Bull. Nov. Comp. Center, Comp. Science, 21 (2004), 85–97

A new technique for updating tree paths on associative parallel processors*

A.S. Nepomniaschaya

Abstract. In this paper we describe in detail a new technique for updating tree paths on a model of associative parallel systems with vertical data processing (the STAR-machine). It includes a new associative parallel algorithm for finding an MST along with the matrix of tree paths and a new associative parallel algorithm for updating tree paths after every change in the underlying graph. We prove correctness of the corresponding procedures and evaluate time complexity. Moreover, we compare two techniques for updating tree paths on the STAR-machine and the CREW PRAM machine.

1. Introduction

Associative (content-addressable) processors constitute a prominent subclass of fine-grained massively parallel SIMD architectures. Recent advances in VLSI technology have made large associative processors and other massively parallel architectures practically realizable [7]. Associative systems with bit-serial (vertical) data processing are best suited to solve nonnumerical problems. Such an architecture performs data parallelism at the base level, provides massively parallel search by contents, and allows one using two-dimensional tables as basic data structures [9].

In this paper, we suggest a new technique for updating tree paths on associative parallel processors. In particular, such a problem arises when we perform dynamic edge update of a minimum spanning tree (MST). Dynamic graph algorithms are designed to handle graph changes. Such algorithms maintain some property of a changing graph more efficiently than recomputation of the entire graph with a static algorithm after every change. The problem of edge updating an MST involves reconstructing a new MST from the current one when an edge is deleted or inserted or its weight changes.

Different techniques are used to solve update problems. In [10], Tarjan proposes a special technique, path compression on balanced trees, to compute functions defined on paths in trees under various assumptions. This technique is applied to solve several graph problems. In [3], Frederickson suggests a graph decomposition and data structures techniques to deal with the edge update problem. In particular, Frederickson presents an $O(m^{1/2})$

*Partially supported by the Russian Foundation for Basic Research under Grant 03-01-00399.

sequential algorithm for the edge update problem, where m is the number of graph edges. In [1], a general technique, called sparsification, for designing dynamic graph algorithms is provided. In particular, the authors propose a sequential algorithm for edge updating a minimum spanning forest in $O(n^{1/2})$ time, where n is the number of graph vertices. In [8], Pawagi and Ramakrishnan propose a technique for updating tree paths on parallel random access machines. Their technique is based on representing an MST in the form of an inverted tree. The corresponding parallel algorithms for the edge update problem take $O(\log n)$ time and use $O(n^2)$ processors. In [6], we briefly consider a new technique for updating tree paths and its use to solve the edge update problem on a model of associative parallel systems with vertical data processing (the STAR-machine). The corresponding parallel algorithms for the edge update problem take $O(q \log n)$ time each, where q is the number of vertices whose tree paths change after deleting an edge from the MST. We assume that each elementary operation of the STAR-machine (its microstep) requires one unit of time.

The main goal of this paper is to describe our technique in detail and to justify its correctness. It includes a new associative parallel algorithm for finding an MST along with the matrix of tree paths and a new associative parallel algorithm for updating tree paths after every change in the underlying graph. These algorithms are represented as the corresponding procedures implemented on the STAR-machine. We prove correctness of these procedures and evaluate time complexity. Moreover, we compare the technique of Pawagi and Ramakrishnan with ours and analyze the main advantages.

2. Model of associative parallel machine

We define the model as an abstract STAR-machine of the SIMD type with vertical processing and simple single-bit PEs. To simulate the access data by contents, we use some *typical* operations for associative systems first presented in Staran [2].

The model consists of the following components:

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

The CU broadcasts an instruction to all PEs in unit time. All active PEs execute it simultaneously while inactive PEs do not perform it. Activation of a PE depends on the data.

Input binary data are loaded in the matrix memory in the form of two-dimensional tables, where each data item occupies an individual row and it is updated by a dedicated PE. The rows are numbered from top to bottom and the columns – from left to right. Both a row and a column can be easily accessed.

