An SIMD Machine for Low-Level Vision

K. BANERJEE

P. S. SAstry

K. R. RAMAKRISHNAN

and

Y. V. VENKATESH

Computer Vision and Artificial Intelligence Group, Department of Electrical Engineering, Indian Institute of Science, Bangalore 560012, India

ABSTRACT

This paper presents an SIMD machine which has been tuned to execute low-level vision algorithms employing the relaxation labeling paradigm. Novel features of the design include: (1) a communication scheme capable of window accessing under a single instruction; (2) flexible I/O instructions to load overlapped data segments; and (3) data-conditional instructions which can be nested to an arbitrary degree. A time analysis of the stereo correspondence problem, as implemented on a simulated version of the machine using the probabilistic relaxation technique, shows a speed up of almost $N^2$ for an $N \times N$ array of PEs.

1. INTRODUCTION

The main requirement of an image-understanding system is to give a comprehensive and unambiguous description of objects present in the scene by processing the intensity information in the image. This processing proceeds in a hierarchical fashion, starting with the raw intensity values in the image grid and culminating in object descriptions. Each level works on the processed output of the lower levels, the output being a symbolic description of the original image in terms of primitives of appropriate detail [1, 2]. This entire processing can be
conveniently split into two parts [1], namely low-level and high-level vision, depending on the type of data to be processed.

In low-level vision, the data for each processing step is in the form of a 2-D array of either intensities or some other partially processed values. At this level, processing requires uniform and independent treatment of the elements of the data array, using neighbor values. The processing proceeds through obtaining successive descriptions of the image in terms of primitives such as zero crossings, edges, textures, depth stamps, etc. The computations do not make use of any scene-specific or object-oriented knowledge. This is in contrast with high-level vision processing, which needs information other than that contained in scene descriptions fed by low-level vision processes. For example, a task like object recognition would need efficient means to store and search through a data base of known object descriptions.

In low-level vision, the computations involved in describing parts of the image in terms of primitives such as zero crossings can be viewed as a labeling problem [3]. In such a model, each grid point or any suitable collection of grid points would represent an object, which has associated with it a set of interpretations called labels. This, along with some constraints on possible allocation of labels to neighboring objects, could be utilized to provide a coherent description of the image in terms of the symbols chosen. These constraints represent general object world knowledge that is used to overcome ambiguities in interpretations. These constraints are local in that they limit the interpretation of objects based on the interpretation of neighbors. The objective is to use them in a set of local processes to arrive at globally consistent interpretations. This method, termed relaxation labeling, can be used as a model for devising algorithms for many low-level vision problems as well as other image-processing tasks [4–6].

Such algorithms are characterized by large volumes of data, and their solution on sequential von Neumann machines is extremely time consuming. In all these algorithms, processing at a point in the image would involve data present in a small region around that point, and identical processing is done at every point. These processing requirements fit into the SIMD schema described by Flynn [7].

This paper presents a special-purpose SIMD machine which can efficiently implement low-level vision algorithms that employ probabilistic relaxation labeling (PRL) techniques. However, the design allows for an efficient execution of other image-processing algorithms as well.

The design of this machine is based on our belief that an architecture tuned for an algorithm model is more efficient in executing that algorithm than general-purpose architectures. This approach will be justified when the algorithm under consideration can tackle a large enough class of problems. In
fact, the PRL method for solving continuous labeling problems (explained in Section 2) allows for the design of efficient parallel algorithms for most low-level vision problems and image-processing tasks.

It has been realized for a long time that in most low-level vision and image-processing tasks, due to the uniformity of computations performed across the 2-D arrays, extraction of data parallelism leads to their efficient execution [8]. Thus the SIMD model of processing is ideal for the parallel implementation of these algorithms.

In the MIMD schema, as outlined by Flynn, parallelism may be extracted from a single task by dividing it into independent subtasks which execute on their individual processors with their own program and data sets. Some amount of explicit communication takes place between the subtasks/processors. Due to this capability of independent processing, these structures are more flexible. However, this flexibility is not very useful in low-level vision, which is characterized by identical computations and uniform data movement. Further, as will be evident in Section 2, in the probabilistic relaxation method it is necessary for processors operating at each of the data in the array to exchange intermediate results with processors in the neighborhood, at each iteration of the algorithm. This forces all the processors to run in explicit synchronism, a situation where the interprocessor communication scheme of MIMD machines would prove inefficient [9]. These are the main reasons for preferring the SIMD schema for a special-purpose machine for low-level vision. However, an MIMD machine for discrete labeling has been proposed by Haralick et al. [10]. At the end of Section 2, we indicate why such structures are not suitable for our purposes.

The PRL method requires processes at each data point, to interact with processes in a window around that point. Based on this requirement, the communication module has been designed so as to access entire data windows (of programmable size) with the execution of a single instruction. This new communication strategy and the nested data-conditional enabling and disabling of PEs are the main features of the design. In Section 5, we compare features of this architecture against those of a few other general-purpose SIMD machines like the ILLIAC IV, MPP, Connection Machine, and PASM, and illustrate how tuning the architecture to an algorithm model serves to make it more efficient for low-level vision tasks.

The paper is organized as follows. Section 2 starts with a brief description of the relaxation labeling paradigm and explains the problems of discrete and continuous labeling. We point out, through an example, why discrete labeling is an unsuitable model for low-level vision. We briefly explain the probabilistic relaxation technique for solving continuous-labeling problems and the computational requirements of algorithms based on it. The next two sections describe
the proposed architecture and its machine-level instruction set. Section 5 compares this machine with other parallel architectures for vision, and brings out its distinct features. Finally, Section 6 describes how the PRL algorithm can be implemented on this machine and presents the time analysis of an algorithm for stereo correspondence.

2. RELAXATION LABELING

Relaxation labeling is a technique for solving a class of problems called labeling problems. This technique is characterized by [11]:

1. The decomposition of a complex process into a network of simple local processes.
2. The use of contextual information in resolving ambiguities.

Davis and Rosenfeld [5] and Zucker [11] have shown how many image-processing and low-level vision problems can be fitted to the relaxation-labeling paradigm. The task of interpreting an image can be split into local processes of assigning labels at individual image data points using neighborhood information to arrive at a globally consistent interpretation of the scene.

A general labeling problem is defined [13] by giving:

1. a set of objects, \( O = \{O_1, O_2, \ldots, O_n\} \);
2. a set of labels for each object, \( L = \{L_1, L_2, \ldots, L_m\} \);
3. a neighbor relation over the objects, and
4. a constraint relation over labels at pairs or \( N \)-tuples of neighboring objects.

