Bull. Nov. Comp. Center, Comp. Science, 19 (2003), 15–24

Associative parallel algorithm performing depth-first search

T.V. Borets

In this paper, we propose a novel associative parallel algorithm performing depth-first search on an abstract model of the SIMD type with vertical data processing (the STAR-machine). This algorithm is represented in two ways: as recursive and non-recursive STAR procedures, whose correctness is verified and time complexity is evaluated.

1. Introduction

Algorithms of systematic traversing of a graph, when each vertex is inspected exactly once or the next traversal of vertex do not affect the run of the algorithm, have an important place in the graph theory. The best known among these algorithms are the depth-first search and the breadth-first search. On the basis of these methods of graph traversing many algorithms were developed, such as graph testing for connectivity, checking spanning trees for optimality, finding the immediate dominators [1], and others.

The implementations of these algorithms on the RAM are well known and described in many books of graph theory, for example [2]. On sequential computers, the algorithms take $O(m+n)$ time, where n is the number of vertices and m is the number of arcs in the given graph.

We are interested in the implementation of the algorithms on associative parallel computers, because they are mainly oriented to solve the non-numerical problems. This class of parallel computers includes the well-known systems STARAN, DAP, MPP, and CM-2. Such architecture provides a massively parallel search by contents and processing of unordered data represented in the form of two-dimensional tables [3].

The Breadth-first search is very naturally performed on this architecture. A group of associative parallel algorithms based on it has been enumerated in [4].

In this paper, we show how to implement the depth-first search on associative parallel computers.

2. Model of associative parallel machine

Now, we will describe the model. It consists of the following components:

- a sequential control unit, where programs and scalar constants are stored;
- an associative processing unit consisting of p single-bit processing elements (the PEs);
- a matrix memory for the associative processing unit.

To simulate data processing in the matrix memory, the STAR–machine uses new data types: **word**, **slice**, and **table**. The types **slice** and **word** simulate bit column access and bit row access, respectively. The type **table** is used for a definition of tabular data.

The language STAR is described in [5]. In this paper, we need the following operations and predicates for slices.

Let X, Y be variables of the type **slice** and i be a variable of the type **integer**. We use the following operations:

SET(Y) sets all the components of Y to $'1'$;

CLR(Y) sets all the components of Y to $'0'$;

$Y(i)$ selects the i-th component of Y;

FND(Y) returns the ordinal number i of the first (or the uppermost) component $'1'$ of Y, $i \geq 0$;

STEP(Y) returns the same result as FND(Y) and then resets the first $'1'$ found to $'0'$.

The predicates ZERO(Y) and SOME(Y) and the bitwise Boolean operations X *and* Y, X *or* Y, *not* Y, X *xor* Y are introduced in the usual way.

Let T be a variable of the type **table**. We employ the following two operations:

ROW(i, T) returns the i-th row of the matrix T;

COL(i, T) returns the i-th column of T.

Following Foster [6], we assume that each elementary operation is performed at once. Then the complexity of algorithm is defined as a number of elementary operations, performed in worse case.

3. Preliminaries

A directed graph (digraph) is a pair of sets (V, A), where $V = \{1, \ldots, n\}$ is a set of vertices, A is a set of arcs, $A \subseteq V \times V$.

Let $e = (u, v) \in A$ be an arc, then the vertex v is the **head** of the arc e and the vertex u is its **tail**.

The depth-first search (DFS) is a technique to traverse a graph systematically. It uses a strategy: to go as deeply as it is possible (there is an outgoing arc that has not been passed) and to search another path, otherwise. The DFS runs until it traverses each vertex that is reachable from the given vertex.

The depth-first search numbers vertices with $1, 2, \ldots, n$ in order, in which they were visited for the first time. The number assigned by the depth-first search to a vertex will be called *depth-first search number* (the DFS-number). The depth-first search also computes a spanning tree of the graph called *the DFS-tree*.

