Distributed Vision with Smart Pixels

Sándor P. Fekete
Algorithms Group
Braunschweig Institute of Technology
Germany
s.fekete@tu-bs.de

Dietmar Fey
Institute for Computer Science
Friedrich-Schiller-University
Jena, Germany
fey@uni-jena.de

Marcus Komann
Institute for Computer Science
Friedrich-Schiller-University
Jena, Germany
marcus.komann@web.de

Alexander Kröller
Algorithms Group
Braunschweig Institute of Technology
Germany
a.kroeller@tu-bs.de

Marc Reichenbach
Institute for Computer Science
Friedrich-Schiller-University
Jena, Germany
marc.reichenbach@googlemail.com

Christiane Schmidt
Algorithms Group
Braunschweig Institute of Technology
Germany
c.schmidt@tu-bs.de

ABSTRACT

We study a problem related to computer vision: How can a field of sensors compute higher-level properties of observed objects deterministically in sublinear time, without accessing a central authority? This issue is not only important for real-time processing of images, but lies at the very heart of understanding how a brain may be able to function.

In particular, we consider a quadratic field of n "smart pixels" on a video chip that observe a B/W image. Each pixel can exchange low-level information with its immediate neighbors. We show that it is possible to compute the centers of gravity along with a principal component analysis of all connected components of the black grid graph in time $O(\sqrt{n})$, by developing appropriate distributed protocols that are modeled after sweepline methods.

Our method is not only interesting from a philosophical and theoretical point of view, it is also useful for actual applications for controling a robot arm that has to seize objects on a moving belt. We describe details of an implementation on an FPGA; the code has also been turned into a hardware design for an application-specific integrated circuit (ASIC).

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems (E.2-5, G.2, H.2-3)—*Geometrical problems and computations*

General Terms

Algorithms

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SCG'09, June 8–10, 2009, Aarhus, Denmark.
Copyright 2009 ACM 978-1-60558-501-7/09/06 ...$5.00.

Keywords

distributed vision, distributed algorithms, sublinear algorithms, principal component analysis, sweepline algorithms.

1. INTRODUCTION

Fast Geometric Algorithms. A major part of the research conducted in computational geometry shoots for much more than just polynomial runtime. Often motivated by applications from areas such as computer graphics or computer vision, some of the most impressive work has resulted in linear-time methods, e.g., for triangulating a simple polygon [6]. However, speed and accuracy of geometric processing may not just matter in terms of theoretical CPU time; it can also make the difference between efficient quality control for ready-made food [20] or letting defective products leave the factory; between a robot arm accurately seizing a large variety of machine parts from a conveyor belt or production coming to a sudden and unplanned, screeching halt [5]; between a baseball player cleanly catching a line drive or being hit squarely into the face [12].

Vision. Throughout human evolution, quality and speed of visual processing has not only been a matter of win or loss, but of life and death. As a result, human vision is not just based on amazing software for image processing, but also on extremely powerful hardware. How does the human brain process visual information and what can we learn from it? This has been one of the most intriguing scientific questions ever, not just in computer science, but also neurobiology, art, and philosophy [16]. Quite clearly, human vision consists of different mechanisms and filters, combining a variety of fast heuristics, parallel algorithms, but also sophisticated algorithms for performing highly specialized analytic tasks.

Processing Visual Information. How do computers process visual information? Typically, the input from a grid of pixels is fed into the CPU, where more or less sophisticated operations are performed. Quite clearly, just looking at the input takes least linear time for whatever problem is to be solved in a deterministic manner. Can we learn anything about or from the functioning of our brain? When discussing human vision, even the innocent expression "looking at the input" alludes to a philosophical issue that goes much deeper when trying to understand image processing

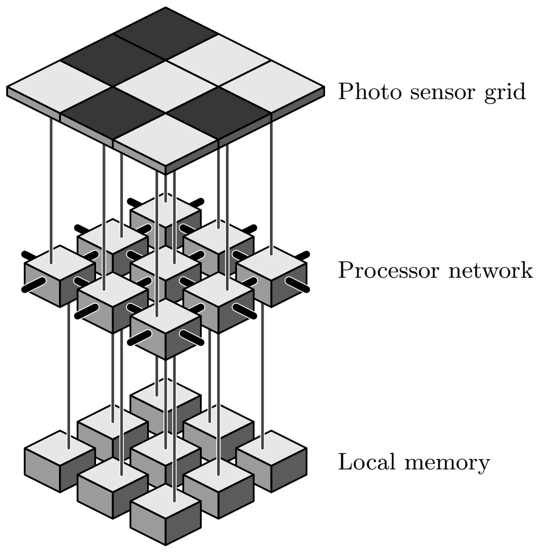


Figure 1: Schematic overview of the hardware.

in the human mind: What behind the retina is it that does the looking? Modeling our mind analogous to a standard centralized algorithmic model leads to an infinite recourse (known, e.g., as Ryle's Regress [18].) As long as we think of algorithms as sentient individuals ("an algorithm must consider all of the input in order to make a decision"), there is no way out.

Sublinear Runtime. Getting to sublinear running times requires dropping most of the input, possibly by resorting to probabilistic methods [7, 8]. A possible alternative comes from making use of parallel algorithms [1]; however, simply slicing up the visual input into a large number of parallel processors will not do the trick, as long as the geometric structure of the input and the communication structure of the processors or input size and processor number are unrelated. But how can they be related for fixed hardware and dynamic inputs—in particular if we are to deal with tasks that require nontrivial computation?

Motivation and Inspiration for This Paper. The work on this paper was triggered by a real-life application, but inspired by the fascination of related algorithmic and philosophical issues. The application comes up in one of the above scenarios: An autonomous robot arm is equipped with a video camera to keep track of large quantities of machine pieces zipping by on a fast conveyor belt. These pieces have to be seized, which requires not just identifying their position, but also their center of gravity along with their orientation in space. Obviously, this is a demanding task that requires ultra-fast and accurate image processing.

The inspiration comes from special hardware that can be used for this task, see Figure 1: A camera chip consists of a rectangular grid of sensor pixels, each one capable of detecting small parts of an image. While a standard pixel is relatively dumb (all it can do is detect light and forward the resulting data to a central processor), a *smart pixel* is able to communicate with its immediate neighbors, do simple computations, and react depending on events. A smart pixel grid is not just a concept: This hardware does exist, albeit at limited grid sizes [11]. All that is missing as an incentive for the actual production of smart pixel grids of higher resolution is a demonstration of the practical rewards—together with algorithmic methods that are capable of exploiting the possibilities: There is a large set of parallel processors, communication is limited to a small local neighborhood, so significant speedup is possible; the hardware design implies that the geometry of input and processor locations is strongly correlated, as are input size and processor number; however, extracting individual parts, their centers of gravity along with their orientation appears to require a combination of local and global computations. Finally, visual processing with a grid of smart pixels has a striking resemblance to image processing in the brain. This means that developing geometric algorithms for this kind of architecture is not just a matter of speedup—it is also an indication that it may be possible to overcome Ryle's Regress by considering a different kind of processor architecture, closely intertwined with a different, distributed algorithmic approach.

