Implementations of Parallel Computation of Euclidean Distance Map in Multicore Processors and GPUs

Duhu Man, Kenji Uda, Hironobu Ueyama, Yasuaki Ito, Koji Nakano
Department of Information Engineering,
Hiroshima University
1-4-1 Kagamiyama, Higashi Hiroshima, 739–8527 Japan

***Abstract*—Given a 2-D binary image of size $n \times n$, Euclidean Distance Map (EDM) is a 2-D array of the same size such that each element is storing the Euclidean distance to the nearest black pixel. It is known that a sequential algorithm can compute the EDM in $O(n^2)$ and thus this algorithm is optimal. Also, work-time optimal parallel algorithms for shared memory model have been presented. However, these algorithms are too complicated to implement in existing shared memory parallel machines. The main contribution of this paper is to develop a simple parallel algorithm for the EDM and implement it in two parallel platforms: multicore processors and a Graphics Processing Unit (GPU). More specifically, we have implemented our parallel algorithm in a Linux server with four Intel hexad-core processors (Intel Xeon X7460 2.66GHz). We have also implemented it in a modern GPU system, Tesla C1060, respectively. The experimental results have shown that, for an input binary image with size of 10000×10000, our implementation in the multi-core system achieves a speedup factor of 18 over the performance of a sequential algorithm using a single processor in the same system. Meanwhile, for the same input binary image, our implementation on the GPU achieves a speedup factor of 5 over the sequential algorithm implementation.**

***Keywords*-Euclidean Distance Map, Proximate Points, Multicore Processors, GPU**

I. Introduction

In many applications of image processing such as blurring effects, skeletonizing and matching, it is essential to measure distances between featured pixels and nonfeatured pixels. For a 2-D binary image with size of $n \times n$, treating black pixels as featured pixels, Euclidean Distance Map (EDM) assigns each pixel with the distance to the nearest black pixel using Euclidean distance as underlying distance metric. We refer reader to Figure 1 for an illustration of Euclidean Distance Map. Assuming that points p and q of the plane are represented by their Cartesian coordinates $(x(p), y(p))$ and $(x(q), y(q))$, as usual, we denote the Euclidean distance between the points p and q by $d(p, q) = \sqrt{(x(p) - x(q))^2 + (y(p) - y(q))^2}$.

Many algorithms for computing EDM have been proposed in the past. Breu *et al.* [1] and Chen *et al.* [2], [3] have presented $O(n^2)$-time sequential algorithm for computing Euclidean Distance Map. Since all pixels must be read at least once, these sequential algorithms with time complexity

$\sqrt{10}$	3	$\sqrt{10}$	$\sqrt{5}$	$\sqrt{2}$	1	$\sqrt{2}$	$\sqrt{5}$	3	$\sqrt{10}$
$\sqrt{5}$	2	$\sqrt{5}$	2	1	0	1	2	2	$\sqrt{5}$
$\sqrt{2}$	1	$\sqrt{2}$	$\sqrt{5}$	$\sqrt{2}$	1	$\sqrt{2}$	$\sqrt{2}$	1	$\sqrt{2}$
1	0	1	2	2	2	2	1	0	1
$\sqrt{2}$	1	$\sqrt{2}$	$\sqrt{2}$	1	$\sqrt{2}$	$\sqrt{5}$	$\sqrt{2}$	1	$\sqrt{2}$
$\sqrt{5}$	2	2	1	0	1	2	$\sqrt{5}$	2	$\sqrt{5}$
$\sqrt{10}$	$\sqrt{8}$	$\sqrt{5}$	$\sqrt{2}$	1	$\sqrt{2}$	$\sqrt{5}$	3	3	$\sqrt{10}$
$\sqrt{10}$	$\sqrt{5}$	$\sqrt{2}$	1	$\sqrt{2}$	$\sqrt{5}$	$\sqrt{5}$	2	$\sqrt{5}$	$\sqrt{8}$
3	2	1	0	1	2	$\sqrt{2}$	1	$\sqrt{2}$	$\sqrt{5}$
$\sqrt{10}$	$\sqrt{5}$	$\sqrt{2}$	1	$\sqrt{2}$	2	1	0	1	2

Figure 1. Illustrating Euclidean Distance Map

of $O(n^2)$ is optimal. Since in any EDM algorithm, each of the n^2 pixels has to be scanned at least once. Roughly at the same time, Hirata [4] presented a simpler $O(n^2)$-time sequential algorithm to compute the distance map for various distance metrics including Euclidean, four-neighbor, eight-neighbor, chamfer, and octagonal. On the other hand, for accelerating sequential ones, numerous parallel EDM algorithms have been developed for various parallel model. Lee *et al.* [5] presented an $O(\log^2 n)$-time algorithm using n^2 processors on the EREW PRAM. Pavel and Akl [6] presented an algorithm running in $O(\log n)$ time and using n^2 processors on the EREW PRAM. Clearly, these two algorithms are not work-optimal. Fujiwara *et al.* [7] have presented a work-optimal algorithm running in $O(\log n)$ time using $\frac{n^2}{\log n}$ $EREW$ processors and in $O(\frac{\log n}{\log \log n})$ time using $\frac{n^2 \log \log n}{\log n}$ $CRCW$ processors. Later, Hayashi *et al.* [8] have exhibited a more efficient algorithm running in $O(\log n)$ time using $\frac{n^2}{\log n}$ processors on the EREW PRAM and in $O(\log \log n)$ time using $\frac{n^2}{\log \log n}$ processors on the CRCW PRAM. Since the product of the computing time and the number of processors is $O(n^2)$ these algorithms are work optimal. Also, it was proved that the computing time cannot be improved as long as work optimality is satisfied, these algorithms are also work optimal. Thus, these algorithms are work-time optimal. Recently, Chen *et al.* [9] have proposed two parallel algorithms for EDM on Linear Array with Reconfigurable Pipeline Bus System [10]. Their

