Virtual Parallelism Support in Reconfigurable Processor Arrays

Massimo Maresca and Hungwen Li

TR-91-041

July, 1991

VIRTUAL PARALLELISM SUPPORT IN RECONFIGURABLE PROCESSOR ARRAYS

Massimo Maresca
International Computer Science Institute
Berkeley, California

Hungwen Li
IBM Research Division
Almaden Research Center
San Jose, California

ABSTRACT

Reconfigurable Processor Arrays (RPAs) are a special class of mesh connected computers in which each node is equipped with a switching system able to internally interconnect its NEWS ports and to establish paths between non neighbor nodes. The best known proposals, in the area of RPAs, are the Mesh with Reconfigurable Bus [1], the Processor Arrays with Reconfigurable Bus Systems [2], the Gated Connection Network [3] and the Polymorphic Processor Array [4]. In this paper we show that only one of these architectures, namely the Polymorphic Processor Array, supports virtual parallelism. The support of virtual parallelism is important because it allows the complexity measurements of the parallel algorithms be scaled to real implementations, where the size of the processor array can be smaller than the problem size. We demonstrate that *1)* the RPAs that allow to establish an arbitrary shape two-dimensional bus do not support virtual parallelism and *2)* the Polymorphic Processor Array, with its connection power limited to one-dimensional buses, supports virtual parallelism.

1. Introduction

Reconfigurable Processor Arrays (RPAs) are a special class of mesh connected computers in which each node is equipped with a switching system able to internally interconnect its NEWS ports. The principle upon which RPAs are based is that of continuously changing the switch setting in each node, in such a way to dynamically establish buses interconnecting different clusters of neighbor processors. The buses are assumed ideal and therefore communication in each cluster is considered instantaneous.

The best known proposals, in the area of reconfigurable processor arrays, are the Mesh with Reconfigurable Bus (MRB) [1], the Processor Arrays with Reconfigurable Bus Systems (PARBS) [2], the Gated Connection Network (GCN) [3] and the Polymorphic Processor Array (PPA) [4]. These architectures differ from each other in

the structure of the switching system located in each node and, as a consequence, feature different reconfiguration capabilities and achieve different performance. However, considering the small cost added to mesh [5], all the RPAs exhibit impressive performances. For example, graph component labeling is claimed to have O(1) complexity in PARBS [2] and image component labeling is claimed to have O(log$n$) complexity in MRB [6], assuming $\sqrt{n} \times \sqrt{n}$ as a processor array size. In comparison the computational complexity of the same tasks on a plain mesh without reconfiguration capability is O($\sqrt{n}$) [7].

We are interested in understanding whether RPAs can be directly used as a basis for the design of massively parallel computers. In order to be directly used as a basis for the design of massively parallel computers, we require, among other properties, that RPAs support virtual parallelism. Virtual parallelism is referred to as a situation in which the number of parallel activities is larger than the number of available processors and some of the activities are executed in sequence with linear performance degradation. The support of virtual parallelism is important because it allows the generalization of the complexity measurements of the parallel algorithms to real situations. The concept of virtual parallelism, widely exploited in today's massively parallel computers, frees the algorithm designer from the need to know the size of the parallel computer he is working on; it is the compiler that automatically transforms the programs written for the case in which as many processors as needed are available into programs for systems of smaller size.

The definition of virtual parallelism used throughout this paper is as follows: let the complexity of an algorithm for a RPA of size equal to the problem size n be denoted as O(p), the RPA supports virtual parallelism if the same algorithm has O($\frac{p}{k}$) complexity when the RPA size m is a submultiple of the problem size ($k = \frac{n}{m} \geq 1$ integer). This definition requires not only that there exists an algorithm performing the same original function on a smaller RPA, but also that the performance of the original algorithm scales down linearly with the size of the RPA.

In this paper we show that only one of the aforementioned proposed RPAs, namely PPA, supports virtual parallelism. The reason of such a different behavior is that, unlike MRB, GCN and PARBS, PPA treats the rows and the columns of the processor array independently and purposely avoids the coupling between the row buses and the column buses. We term such a coupling as a two-dimensional bus and show that it does not support virtual parallelism. We draw the conclusion that only the PPA architectural model can be directly and effectively used as a basis for the design of massively parallel computers.

The paper is organized as follows: in the next section we show that the RPAs that support the establishment of two-dimensional buses do not support virtual parallelism, in section 3 we show that PPA supports virtual parallelism, and in section 4 we discuss the

importance of this result and provide some concluding remarks.

2. RPAs supporting two-dimensional buses

Def. 1. A *one-dimensional* bus is a bus that interconnects only nodes located in the same row (or in the same column) of a RPA. A *row-bus* (*column-bus*) is a bus interconnecting only nodes located in the same row (column).

Def. 2. A *two-dimensional bus* is a bus that interconnects at least two nodes located in different rows and columns of a RPA.

A two-dimensional bus is implemented by means of a combination of row and column buses, as shown in Fig. 1. As an example of a very long and articulate two-dimensional bus, in Fig. 2 we show a two-dimensional bus that interconnects the connected nodes of an undirected graph G mapped on a processor array, according to the technique proposed in [2]. The nodes of the graph are associated with the diagonal nodes of the processor array and the upper right half of the graph adjacency matrix A (which is symmetrical) is mapped on the upper right half of the processor array. The desired graph is established on the processor array by commanding the switches corresponding of '0' elements of A to connect their E and W ports horizontally and their S and N ports vertically and the switches located in correspondence of '1' elements of A to connect all the NEWS ports together.

The argument of supporting one-dimensional versus two-dimensional buses in RPAs has been treated as a hardware implementation issue and its impact on programming was not understood. At the first glance the two-dimensional bus model is more powerful than its one-dimensional counterpart. However, as to be shown in Theorem 1, the two-dimensional bus model does not support virtual parallelism while the one-dimensional bus model does, as to be shown in Theorem 2. Consequently, programs written for the two-dimensional bus model may depend on the system size, which is a significant drawback in the software development for massively parallel computers.

On the contrary, the complexity of the algorithms claimed for the one-dimensional bus model is independent of the system size and can be scaled to a real implementation of arbitrary size. This paper addresses the issue that the bus implementation may impact programming. We show such an impact by Theorem 1 in this section and Theorem 2 in section 4. We state and prove Theorem 1 next.

Theorem 1: A RPA allowing the establishment of two-dimensional buses does not support virtual parallelism.