Problem Definition and Main Results. Formally, we define our problem as follows: We are given a $W \times H$ black-and-white pixel image: black pixels belong to objects, white corresponds to background. To get rid of special cases in the notation below, we assume the image to be embedded in an infinitely-sized grid of white. Other than the different colors, the pixels are the same. They have no unique identifier, i.e., neither an ID nor coordinates are associated with the pixels. Each pixel has eight communication lines to the direct neighbors (w.r.t. L_∞). We may utilize moving agents on the pixel field with only local behavior and knowledge.

The black pixels define a set of objects, each of which is a maximal connected subset of them, where connectivity is defined horizontally and vertically only. That is, two black pixels on diagonally adjacent cells do not neccessarily belong to the same object.

The task is as follows: When the algorithm finishes, there is one representative pixel for each object. This pixel provides access to descriptive information on the object, namely the size (i.e., number of pixels), the center of gravity, and the orientation (which we define as the direction of maximum variance).

The main algorithmic results of this paper is a sweepline algorithm that correctly extracts the demanded attributes. This algorithm runs in $O(W + H)$ and is able to distinguish several objects within a single image. Moreover, an implementation in actual hardware shows that our algorithm is applicable for our scenario, featuring low power consumption and high chip frequency.

The rest of the paper is organized as follows. In the following Section 2 we describe related work, a description of the algorithm is given in Section 3. Section 4 provides proofs of the running time of the algorithm. The results of an implementation in hardware are presented in Section 5. In the final Section 6 we discuss possible implications and extensions.

2. RELATED WORK

Object Detection. We aim at detecting and describing the (potentially unknown) objects in a given B/W image. While we are interested in using distributed algorithms on a field of sensors, object detection, pattern recognition and object recognition (see e.g. [13]) are well-studied, important problems. All these are in the scope of computer vision (see [4]), which of course also involves precedent tasks as describing an image, on this basis recognition—recognizing local discontinuities in intensity for identifying edges and, more sophisticated, segments, shape and clusters. Object, pattern or face recognition concentrates on certifying whether pre-defined or learned objects (patterns, faces) or object classes are present in the given image. Object identification deals with an even more specified task: within a given class (e.g., cars) a certain object is to be identified, see [10]. By contrast, we want to detect an unknown object (or objects) and determine, e.g., its center of gravity. While the task of distinguishing the object from the background is—as we face monochrome images—not our focus of interest.

Principal Component Analysis. For the detected objects we are interested in several attributes, like centroid position or orientation. These attributes can be described by moments. For given data sets, attributes like orientation can be determined with a Principal Component Analysis (PCA) [14, 19]. This analysis is widely used for statistical analysis of data, e.g., from neuroscience, gene expression data (see [21]), as well as in computer vision, for both representation (see [13]) and image compression.

Determining the eigenvectors of the covariance matrix and ordering these by the related eigenvalue (highest to lowest), gives the components in order of significance. That is, for visualized data with coordinates the first principal component gives the orientation.

Distributed Algorithms. Our main contribution in this paper is a distributed sweep over the objects. A straightforward approach would be to run standard distributed algorithm on the graph induced by the object pixels. Any leader election or spanning tree algorithm could be used instead of our algorithm, see [2, 3, 9]. A distributed algorithm that stays within the object pixels can not have a better runtime than $\Omega(m^2)$ for objects on an $m \times m$-pixel image, as there are objects with such a graph diameter. Our algorithm fully exploits the geometric structure of the underlying network by also running on non-object pixels, resulting in a distributed time complexity of $O(m)$.

Own work. We published a predecessor paper [15] to this one. It contains the same distributed mechanism to compute the desired values as in this paper, so we keep the according Section 3.1 informal here. The mentioned paper does not contain the distributed sweep algorithm, which is the major contribution here. Instead, we described a heuristic that works reasonably well on convex and well-separated objects, yet did not allow us to prove any quality measures.

3. THE ALGORITHM

In this section we describe the problem settings as well as the parts of our algorithm.

Pixel Grid. We assume there is an infinite grid of smart pixels. All pixels are completely identical, that is, they have neither a unique ID, nor coordinates available. This assumption is helpful for our application, as it allows us to reset the grid to a clean initial state between runs, i.e., without having to run a global initialization algorithm.

Each pixel p can communicate with its 8 direct neighbors, this set is denoted by $\Gamma(p)$. We define $\overline{\Gamma}(p) := \Gamma(p) \cup \{p\}$. The pixels in $\Gamma(p)$ are identified by direction and denoted $\mathsf{N}(p)$, $\mathsf{NW}(p)$, $\mathsf{W}(p)$, $\mathsf{SW}(p)$, $\mathsf{S}(p)$, $\mathsf{SE}(p)$, $\mathsf{E}(p)$, and $\mathsf{NE}(p)$, with the obvious interpretation.

Distributed Model. We use the synchronized $\mathcal{LOCAL}$ model by Peleg [17]. To be precise, our pixel network runs in synchronized rounds. In each round, each pixel may perform any computation on the data it has available, and it may send any amount of data to its neighbors.

Objects. There is a photo $\mathcal{X}$ overlayed on the pixel grid. Each pixel p either sees part of the objects (if $p \in \mathcal{X}$), or it sees the background. In the first communication round, the pixels exchange this information with their neighbors. In the algorithm below, we assume each pixel p to know the object status of all pixels in $\overline{\Gamma}(p)$. $\mathcal{X}$ may consist of multiple objects. We say that two pixels in $\mathcal{X}$ belong to the same object, if they can be connected by a path in $\mathcal{X}$ using only horizontal and vertical steps (that is, an object is a connected component of $\mathcal{X}$ in the canonical 4-regular grid graph).

3.1 Moments

We already described our scheme for calculating the moments (without the sweep algorithm) in a previous paper [15]. Here we just summarize it, so that the reader can get a complete picture of the algorithm.

Let $\mathcal{X}' \subseteq \mathcal{X}$ be an object in $\mathcal{X}$. Our aim is to calculate

1. The size $|\mathcal{X}'|$ of the object.
2. The center of gravity $(\mu_{\mathrm{x}}, \mu_{\mathrm{y}})$ of the object, i.e.,
$$\mu_{\mathrm{x}} = \frac{1}{|\mathcal{X}'|} \sum_{p \in \mathcal{X}'} x_p \ , \quad \mu_{\mathrm{y}} = \frac{1}{|\mathcal{X}'|} \sum_{p \in \mathcal{X}'} y_p \ , \qquad (1)$$
where (x_p, y_p) are the coordinates of pixel p in some global coordinate system.
3. The second moments in the form of the covariance matrix
$$\begin{pmatrix} \sigma_{\mathrm{x}}^2 & \sigma_{\mathrm{xy}} \\ \sigma_{\mathrm{xy}} & \sigma_{\mathrm{y}}^2 \end{pmatrix}, \qquad (2)$$
where we see the object as a point distribution on $\mathbb{R}^2$. The two Principal Component Axes are then defined by the Eigenvectors of this matrix, see [19]. The application uses them to identify the direction of greatest variance, which is used as the orientation (rotation) of the object on the belt.

