Parallelizing OCR

Eduardo Montesinos

University of Karlsruhe
Informatics for Engineering and Science. Am Fasanengarten 5
W-7500 Karlsruhe 1, Germany

Abstract

This paper describes how to parallelize Optical Character Recognition (OCR) in EFECT, a parallel dataflow-oriented graphical programming language. The first part describes an extraction algorithm for relevant features from isolated character-patterns. The second part implements a neural network, which identifies the patterns by the extracted features. A short description of the EFECT language is included. The EFECT program is graphically represented, like the EFECT editor (Efe) does. Finally, the output of some test runs on a personal computer will be discussed.

1 Introduction

Computers have become an essential tool for many applications. Enormous amounts of data are being processed and stored every day. One application is to store and process text, that previously has to be fed into the computer. To avoid the tedious, manual input, algorithms have been developed to enable the computer to do this automatically. The first step is to read in the raw data with a scanner, resulting in an array of pixels. Then the computer has to find and recognize the text. This is done with OCR-software.

Optical Character Recognition (OCR) is already being used for automatically reading printed and handwritten text today by mailing companies and in reading facilities for blind people.

How OCR-systems work (fig. 1.1):

1) While learning, a sample text is present in the raw and the correctly (manually) interpreted format. For each symbol in this text, the computer extracts relevant features, with which a knowledgebase is built.

2) The recognition phase works as follows. From the raw data, the possible words and symbols have to be isolated. The symbols are transformed into the computer's character-code, using the previously generated knowledgebase. Finally, the interpreted characters are re-assembled into words, forming the text, as the computer saw it.

Fig. 1.1: How OCR-systems work

OCR-systems are judged on two criteria: speed and reliability. To obtain both, a simple, reliable and fast feature extraction method has been chosen to be executed in a parallel environment. As the EFECT environment has not been completed yet, the algorithms have been implemented and tested on a personal computer, ignoring the speed up when using a parallel architecture.

The main features of this paper are:
- Feature extraction
- Storing knowledge in a neural network
- Implementing a parallel learning and recognition-algorithm in EFECT

Isolating words and symbols from the raw data will only be roughly described.

Finally, some test runs and their results are presented and discussed. Lacking a scanner, the raw data was produced, using a pixel-oriented painting program. The character isolation was done interactively.

2 Feature extraction

The program uses a feature extraction method described in [3], which has been extended in order to improve its reliability.

The feature extraction generates certain characteristic information from the smallest possible pixel matrix containing one single symbol. If these matrices are more
than twice as high as wide, they will be expanded in width. The extracted features should be both different for differing symbols and similar for similar symbols, in order not to overreact on differences in style of writing. During the recognition phase, a valuation function will use the extracted information to identify symbols.

Features are extracted by dividing the pixel matrix into \( s \) lines and \( s \) columns. For each line and column, the percentage of foreground pixels is determined, resulting in \( 2s \) numbers. Every number is divided by the largest number in the set, resulting in a characteristic vector of normalized coefficients, which can be seen as the discrete distribution functions \( V(x) \) and \( V(y) \) (fig. 2.1). Without the normalization, identical symbols with differing linethickness would generate quite different characteristic vectors.

Scanned pictures are usually not of perfect quality, because of hardware limitations or a bad original. Fig. 2.2 shows a symbol with a spot and the corresponding distribution functions of the symbol with and without spot. Obviously, some of the two functions' coefficients differ more than they should. In order to minimize the local impact of spots on the distribution functions, the coefficients are transformed, using the well known Walsh-transformation, which gives good results and is very easy and fast to compute. To use the implemented Walsh-transformation, the number of lines and columns must be a power of 2.

The discrete transformation of the normalized coefficients \( \{ V(x) \} \) into Walsh-coefficients \( \{ w_r \} \) leads to:

\[
\sum_{j=1}^{2^k} V(x_j) \cdot f_r(x_j) \cdot \frac{1}{2^k} = w_r \\
x_j = \frac{j}{2^k} \\
k \leq \lfloor N \rfloor
\]

Several tests have shown, that a segmentation of 4 lines and 4 columns per sub-matrix gives the best results. Using 2 lines and columns hides important details, while using 8 or more overvalues differences in style.