Proof: We prove the theorem by showing that a two-dimensional bus on a RPA of size $\sqrt{n} \times \sqrt{n}$ cannot be emulated in $O(\frac{n}{k})$ iterations on a RPA of size $\sqrt{\frac{n}{k}} \times \sqrt{\frac{n}{k}}$ where k is an integer. The proof is accomplished by a counter-example. Fig. 3 illustrates a two-

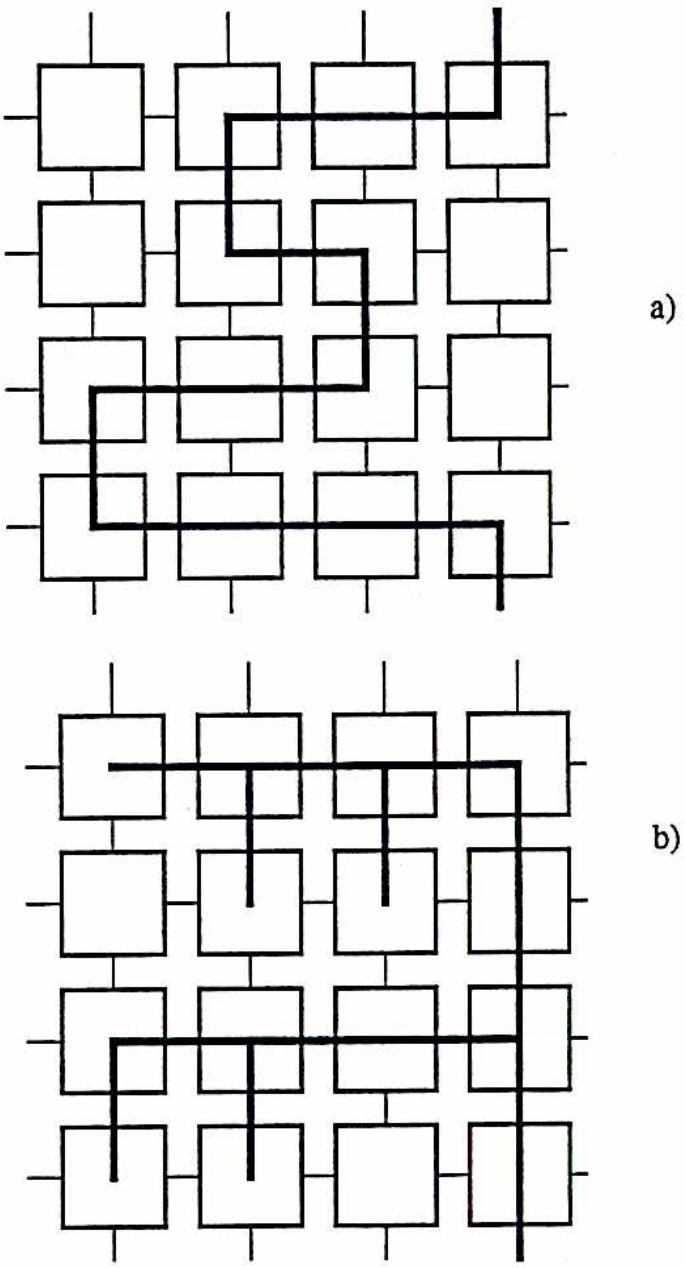
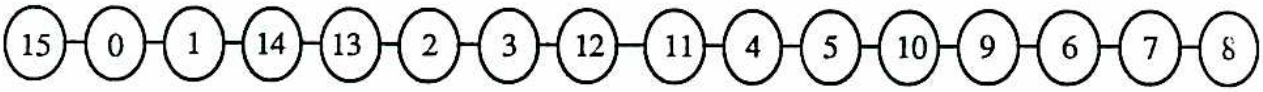


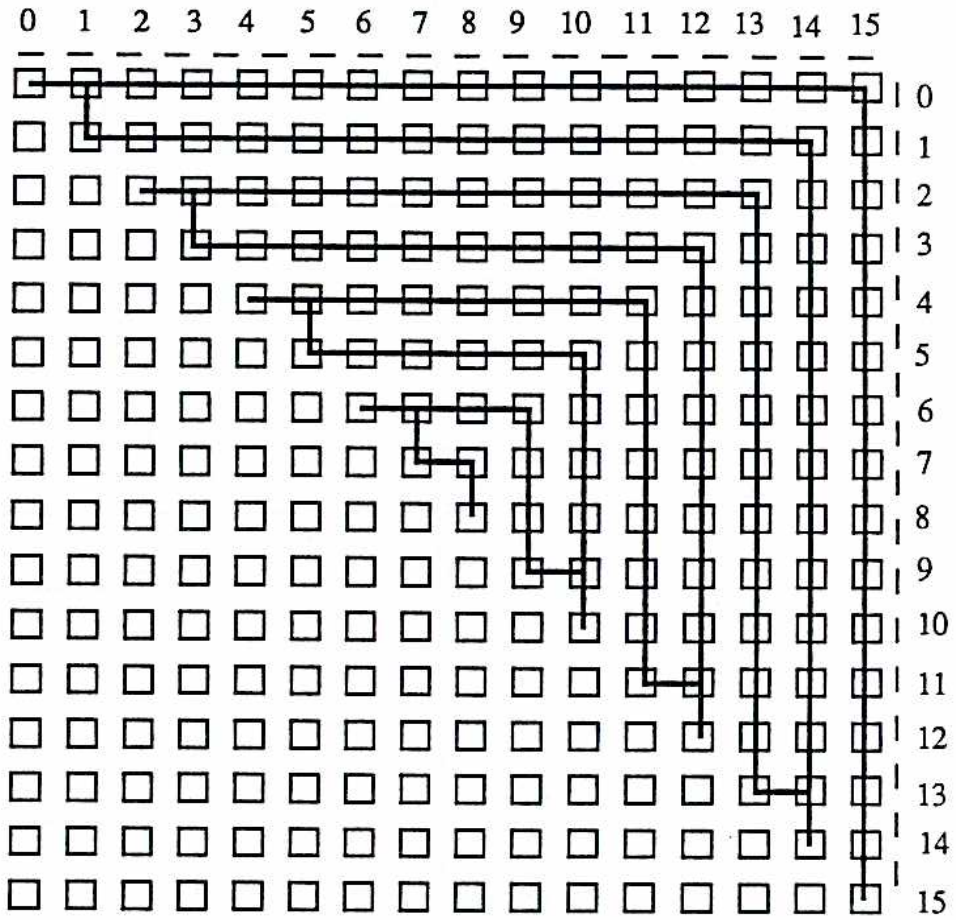
Fig. 1 - a) zig-zag shaped two-dimensional bus and b) tree shaped two-dimensional bus.



a) Graph

1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
0 0 1 1 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 1 1 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 1 1 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

b) Adjacency Matrix



c) Two-dimensional bus established

Fig. 2 - Mapping a graph G onto a PARBS and creating a bus interconnecting all the connected nodes. a) graph, b) adjacency matrix and c) corresponding bus on the PARBS.

