Contexts and Vision

Vito Di Gesu'(+,*) and Francesco Isgro'(+,#)

TR-95-038

August 7, 1995

Abstract

This report deals with problems of representation and handling of concurrent processes in multi-processor machines or in distributed and co-operating systems oriented to image analysis. For this purpose, the definition and some formal properties of a new synchronization engine, named "context" are given. Contexts are introduced as object variables in pictorial languages to represent distributed computation on spatial data. In particular, details of its implementation on the PIctorial C Language (PICL) are given. Operations are defined on the contexts space; the existing relations between contexts, formal languages, and graphs are considered, and have been used to optimize the implementation of contexts inside PICL.

Key words: *Parallel languages, Concurrence, Graph theory, Image Analysis*

+ Dipartimento di Matematica ed Applicazioni, University of Palermo, Via Archirafi 34 - 90123 Palermo, ITALY digesu@ipamat.math.unipa.it, isgro@ipamat.math.unipa.it

* ICSI, USA - digesu@icsi.berkeley.edu

\# Department of Computing and Electrical Engineering, Univ.of Edinburgh, UK - isgro@cee.hw.ac.uk

1. Introduction

Massive parallel machines (MPM) may be thought as sets of processing elements (PE's) interconnected according to some defined architecture topology (linear array, mesh array, cube and hypercube, pyramid,...). Data and instructions are distributed among the PE's, the computation spans from Single Instruction Multi-Data (SIMD) to Multy Instructions Multi-Data (MIMD) paradigms. Formally, MPM's can be defined as a cellular automata [1]:

$$MPM = <PE,R,T,S,I,O>$$

Where: *PE* is a set of processing elements, R is a relation defined on $PE \times PE$, S is a set of states ($I, O \subseteq S$), and T is a transition function. R defines the network topology, while T drives the flow of the information and the computation paradigm of the MPM.

The fast development of the hardware requires the design of new programming languages suitable for the implementation of parallel algorithms on MPM's. They must support both parallel data-structures and instructions, to fit data and parallel computation with the hardware architecture. Moreover, they must provide an efficient and natural handling of concurrent processes, especially whenever MIMD or M-SIMD computation is performed.

Pictorial languages have been developed for mesh architecture; e.g. the Parallel Pascal [2,3] has been designed and implemented for the MPP machine and a C-dialect is used to program the CLIPx-series machines [4]. Some progresses have been made in the design of high level languages in order to implement high level image processing functions (image understanding) on multi-processors machines. Connection LISP has been implemented to program the Connection Machine [5]. Concurrent PROLOG has been developed inside the Japanese fifth generation computing project. It allows searches to be performed in parallel in order to compute billion of logical inferences per second. It is the basis for the kernel language ICOT's parallel inference engine [6]. Pictorial languages have also been developed in order to handle high level vision problems. They allow to define pictorial objects and operations on them, as well icon-guided navigation throughout the pictorial data bases [7,8,9]. Object oriented parallel languages are recently under development; e.g. pSather [10] is part of the software environment of the multi-processor machine CNS-1 [11].

In this report the definition and the properties of the *contexts*, as a new paradigm to handle concurrency on MPM systems, are also given.

MPM's may be also thought as computers with intelligent memory, in the sense that each memory cell contains data and operates on them. Data may be sent from one cell to another. Contexts may be considered as a partition of this intelligent memory space, in which finalized computation is performed.

During a parallel computation several contexts may be active and all the data and instruction defined on them may be homogeneous or not, depending by the computational paradigm (SIMD, MIMD, MSIMD). PE's belonging to the same context contain homogeneous data and cooperate to the execution of a common process. In this sense the elaboration is data-driven.

If the computation is SIMD then the opening of a context, x, causes the disabling of all the PE's not in x and the enabling of all the PE's in x. A more complex synchronization mechanism is present if the computation is MIMD, and M-SIMD.

In the case of vision algorithms *contexts* select the subset of processors on which pictorial operations are executed. From this point of view they are, also, treated and defined as pictorial data, that can represent both local and global vision elements.

Contexts have been firstly introduced in the Pyramid C Language (PCL) [12,13], as object variables, to handle concurrent processes on the pyramid machine: PAPIA [14,15,16]. The PCL is an extension of the C language, where parallel instructions and hierarchical-data-structures, oriented to image analysis computation, are included. PCL allows the execution of asynchronous processes, when the processors of the pyramid layers operate in Multy Single Instruction Multy Data (M-SIMD) mode.

Contexts variables are also included in the visual language PICL (PIctorial C Language)[17], to program the HERMIA machine (Heterogeneous and Reconfigurable Machine for Image Analysis) [18]. The PICL language is an extension of PCL to reconfigurable networks machines. It includes pictorial data and pictorial operators, and reconfiguration network instructions.

Informally, contexts select subsets of processors of an MPM, that are enabled (disabled) by the *open* (*close*) context statement. The list of processes and variables declared between the *open*(C)...*close*(C) statements are allocated on the context C. The execution of a list of processes $\{P_1,P_2,...,P_N\}$ related to a set of contexts $\{C_1,C_2,...,C_K\}$ may be sketched as follows:

"two processes P_1 and P_2, allocated on C_1 and C_2, may be executed asynchronously only if $C_1 \cap C_2 = \phi$, i.e. the processors addressed by the two contexts are set-disjoint and the *open*/*close* statements of C_1 and C_2 are not interleaved in the program sequence, i.e. *close*(C_1) precedes *open*(C_2)".

Note that in case of interleaved contexts, a dependence relation may hold between the processes, and the execution of the processes can't be asynchronous. In this report a dependence graph mechanism is introduced to synchronize processes declared inside different contexts.

Figure 1 shows an example of relation between processes and contexts. The processes P_1, P_2, and P_4 or P_1, P_3, and P_4 may run asynchronously, because they are allocated on three not intersecting contexts. Note that this condition is only necessary, in fact the *open*/*close* statements of the three contexts must be also not interleaved. The processes P_2 and P_3 must be synchronized, because they are allocated on the same context (C_2).

The correspondence between context and network topology is very useful to express computation in vision problems. For example, the low level vision phase of an image analysis procedure is often restricted to local computations; which are performed on disjoint, or partially shared, sub-scenes. The results of such computation may define sub-scenes of

interest to be used at the higher levels of the analysis. For this purpose sets of processors (disjoint or partially shared) are assigned to each sub-scene to solve asynchronously region tasks.

Higher performance is expected, whenever the best match between contexts and processes is reached and the network topology fits with the data structure. From this point of view contexts are, also, treated and defined as data, they may be assigned to a constant-context, or to an expression of contexts. As already said, they are dynamically enabled and disabled by means of *open*/*close* statements.

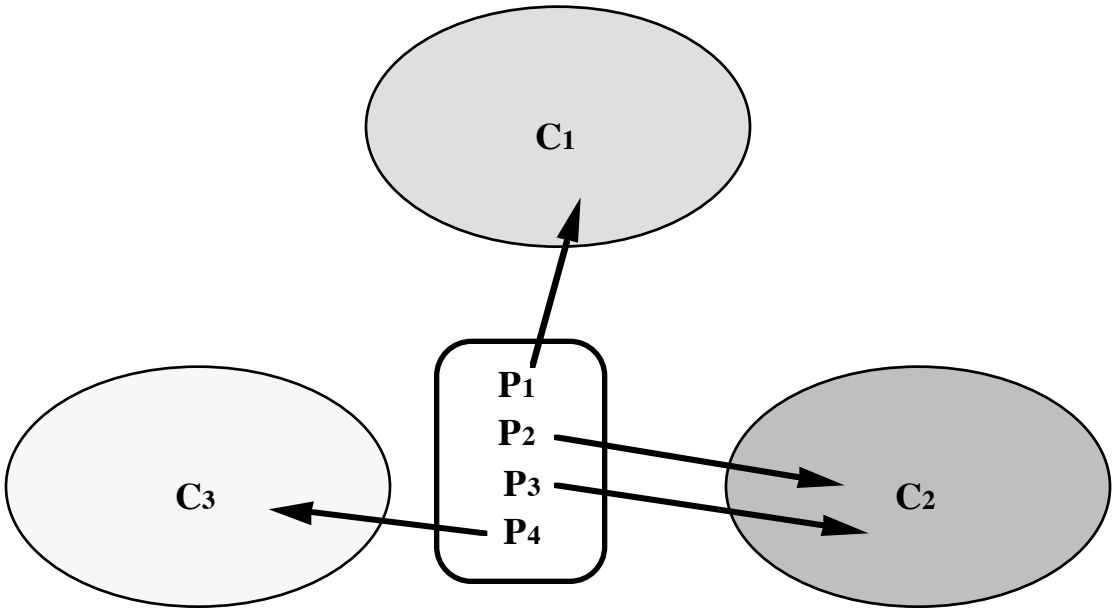


Figure 1. The processes P_1, P_2, and P_4 or P_1, P_3, and P_4 may run asynchronously. The processes P_2 and P_3 must be synchronized.

Contexts are also useful to control the computation of Distributed Co-operating Systems (DCS). Element of a DCS is a set of states, S, representing processes (A), models (M), and data (D). For example, the set D may represent input and output *data*, collected by sensors or produced by a given computation, M may represent *models* of the system environment, elements of A may be algorithms. Here, the term *algorithm* has a wide meaning, it could be a single algorithm or a sequence of processes. Moreover, each algorithm could run on a set of processing units connected in a given topology. Two set of states: $I \subseteq D$ (the set of input states), and $O \subseteq D \cup M$ (the set of output states) are finally introduced to enable communication between a DCS and the environment (*World*). The dynamic evolution of the DCS's is described by mean of a set $\Delta \equiv \{\Delta_1, \Delta_2, ..., \Delta_i, ..., \Delta_K\}$ of transition functions. In case of DCS dedicated to information fusion, examples of transition functions are:

- data / algorithms $\Delta_1: \mathcal{P}(D) \rightarrow \mathcal{P}(A)$
- algorithms / data $\Delta_2: \mathcal{P}(A) \rightarrow \mathcal{P}(D)$
- models / algorithms $\Delta_3: \mathcal{P}(M) \rightarrow \mathcal{P}(A)$
- algorithms / models $\Delta_4: \mathcal{P}(A) \rightarrow \mathcal{P}(M)$
- algorithms / algorithms $\Delta_5: \mathcal{P}(A) \rightarrow \mathcal{P}(A)$

Such computational model can be formally represented by means of the automata DCS=<S,I,O,Δ,W>. The computation will evolve on the basis of the values taken by a mask function $W:S\times S\rightarrow\{0,1\}$. When W(X,Y)=1 for the transition $\mathcal{P}(X)\rightarrow\mathcal{P}(Y)$ and an appropriate *firing* condition is verified, then information flows from X to Y. The nature of the information depends on the sets X and Y.

