A Parallel Algorithm for Minimum Cost Path Computation on Polymorphic Processor Array*

P. Baglietto♣, M. Maresca♠ and M. Migliardi♣

♣ DIST - University of Genoa
via Opera Pia 13 - 16145 Genova, Italy
email baglietto@dist.unige.it

♠ DEI - University of Padua
via Gradenigo - Padova, Italy
email maresca@dei.unipd.it

Abstract. This paper describes a new parallel algorithm for Minimum Cost Path computation on the Polymorphic Processor Array, a massively parallel architecture based on a reconfigurable mesh interconnection network. The proposed algorithm has been implemented using the Polymorphic Parallel C language and has been validated through simulation. The proposed algorithm for the Polymorphic Processor Array, delivers the same performance, in terms of computational complexity, as the hypercube interconnection network of the Connection Machine, and as the Gated Connection Network.

1 Introduction

This paper describes a parallel algorithm for the computation of the Minimum Cost Path (MCP) on the Polymorphic Processor Array (PPA), a massively parallel architecture based on a reconfigurable mesh interconnection network.

The PPA architecture, which supports the proposed MCP algorithm, is particularly interesting due to the fact that, unlike other reconfigurable processor arrays [1], the PPA architecture can be effectively implemented in hardware [2] and to the fact that it can be programmed at a high level [3]. The PPA is a two-dimensional mesh connected computer in which each node is equipped with a switch able to interconnect its four ports. PPA changes the switch setting, as part of the instruction, to speed up the data exchange between nodes; it shortens, with respect to the simple mesh, the distance between the nodes that have to communicate by short-circuiting all the intermediate nodes between the source and destination nodes.

The MCP algorithm proposed in this paper exploits the specific features of the reconfigurable interconnection network of the PPA architecture and takes advantage of the natural matching both between the data structure of the problem (i.e. the matrix of the weigths associated to each edge of a graph) and that of the PPA architecture (a mesh of processing elements) and the capability of the interconnection network to let information flow from one node to a set of nodes by means of reconfigurable busses.

* The work described in this paper has been supported by research grants from CNR (Italian Consiglio nazionale delle Ricerche) and MURST (Ministero dell'Università e della Ricerca Scientifica).

The proposed algorithm has been implemented using the Polymorphic Parallel C language and has been validated through simulation.
In the following sections we first summarize the PPA computational and programming models. Then we describe the algorithm able to carry out the MCP computation and finally we give some concluding remarks

2 Features of the Polymorphic Processor Array

Like most massively parallel computers, PPA [2] is a processor array based on a mesh interconnection network. Unlike regular mesh Processing Elements (PEs), PPA PEs are equipped with switches which allow the array to reconfigure itself. The PPA PEs are connected by means of two sets of busses(see figure 1a) and the data movement direction is the same for all the PEs. This means that, at any given time, all the nodes send data in the same direction (*North*, *East*, *West* or *South*), which is selected by the SIMD program controller. On the contrary, the switch box configuration can be different in each PE depending on a local condition.
Two switch-box configurations are allowed, called *Open* and *Short*: in the *Open* configuration, a switch-box disconnects the two busses by which it is traversed, and allows the corresponding PE to inject data into one of them; in the *Short* configuration, on the contrary, a switch box lets the data propagate on the busses, according to the current data movement direction, and prevents its PE from injecting data into the bus system. Let us consider, as an example, the case in which the switch-box orientation programmed by the controller is *East*, such as shown in figure 1b: in each node the PE receives data from the *West* port whereas, depending on the switch-box configuration, either the *West* port is connected to the *East* port (short switch-box configuration) allowing data to proceed from *West* to *East*, or the PE itself is connected to the *East* port, allowing the PE to send data out (*Open* switch-box configuration). Thus it is possible to dynamically partition the two sets of PPA busses into a number of independent sub-busses that group cluster of adjacent nodes.
The specific features of the PPA computational model are captured in the PPA programming model which is expressed by means of the Polymorphic Parallel C (PPC) high level language which, like many other parallel languages, is derived from