The problem is to find an assignment of labels to the objects in a manner which is in agreement with the constraint relations. These constraint relations, also referred to as compatibility functions, locally evaluate the assignment of labels over neighboring objects. These compatibility functions can be either logical or real-valued. The labeling problem is termed discrete in the former case and continuous in the latter.

Many low-level vision problems can be posed as continuous labeling problems. For example, consider the case of stereopsis, which is the process of recovering depth information from a pair of images of the same scene. When a scene is viewed from two vantage points, the images obtained are slightly disparate, i.e., the positions of the two image elements corresponding to the same point in the scene are laterally shifted, the shift (called disparity) being inversely proportional to the depth of the viewed point in the scene. In order to recover the depth information, we calculate the disparity values for a set of
primitives, such as zero crossings, in the image. Since \textit{a priori} there are many points in the second image that can correspond to a point in the first image, there is an ambiguity in assigning the disparity values. This is to be reduced by making use of some general observations—for example, that the depth varies continuously almost everywhere in a scene, since almost all surfaces are smooth. It is easy to see that this fits into the model of the continuous labeling problem. Here the objects are zero crossings and the labels are disparity values. Compatibility expressions convey how well two disparity values fit two neighboring zero crossings, based on the constraint mentioned above. But since there do exist surface discontinuities, it is not possible to assert that two disparity values of neighboring objects are totally compatible or totally incompatible. In other words, the compatibility functions cannot be binary valued, and the problem cannot be expressed as a discrete labeling problem.

\textit{Probabilistic relaxation} is a technique using which low-level vision problems can be solved through local cooperative processes in the framework of a continuous labeling problem. Here, to start with, a probability distribution over the label set is assigned for each object. This initial probability distribution specifies possible interpretations for various points in the image. At each iteration this distribution is updated through either a deterministic or a stochastic rule, using the neighborhood information, in an effort to arrive at a globally consistent and unambiguous interpretation. In deterministic algorithms [12, 13], the updating of probability distribution for an object requires the values of current probability distributions of the neighboring objects. On the other hand, an algorithm with a stochastic updating rule [4] has smaller communication overheads. Here, at each object, a label is selected based on the current probability distribution, and the updating rule needs to know only the label selected by the neighboring objects at that instant. In all these algorithms, the probability distributions tend asymptotically to unit vectors giving an unambiguous interpretation of the scene.

The machine described in this paper is intended for parallel implementation of probabilistic relaxation algorithms for low-level vision in an SIMD framework. As mentioned in Section 1, MIMD architectures are not very useful for this purpose. Generally, three different MIMD structures have been considered for data-intensive tasks like image processing and low-level vision [9]. Of these, the shared-memory and bus-based structures are seen to be inefficient for PRL algorithms, due to the accessing of neighbor data at each iteration. Loop architectures like the ZMOB have been considered for the local computations in image processing [24], where one seeks to avoid contentions by loading overlapping segments into each processing element. Thus each processor contains all the neighbor data needed for the processing—for example, in image filtering using a mask. But in the probabilistic relaxation method, it is easy to see that this preloading of data would not suffice. This is because the processor
operating at each object needs intermediate results from the neighboring processors (e.g. the current probability distribution or the current label), and hence preloading of all neighboring data values will not help.

There is an MIMD architecture for the discrete labeling problem [10], which uses the backtracking algorithm with some lookahead. As explained in this section, most low-level vision problems can be posed only as continuous labeling problems. Since here the compatibility functions are real valued, backtracking-type algorithms are not very useful.

Array-processor architectures have however been designed for solving certain other classes of AI problems using discrete labeling, for example the SNAP [15]. Here the neighborhood relation over the objects is in the form of a semantic network. Since in this case processing consists mainly of symbolic manipulations and also the neighborhood data movement is not uniform, the architectural features of the SNAP are very different from the present architecture. These differences are indicated in the later sections.

3. MACHINE DESIGN

The proposed machine is a peripheral array processor which belongs to the general class of SIMD machines. The array is a matrix of PEs (processing elements), whose operation is controlled by a set of instructions broadcast from the host and decoded by the control interface (CIF) (Figure 1).

The main points that were considered to arrive at the various features of the design are as follows:

(1) Each PE requires advanced computational capabilities such as floating-point arithmetic. This is because the PRL algorithm deals with real-valued vectors, and floating-point treatment of these values ensures that the necessary precision is maintained throughout the processing.

(2) The I/O scheme to input image data onto the array should be flexible enough to enable the user to load overlapped segments of the original image into each PE.

Fig. 1. System configuration.
(3) The communication scheme of the system must be capable of efficiently accessing values from neighboring PEs in a window around each point. The size of the neighborhood should be controllable by the user. Since each step of the PRL algorithm needs data from all the neighbors in the window, it would be inefficient if the communication scheme required each PE to access each of its neighbors under explicit program control.

Operation of the machine proceeds by first loading the data onto the array processor (one datum per PE or one subimage per PE), followed by the broadcasting of instructions to the PEs. The PEs operate on the data independently, and each of them generates its own results, which are stored in its local memory. During neighborhood processing the necessary data are accessed by communication links which connect the PEs to their eight neighbors lying in a $3 \times 3$ window (Figure 3).

Each row of PEs sits on a rowbus (Figure 2), which is used to broadcast instructions and data to the PEs. All rowbuses are in turn connected to a single
bus emanating from the CIF. The rowbuses make it possible to input data to the PEs in several modes (see Section 4).

A single PE consists of three modules (Figure 4):

1. a computational unit,
2. a communication unit,
3. a memory module.

3.1. THE COMPUTATIONAL UNIT

This unit is a full-fledged arithmetic unit capable of integer and real (double-precision floating-point) arithmetic as well as Boolean operations. Of the six registers available, the A, B, and V are data registers, and the 1x1, 1x2, and MAR are address registers. Apart from these there is a separate status register.

All the address registers may be operated in the auto-increment mode. The 1x1 and 1x2 registers are used explicitly for sequentially accessing contiguous data segments in the local memory, while the MAR can be used for random accessing of the local memory also. Due to the multiplicity of the address registers, it is not necessary to load and unload a single MAR while working simultaneously with two data sets residing in the local memories of the PEs.

Prior to the execution of any two operand instructions, the operands are loaded into the A and B registers. In the case of a single operand instruction the operand is loaded a priori into the A register. The V register, apart from providing temporary storage for the partial results of an operation, holds the
data that must be routed to the neighbors during a communication operation. The operands that are loaded into the $A$ and $B$ registers could be (1) local data stored in the local memories, (2) neighbor data also present in the local memory, (3) global data sent over the row buses, or (4) the contents of the address registers of the local memories.