As the pixels do not know the global coordinate system, we cannot compute these values directly. Instead, we employ a mechanism of cumulating relative weights. We construct a scheme by which agents sweep over the object. The agents "consume" the "weight" of object pixels as they pass, and they eventually collect these picked up weights in a single pixel. Each object pixel has a weight of 1, and this weight can only be consumed once.

Each agent on a pixel (x, y), and possessing the weights for object pixels $X \subseteq \mathcal{X}'$ carries the following six-tuple:

$$\begin{aligned}(|X|,\ \sum_{p \in X}(x - x_p),\ \sum_{p \in X}(y - y_p),\ \sum_{p \in X}(x - x_p)^2,\\ \sum_{p \in X}(y - y_p)^2,\ \sum_{p \in X}(x - x_p)(y - y_p)).\end{aligned} \qquad (3)$$

This tuple has a number of nice properties. These are easy to confirm, a full description with proofs can be found in [15]:

- These values are sufficient to compute the aforementioned moments, if (x, y) is known. We assume that after our algorithm finished, a centralized processor will pick up the tuple from the single pixel per object, compute the moments, and uses them for whatever activity the machine was built.

- When an agent moves from (x, y) to an adjacent pixel, it is computationally trivial to update the tuple for the new position. For example, when moving a tuple $(m, s_x, s_y, s_{xx}, s_{yy}, s_{xy})$ to the east, $(m, s_x - m, s_y, s_{xx} - 2s_x + m, s_{yy}, s_{xy} - s_y)$ is the correct tuple for the new position. Addition and subtraction are sufficient for all necessary updates.
- If an agent picks up the weight of the pixel it currently resides on, it simply has to increment the first tuple entry.
- If an agent receives the cumulative weight of another agent on the same pixel, it just computes the component-wise sum of the two tuples.

So all that is left to do is to define a rule set by which agents can sweep over the objects, pick up the weights, and let the weights for each object cumulate in a single pixel. This is what we describe next.

3.2 Agents

Let A denote the set of all agents. An agent $a \in A$ has the following variables in addition to those for calculating moments:

- $C_a \subseteq A$, a set of partners. Partners are agents that belong to the same object, living in an adjacent row. Once agents decide to partner, they move together and stay within a horizontal distance of at most 1. Initially, $C_a = \varnothing$. Partnership will be mutual throughout the algorithm, so one can think of partnership as an undirected graph on the set of agents. This graph is denoted by $\mathcal{C} = (A, C)$.
- h_a, a boolean stating whether the agent has ever left the object it started in. The variable is initialized as false (as agents are always created on object pixels). When the agent moves onto a non-object pixel, it sets $h_a \leftarrow$ true.

We denote the set of all agents that are currently on a pixel p by $A(p) \subseteq A$. We assume that each agent has access to the following information:

- Which pixels of $\overline{\Gamma}(p)$ are object pixels.
- If it is currently on an object pixel, whether the pixel's weight has been consumed already.
- The state of variables $C_{a'}$ and $h_{a'}$ of each agent $a' \in \cup_{p' \in \overline{\Gamma}(p)} A(p')$. We will discuss how to make this information available in Section 3.3 below.

Initially, there is an agent a on every object pixel that has no object pixel to its left, with $h_a =$ false and $C_a = \varnothing$.

Our algorithms runs in iterations. In each iteration, each agent performs the following steps in sync with the other agents, here written from the perspective of an agent a on pixel p:

1. **Consume weight:** If $h_a =$ false and the weight on the pixel has not been consumed yet, consume it (see Section 3.1).
2. **Find partners:** If $h_a =$ false, see if there are any agents $a' \in \cup_{p' \in \Gamma(p): y_{p'} \neq y_p} A(p')$ (i.e., on neighbor pixels in adjacent rows), which also have $h_{a'} =$ false. Add those to C_a.
3. **Pass on cumulative weights:** If there is an agent $a' \in C_a$ in the row below p, pass all collected moment weight to a'. Ties are broken arbitrarily.
4. **Merge agents:** If there is another agent $a' \in A(p)$ (i.e., on the same pixel as a), and a and a' have a common partner, they conclude they belong to the same object. Then, they merge into a new agent a'' with $h_{a''} = h_a \wedge h_{a'}$ and $P_{a''} = P_a \cup P_{a'}$. This is done as a shrinking of $\{a, a'\}$ in $\mathcal{C}$, with the obvious consequences for the partner edges incident to them. They also merge the accumulated weights (see Section 3.1).
5. **Decide whether to stay or go:** The agent decides to move forward, unless any of these conditions hold:
   (a) $C_a \cap A(\mathsf{NW}(p)) \neq \varnothing$ or $C_a \cap A(\mathsf{SW}(p)) \neq \varnothing$ (i.e., it has a partner in the column to the left),
   (b) $\mathsf{N}(p) \in \mathcal{X}$, but $C_a \cap A(\mathsf{N}(p)) = \varnothing$ (i.e., there is an object pixel in the north, but no partner has been found on it yet), or
   (c) $\mathsf{S}(p) \in \mathcal{X}$, but $C_a \cap A(\mathsf{S}(p)) = \varnothing$ (the same as the previous, just for the south).
   (d) $\mathsf{E}(p) \notin \mathcal{X}$ and $C_a \cap A(\mathsf{NE}(p)) = C_a \cap A(\mathsf{SE}(p)) = \varnothing$, i.e., it will not move onto non-object pixels unless it is "dragged" by a partner further to the right.
6. **Stay or go:** If a decided to move in step 5 above, it now transfers itself to $\mathsf{E}(p)$. If $\mathsf{E}(p) \notin \mathcal{X}$, it sets $h_a \leftarrow$ true.

The algorithm ends in the first iteration in which

- No agent could pass cumulative weights in step 3, and
- No agent decided to move in step 5.

The agents are not aware whether they have finished, so this stopping criterion is not reflected in the algorithm. In the application, there is an external controller that will wait $2W + 2H$ rounds, after which the algorithm is guaranteed to be finished, see Section 4.

An example of this algorithm is shown in Figure 2. In iteration #1, the agents are residing on left-most object pixels. They move fast while they are on object pixels. Once they leave it, they have to be "dragged" according to rule 5d by partner agents that are still inside (#26). As agents only merge when they have proof that they belong to the same object (step 4), the sweeps for the letters can cross without affecting each other (#47 and #90). In the end, there is one vertical line for each object, with the cumulated weight waiting for pickup at the bottom-most agent (#259).

3.3 Implementation on Active Pixels

Implementing the algorithm on a grid of smart pixels is straightforward: Each pixel maintains a list of the agents that currently reside on it and evaluates the agent's decisions. Note that an actual synchronization with neighbors is only necessary after steps 4 and 6, all other steps can be performed with already available information.

