Transitive Closure and Graph Component Labeling on realistic Processor Arrays based on Reconfigurable Mesh Network

Massimo Maresca
International Computer Science Institute
Berkeley, California

Pierpaolo Baglietto
DIST - University of Genoa
Genoa - Italy

Abstract
This paper presents an $O(\log^2 n)$ complexity parallel algorithm for graph component labeling and transitive closure on reconfigurable processor arrays. Although a lower complexity algorithms for the same tasks on reconfigurable processor arrays have been proposed (see for example [11]), the reconfigurable processor arrays supporting such algorithms have communication capabilities which are, in our opinion, neither scalable nor suitable for VLSI implementation. On the contrary the algorithm presented in this paper is designed for a realistic class of reconfigurable processor arrays, called Polymorphic Processor Arrays, which is both scalable and suitable for VLSI implementation.

1. Introduction

In a recent paper on the IEEE Transaction on Parallel and Distributed Systems Wang and Chen present "Constant Time Algorithms for the Transitive Closure and for Some Related Graph Problems on Processor Arrays with Reconfigurable Bus Systems" (PARBS) [11]. According to such a paper, the PARBS model performs better than all the known parallel models of computation, including CRCW-PRAM, the most powerful one, which, as well known, only supports $O(\log n)$ algorithms for the same tasks [2]. The authors mention the Polymorphic-Torus system (PT) [3,4], an experimental massively parallel architecture based on reconfigurable mesh interconnection network, to claim that the results of their study are not only valid in theory, but also in the context of real computer architectures.

We believe that the generalization of the PT system to the PARBS model is to be reconsidered. Based on the experience gained in the PT project we have defined a computation model, or abstract architecture, called Polymorphic Processor Array (PPA), specifically matching the PT implementation; PPA is less powerful than PARBS and only supports poly-logarithmic complexity algorithms for transitive closure and related graph problems, but, unlike PARBS, it models a realistic class of reconfigurable processor arrays, the implementation of which was proven possible and cost-effective.

2. PPA Logical Architecture

Polymorphic Processor Array (PPA) is a computation model for SIMD parallel computers; a PPA, the logical architecture of which is shown in Fig. 1, consists of a stack of three planes, respectively called processor plane, or P-plane, memory plane, or M-plane, and switch plane, or S-plane. The P-plane is a 2-dimension array of processors, or processing elements, called $PE_{ij}$, the M-plane is a 2-dimension array of memories called $M_{ij}$, and the S-plane is a 2-dimension array of switch-boxes called $S_{ij}$ (we assume $ij = 0, 1, ..., \sqrt{n}-1$). Each set

![Fig. 1 - Polymorphic Processor Array](image-url)
The PPA switch-box logical structure is shown in Fig. 2: a switch-box consists of an ideal switch which can be oriented along any of the four mesh directions (we shall refer to the NEWS orientation of the switch as the switch-box orientation) and which can be opened and shorted under program control (we shall refer to the OPEN/SHORT configuration of the switch as the switch-box configuration). The switch-box orientation is programmed by the central program controller through the switch instruction and is the same for all the nodes. On the contrary, the switch-box configuration is programmed by the corresponding processing element, depending on local data, and can be different in each node.

The basic assumptions upon which the PPA model is based are that 1) the mesh links introduce no delay and 2) the switch-boxes introduce no delay. From the implementation standpoint, considering that SIMD computers are synchronized by a central clock, such basic assumptions imply that a system can be modeled by PPA only if the propagation delay of a datum through the longest path between any pair of processors is smaller than the clock period. However, even when this condition is not respected, the PPA model can be effectively used to model the behavior of a system and to write algorithms; of course the longer delays have to be taken into account during program compilation.

Although reconfiguration undoubtedly improves mesh connectivity, the PT experience has shown that implementation problems exist which suggest to impose some limitations to the reconfiguration capabilities of a reconfigurable processor array. These limitations, which distinguish PPA from PARBS, are the following:

- **PPA allows only one-direction data movement at any given time**: Unlike PARBS, at any given time PPA supports data propagation only towards one of the four NEWS directions, determined by the central program controller.
- **PPA allows only straight-line data movement**: Unlike PARBS, at any given time, PPA supports data propagation either along the horizontal direction or along the vertical direction; switch-boxes are only allowed to interconnect opposite ports (N->W, W->E, S->N and N->S) and no stair-like interconnection paths (which would require switch-boxes to implement the N->E, N->W, S->E, S->W, E->N, E->S, W->N and W->S connections) can be established.