978-0-7695-4277-5/10 $26.00 © 2010 IEEE
DOI 10.1109/IC-NC.2010.55

first algorithm can computes EDM in $O(\frac{\log n \log\log n}{\log\log\log n})$ time using n^2 processors and second algorithm can compute EDM in $O(\log n \log\log n)$ time using $\frac{n^2}{\log\log n}$ processors.

In practice, now many applications has employed both general Multi-Core Processors and emerging GPUs (Graphics Processing Unit) as real platforms to achieve an efficient acceleration. We have also implemented and evaluated our parallel EDM algorithm in the both platforms, a Linux server with four Intel hexad-core processors (Intel Xeon X7460 2.66GHz [11]) and a modern GPU (Graphics Processing Unit) system, Tesla C1060 [12], respectively. The experimental results show that, for an input binary image with size of 10000×10000, our parallel algorithm can achieve 18 times speedup in the multi-core system over the performance of a sequential algorithm. Further, for the same input image, our parallel algorithm for the GPU system achieves a speedup factor of 5.

The remainder of this paper is organized as follows: Section II introduces the proximate points problem for Euclidean distance metric and discuss several technicalities that will be crucial ingredients to our subsequent parallel EDM algorithm. Section III shows the proposed parallel algorithm for computing Euclidean distance map of a 2-D binary image. Section IV exhibits the performance of our proposed algorithm on various multi-core platforms. Finally, Section V offers concluding remarks.

II. Proximate Points Problem

In this section, we review the proximate problem [8] along with a number of geometric results that will lay the foundation of our subsequent algorithms. Throughout, we assume that a point p is represented by its Cartesian coordinates $(x(p), y(p))$.

Consider a collection $P = \{p_1, p_2, ..., p_n\}$ of n points sorted by x-coordinate, that is, such that $x(p_1) < x(p_2) < ... < x(p_n)$. We assume, without loss of generality, that all the points in P have distinct x-coordinates and that all of them lie above the x-axis. The reader should have no difficulty to confirm that these assumptions are made for convenience only and do not impact the complexity of our algorithms.

Recall that for every point p_i of P the locus of all the points in the plane that are closer to p_i than to any other points in P is referred to as the *Voronoi polygon* associated with p_i and is denoted by $V(i)$. The collection of all the Voronoi polygons of points in P partitions the plane into the Voronoi diagram of P (see [13], p. 204). Let I_i, $(1 \leq i \leq n)$, be the locus of all the points q on the x-axis for which $d(q, p_i) \leq d(q, p_j)$ for all p_j, $(1 \leq j \leq n)$. In other words, $q \in I_i$ if and only if q belongs to the intersection of the x-axis with $V(i)$, as illustrated in Figure 2. In turn, this implies that I_i must be an interval on the x-axis and that some of the intervals I_i, $(2 \leq i \leq n-1)$, may be empty. A point p_i of P is termed a *proximate point* whenever the interval I_i is nonempty. Thus, the Voronoi diagram of P partitions the x-axis into *proximate intervals*. Since the point of P are sorted by x-coordinate, the corresponding proximate intervals are ordered, left to right, as $I : I_1, I_2, ..., I_n$. A point q on the x-axis is said to be a *boundary point* between p_i and p_j if q is equidistance to p_i and p_j, that is, $d(p_i, q) = d(p_j, q)$. It should be clear that p is boundary point between proximate points p_i and p_j if and only if the q is the intersection of the (closed) intervals I_i and I_j. To summarize the previous discussion, we state the following result;

Proposition 2.1: The following statements are satisfied:

1) *Each I_i is an interval on the x-axis;*
2) *The intervals $I_1, I_2, ..., I_n$ lie on x-axis in this order, that is, for any nonempty I_i and I_j with $i < j$, I_i lies to the left of I_j.*
3) *If the nonempty proximate intervals I_i and I_j are adjacent, then the boundary point between p_i and p_j separates $I_i \cup I_j$ into I_i and I_j.*

Referring again to Figure 2, among the seven points, five points p_1, p_2, p_4, p_6 and p_7 are proximate points, while the others are not. Note that the leftmost point p_1 and the rightmost point p_n are always proximate points.