The associative processing unit is represented as h vertical registers, each consisting of p bits. A vertical register can be regarded as a one-column array that maintains an entire column of a table. Bit columns of tabular data are stored in the registers which perform the necessary bitwise operations.

To simulate data processing in the matrix memory, we use data types **slice** and **word** for the bit column access and the bit row access, respectively, and the type **table** for defining the tabular data. Assume that any variable of the type **slice** consists of p components. For simplicity, let us call "slice" any variable of the type **slice**.

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

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

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

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

FND(Y) returns the ordinal number of the first (the uppermost) $'1'$ of Y;

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

In the usual way, we introduce the predicate SOME(Y) and the bitwise Boolean operations X *and* Y, X *or* Y, *not* Y, and X *xor* Y.

Let T be a variable of the type **table**. We use 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 the matrix T.

Remark. Note that the STAR statements are defined in the same manner as for Pascal. They will be used for presenting our procedures.

We will employ the following two basic procedures implemented on the STAR-machine [4]. They use a global slice X to mark by $'1'$ *positions* of rows which will be processed.

The procedure MATCH(T, X, v, Z) defines in parallel positions of the given matrix T rows 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 MIN(T, X, Z) defines in parallel positions of the given matrix T rows, where minimum elements are located. It returns the slice

Z, where $Z(i) = '1'$ if and only if $\text{ROW}(i, T)$ is the minimum element in T and $X(i) = '1'$.

As shown in [4], the basic procedures run in $O(k)$ time each, where k is the number of columns in T.

3. Finding MST along with tree paths

Let $G = (V, E)$ denote an *undirected graph*, where V is a set of vertices and E is a set of edges. Let w denote a function that assigns a weight to every edge. We assume that $V = \{1, 2, \ldots, n\}$, $|V| = n$, and $|E| = m$.

A *path* from v_1 to v_k in G is a sequence of vertices v_1, v_2, $\ldots$, v_k, where $(v_i, v_{i+1}) \in E$ for $1 \leq i < k$. If $v_1 = v_k$, then the path is called a *cycle*.

A *minimum spanning tree* $T = (V, E')$ is a connected acyclic subgraph of G, where $E' \subseteq E$ and the sum of weights of the corresponding edges is minimum.

Let every edge (u, v) be matched with the triple $\langle u, v, w(u, v)\rangle$. In the STAR-machine matrix memory, a graph is represented as association of matrices *left*, *right*, and *weight*, where every triple $\langle u, v, w(u, v)\rangle$ occupies an individual row, and $u \in$ *left*, $v \in$ *right*, and $w(u, v) \in$ *weight*. We will also use a matrix *code*, whose every i-th row saves the binary representation of vertex v_i. Let us agree to use a slice Y for the matrix *code*, a slice S for the list of triples, and a slice T for the MST.

In [5], we have proposed an associative version of the Prim-Dijkstra algorithm for finding an MST starting at a given vertex v. The corresponding procedure MSTPD returns a slice T, where *positions* of edges belonging to the MST are marked by $'1'$.

Dynamic graph algorithms require, in particular, a fast method for finding a tree path between any pair of vertices. To this end, by means of *minor* changes in the procedure MSTPD, we build an MST along with a matrix M, whose every i-th column saves *positions* of edges belonging to the tree path from vertex v_1 to vertex v_i. The corresponding procedure MSTPaths returns the slice T and the matrix of tree paths M. To define a tree path joining each pair of vertices, we perform the operation *xor* between the corresponding columns of the matrix M.

The procedure MSTPaths runs as follows. Initially, it sets zeros in the first column of M and saves the root v_1 being the first vertex of the fragment T_S. By analogy with MSTPD, at every iteration, it defines both the *position* of the current edge (say, γ) and the corresponding *new* vertex v_k being included in T_S. Moreover, it defines end-point v_l of γ included in T_S *before* this iteration. The tree path from v_1 to v_k is obtained by adding the *position* of γ to the tree path from v_1 to v_l defined before. This path is written in the k-th column of M.