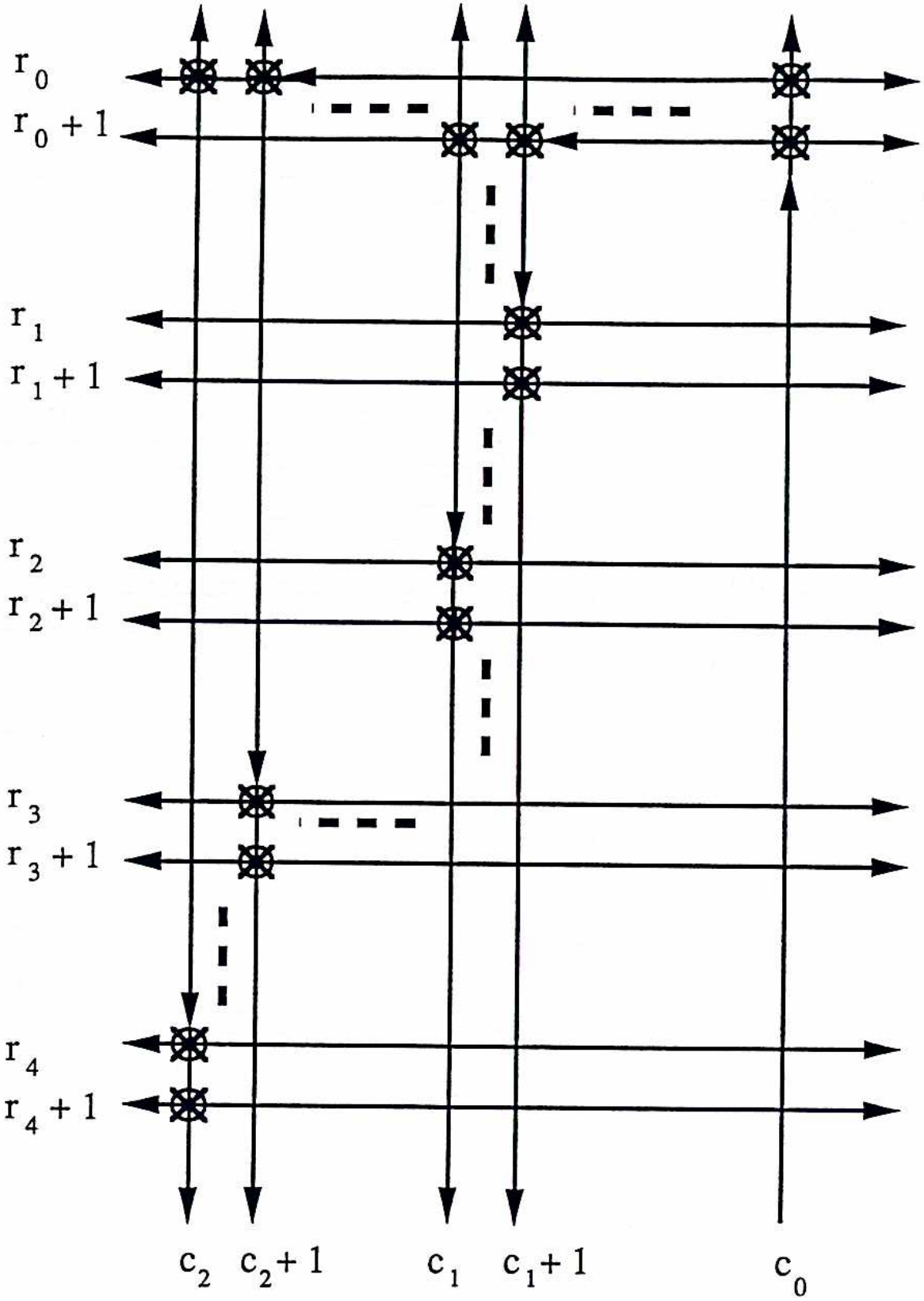


Fig. 3 - Example of a two-dimensional bus not supporting virtual parallelism

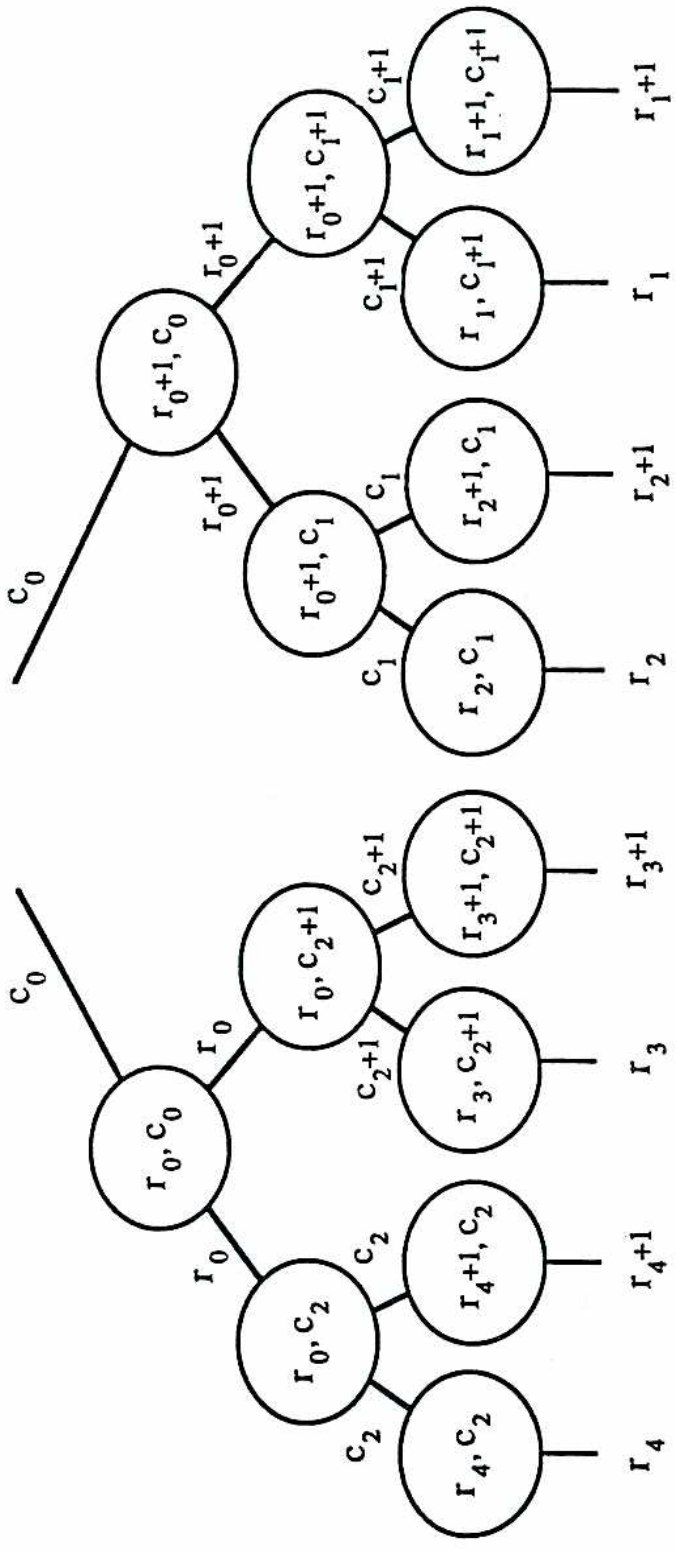


Fig. 4 - Logical way of looking at the bus of Fig. 3.