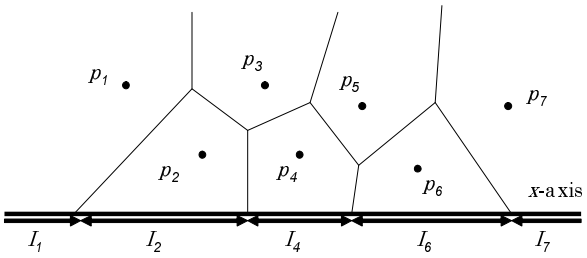


Figure 2. Illustrating proximate intervals

Given three points p_i, p_j, p_k with $i < j < k$, we say that p_j is *dominated* by p_i and p_k whenever p_j fails to be a proximate point of the set consisting of these three points. Clearly, p_j is dominated by p_i and p_k if the boundary of p_i and p_j is to the right of that of p_j and p_k. Since the boundary of any two points can be computed in $O(1)$ time, the task of deciding for every triple (p_i, p_j, p_k), whether p_j is dominated by p_i and p_k takes $O(1)$ time using single processor.

Consider a collection $P = \{p_1, p_2, ..., p_n\}$ of points in the plane sorted by x-coordinate, and a point p to the right of P, that is, such that $x(p_1) < x(p_2) < ... < x(p_n) < x(p)$. We are interested in updating the proximate intervals of P to reflect the addition of p to P, as illustrated in Figure 3.

We assume, without loss of generality, that all points in P are proximate points and let $I_1, I_2, ..., I_n$ be the corresponding proximate intervals. Further, let $I'_1, I'_2, ..., I'_n, I'_p$ be the updated proximate intervals of $P \cup \{p\}$. Let p_i be a point such that I'_i and I'_p are adjacent. By point 3 in Proposition 2.1, the boundary point between p_i and p

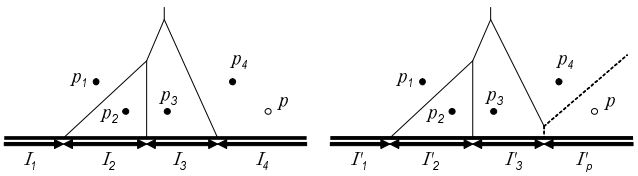


Figure 3. Illustrating the addition of p to $P = \{p_1, p_2, p_3, p_4\}$

separates I'_i and I'_p. As a consequence, point 2 implies that all the proximate intervals $I'_{i+1}, ..., I'_n$ must be empty. Furthermore, the addition of p to P does not affect any of the proximate intervals I_j, $1 \leq j \leq i$. In other words, for all $1 \leq j \leq i$, $I'_j = I_j$. Since $I'_{i+1}, ..., I'_n$ are empty, the points $p_{i+1}, ..., p_n$ are dominated by p_i and p. Thus, every point p_j, $(i < j \leq n)$, is dominated by p_{j-1} and p; otherwise, the boundary between p_{j-1} and p would be to the left of that of that between p_j and p. This would imply that the nonempty interval between these two boundaries corresponds to I'_j, a contradiction. To summarize, we have the following result:

Lemma 2.2: There exists a unique points of p_i of P such that:

- *The only proximate points of $P \cup \{p\}$ are $p_1, p_2, ..., p_i, p$.*
- *For $2 \leq j \leq i$, the point p_j is not dominated by p_{j-1} and p. Moreover, for $1 \leq j \leq i-1$, $I'_j = I_j$.*
- *For $i < j \leq n$, the point p_j is dominated by p_{j-1} and p and the interval I'_j is empty.*
- *I'_i and I'_p are consecutive on the x-axis and are separated by the boundary point between p_i and p.*

Let $P = \{p_1, p_2, ..., p_n\}$ be a collection of proximate points sorted by x-coordinate and let p be a point to the left of P, that is, such that $x(p) < x(p_1) < x(p_2) < ... < x(p_n)$. For further reference, we now take note of the following companion result to Lemma 2.2. The proof is identical and, thus, omitted.

Lemma 2.3: There exists a unique points of p_i of P such that:

- *The only proximate points of $P \cup \{p\}$ are $p, p_i, p_{i+1}, ..., p_n$.*
- *For $i \leq j \leq n$, the point p_j is not dominated by p and p_{j+1}. Moreover, for $i+1 \leq j \leq n$, $I'_j = I_j$.*
- *For $1 \leq j < i$, the point p_j is dominated by p and p_{j+1} and the interval I'_j is empty.*
- *I'_p and I'_i are consecutive on the x-axis and are separated by the boundary point between p and p_i.*

The unique point p_i whose existence is guaranteed by Lemma 2.2 is termed the *contact point* between P and p. The second statement of Lemma 2.2 suggests that the task of determining the unique contact point between P and a point p to the right or the left of P reduces, essentially, to binary search.

Now, suppose that the set $P = \{p_1, p_2, ..., p_{2n}\}$, with $x(p_1) < x(p_2) < ... < x(p_{2n})$ is partitioned into two subsets $P_L = \{p_1, p_2, ..., p_n\}$ and $P_R = \{p_{n+1}, p_{n+2}, ..., p_{2n}\}$. We are interested in updating the proximate intervals in the process or merging P_L and P_R. For this purpose, let $I_1, I_2, ..., I_n$ and $I_{n+1}, I_{n+2}, ..., I_{2n}$ be the proximate intervals of P_L and P_R, respectively. We assume, without loss of generality, that all these proximate intervals are nonempty. Let $I'_1, I'_2, ..., I'_{2n}$ be the proximate intervals of $P = P_L \cup P_R$. We are now in a position to state and prove the next result which turns out to be a key ingredient in our algorithms.