The status register of each PE consists of two flags, namely the sign and the zero flag. The presence of the status register allows the enabling/disabling of PE's without any intervention from a controller for the purpose of status evaluation of individual PEs. This enabling/disabling is functionally equivalent to a distributed masking scheme. One of the main advantages of such a scheme is that it facilitates the formulation of powerful data-conditional instructions such as **WHILE** loops and **IF-THEN-ELSE** constructs at the machine level (Section 4.5).

In order to nest conditional instructions, the status register is associated with a nesting counter. This counter is initially set to zero and incremented or decremented under the control of these conditional instructions. This capability of directly nesting data-conditional instructions is a unique feature of the machine.

### 3.2. THE COMMUNICATION MODULE

The communication module (Figure 5) is responsible for neighbor data accessing. It uses the uniform and synchronous data movement during a communication step to achieve this. Each PE's neighbor bus is able to transfer data to all the direct neighbors in a $3 \times 3$ window. Data are input to the PE from the direct neighbors through a multiplexer.

The communication module of each PE consists of a read/write logic unit to sequence the accessing of neighbors, and a neighbor buffer to store the accessed values. The execution of a single communication instruction will cause all the neighbor values lying in a $3 \times 3$, $5 \times 5$, or $7 \times 7$ window (the size being specified as a parameter of the instruction) to be transferred to the local memory of the PEs starting from a location which is also specified in the instruction. The accessing of neighbor data takes place through cycles of communication. During a cycle, every PE in the array accesses its eight directly connected neighbors in a sequence of clock cycles. In order to access a $3 \times 3$ window a single communication cycle will suffice in which the eight neighbors of the window are accessed. For $5 \times 5$ and $7 \times 7$ windows, two and three communication cycles are required respectively. In the first cycle of communication, the v-register contents of each of the eight directly connected neighbors in a $3 \times 3$ window are accessed. These values are stored simultaneously in both the neighbor buffer and the local memory. In the next cycle of communication each
of the PEs accesses two of the values stored in the neighbor buffers for each of its eight immediate neighbors. This enables the PEs to access the sixteen neighbor values present in the second layer of a $5 \times 5$ window (Figure 6a). These values are also stored in the neighbor buffer as well as the local memory. In the third cycle of communication each PE accesses three values from the neighbor buffers of its direct neighbors, to obtain the 24 neighbor values lying in the outer layer of a $7 \times 7$ window. On decoding a neighbor communication instruction, the CIF decides on the number of communication cycles that must be executed.

Fig. 6. (a) Neighborhood configuration. (b) Contents of $\text{PE}_0$ neighbor buffer.
CONTENTS OF PE₀ NEIGHBOUR BUFFER
AFTER 1ˢᵗ CYCLE (1 ACCESS PER DIRECTION)

CONTENTS OF PE₀ NEIGHBOUR BUFFER
AFTER 2ⁿᵈ CYCLE (2 ACCESS PER DIRECTION)

CONTENTS OF PE₀ NEIGHBOUR BUFFER
AFTER 3ʳᵈ CYCLE (3 ACCESS PER DIRECTION)

(b)

Fig. 6. Continued.
Although each PE is physically connected to only its eight direct neighbors, this communication scheme allows each PE to access neighbors outside this window. This is possible through the use of neighbor buffers, which act like FIFO channels to read in the neighbor values from one end and read them out simultaneously through the other. The simultaneous writing of neighbor values into the local memories allows the data to be processed during subsequent instructions without explicitly having to load the accessed values under program control. The index registers can be used to create a logical partition in the local memory for holding the neighbor data [Figure 6(a) and (b)].

Apart from the window-accessing capability described above, it is also possible for each PE to access the v-register contents of any specific member of its $3 \times 3$ neighborhood. Thus, by multiple accesses, distant neighbors lying in any particular direction may be accessed.

The communication modules of PEs which have been disabled due to an earlier data conditional instruction continue to function in order that data to distant enabled PEs may be conveyed without interruption. Thus only the computational unit and memory module are disabled when a PE is disabled.

3.3. THE MEMORY MODULE

The local memory of each PE functions as a data memory and can be logically split into several partitions. Each partition can be used to hold different data sets. The tx1 and tx2 registers allow for direct accessing of these logical memory partitions without the overhead of loading and reloading a single MAR. This facilitates fast processing of local arrays of data. It is possible to load the index registers initially and to operate them in the auto-increment mode in order to process data sets sequentially.

4. THE INSTRUCTION SET

The different groups of instructions supported by this machine are as follows:

(1) input/output instructions,
(2) neighbor-communication instructions,
(3) computational instructions,
(4) data-movement instructions,
(4) conditional instructions.

The structure of each instruction is as follows:

\[ \text{opcode, parameter 1, parameter 2, ..., parameter 5} \]

A maximum of five parameters can be specified in an instruction.
An array-processor program is stored as a list of instructions in the host memory. The instructions are individually broadcast to CIF, which decodes them and sends appropriate signals along the rowbuses.

The instruction set for the machine has been formulated keeping in mind the processing requirements of the problems of interest. This instruction set makes it possible to code the PRL algorithm efficiently on the machine. In the designed machine, the input/output scheme allows overlapped segments of data to be configured into the local memories of neighboring PEs. This feature, which is necessary for parallel processing of overlapped data segments, is required during the computation of initial probability vectors in the PRL algorithm. Using the flexibility provided by this I/O scheme, it is possible to completely eliminate the neighbor-data accessing in certain image-processing algorithms [28].

The communication instructions have been coined as to completely utilize the advantages of the buffered window-accessing scheme. They cater to the requirements of local processing at every iteration by the PRL algorithm.

Besides these instructions, there are standard instructions for data transfer between the various registers, registers and memory, etc. Computational instructions allow the user to perform integer and real arithmetic as well as Boolean operations. The instructions which are used for real-valued data handling are vital to the PRL algorithm, since they allow the manipulation of the elements of the probability vectors which are used extensively during the execution of the PRL algorithm.

4.1. INPUT/OUTPUT INSTRUCTIONS

The three instructions belonging to this group are as follows.

4.1.1. INGLDT, MA, NR, NUR, NC, NUC

This instruction configures the relevant data sets into the local memories of the PEs. The data are broadcast by the host onto the rowbuses via the CIF. Once the data are on the rowbus, it is possible for any PE that requires the data to write them into its local memory. The various parameters are as follows.

The image data are stored in a local memory block sequentially, starting from the location MA. The rest of the parameters are explained using the following example.

