladner, R.E., Fischer, M.J.: Parallel prefix computation. JACM **27**(4), 831–838 (1980)
18. Lerallut, R., Decencière, É., Meyer, F.: Image filtering using morphological amoebas. Image and Vision Computing **25**(4), 395–404 (2007)
19. Level Otsu, N.: A threshold selection method from gray-level histogram. IEEE Transactions on Systems, Man and Cybernetics **9**(1), 62–66 (1979)
20. Man, D., Uda, K., Ueyama, H., Ito, Y., Nakano, K.: Implementations of parallel computation of Euclidean distance map in multicore processors and GPUs. In: ICNC, pp. 120–127 (2010)

21. Man, D., Uda, K., Ueyama, H., Ito, Y., Nakano, K.: Implementations of a parallel algorithm for computing euclidean distance map in multicore processors and gpus. International journal of networking and computing **1**(2), 260–276 (2011)
22. Mennillo, L., Cousty, J., Najman, L.: A comparison of some morphological filters for improving ocr performance. In: Mathematical Morphology and Its Applications to Signal and Image Processing, pp. 134–145. Springer International Publishing (2015)
23. Meyer, F., Angulo, J.: Micro-viscous morphological operators. Mathematical Morphology and its Application to Signal and Image Processing (ISMM 2007) pp. 165–176 (2007)
24. Najman, L., Cousty, J.: A graph-based mathematical morphology reader. Pattern Recognition Letters **47**, 3–17 (2014)
25. Pham, T.Q.: Parallel implementation of geodesic distance transform with application in superpixel segmentation. In: DICTA, pp. 1–8. IEEE (2013)
26. Ronse, C., Serra, J.: Algebraic foundations of morphology. Mathematical Morphology: from theory to applications pp. 35–80 (2013)
27. Rosenfeld, A., Pfaltz, J.L.: Distance functions on digital pictures. PR **1**(1), 33–61 (1968)
28. Saito, T., Toriwaki, J.I.: New algorithms for euclidean distance transformation of an n-dimensional digitized picture with applications. Pattern recognition **27**(11), 1551–1565 (1994)
29. Serra, J.: Image analysis and mathematical morphology, v. 1. Academic press (1982)
30. Shyu, S.J., Chou, T., Chia, T.L.: Distance transformation in parallel. In: Proc. Workshop Combinatorial Math. and Computation Theory, pp. 298–304 (2006)
31. Soille, P., Breen, E.J., Jones, R.: Recursive implementation of erosions and dilations along discrete lines at arbitrary angles. PAMI **18**(5), 562–567 (1996)
32. Svolos, A.I., Konstantopoulos, C.G., Kaklamanis, C.: Efficient binary morphological algorithms on a massively parallel processor. In: IPDPS, p. 281 (2000)
33. Vincent, L.: Graphs and mathematical morphology. Sig. Proc. **16**(4), 365–388 (1989)
34. Youkana, I., Cousty, J., Saouli, R., Akil, M.: Parallelization strategy for elementary morphological operators on graphs. In: International Conference on Discrete Geometry for Computer Imagery, pp. 311–322. Springer (2016)