Contexts in DCS represent sub-sets of states and transition functions on which to perform the computation (asynchronously or not). In Figure 2, an example of a DCS, with related contexts, is given. In this case, contexts individuate sub-graphs of a DCS (states and transition functions). The dependencies among contexts are also represented by transition functions.

The machine M-VIF (Machine Vision Based on Information Fusion) is an example of DCS [19]; it has been designed to integrate visual information collected by distributed sensors processed by distributed processing units. The visual computation is based on information integration, and can be formulated in terms of functional modules. Context variables will be also implemented on the iconic distributed language VIVA (Visual Interface for Vision Algorithms), that has been designed to program the machine M-VIF [20].

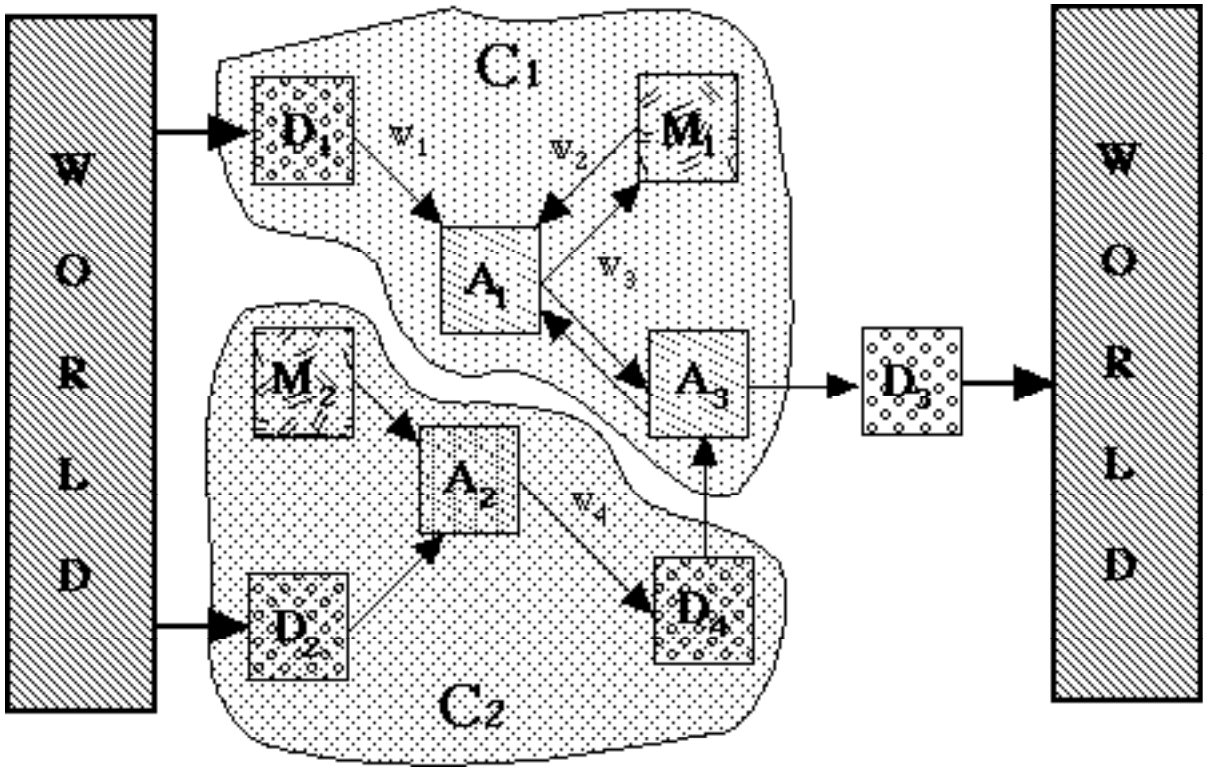


Figure 2. An example of DCS and its contexts.

A formal definition of context is presented in Section 2. Properties of contexts are given for some classes of machine-architecture, widely used in image analysis in Section 3. A model of

concurrence, based on the context-operators, and related properties will be described in Section 4. An example of PICL algorithm with context is given in Section 6. Final remarks are given in Section 5.

2. Definition of context

In the following, PE denotes the set of all PE's of a MPM. Given an arbitrary set A, $\mathcal{P}(A)$ denotes the set of parts of A, while $\tau=\mathcal{P}_2(PE)$ denotes the set of all unordered pairs of elements of PE, i.e. all undirected links between processing elements. Let x and y be elements of PE then their pair will be indicated by [x,y], note that [x,x] is a possible pair.

Given the space $<\mathcal{P}(PE), \mathcal{P}(\tau)>$, then the Space of Contexts is the sub-space:

$$U=\{<X,\sigma> : X\in\mathcal{P}(PE), \sigma\subseteq\mathcal{P}_2(X)\}$$

U is a partial ordered lattice with respect to the operations $\cap$ and $\cup$.

For a given $\tau'\in\mathcal{P}(\tau)$ the element $<PE, \tau'>\in U$ is called the Universe Context (UC). In the case of a MPM, an UC represents the set of all the PE's connected according to a given architecture topology.

Given an UC the set of contexts (CN) is then defined as follows:

$$CN(\tau')=\{<X,\sigma> : X\in\mathcal{P}(PE), \sigma\subseteq\mathcal{P}_2(X)\cap\tau'\}$$

From the previous definition it follows that $U=CN(\tau)$. An element of a $CN(\tau')$ is named context related to the given UC. Therefore in a context the interconnections between the corresponding PE's are maintained according to the MPM network topology. A context is said to be empty iff $\sigma=\phi$. This definition is justified by the fact that in this case the network topology is completely destroyed.

EXAMPLE 1. Let it be $PE = \{PE_1, PE_2, PE_3, PE_4\}$, $\tau = \{[PE_i, PE_j] : \text{for } i,j=1,2,3,4\}$ the set of possible inter-processors links, and $\tau' = \{[PE_1, PE_2],[PE_1, PE_3],[PE_2, PE_4]\}$, then the list of possible contexts is derived from the application of the previous definition. Below, a few of them are listed:

$C_1 = <\{PE_1, PE_2\},\{[PE_1, PE_2]\}>$
$C_2 = <\{PE_1, PE_3\},\{[PE_1, PE_3]\}>$
$C_3 = <\{PE_2, PE_4\},\{[PE_2, PE_4]\}>$
$C_4 = <\{PE_1, PE_2, PE_3\},\{[PE_1, PE_2], [PE_1, PE_3]\}>$
$C_5 = <\{PE_1, PE_2, PE_4\},\{[PE_1, PE_2], [PE_2, PE_4]\}>$
$C_6 = <\{PE_1, PE_2, PE_3,PE_4\},\{[PE_1, PE_2], [PE_1, PE_3], [PE_2, PE_4]\}>$

Note that $C_6 = UC$. Figure 3 shows the contexts listed above.

Given a subset X of *PE* the set $\Pi_X = \{x : x \in X \text{ and } \exists\ y \in PE\text{-}X$ such that $[x,y]\}$ represents those elements of X connected to the complement of X. In the case of the EXAMPLE1, if $X=\{PE_2, PE_4\}$ then $\Pi_X = \{PE_2\}$. This definition has been introduced to define the complement of a context.

A set of operations, *sop*, can be defined on *CN*. The introduction of context operators allows us to treat contexts as object dynamic variables in a programming languages. This feature is very useful in vision applications implemented on MPM or CDS, in fact contexts could allow us to assign processors to spatial shapes, or to processes, according to a certain level of the vision tasks.

Given two contexts (X,σ) and (Y,γ), the following basic *sop* are defined on *CN* :

union	$<X,\sigma> \mid <Y,\gamma> = <X \cup Y, \sigma \cup \gamma>$
intersection	$<X,\sigma> \,\&\, <Y,\gamma> = <X \cap Y, \sigma \cap \gamma>$
complementation	$\sim<X,\sigma> = <(PE\text{-}X) \cup \Pi_X , \sim\sigma)$

Operations of difference (-) and symmetric difference (\) are defined starting from the previous definitions, by using the usual set-definitions. The *CN* maintains the partial ordered lattice structure with respect these operations.