Lemma 2.4: There exists a unique pair of proximate points $p_i \in P_L$ and $p_j \in P_R$ such that

- *The only proximate points in $P_L \cup P_R$ are $p_1, p_2, ..., p_i, p_j, ..., p_{2n}$.*
- *$I'_{i+1}, ..., I'_{j-1}$ are empty, and $I'_k = I_k$ for $1 \leq k \leq i-1$ and $j+1 \leq k \leq 2n$.*
- *The proximate intervals I'_i and I'_j are consecutive and are separated by the boundary point between p_i and p_j.*

Proof: Let i be the smallest subscript for which $p_i \in P_L$ is the contact point between P_L and a point in P_R. Similarly, let j be the largest subscript for which the point $p_j \in P_R$ is the contact point between P_R and some point in P_L. Clearly, no point in P_L to the left of p_i can be proximate point of P. Likewise, no point in P_R to the left of p_j can be a proximate point of P.

Finally, by Lemma 2.2, every point in P_L to the left of p_i must be a proximate point of P. Similarly, by Lemma 2.3, every point in P_R to the right of p_i must be a proximate point of P, and proof of the lemma is complete. ■

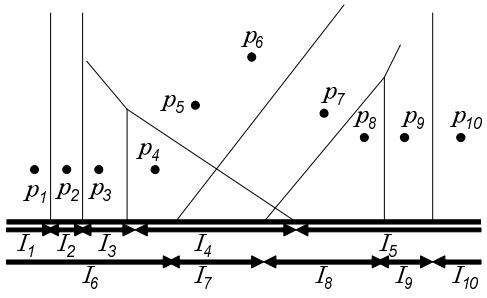
The points p_i and p_j whose existence is guaranteed by Theorem 2.4 are termed the *contact points* between P_L and P_R. We refer the reader to Figure 4 for an illustration. Here, the contact points between $P_L = \{p_1, p_2, p_3, p_4, p_5\}$ and $P_R = \{p_6, p_7, p_8, p_9, p_{10}\}$ are p_4 and p_8.

Next, we discuss a geometric property that enables the computation of the contact points p_i and p_j between P_L and P_R. For each point p_k of P_L, let q_k denote the contact point between p_k and P_R as specified by Lemma 2.3. We have the following result.

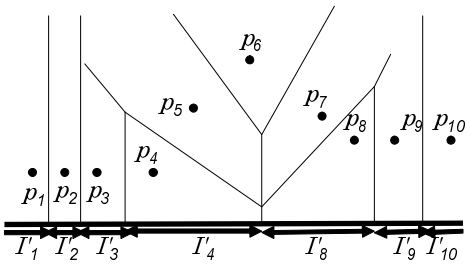
Lemma 2.5: The point p_k is not dominated by p_{k-1} and q_k if $2 \leq k \leq i$, and dominated otherwise.

Proof: If p_k, $(2 \leq k \leq i)$, is dominated by p_{k-1} and q_k, then I'_k must be empty. Thus, Lemma 2.4 guarantees that p_k, $(2 \leq k \leq i)$, is not dominated by p_{k-1} and q_k. Suppose that p_k, $(i+1 \leq k \leq n)$, is not dominated by p_{k-1} and q_k. Then, the boundary point between p_k and q_k is to the right of that between these two boundaries corresponds to I'_k, a contradiction. Therefore, p_k, $(i+1 \leq k \leq n)$, is dominated by p_{k-1} and q_k, completing the proof. ■

Lemma 2.5 suggests a simple, binary search-like, approach to finding the contact points p_i and p_j between two sets P_L and P_R. In fact, using a similar idea, Breu et al. [1]



(a) Proximate interval of each point in two sets



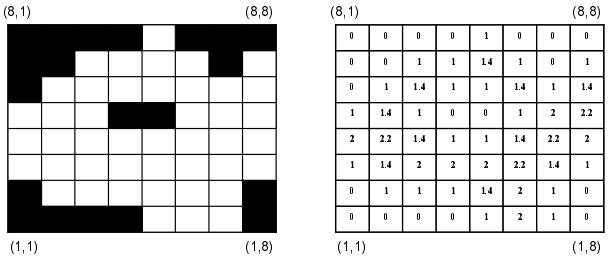
(b) Merge of two point sets and their contact points

Figure 4. Illustrating the contact points between two sets of points

proposed a sequential algorithm that computes the proximate points of an n-point planar set in $O(n)$ time. The algorithm in [1] uses a stack to store the proximate points found.

III. Parallel Euclidean Distance Map of 2-D Binary Image

A binary image I of size $n \times n$ is maintained in an array $b_{i,j}$, $(1 \leq i, j \leq n)$. It is customary to refer to pixel (i, j) as *black* if $b_{i,j} = 1$ and as *white* if $b_{i,j} = 0$. The rows of the image will be numbered bottom up starting from 1. Likewise, the columns will be numbered left to right, with column 1 being the leftmost. In this notation, pixel $b_{1,1}$ is in the south-west corner of the image, as illustrated in Figure 5(a). In Figure 5(a), each square represents a pixel. For this binary image, its final distance mapping array is shown in Figure 5(b).