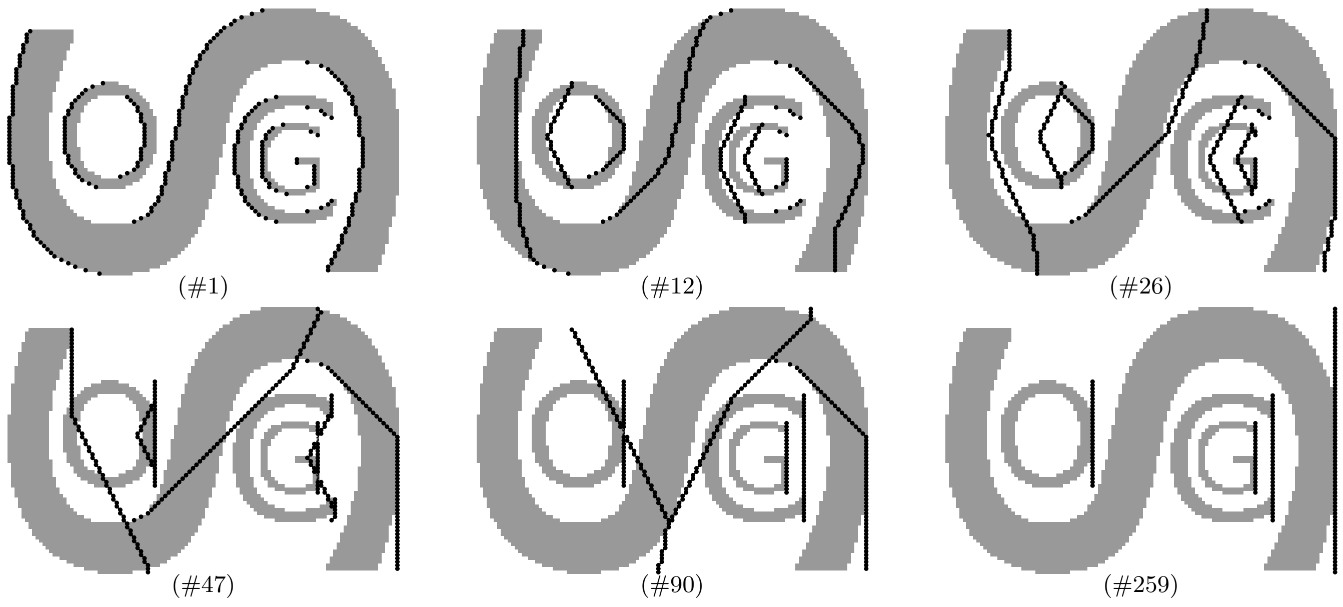


Figure 2: Algorithm running on the SoCG Alphabet Soup.

4. ANALYSIS OF THE ALGORITHM

4.1 Correctness

THEOREM 1 (CORRECTNESS). *If the algorithm stops, the weight of all pixels belonging to the same object is accumulated in a single agent. The agents for this object form a vertical line.*

PROOF. First, note how partnership is established: Initially, no agent has any partners. An agent a will only add another agent a' as a partner in step 2 if neither of them has ever left the object they were created in. Merging (step 4) only happens among agents that are transitive partners. Finally, an agent will never move until it has found partners in both adjacent rows (step 5, conditions (b) and (c)), unless there are none needed (because the adjacent pixels are non-object). So agents stemming from different object never interfere with each other.

Now consider the situation where the agents belonging to an object are not in a vertical line. All agents in the leftmost column have found their partners now. Now either one agent has an object pixel in $\mathsf{E}(p)$, in which case he will move. Otherwise there must be an agent with a partner to the right (in $\mathsf{NE}(p)$ or $\mathsf{SE}(p)$), in which case this agent will move.

So in the end, all the agents form a vertical line. Assuming they did not do that already, they will now establish partnership to become a single connected (w.r.t. partnership) line, and they will pass all the weight of the object pixels down to the bottom-most agent. $\square$

4.2 The Algorithm's Runtime Complexity

Given that agents belonging to different objects do not influence each other, we restrict the following analysis to the case of a single object $\mathcal{X}$. We will look at objects with increasing complexity. We start our proof with *paths*, i.e., objects where each occupied pixel has at most two neighboring object pixels. On a pixel field we may describe these paths by alternating vertical and horizontal lines. We denote the length of the vertical and horizontal parts by H_i+1 (height) and $W_i + 1$ (width), and associate a positive sign with a downwards vertical and a rightwards horizontal line, the path is $(H_1, W_1, H_2, W_2, \ldots, H_V, W_V)$. To be precise, we start with a single pixel. To it, we attach $|H_1|$ pixels downwards (if $H_1 > 0$) or upwards ($H_1 < 0$). Then we attach $|W_1|$ pixels to the path's end, horizontally to the right ($W_1 > 0$) or left ($W_1 < 0$). We repeat this process for H_2, W_2, ..., H_V, W_V.

THEOREM 2 (L-SHAPED PATHS). *Let $\mathcal{X}$ be an L-shaped path (H_1, W_1), see Figure 3. The agent on the horizontal line moves with speed $1/2$, it takes additional time of H to gain a vertical line at the rightmost end.*

PROOF. We have $H_1 + 1 = H, W_1 + 1 = W$. Let a_b be the agent initially located at the bend of the L, see Figure 3. Let the others be $a_1, \ldots, a_k$ (named starting from the agent initially located above a_b). All agents create partnership edges, i.e., $a_b \in C_{a_1}, a_1 \in C_{a_2}, \ldots, a_{k-1} \in C_{a_k}$. In the first iteration a_b has no partner to the left, a partner above and no object below and in moving to the right a_b will not be away from its home object. Thus, a_b moves in the first iteration. The others do not move, as they would be leaving their home object and have no partners in $\mathsf{NE}(p)$ or $\mathsf{SE}(p)$ (see rule 5d). In the second iteration a_b has to wait as its partner a_1 is to the left ($\mathsf{NW}(p)$). With a_b being in $\mathsf{SE}(p)$ for a_1, a_1 is allowed to go now. The other a_i's still wait. So, in the third iteration a_b is again allowed to move, as is a_2 (with a_1 in $\mathsf{SE}(p)$), all other agents wait.

This process goes on, a_b moves in every second iteration, i.e., with speed $1/2$, and the other pixels form a rearwards line with slope 2 (at any point in time two agents occupy pixels in the same column). Thus, a_b reaches the right end after $2W - 1$ time units. As all the other agents are also moving with speed $1/2$ it takes additional $H - 1$ time units until the vertical line at the right end is reached. $\square$

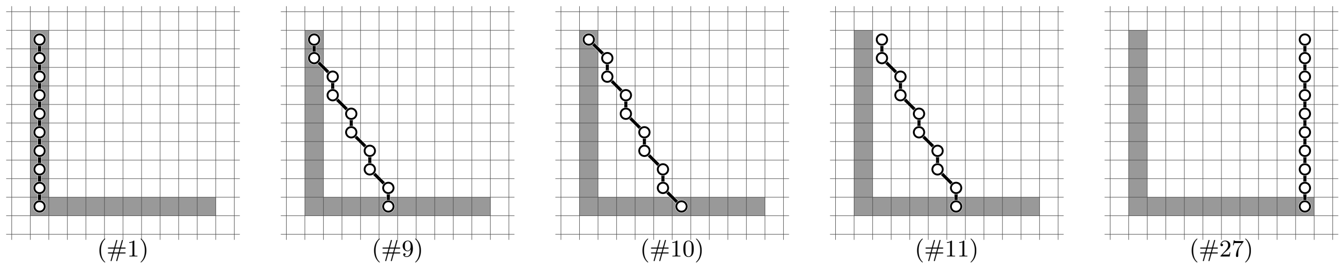


Figure 3: An L-shaped path and the algorithm sweeping over it.