3. PPA Programming Model

The PPA programming model is expressed by means of a programming language, called PPC (Polymorphic Parallel C), featuring three extensions over the C language. The first extension is the parallel type modifier, which can be used to transform a variable of any fundamental C data type into a parallel variable, which is in a two-dimension array of elements stored in the M-plane. The second extension is the where ... elsewhere statement, which allows to selectively enable the processors which must execute an instruction. The third extension is a set of PPA specific communication and combination primitives. Such primitives are based upon the partitioning of PPA in a set of sub-busses; the orientation and the configuration of the switch-boxes partition a PPA in such a way that each sub-bus includes a node in the OPEN configuration and all the nodes in the SHORT configuration, proceeding toward the current orientation, up to the next node in the OPEN configuration. The following two theorems introduces the communication primitives which will be used in the graph component labeling algorithm and the related notation. The proof of the two theorems can be found in [7].

**Theorem 1 (Broadcast).** Given an orientation and a configuration, which partition a PPA in a set of clusters.
of nodes, in each cluster simultaneously the value of the element of a parallel variable associated to the processor corresponding to the switch-box in the OPEN configuration can be broadcast to all the other processors in \( O(1) \) time.

**Corresponding PPC statement:**

\[
\text{broadcast}(\text{src}, \text{orient}, \text{config})
\]

**Theorem 2 (Selected Minimum Computation).** Given an orientation and a configuration, which partition a PPA in a set of clusters of nodes, and given a parallel logical condition \( L \), in each cluster simultaneously the minimum among the values of all the elements of a parallel integer variable corresponding to TRUE values of \( L \) can be computed and made available to all the processors in the cluster in \( O(h) \) time.

**Corresponding PPC statement:**

\[
\text{sel} \_\text{min}(\text{src}, \text{orient}, \text{config}, L)
\]

### 4. Graph Component Labeling Algorithm

Let \( G = (V, E) \) be an undirected graph in which \( V \) represents the set of vertices \((|V| \leq n)\) and \( E \) represents the set of edges, and let \( A \) be the adjacency matrix of graph \( G \), in which \( a_{ij} = 1 \) if there is an edge between vertex \( i \) and vertex \( j \) and \( a_{ij} = 0 \) if there is not. Two vertices are said to be directly connected if there is an edge between them; two vertices are said to be connected if they are directly connected, or if they are connected to vertices which are directly connected. Labeling the connected component of \( G \) consists of assigning to each vertex \( i \) a label \( l_i \) which is the minimum index of the vertices connected to it.

The PPA graph component labeling (GCL) algorithm proposed here follows the strategy proposed by Shiloack and Vishkin for the CRCW-PRAM model. The readers interested in the formal proof of the algorithm may refer to [2].

The PPC code of the PPA algorithm for GCL, corresponding to the PPC routine \( \text{graph} \_\text{label}() \), uses two parallel global variables, namely \( \text{vrtx} \_\text{num} \) and \( \text{label} \).

- \( \text{vrtx} \_\text{num} = \text{vrtx} \_\text{num} = i \) contain the index assigned to vertex \( i \) during the initialization phase and do not change during the algorithm.

- \( \text{label} = \text{label} \) contain the label assigned to vertex \( i \) and are updated at each iteration.

The algorithm, whose code is shown in Fig. 3, consists of the following steps:

- **Initialization** (statements 26 through 33): Global variables \( \text{vrtx} \_\text{num} \) and \( \text{label} \) are initialized, both in row 0 and in column 0, with the indices of the vertices they represent (\( \text{vrtx} \_\text{num} \)) and with the labels initially assigned (\( \text{label} \)), which coincide with the vertex indices. In other words, initially the graph consists of \( n \) components, each made up of one vertex, and each component has the index of its vertex as a label.

- **Reduction** (statements 34 through 37): A loop of \( \sqrt{n} \) iterations is carried out. At each iteration two functions, namely \( \text{shortcut}() \) and \( \text{hook}() \), are executed.