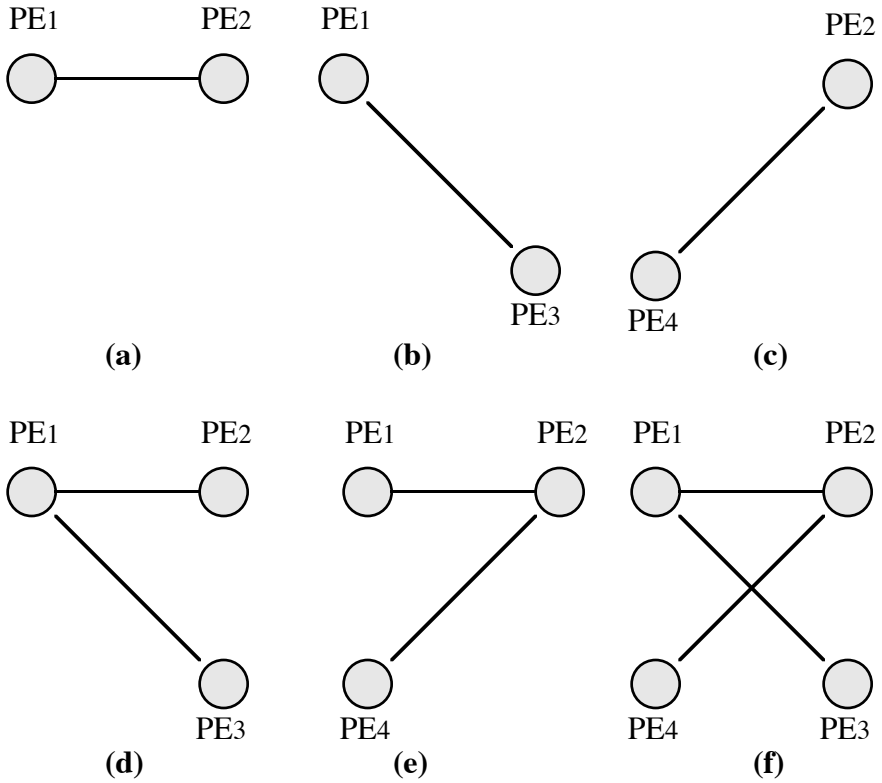


Figure 3. Drawing of the contexts defined in the text: a) C_1; b) C_2; c) C_3; d) C_4; e) C_5; f) C_6.

EXAMPLE 2. Referring to EXAMPLE 1, consider the two contexts:

$$C_1 = <\{PE_1, PE_2\},\{[PE_1, PE_2]\}>$$
$$C_2 = <\{PE_1, PE_3\},\{[PE_1, PE_3]\}>$$

then:

$$C_1 \cup C_2 = <\{PE_1, PE_2, PE_3\}, \{[PE_1, PE_2], [PE_1, PE_3]\}>$$
$$C_1 \cap C_2 = <\{PE_1\},\phi >$$
$$\sim C_1 = <\{PE_1, PE_2, PE_3,PE_4\},\{[PE_1, PE_3], [PE_2, PE_4]\}>$$

See also Figure 4.

The design of suitable network topologies has been one of the main problems, since parallel computers have been designed and implemented. A suitable network topology should optimize the mean data paths length , that implies a closeness between data and processes acting on them. For example, early vision needs *pixel* and/or *local* computation, i.e. data communication is performed between nearest PE's; in this case *mesh* or *linear* topologies have proved to be satisfactory.

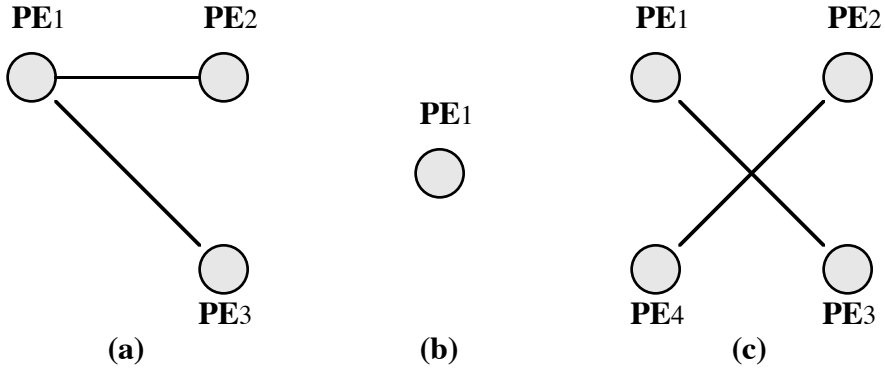


Figure 4. Graphical representation of the *sop* 's in EXAMPLE 2:
a) $C_1 \cup C_2$; b) $C_1 \cap C_2$; c) $\sim C_1$.

The previous considerations suggest as interesting for computation purposes those contexts that preserve some of all properties of the UC topology (i.e. of the interconnection network of the MPM or CDS) after the application of a *sop*. In the following these contexts are named *Basic Context*, BC, and their class is denoted by *BC*. The use of BC's, whose processors are connected in a topology isomorphic to that of the MPM (for example in a mesh contexts are all sub-meshes), may simplify the control of asynchronous processes, as a matter of fact the control-unit is usually efficiently designed to drive a specific network topology.

3 Architecture topologies and contexts

In this section formal definitions of *BC* are provided for a class of common network topologies, which are used in machine vision problems [21,22,23,24,25,26]. Most of them are based on PE's, which are interconnected with a static topology (the links, connecting the PE's are physically and logically defined at the design stage). The network topology may range from the linear array to the hypercube (see Figure 5).

Multi-processor architectures can be represented by a graph, nodes of which are the PE's, and the arcs connect linked PE's. Spatial data are partitioned among the PE's depending on their density (large or low granularity). The partition strategy depends on the algorithm and the computation paradigm used:

- SIMD (Single Instruction Multi-Data). The same instructions set is performed on different data. In this case data are partitioned in sub-sets of equal size. In the case of high granularity each PU could correspond to one pixel value.

- MIMD (Multi-Instructions Multi-Data). Different sets of instructions act on different data. In this case the partition of the data among the PU's depend on the specific algorithm.

- M-SIMD paradigm derives from the combination of the previous ones, it can be useful in order to describe the computation of heterogeneous machine vision systems, composed of different functional hardware modules.

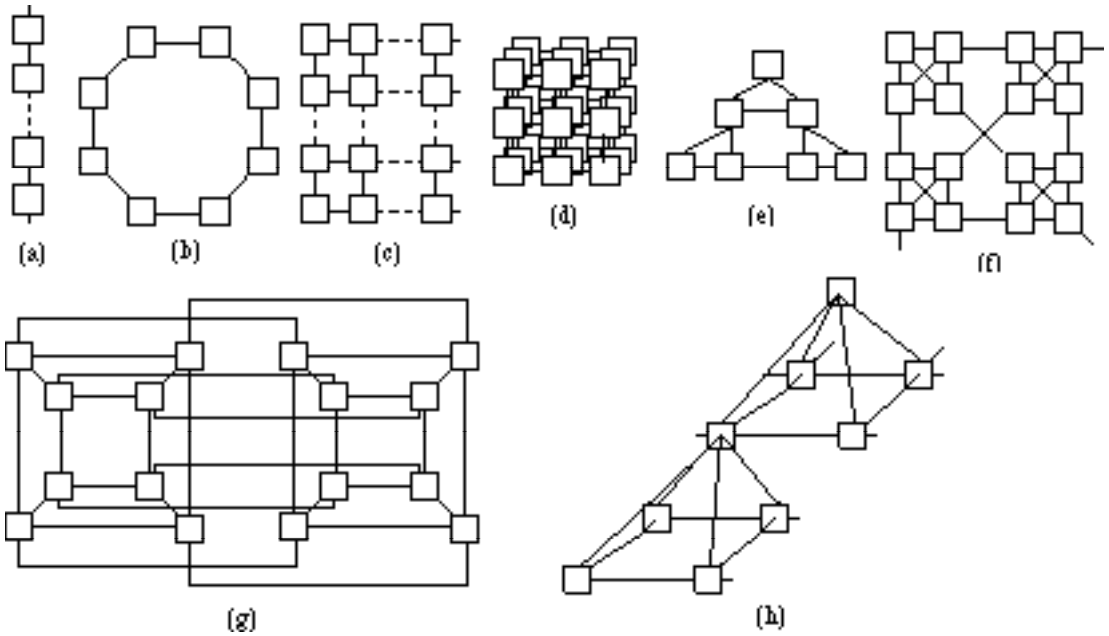


Figure 5. Examples of network topologies: (a) linear; (b) ring; (c) 2D mesh; (d) 3D mesh; (e) tree; (f) WH-recursive network; (g) hypercube; (h) pyramid.

Image analysis algorithms, implemented on multi-processor machines, require an heavy navigation of data among PE's, and the mapping of the data on the PE's should minimize such overhead. Static topologies are tuned only for a given class of problems. Full connection could be the solution for data communication, however the actual limits of the VLSI technology do not allows high link density (for example, 1000 links per processor). Full reconfigurability of the network topology seems to be more realistic, and efforts are now done in this direction. In fact, run time network reconfiguration may allow the tuning of the network path-ways with the kind of data exchange foreseen by a given algorithm.

Contexts are dynamic variables, and this feature allows us to implement and express a network reconfigurability that depends on the current status of the processes. In fact, the opening statement determine the network topology, depending on the context declaration and assignment. Among contexts, BC's are of some interest because of their closure properties.

2.1 Contexts in a linear array

A linear array, LA, is an ordered list of processing elements:

$$LA = (PE_1, PE_2, ..., PE_i, ..., PE_n)$$

Each PE has one input register, I_k, and one output register, O_k, the inter-processor communications are performed by means of these registers. A PE_k may communicate with PE_{k-1} and PE_{k+1} respectively. A linear array may be, also, thought as a sequence. The UC is defined on the whole array, with $\tau' \equiv \{[PE_k, PE_{k+1}] : 1 \le k < n\}$. An LA may be considered as ring array by changing the previous definition of $\tau' \equiv \{[PE_k, PE_{(k+1) mod\ n}] : 1 \le k < n\}$.

Def.1. A subset of UC is said Basic Context, BC, iff its PE's are interconnected in a linear sub-array:

$$BC(\tau') = \{ \langle X, \sigma \rangle : \sigma \subseteq \tau' \}$$

where

$$X = \{PE_k : 1 \le k_1 \le k \le_2 \le n\}$$

Then BC on LA are one dimensional discrete intervals. From the definition it follows that BC=UC iff $k_1 = 1$ and $k_2 = n$.

In the following, a BC will be determined in a unique way by X in order to simplify the notation, in fact σ will be always a sub-set of τ'. The following properties may be easily proved:

Prop.1. Let $A, B \in BC$ then $C = A \& B \in BC$.