Let the size of the subimage to be input into the local memory of each PE be $n \times n$. Then

\[
\begin{align*}
\text{NR} &= \text{no. of rows} = n, \\
\text{NC} &= \text{no. of columns} = n.
\end{align*}
\]
Of this subimage, if the number of overlapped data between adjacent PEs is $p$ (in terms of columns) and $q$ (in terms of rows), then

\[
\text{NUC} = \text{no. of unique columns} = n - p,
\]

\[
\text{NUR} = \text{no. of unique rows} = n - q.
\]

Let $\text{MA} = 1$, $\text{NR} = 3$, $\text{NUR} = 2$, and $\text{NUC} = 1$. Let

\[
A = \begin{bmatrix}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\
17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 \\
25 & 26 & 27 & 28 & 29 & 30 & 31 & 32 \\
33 & 34 & 35 & 36 & 37 & 38 & 39 & 40 \\
41 & 42 & 43 & 44 & 45 & 46 & 47 & 48 \\
49 & 50 & 51 & 52 & 53 & 54 & 55 & 56 \\
57 & 58 & 59 & 60 & 61 & 62 & 63 & 64
\end{bmatrix}
\]

be the data array in the host. Then $\text{PE}(1,1)$ will obtain the subarray $B$ given by

\[
B = \begin{bmatrix}
1 & 2 & 3 \\
9 & 10 & 11 \\
17 & 18 & 19
\end{bmatrix},
\]

and $\text{PE}(1,2)$ will obtain the array

\[
C = \begin{bmatrix}
2 & 3 & 4 \\
10 & 11 & 12 \\
18 & 19 & 20
\end{bmatrix}.
\]

From the two data submatrices, it is obvious that the amount of overlap in terms of columns is 2, as needed, since $\text{NC} - \text{NUC} = 3 - 1 = 2$. $\text{PE}(2,1)$ will receive the submatrix

\[
D = \begin{bmatrix}
17 & 18 & 19 \\
25 & 26 & 27 \\
33 & 34 & 35
\end{bmatrix}.
\]
The amount of data overlap between PE(1,1) and PE(2,1) in terms of rows is 1, since \( NR - NDR = 3 - 2 = 1 \).

After decoding this instruction, the parameters are input to a set of programmable counters. These counters enable a set of PEs so that they are able to access the necessary elements of the image data array when they are placed on the rowbus. This ensures that no element of the image data array, resident on the host, need be accessed or sent over the bus twice, even if it is to reside in the local memories of more than one PE.

4.1.2. \( \text{OTGLDT, MA, NR, NUR, NC, NUC} \)

This instruction operates very much the same way as INGLDT, except that in this case the data are output to the host. Normally, \( NR = NUR \) and \( NC = NUC \), as only unique result values are output from each PE and overlapping of data is not needed.

4.1.3. \( \text{INGLIM, data, reg} \)

This instruction is used to broadcast global constants to all the PEs. The effectiveness of the rowbus becomes apparent in this case, since it is not necessary to pump the data sequentially through the columns of the PEs as is done in several other array-processing systems, for example MPP [16]. The first parameter in the instruction specifies the global data value, and the second parameter, the destination register in each PE.

These I/O instructions assume an open data file in the host. Special array-processor directives configure the data suitably within the host. These directives include the OPEN and START directives. The former opens a particular data file in the host, while the latter indicates the location (ith row and jth column) in the data file, starting from which point an entire block of data may be read into the array.

4.2. \( \text{NEIGHBOR COMMUNICATION INSTRUCTIONS} \)

The two instructions belonging to this group are:

4.2.1. \( \text{INNDTAS, dir} \)

This instruction is useful whenever it is necessary to input data from an arbitrary neighbor lying within a \( 3 \times 3 \) window, i.e. any of the nearest neigh-
bors. The parameter "dir" specifies the particular neighbor. By the multiple execution of this instruction data residing with distant PEs may be accessed.

4.2.2. **INNDTS, size, MA**

This instruction allows for storing in the local memories data present in a window whose size is programmable. The parameter "size" indicates the size of the window, while MA indicates the starting location of the block which should contains the accessed neighbor data in the local memory. The execution of this single instruction causes all the neighbor values in a window to be accessed and stored in the local memories of the PEs. The operation of this instruction is described below:

**Procedure INNDTS (size, MA)**

Begin

If (size = 3) then
  call THREE
endif

If (size = 5) then
  call THREE
  (* this inputs the first layer of a $5 \times 5$ window (Figure 6) *)
  call FIVE
  (* this inputs the outermost layer of the $5 \times 5$ window (Figure 6) *)
endif

If (size = 7) then
  call THREE
  (* this inputs the first layer of a $7 \times 7$ window *)
  call FIVE
  (* this inputs the second layer of a $7 \times 7$ window *)
  call SEVEN
  (* this inputs the outermost layer of the $7 \times 7$ window *)
endif

end (* INNDTS *)

**Procedure THREE (MA)**

Begin

Do over the entire array—

Input into the neighbor buffer and the local memory of all PEs (starting from memory location MA in their local memory) the contents of the v registers of its eight neighbors, in a cyclic manner. From each of the direct neighbors read one data value.

enddo

end (* three *)
Procedure FIVE  
Begin  
Do over the entire array—  
Input into the neighbor buffer and local memory of all PEs (starting from memory location $MA + 8$ of their local memory) the contents of the neighbor buffers of its eight direct neighbors, in a cyclic manner. From each of the direct neighbors read two data values.  
enddo  
end (* FIVE *)

Procedure SEVEN  
Begin  
Do over the entire array—  
Input into the neighbor buffer and local memory of all PEs (starting from memory location $MA + 24$ of their local memory) the contents of the neighbor buffers of its eight direct neighbors, in a cyclic manner. From each of the direct neighbors read three data values.  
enddo  
end (* seven *)

The INNDTS instruction is executed by calling the appropriate procedure.

4.3. COMPUTATIONAL INSTRUCTIONS

These instructions are responsible for carrying out the arithmetic (real and integer) and Boolean operations. The integer and Boolean operations are especially suited to image processing, while the real-arithmetic instructions are necessary for PRL algorithms. The operands of any computational instructions must be assembled a priori in the A and B registers.

The arithmetic operations possible on the proposed machine are the standard addition, subtraction, multiplication, and division of integer and real numbers. In all the instructions below "dest" stands for the destination of the results. It could be the A, B, V, IX1, IX2, or the MAR register. The instructions below are for integer arithmetic:

(1) IADD, dest,  
(2) ISUB, dest,  
(3) IMUL, dest,  
(4) IDIV, dest.

The real-arithmetic instructions are

(5) RADD, dest,  
(6) RSUB, dest,
The Boolean instructions are

(7) \texttt{LAND}, dest,
(8) \texttt{LOR}, dest,
(9) \texttt{LNOT}, dest.

A set of instructions have also been included for the shifting of operands by a single bit. They are

(10) \texttt{ROL}, dest (shift left),
(11) \texttt{ROR}, dest (shift right).