(a) Binary image (b) Mapping array

Figure 5. A binary image and its mapping array

The *Voronoi map* associates with every pixel in I the closest black pixel to it (in the Euclidean metric). More formally, the Voronoi map of I is a function $v : I \rightarrow I$ such that, for every (i, j), $(1 \leq i, j \leq n)$, $v(i, j) = v(i', j')$ if and only if

$$d((i, j), (i', j')) = min\{d((i, j), (i'', j'')) \mid b_{i'',j''} = 1\},$$

where $d((i, j), (i', j')) = \sqrt{(i - i')^2 + (j - j')^2}$ is the Euclidean distance between pixels (i, j) and (i', j').

The *Euclidean distance map* of image I associates with every pixel in I in the Euclidean distance to the closest black pixel. Formally, the Euclidean distance map is a function m: $I \rightarrow R$ such that for every (i, j), $(1 \leq i, j \leq n)$, $m(i, j) = d((i, j), v(i, j))$.

We now outline the basic idea of our algorithm for computing the Euclidean distance map of image I. We begin by determining, for every pixel in row j, $(1 \leq j \leq n)$, the nearest black pixel, if any, in the same column of subimage of I. More precisely, with every pixel (i, j) we associate the value

$$d_{i,j} = min\{d((i, j), (i', j')) \mid b_{i',j'} = 1, 1 \leq j' \leq n\}.$$

If $b_{i',j'} = 0$ for every $1 \leq j' \leq n$, then let $d_{i,j} = +\infty$. Next, we construct an instance of the proximate points problem for every row j, $(1 \leq j \leq n)$, in the image I involving the set P_j of points in the plane defined as $P_j = \{p_{i,j} = (i, d_{i,j}) \mid 1 \leq i \leq n\}$.

Having solved, in parallel, all these instances of the proximate points problem, we determine, for every proximate point $p_{i,j}$ in P_j, its corresponding proximity interval I_i. With j fixed, we determine, for every pixel (i, j) (that we perceive as a point on the x-axis), the identity of the proximity interval to which it belongs. This allows each pixel (i, j) to determine the identity of the nearest pixel to it. The same task is executed for all rows $1, 2, ..., n$ in parallel, to determine, for every pixel (i, j) in row j, the nearest black pixel. The details are spelled out in the following algorithm:

Algorithm *Euclidean Distance Map(I)*

Step 1. For each pixel (i, j), compute the distance $d_{i,j} = min\{d((i, j), (i', j')) \mid b_{i',j'} = 1, 1 \leq j' \leq n\}$ to the nearest black pixel in the same column as (i, j) in the subimage of I.

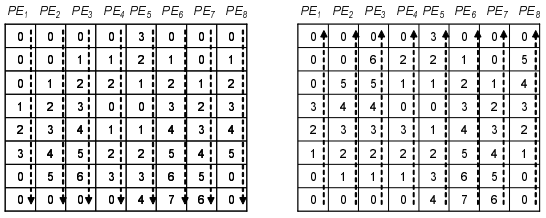
Step 2. For every j, $(1 \leq j \leq n)$, let $P_j = \{p_{i,j} = (i, d_{i,j}) \mid 1 \leq i \leq n\}$. Compute the proximate points $E(P_j)$ of P_j.

Step 3. For every point p in $E(P_j)$ determine its proximity interval of P_j.

Step 4. For every i, $(1 \leq i \leq n)$, determine the proximate interval of P_j to which the point $(i, 0)$ (corresponding to pixel (i, j)) belongs.

We assume that there are k processors PE(1), PE(2), ..., PE(k) available. The parallel implementation of above algorithm is shown in follow:

Step 1. Partition the input image I into k subimages $I_1, I_2, ..., I_k$ along with column wise. For every pixel of each subimage I_i ($1 \leq i \leq k$), corresponding processor PE(i) computes the distance to the nearest black pixel in the same column. In real implementation, first, each processor travels every column of corresponding subimage from up to bottom to compute that distance, as illustrated in Figure 6(a) (its original input image is shown in Fig 5). Second, each processor again travel every column of corresponding subimage from bottom to up to compute that distance, as illustrated in Figure 6(b). Finally, each processor selects a minimum value of calculated two distances as final value of the distance. It is clear that the time complexity of this step is $O(n^2/k)$.



(a) process with up to bottom (b) process with bottom to up

Figure 6. Process each column with two directions

Step 2. Again, we compute Euclidean distance map of input image I along with row wise.

Step 2.1. Partition the input image into k subimages $I'_1, I'_2, ..., I'_k$ along with row wise. For every row of each subimage I'_i ($1 \leq i \leq k$), each processor PE(i) ($1 \leq i \leq k$) computes the proximate points using the theorem of proximate points problem as foundation, as illustrated in Figure 7 and Figure 8. In Figure 8, the Voronoi polygons correspond to the 5th row (shaded row) of the image illustrated in Fig 7. The obtained proximate points are saved in a stack. It should be clear that each column has its own corresponding stack. Therefore, in order to add a new proximate point to the stack, we need to calculate boundary points of this new point and existed proximate points which are kept in the stack. Then according to locus of boundary points, we decide which point need to be deleted from the stack.