Prop.2. Let $A, B \in BC$ then $C = A | B \in BC$ iff one of these conditions is satisfied:

a) $A \& B \neq \phi$;
b) $A = \{ PE_k \mid 1 \le k_1 \le k \le k_2 < n \}$ *and* $B = \{ PE_k \mid k_2 + 1 \le k \le k_3 \le n \}$

Prop.3. Let $A \in BC$ then $C = \sim A \in BC$ iff one of conditions is satisfied:

a) A=UC or $A = \phi$;
b) $PE_1 \in A$
c) $PE_n \in A$

2.2 Contexts in a mesh array

Mesh arrays, MA, are two-dimensional array of PE's:

$$MA = \{PE_{i,j} \mid 1 \le i \le n,\ 1 \le j \le m\}$$

In the following, without loosing in generality, square MA's are considered (n=m).

Mesh arrays may also be defined as the Cartesian product of two linear arrays, LA_1 and LA_2, ($MA=LA_1 \times LA_2$). To each $PE_{i,j}$ a set of four ($NF_{i,j}$) or eight ($NE_{i,j}$) neighbors can be associated, connected for example by I/O registers; they are defined as follows:

$NF_{i,j} = \{PE_{i-1,j}, PE_{i+1,j}, PE_{i,j-1}, PE_{i,j+1}\}$
$NE_{i,j} = \{PE_{i-1,j-1}, PE_{i-1,j}, PE_{i-1,j+1}, PE_{i,j-1}, PE_{i,j+1}, PE_{i+1,j-1}, PE_{i+1,j}, PE_{i+1,j+1}\}$

The neighbor definitions of a mesh arrays match very well with spatial data, allowing higher performance in the computation of low level vision algorithms, as already pointed out. Therefore, the definition of BC's on a MA, and the condition for which *sop*'s are closed, i.e. $\forall$ x,y$\in$*BC* x*sop* y$\in$*BC*, are interesting on machine vision applications.

Two suitable definition for τ' are NF and NE:

$$\tau' = \{[PE_{i,j}, PE_{i+r,j+k}] : 1 \le i,j < n,\ 0 < r+k \le 1\} \text{ in case of NF}$$
$$\tau' = \{[PE_{i,j}, PE_{i+r,j+k}] : 1 \le i,j < n,\ 0 < r+k \le 2\} \text{ in case of NE}$$

The definitions and properties given below hold for both definitions.

Def.2. Basic Contexts of an MA are:

$$BC = \{ PE_{i,j} \mid 1 \le i_1 \le i \le i_2 \le n,\ 1 \le j_1 \le j \le j_2 \le n\}$$

BC of an MA are discrete rectangles of the plane. The projection of a BC on LA_i (i=1,2) is indicated by BC_i. We denote by BC_i the set of basic context of LA_i. It easy to prove that:

Prop.4. Let A,B$\in$*BC* then C=A|B$\in$*BC* iff the following conditions are satisfied:

a) $A_i = B_i$;
b) $A_j \mid B_j \in BC_j$ where i,j=1,2 and i$\neq$j.

Prop.5. Let A , B$\in$*BC* then C=A&B$\in$*BC* .

Prop.6. Let A$\in$*BC* then C=~A$\in$*BC* iff one of conditions is satisfied:

a) A=UC or A=ϕ ;
b) $A_i=LA_i$ and $\sim A_j \in BC_j$) where i,j=1,2 and i$\neq$j.

The conditions stated in Proposition 6 imply that the complement of C$\in$*BC* is not usually in *BC* . The operators "-" and "\", may be defined by means of the union, the intersection and the complementation. It is easy to show (see Figure 6a) that A-B$\in$*BC* iff one vertical or horizontal slice of A is contained in B; the condition A\B$\in$*BC* holds iff A&B=ϕ and they

have two edges adjacent of equal size (see Figure 6b).

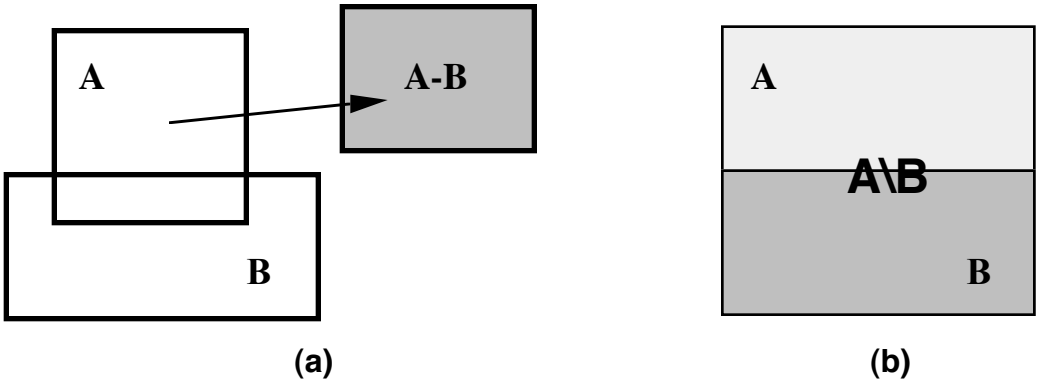


Figure 6. a) Geometric condition to have A-B∈*BC* ; b) geometric condition to have A\B∈*BC*.

2.3 Contexts in a pyramid array

Formally a regular pyramid array, PY, is a set of processors addressed by 3-tuples (i,j,k), representing the row, the column, and the level respectively. The three co-ordinates are related as follows:

$$PY = \{ PE_{i,j,k} : 0 \le k < L \text{ and } 0 \le i < 2k \text{ and } 0 \le j < 2k \}$$

The set PY is also known as the hierarchical domain of the pyramid with L levels [11], see Figure 7a.

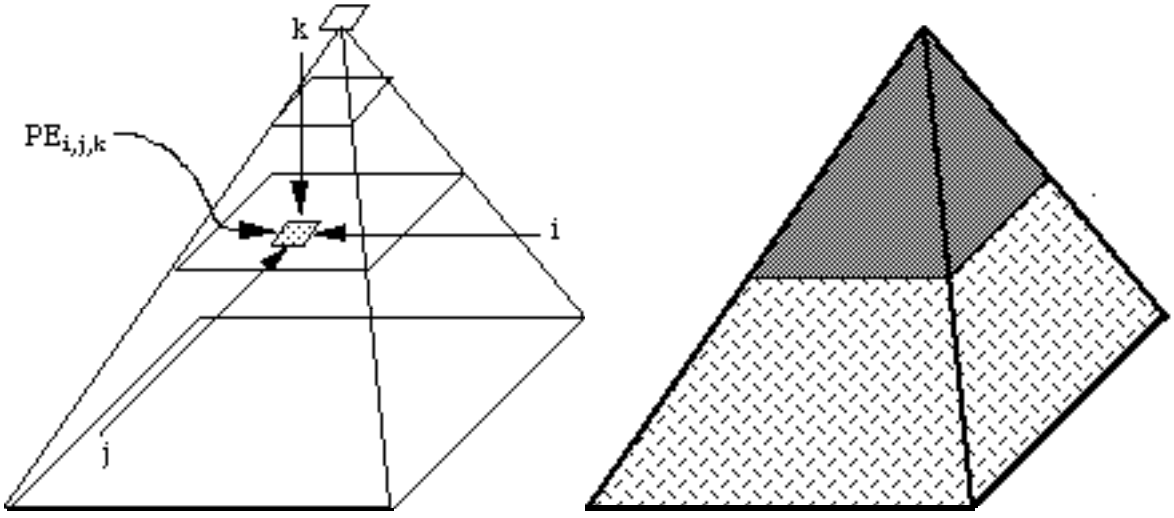


Figure 7. a) The regular pyramid; b) stacks and trunk of pyramids.

Each cell corresponds to a single PE. Each PE has a set of neighbors, SN, linked to it (see Figure 8). They are respectively the father (13), the four sons (9,10,11,12) and four

(2,4,6,8) or eight brothers (1,2,3,4,5,6,7,8), depending on the connection degree in each layer, that usually has a mesh topology. Therefore the number of elements in SN ranges from 9 to 13.

Two relations τ' can then be defined to characterize the architecture topology of a PY:

$$\tau' \equiv \{x \in PY : [x,i] \text{ for } 1 \le i \le 13\}$$
$$\tau' \equiv \{x \in PY : [x,2], [x,4], [x,6], [x,8], [x,9], [x,10], [x,11], [x,12], [x,13]\}$$

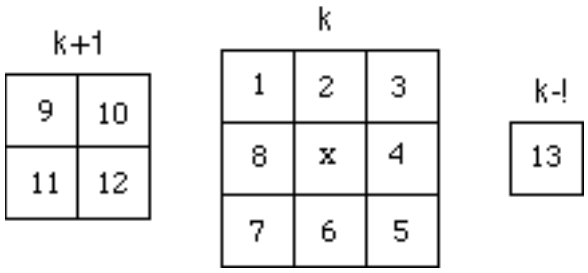


Figure 8. The network topology of the PY.

Pyramids array can be considered as stack of trunks of pyramid (see Figure 7b), moreover meshes are trunk of pyramids of height 1. Starting from this consideration it turns useful to define BC of a PY as a sub-set of PE's connected in a topology, that represents trunk of pyramids:

Def.3. A subset of UC is said Basic Context, BC, iff its PE's are interconnected as follows:

$$BC = \langle i,j,k,n_1,n_2,r \rangle = \{(x,y,z) : k \le z < k+r, \text{ and } i \le x \le i+n_1-1,\ j \le y \le j+n_2-1 \text{ for } z=k \text{ and } 2i(z-k) \le x \le 2(z-k)(i+n_1)-1,\ 2j(z-k) \le y \le 2(z-k)(j+n_2)-1\}$$

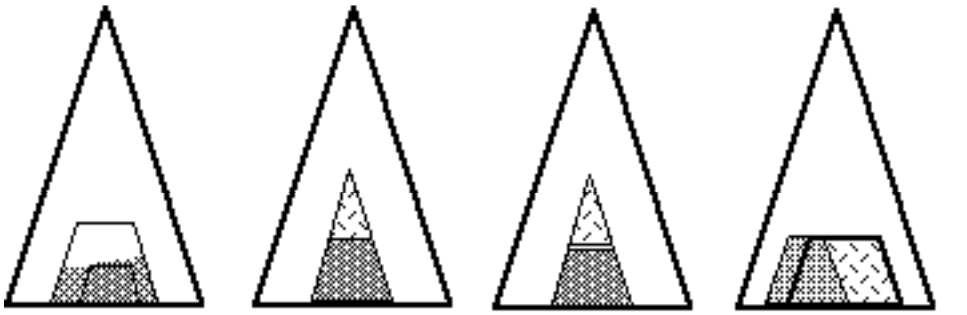
Here i and j are the coordinates of the top-most and left-most PE on the k-th layer, r is the number of layers (deep) of the context, n_1 and n_2 are the linear dimensions of k-th layer. From this definition it follows, for example, that UC=<0,0,0,1,1,L>, is the whole pyramid array with L layers. Also a single PE may be considered as a BC, <i,j,k,1,1,0>, it follows that all the subsets of UC may be generated from the BC$\in$*BC*.