4.4. \textit{DATA MOVEMENT INSTRUCTIONS}

These instructions are necessary for the movement of data among registers or between registers and memory. They are used to load the operands into the $A$ and $B$ registers before the execution of computational instructions. Their syntax is as follows:

4.4.1. \texttt{LOAD}, dest, sorc

“dest” stands for the destination of the data which are accessed from the “sorc.” If either of these parameters has the value ix1 or ix2, then the data are read from or written into the memory location pointed to by the contents of the ix1 or ix2 registers.

4.4.2. \texttt{MODIFY}, dest, sorc

This instruction is used to load into or load from the ix1 and ix2 registers.

4.5. \textit{CONDITIONAL INSTRUCTIONS}

These instructions emulate three high-level language constructs:

(1) an IF-THEN-ELSE structure,
(2) a WHILE structure,
(3) a DO loop.
It is obvious that, due to the multiplicity of the data in a general SIMD machine, a data-conditional branch is extremely difficult for the controller to execute, since it would involve the examination of all data values in the array. In the proposed machine the status register of the PEs is used to disable or enable each individual PE. This enabling/disabling is effected by the PE itself, and thus it may be viewed as a distributed masking scheme, without the intervention of the controller to examine an individual PE status.

The **IF-THEN-ELSE** structure has been implemented for branching on the following conditions:

1. \((A) = (B)\),
2. \((A) < (B)\).

\([N]\) indicates the contents of register \(N\). An **IF-THEN-ELSE** clause like

\[
\text{if } c \text{ then } \quad \text{S1} \\
\text{else} \quad \text{S2} \\
\text{endif}
\]

where S1 and S2 are arbitrary sets of statements and \(c\) is one of the abovementioned conditions, is coded as

\[
\text{IFSUB} \\
\text{CHECK, } \text{cond} \\
\text{statements for S1} \\
\text{FLIP} \\
\text{statements for S2} \\
\text{ENDIF}
\]

1. **IFSUB**: This instruction, when executed, subtracts the contents of register \(B\) from the contents of register \(A\) and sets the sign or the zero flag as the case may be. An \text{ISUB} or \text{RSUB} will not have any effect on the contents of the sign or the zero flag.

2. **CHECK, cond**: Here “\(\text{cond}\)” is either \text{SIGN} or \text{ZERO}. If either of these flags is set, then PEs for which it is not set are disabled. Thus the enabled PEs execute the S1 statements.

3. The **FLIP** instruction disables active PEs and enables inactive ones, thereby emulating the “else” part.

4. The **ENDIF** instruction activates all inactive PEs at the end of the **IF-THEN-ELSE** structure.

The **WHILE** construct checks the following conditions:

1. \((A) = (B)\),
2. \((A) < (B)\).
A WHILE construct in the array-processor program has the following structure:

```
DOWHIL
LOAD, A, a
LOAD, B, b
WSUB
TEST, cond (cond = SIGN or ZERO)
EDWHIL
```

The DOWHIL instruction marks the beginning of a WHILE loop. The LOAD instructions set up the accumulators to check the condition of the WHILE loop. The WSUB instruction sets the SIGN or ZERO flag depending on the values of the A and B registers.

The TEST is an instruction which checks the ORed status of all the PEs and transfers control to the instructions within the loop or outside the loop. It also acts like a CHECK instruction and disables PEs depending on their structure. The EDWHIL instruction marks the end of the WHILE loop.

Nesting of these constructs is possible through the nesting counter. The nesting counter of all enabled PEs is always zero. The status of each PE is individually maintained by the PE itself throughout the various nesting levels. The rules followed to achieve this are as follows:

1. A disabled PE increments its nesting counter by one every time an IFSUB or a WSUB is encountered, and decrements it by one (if its nesting counter 0) every time it encounters an ENDF or an EDWHIL instruction.
2. The instructions which enable disabled PEs, like FLIP, ENDF, and EDWHIL, are executed by disabled PEs only if their nesting counter is zero.

Using the nesting counter, IF-THEN-ELSE can be nested to an arbitrary level without any overhead at the CIF. To nest the WHILE structure, the CIF maintains a stack for a return address. At present this allows for only a depth of five in nesting the WHILE structure. These instructions can be arbitrarily interleaved.

The DO-loop is simulated by two instructions:

1. DO N,
2. ENDDO.

Here N is an integer specifying the number of times a loop may be traversed. The ENDDO marks the end of the loop. Nesting of DO loops is allowed up to five levels. These instructions are executed by special hardware on the CIF, which consists of two stacks. One of these keeps track of the loop index, while the other holds the return addresses within the loops.

This nesting mechanism with the explicit nesting primitives makes it possible to write array-processor programs using high-level language constructs. Also
compilation of standard high-level languages into object code becomes very easy due to the presence of these powerful assembly language opcodes.

5. DISCUSSION

The SIMD machine presented in this paper is specifically intended for solving low-level vision and image-processing problems using the relaxation-labeling model. In this section, we compare the features of this machine with other SIMD machines, such as the MPP, the Connection Machine, the PASM, and the ILLIAC IV, which are suited for data-intensive computations. The comparison is made with respect to the following features:

1. Computational operations,
2. input-output operations,
3. neighbor-data accessing,
4. execution of data-conditional instructions.

5.1. COMPUTATIONAL OPERATIONS

The processing requirements of vision and image-processing algorithms range from logical operations on binary images to complex processing of real-valued arrays. The main computation involved in the PRL algorithm is the updating of label probabilities, which are vectors with real-valued components. In addition, in the algorithm described in [14], the computational requirements also include random-number generation at each iteration. This extra computation is more economical because it drastically reduces the information to be accessed from each neighbor. All these requirements necessitate a powerful arithmetic capability within each PE. Thus each PE is designed to have a full-fledged arithmetic unit capable of both real and integer computations.

In both the MPP [16] and the Connection Machine [17] each of the PEs operates on a single-bit datum. Hence these machines are highly suited to the processing of binary images. Grey-level processing is done on these arrays by processing bit planes of these images in parallel. However, a large amount of hardware is necessary to store and sequentially access the bit planes of the image in parallel (e.g. staging memory in the MPP [18]). Batcher [19] has pointed out that one of the advantages of using bit-slice processing is that nonstandard data formats can be supported by the machine. However, such formats are extremely rare in low-level vision and image processing. Hillis [17] has on the other hand adopted the bit-slice strategy for achieving "fine-grained parallelism." By going down to the level of processing bit slices in the Connection Machine, the basic clock rate of the system can be increased tremendously. However, the drop in throughput while executing floating-point operations on
the MPP [19] indicates the unsuitability of the bit-slice strategy to real-valued data processing.