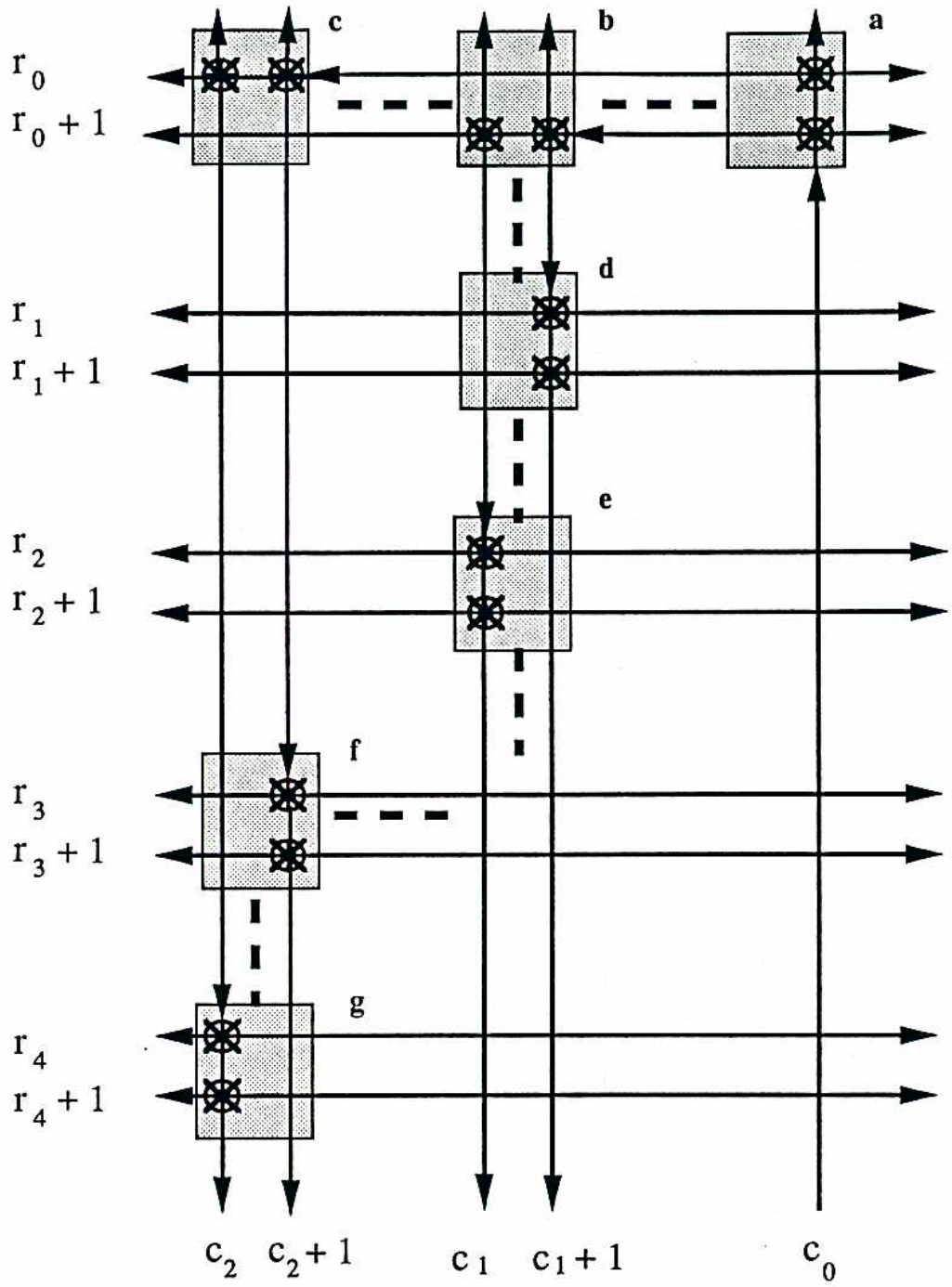
dimensional bus interconnecting the nodes in rows r_0 , r_0+1, r_1, r_1+1, r_2, r_2+1, r_3, r_3+1, r_4 and r_4+1 and in columns c_0, c_0+1, c_1, c_1+1, c_2 , c_2+1. The switches drawn in Fig. 3 implement the interconnection among the four NEWS ports, while the other switches, not drawn in Fig. 3 for simplicity, only implement horizontal (E<->W) and vertical (S<->N) interconnections. Such a bus can be established in MRB, GCN and PARBS.

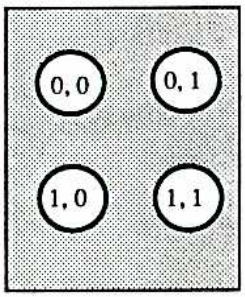
Let us consider the case of a datum arriving from the S direction in column c_0. Fig. 4 shows such a flow of the datum on the bus and the forking associated with the flow when the switches interconnecting the row-bus and the column-bus are encountered. The labels of the nodes in Fig. 4 indicate the coordinates of such switches while the labels of the edges show the rows and the columns added to the two-dimensional bus after crossing a switch. There are 8 different rows, namely r_1, r_1+1, r_2, r_2+1, r_3, r_3+1, r_4 and r_4+1, which require three switches to be traversed.

To establish the same bus on a physical RPA of size $\frac{\sqrt{n}}{2} \times \frac{\sqrt{n}}{2}$, four virtual nodes are mapped onto each physical node as shown in Fig. 5. A pair of virtual links, such as r_0 and r_0+1 r_1 and r_1+1, c_1 and c_1+1 etc., share a physical link accordingly. A 'virtual' configuration, i.e. the switch setting, is used to emulate one of the four virtual switches represented by the very node. The (0, 0), (0, 1), (1, 0) and (1, 1) states of a node in Fig. 5 represent four virtual configurations, one for each virtual switch. To support virtual parallelism, the physical RPA needs to emulate the two-dimensional bus in $O(k)$ iterations (in this case we expect 4 iterations) by properly combining the switch setting of the nodes at each iteration. On the contrary we show that the paths from c_0 to the 8 rows that require three switches to be traversed, namely, $(c_0 \rightarrow r_1)$, $(c_0 \rightarrow r_1+1)$, $(c_0 \rightarrow r_2)$, $(c_0 \rightarrow r_2+1)$, $(c_0 \rightarrow r_3)$, $(c_0 \rightarrow r_3+1)$, $(c_0 \rightarrow r_4)$ and $(c_0 \rightarrow r_4+1)$, must be emulated one at a time, through a loop of 8 iterations.

The reason why such paths must be emulated one at a time is that any pair of such paths share at least one physical switch and require it to take a different configuration. For example $(c_0 \rightarrow r_2)$ and $(c_0 \rightarrow r_2+1)$ share switches *a*, *b* and *e* and require switch *e* to take the (0, 0) virtual configuration to emulate $(c_0 \rightarrow r_2)$, and the (1, 0) virtual configuration to emulate $(c_0 \rightarrow r_2+1)$. TABLE I shows the virtual configurations to be used in controlling each physical switch in order to establish the conflicting paths.