Now, we propose the procedure MSTPaths.

```
procedure MSTPaths(left,right,weight: table; code: table;
                   S: slice(left); var T: slice(left);
                   var M: table);
var i,k,l: integer; S1,N1,N2,X,Z: slice(left);
    F,Y: slice(code); node,node1: word;
Begin CLR(N1); CLR(N2); SET(Y);
  CLR(T); COL(1,M):= N1;
  node:=ROW(1,code);
  S1:=S; Z:=S;
  while SOME(Z) do
    begin MATCH(left,S1,node,X); N1:=N1 or X;
      MATCH(right,S1,node,X); N2:=N2 or X;
      X:=N1 and N2; S1:=S1 and (not X);
/* Positions of edges forming a cycle are deleted from the
   slice S1. */
      Z:=N1 or N2; Z:=Z and S1;
/* Positions of candidates for including into T_S are selected
   by ones in the slice Z. */
      if SOME(Z) then
        begin MIN(weight,Z,X); i:=FND(X);
          T(i):='1'; S1(i):='0';
/* The edge from the i-th position is added to T_S. */
          if N1(i)='1' then
            begin node:=ROW(i,right);
              node1:=ROW(i,left);
            end
          else begin node:=ROW(i,left);
                 node1:=ROW(i,right);
               end;
/* The variable node saves a new vertex. */
          MATCH(code,Y,node,F); k:=FND(F);
          MATCH(code,Y,node1,F); l:=FND(F);
          X:=COL(l,M); X(i):='1';
          COL(k,M):=X;
        end;
    end;
End;
```

Now, we explain the construction allowing one to obtain the current column of the matrix M.

At any iteration of the procedure MSTPaths by means of the slice $N1$ (respectively $N2$), we accumulate *positions* of edges whose left (respectively right) vertex belongs to the fragment T_S. After selecting the *position* of the current new edge γ being included into T_S, we determine whether it belongs to $N1$. If it is true, the right end-point of γ (say, v_k) is the new vertex included in T_S and its left end-point (say, v_l) has been included in T_S before. Otherwise, we determine vertices v_k and v_l using the slice $N2$. Knowing the tree path from v_1 to v_l, we obtain the tree path from vertex v_1 to vertex v_k by adding the position of γ to it.

Correctness of the procedure MSTPaths is proved by induction on the number of tree edges.

It is easy to check that this procedure takes the same time $O(n \log n)$ as the procedure MSTPD for finding an MST in undirected graphs.

Without loss of generality, we will assume that initially an MST is always given along with the matrix of tree paths.

4. Updating tree paths

Let a new MST be obtained from the underlying one by deleting an edge (say, γ) located in the l-th position and inserting an edge (say, δ) located in the k-th position. Let $Y1$ be a connected component of G obtained after deleting γ. The algorithm for updating tree paths will determine *new* tree paths for all vertices from $Y1$.

Let v_{del} and v_{ins} be end-points of the corresponding edges γ and δ that belong to $Y1$. Let P be a slice that saves *positions* of tree edges joining v_{ins} and v_{del}. Obviously, directions of edges on the path $[v_{del} \to v_{ins}]$ will be reversed in the new MST.

Let us agree, for convenience, that a tree path from v_1 to any vertex v_s is denoted by p_s before updating the MST and by p'_s after updating the MST.

The associative parallel algorithm starts at vertex v_{ins}. Note that p'_{ins} (the slice W) is known.

The algorithm carries out the following stages.

At the *first* stage, make a copy of the matrix of tree paths M, namely $M1$. The matrix $M1$ will save tree paths $before$ updating the current MST. Write p'_{ins} in the corresponding column of M. Mark vertex v_{ins} by $'0'$ in the slice $Y1$. Then fulfil the statement $r := ins$.

While P is a non-empty slice, repeat stages 2 and 3.

At the *second* stage, determine vertices not belonging to P that form a subtree of the MST with root v_r if any. For every $v_j \neq v_r$ from this subtree, compute p'_j as follows:

$$p'_j := (p_j \ and \ (not \, p_r)) \ or \ p'_r. \tag{1}$$

Write p'_j in the corresponding column of M. Mark v_j by $'0'$ in the slice $Y1$.

