3. Description of Algorithms for Parallel Implementation

The description of an algorithm consists of a set of mathematical equations. Let's analyse for example the matrix-vector multiplication

$$\underline{c} = \underline{\underline{A}}\ \underline{b} \quad \textit{or}$$
$$c_i = \sum_{j=1}^{N} a_{i,j} b_j \tag{3.1}$$

(see figure 2.5). As it is mentioned earlier, for parallel realizations concurent - parallel and pipeline - subtasks are needed. It is quite straightforward to notice that each c_i can be computed *parallel*, no data transfer is needed. It is also straightforward to rewrite equation 3.1 to a *recursive* form:

$$c_i^{(j+1)} = c_i^{(j)} + a_i^{(j)} * b^{(j)} \tag{3.2}$$

where: $c_i^{(1)} = 0$
$a_i^{(j)} = a_{ij}$
$b^{(j)} = b_j$

so it is well suited for pipeline realization. In these equations the so called *single assigment formulation* is used. It means that *every variable is assigned one value only during the execution* of the whole algorithm. Of course it is not a memory saving formula, but later, this single assigment formulation will help to make parallelism and defining Dependency Graphs.

Another important fact to point out, that *time index* (the upper index) using simple assigment formulas can easily transferred to *spatial index* which plays also an important role in creation of Dependency Graphs.

3.1 Snapshots

Snapshot is a description of array activities at a particular time instant. They are the most natural tools to design, check or verify an array algorithm. On the other hand it is quite redundant to describe the whole algorithm. As a matter of fact the figures in the previous sections are very similar to snapshots.