In the following, given a context A, A_k represents the k-th layer. Moreover a pyramid, A, of depth L can be recursively augmented by linking, according to rule τ', A_L to a new layer A_{L+1}. The following closure properties are easily derived from **Def.3**.

Prop.7 Given A, B$\in$BC with deep L_A and L_B then C=A|B$\in$*BC* iff one of the following properties holds (see Figures 9):
a) A$\supseteq$B with deep L_A or B$\supseteq$A with deep L_B ;
b) $A_0 \equiv B_{L_B}$ or $B_0 \equiv A_{L_A}$ with deep L_A+L_B-1;
c) $A_0 \equiv B_{L_B+1}$ or $B_0 \equiv A_{L_A+1}$ with deep L_A+L_B;
d) $A_0 \cap B_0 \neq \phi$, $L_A=L_B$ and $A_L \cap B_L$ is a common layer of A_L and B_L.

Prop.8 Let A,B∈*BC* then C=A&B∈*BC* .

Prop.9 Let A∈*BC* then C=~A ∈*BC* iff
a) A=UC or A=ϕ;
b) A=<0,0,0,1,1,r> or A=<0,0,k,2^k,L-k>



(a) (b) (c) (d)
Figure 9. Geometrical conditions to have the *sop* "|" closed.

The operators - and \, may be defined by means of the union, the intersection and the complementation therefore the contexts obtained as A-B and A\B are not usually in *BC*. Their closure can be derived again by using the same kind of geometric considerations that have been made to demonstrate the previous properties.

2.4 Context in the hypercube array

Hypercube arrays, HY, are sets of PE's in a d-dimension binary space H_d, they may be considered as a graph, nodes of which are the PE's and the arcs are the connections between them. Each node is a point in H_d, and it is represented by a binary d-tuple $(x_1,x_2,...,x_d)$ with $x_i \in \{0,1\}$, therefore:

$$HY = \{ PE_x : x \equiv (x_1,x_2,...,x_d) \in H_d \}$$

In the following PE_x will be identified by its address, x. Two PE's $x \equiv (x_1,x_2,...,x_d)$ and $y \equiv (y_1,y_2,...,y_d)$ are connected iff their binary representation differs for a single bit, i.e. their Hamming distance, h, is equal to 1; it follows:

$$\tau' = \{[x,y] : h(x,y) = 1\}$$

Basic contexts of HY are hypercube of dimension m≤d (H_m), and BC=UC, for m=d.

The definition of hypercube in H_d does not allow one to define a BC≠UC, that is also an hypercube in H_d with dimension m≤d. More formally:

Def.4 BC's of an HY in H_d are those subsets, H_m, of PE's such that:

$$\exists\ k_1, ..., k_{d-m} \in \{1,..,d\}: \forall\ x,y \in BC \Rightarrow b^x_{k_h} = b^y_{k_h} \quad \text{for } h=1,..,d-m$$

where b^x_i is the i-th bit of the number representing x. From this definition it follows that:

- UC is an HY in H_d i.e. the whole hypercube machine. The set of BC is in H_m and is denoted by BC_m. It is easy to see that $|BC_m|$=**bino**(d,k)x2^{d-m}, here **bino** indicates the binomial coefficient. The whole *BC* is the union of the BC_m, $0 \le k \le d$, then its cardinality is equal to 3^d;
- a basic contest BC$\in$*BC* is characterized by n-m equations of the form $x_i=b_i$ where $i \in \{1,..,d\}$ and $b_i=0,1$.

In the following, given a basic contest A, we denote by I(A) the set $\{k_1, ..., k_{d-m}\}$, and by dim(A) the integer m for which A$\in BC_m$. A basic context can be also described by the d-tuple:

$$(*,...x_{k1}, *...*,x_{k2},*...*,x_{kd-m},*,x_d)$$

Where, the "*" stands for any value {0,1}. The following properties hold for the *sop* defined in *BC*.

Prop.10 Let A,B$\in$*BC* characterized by the equations $\{y_j=b_j\}$ and $\{x_i=b_i\}$ respectively then A|B$\in$*BC* iff one of following conditions is satisfied:

a) A$\supseteq$B or B$\supseteq$A
b) I(A)=I(B) and $\exists$ i$\in$I(B) such that $x_i \neq y_i$

Prop.11 Let A,B$\in$*BC* then A&B$\in$*BC* .

Prop.12 Let A$\in$*BC* then ~A$\in$*BC* iff dim(A)=d-1.

4. A concurrency model based on contexts

The implementation of parallel programming languages on MPM's requires a careful design of the concurrence model. An efficient solution of a vision problems requires the synchronization of several tasks acting on an image. Concurrent tasks may act on the whole image or on the same area, or they can perform asynchronous computation on different partitions of the image. The results are then combined to perform the next steps of the analysis. Moreover, more tasks may interact by exchanging data and instructions.

Synchronization is required whenever a critical section is present in a concurrent program. A critical section is a sequence of statements that must appear to be executed as an atomic operation. Synchronization can by acquired enabling processes to communicate via shared variables, for example semaphores [27] are integer variables introduced for this purpose. Another model uses the monitor, which are a structure consisting of variables representing the state of some resource, procedures to implement operations on that resource, and initialization code. Monitors are used in Concurrent Pascal [28] and Modula [29]. In pSather object language [10] the synchronization is also handled with a mechanism related to monitors, which is named *gates*. Gates are implemented on both shared and distributed

memory multiprocessors.

Several concurrence-models have been proposed in the literature; efforts to express the dependence between parallel tasks in structured programming languages have been proposed. Cobegin and coend [30] are structured way of indicating a set of statement that can be executed in parallel. The statement:

cobegin S_1; S_2; S_3;...S_n; **coend**

causes statement S_1, S_2, S_3,...S_n to be executed concurrently. It is possible to nest concurrent process. For example the statement S_3 could be:

cobegin S_{3_1}; S_{3_2}; **coend**

The processing continues beyond the coend only after all the S_i's have completed execution. Any concurrent program can be represented by a process flow graph , which is acyclic and directed. Nodes are the processes and the edges represent execution constraints (i.e. and edge from i to j indicates that process j cannot begin execution until process i has finished execution [31].

Fork and Join statements are an alternative model of concurrence [32]. The fork is similar to a procedure call in the sense that it enables the commencement of a particular routine. However, unlike a procedure call, the calling process continues execution. The invoking process can synchronize with termination of the forked process by executing the join statement. When the invoking process encounters the join statement, it cannot continue execution until the invoked process has terminated. For example let A and B be two concurrent programs, then their execution (active programs) may be sketched as below:

program A	program B	active programs
…	…	A
fork B;	…	A
…	…	A, B
join B;	…	A, B
…	…	B
…	end;	B
…		A

Fork and Join statements are used also in the UNIX operating system (trademark of Bell Laboratories).

Concurrent Programming Languages can be, also, classified on the basis of its concurrence model as:

- Procedure oriented languages. The process interaction is based on shared variables. The processes have access to every data object. They are tailored for shared memory architecture. Examples of procedure oriented languages are Concurrent Pascal, Modula, Mesa, Edison.

- Message oriented languages. They are built upon the primitives send and receive. Unlike the procedure oriented languages they do not give processes access to every data object. They require fast and efficient communication network (busses and link between the PE's). Examples of message oriented languages are Gypsy, OCCAM, PLITS.

In the case of computer vision algorithms higher performance is expected whenever the best match between *processes* and *sub-scenes* on which they act is reached. For example PICL treats contexts as dynamic variables, and their declaration is:

context <ident> *of* <network-type>;
<network-type>::= pyramid / mesh / linear / cube{d} / ...

Context variables are declared at the beginning of the *main* program. The UC is declared by means of a define statement. In the case of the HERMIA-machine, the declaration is:

context A,B,C of *pyramid*;

defines two context with a pyramid architecture. In this case the context variable is a 6-tuple: <ident>=(x,y,z,l_1,l_2,d) of integer expressions, and the assignment of a context can be made in run time, Moreover, it can be the result of previous computations. For example an edge detector could define as context the set of the PE's in which are stored the pixels of the image borders. On the borders context could be then executed a shape analysis algorithm.

Let UC=(1,1,1,1,1,8), i.e. a pyramid array with 8 levels, then example of assignment are:

A=(2,2,3,8,8,3);
B=(16,16,3,4,4,2);

This assignments are performed inside the UC context, (which is the default context settled at the beginning of the computation), they represent non intersecting trunks of pyramids. Moreover, PICL includes the following context operators:

C = A; C = A&B; C=A | B; C=A-B; C=A\B;

The *open*/*close* statements define the life of a context in parallel algorithm, these operators are defined and act on the context space *CN*.

The declaration of data in a context can be done outside of the context by an explicit declaration:

<struct type> *of* <simple type> <ident> *in* <context ident>;

For example: *pyra of float* P[2,3] *in* C; is an explicit declaration of the pictorial variable P inside the context C. All declaration statements inside a context must be made after the instruction *open*, and are implicit. The following declaration has the same effect as the previous one:

open(C); *pyra of float* P[2,3];.....*close*(C);

In any case the variable P will be allocated dynamically with its context.