In the STAR-machine matrix memory, a directed graph will be represented as an association of the matrices *left* and *right*, where each edge $(u, v) \in A$ is matched with the pair $\langle u, v \rangle$. We also use the matrix *code* containing binary codes of vertices. Let us recall that vertices are integers represented as binary strings. A tree will be represented as a slice, in which the positions of tree arcs will be marked by $'1'$.

4. Associative parallel algorithm performing depth-first search

On sequential computers, the depth-first search can be represented in two ways: by a recursive procedure and by a procedure using a stack. The associative parallel algorithm can be represented by the same ways. Both implementations will be shown below. They use the following input parameters:

- matrices *left*, *right*, and *code*, which give the problem graph;
- *root*, saving the binary code of the selected vertex.

After execution of the procedure

- the matrices *left*1, *right*1 give the graph arcs in the DFS-numbering;
- the matrix NV stores the DFS-numbers of the vertices;
- the slice T stores the positions of arc belonging to DFS-tree. We assume that it is empty before algorithm execution.

Before describing the algorithm, we adduce some basic and auxiliary procedures.

4.1. Basic and auxiliary procedures. We will use the following two basic procedures from [7].

The procedure MATCH(T, X, v, Z) defines in parallel positions of those rows of the given matrix T, which coincide with the given pattern v written in binary code. It returns the slice Z, where $Z(i) = '1'$ if and only if $\mathrm{ROW}(i, T) = v$ and $X(i) = '1'$.

The procedure WMERGE(w, X, T) writes in parallel the word w in those rows of the given matrix T, whose positions are marked by $'1'$ in the slice X. It returns the matrix T, where $\mathrm{ROW}(i, T) = w$ if $X(i) = '1'$, and unchanged, otherwise.

These procedures take $O(k)$ time, where k is a number of bit columns.

Now, we propose the following auxiliary procedure that can be found in the appendix.

The procedure OrdNode(*left*, *right*, *code*, *node*, *ord*, X, *left*1, *right*1, NV, Y) uses the matrices *left*, *right*, and *code* to represent a given graph, and binary word *node* to represent a vertex to set a DFS-number. The current DFS-number is stored in integer variable *ord*. The procedure returns the slices X and Y, and the matrices *left*1, *right*1, NV.

It runs as follows:

- finds the binary code for *ord*;
- finds the position k of *node* in the matrix *code*, and writes the binary code of *ord* to the k-th string of the matrix NV;
- finds positions of strings that coincide with *node* in the matrix *left* and writes the DFS-number of *node* in the corresponding strings of the matrix *left*1 and $'1'$ in the corresponding positions of Y, if there are $'1'$ in X.
- finds positions of strings that coincide with *node* in the matrix *right* and writes the DFS-number of *node* in the corresponding strings of the matrix *right*1 and $'0'$ in the corresponding positions of X.

Now, we give some explanation to procedure's run. The problem graph is shown in Figure 1. We want to number vertex 5 and assume that vertices

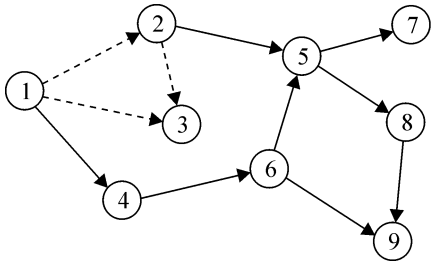

Figure 1. The graph just before execution of OrdNode

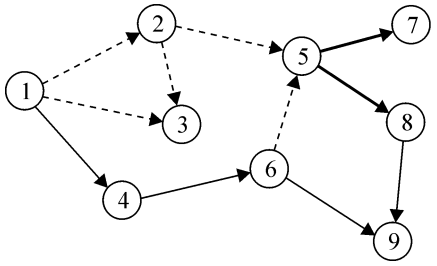

Figure 2. The graph just after execution

1, 2, 3 are numbered. Then *node* is equal to the binary code of 5, $ord = 4$. The slice X stores the positions of solid arcs. Figure 2 shows the graph just after numbering the vertex. The slice Y saves the positions of bold arcs, $ord = 5$. The positions of the arcs $(2, 5)$ and $(6, 5)$ are marked by $'0'$ in X.