At the *third* stage, select position i of an edge from P incident on vertex v_r. Then define its end-point (say, v_q) being adjacent with v_r. Further, determine the new tree path p'_q and write it in the corresponding column of

M. Now, mark the edge position i by $'0'$ in the slice P and vertex v_q by $'0'$ in the slice $Y1$. Finally, perform the statement $r := q$.

At the *fourth* stage, since P is an empty slice, the vertices marked by $'1'$ in the slice $Y1$ form a subtree of the MST with root v_r determined just now. For every $v_j \neq v_r$ from this subtree, define p'_j using formula (1). Write p'_j in the corresponding column of M. Then mark vertex v_j by $'0'$ in the slice $Y1$.

The algorithm terminates when slices P and $Y1$ become empty.

In [6], we illustrate the run of this algorithm.

On the STAR-machine, it is implemented as procedure TreePaths which uses the following input parameters: matrices *left*, *right*, and *code*, vertices v_{ins} and v_{del}, the number of graph vertices n and the position l of the deleted edge. It returns the matrix M for the new MST and slices W and P.

Initially, the slice W saves the *new* tree path from v_1 to v_{ins}, the slice P saves *positions* of edges from the tree path joining v_{ins} and v_{del}, and the slice $Y1$ saves *vertices* whose tree paths will be recomputed.

We first propose the auxiliary procedure Update. Using formula (1), it recomputes tree paths for any subtree whose vertices are adjacent with root v_r and do not belong to the path from P. In this procedure, vertices of the subtree are marked by $'1'$ in *node*1, the slice W saves p'_r and the slice Z saves p_r.

```
procedure Update(M1: table; W,Z: slice(left); var node1: word;
                 var M: table);
var Z1: slice(left); j: integer;
Begin while SOME(node1) do
  begin j:=STEP(node1);
    Z1:=COL(j,M1);
/* The old path from v_1 to v_j is saved in Z1. */
    Z1:=Z1 and (not Z);
/* We delete the old path from v_1 to v_r from Z1. */
    Z1:=Z1 or W;
/* The new path from v_1 to v_j is saved in Z1. */
    COL(j,M):=Z1;
  end;
End;
```

Before presenting the procedure TreePaths, we explain how to determine a subtree whose vertices are adjacent with root v_r and do not belong to the tree path from P. To this end, we first determine the position i of an edge from P incident on v_r. Then all vertices reachable from v_r will be marked by $'1'$ in the i-th row of the matrix $M1$. Among them, we have to exclude the vertices being updated before.

Now, we propose the procedure TreePaths.

```
procedure TreePaths(left,right: table; code: table;
                    l,n,ins,del: integer; var M: table;
                    var P,W: slice(left));
/* New tree paths for vertices from the connected component Y1
   will be written in the matrix M. */
var M1: table; N1,N2,X,Z: slice(left); A,B: slice(code);
    current,node1,prev: word(M); node: word(code);
    i,q,r: integer;
/* Initialization. */
Begin CLR(prev); SET(A);
/* The first stage. */
  TCOPY(M,n,M1); Z:=COL(ins,M1);
  COL(ins,M):=W;
/* A new path from v_1 to v_ins is written in the corresponding
   column of M. */
  r:=ins; node:=ROW(r,code);
/* The second stage. */
  while SOME(P) do
    begin MATCH(left,P,node,N1);
      MATCH(right,P,node,N2);
      X:=N1 or N2; i:=FND(X);
/* We define the position i of an edge from P incident on v_r. */
      node1:=ROW(i,M1);
/* Vertices whose tree paths include the edge from the i-th position
   are marked by '1' in node1. */
      current:=node1;
      node1:=node1 and (not prev);
      prev:=current;
/* By means of prev, we save the updated vertices. */
      node1(r):='0';
/* Here, v_r is a subtree root. */
      if SOME(node1) then Update(M1,W,Z,node1,M);
/* The third stage. */
      if N1(i)='1' then node:=ROW(i,right)
      else node:=ROW(i,left);
/* The binary code of a new subtree root is saved in node. */
      MATCH(code,A,node,B);
      q:=FND(B);
/* Here, v_q is a new subtree root. */
      W(i):='1'; COL(q,M):=W;
```

```
   /* A new tree path from v1 to vq is written in the corresponding
      column of M. */
20.       Z:=COL(q,M1); P(i):='0';
21.       r:=q;
22.     end;
   /* The fourth stage. */
23.   node1:=ROW(l,M1);
24.   node1:=node1 and (not prev);
25.   node1(r):='0';
26.   if SOME(node1) then Update(M1,W,Z,node1,M);
27. End;
```

Correctness of this procedure is established by means of the following theorem.