Corollary 1. *Theorem 2 holds for an L-shaped path reflected along the x-axis.*

Theorem 3 (Windy paths of uniform height). *Let $\mathcal{X}$ be an x-monotone path with $W_i > 0, \forall i$, $|H_i| + 1 = H, \forall i$ and alternating signs of the H_i's, see Figure 4. It takes at most $2W + H$ iterations to build up the right vertical line.*

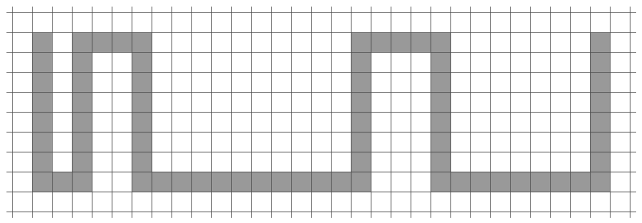


Figure 4: Windy path of uniform height.

Proof. By induction on the number of V vertical lines. The case $V = 1$ is covered by Theorem 2. Assuming the theorem is correct for V, we now show that it also holds for $V + 1$:

$\mathcal{X}$ can be covered by overlapping L-shaped paths, (H_1, W_1), $(H_2, W_2), \ldots, (H_V, W_V)$. Agents located in a pixel above or below such an overlap have to wait for an agent to arrive from the left before they can move, due to rules 5b and 5c. If the waiting time is long enough $(2H-2)$ this results in agents standing in a line with slope 1 (see Figure 5). Whenever the agents from the left arrive these lines of merged agents are allowed to move and do so with speed 1/2 (as there is always one agent located on a horizontal line, allowed to go whenever its partner caught up, thereby dragging the rest).

Now let us the consider the situation in the first piece (H_1, W_1). No agent initially located on the vertical line added at the left has to wait for some agents approaching from the left. Thus, the agent initially located at the bend of the L, see Theorem 2, walks with speed 1/2 towards the next vertical line (arriving at iteration $2W_1 - 1$). The agents from the first vertical line approach the second in a line of slope 2, after iteration $2W_1 - 1$, as seen by the following two cases:

1. $W_1 < H/2$: Some agents are still located on the first vertical line. When the first agent reaches the second vertical line the agent with which will merge is not yet allowed to move, see the #5 in Figure 6. The slants move (horizontally) through the L-shaped parts, as seen in #6 and #7 of Figure 6.

   At some point the merged agents are allowed to move. This happens after $W_1 + H/2$ iterations. Thus, now all agents of the first vertical line keep on moving until they reach the final position. This happens in iteration $(2W_1 + H) + W_1 + H/2 < 2W_{V+1} + H + 2W_1 = 2W + H$.
2. $W_1 \geq H/2$: It may take additional time of $H^- \leq H$ until the merged agents are allowed to move, with the H_1 agents from the first vertical line standing in a vertical line again (analog to $W_L < H/2$). The agents keep on moving, collecting (by merging) all the others (time $2W_{V+1}$) and finally a time of $\max(H^-, H - H^-)$ is needed to gain the vertical line. Thus, we have $2W_1 + 2W_{V+1} + H^- + \max(H^-, H - H^-) \leq 2(W_1 + W_{V+1}) + H = 2W + H$.

Together, the two cases prove the claim. □

Theorem 4 (Windy paths of arbitrary heights). *Let $\mathcal{X}$ be an x-monotone path with $W_i > 0 \forall i$ and alternating signs of the H_i's, see Figure 7. It takes at most $2W + H$ iterations to build up the right vertical line.*

Proof. To prove the runtime we modify the $\mathcal{X}$, and show that our algorithm is not faster on the modified object $\mathcal{X}'$, yet still runs in at most $2W + H$ on $\mathcal{X}'$.

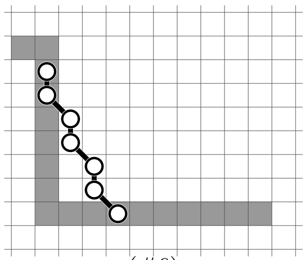
The object $\mathcal{X}'$ consists of all vertical lines of $\mathcal{X}$ extended to the length of H and the horizontal lines shifted to the lowest resp. highest position in the bounding box, see Figure 7. We claim that our algorithm is not faster on $\mathcal{X}'$ than on $\mathcal{X}$. Then, the claimed runtime of $2W + H$ results from Theorem 3. We prove the claim in three steps:

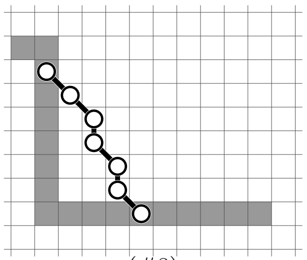
Claim 1: *The time the agents need to reach the second vertical line from the first does only depend on W_1.* The agents starting from the first vertical line need not wait for agents approaching from the left. Theorem 2 shows that the first agent reaches the next vertical line after $2W_1 - 1$, independent on the height H_1.

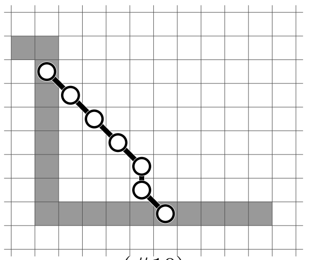
Claim 2: *For agents initially located on vertical lines other than the first an increased H_i may cause some agents to be able to move further to the right. Nevertheless, at least one agent stays on the line and is not picked up earlier.* The first part follows from the fact that a bigger H_i, determining the “leash length” of the agent moving on the horizontal line (with the waiting pixel located above or below an overlap of L-shaped parts holding the leash), gives this moving agent more freedom. The other agent has to wait (see Figure 5).

As the agents of the first vertical line keep on moving with speed 1/2 it is picked up “in time” (and when moving to the right a line with slope at least 2 is reestablished).

Claim 3: *For a smaller height, H_i, the merging point enabling the agents initially located on the next vertical line to move, is not reached later. That is, with a bigger H_i the waiting rule 5d is not resolved earlier.* We need to distinguish two cases:







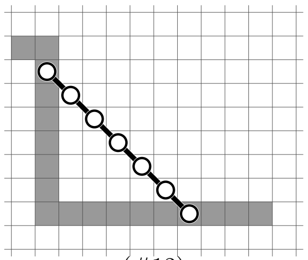
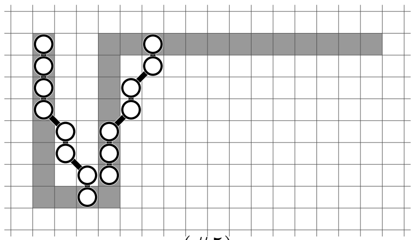
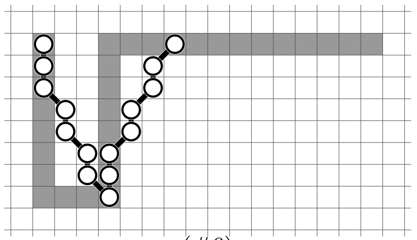


Figure 5: Agents initially located on an L-shaped path with a waiting agent form a line of slope at least 1.