In the example of Fig. 5 only 8 leaves of a signal-branching tree are in conflict with each other as explained in TABLE I. A generalization can be seen in Fig. 6, in which 2^L leaves are possibly in conflict due to the possibility that L levels of switches need to be traversed. When two or more (i.e. k) switches are virtualized in one physical node as illustrated in the box in Fig. 6, the RPA can only emulate one leave of the tree at a time since each physical switch can only assume one virtual configuration at a time. Since the





Virtual configurations inside each physical node

Fig. 5 - The two-dimensional bus of Fig. 3 mapped on a smaller size RPA.

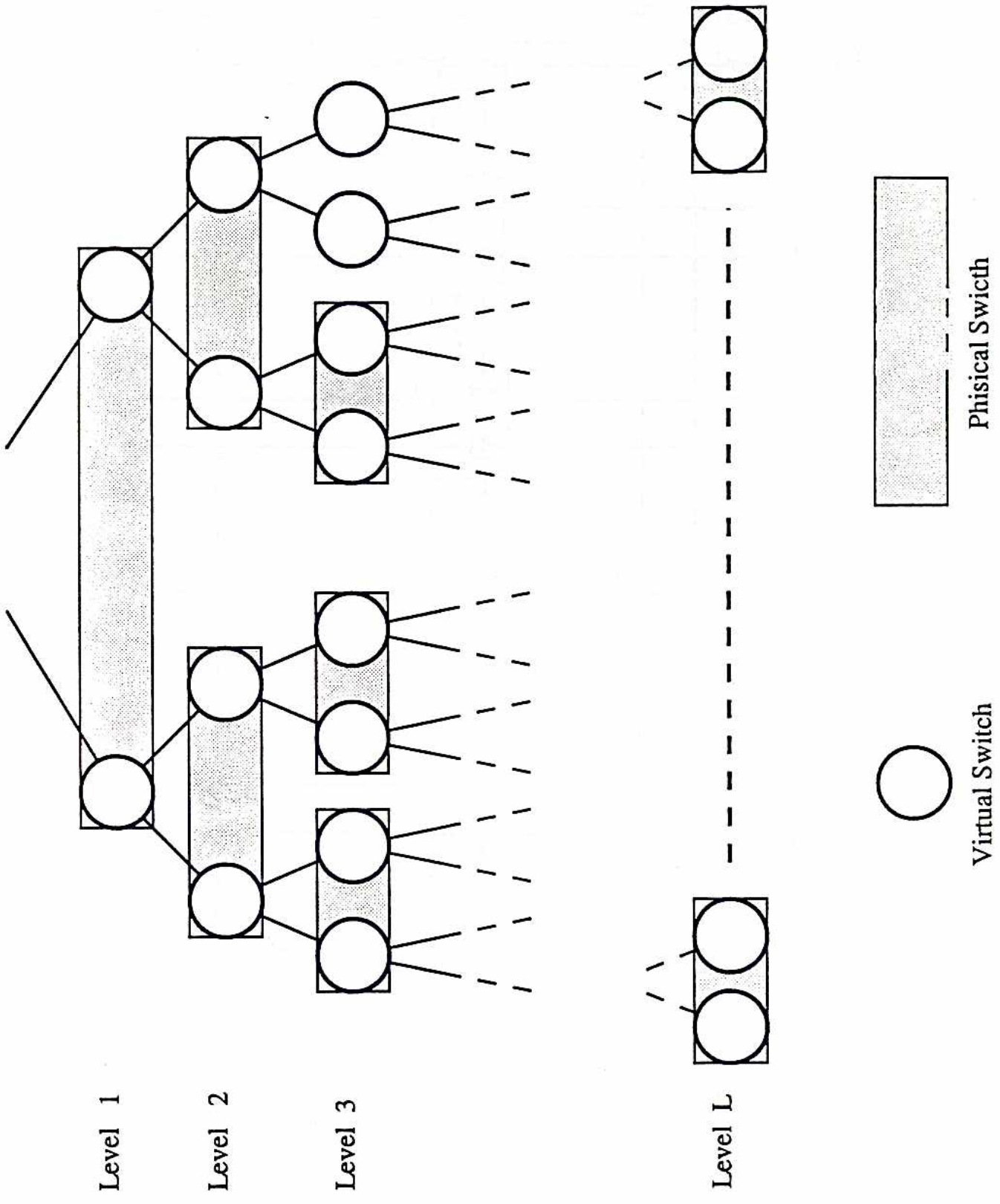


Fig. 6 - Generalization of the tree in Fig. 4

PATH	PHYSICAL SWITCH	VIRTUAL CONFIG.	PHYSICAL SWITCH	VIRTUAL CONFIG.	PHYSICAL SWITCH	VIRTUAL CONFIG.
$c_0 \rightarrow r_1$	a	1, 1	b	1, 1	d	0, 1
$c_0 \rightarrow r_1+1$	a	1, 1	b	1, 1	d	1, 1
$c_0 \rightarrow r_2$	a	1, 1	b	1, 0	e	0, 0
$c_0 \rightarrow r_2+1$	a	1, 1	b	1, 0	e	1, 0
$c_0 \rightarrow r_3$	a	0, 1	c	0, 1	f	0, 1
$c_0 \rightarrow r_3+1$	a	0, 1	c	0, 1	f	1, 1
$c_0 \rightarrow r_4$	a	0, 1	c	0, 1	g	0, 0
$c_0 \rightarrow r_4+1$	a	0, 1	c	0, 1	g	1, 0

TABLE I : Sequence of virtual configurations to be used as physical configurations to establish the two-dimensional bus in Fig. 5.

size of the tree L may grow with the size of the processor array, the complexity of two-dimensional bus emulation is $O(L)$ and is dependent of the size of the RPA. It therefore follows that a RPA allowing the establishment of two-dimensional buses does not support virtual parallelism.
Q.E.D.

3. RPAs supporting one-dimensional buses: the Polymorphic Processor Array

Unlike the other RPAs currently proposed, PPA does not allow arbitrary shaped two-dimensional buses. PPA is a processor array in which each node is equipped with a switch like the one shown in Fig. 7, that can be oriented along any of the four mesh directions (we call the NEWS orientation of the switch the switch *orientation*) and that can be opened and shorted under program control (we call the OPEN/SHORT configuration of the switch the switch *configuration*). While the switch *configuration* is programmed locally in each node, the switch *orientation* is programmed by the central program controller and is the same for all the nodes. Due to the fact that all the switches share the same *orientation*, only one-dimensional communication is possible in PPA.