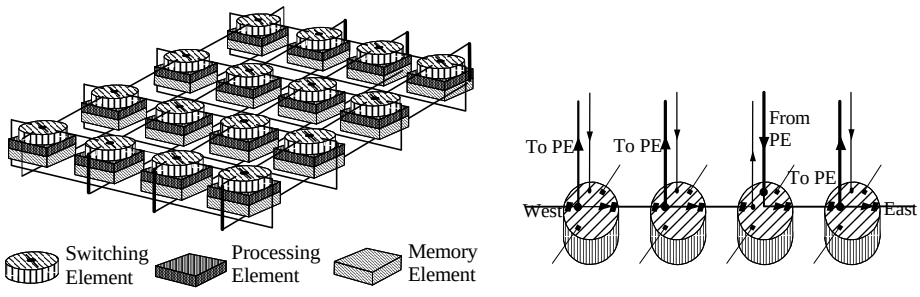

Figure 1 - PPA architecture.

the C language. The main features of the PPC parallel programming language are (see [3] a complete description of the language):

a memorization class called parallel, which allows to specify that a variable must be allocated in multiple copies in the local memory of each PE instead of in the memory of the central controller.
Therefore a parallel data declaration, as every data declaration in C, consists of a memorization class (usually optional, but required for parallel data declarations) and a type identifier followed by at least one variable name. For example:

```
parallel float A;
```

denotes an array of floating point variables each element of which is associated to a different local memory.

a control structure called where which allows to define the status of each PE and to partition the PEs into two sets. The first set is composed by PEs that verify the *expression*: the PEs belonging to this set execute only the first instruction group. The second set is composed by all remaining PEs that execute the second instruction group.
The new control structure is denoted by the syntax:

```
where (expression)
    <instruction group 1>;
elsewhere
    <instruction group 2>;
```

a set of communication primitives which allows to easily exploit the specific communication mechanisms provided by PPA. Primitives belonging to this set and used to implement the EDT algorithm are:

```
shift(src, dir)
```

which allows to send data associated to the *src* variable from each PE to its nearest neighbor along a selected *dir* direction, and

```
broadcast(src, L, dir)
```

which models the basic data movement technique of PPAs. Let *L* be a parallel logical variable each element of which, assigned to a PPA node, can assume a value in the set (*Open, Short*); let *src* be a parallel variable of any type, and let *dir* be a scalar variable belonging to the set (*North*, *East*, *West*, *South*). Variable *L* partitions a PPA into a set of clusters of nodes, depending on the value of *dir*. Once a PPA has been partitioned into a set of clusters of nodes, in each processor the *broadcast* communication primitive returns the value of the element of the variable *src* corresponding to the extreme node of the cluster the processor belongs to.

3 Minimum Cost Path Algorithm

Let us now consider the generic problem of finding the minimum cost path (MCP) from all the vertices *n* of a graph *G* == (V, E) to one specific destination vertex, called *d*, and let us use a dynamic programming approach to solve such a problem. Let *G* be represented by matrix W, each element of which, w_{ij}, is the weight associated to the edge from vertex *i* to vertex *j*; if no edge exists from vertex *i* to vertex *j*, then w_{ij} = MAXINT, that is an infinite value. The PPA algorithm for MCP is a modification of the algorithm presented in [4] for the Connection Machine and in [5] for the Gated

Connection Network.
In the minimum_cost_path() code showed below parallel global variable *W* and global variable *d* are the input data and respectively contain the weight matrix and the index of the destination vertex, while parallel global variables *SOW* (Sum of Weights) and *PTN* (Pointer To Next) are the output data and, at the end of the algorithm, respectively contain the cost and the structure of the MCPs. Only the elements in the *d*-th row of *SOW* and *PTN* are meaningful: SOW_{id} contains the sum of the weights in a MCP from *i* to *d*, while PTN_{id} contains the index of the vertex following *i* in a MCP to *d*. The algorithm works as follows:

Step 1 - *Initialization* (statements 4 through 7).
Each element of *SOW* in the *d*-th row, that is SOW_{id} , $i = 0 , ... , \sqrt{n} - 1$, is initialized with the weight associated to the link from vertex *i* to vertex *d*.
Each element of *PTN* in the *d*-th row, that is PTN_{id} , $i = 0 , ... , \sqrt{n} - 1$, is initialized with the index of the next vertex, after vertex *i*, in a 1-edge MCP to vertex *d*, that is *d* itself.
At the end of step 1, all the MCPs from any vertex i to vertex d have been computed assuming only 1-edge paths. Of course if there is no 1-edge path from a vertex *i* to vertex *d*, that is if there is no edge from vertex *i* to vertex *d*, then SOW_{id} = MAXINT.

Step 2 - *RMCP computation* (statements 8 through 20).
A do-while loop of *t* iterations is executed, where *t* is the maximum MCP length from any vertex *i* to vertex *d*. At each iteration *k*, the *SOW*s are updated considering up to *k*-edge paths. Each element of the *d*-th row of *SOW*, that is SOW_{id} , $i = 0 , ... , \sqrt{n} - 1$, is updated with the minimum between its old value, that is the cost of the path already computed, and the minimum among the sums of w_{ij} and SOW_{jd} for all *j*'s, that is the cost of a new path one edge longer than the previous one. If a SOW_{id} does not change, then it means that adding the possibility of having paths one edge longer than at the previous iteration does not reduce the cost of the path from vertex *i* to vertex *d*; in this case PTN_{id} does not change either. On the contrary, if a SOW_{id} changes, then it means that the possibility of having paths one edge longer than at the previous iteration leads to a less expensive path from vertex *i* to vertex *d*; in this case PTN_{id}, which contains the index of the next vertex in the MCP, is replaced with the new index of the next vertex in the new MCP.
The updating of the *SOW*s is done the following way:

- The *SOW*s in the *d*-th row are broadcast along the columns, in such a way that any PPA vertex at coordinates *ij* is loaded with $SOW_{jd} + w_{ij}$ (statement 10), that is the sum of the weight from vertex *i* to vertex *j* and of the SOW from vertex *j* to the destination *d*.
- The minimum among the sums of the weights from vertex *i* to the destination *d* using any of the other vertices as the first vertex after *i* is computed in parallel for each vertex *i* using the PPA min() combination primitive along the PPA rows (statement 11). Such a minimum is loaded in min_sow_{ii}.
- Once found such a minimum, the updating of the *PTN* pointer is done by loading in PTN_{ii} the index, or one of the indices in case there is more than one, of the next

vertex belonging to the MCP (statement 12).

- The *SOW* and *PTN* variables are then sent to the *d*-th row to be used as the starting values for the next iteration (statements 17 and 18).

```
1:  minimum_cost_path()
2:  {
3:  parallel int OLD_SOW;
4:  where (ROW == d){
5:     SOW = W;
6:     PTN = d;
7:     }
8:  do
9:     where (ROW != d) {
10:        SOW = broadcast (SOW, SOUTH, ROW == d) + W;
11:        MIN_SOW = min (SOW, WEST, COL == (√n  - 1));
12:        PTN = selected_min (COL, WEST, COL == (√n  - 1), MIN_SOW == SOW);
13:        }
14:   where (ROW == d) {
15:        OLD_SOW = SOW;
16:        SOW = broadcast (MIN_SOW, SOUTH, ROW == COL);
17:        where (SOW != OLD_SOW)
18:             PTN = broadcast (PTN, SOUTH, ROW == COL);
19:        }
20: while (at least one SOW in row d has changed);
21: }
```