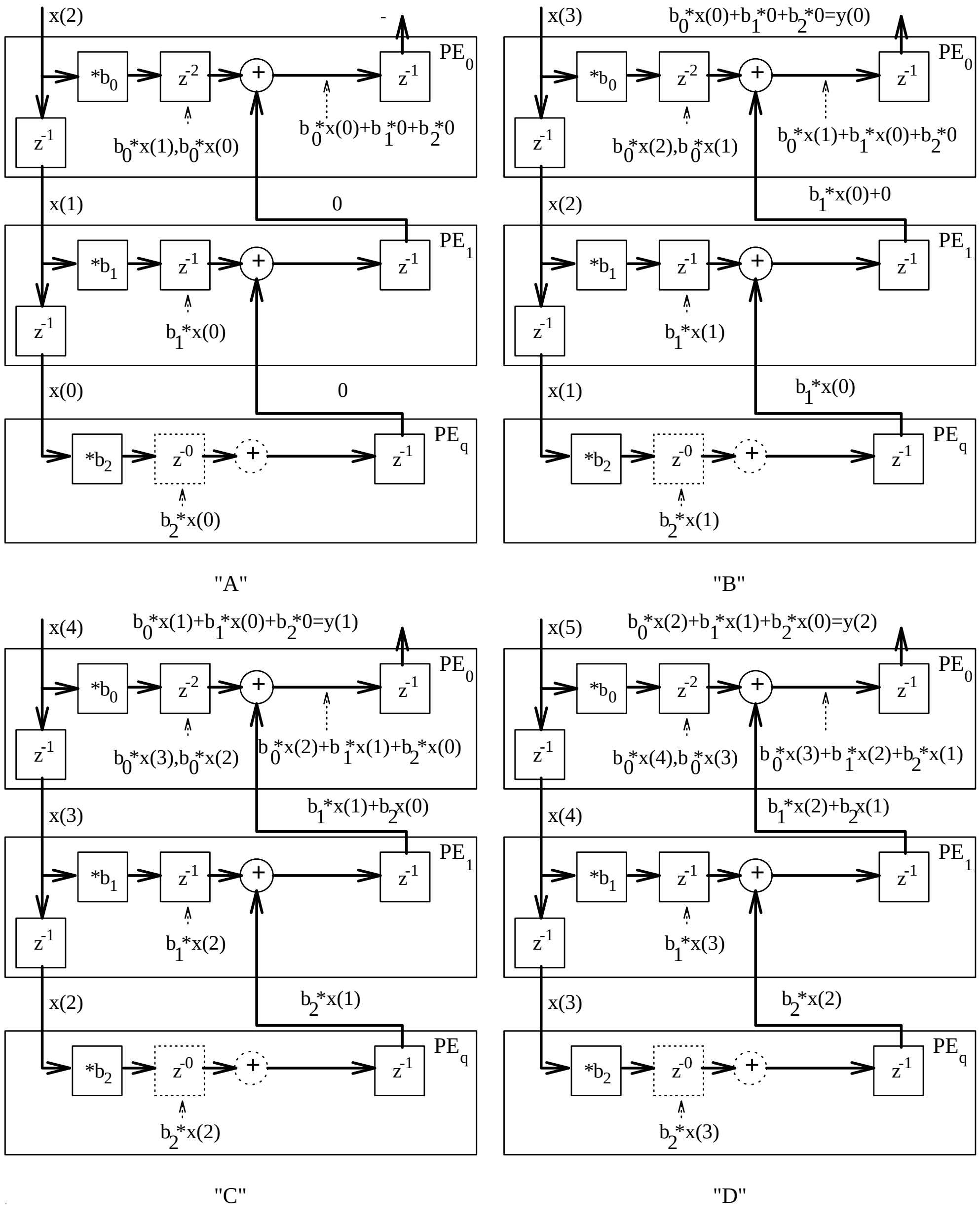

Figure 3.1: Snapshots of FIR filter realization: steps 2,3,4 and 5 are on A,B,C and D parts respectively

Recalling figures connected to FIR filter realizations (figures 2.3, 2.4 and 2.8, the latter one is the most interesting to verify the necessity of extra z^{-i} elements of systolic realization. The figure 3.1 shows the snapshots for $q = 2$.

3.2 Dependency Graphs (DG)

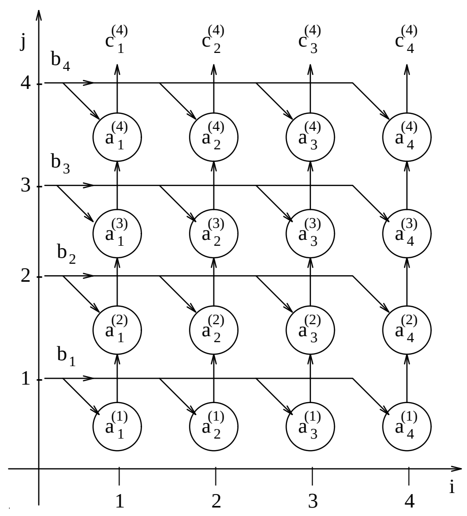

Figure 3.2: DG of matrix-vector multiplication using global communication

Dependency Graph (DG) is a graph that *shows the dependence of computational elements* (input data, constants, temporary results and output data) *appearing in an algorithm. DG can be considered as a graphical representation of a single assigment algorithm.*

For example recall matrix vector multiplication using equation 3.2. Figure 3.2 shows the connected DG. On this figure, and also on the latter ones the index spaces are explicitly shown using axes i and j.

Equation 3.2 can be transferred to another form:

$$c_i^{(j+1)} = c_i^{(j)} + a_i^{(j)} * b_i^{(j)} \tag{3.3}$$

$$b_{i+1}^{(j)} = b_i^{(j)}$$

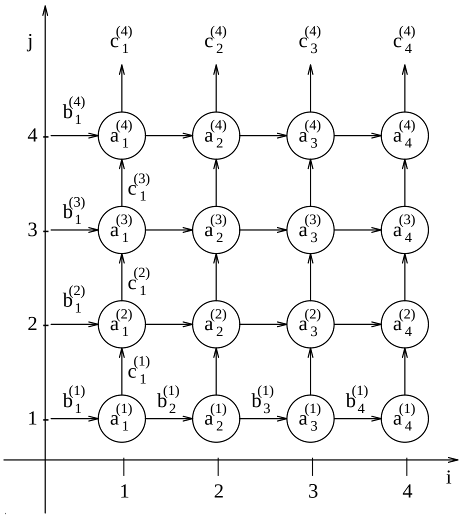

Figure 3.3: DG of matrix-vector multiplication with local communication

where:
$c_i^{(1)} = 0$
$a_i^{(j)} = a_{ij}$
$b_i^{(j)} = b_j$

Only the indexes of $\underline{b}$ vector elements are changed to a recursive one. This change has only a symbolic meaning from the mathematical point of view, but looking at this equation as a *single assigment algorithm*, in every step a new variable appears and it has to be passed to the next PE. This method gives the key to translate global interconnections to local ones. The DG created from equation 3.3 is shown on figure 3.3.