The operation of each PPA node depends both on the *orientation* and on the *configuration* of the corresponding switch. Let us consider, as an example, the case in which the switch *orientation* is E, such as shown in Fig. 8 (the extension to the other three directions is straightforward). Each node, regardless of its switch *configuration*, always receives a message from its W port; on the contrary, depending on its switch

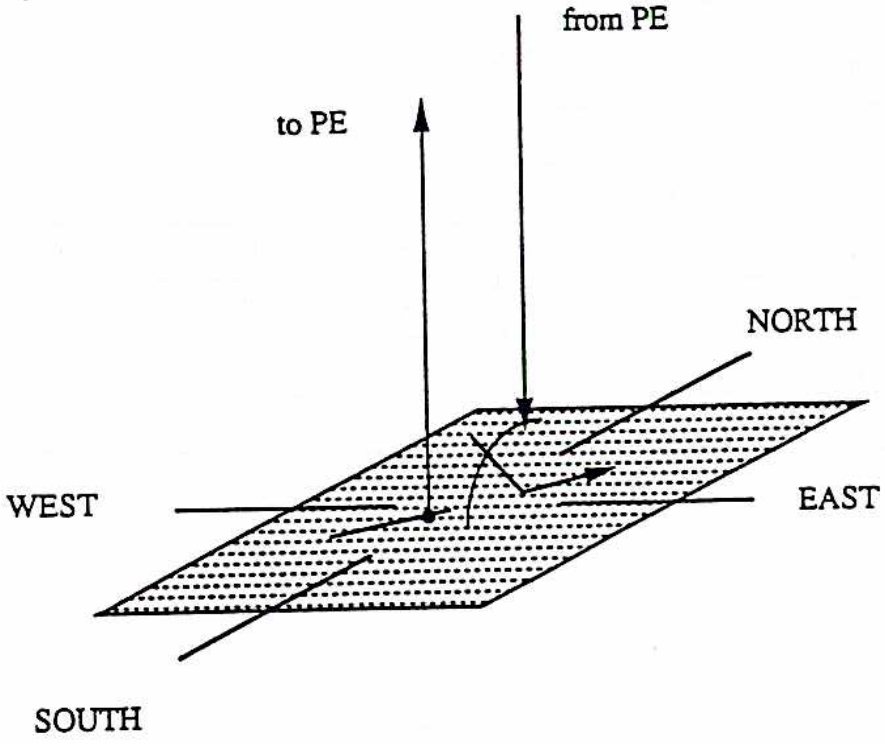


Fig. 7 - PPA switch

configuration, each node either connects its W port to its E port (e.g. nodes a and b in SHORT *configuration*), or connects itself to the E port (e.g. nodes c and d in OPEN *configuration*).

An *orientation* and a *configuration*, denoted as a pair (O, C), determine the PPA operation. We notice that a pair (O, C) partitions the PPA nodes into a set of clusters of nodes with each cluster corresponding to a one-dimensional bus (Fig. 9). Each cluster includes one node in the OPEN *configuration* and all the other nodes in the SHORT *configuration*, along *orientation* O, up to the next node in the OPEN *configuration* (not included). A cluster is identified by the coordinates of the node in the OPEN *configuration*. More precisely, $\forall\ ij = 0, 1, \cdots, \sqrt{n}-1$, node $N_{ij} \in$ cluster C_{st} where st are:

- case 1: $O = N$
  - $s = (i-k)\ \%\ \sqrt{n}$, $\quad k = \min_{l=0,\ldots,\sqrt{n}-1}(l \mid C_{(i+l)\ \%\ \sqrt{n},\, j} = OPEN)$
  - $t = j$

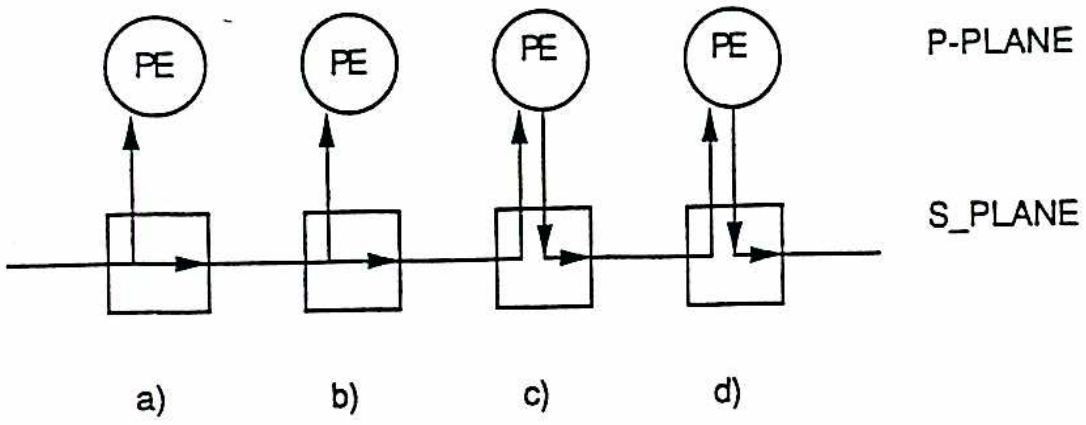


Fig. 8 - Data propagation in PPA

- case 2: $O = E$
  - $s = i$
  - $t = (j - k) \% \sqrt{n}, \quad k = \min_{l=0,..,\sqrt{n}-1}(l \mid C_{i,(j-l)\%\sqrt{n}} = OPEN)$
- case 3: $O = W$
  - $s = i$
  - $t = (j + k) \% \sqrt{n}, \quad k = \min_{l=0,..,\sqrt{n}-1}(l \mid C_{i,(j+l)\%\sqrt{n}} = OPEN)$
- case 4: $O = S$
  - $s = (i + k) \% \sqrt{n}, \quad k = \min_{l=0,..,\sqrt{n}-1}(l \mid C_{(i-l)\%\sqrt{n},j} = OPEN)$
  - $t = j$

Communication in PPA can be expressed through the *broadcast* primitive defined as follows:

Broadcast Communication Primitive: Given

- a pair (O, C) partitioning a PPA into a set of clusters of nodes,
- a two-dimensional array *SRC* mapped on a PPA one element per node and
- a two-dimensional array *DST* mapped on a PPA one element per node,

the *broadcast* communication primitive performs the assignment of the element of *SRC* corresponding to the node in the OPEN *configuration*, i.e. SRC_{st}, to all the elements of