4.2. Representation of the algorithm using recursion. In this subsection, we will describe a recursive algorithm for finding the DFS-tree. This is performed by a procedure DFS$*$ using the above parameters.

The procedure also uses the following input variables. Integer *ord* stores the current number in decimal code. All arcs ingoing to unnumbered vertices are marked by $'1'$ in the slice X. We assume that $ord = 1$ and the slice X saves only $'1'$ (it can keep some $'0'$ in the case, when we do not need to traverse any part of the graph).

The procedure also uses some additional variables: *node* keeping the binary number of a vertex to be numbered by the current *ord*; the slice Y saving the positions of arc outgoing from the *node* to unnumbered descendants.

The idea of the algorithm:

1. Give the number to the vertex *root* and remember the positions of arcs ingoing to unnumbered descendants.
2. While the set of such arcs is non-empty:
   - choose the first descendant *node*;
   - add the arc $(root, node)$ to the DFS-tree;
   - repeat the procedure for the vertex *node*.

```
  Procedure DFS*(left, right, code: table; root: word;
                 Var X: slice; var ord: integer;
                 Var left1, right1: Table; Var NV: Table;
                 Var T: slice);
  Var Y: Slice; node: word; j: integer;
 1 Begin
 2   OrdNode(left,right,code,root,ord,X,left1,right1,NV,Y);
 3   while SOME(Y) do
 4   begin
 5     j:=STEP(Y);
 6     T(j):='1';
 7     node:=ROW(j,right);
 8     DFS*(left,right,code,node,X,ord,left1,right1,NV,T);
 9     Y:=Y and X;
10   end;
11 End;
```

Theorem. *Let a directed graph $G = \langle V, A \rangle$ with the selected vertex root be given as described above. Let the slice X and ord satisfy the above requirements. Then the procedure DFS$*$ adds the positions of arcs belonging to the DFS-tree to the slice T. The matrix NV stores the DFS-numbers of the traversed vertices. The matrices $left1$ and $right1$ give the problem graph in the DFS-numbering.*

Proof. It will be given by induction on p, where p is the number of vertices accessible from the *root*.

Base of induction $p = 1$: As a result of execution of OrdNode in line 2 the vertex *root* takes the DFS-number ord_{in}. The slice X does not contain the positions of arcs going at *root*, $ord_{out} = ord_{in} + 1$. In the matrices $left1$ and $right1$, the entry of the vertex *root* are changed by the binary code of ord_{in}. There is a single $'1'$ in the slice Y. Its position corresponds to the arc going to the unnumbered vertex. This position is added to the slice T representing the DFS-tree. The number of this unnumbered vertex is saved in the variable *node*. After that the procedure DFS$*$ is performed for the vertex *node*. After execution of the procedure OrdNode for the vertex *node*, the slices X and Y are empty because all arcs going to either *root* or *node* have been deleted from X. Go out to the first level of recursion. Because the value of X is transmitted above, the procedure stops its run. As a result of the procedure only one arc $(root, node)$ is added to the slice T. This arc is the DFS-tree.

Step of induction. Let the theorem be proved for $p - 1$, prove it for p. After execution of the procedure OrdNode the vertex *root* has a DFS-number, the slice Y stores the positions of the arcs going from the *root* to unnumbered vertex. Since there are $p \neq 0$ vertices accessible from *root*, the slice Y is not empty. An arc to continue traversing is chosen from the arcs marked by $'1'$ in Y and the *node* saves the head of this arc. Let V_{node} be the set of vertices reachable from the vertex *node* along the arcs marked by $'1'$ in X. $|V_{node}| \leq p - 1$ because there are no arcs going to the *root*.

1. $|V_{node}| = p - 1$. Then by the proved earlier the arc $(root, node)$ and the DFS-tree rooted in *node* are marked by $'1'$ in the slice T after execution of line 8. All the vertices are numbered and the slice X is empty. And after the execution of line 9 the slice Y is empty too. The procedure stops its run.