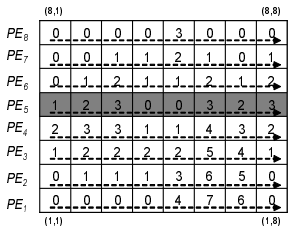


Figure 7. Processing with row wise

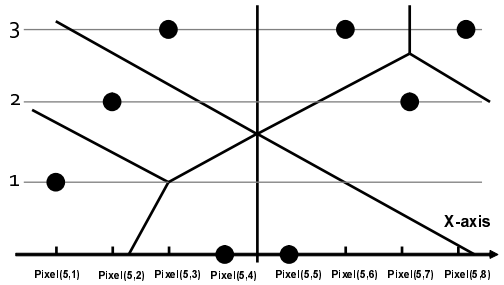


Figure 8. Voronoi polygons

Step 2.2. For every row of each subimage I'_i ($1 \leq i \leq k$), each processor PE(i) determines proximate intervals of obtained proximate points by computing boundary point of each pair of adjacent proximate points. The boundary point of each pair of adjacent proximate points can be obtained by calculating the intersection point of two lines, one line is x-axis and another is the normal line of the line which connects two adjacent proximate points. We refer reader to Figure 9 for the illustration. Each pair of adjacent proximate points can be obtained from the stack.

Step 2.3. According to the locus of boundary points obtained from Step 2.2, each processor determines the closest black pixel to each pixel of input image. The distance between a given pixel and its closest black pixel is also calculated in an obvious way.

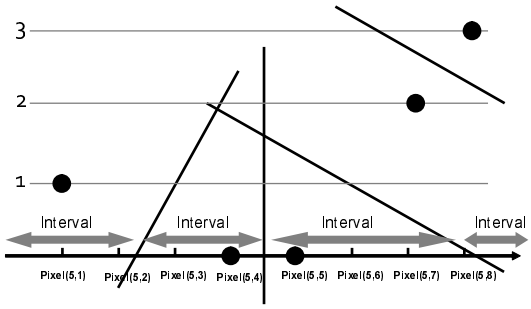


Figure 9. Proximate intervals

It is clear that, the whole Step 2 can be implemented in $O(\frac{n^2}{k})$ time using k processors.

Theorem 3.1: For a given binary image I with the size of $n \times n$, Euclidean Distance Map of image I can be computed in $O(\frac{n^2}{k})$ time using k processors.

IV. Experimental Results

We have implemented and evaluated our proposed parallel EDM algorithm in the following two platforms, a general multi-core processors system and a modern GPU system, respectively. The multi-core processors system is a Linux server with four Intel hexad-core processors (Intel Xeon X7460 2.66GHz [11]), that is, there are twenty four cores available. Each multi-core processor has its own local three-level caches that are 64KB L1 cache, 3MB L2 cache and 16MB L3 cache. The capacity of the main memory is 128GB. The experimental GPU system is a Tesla C1060 [14] which consists of 240 Streaming Processor Cores and 4GB global memory.

As known, in general, a matrix is stored in a row-major fashion in memory. Therefore, for a given program, different access modes will result in different performances. In order to find out in which access mode the implementation of our parallel EDM algorithm can achieve the best performance, we have evaluated our parallel EDM algorithm in the following four access modes: HVHV (Horizontal-Vertical-Horizontal-Vertical) access mode, HHVV (Horizontal-Horizontal-Vertical-Vertical) access mode, VVHH (Vertical-Vertical-Horizontal-Horizontal) access mode and VHVH (Vertical-Horizontal-Vertical-Horizontal) access mode. We refer reader to Figure 10 for illustrating HVHV access mode. As illustrated in Figure 10, for implementing Step 1 of the proposed algorithm, each processor reads the corresponding subimage along with row wise, and processes the subimage. After that, each processor writes the processing results into an extra array along with column wise. In implementation of Step 2, each processor reads the extra array along with row wise and processes the obtained data. Then, each processor writes the processed data into the extra array along with column wise again. In final, we can obtain Euclidean distance map for the input image. It is clear that, the name of HVHV access mode comes from the row-wise read and column-wise write of Step 1, the row-wise read and column-wise write of Step 2. In the same way, we can understand other access modes easily.