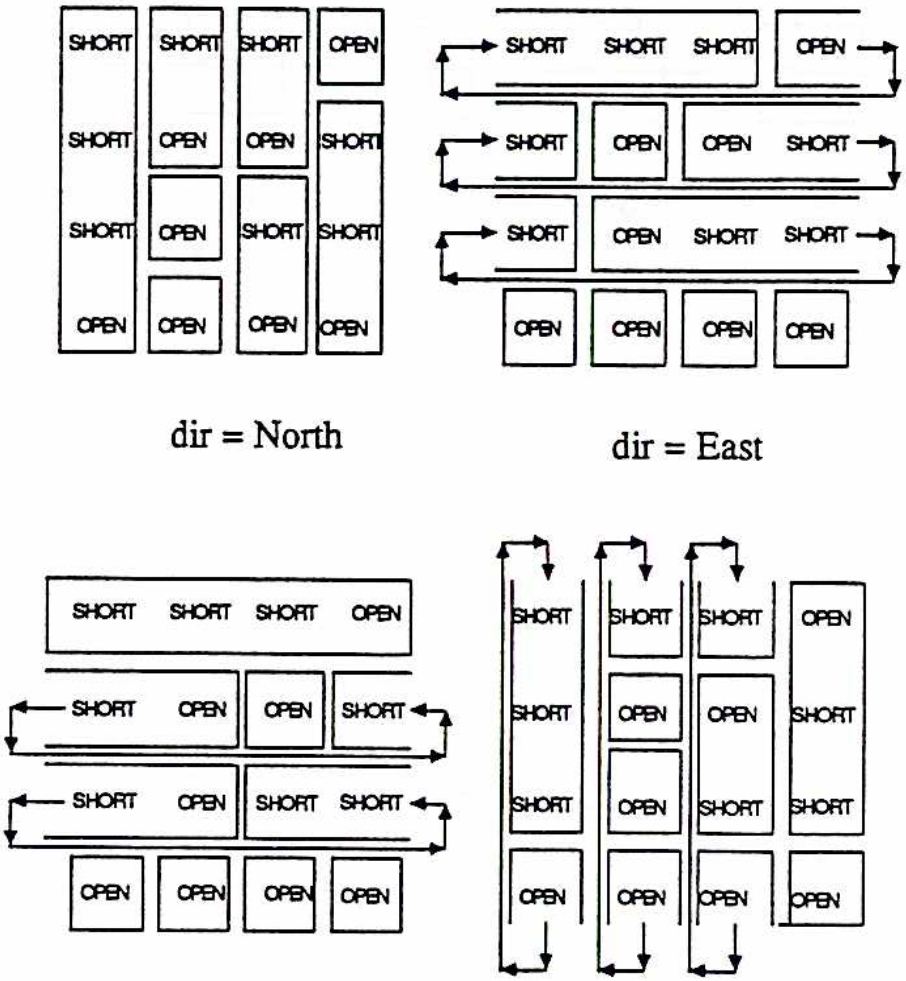


Fig. 9 - Partitioning a PPA in a set of clusters of nodes.

DST corresponding to the nodes in the same cluster, i.e. $DST_{ij} \leftarrow SRC_{st}\ \forall N_{ij} \in C_{st}$. The *broadcast* just described occurs simultaneously in all clusters.

Broadcasting is carried out in constant time in PPA by taking advantage of the one-dimensional buses associated to the clusters. In the rest of the paper we will refer to the *broadcast* communication primitive by the following notation:

DST = broadcast (SRC, O, C);

Broadcast is the fundamental communication primitive for PPA. Any PPA algorithm involving any type of communication can be implemented upon *broadcast*. For example, a frequently used communication primitive such as *shift* can be built upon *broadcast* as explained in the following example:

Example:

DST = shift (SRC, E);

can be written as:

```
where (even column number)
   DST = broadcast(SRC, E, odd column nodes OPEN);
elsewhere
   DST = broadcast(SRC, E, even column nodes OPEN);
```

Similarly, all other operations requiring communication can also be decomposed down to the *broadcast* primitive. We will show in the next section that the *broadcast* fundamental communication primitive can be virtualized in PPA.

4. Virtual parallelism in PPA

In this section we demonstrate that PPA supports virtual parallelism by proving that any algorithm having $O(p)$ complexity on a virtual PPA of size $\sqrt{n} \times \sqrt{n}$, has $O(kp)$ complexity on a physical PPA of size $\sqrt{m} \times \sqrt{m}$, where $n = km$ (k integer). Considering that a PPA algorithm can only include operations that do not require communication and operations that do require communication, and considering that these latter operations are all expressed in terms of the *broadcast* communication primitive, we only have to prove that

1) any operation requiring no communication and having $O(p)$ complexity on a $\sqrt{n} \times \sqrt{n}$ PPA, has $O(kp)$ complexity on a $\sqrt{m} \times \sqrt{m}$ PPA, where $k = \frac{n}{m}$.

2) the *broadcast* communication primitive has $O(k)$ complexity for a $\sqrt{n} \times \sqrt{n}$ array on a $\sqrt{m} \times \sqrt{m}$ PPA, where $k = \frac{n}{m}$.

A $\sqrt{n} \times \sqrt{n}$ array X is mapped on a $\sqrt{m} \times \sqrt{m}$ PPA as follows: node $N_{st}, st = 0, 1, \cdots, \sqrt{m} - 1$ contains all the elements x_{ij} of the original array X, such that

$$i = s\sqrt{k}, s\sqrt{k} + 1, s\sqrt{k} + 2, \cdots, (s+1)\sqrt{k} - 1$$
$$j = t\sqrt{k}, t\sqrt{k} + 1, t\sqrt{k} + 2, \cdots, (t+1)\sqrt{k} - 1$$

We prove the virtual parallelism theorem by establishing two lemmas first.

Lemma 1. Any operation requiring no communication and having $O(p)$ complexity on a $\sqrt{n} \times \sqrt{n}$ PPA, has $O(kp)$ complexity on a $\sqrt{m} \times \sqrt{m}$ PPA, where $k = \frac{n}{m}$.

Proof: Trivial: the operation is repeatedly executed over the arrays of size $\sqrt{k} \times \sqrt{k}$ stored in each node, for all the nodes in parallel.

Lemma 2. The *broadcast* communication primitive has $O(k)$ complexity for a $\sqrt{n} \times \sqrt{n}$ array on a $\sqrt{m} \times \sqrt{m}$ PPA, where $k = \frac{n}{m}$.

Proof: For brevity, we only demonstrate the lemma for the case in which *orientation* = *S*; the generalization to the other three cases is straightforward. Let *SRC* and *DST* be two arrays of size $\sqrt{n} \times \sqrt{n}$ mapped onto a a PPA of size $\sqrt{m} \times \sqrt{m}$ where $m = \frac{n}{k}$ and k is an integer. Let the *orientation* $O = S$, and let the *configuration* C be an array of size $\sqrt{n} \times \sqrt{n}$ with arbitrary OPEN/SHORT elements. In the following, for a generic array X we will use the notation X_{st} to refer to the elements of X allocated in the PPA nodes at the same address.

The broadcasting of the $\sqrt{k}$ columns of *SRC* allocated in each node (each column having $\sqrt{k}$ elements) is done sequentially and independently and requires $\sqrt{k}$ iterations; the specific steps executed to broadcast each column $v = 0, \cdots, \sqrt{k}-1$ are shown in the pseudo-code in Fig. 10 and described next:

Step 1. Column v is scanned from row 0 to row $\sqrt{k}-1$ to emulate the broadcasting of the elements of *SRC* corresponding to OPEN virtual nodes to all the SHORT virtual nodes belonging to the same cluster and residing in the same PPA physical node. Whenever a C_{uv} = OPEN is found in a physical node, the corresponding SRC_{uv} value is copied into the elements of *DST* associated with the subsequent elements of *DST*, i.e. $DST_{u+1,v}, DST_{u+2,v}, \cdots$, till the next OPEN element of C (not included).