In both PASM [20] and the ILLIAC IV [21], as in our machine, a word-parallel processing scheme has been adopted. In addition the facilitating the high-precision handling of real-valued data, the wider word length leads to the possibility of a more powerful instruction set. This is because here the data manipulation is not purely logical in nature as it is in bit-slice processing. Thus it is possible to include instructions that emulate features of high-level languages, at the machine level. The SNAP [15] machine also allows for powerful computations, but mostly from the point of view of symbolic manipulations required in AI. For example, it includes content-addressable memories, a complexity which is superfluous for low-level vision problems.

5.2. INPUT/OUTPUT

Configuring data onto the PE memories from a host or a secondary storage device constitutes one of the unavoidable overheads in an SIMD machine. There are two different aspects with respect to which the process can be optimized. The first is to consider whether some or all of the PEs can be loaded in parallel with their respective data sets. The second aspect is to avoid redundancy of data transmission when the data sets of the PEs have a nonzero intersection.

The most ideal case for parallel loading of data is when all PEs can be simultaneously loaded from the host. This would necessitate a very wide data bus from the host and the attendant hardware for making all the data elements available on the bus at one time. The I/O proposed for the ILLIAC IV [22] embodies this feature. In this machine, the data link is 1024 bits wide, allowing 16-bit data words to be loaded in parallel to the 64 PEs of the array. On the MPP which is a $128 \times 128$ array, columns of 128-bit data are loaded in parallel. Thus, to transfer a single bit-plane onto the array, 128 such columns are to be transferred. Since here data loading can proceed concurrently with processing, the remaining bit-planes need only one machine cycle to actually get loaded. This I/O operation is controlled by the stager memory which has to load from the host the entire data array through a serial link. Thus, for an $N \times N$ array of $n$-bit data, the total I/O time would be $N_2 T_{I/O} + (128 + n) T_{I/O}$ where $T_{I/O}$ is the transfer time on the host serial link, and $T_{I/O}$ is that between the stager memory and the array.

In PASM [23], the PEs can be divided into partitions. Let $M$ be the total number of PEs and $Q$ the number of partitions. Then $M/Q$ is the number of PEs per partition. A separate memory storage unit (MSU) and a disk is provided for each set of $M/Q$ processors. All the PEs in a partition can be
loaded in parallel by operating all the MSUs together. To load an $N \times N$ array, the total I/O time would be $N_2T_{1/O}/Q + (N_2T_{1/O}Q)/M$ where $T_{1/O}$ is the data transfer time between the disk and MSU and $T_{1/O}$ is the transfer time between the MSU and PE. (Here the transfers are made in terms of words since PASM is capable of word-parallel processing.) On the connection machine, the data may be input either through a host array link or a 500 Mbits per second high speed I/O link.

In all the above cases, if the same data words are to be loaded into more than one PE, repeated transfer of the data words are needed. Thus, the effective number of data items to be transferred would be $N_2$, where $\beta > 1$, indicates the extent of overlapping in the data sets of neighboring PEs. Thus, none of the above schemes optimize with respect to multiple transfers of the same data words in cases of overlapped data segments.

In the present machine, the view adopted is to optimize with respect to multiple transfers of data words, i.e., a data word need be transferred only once. To load an $N \times N$ array on our machine, the time taken would be $N_2T_{1/O} + N_2T'_{1/O}$, where $T_{1/O}$ is the data transfer time between host and CIF, and $T'_{1/O}$ is that over the rowbus. However, our INGLDT instruction allows for loading arbitrarily overlapped data segments into each PE. Each element of the data array is transmitted only once from the host, and while it is on the rowbus, all PEs that need it are enabled to load it into their local memories simultaneously.

This is the only parallel loading exhibited by the proposed machine. Thus, in cases where nonoverlapped data segments are to be loaded, the process of loading the PEs is entirely sequential.

In most PRL algorithms, the original data array is utilized for computing initial probability vectors, and to decide on the relevant neighbors. For both these purposes, configuring overlapped data segments onto the array is necessary for efficient parallel computation. Hence we chose to optimize the I/O with respect to multiple transfers rather than using a scheme to load, in parallel, mutually exclusive data sets into the local memories of the PEs. An additional advantage is that in this case the hardware complexity for implementing the I/O scheme is very much smaller than that in PASM or MPP.

5.3. NEIGHBOR-DATA ACCESSING

A large variety of low-level vision and image-processing problems require neighbor data for the processing. In purely image-processing tasks like convolution, the processing at each point needs only the original image data in a neighborhood around that point. Thus such tasks can be efficiently executed even on loop-based MIMD architectures like ZMOB [24], by preloading the
neighbor data into each processor as explained in Section 2. (Even on our machine, image-processing tasks like convolution can be executed without any neighbor-communication overheads by preloading overlapped data [28].) In contrast, in most image-interpretation or low-level vision tasks, neighboring processors need to exchange intermediate results to arrive at an unambiguous and globally consistent interpretation of the scene. Probabilistic relaxation can be considered as a model for such local cooperating processes. For executing these algorithms, the neighbor-communication overhead cannot be dispensed with by preloading neighbor data from the original image. Hence any architecture that executes low-level vision algorithms should be capable of efficiently accessing the state of processes in a window around each point, and since the accessing is required at every step of the algorithm, it is desirable that this process should go on with as little intervention by the programmer as possible. These are the main considerations for arriving at our communication scheme.

The INNDTS instruction supported by our machine provides the user with a window-accessing capability. It allows the user to choose the neighborhood size required. The 7x7 limit has been chosen because this window size would suffice for "local" processing. The limit can be easily extended without necessitating any major design changes at the hardware level. By executing this single instruction, the user can store all the neighbor data into the local memory. In problems where the relevant neighbors might be data dependent, the choice can be made by examining the individual data values. Apart from this, through the INNDTAS instruction, each PE can access an arbitrary neighbor. This should be useful when the user needs only a single neighbor datum from a local window or from a distant PE. No support in terms of an automatic routing algorithm is provided for accessing a PE outside a 7x7 window. This accessing is to be effected through multiple execution of the INNDTAS instruction. The main reason behind this design decision is that local processing is a more important factor for low-level vision processing and that the frequency of accesses from distant PEs is small. Further, since an 8-connected mesh has been adopted, cascading of values from distant PEs would require fewer steps.

In the earliest SIMD machines like the CLIP [25] family, the neighbor information accessed at processing time is a predefined function operating on all the neighbor values, the function being selectable from a set supported by the machine. The neighbor-function generator of the CLIP family operated on single-bit data words, and hence such architectures are not useful in the probabilistic-relaxation framework. Further, the processing of relevant neighbor data used in the probabilistic updating varies from problem to problem, and it is not possible to capture all these configurations through a set of neighbor-data functions.