Processes running in several contexts exchange information via communication networks, the processes synchronization is modeled by means of *dependency-graphs* in fashion which is similar to the *cobegin... coend* construct.

The following segment of PICL program shows an *open* and *close* of the contexts above declared:

```
.
open(A);
.
P
Q
.
close(A);
.
open(B);
.
S
.
close(B);
.
```

The processes P and Q are executed sequentially inside the context A, a process S can be executed in parallel inside the context B.

The *open*/*close* statement can appear inside a PICL program in the following sequences:

```
open(A);          open(A);          open(A);
ins_A;            ins1_A;           ins1_A;
close(A);         open(B);          open(B);
.                 ins1_B;           ins1_B;
.                 ins2_A;           ins2_A;
open(B);          close(A);         close(B);
ins_B             ins2_B;           ins3_A;
close(B);         close(B);         close(A);
   (a)               (b)               (c)
```

Figure 10. (a) Independent contexts; (b) interleaved contexts; (c) dependent contexts

Three configurations can be considered (see Figures 10a,b,c):

- Independent contexts (Figure 10a). In this case the processes in context A and B may be executed asynchronously.
- Interleaved contexts (Figure 10b). In this case the processes in context B will start after the execution of ins1_A; ins1_B and ins2_A are executed asynchronously; ins2_B is executed after ins1_B. Note that in this case, after the statement close(A) it would be possible to open again A, and to start the execution of the processes in A after the execution of ins2_B.

- Dependent contexts (Figure 10c). In this case the processes in B start after the execution of ins1_A; ins1_B and ins2_A are executed asynchronously; ins3_A is executed after ins2_A. Note that it is forbidden to open A if it has not been closed previously.

Concurrency has been treated in the framework of formal languages by means of the shuffle operator [33], the introduction of the partial ordered multisets (pomsets), as a generalization of the total orders of string, allowed to develop an axiomatic theory of the shuffle and has given the possibility to model concurrence with partial orders [34].

A PICL program can be considered as a sequence of *open/close* statements, and the execution of the processes will be determined by their ordering in the source program. In the following the existing relations between the *open/close* model, formal languages and graphs will be evidenced; o_x and c_x will denote the *open* and the *close* of the context x. An optimization of the implementation of contexts in the PICL language has been reached by using the results of next section.

4.1 Contexts and words

It is possible to characterize the sequences of *open/close* statements in a parallel program as words of a language. Let us denote the finite sets of contexts by the set {1,2,...n}.

Given the opening sets $O=\{o_1,o_2,o_3,...,o_n\}$ and the closing set $C=\{c_1,c_2,c_3,...,c_n\}$, we define the alphabet $\sum=O\cup C$. Moreover, a bijective application, R, from $\sum$ into itself is defined as follows $R(o_i)=c_i$ and $R(c_i)=o_i$.

Def 5. The Context-Language, CL, on the alphabet $\sum$, is the set of words of $\sum^*$:

$$w = a_1a_2a_3...a_m \qquad a_i\in\sum \ \ \forall \ \ i=1,2,...,m$$

satisfying the following conditions:

1) w = xw'y with $x\in O$ and $y\in C$
2) $\forall\ i<j \ \ a_i=a_j\in O \ \exists\ i<k<j$ such that $R(a_k)=a_i$
3) $\forall\ i<j \ \ a_i=a_j\in C \ \exists\ i<k<j$ such that $R(a_k)=a_i$

The condition 2) (3)) states that each open (close) statement must be preceded by its corresponding close (open). Note that the first open statement does not follows constraint 2). In other words, constraints 2) and 3) do not allow nested *open/close* of the same context; this can be considered as a semantic condition. In fact, the interpretation of the word: $o_1o_1c_1c_1$ is ambiguous: *are the two context interleaved, or is one nested into the other*? The following words are allowed: $o_1o_2c_2o_3c_1c_3$, $o_2o_1c_2c_1o_1c_1o_2c_2$.

The following property derives from definition 5.

Prop.13 Let scan $w=a_1a_2....a_m$ from left to right, if $a_i\in O$ then $Occ(a_i,i)-Occ(R(a_i),i)=1$.

Here Occ(x,i) indicates the number of occurrence of x in w when position "i" is reached.

A straightforward parsing algorithm derives from Proposition 13, to test the syntactic correctness of the *open/close* statements in PICL. In fact, for this purpose an array of stacks of dimension |w| is required; the operation are "pop-in" when $a_i \in O$ and "pop-out" when $a_i \in C$, the conditions to be tested are: for each $1 \le i < m$ $0 \le S_i \le 1$, for $i=m$ $S_j=0$ for $j=1,2,...,m$. The computational complexity is then O(|w|).

Prop.14 Given $w=a_1a_2....a_m$ $\exists$ $k<i$ such that $a_k \in O$ $a_i \in C$ $a_k = R(a_i)$ and $\nexists$ $k \le j \le i$ such that $a_j = R(a_i)$.

This property follows directly from Definition 5, the pair (a_k,a_i) satisfying proposition 14 is saidto be *allowed*.

In the next section it will be shown that the parsing of a context word allows also to built in O(|w|) steps the dependency graph, that is the kernel of the synchronization engine in PICL.

4.2 Contexts and graphs

In the following, the dependence graph, DG, is introduced, it is a representation of a context word, $w \in CL$, and its nodes are the occurrences of the *allowed pairs* (o_x,c_x), $x \in CN$ in w.

An *allowed pair* may appear 'n' times in a word, and it will be represented by 'n' separate nodes, that we call bugs nodes, following the same notation used in Petri nets theory [35], here the set of allowed pairs is denoted by OC. Moreover, given a word $a_1a_2....a_m$, we denote the natural ordering by $a_i \prec a_k$ iff $i<k$. In the following, in order to simplify the notation, the nodes of DG will be identified with the contexts, that are opened and closed.

Def.6 Given a $w \in CL$ a dependency graph $DG=<OC,E,\Lambda>$ is associated to it; DG is direct, simple, ordered, and arc-labeled, where:

OC is the bug nodes;

E is the set of arcs, representing the dependence relation, *dep*, defined as follows:

- $\forall$ $x \equiv y \in OC$ x*dep* y, iff:
  a) $c_y \prec o_x$;
  b) $\nexists$ $z \equiv x$ such that $o_z \prec o_x$ and $c_z \prec c_x$ in the word.

This case corresponds to the configuration: $w=...o_y...c_y...o_y...c_y...$

- $\forall$ $x \neq y \in OC$ x*dep* y, iff:
  a) o_y p o_x;
  b) $\nexists$ $z \neq x$ such that $o_z \prec o_x$ in the word.

This case corresponds to the configurations: $w=...o_y...o_x....c_y....c_x...$ or $w=\ldots o_y \ldots o_x \ldots . c_x \ldots c_y \ldots .$

Λ represents the labeling function $\Lambda: E \rightarrow \{r,s,b\}$, where r stays for right, s for sun and b for brother. This function is defined as follows:

$\Lambda(y,x) = r$ iff (see Figure 11a):
a) x*dep* y
b) $c_y \prec c_x$

$\Lambda(y,x)$ = s iff (see Figure 11b):
a) x*dep* y;
b) $c_x \prec c_y$

$\Lambda(y,x)$ = b iff x*dep* y *and* x=y (see Figure 11c).

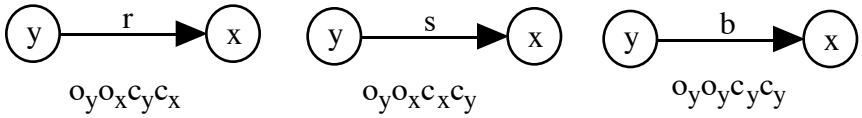


Figure 11 (a) right dependence; (b) son dependence; (c) brother dependence.

Note that the arc x*dep* y with x=y has been introduced, because in this case the contexts x and y address the same PE's. The arcs of a DG are oriented from y to x if x*dep* y. From the algebraic point of view the relation *dep* is a partial order. Figures 12a,b show a context word and the related DG graph.

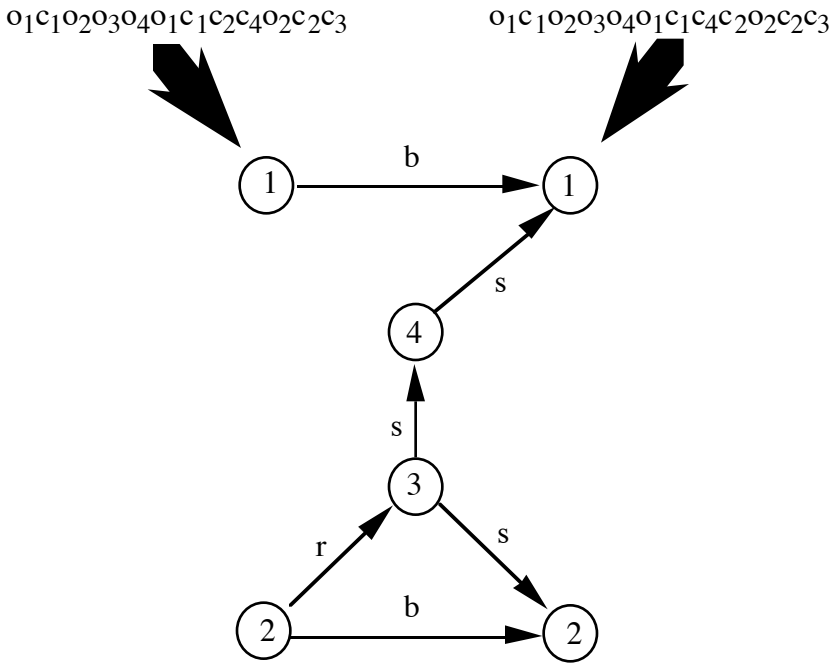


Figure 12. Context words and related DG graph.

From the example in Figure 12 it clear that to a context word corresponds a unique DG graph, but the same DG graph can be derived from a sets of words of CL. Moreover, the same word represents equivalent executions of different processes. In fact, the real computation depends on the processes defined inside each context.