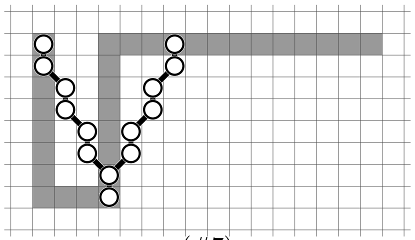


Figure 6: Case $W_L < H/2$ in the proof of Theorem 3.

1. In case the first vertical line is longer than (or equal to) the second ($|H_i| \geq |H_{i+1}|$), the time until merged agents are allowed to move depends on the height: the slant moves (horizontally) through the L-shaped parts—this takes longer for a bigger height.
2. In case the first vertical line is shorter than the second ($|H_i| < |H_{i+1}|$), the "new" agents added parallel to agents from the $(i+1)$th vertical line may overstep the second vertical line before some agents initially located in the second vertical line are enabled to move. Nevertheless, the pulling agent is still one of the second vertical line and consequently not to the left of the "new" ones. The slant keeps on moving horizontally through the L-shaped path, enabling the agents of the two vertical lines to move at the same time as in $\mathcal{X}$.

The claim now follows from Theorem 3. □

Theorem 5 (X-monotone paths). *Let $\mathcal{X}$ be an X-monotone path with $W_i > 0, \forall i$, see Figure 8. It takes at most time $2W + H$ to build up the right vertical line.*

Proof. Again, we construct an object $\mathcal{X}'$ and show that the algorithm is not faster on $\mathcal{X}'$ than on $\mathcal{X}$. Consider the sign of the vertical lines (i.e., whether the next horizontal line is situated below or above the last one). Swing out all stairs with a row of vertical lines of the same sign, see Figure 8. Hence, $\mathcal{X}'$ is an object from Theorem 4, and the running time is at most $2W + H$.

We claim that merged agents are not allowed to move earlier on $\mathcal{X}'$. Let the vertical lines of the first staircase be $\ell_1, \ldots, \ell_r$, from left to right. We compare it to the corresponding L-shaped path (with height $(\sum_{i=1}^{r} H_i) + 1$ and width $(\sum_{i=1}^{r} W_i) + 1$).

When the agents of $\mathcal{X}'$ gain a line of slope 2 for the first time, the agents from $\ell_1, \ldots, \ell_r$ are at most in lines with slope 2 (as they are shorter), some agents may have gone further to the right, resulting in lines with slopes not less than 1.

The agents of ℓ_1 reach the merge point after $2W_{\ell_1} - 1$. At this time we have a parallel line in $\mathcal{X}'$ with all agents to the left of the ones from $\ell_1, \ldots, \ell_r$. □

Theorem 6 (Arbitrary paths). *Let $\mathcal{X}$ be any path, see Figure 9. It takes at most $2W + H$ iterations to build up the right vertical line.*

Proof. We construct a set $\mathcal{X}'_1, \ldots, \mathcal{X}'_z$ of objects (paths) by splitting the vertical lines of $\mathcal{X}$ at the rightmost position of the bounding box, if these exist, and projecting the maximal height H to the left of each part. Note that this may result in circles. Then, we take the longest of the paths $\mathcal{X}'_1, \ldots, \mathcal{X}'_z$ (in terms of time), let this path with the projection on the left be $\mathcal{X}'$. We claim that $\mathcal{X}'$ is not processed faster than $\mathcal{X}$. We prove the claim in two steps:

Claim 1: *The height stays the same.* Is obvious.

Claim 1: *The new created agents will not pick up agents to the right earlier than these will be enabled to move in $\mathcal{X}$.* We add a vertical line, i.e., the agents have to be pulled by an agent (or more) on an existing horizontal line. This agent keeps its speed and is never to the left of the added agents before they potentially merge with "old" agents: When they merge with "old" agents the merged agents inherit the conditions for both.

Hence, if they had to wait they still do, but as waiting for agents from the left they will be enabled to move.

Thus, the new added agents on the first (leftmost) vertical line will not influence on the time "old" agents from the leftmost end need to reach the right most end ($2W + H^*, H^* \leq H$). They are at least pulled and may then build a vertical line (time $H - H^*$), resulting in running time of at most $2W + H$. □

Theorem 7 (General case). *Let $\mathcal{X}$ be the given object. The running time of the algorithm is at most $2W + H$.*

Proof. We may consider $\mathcal{X}$ as a graph $G_{\mathcal{X}}$: $\mathcal{X}$ occupies pixels, the nodes, and neighboring (4-neighborhood) pixels

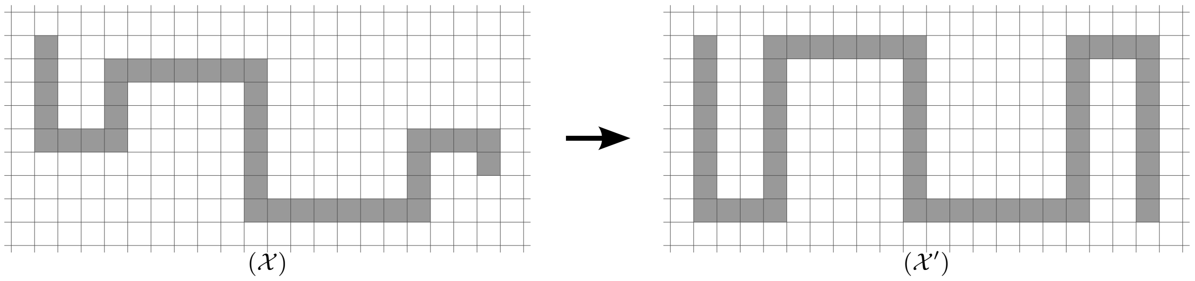


Figure 7: Windy path $\mathcal{X}$ and reduction $\mathcal{X}'$ to the previous case.

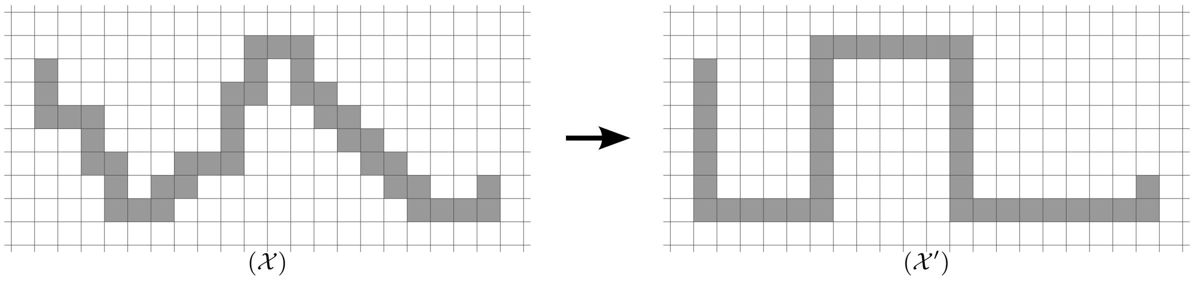


Figure 8: X-monotone path $\mathcal{X}$ and its reduction $\mathcal{X}'$.

are adjacent, see Figure 10. We prove the running time in four steps:

Claim 1: *Walking inside an object is faster than walking outside of an object.* Is obvious.