In ILLIAC IV, the processors communicate through a 4-connected mesh. For accessing any neighbor, an explicit instruction is needed. This is similar to the
INNDTAS instruction except for the fact that automatic routing of distant neighbors is supported. However to access all the neighbors in a (for example) $5 \times 5$ window, 25 distinct instructions need to be executed, with the attendant decoding overheads. Through the buffered accessing mechanism adopted in the proposed machine, these overheads are dispensed with.

The 4-connected mesh on the MPP is identical to the connection scheme employed on ILLIAC IV. On the MPP, distant PEs need to be accessed by cascading the values through the network by the multiple execution of the neighbor-accessing instructions. Accessing through the grid network can be done by buffering distant data values in the local memories. However, this would involve an extra memory read per data value and hence does not decrease the overall communication time. The unbuffered scheme of the MPP, on the other hand, operates in very much the same way as the INNDTAS instruction. However, since no diagonal data movement is allowed, it would be more inefficient.

In the Connection Machine, the PEs may communicate through either the NEWS grid or the router network. Through the NEWS grid the communication scheme would be identical to that on the MPP and hence would be inefficient for window accessing. In case of the router network, each router serves the communication requirements of sixteen PEs packed into a chip and honors the requests of only four of these during a communication cycle. Thus, communication through the router network slows down the data accessing. During window accessing, explicit data transfer instructions need to be executed for accessing each of the data values in the window. Further, for all the PEs within a chip it can be assumed that on the average, at least half of their neighbors will lie on different chips, and hence will require communication between routers. This will result in further slowing down the accessing mechanism. However for accessing distant PEs, the router network would perform much better than cascading data through the mesh as is required on our machine or the MPP.

In PASM [26, 27], the interconnection network is reconfigurable, and by executing a proper routing function, any PE can be accessed in a single step. In addition to the hardware complexity involved in this network, the window accessing still requires execution of multiple instructions. The hardware in our buffered neighbor-accessing scheme is very much less than that needed for the reconfigurable network. On the other hand, accessing of distant neighbors on the PASM is much more efficient than the cascading needed on our machine through the INNDTAS instruction. But as stated above, in considering the suitability for low-level vision problems, the window-accessing capability is a more important yardstick.

Table 1 compares the complexity, in terms of communication steps, in accessing windows of different sizes on all the machines mentioned above. The overhead of decoding multiple instructions is not included.
5.4. EXECUTION OF DATA-CONDITIONAL INSTRUCTIONS

Execution of data-conditional instructions is generally difficult on an SIMD machine, due to the multiplicity of data. Execution of such instructions in an SIMD environment requires a subset of PEs to be enabled or disabled, depending on local data, during the execution of subsequent instructions. On our machine, this is supported through the presence of status registers, which are set or reset based on the local computations. This is used for implementing data-conditional instructions like IF-THEN-ELSE and WHILE at the machine level. A novel feature of the data-conditional instructions on our machine is that through the use of a nesting counter, these instructions can be nested. The scheme involves no overhead on the part of the controller either to examine the data in the PEs or for mask formation. The only construct for which the controller need examine the PE status is for the WHILE instruction to check when all PEs are disabled, in order to terminate the loop. This is easily achieved by examining the wire-ORed status flags of the PEs. (Disabled PEs do not interrupt communication.)

The data-conditional instructions on the ILLIAC IV are similar to those on our machine, in that each PE is disabled according to a local status register. But it is not possible to implement nested data-conditional instructions on this machine.

The data-conditional masking on the PASM, using the “if any” or “if all” instructions, performs in very much the same way as our IF-THEN-ELSE or WHILE construct does. However, the nesting mechanism of these instructions is achieved through a run-time stack. On the MPP, the PE status is governed by its G register. To execute a data-conditional instruction one has to process $n$-bit planes to set the G register. Another aspect of the MPP's masked instruction

### TABLE 1
Communication Overheads for Various SIMD Machines

<table>
<thead>
<tr>
<th>Neighborhood size</th>
<th>Steps</th>
<th>Connection machine (router access)</th>
<th>Proposed network machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neighbor</td>
<td>Buffer</td>
<td>Unbuffered</td>
<td>ILLIAC IV</td>
</tr>
<tr>
<td>Neighborhood size</td>
<td>access</td>
<td>access</td>
<td>(A)</td>
</tr>
<tr>
<td>3 x 3</td>
<td>$2 \times 8 + 2 \times 2$</td>
<td>$4 + 2 \times 4$</td>
<td>4 + 8</td>
</tr>
<tr>
<td>5 x 5</td>
<td>$2 \times 28$</td>
<td>$4 \times 2 + 4 \times 3$</td>
<td>$8 + 12$</td>
</tr>
<tr>
<td></td>
<td>$+ 2 \times 4$</td>
<td>$+ 4 \times 4$</td>
<td>$+ 16$</td>
</tr>
<tr>
<td>7 x 7</td>
<td>$2 \times 60$</td>
<td>$4 \times 3 + 4 \times 4$</td>
<td>$12 + 16$</td>
</tr>
<tr>
<td></td>
<td>$+ 2 \times 6$</td>
<td>$+ 4 \times 5 + 4 \times 6$</td>
<td>$+ 20 + 24$</td>
</tr>
</tbody>
</table>
execution which is disadvantageous is that masked PEs do not participate in data routing.

Masked operations on the Connection Machine are also similar to these schemes. Here also there is no nesting of data-conditional instructions.

6. IMPLEMENTATION

In this section, we illustrate how the PRL algorithm as posed in [14] is implemented on the present architecture. Taking the stereo correspondence problem as an example, we estimate the speedup on a simulated version of the machine.

Any algorithm using the PRL technique consists of the following steps. Initially, each object is associated with a label probability vector, the elements of which are calculated for each object. Next, the processing takes the entire object set through some iterative steps. During this iterative process, each object chooses a label at random based on the label probability vector, and for each individual choice, a response is calculated according to the labels chosen by the object's neighbors. This response is used to update the probability vectors. The iterative processing ends when a consistent labeling is obtained over the entire array. Since the processing steps are identical for every element of the object set, it is possible to extract data parallelism from this algorithm.

On this architecture, data parallelism is exploited by assigning to each PE a single object or a subset of objects, depending on the size of the array. The data are fed into the local memories via the rowbus by executing an I/O instruction. This transfer represents an overhead which is associated with any peripheral unit communicating with a host. Constants used during the computation steps may also be broadcast and stored in the local memory of PEs.