The following properties hold for DG graphs:

Prop.15 $\forall$ x$\in$OC then no more than two z,y$\in$OC exist such that x*dep* y and x*dep* z. This

means that x may be end-node of no more than two arcs in DG.

Prop.16 If x∈OC is end-node of two arcs, then their labels are different and belong to the set {b,r} or {b,s}.

Prop.17 Sub graphs of DG, which contain arcs with the same label represent tomsets (totally ordered multistes).

This result can be demonstrated by an exhaustive analysis on the nine possible combinations of the labels {b,r,s}:

*bb	→	b
br	→	r
bs	→	s
rb	→	r
*rr	→	r
rs	→	{r,s}
sb	→	{r,s}
sr	→	{r,s}
*ss	→	s

The right side of the previous list gives the label of the transitive closure of DG, the lines marked with the '*' prove the assert. Note that the labeling deriving from arc-sequences labeled rs, sb, and sr is ambiguous.

The result in Proposition 17 can be useful to simplify some dependencies among processes defined in different contexts.

5. Implementation notes and examples

In section 4 has been shown that the syntactic correctness of a context word can be performed in O(|w|) steps. During the compilation the DG graph is also derived from the context word related to the source PICL program. This can be done again in O(|w|) steps, by using a two phase parsing algorithms, and taking in account of the definitions given in the previous section.

In the first phase the sets OC is built inside index table, IT, which contains the ordered list of bugs. For each bugs: the name, x, the position, i_x, of o_x, and the position, j_x, of c_x in w are given. In the second phase this ordered table is used to compute E and the labeling of the arcs.

It can be shown that the construction can be done in linear time with the size of IT, because of the natural ordering. In fact, the kind of labeling is retrieved by using the entries of the current bugs in IT with those of the *master* bug, to test the conditions given in the Definition 6. The *master* bug is pointed by the top of a stack, used to store the position of the *open* statement

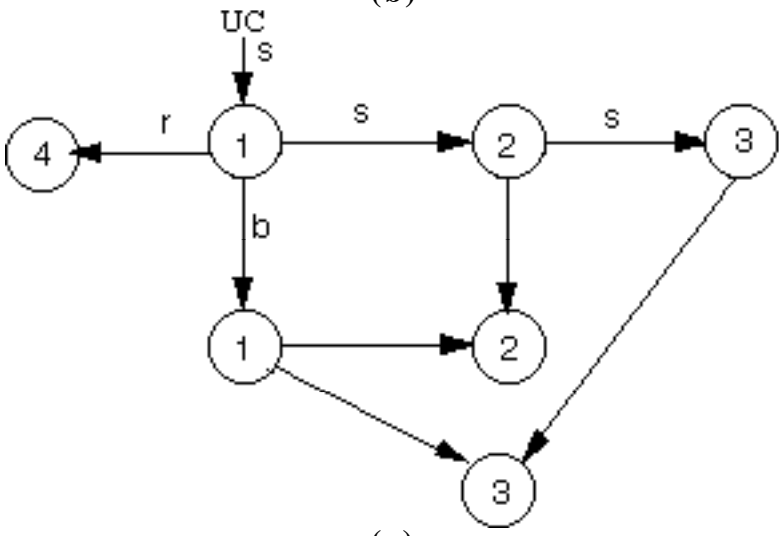
of each x, and to free it after the *close* statement has been read. In Figure 13 an example of how the algorithm works is given.

$w = o_1o_2o_3c_3c_2o_4c_1c_4o_1o_2c_2o_3c_3c_1$

(a)

	x	i	j
1	1	1	7
2	2	2	5
3	3	3	4
4	4	6	8
5	1	9	14
6	2	10	11
7	3	12	13

(b)



(c)

Figure 13. (a) syntactically input context word; (b) IT table; (c) DG graph.

5.1 DG driven execution

Given a context x we indicate with Ins(x) the sequence of instructions that run on x and with Dat(x) the set of data defined on x. In the following the notation [Ins(x),Ins(y)] indicates the asynchronous execution of Ins(x) and Ins(y), their synchronous execution is denoted by (Ins(x),Ins(y)). Four rules drive the execution of the instructions:

R1: Given two contexts x and y such that $\Lambda(y,x)=s$ then:

a) Ins(x) may be executed on Dat(x), while x is open;
b) ins(y) may be executed on Dat(y), while y is open;
c) instructions are executed by following the sequence: (Ins(y),[Ins(y),Ins(x)],Ins(y)).

R2: Given two contexts x and y such that $\Lambda(y,x)=r$ then:

a) Ins(x) may be executed on Dat(x), while x is open;
b) ins(y) may be executed on Dat(y), while y is open;
c) instructions are executed by following the sequence: (Ins(y),[Ins(y),Ins(x)],Ins(x)).

R3: Given two contexts x and y such that $\Lambda(y,x)=b$ then:

a) Ins(x) may be executed on Dat(x), while x is open;
b) ins(y) may be executed on Dat(y), while y is open;
c) instructions are executed by following the sequence: (Ins(y),Ins(x)).

R4: Given two not connected nodes x and y in DG then:

Ins(x) and Ins(y) may be executed asynchronously only if $x\&y=\phi$. Otherwise they must be executed in one of the two sequences: (Ins(x),Ins(y)) or (Ins(y),Ins(x)).

The previous rules may be combined to form more complex sequences that must be satisfied on each context. For example rules **R1** and **R3** or **R2** and **R3** are merged if x depends from two contexts: y and z. Two cases can be considered for each combination:

- $\Lambda(y,z)=\phi$ and $\Lambda(y,x)=s$, $\Lambda(z,x)=b$. In this case the sequence of the instruction is: ([Ins(y),Ins(z)],[Ins(x),Ins(z)],Ins(z))
- $\Lambda(y,z)=r$ and $\Lambda(y,x)=s$, $\Lambda(z,x)=b$. In this case the sequence of the instruction is: (Ins(y),[Ins(y),Ins(z)],[Ins(x),Ins(z)],Ins(z))

The execution of a sequence of contexts has been implemented on the PICL language by a firing technique applied to the corresponding DG graph, in our case the number of token are ≤ 2. The first implementation has been realized on the first prototype of the HERMIA machine in C++ language.

5.2 Examples of PICL algorithms

Below two examples of image analysis algorithms, written in PICL are proposed. Both of them have been implemented on the HERMIA machine; the first algorithm is executed on an emulated pyramid with 8 levels; the second on a hypercube of dimension 4. Both networks have been implemented by using 16 complex nodes of the intermediate module of HERMIA.

The first algorithm computes the discrete convolution of an image, M, of linear size N, by means of the kernel, K, of linear size D, M1 is the resulting convoluted image. The dimension N is a multiple of D, N=mxD. The algorithm is based on the equation:

$$M1 = M \otimes K$$

Where $\otimes$ is the convolution operator. Figure 14 shows the corresponding PICL algorithm.

The identifier "image" and "kernel" are the input files containing the image to be de convoluted and the kernel, while "decon" is the output file containing the de convoluted image. The image, K, represent the duplication of an mxm sub matrix, of linear size D, containing the de convolution kernel. Shift and broadcasting instructions are used inside the context C, which is dynamically allocated with the assignment C=(*,*,MAXH-2,1,1,3). The processors above the level MAXH-2 are abilitated to execute other algorithms.

```
# define MAXH 7, D 4;
# define UC (0,0,0,1,1,8);
main()
   {
     int i, j;
     context C of pyra;
     C = (*,*,MAXH-2,1,1,3)
     open(C);
     mask S[9] in C;
     pyra of float P[2,3] in C;
     ima of float M[128,128], M1[128,128],
                  K[128,128] in C;
     load(M, image, 2); load(K, kernel, 2);
     shift(shift(M1, D, N), D, 4);
     shift(shift(M, D/2, N), D/2, 4);
     S = (000001000);
     for (i = D; i >1; i--);
       {
         P[0] = sum( M*K );
         sendown(=,P,0,2,S);
         M' = P[2];
         shift(M',1,0); shift (M,1,0);
       }
       shift(M1,1,6);  shift(M,1,6);
       shift(M1,1,4);  shift(M,1,4);
     }
     unload(M1,decon,2);
     close(C);
   }
```

Figure 14. PICL program to perform the convolution of an image on an a pyramid array.

The second algorithm performs real time image acquisition and object selection. The total number of PE's is 16; each of them has 6 links; 8 PE's are dedicated to active I/O functions, 4 PE's to segment the input image M, and 4 PE's to perform a matching algorithm to retrieve objects. Figure 15 shows the four contexts that are used to execute the algorithm. In Figure 16 the corresponding PICL algorithm is shown.

C1,C2,C3,C4 are four contexts of type hypercube in H_2, the variable M is the input image, S is the segmented, and P is the shape prototype, M1 is the image displayed. Note that each of them are defined in the appropriate context. The prototype image is supposed already loaded in the memory of the PE's in C3.