Notice that *DG doesn't tell anything about timing.* Of course timing is critical from the computational point of view, but DG offers to show only the dependencies between data elements. As it will be discussed later, the systolic realization of figure 3.3 needs much more cycles due to the propagation of $b_i^{(j)}$ elements than the other one on figure 3.2 with global communication. This is the tradeoff between localization and speed.

Figures 3.2 and 3.3 show *not complete DG-s*, it means that they don't tell anything on computations in nodes. In most of the cases it is quite laborious to make the whole graphical representation of computation for each node. As it is already discussed, our goal to create a homogenous processor network, so every processor will execute more or less the same

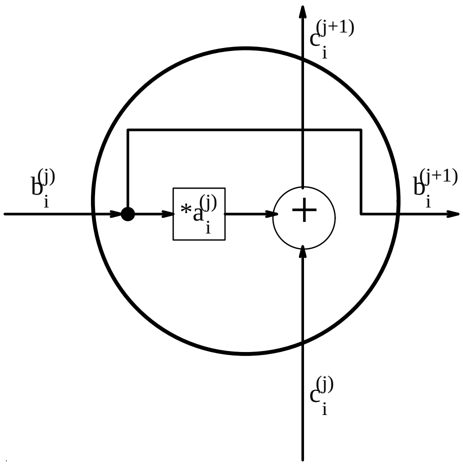

Figure 3.4: A detailed node of DG shown on fig. fdgmvecl

computation. In most of the cases the more simple, not complete DG will be used, and the computations of nodes are defined by recursive equations. A complete node computation has been drawn on figure 3.4 to demonstrate a node element of our example shown on figure 3.3.

It worth to show the DG for the FIR filtering discussed in sections 2.1 and 2.5.1 on figures 2.3, 2.4 and 2.8, but first let's notice, that FIR filtering is a simple convolution of the coefficients with the input sequence, and the coefficients are equal to the impulse response of the FIR filter. The difference between them is only philosophical, in FIR filtering the input sequence is usually much longer than the number of coefficients, while in convolution there is no any restriction. So in the next example we show the DG of the convolution, which is similar to the DG of the FIR filtering.

Using equation of convolution

$$y(i) = \sum_{j=i-M+1}^{N-1} u(j)w(i-j) \quad \text{if} \quad i = M..(N+M-2) \tag{3.4}$$

the recursive counterpart is straightforward:

$$y_i^{j+1} = y_i^j + u(j) * w(i-j) \tag{3.5}$$

where the index ranges are:
if $i = 0..(M-1)$ than $j = 0..i$ and

if $i = M..(N+M-2)$ than $j = (i-N+1)..(N-1)$
$u(i) \quad i = 0..(N-1)$
$w(i) \quad i = 0..(M-1)$
and the initial conditions are:
$y_i^0 = 0$ for $i = 0..(M-1)$ and
$y_i^{i-M+1} = 0$ for $i = M..(N+M-2)$,

DG of this recursive convolution with $N = 3$ and $M = 4$ is on figure 3.5. This DG has hard global interconnections. Rewriting the original equation in similar way as equation 3.2 of the matrix-vector multiplication was transformed to equation 3.3, the recursive and single assigment form of convolution is:

$$y_i^{(j+1)} = y_i^{(j)} + u_i^{(j)} * w_i^{(j)} \tag{3.6}$$

$$u_{i+1}^{(j)} = u_i^{(j)}$$

$$w_{i+1}^{(j+1)} = w_i^{(j)}$$

where the index ranges are:
if $i = 0..(M-1)$ than $j = 0..i$ and
if $i = M..(N+M-2)$ than $j = (i-M+1)..(N-1)$
and the initial conditions are:
$y_i^{(0)} = 0$ for $i = 0..(M-1)$ and
$y_i^{(i-M+1)} = 0$ for $i = M..(N+M-2)$,
$u_k^{(k)} = u(k)$ for $k = 0..(N-1)$,
$w_k^{(0)} = w(k)$ for $k = 0..(M-1)$.