Theorem. *Let an undirected graph G with n vertices be given as association of matrices left and right. Let a matrix code save binary representations of vertices. Let an edge from the l-th position be deleted from the minimum spanning tree T. Let del be end-point of the deleted edge and ins be end-point of the inserted edge that belong to the connected component $Y1$. Then the procedure TreePaths returns the updated matrix M and the slices P and W.*

Proof. We prove this by induction on the number of edges k belonging to the slice P.

Basis is checked for $k = 1$. On performing lines 1–4, the variable *prev* consists of zeros, matrix $M1$ is a copy of M, the slice Z saves p_{ins} and p'_{ins} is written in the corresponding column of M, the current vertex v_r coincides with v_{ins}, and its binary code is saved by means of the variable *node*.

Further, on fulfilling lines 6–9, we first determine the position i of an edge from P incident on v_r. Then, we determine vertices whose tree paths include this edge and mark them by $'1'$ in the variable $node1$. On performing lines 10–12, we first save the current value of $node1$ and then vertices being updated before this step along with root v_r are deleted from $node1$. Hence, after performing line 13, $node1$ saves a subtree whose vertices are adjacent with v_r and do not belong to the tree path from P. If $node1$ is nonempty, we determine new tree paths for all vertices from this subtree using the auxiliary procedure Update (line 14). Further, we execute the next stage.

At the third stage, on performing lines 15-18, the variable *node* saves the binary code of a new subtree root v_q. Then on fulfilling line 19, we determine a new tree path to v_q and write it in the q-th column of M. After that on fulfilling lines 20–21, we save p_q in the slice Z, delete the edge position i from the slice P, and perform the statement $r := q$.

Since P is an empty slice, we perform the fourth stage. Here, on fulfilling lines 23–25, we first save the connected component $Y1$ by means of

$node1$. After that, vertices updated before this step and root v_q are deleted from $node1$. If $node1$ becomes empty, we jump to end of this procedure. Otherwise, we determine new tree paths for all vertices of the subtree with root v_q using the auxiliary procedure Update. Since $node1$ becomes empty after performing Update, we go to the end.

Step of induction. Let the assertion be true for $k \geq 1$. We will prove this for $k+1$.

Let the slice P save positions of $k+1$ edges from the tree path $[v_{del} \rightarrow v_{ins}]$. Let the edge (v_t, v_{del}) belong to this path. Then we represent the tree path $[v_{del} \rightarrow v_{ins}]$ as $(v_{del}, v_t)[v_t \rightarrow v_{ins}]$, where the path $[v_t \rightarrow v_{ins}]$ consists of k edges. By inductive assumption, after updating the tree path $[v_t \rightarrow v_{ins}]$, the new tree paths for vertices from subtrees rooted at vertices $v_{ins}, \ldots, v_t$ from P are written in the corresponding columns of M, the variable $prev$ saves the subtree rooted at v_t, the variable q saves vertex v_t, the slice Z saves p_q while W saves p'_q, and the slice P saves position of the edge (v_t, v_{del}). Since P is nonempty, we perform the $(k+1)$-th iteration.

In the same manner as in the basis, we first determine position i of the edge (v_t, v_{del}) incident on v_t. Then by means of $node1$, we save the subtree rooted at vertex v_t if any. After that, we determine new tree paths for all vertices from this subtree and write them in the corresponding columns of the matrix M.

At the third stage, we first determine a new root v_{del}. Then we define p'_{del} and write it in the corresponding column of the matrix M. After that, the edge position is deleted from P. Therefore it becomes empty.