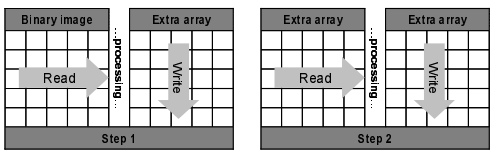


Figure 10. HVHV access mode

Our proposed algorithm has implemented in C language with OpenMP 2.0 (Open Multi-Processing) in that of multi-core processor system. The OpenMP is an application programming interface that supports shared memory environment [15]. It consists of a set of compiler directives and library routines. By using OpenMP, it is relatively easy to create parallel applications in FORTRAN, C, and C++. Table I has shown the performance of our proposed algorithm with different access modes in the multi-core processors system. The size of the input image is 10000×10000. In the table, each measurement is an average value of 20 experiments and, Step 1 and Step 2 is corresponding steps of our proposed parallel algorithm. It is clear that, in HVHV access mode, our implementation can achieve the best performance and it can obtain approximate 18 times speedup. The table also exhibit the scalability of the proposed algorithm. As shown, our proposed algorithm can scale well with the number of using cores smaller than or equal to 4. Actually we have implemented the proposed algorithm in a multi-processor system with 4 multi-core processors. Therefore when the number of using cores is smaller than or equal to 4, all the using cores will be distributed into different multi-core processors. Consequently each level cache of a multi-core processor occupied by only one core. It means only one core utilizing all the available cache. But, when the number of using threads is more than 4, then the scalability of our implementation is decreasing significantly. One main reason of the phenomenon is that, when the number of using cores is larger than 4, then L2 and L3 cache of each multi-core processor will be shared by multiple cores. It will decrease the efficiency of our implementation significantly. Meanwhile, many other factors such as Memory-CPU bus bandwidth, communication overhead and synchronization overhead also play the important roles in the scalability. Hence we can understand why the real speedup is decreasing along with increasing the number of using processors. On the other hand, we have also evaluated the proposed parallel algorithm with the different sized input images. Table II shows the performance of the proposed parallel algorithm for processing images with different sizes. As shown in the table, for processing small sized images, the performance of multiple cores is poor than the performance of single core, because in comparison with total execution time, there is considerable overhead due to parallel processing.

In another way, our proposed algorithm has been implemented in that of GPU system using CUDA (Compute Unified Device Architecture) [12], a general purpose parallel computing architecture. Actually, CUDA is a new parallel programming model and instruction set architecture. CUDA comes with a software environment that allows developer to use C-like high-level programming language. Table III has shown the performance of our proposed algorithm with different access modes in that of GPU system. The input image is the same image used in the multi-core implementation. Recall that the size of input image is 10000×10000. As shown, in VHVH access mode, our proposed algorithm

Table I
PERFORMANCE OF PROPOSED ALGORITHM IN MULTI-CORE PROCESSORS SYSTEM WITH DIFFERENT ACCESS MODE (n=10000)

(a) HVHV access mode

Num of using cores	Step 1 [s]	Speedup	Step 2 [s]	Speedup	Total [s]	Speedup
1	1.8830	1.000	7.1138	1.000	8.9968	1.000
2	0.9740	1.933	3.6592	1.944	4.6333	1.941
4	0.5136	3.666	1.8465	3.852	2.3601	3.812
8	0.3007	6.262	0.9525	7.468	1.2532	7.179
12	0.2254	8.354	0.7063	10.071	0.9317	9.656
16	0.1776	10.602	0.5723	12.430	0.7499	11.997
20	0.1614	11.666	0.4868	14.613	0.6482	13.879
24	0.1617	11.645	0.3299	21.563	0.4916	18.301

(b) HHVV access mode

Num of using cores	Step 1 [s]	Speedup	Step 2 [s]	Speedup	Total [s]	Speedup
1	0.5270	1.000	11.2238	1.000	11.7508	1.000
2	0.2690	1.959	5.6299	1.993	5.8989	1.992
4	0.1694	3.110	2.9462	3.809	3.1156	3.771
8	0.1074	4.906	1.4563	7.707	1.5637	7.514
12	0.1050	5.019	1.2843	8.739	1.3893	8.458
16	0.1040	5.067	0.9707	11.562	1.0747	10.934
20	0.1065	4.948	0.8423	13.325	0.9488	12.384
24	0.1089	4.839	0.9488	11.829	1.0577	11.109

(c) VVHH access mode

Num of using cores	Step 1 [s]	Speedup	Step 2 [s]	Speedup	Total [s]	Speedup
1	4.6438	1.000	6.1536	1.000	10.7974	1.000
2	2.3806	1.950	3.0819	1.996	5.4625	1.976
4	1.2133	3.827	1.5607	3.942	2.774	3.892
8	0.6396	7.260	0.7971	7.719	1.4367	7.515
12	0.5992	7.750	0.5311	11.586	1.1303	9.552
16	0.5809	7.994	0.4036	15.246	0.9845	10.967
20	0.5583	8.317	0.3253	18.916	0.8836	12.219
24	0.6363	7.298	0.2707	22.732	0.9070	11.904

(d) VHVH access mode

Num of using cores	Step 1 [s]	Speedup	Step 2 [s]	Speedup	Total [s]	Speedup
1	2.4308	1.000	8.6189	1.000	11.0497	1.000
2	1.2310	1.974	4.4083	1.955	5.6393	1.959
4	0.6307	3.854	2.2258	3.872	2.8565	3.868
8	0.3775	6.439	1.3138	6.560	1.6913	6.533
12	0.2605	9.331	0.8333	10.343	1.0938	10.102
16	0.2439	9.966	0.6569	13.120	0.9008	12.266
20	0.1970	12.339	0.5417	15.910	0.7387	14.958
24	0.1733	14.026	0.4617	18.667	0.6350	17.401

can achieve the best performance.

Table III
PERFORMANCE OF PROPOSED ALGORITHM ON GPU SYSTEM WITH DIFFERENT ACCESS MODE (n=10000)

Access mode	Step 1 [ms]	Step 2 [ms]	Total time [ms]
VHVH	572.686	1151.903	1724.589
VVHH	439.841	1794.226	2234.067
HVHV	767.606	1691.302	2458.908
HHVV	1005.373	931.2120	1936.585

Table IV has shown the speedup of our GPU implementation comparing with the performance of single CPU implementation. It is clear that, for processing image with size of 10000×10000, our GPU implementation has achieved approximate 5 times speedup.