The PRL algorithm begins with the calculation of the initial label probability vectors for each of the objects. Although, in general, the actual values used during this computation are problem dependent, in every case the elements of the probability vector are computed by comparing the feature values of the individual objects with those of the labels. The label features are broadcast a priori to the entire set of PEs simultaneously. Thus the communication overheads are distributed over the set of PEs. This computation is entirely local to each PE. Thus instructions belonging only to the computational group and the data-movement group would be required.

In the next step each PE chooses a label and calculates the response to this choice, based on the labels chosen by its neighbors. The choice of a label by each PE is made entirely locally. Here the instructions of the computational group are used extensively, since the processing involves the generation of a random number and comparison with the elements of the probability vector.
The advanced arithmetic unit is capable of handling these computational requirements. During the calculation of the response, it is essential for each PE to access the labels chosen by its neighbors. The appropriate neighborhood is a window around that point, and accessing of the neighborhood is done very efficiently using the INNDTS instruction. The advantage gained by using the algorithm in [14] is that the neighbor data reduce to the label value chosen by the neighbor. This value is an integer, and the number of data to be communicated does not depend on the problem size. This is in contrast with other probabilistic relaxation algorithms, where entire probability vectors are to be accessed from neighbors, and hence the number of elements communicated would depend on the number of possible labels.

The calculation of the response is done by each PE, by using the relevant neighbors from the accessed data set. The response is in turn used to update the elements of the label probability vector. This is again a local operation requiring, mainly, the computational group of instructions.

The process of choosing labels and updating the label probability vector is iterated over the entire set of PEs until all the PEs have converged to a consistent labeling. The termination of the processing, when convergence conditions have been reached, is brought about either by executing a WHILE loop or by executing the iterative steps a very large number of times using a DO instruction.

**TIME ANALYSIS OF THE STEREO CORRESPONDENCE PROBLEM**

In order to evaluate the parallel machine, we take the example of a stereo correspondence problem and calculate the speedup achievable on the machine. (A complete analysis is available in [28].) Speedup is defined as the ratio of the time taken by the algorithm on a sequential machine ($T_{seq}$) to the time taken to run the same algorithm on a parallel machine ($T_{par}$):

$$speedup = \frac{T_{seq}}{T_{par}}$$

In order to execute the stereopsis algorithm on a sequential machine, the processing is iterated over the entire data set. The total computation at each object can be divided into two steps:

1. Calculation of initial probability assignments.
2. The iterative loop for converging to the consistent labeling.

The first step involves comparing features of the candidate points in the right image with the features of the chosen point in the left image. Let the total computation at this step be $T_{FL1}$. The second step involves the following
operations:

1. Generation of a random number.
2. Selection of a label.
3. Calculation of the response to this label selection.
4. Updating of label probabilities.

Let $T_{FL2}$ be the time per iteration in this step. If the image contains $N_2$ points and the number of iterative steps is $p$, then

$$\text{time for sequential run of algorithm} = N_2 T_{FL1} + pN_2 T_{FL2}.$$  

In the case of the parallel implementation, the same computation for each object is done by the processor assigned to that object. Hence assuming an $N \times N$ array of PEs, the time taken for the parallel implementation should

$$T_{\text{par}} = T_{FL1} + pT_{FL2},$$

giving rise to a speedup of $N_2$. This is because the algorithm is ideally data parallel.

To execute this algorithm on a parallel machine some additional overheads are involved. The first step involves the configuration of data on the array of PEs so that the calculation at each object can proceed concurrently. This requires transferring seven $N \times N$ arrays (three for the features of the left image, three for the features of the right image, and one for the seed of the random-number generation). This involves a time of $7N^2 T_{1/O}$, where $T_{1/O}$ is the time for the transfer of one byte of data from the host to the array processor. In addition to this, in each iteration in the second step, the labels chosen by the neighbors have to be accessed, involving an extra $T_{1NDTS}$ time units per iteration ($T_{1NDTS}$ is the time for a synchronous data access). Thus we have

$$T_{\text{par}} = T_{FL1} + pT_{FL2} + 7N_2 T_{1/O} + pT_{1NDTS}$$

Thus

$$\text{speedup} = \frac{N_2}{1 + \frac{pT_{1NDTS}}{pT_{FL2} + T_{FL1}} + \frac{7N_2 T_{1/O}}{pT_{FL2} + T_{FL1}}},$$

Using typical values for our proposed implementation (time for one floating-point add is 11 microseconds, time for one floating point multiply is
twice the time for one floating-point add, I/O rates are 0.4 Mbyte/sec, synchronous clock frequency is 8 MHz), the second and third terms in the denominator of the expression for speedup have values 0.03 and 0.01 respectively. Thus theoretically the speedup is less than the ideal by a very small factor. It should be noted that we assume the array to be sufficiently large. If, due to size limitations, we have to process more than one object per PE, the speedup will decrease. A point worth noting is that the second term in the denominator, which represents the fraction of communication overhead, is independent of the data-array size.

7. CONCLUSIONS

This paper describes a new SIMD machine for the parallel implementation of low-level vision and image-processing algorithms. The machine is designed to efficiently execute algorithms in the relaxation-labeling paradigm, and hence can exploit the data parallelism inherent in the problem.

Each of the PEs in the proposed design is an advanced processing unit capable of real and integer arithmetic. It operates on entire words at a time, i.e., the form of processing is word parallel, a feature not found in machines like the MPP or the Connection Machine, where bit-serial processors operate on bit planes in parallel. The departure from this mode of processing is necessitated by the fact that algorithms like the PRL algorithm deal with real values, which are best represented in the floating-point format. It has been found that operations on floating-point numbers using bit-serial processors prove quite inefficient [19].

In the proposed machine, the buffered communication scheme allows the access of data from neighbors which are not directly connected to a PE. This satisfactorily meets the need for obtaining local data during processing. It is also worth noting that the neighbor data can be individually accessed and used in arithmetic operations, a feature which is different from the cellular logic arrays such as the CLIP family of processors [25]. The communication scheme permits accessing of data windows of programmable size around each point. This makes the machine more efficient for low-level vision problems.

Based on the theoretical analysis for computations needed for the stereopsis algorithm on the parallel architecture, it is found that a speedup of almost $N_2$ is achieved by using an array of processing elements.

From the implementation of the stereopsis algorithm, it is clear that other low-level vision algorithms such as surface interpolation or grouping problems for primal sketch, which can be posed as labeling problems, can be easily executed on this system. Other image-processing problems like convolution can
also be done on this architecture by preloading appropriate windows into the PEs [28].

At present the architecture is being evaluated on a simulator, and it is proposed to build the system using off-the-shelf hardware.

REFERENCES


*Received 1 June 1987*