Claim 2: *Reducing $G_{\mathcal{X}}$ to its leftmost edges whenever vertical edges are parallel (and deleting the horizontal edges in these areas, preserving connecting edges) we gain a tree, $T_{\mathcal{X}}$.* Is obvious.

Claim 3: *The algorithm does not run faster on $T_{\mathcal{X}}$ than on $G_{\mathcal{X}}$.* Results from (1) and (2).

Claim 4: *The running time on $T_{\mathcal{X}}$ is at most $2W + H$.* In $T_{\mathcal{X}}$ we consider all paths that end on the right. Then, we take the longest (in terms of time) of these, let it be P (decide on P from left to right at branching points).

Analog to the proceeding in the proof of Theorem 6 we project the maximal height H to the left of P.

We do not shorten the running time of "old" agents on P from the left to the right. Arguments like in the proof of Theorem 6 yield the $2W + H$. □

4.3 A Worst-Case Example

In the previous analysis, we used the $\mathcal{LOCAL}$ model [17], in which a communication round is sufficient for a pixel to perform the computations and communication for all agents currently stored on it.

Figure 11 shows a worst-case example for the algorithm. It can be generalized to larger sizes, in which some pixels have $\Omega(W)$ agents residing simultaneously on them. As we have never seen more than three agents on a pixel for any realistic input, we consider this issue irrelevant for practical applications.

Note that there is a trivial upper bound of $\lceil W/2 \rceil$ on the maximal number of agents on a pixel, as agents move only horizontally and cannot be generated next to each other. So, in the more restricted $\mathcal{CONGEST}$ model [17], where a communication round only admits messages of size $O(\log(WH))$, our algorithm has a runtime complexity of $O(W(W + H))$.

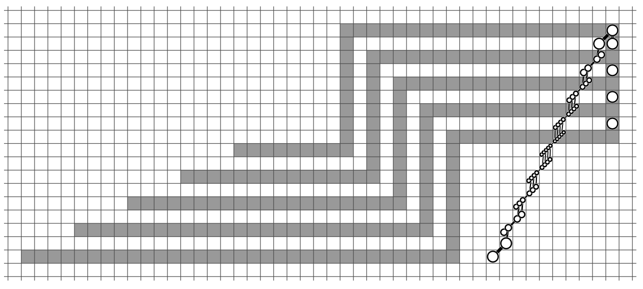


Figure 11: Worst-case on agents per pixel.

However, we believe that this model does not reflect the platform for which the algorithm was developed, as it was specifically designed so that every cell's memory can be transferred to a neighbor in a single round.

5. IMPLEMENTATION

In the previous section, we have proven that our algorithm is highly efficient—in theory. To see how applicable our approach is, we implemented it for actual hardware in VHDL.

A substantial part of this implementation deals with turning the system model (active pixels with local memory, organized in a grid) into hardware. Each pixel becomes a processing element (PE) on the final chip. Each of these PEs consists of a control unit (to steer the propagation), local memory, arithmetic units, and connectivity to its neighbors. We implemented the algorithm with a four-pixel neighborhood (as opposed to eight in the previous descriptions) in order to reuse a design of a former algorithm [15]. To mimic the functionality of an eight-pixel neighborhood, we implemented some additional flags to store information about diagonal neighbors. The advantage of using a four-pixel neighborhood network is that we can stint some connection lines.

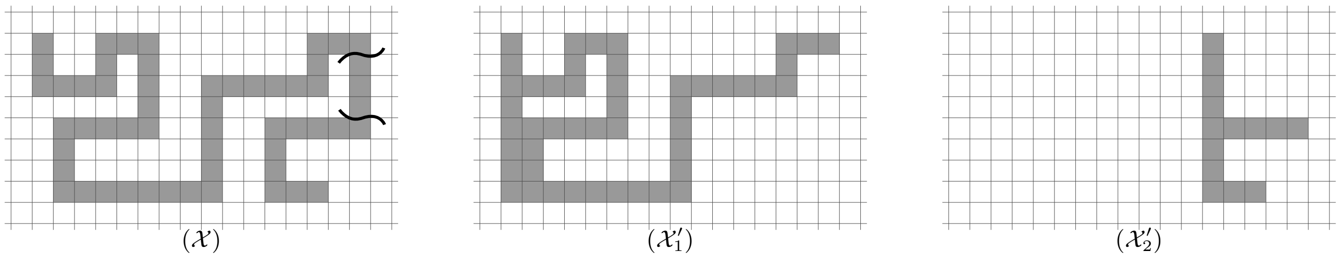


Figure 9: Arbitrary path $\mathcal{X}$ being transformed into simpler pieces $\mathcal{X}'_1$ and $\mathcal{X}'_2$.

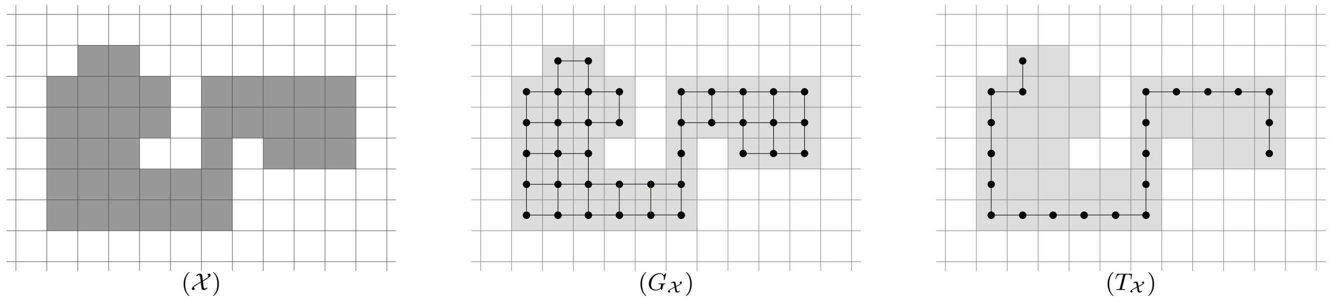


Figure 10: Reducing the general case to analyzing the most complex subpath.

State	Description
S0	Initial State
S1	Edge detected Waiting to find neighbours for first time
S2	Walking
S3	Waiting for neighbors
S4	Agent moves from West Updating registers

Table 1: Description of states

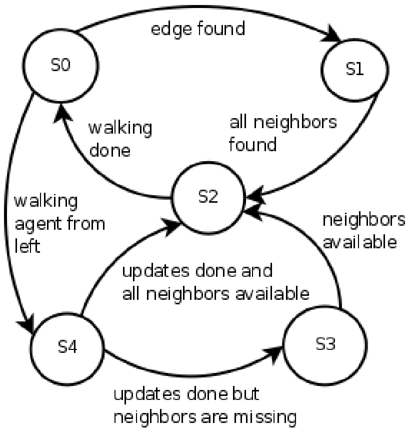


Figure 12: State machine for the control unit

Picture Size	Slice Register (%)	Slice Lut (%)	Lut FF Pairs (%)	f_{max} $[MHz]$
4x4	1098 (0)	5527 (4)	747 (12)	159
8x8	4442 (3)	24593 (20)	3032 (11)	183
16x16	17554 (14)	95054 (77)	12162 (12)	113

Table 2: Results of Virtex5-FPGA synthesis.