To limit the amount to be processed by the testprogram, only capital letters were considered. Problems arise when...
processing letters, that look similar in both cases, like 'C' and 'c', 'O' and 'o' or 'P' and 'p'. These problems could be mastered by introducing an extra feature like the relation between the height of the line and distance between the top of the character and the top of the line (fig. 2.7).

Fig. 2.7: How to distinguish between capital letters and lower case letters

Ending this chapter, two different OCR methods will be compared to the implemented method (an overview over more OCR methods can be found in [1] and [2]).

1) Bitmap comparison bitwise compares several reference-patterns with the input. If enough pixels are identical, both patterns are considered to be the same. This fast method fails on changing font size or type.

2) Topological methods classify patterns by analyzing their geometry. Features like curves, edges, crossings, lines, etc. are extracted to identify symbols. Unfortunately, this can only be done, when the input has thoroughly been massaged by several time absorbing digital filters, like noise reduction and line thinning.

The implemented method has neither deficiencies. It adapts to changing font sizes and types, and it does not need preprocessing of the input. To demonstrate this, the used examples are handwritten letters of varying size, sometimes polluted with random pixels.

3 The neural network

Neural networks remember facts without the need for explicit databases. As no transport of data has to occur, processing speed improves.

From the many different kinds of neural networks (like back propagation), a variant of competitive learning was chosen. Fig. 3.1 shows the structure of the used neural network.

The network consists of three layers. The input-layer contains the input-units. The hidden layer, which is not visible to the user, contains clusters of units, in which the knowledge about the symbols to be recognized is stored. The output-layer contains one cluster of several output-units.

Each input-unit simply outputs its unprocessed input to every unit in the hidden layer. The units in the hidden layer compute an output-value from both the input-values and their internal state (Fig. 3.2). The clusters select the lowest output-value from their units as their own output-value, which is the input-value for their output-unit.

This implemented variation concerns the computation of the units’ output and the recalculation of the units’ inner state during the learning phase. As the computation as described in [5] during the recognition phase did not give usable results, a new computation method with better results was developed and implemented.

The new computation of the units’ output:

\[ s_{e_1} = \sum_{j=1}^{n} (c_j - a_{1,j})^2 - \text{bias}_1 \]

The new computation of the new inner state of the winning unit:

\[ c_{c_1} := \text{counter}_1 \div \text{div} 5 + 1 \]
\[ a_{1,j} := \frac{c_{c_1}}{c_{c_1} + 1} \cdot a_{1,j} + \frac{1}{c_{c_1} + 1} \cdot c_j \]

(div: integer division)

Before the neural network can be used for recognizing symbols, it has to ‘learn’ about the symbols it has to be able to distinguish between. To accomplish this, patterns
from a reference-text are used for extracting the needed coefficients. The neural network is initialized as follows: the $a_{ij}$ are set to random values between $-1$ and $1$, the internal counter $i$ is set to 1 and the bias $j$ is set to 0. Each symbol to be recognized is assigned to a single cluster in the hidden layer. To train a cluster for recognizing a symbol, its symbol's reference-coefficients are used as the input for the cluster's units. To make the network more flexible, several representations of the same symbol should alternatively be used as input. Within the cluster, the unit to produce the smallest output $m_i$ is the winning unit. The winning units' $a_{ij}$ are modified, so that the output will be even smaller, when the same coefficients are used as input. Also, the internal counter $i$ is incremented by 1 and the bias is decreased by a small value and set to 0 if the result was negative. The losing units' biases are increased by a small value, so that they are more likely to win next time. The presentation of the different representations for a symbol should be repeated, until all units have won about 100 times.