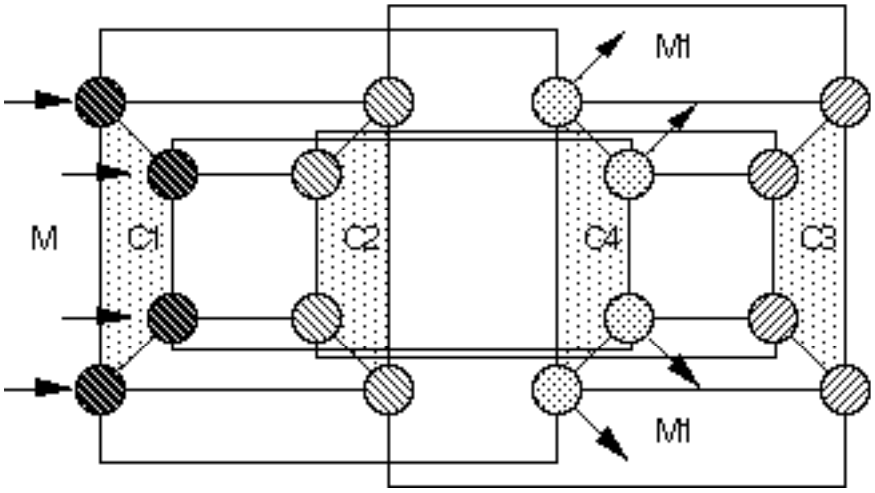


Figure 15. The four contexts C1, C2, C3, C4 in the hypercube network.

```
#define UC (*,*,*,*), d 4, m 2;
main()
   {
    context C1,C2,C3,C4 of hype(2);
    ima of int M[128,128], P[128,128] in C3;
    ima of int S[128,128] in C2 | C3, M1[128,128] in C3 | C4;
    C1 = (0,*,*,0); C2 = (0,*,*,1);
    C3 = (1,*,*,0); C4 = (1,*,*,1);
    open(C1);
    while(EOF==false)
        {
        load(M);
        deconv(M);
        open(C2);
          segmentation(M,S);
          open(C3);
            M1 = 0;
            where(matching(S,P))
              {
              M1=M;
              open(C4);
                unload(M1);
              close(C4);
              }
          close(C3);
        close(C2);
        }
    close(C1);
    }
```
Figure 16. A PICL algorithm for object retrieval.

6. Conclusions

The report describes how concurrent processes are handled in the pictorial language, PICL, designed for image analysis and implemented on the multi-processor machine HERMIA. The concept of context has been introduced, as a hardware-data relation, which is especially useful in the case of distributed spatial data.

The relation between the *open*/*close* statements, formal languages and dependency graphs has been analyzed. The results allow an easy and fast implementation inside PICL. Contexts are handled inside a PICL program as normal dynamic variables, this make easier the handling of concurrent processes and the control of the flow of the program. Moreover, contexts have geometrical and topological properties that allow us to put the computation into the proper data allocation.

This approach seems to be promising because it allows handling concurrence at several levels of system hierarchies. This feature could be very useful in the case of DCS. Taking into account that they will became more and more popular as soon as faster communication buses are commercially available (1Gbit/sec).

Moreover, DCS are characterized by an huge amount of states and parameters, that are distributed through several elementary sub-system units ($\geq$1000). Functional dependencies exist between state and parameters. Automatic control assessment and motion detection in risky environments are examples of distributed systems. In these cases, visual data are usually collected from multi-sensors and their elaboration is carried out on local processing units, which are logically interconnected to share and interchange knowledge (models, data and algorithms). The implementation of contexts in visual distributed environment will provide a synthetic view of the distributed system behavior, and guides to understand local and/or global computation phases.

In fact, the design and the implementation of algorithms on multi-processor machines depends on the distribution of data and processes among the processing units, therefore the dynamic control is relevant to optimize and to tune the execution of processes. Moreover, dynamic visual tools allows us to realize such user/machine interaction in a natural and efficient way.

Further formal work is required to study the composition of the operations on a DG graphs and their transitive closure. The investigation and the design of visual contexts inside iconic distributed languages examples will be also a challenging development of the work that has been done until now; the first prototype of the M-VIF machine will be used to experiment the usefulness of contexts in distributed systems for applications in vision and perception.

Acknowledgment. We thank Jerome Feldman for reading this report, and for making suggestions towards its improvement.

References

[1] M.J.B.Duff, D.M.Watson, T.J.Fountain, G.K.Shaw, "A Cellular Logic Array for Image Processing", *Pattern Recognition*, Vol.5, pp.229-247, 1973.

[2] A.P.Reeves, "Parallel PASCAL:An Extended PASCAL for Parallel Computers", *Journal of Parallel and Distributed Computing*, Vol.1, pp.64-80, 1984.

[3] A.P.Reeves, "Languages for Parallel Processing", in *Proc.of 2nd Int.Workshop on Data Analysis in Astronomy*, V.Di Gesu', L.Scarsi, P.Crane, J.H.Friedman, S.Levialdi, edits., Plenum Press, 1986.

[4] D.E.Reynolds, G.P.Otto, "Software Tools for CLIP4", *Report No.82/1*, 1982.

[5] W.Hillis, G.L.Steel, "The Connection Machine Lisp Manual", *Thinking Machine Corporation*, Cambridge, MA, 1985.

[6] E.Y.Shapiro, "Concurrent Prolog: A progress report", *Computer,* Vol.9, No.8, pp.44-58, 1986.

[7] K.Preston, Jr., "Progress in Image Processing Languages", in *Computing Structures for Image Processing*, edit.M.J.B.Duff, Academic Press, London, pp.195-211, 1983.

[8] "Proceedings IEEE Workshop on Visual Languages", *IEEE Computer Society Press,* October, 1989.

[9] Man-Chi Pong, "A Graphical Language for Concurrent Programming", *IEEE Workshop on Visual Languages*, Dallas, 1986.

[10] S.Murer, J.A.Feldman, C-C.Lim, M-M.Seidel, “pSather: Layered Extentions to an Object-Oriented Language for Efficient Parallel Computation”, *ICSI Tech.Rep. TR-93-028*, Berkeley, December 1993.

[11] K.Asanovic, J.Beck, T.Callahan, J.Feldman, B.Irissou, B.Kingsbury, P.Kohn, J.Lazzaro, N.Morgan, D.Stoutamire, and J.Wawrzynek, “CNS-1 Architecture Specification, *ICSI Tech.Rep. TR-93-021*, Berkeley, April 1993.

[12] V.Di Gesu', "A High Level Language for Pyramidal Architectures", *Proc.NATO ASI Workshop on Pyramidal Systems for Computer Vision*, V.Cantoni, S.Levialdi (eds.), Maratea, 1986.

[13] V.Di Gesu', "PCL: A Pyramid Language", *Tech.Rep.DMAT-2-88*, Univ.of Palermo, 1988.

[14] T.A.Pavlidis, S.L.Tanimoto, "A Hierarchical Data Structure for Picture Processing", *Comp.Graphycs & Image Processing*, Vol.4, 1975.

[15] V.Cantoni, L.Carrioli, O.Catalano, L.Cinque, V.Di Gesu', M.Ferretti, G.Gerardi, S.Levialdi, R.Lombardi, A.Machi', R.Stefanelli, "The PAPIA Image Analysis System", *SPIE International Symposium on Image Processing*, Cannes, 1985.

[16] G.Gerardi, "The PAPIA Controller: Hardware Implementation", *Proc.NATO ASI Workshop on Pyramidal Systems for Computer Vision*, V.Cantoni, S.Levialdi (eds.), Maratea, 1986.

[17] F.Chiavetta,V.Di Gesù,G.Gerardi,D.Tegolo, "The Pictorial Language and the Software Environment of the HERMIA Machine", *1st Israely/Italian workshop on Computer Vision*, Capri, 1991.

[18] G.Gerardi, F.Chiavetta, V.Di Gesù, D.Tegolo, "HERMIA: An Heterogeneous and Reconfigurable Machine for Image Analysis", *MVA IAPR Workshop on Machine Vision Applications*, Tokio, pp.397-400, Nov.28-30, 1990.

[19] Vito Di Gesu', Gaetano Gerardi, and Domenico Tegolo, "M-VIF: a Machine -Vision based on Information Fusion", *proceedings of CAMP'93, (ed. M.A.Bayoumi, L.S.Davis, K.P.Valavanis), IEEE Comp.Soc.Press*, pp.428-435, New Orleans 1993.

[20] V.Di Gesù, G.Gerardi, F.Isgrò, D.Tegolo, "Visual distributed environment for distributed systems", in *Proc.CAMP'95, IEEE Comp.Soc.Press*, in press.

[21] K.E.Batcher, "Design of a massively parallel processor", *IEEE Transactions on Computers*, vol.C-29, pp.836-840, 1980.

[22] M.J.B.Duff: CLIP4: A Large Scale Integrated Circuit Array Parallel Processor. Proc. 3rd International Joint Conf. Pattern Recog. Coronado, 1976.

[23] R.Miller and Q.F.Stout, "Geometric algorithms for digitized pictures on a mesh-connected computer", *IEEE Trans.PAMI-7*, pp.216-228, 1985.

[24] V.Di Gesù, "An Overview on Pyramid Machines for Image Processing", *Information Sciences Journal*, Vol.47, pp.17-34,1989.

[25] V. Cantoni, V. Di Gesù, M. Ferretti, S. Levialdi, R. Negrini, R.Stefanelli, "The Papia System", in *Journal of VLSI Signal Processing*, Vol.2, pp.195-217, 1991.

[26] W.D.Hillis, "The Connection Machine", Cambridge MA: the MIT Press, 1992.

[27] E.W.Dijkstra, "Cooperating Sequential Processes", in *Programming Languages*, F.Genuys ed., Academic Press, NY, pp.43-112, 1968.

[28] H.Brinch, "The programming Language Concurrent Pascal", *IEEE Trans.on Software Engineering*, SE-1, No.2, pp.199-206, 1975.

[29] N.Wirth, "Modula: A Language for modular Multiprogramming", *Software Practice and Experience*, Vol.7, pp.33-35, 1977.

[30] E.W.Dijkstra, "Solution and Problem in Concurrent Programming", *Comm.of ACM*, Vol.8, pp.569-570, 1965.

[31] A.C.Shaw, "The Logical Design of Operating Systems", Prentice Hall, Englewood Cliffs, NJ, 1974.

[32] M.Conway, "A Multiprocessor System Design", in *Proc.AFIPS Fall Joint Comput. Conf.*, Sparten Books, NY, pp.139-146, 1963.

[33] J.Loren Gischer, "Partial Orders and Axiomatic Theory of Shuffle", *Report No.STAN-CS-84-1033*, Stanford Univ., Dept.of Computer Science, 1984.

[34] V.Pratt, "Modelling Concurrency with Partial Orders", *Report No.STAN-CS-86-1113*, Stanford Univ., Dept.of Computer Science,1986.

[35] W.Reising, "Petri Nets: An Introduction", Springer-Verlag, 1985.