- \( \text{shortcut}() \) assigns each vertex \( i \) a label equal to the label of the vertex having index equal to the current label of \( i \).

- \( \text{hook}() \) assigns each vertex \( i \) a new label corresponding to the minimum, over the set of vertices having the same label \( l \) as \( i \) among the minima of the labels of the vertices connected to at least one of such vertices.

\( \text{shortcut}() \) is performed as follows: the labels are broadcast from the first column to \( E \) (statement 17), and both the labels and the vertex indices are broadcast from the first row to \( S \) (statements 15 and 16). The labels broadcast from the first row are then sent to the \( W \) direction (statement 19) and loaded in the first column of variable \( \text{label} \) from the first node (in each row) in which the label broadcast from the first column and the vertex index broadcast from the first row are equal.

\( \text{hook}() \) is performed as follows: the labels are broadcast both from the first row to the \( S \) direction (statement 3) and from the first column to the \( E \) direction (statement 4). The minima among the labels are computed on each column along the vertical direction (from \( S \) to \( N \)), considering only the nodes corresponding to elements \( a_{ij} \) of the adjacency matrix \( A \) equal to 1 (statement 6). Such minima are then broadcast to \( S \) (statement 7) and then the minima among them are computed on each row along the horizontal direction (from \( E \) to \( W \)), considering only the vertices having the same label (statement 9).

```plaintext
1: hook () {  
2: \parallel \text{int} l_\_s, l_\_e, t_1, t_1_\_s;  
3: l_\_s=\text{broadcast}(\text{label}, S, \text{ROW}=0);  
4: l_\_e=\text{broadcast}(\text{label}, E, \text{COL}=0);  
5: \text{where} (\text{ROW} == 0);  
6: l_1=\text{sel} \_\text{min}(l_\_e, N, \text{ROW}==\sqrt{n} -1, A);  
7: l_\_s=\text{broadcast}(l_1, S, \text{ROW}==0);  
```
8: where (COL == 0)
9: sel_min(t1_s,w,COL=0-1,l_s=l_e);
10 first-col-to-first-row (label);
11 }
12:
13: shortcut () {
14: parallel int v_s, l_s, l_e;
15: v_s=broadcast(vrtx_num,S,ROW==0);
16: l_s=broadcast(label,S,ROW==0);
17: l_e=broadcast(label,E,COL==0);
18: where (COL == 0)
19: label=broadcast(l-s,W,l-e==v-s);
20: first-col-to-first-row (label);
21: }
22
23: graph_label ()
24: {
25: int cnt;
26: where (ROW == 0) {
27: vrtx_num = COL;
28: label = COL;
29: }
30: where (COL == 0){
31: vrtx_num = ROW;
32: label = ROW;
33: }
34: for (cnt = 0; cnt < log √n + 1; cnt++) {
35: shortcut ();
36: hook();
37: }
38: }
Fig. 3 - Graph Component Labeling in PPA

The complexity of GCL in PPA is O(log²n), considering that graph_label() includes a loop of √n iterations (complexity O(√n) = O(log n)), and that hook(), which is called at each iteration of such a loop, involves the computation of the minimum among √n bit numbers (complexity O(log √n) = O(log n) (Theorem 2). It is interesting to see that in the GCL algorithm PPA works as a CRCW-PRAM in the concurrent writes of which the minimum among the values concurrently written in the shared memory is actually written, as opposed to the the CRCW-PRAM model adopted by Shiloach and Vishkin, in which, in case of conflict, the value written in memory is unpredictable. The loss in performance with respect to the Shiloach and Vishkin algorithm (O(logn)) is due to the fact that the concurrent writes with deterministic result are emulated in PPA with O(log n) complexity.

5. Conclusion

We have presented a O(log² n) algorithm for connected component labeling in Polymorphic Processor Arrays (PPA). The same algorithm can be used to compute the transitive closure of a matrix, as shown by Wang and Chen [1]. PPA is a realistic reconfigurable processor array architecture. Although more efficient algorithms have been proposed for connected component labeling and transitive closure on other types of reconfigurable processor arrays [1, 5, 6], the algorithm presented here is optimum considering that the specific communication capabilities required are scalable and suitable for VLSI implementation, as the Polymorphic-Torus experience has demonstrated.

References