Step 2. Column v is scanned from row 0 to row $\sqrt{k}-1$ to simultaneously compute the following two values in each PPA node:

- $P\_SRC = SRC_{uv}$ *where* $u = \max(0, 1, \cdots, \sqrt{k}-1)$ and $C_{uv} = OPEN$
- $P\_C = SHORT$ if $C_{uv} = SHORT \ \forall u = 0, 1, \cdots, \sqrt{k}-1$.

Step 3. The *broadcast* communication primitive is executed using *P_C* as a configuration, *P_SRC* as a source and another array called *P_DST*, of size $\sqrt{m} \times \sqrt{m}$, as a destination.

Step 4. Virtual column v is scanned from row 0 to row $\sqrt{k}-1$ to complete the broadcasting. The broadcasting is completed by simultaneously copying, in each PPA node, *P_DST* into all the elements of *DST* up to the first virtual node $N_{uv} = OPEN$, i.e. DST_{0v}, DST_{1v}, ..., DST_{kv} if $C_{0v} = C_{1v} = \ldots = C_{kv}$ = SHORT and $C_{(k+1)v}$ = OPEN.

The complexity of the algorithm can be evaluated by observing that *step 1* takes $O(\sqrt{k})$ iterations, *step 2* takes $O(\sqrt{k})$ iterations, *step 3* takes O(1) iterations and *step 4*

takes $O(\sqrt{k})$ iterations, thus leading to an overall complexity of $O(\sqrt{k})$. Since the algorithm must be executed $\sqrt{k}$ times to cover all the virtual columns, the overall complexity is $O(k)$.
Q.E.D.

We now state and demonstrate the theorem demonstrating that PPA supports virtual parallelism:

```
/* Loop over the virtual columns mapped on the same physical nodes */

for (v = 0; v < √k ; v++) {

   /* Step 1 */
   for (u = 0 ; u < √k ; u++) {
      if (C_uv == OPEN) tmp = SRC_uv ;
      DST_uv = tmp;
   }

   /* Step 2 */
   P_C = SHORT;
   for (u = 0 ; u < √k ; u++) {
      if (C_uv == OPEN) {
         P_C = OPEN;
         P_SRC  = SRC_uv ;
      }
   }

   /* Step 3 */
   P_DST = broadcast (P_SRC, S, P_C);

   /* Step 4 */
   tmp = TRUE;
   for (u = 0 ; u < √k ; u++) {
      if (tmp)
         if ((tmp = (C_uv == OPEN)) DST_uv = P_DST;
   }
}
```

Fig. 10 - Implementation of virtual broadcast on PPA: case $O = S$.

Theorem 2 (Virtual Parallelism). Any algorithm having $O(p)$ complexity on a virtual PPA of size $\sqrt{n} \times \sqrt{n}$, has $O(kp)$ complexity on a PPA of size $\sqrt{m} \times \sqrt{m}$, with $n = km$ and k an integer.

Proof: Considering that the operations performed by a PPA can be either operations requiring no communication, which support virtual parallelism as shown by Lemma 1, or operations requiring communication based upon the *broadcast* communication primitive, which supports virtual parallelism as shown in Lemma 2, theorem 2 therefore follows. Q.E.D.

5. Discussion and concluding remarks

In this paper we have shown that among the reconfigurable processor arrays currently proposed, namely the Mesh with Reconfigurable Bus (MRB) [1], the Processor Arrays with Reconfigurable Bus Systems (PARBS) [2], the Gated Connection Network (GCN) [3] and the Polymorphic Processor Array (PPA) [4], only PPA supports virtual parallelism; the support of virtual parallelism makes PPA a more attractive architectural model, bacause it can be directly used for the design of massively parallel computers.

The importance of virtual parallelism in a massively parallel computer is to faithfully keep the complexity claim made in the model independent of the system size, so that the size-invariance of a program can be achieved automatically by the compiler. Furthermore, programmers can write programs demanding as many processors as needed without being concerned with the implications of the system size.

The reason why PPA supports virtual parallelism and the other reconfigurable processor arrays do not is that PPA allows only one-dimensional buses but not arbitrary shaped buses in two-dimension. Allowing two-dimensional buses makes reconfigurable processor arrays so powerful that in some cases it leads to O(1) complexity algorithms, such as in the graph problems in [2]. However, these very attractive complexity results hide a fundamental weakness of the underlying models, that is the lack of support of virtual parallelism. As a consequence, these results are only valid for the case in which the problem size is the same as the processor size.

An important consequence of supporting virtual parallelism is that PPA algorithms can follow a scalable programming model. A PPA programming language has been proposed and various programming tools, like simulators and debuggers [8], have been developed to ease the programming task. On the contrary, when two-dimensional buses are allowed, the switch setting must be explicitly controlled by the programmer, which makes programming more complex and inflexible.

References

1. R. Miller, V. K. Prasanna-Kumar, D. Reisis and Q. F. Stout, Meshes with Reconfigurable Buses, *Proc. MIT Conference on Advanced Research in VLSI pp. 163-178, (1988)*.

2. B. F. Wang and G. C. Chen, Constant Time Algorithms for the Transitive Closure and Some Related Graph Problems on Processor Arrays with Reconfigurable Bus System, *IEEE Trans. on Parallel and Distributed Systems* Vol. 1, Num. 4, pp. 500-507, (Oct. 1990).

3. D. B. Shu and J. G. Nash, The Gated Interconnection Network for Dynamic Programming, *Concurrent Computations*, S. K. Tewsburg, B. W. Dickinson and S. C. Schwartz (editors), Plenum Publishing Company, pp. 645-658, .

4. H. Li and M. Maresca, Polymorphic-Torus Network, *IEEE Trans. on Computers* Vol. 38, Num. 9, pp. 1345-1351, (Sept. 1989).

5. M. Maresca and H. Li, Connection Autonomy in SIMD Computers : a VLSI implementation, *Journal of Parallel and Distributed Computing* Vol. 7, pp. 302-320, (1989).

6. R. Miller, V. K. Prasanna-Kumar, D. I. Reisis and Q. F. Stout, Data Movement Operations and Applications on Reconfigurable VLSI Arrays, *International Conference on Parallel Processing* pp. 205-208, (1988).

7. D. Nassimi and S. Sahni, Finding Connected Components and Connected Ones on a Mesh-Connected Parallel Computer, *SIAM Journ. of Computing* Vol. 9, pp. 744-757, (1980).

8. G. Boreanaz, G. Dirosa, M. Migliardi, E. Orione, P. Baglietto, M. Maresca and A. L. Frisiani, Simulation of Parallel Algorithms for SIMD Massively Parallel Computers, *Proc. ISCS (Italian Society for Computer Simulation) Conference*, Rome, 1990.