2. $0 \leq |V_{node}| < p - 1$. Then after execution of line 8 the positions of the arc $(root, node)$ and the arcs belonging to the DFS-tree rooted in *node* are marked by $'1'$ in the slice T. All the numbered vertices are not accessible from the *root* along the arcs marked by $'1'$ in the slice X. But some of vertices accessible from *root* are unnumbered. The positions of all arcs going to them are marked by $'1'$ in the slice X. After execution of line 9 the arcs going from *root* to the unnumbered vertices are marked in the slice

Y and it is not empty. Let us get onto the graph, the set of vertices of which is equal to $V \setminus (\{node\} \cup V_{node})$. In this graph, there are less than p vertices accessible from *root*. Then by the proved, the procedure constructs the DFS-tree for this graph. Joining the DFS-trees for both subgraphs gives the tree of the original graph. □

Since each vertex is numbered only one time and we go only through tree arcs (no more than two times), the procedure takes $O(n \log n)$ time.

4.3. Representation of the algorithm using stack. In this subsection, we describe the non-recursive algorithm of the DFS-tree construction. Like in the previous case, the procedure DFS uses the matrices *left*, *right*, and *code*, and the binary word *root* and returns the DFS-tree as the slice T and the matrix of the DFS-numbers NV. The matrices *left*1 and *right*1 store arcs of the graph in the DFS-numbering.

The parameters X and *ord* become auxiliary. Moreover, we need some more auxiliary variables.

The matrix *LIFO* simulates a stack. Addition of a column to the *LIFO* is performed in the following way: after the vertex *node* gets the DFS-number and an arc going to unnumbered descendant is chosen, the positions of other such arcs (if any) are put to a column of the matrix *LIFO*. So in each column there are positions of arcs going from the same vertex. The deletion of a column from *LIFO* is performed when the vertex *node* has no unnumbered descendants. Integer variable *lif* stores the depth of the stack.

Idea of the algorithm:

1. Initialize the variables.
2. Number the *root* and save the positions of arcs ingoing to unnumbered descendants.
3. While there are some arcs going to unnumbered vertices, do the following:
   - if the vertex having the last DFS-number has no unnumbered descendants, then seek a vertex in the stack that has maximum DFS-number from the vertices having arcs going to unnumbered descendants; remember the positions of such arcs;
   - choose an arc to continue the traversing and mark its position in the tree; add the other arcs to the stack;
   - number the head of the arc, remember the positions of arcs going to unnumbered descendants.

```
Procedure DFS(left, right, code: table; root: word;
              Var left1, right1: table; Var NV: table;
              Var T: slice);
Var LIFO: table; {simulates a stack of arcs}
    lif, {gives a depth of the stack}
    ord, {current DFS-number}
    k: integer;
    X, Y: slice {left}
    node: word;
Begin
1   SET(X); CLR(T); SET(U); ord:=1; lif:=1;
2   OrdNode(left,right,code,root,ord,X,left1,right1,NV,Y);
3   while SOME(X) do
```

(∗ X keeps the positions of arcs going to the unnumbered vertices ∗)

```
4   begin
5     while ZERO(Y) do
6     begin lif:=lif-1;
7       Y:=COL(lif,LIFO);
8       Y:=Y and X;
9     end;
10    k:=STEP(Y);
11    node:=ROW(k,right);
```

(∗ We add the arc to the tree ∗)

```
12    T(k):='1';
13    if SOME(Y) then
14    begin
```

(∗ If there are any arcs going from *node* to unnumbered vertices, then we put them to the stack ∗)

```
15      COL(lif,LIFO):=Y;
16      lif:=lif+1;
17    end;
18    OrdNode(left,right,code,node,ord,X,left1,right1,NV,Y);
19  end;
End;
```

Lemma. *If all vertices of the given graph are accessible from the root, then the cycle 5–9 stops its run, while the slice Y stores at least a single '1' and $lif \geq 1$.*

Proof. The decrease of *lif*'s value occurs within the cycle 5–9.