The resulted DG is shown on figure 3.6, and it seems local and homogenous, suited very well to array processor realization.

To give some further formal statements on Dependency Graphs let's declare some terminology:

- A general *graph* $G = [N, A]$ consists of two sets N and A, where elements of set N called *nodes* and elements of set A called *arcs or edges.*

- Each arc, $a \ni A$ connects a pair of nodes $n_i, n_j \ni N$ pointing from the *initial endpoint* node n_i to the *terminal endpoint* node n_j. The direction indicated by an arrow.

- An arc whose endpoints are the same node is called *loop.*

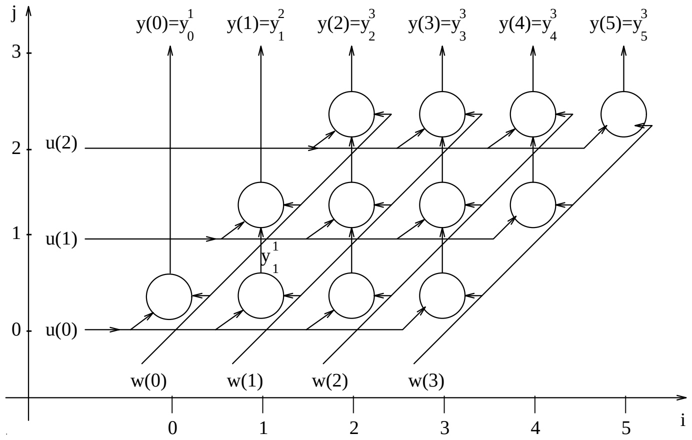

Figure 3.5: DG of convolution with global connections

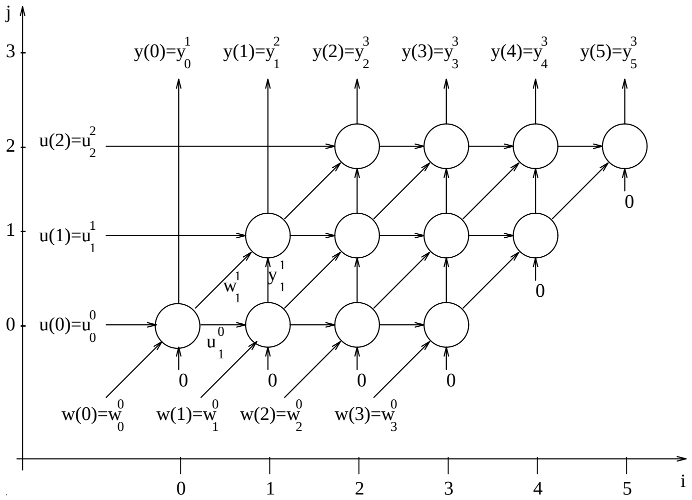

Figure 3.6: DG of convolution with local connections

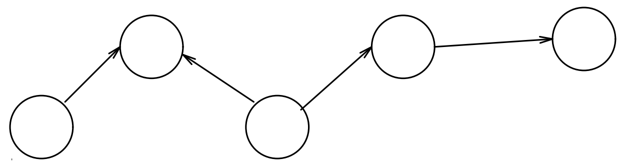

Figure 3.7: A graph chain

- A *chain* is a sequence of arcs with common endpoits sequentally after each other, without regard to the direction (see figure 3.7). The chain has two free endpoints which have only one connection, while all the other nodes has two connections.
- A *path* is a chain, where all of the arcs are directed to the same direction (there isn't any node in the path with both connected arcs being terminal or initial endpoints). The path has *initial and terminal node* called *starting and endpoints of the path*, defined by the nodes have only arc connected with initial and terminal endpoint respectively.
- A *cycle* can be formed from a path, setting the same node as the terminal and the initial node.
- An *undirected cycle* can be formed from a chain which is not path, connecting the free nodes.
- A graph is *connected* if there exists a chain between every pair of nodes.
- A graph is *strongly connected* if there exists a path from each node to all other nodes.