Each PE's control unit is implemented as a finite state machine (see Figure 12 and Table 1). State *S0* is the initial state for each pixel. If a pixel is at the left edge of an object, a new agent is born and the state is changed to *S1*. In state *S1*, the agent waits for the first arrival of a neighbor. After finding its neighbors, the agent's state switches to *S2*, allowing the agent to walk. When the agent moves, it leaves the pixel position, which is thus returned to state *S0*. If a pixel p observes an agent with state *S2* in $\mathsf{W}(p)$, it moves that agent onto itself and sets its state to *S4*. This initiates the phase after walking, where register updates are done. Afterwards, the agent has two options. If all previous partners are still visible, it moves again (i.e., switches to state *S2*). If some partner has not moved, and is thus not in the four-pixel neighborhood, it waits (by switching to state *S3*).

An important issue of the algorithm is that several agents can occupy the same pixel. The maximal number is in $\Theta(W)$, which cannot be implemented as is. For practical purposes, we found that a small constant number of agent memory slots in each pixel are fully sufficient. In this sample implementation, we decided to host at most three agents at a time. We simulated the algorithm in ModelSim and tested it successfully. After testing, we synthesized the design for two technologies: an FPGA and an ASIC. For the FPGA, we decided to model the memory of the pixel positions in BRAMS in order to reduce the slice requirements. The results of FPGA synthesis can be seen in Table 2. It shows that a size of 16×16 is easily achievable in reconfigurable hardware together with reasonable clock rates.

ASIC synthesis was executed for a 180nm UMC CMOS process. Because of the large processor and memory usage, we only synthesized a resolution of 8×8. For this size, the complete vision chip required $1.84mm^2$ and consumed $51.5mW$. Due to the linear scalability of this architecture, it is possible to calculate values for higher resolutions, too. For example, an industrial resolution of 128x128 pixels results in a chip with an area of $2.1cm \times 2.1cm$ and $14.5W$ power consumption. A more modern 90nm process technology would only require an area of $130mm^2$ for the same resolution.

To summarize these results, we are convinced that our algorithm can be turned into ASICs even for image sizes that are typical for industrial applications. The low power consumption and high chip frequency indicate that industrially manufactured chip would indeed outperform the current solutions with a classic, centralized design.

6. CONCLUSION

We have presented a novel sweepline algorithm for the computation of a Principal Component Analysis on a smart pixel grid. Our algorithm is localized and achieves a distributed time complexity of $O(W + H)$ for objects on a grid of $W \times H$ pixels. This is particularly striking as there may be $\Omega(WH)$ object pixels that need to be processed. Besides possessing an unsurpassed speed, our algorithm is simple enough to be turned into a hardware design for smart pixel chips. We reported on synthesis results for FPGAs and ASICs, showing how our algorithmic contribution makes smart pixels a strong competitor for current high-end industrial image processing applications.

In our opinion, the biggest issue with our algorithm is that several agents can accumulate in a single pixel. This rarely happens with real-world data, and it can be easily resolved by retrying the analysis a few milliseconds later, when the belt has moved slightly. Our future goal is to improve the theoretical aspects of this issue.

Interesting future directions lie in extending the algorithm to detect high-level properties of the objects, such as shape, topological type, thickness, and more. Even topics like distributed optical character recognition (OCR) are now within reach. Quite clearly, this offers a vast spectrum of exciting possibilities—both from a practical point of view, as well as from the theoretical challenge of understanding the fundamental philosophical issue how local computation based on very limited information can lead to methods for computing higher-level concepts.

Acknowledgements

Christiane Schmidt was supported by DFG grant Fe407/14-1, project "RoboRithmics". Marcus Komann was supported by DFG project "Marching Pixels".

7. REFERENCES

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[2] B. Awerbuch. Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems. In *STOC '87: Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, pages 230–240, New York, NY, USA, 1987. ACM.

[3] B. Awerbuch, B. Berger, L. Cowen, and D. Peleg. Fast distributed network decompositions and covers. *Journal of Parallel and Distributed Computing*, 39:105–114, 1996.

[4] D. H. Ballard and C. M. Brown. *Computer Vision.* Prentice Hall, 1982.

[5] C. Chaplin. Modern times, 1936.

[6] B. Chazelle. Triangulating a simple polygon in linear time. *Discrete & Computational Geometry*, 6:485–524, 1991.

[7] B. Chazelle. Who says you have to look at the input? the brave new world of sublinear computing. In *SODA '04: Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, page 141, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[8] B. Chazelle, D. Liu, and A. Magen. Sublinear geometric algorithms. *SIAM J. Comput.*, 35(3):627–646, 2005.

[9] M. Elkin. A faster distributed protocol for constructing a minimum spanning tree. *J. of Computer and System Sciences*, 72(8):1282 – 1308, 2006.

[10] A. Ferencz, E. G. Learned-Miller, and J. Malik. Learning hyper-features for visual identification. In *Neural Information Processing Systems*, 2004.

[11] D. Fey, L. Hoppe, A. Loos, M. Förtsch, and H. Zimmermann. Parallel optical interconnects with mixed-signal OEIC and fibre arrays for high-speed communication. In *Proceedings of SPIE*, volume 5453 of *Micro-Optics, VCSELs and Photonic Interconnects*, Strasbourg, France, 2004. Photonics Europe.

[12] R. Goldstein. Herb score, pitcher derailed by line drive, dies at 75. *International Herald Tribune*, Nov 12, 2008.

[13] R. C. Gonzalez and R. E. Woods. *Digital image processing.* Prentice-Hall, third edition, 2008.

[14] I. T. Jolliffe. *Principal Component Analysis.* Springer, second edition, 2002.

[15] M. Komann, A. Kröller, C. Schmidt, D. Fey, and S. P. Fekete. Emergent algorithms for centroid and orientation detection in high-performance embedded cameras. In *CF '08: Proceedings of the 2008 conference on Computing frontiers*, pages 221–230, New York, NY, USA, 2008. ACM.

[16] D. Marr. *Vision: A computational investigation into the human representation and processing of visual information.* Freeman, 14th edition, 2000.

[17] D. Peleg. *Distributed computing: a locality-sensitive approach.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[18] G. Ryle. *The concept of mind.* University of Chicago Press, 1949.

[19] J. Shlens. A tutorial on principal component analysis, 2005. http://www.snl.salk.edu/~shlens/pub/notes/pca.pdf.

[20] B. Steckemetz. Quality control of ready-made food. In *DAGM-Symposium*, pages 153–159, 1995.

[21] M. E. Wall, A. Rechtsteiner, and L. M. Rocha. Singular value decomposition and principal component analysis. In Daniel P. Berrar, Werner Dubitzky, and Martin Granzow, editors, *A Practical Approach to Microarray Data Analysis*, pages 91–109. Springer, 2003.