The minimum among the values of all the elements of a parallel integer object of size *h* bits can be computed and made available to all the processors in a cluster in O(*h*) time. The minimum computation is carried out by the min() and selected_min() routines. We present below the code of the min() routine only (the code for the selected_min routine is similar):

```
1   :  parallel int min (src, orientation, L)
2   :  parallel int src ;
3   :  enum {NORTH,EAST,SOUTH,WEST} orientation;
4   :  parallel logical L;
5   :  {
6   :  int j;
7   :  parallel logical enable = 1;
8   :  for (j=h-1; j≥0; j--)
9          where (     broadcast(or(!bit(src, j) && enable, orientation, L), orientation, L)
                       && bit(src, j)   )
10             enable = 0;
11  :  where (L)
12  :        src = broadcast (src, opposite(orientation), enable);
13  :  return(broadcast (src, orientation, L));
14  :  }
```

Here h denotes the size of integer objects, opposite (x) denotes the direction opposite to x, and bit(x, i) denotes a parallel function returning the value of the i-th bit plane of parallel object x.

The min() and selected_min() algorithms examine all the values to be compared simultaneously, bit by bit, starting from the most significant position; at each iteration,

if at least one 0 is found, all the values having 1 at that position are excluded from the following comparisons (note that while the min() algorithm starts considering all the values, the selected_min() algorithm starts considering a subset of the values defined by its fourth input parameter). At the end of the scanning of src, the minimum values just found are sent to the first nodes of the clusters (statements 11-12) and then broadcast to all the nodes of the clusters (statement 13). Considering that all the statements have O(1) complexity, and that a h-iteration loop must be executed, the two algorithms have O(*h*) complexity.
The complexity of MCP in PPA is therefore O(*p* log *h*), where *p* is the maximum length of the MCPs to the destination vertex *d* and log *h* is due to the complexity of the min and selected_min operations carried out at each iteration.

4 Concluding Remarks

The Polymorphic Processor Array (PPA) computation and programming models reveals an interesting relationship between the *power of the model* and *programming*. The *row/column only* PPA is a less powerful model with respect to the Reconfigurable Mesh[1], the Gated Connection Network (GCN)[5] and the Processor Arrays with Reconfigurable Bus System (PARBS) [6], and exhibits a limited number of reconfiguration capabilities. Nevertheless it is hardware implementable and enjoys the programming efficiency as the MCP algorithm shows.
The complexity of Minimum Cost Path (MCP) algorithm on PPA is O(p log h), where p is the maximum length of the MCPs and log h (where h is the number of bits for integer representation) is due to the complexity of the min and selected_min PPA operations used in the MCP algorithm; PPA delivers the same performance, in terms of computational complexity, as the hypercube interconnection network of the Connection Machine, and as the Gated Connection Network..

References

[1] R. Miller, V.K. Prasanna-Kumar: D.I. Reisis and Q. F. Stout, Parallel *Computations on Reconfigurable Meshes.* IEEE Trans. on Computers, Vol. 42, No. 6, pp. 678692, June 1993

[2] M. Maresca, H. Li and P. Baglietto: *Hardware Support for Fast Reconfigurability in Processor Arrays*. Proc. of the International Conference on Parallel Processing (ICPP), St. Charles (IL), August 1993.

[3] M. Maresca and P. Baglietto: *A Programming Model for Reconfigurable Mesh Based Parallel Computers*. Proc. of the Working Conference on Massively Parallel Programming Models (IEEE Press), Berlin (Germany), Sept 20-22, 1993.

[4] W. D. Hillis: *The Connection Machine*. The MIT Press, Cambridge (MA), 1985.

[5] D. B. Shu, J. G. Nash: *The Gated Interconnection Network for Dynamic Programming*. Concurrent Compu-tations, S. K. Tewsburg, B. W. Dickinson and S. C. Schwartz eds., Plenum Publishing Company, pp. 645-658.

[6] B. F. Wang and G. C. Chen: *Constant Time Algorithms for the Transistive Closure and Some Related Graph Problem on Processor Arrays with Reconfigurable Bus System*. IEEE Trans. on Parallel and Distribuited Systems Vol. 1, No. 4, pp. 500-507, 1990.