Let us consider the run of this cycle more exactly. The condition means that there are no arcs going from the considered vertex to unnumbered

vertices. Just before the considered vertex is the vertex having the maximal DFS-number. Then we return along the stack. We remind that each column of *LIFO* stores the positions of arcs, which are going from the same vertex v to vertices, which had no DFS-number before numbering the vertex v; some of these vertices have been numbered afterwards.

Let us suppose that $lif = 0$. It means there are no arcs going from a numbered vertex to an unnumbered vertex (otherwise *lif* stores the number of the column, where such arc is). But as there are some arcs going to unnumbered vertices (the procedure has not stopped its run), there is some vertex not accessible from *root*. But this contradicts the conditions of the lemma. □

It should be noted that stack simulation on the STAR-machine has been presented in [8].

5. Conclusion

In this paper, we have proposed a novel associative parallel algorithm performing depth-first search on a directed graph represented on the STAR-machine as a list of pairs. To perform the DFS on undirected graph each edge (u, v) is to be written in the list as $\langle u, v\rangle$ and $\langle v, u\rangle$. In this case, the algorithm also takes $O(n \log n)$ time, because its complexity does not depend on the number of arcs. Recall that it is less than the implementation on sequential computers. It is because we can remove the positions of arcs going to a numbered vertex at once. There is the factor $O(\log n)$ because vertical processing.

The algorithm have been represented as the recursive and non-recursive procedures. Making some modifications we can obtain the algorithm returning the DFS-tree represented a matrix of paths. It may be useful for solving some problems.

Our implementation of depth-first search opens wide field for associative parallel realization of many sequential algorithms using it.

References

[1] Lengauer T., Tarjan R.E. A fast algorithm for finding dominators in a flowgraph // ACM Translations on Programming Languages and Systems. – July 1979. – Vol. 1, № 1. – P. 121–141.

[2] Thomas C.H., Leiserson C.E., Rivest R.L. Introduction to Algorithms. – New York: McGraw-Hill, 1990.

[3] Potter J.L. Associative Computing: A Programming Paradigm for Massively Parallel Computers. – New York and London: Kent State University, Plenum Press, 1992.

[4] Nepomniaschaya A.S., Borets T.V. Associative parallel algorithm of checking spanning trees for optimality // Joint NCC & IIS Bulletin, Series Comp. Science. – Novosibirsk: NCC Publisher, 2002. – Issue 17. – P. 75–88.

[5] Nepomniaschaya A.S. Language STAR for associative and Parallel computation with vertical data processing // Proc. of the Intern. Conf. "Parallel Computing Technologies". – Singapure: World Scientific, 1991. – P. 258–265.

[6] Foster C.C. Content Addressable Parallel Processors. – New York: Van Nostrand Reinhold, 1976.

[7] Nepomniaschaya A.S., Dvoskina M.A. A simple implementation of Dijkstra's shortest path algorithm on associative parallel processors // Fundamenta Informaticae. – Amsterdam: IOS Press, 2000. – Vol. 43. – P. 227–243.

[8] Nepomniaschaya A.S. Efficient implementation of Edmonds' algorithm for finding optimum branchings on associative parallel processors // Proc. of the Eighth Intern. Conf. on Parallel and Distributed Systems (ICPADS'01). – KyongJu City, Korea: IEEE Computer Society Press, 2001. – P. 3–8.

Appendix

```
Procedure OrdNode(left, right: table; code: table; node: word;
                  var ord: integer; Var X: slice{left};
                  Var left1, right1: table; Var NV: table;
                  Var Y: slice{left});
Var U, V: slice{code}; Z: slice{left}; w: word; J: integer;
Begin
  SET(Z); SET(U);
  MATCH(code,U, node,V); j:=FND(V);
  w:=ROW(ord,code);
```

(* We give the DFS-number *)

```
  ROW(j,NV):=w; ord:=ord+1;
  MATCH(right, Z, node, Y); WMERGE(w, Y, right1);
```

(* We remove all arcs going to *node* *)

```
  X:=X and (not Y);
  MATCH(left, Z, node, Y); WMERGE(w, Y, left1);
  Y:=Y and X;
End;
```