Some general statements on DG-s and algorithms:

- An algorithm is *computable* if and only if its complete DG contains no loops or cycles.
- A dependence of two nodes called *local dependency* if the length of the arc is independent of the problem size.
- An algorithm is *locally recursive* if the corresponding DG has only local dependences and most of the nodes executes the same kind of computation. Exeptions can be for example the I/O interface nodes.
- A data called *transmittent data* if the node transmits it without any modification. All other data called *nontransmittent data*.

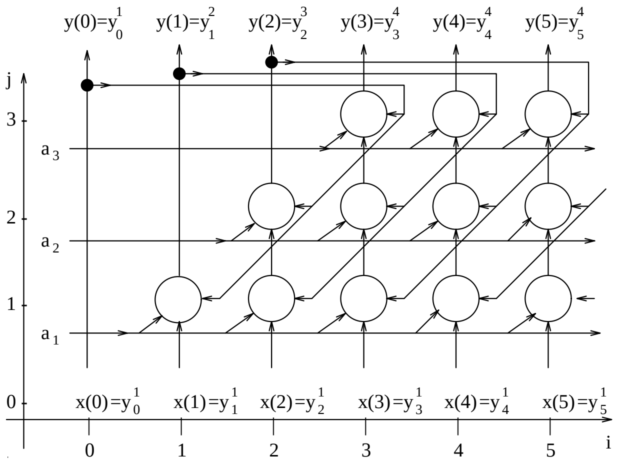

Figure 3.8: DG of AR filtering problem with global connections, and normal order

- A data called *broadcast data* if the same input data is used by more nodes.

The way to create a locally recursive algorithm is not straightforward in all of the cases. There is no systematic way, but some heuristic strategy can be given:

- Try to convert broadcast data to transmittent data.
- Try to change the direction of index changes, so change the order of the subsequent computations.
- Try to localize the communication declaring new *intermediate variables* which helps to store and propagate output data of a node to be used by other nodes.

For the first point an example was the convolution on figures 3.5 and 3.6. For the latter two an illustrative example is the AR (IIR) filtering problem.
Recall the basic equation a recursive counterpart is:

$$y_i^{(j+1)} = y_i^{(j)} + a^{(j)} y(i-j) \tag{3.7}$$

where
$y_i^{(1)} = x(i)$
$a^{(j)} = a_j \; i = 0..N$
$j = 1..p$

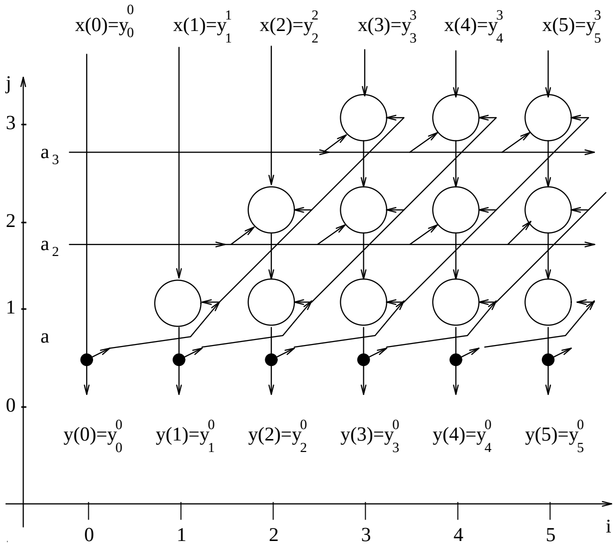

Figure 3.9: DG of AR filtering problem with global connections, and changed order

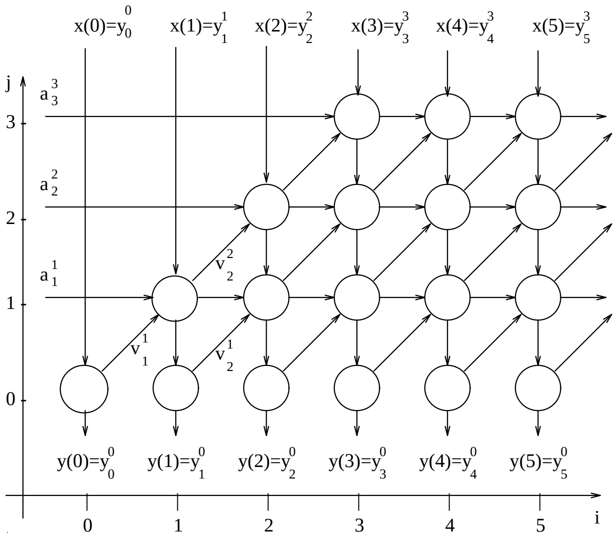

Figure 3.10: DG of AR filtering problem with local connections

The result is on figure 3.8. There are two sets of broadcast data the coefficients a_j and the result sequence $y(i-j)$. Changing the order of the summation results a new algorithm:

$$y_i^{(j-1)} = y_i^{(j)} + a^{(j)} y(i-j) \tag{3.8}$$

where
$y_i^{(p)} = x(i)$
$a^{(j)} = a_j \ i = 0..N$
$j = p..0$

As it is quite clear from the resulted figure 3.9, from the result sequence $y(i)$ a new intermediate variable can be formed, and this variable can be transfered to transmittent data. Of course $a^{(j)}$ can also be converted to transmittent data:

$$\begin{gathered} y_i^{(j-1)} = y_i^{(j)} + a_i^{(j)} v_i^{(j)}, \\ v_{i+1}^{(j+1)} = v_i^{(j)} \text{and} \\ a_{i+1}^{(j)} = a_i^{(j)}, \end{gathered} \tag{3.9}$$

where
$y_i^{(i)} = x(i)$ for $i = 0..p$ and $j = i..0$
$y_i^{(p)} = x(i)$ for $i = (p+1)..N$ and $j = p..0$
$a_j^{(j)} = a(j)$
$v_{i+1}^{(1)} = y_i^{(0)}$

The resulted locally recursive DG is on figure 3.10.

3.3 Signal Flow Graphs (SFG)

The signal flow graph (SFG) offers a poweful abstraction and graphical representation for problems in signal processing computations. SFG consists of processing nodes, communicating edges and delays. In general, a node is oftern denoted be a circle representing an arithmetic or logic function performed with zero delay, such as multiply or add. An edge denotes either a dependence relation or a delay. The SFG representation derives its power from the fact that the computations are assumed delay-free, i.e. they take no time at all. Consequently, the burden of tracing the detailed time-space activities associated with pipelining are eliminated. Moreover any delay in the system has to be explicitly introduced in the form of so called daley branches. These delay branches allow history-sensitive systems to be described in a clear and unambiguous way.

A complete SFG description should include both functional and structural description parts. The functional description defines the behavior within a node, the structures description specifies the interconnection (edges and delays) between the nodes.

As compared with the DG, the SFG has the following properties:

1. The SFG can be viewed as a simplified graph. This means that the SFG is a more concise representation than the DG.

2. The SFG is more specific, i.e. it is closer to hardware level design. Therefore, the SFG also dictates the type of arrays that will be obtained.

3. While there are no loops in any DG, the SFG can have loops, as long as is at least one delay on each loop.

3.4 Mapping from DG to SFG

In mapping frm DG to SFg thre are two basic considerations.

1. To which processor should an operation assigned (allocation).

2. What is the ordering of the operations when they are assigned to a processors. (time scheduling)

Processor assigment (allocation). As a simple example, a mapping method can be applied, in which nodes of the DG along one straight line are assigned to a common PE. Since the DG of a locally recursive algorithm is very regular, the projection maps the DG onto a lower dimensional lattice of points, known as the processor space. The result of this projection is represented by an SFG.

Scheduling. The projection should be accompanied by a scheduling scheme, which specifies the sequence of the operations in all the PEs. A schedule function represents a mapping from the N-dimensional index space of the DG onto a one dimensional time space. A linear schedule is based on a set of parallel and uniformly spaced hyperplanes in the DG. These hyperplanes are called equitemporal hyperplanes, all the nodes on the same hyperplane must be processed at the same time. There is a partial ordering among the computations, inherent in the algorithm, as specified by the DG. More specifically, if there is a directed path from node x to node y, then the computation represented by node x is completed. The feasibility of a schedule is determined by the partial ordering and the processor assignment scheme.

The description of array processing activities, in terms of the SFG representations, are oftern easy to comprehend. The abstraction provided by the SFG is very powerful to use, and yet the transformation to an SIMD, systolic array or wavefront array can often be accomplished automatically. Some detailed examples can be found in the literature.