Table IV
PERFORMANCE OF PROPOSED ALGORITHM ON GPU SYSTEM FOR PROCESSING IMAGES WITH DIFFERENT SIZES

Size of image		CPU	GPU	Speedup
100 × 100	Step 1 [ms]	0.087	0.290	0.300
	Step 2 [ms]	0.679	0.631	1.070
	Total [ms]	0.766	0.921	0.831
1000 × 1000	Step 1 [ms]	9.905	3.861	2.565
	Step 2 [ms]	69.063	7.392	9.342
	Total [ms]	78.969	11.253	7.017
10000 × 10000	Step 1 [ms]	1883.005	572.686	3.288
	Step 2 [ms]	7113.811	1151.903	6.175
	Total [ms]	8996.816	1724.589	5.216

V. CONCLUDING REMARKS

In this paper, we have presented an optimal parallel algorithm for computing Euclidean Distance Map (EDM)

Table II
PERFORMANCE OF PROPOSED ALGORITHM IN MULTI-CORE PROCESSORS SYSTEM FOR PROCESSING IMAGES WITH DIFFERENT SIZES

Size of input image	Num of using cores	Step 1 [s]	Speedup	Step 2 [s]	Speedup	Total [s]	Speedup
100×100	1	0.000087	1.000	0.000679	1.000	0.000766	1.000
	24	0.011183	0.007	0.000348	1.95	0.011531	0.066
1000×1000	1	0.009905	1.000	0.069063	1.000	0.078969	1.000
	24	0.013467	0.735	0.003731	18.510	0.017198	4.59
10000×10000	1	1.8830	1.000	7.1138	1.000	8.9968	1.000
	24	0.1617	11.645	0.3299	21.563	0.4916	18.301

of a 2-D binary image. Using proximate points problem as preliminary foundation, we have proposed a simple but efficient parallel EDM algorithm which can achieve $O(\frac{n^2}{k})$ time using k processors. To evaluate the performance of the proposed algorithm, we have implemented it in a Linux server with four Intel hexad-core processors (Intel Xeon X7460 2.66GHz) [11] and a modern GPU (Graphics Processing Unit) system, Tesla C1060 [14], respectively. The experimental results have shown that, for an input binary image with size of 10000×10000, the proposed parallel algorithm can achieve 18 times speedup in the multi-core system, comparing with the performance of general sequential algorithm. Meanwhile, for the same input image, the proposed parallel algorithm can achieve 5 times speedup in that of GPU system.

REFERENCES

[1] H. Breu, J. Gil, D. Kirkpatrick, and M. Werman, "*Linear Time Euclidean Distance Transform Algorithms*," IEEE Trans. Pattern Analysis and Machine Intelligence, vol.17, no.5, pp.529–533, May 1995.

[2] L. Chen, "*Optimal Algorithm for Complete Euclidean Distance Transform*," Chinese J. Computers, vol.18, no.8, pp.611–616, 1995.

[3] L. Chen and H.Y.H. Chuang, "*A Fast Algorithm for Euclidean Distance Maps of a 2-D Binary Image*," Information Processing Letters, vol.51, pp.25–29, 1994.

[4] T. Hirata, "*A Unified Linear-Time Algorithm for Computing Distance Maps*," Information Processing Letters, vol.58, pp.129–133, 1996.

[5] Y.-H. Lee, S.-J. Horng, T.-W. Kao, F.-S. Jaung, Y.-J. Chen, and H.-R. Tsai, "*Parallel Computation of Exact Euclidean Distance Transform*," Parallel Computing, vol.22, no.2, pp.311–325, 1996.

[6] S. Pavel and S.G. Akl, "*Efficient Algorithms for the Euclidean Distance Transform*," Parallel Processing Letters, vol.5, no.2, pp.205–212, 1995.

[7] A. Fujiwara, T. Masuzawa, and H. Fujiwara, "*An Optimal Parallel Algorithm for the Euclidean Distance Maps of 2-D binary images*," Information Processing Letters, vol.54, pp.295–300, 1995.

[8] T. Hayashi, K. Nakano, and S. Olariu, "*Optimal Parallel Algorithm for Finding Proximate Points, with Applications*," IEEE Transactions on Parallel and Distributed Systems, vol.9, no.12, pp.1153–1166, Dec. 1998.

[9] L. Chen, P. Yi, Ch. Yixin, and X. Xiaohua, "*Efficient Parallel Algorithms for Euclidean Distance Transform*," The Computer Journal, vol.47, no.6, pp.694–700, 2004.

[10] Li. K and Z.S. Q, *Parallel Computing Using Optical Interconnections*, Boston, USA, Kluwer Academic Publishers, 1998.

[11] Intel Corporation., "Intel Xeon Processor 5000 Sequence". http://www.intel.com/products/processor/xeon7000/

[12] NVIDIA Corporation., "NVIDIA, CUDA Architecture". http://www.nvidia.com/object/cuda_home_new.html

[13] F.P. Preparata and M.I. Shamos, *Computational Geometry: An Introduction*, third corrected printing edition, Berlin: Springer-Verlag, 1990.

[14] NVIDIA Corporation., "Tesla C1060 Computing Processor". http://www.nvidia.com/object/product_tesla_c1060_us.html

[15] OpenMP.org, "OpenMP Application Program Interface". http://www.openmp.org