For understanding what the neural network is trying to do, it is handy to imagine an $n$-dimensional space ($Sp_n$), in which the Walsh-coefficients are $n$-dimensional vertices ($wcv$) and the units in the clusters in the hidden layer are $n$-dimensional spheres. Sphere $i$ has a center, which is the vector $(a_{i1}, a_{i2}, \ldots, a_{in})$ and a radius, which is the unit's bias. The unit's output is the square of the minimal distance between the unit's surface and the Walsh-coefficients $wcv$. So, the unit nearest to the $wcv$ is the winner during the learning phase and will be moved towards this $wcv$. As patterns with small differences will produce $wcv$ with small differences, i.e. a small distance apart in $Sp_n$, a small amount of units can perform well enough for recognizing the right symbol. This decreases the needed amount of memory and computation time. Fig. 3.3 shows how the units should find the best positions in $Sp_n$.

[Diagram of moving units]

The bias is essential for effective learning. Fig. 3.4 shows a sample learning phase of two units and a given set of $wcv$s without using a bias, starting from a randomly distributed set of two units (3.4a). Once a unit has moved near enough a set of closely related $wcv$s (3.4b), the other unit will never win and so, the result (3.4c) is very disappointing.

Fig. 3.5 shows the same initial situation for learning (3.5a), only that now the bias is used. First, only one unit moves towards the group of $wcv$s. As soon as the first unit does not move fast enough towards a common goal anymore, the other unit's bias takes over (3.5b and 3.5c) and makes it win (3.5d). Now, the bias must decrease, as its chance for winning again has increased anyway. The result of this kind of learning is quite pleasing (3.5e).

[Figures 3.4 and 3.5 showing learning with and without bias]

Like mentioned above, the number of units per cluster is not fixed to the number of $wcv$s used for learning. When using more units per cluster than there are $wcv$s, no advantages can be achieved and more memory will be needed for longer computations. When using too few units, errors will occur, like is demonstrated in fig. 3.6, which shows a sample network after the learning phase. The units for symbol 'A' has taken a good position for it's $wcv$s during the learning phase. Unfortunately, the 'A' unit is closer to the 'B' $wcv$ at position (1) than the 'B' unit, which has a good position, when only considering the 'B' $wcv$s. On the other hand, the 'A' $wcv$ at position (2) will be recognized as being a 'B' symbol. Basically, one can say that there should be at least as many units per cluster as there are different font types. Fig. 3.6 shows $wcv$s for two font types for symbols 'A'

[Figures 3.6 showing problems concerning too few units]

49
and B, which can be noticed by the two groups of very similar wcv's for each symbol.

Annotation:

Neural networks can become quite complex. The used one is very simple, to allow easy parallelisation. All clusters in the hidden layer can work independently at all times, and all units in the hidden layer can work independently during the recognition phase.

4 Parallelisation in EFECT

This chapter shows how the demonstrated OCR-system could be implemented in a graphical, dataflow-oriented, parallel programming language, called EFECT (extended dataflow chart). The shown graphical representation of the program's parts look like the layout of the final version of the EFECT editor. Now, a brief introduction to the programming language and its symbol is given. Detailed information about EFECT can be found in [6] and about Efe in [7].

4.1 Description of the EFECT language

EFECT is a dataflow-oriented language, the operations of which are larger and implemented in a sequential programming language. For that reason, EFECT can be seen as a macro-dataflow language.

As the dataflow paradigm states, operations, i.e., functions, are executable only when all their incoming parameters are valid. The main characteristic feature of the EFECT language are predefined functions with different operational behaviour. These are:

- general functions
- persistent functions
- conditional branches
- pipeline operations
- tunnels

and functions containing only structural information

- hierarchies
- recursions

General and persistent functions have input and output parameters of fixed datatype. These functions trigger, when all input data is valid and release one dataset on each output on termination. Persistent functions may have an internal state, which is remembered between subsequent calls.

A special kind of function is the tunnel. Tunnels only have one input and one output. They implement write-once read-multiple variables. In a parallel program, a tunnel can be written once and used in different places in the program. The dataset put into the tunnel is transferred to all tunnel-outputs as often as requested.

The usual conditional statement is realized by a function with one input and several outputs. The function has readonly access to the data on the input. In dependence of the contents of the input, the conditional branch decides about the further direction of that dataset. The dataset leaves the function unchanged through one of its outputs. The conditional branch implements a data-dependent splitting of the dataflow. It substitutes the decider, switcher and merger of the usual dataflow scheme. With that, the control-dataflow has become superfluous, which makes reading and understanding of the program more intuitive.

Generators and collectors are part of the pipeline-concept. The generator has exactly one output and the collector has exactly one input. During execution, the generator successively produces datasets on the output, which is terminated by an end-token. The collector collects the generated datasets and triggers on the end-token. The generated datasets are processed by the functions between the generator and the collector in a pipelined manner.

Hierarchies and recursions are references to a subgraph of the whole dataflow program. If a function's subgraph contains the function itself, a recursion is specified.

4.2 The learning phase in EFECT

First, the function gen-list is called, which generates a list of wcv's from a scanned text. This list (like fig. 4.1) is used as input for the function learn (fig. 4.2). This list is recursively divided (divide and conquer, D&C) into two sublists of equal length (function split-list), until both sublists contain wcv's for one symbol (branch single_char). As clusters work independently from one another, the learning for each cluster (function teach_NN) can be done concurrently. After the learning, the internal states of all units in the clusters, are merged into a list (function merge_list) and saved in a file for future use.

![Fig. 4.1: list of coefficients](image)

F — O — P —
coeffs coeffs coeffs
coeffs coeffs coeffs
coeffs coeffs coeffs

Fig. 4.1: list of coefficients
4.3 The recognition phase in EFFECT

The list of internal states of the units in the hidden layer, saved by the learning program, is accessible through the readonly tunnel set_\text{NN}, which is initialized by the persistent operation load_\text{NN}, which is part of the operation init which is called by the runtime system before \text{main} is called. Tunnels avoid extra passing of constant data (see also [6]). The operation main loads raw data, e.g., a pixelmatrix read by a scanner, transforms it into readable text and saves it for further usage.

Operation doc_\text{op} (fig. 4.3) consists of a hierarchy of splitting operations, merging operations and an operation to recognize a single symbol, together forming a set of nested D\&Q algorithms.

The pixel matrix is split into two parts of roughly the same size (split-doc), until the pixel matrices contain single lines (branch one-line, as shown in fig. 4.4). After translating the pixel lines into text lines in line_\text{op}, the text lines are merged (line-merge) into the requested text file. The operation line_\text{op} more or less does the same as doc_\text{op}, splitting lines into words (split-line). To finish up, line_\text{op} calls word_\text{op} as its simple-solve operation.

Operation word_\text{op}, which is the innermost D\&Q algorithm, splits strings without whitespace into substrings (split-word), until the substrings consist of single symbols (branch one-char). The simple-solve for word_\text{op} consists of computing the Walsh-coefficients in calc_coeffs and recognizing the symbol in char_\text{op}, which is the neural network. After that, symbols are merged to strings, which are merged into words back in word_\text{op} and merged into lines back in line_\text{op} and finally merged into the completed text back in doc_\text{op}.

Operation char_\text{op} (fig. 4.5), the neural network, consists of three operations, one for each layer. The input layer (generator to_clusters) distributes the wcv to the clusters through a data-stream, thus enabling the parallel computation of the clusters. The clusters' results are returned in a data-stream. The output layer ( collector min_cluster) absorbs the data-stream and selects the minimal element in the data-stream and outputs the character, which corresponds to the selected minimal element.
5 Conclusions

At this point, some results obtained by running the OCR-system on a personal computer will be presented. The table (fig. 5.1) shows the error-rate under different conditions. For letting the network with 26 clusters learn, the number of units per cluster and the number of different alphabet-patterns were varied. Afterwards, the OCR-system had to recognize 15 handwritten alphabets (two of them are shown in fig. 5.2), resulting in 390 symbols.

Acknowledgement

The authors received helpful support from Wolfgang Baron, who helped to improve the English of this paper.

References