At the fourth stage, we determine new tree paths for all vertices of the subtree rooted at v_{del} if any and write them in the corresponding columns of M.

Hence, after executing the procedure TreePaths, the new tree paths for all vertices of the connected component $Y1$ are written in the corresponding columns of M. $\square$

It is easy to check that the procedure TreePaths takes $O(h \log n)$ time, where h is the number of vertices in the connected component $Y1$.

5. The use of inverted trees for updating tree paths

As shown in [8], dynamic graph algorithms require fast computations of the following tree properties: finding a tree path that joins each pair of vertices; finding subtrees obtained after deleting an edge from the tree; choosing the maximum weight edge lying on such a path.

On the STAR-machine, the first two properties are satisfied by means of the matrix of tree paths M and the third property is satisfied by means of the basic procedure MAX.

In [8], Pawagi and Ramakrishnan propose a technique for updating tree paths on parallel random access machines. To describe it, we need the following definitions from [8].

Let r be the root of a tree. A vertex u is called an *ancestor* of vertex v if u is on the path from vertex v to the root r. A *father* of a vertex is its immediate ancestor. An *inverted tree* is a rooted tree, where every vertex points to its father.

Let $[u - v]$ denote an undirected path from vertex u to vertex v. The *distance* from vertex v to the root r is the number of edges in $[v - r]$.

Now, we briefly describe this technique which uses representing an MST as an inverted tree.

As a model of computation, a CREW PRAM machine is used. Initially, by means of the method of Tsin and Chin [11], a given MST is transformed into an inverted tree. To update tree paths, the matrices F^+, D^+, and M^+ are employed.

The inverted tree is represented as the matrix F^+ that saves the *paths* from all vertices to the root. Each element $F^+[i,k]$ $(1 \leq i \leq n,\ 0 \leq k < n)$ saves the k-th ancestor of vertex v_i.

After computing the matrix F^+, a one-dimensional array D^+ is determined, where every i-th row saves the *distance* from vertex v_i to root v_r. After that, each row of F^+ is shifted right so that all vertices v_r except the leftmost one are eliminated. Therefore the rightmost column of the matrix F^+ contains root v_r. Knowing the matrix F^+, one can determine a tree path joining each pair of vertices by locating their leftmost common vertex in the corresponding rows of F^+.

To compute the maximum weight edge on the tree path joining every pair of vertices, at first, a matrix E^+ is determined, where each element $E^+[i,k]$ saves the maximum weight edge on the tree path from v_i to its k-th ancestor. Further, by means of the algorithm from [8], the maximum weight edge on the tree path joining every pair of vertices is determined using matrices E^+ and F^+. Then for all pairs (v_i, v_j) the maximum weight edge on the tree path joining these vertices is stored in a matrix M^+. As shown in [8], the matrices F^+, D^+, and M^+ are computed in $O(\log n)$ time using $O(n^2)$ processors each.

Let us compare two techniques for updating tree paths.

On the STAR-machine, a graph is represented as a list of triples, while on the CREW PRAM machine, it is given as an adjacency matrix.

On the STAR-machine, the procedure MSTPaths returns an MST *along with* the matrix of tree paths M, whose every i-th *column* saves the tree path from v_1 to v_i. On the CREW PRAM machine, an MST is given as an inverted tree. Knowing the inverted tree, a matrix F^+ is computed. Its every i-th *row* saves the tree path from v_1 to v_i.

On the STAR-machine, a tree path between any pair of vertices is obtained by using the bitwise Boolean operation *xor* between the corresponding columns of the matrix M. On the CREW PRAM machine, a tree path between any pair of vertices is obtained after performing a binary search on the corresponding rows of the matrix F^+ to locate their leftmost common vertex.

On the STAR-machine, the maximum weight edge on the tree path joining each pair of vertices (v_i, v_j) is determined by means of the basic procedure MAX. On the CREW PRAM machine, the corresponding maximum weight edge has been written in $M^+[i,j]$. To compute M^+, the matrices E^+ and F^+ are used.

Finally, we consider how to determine two subtrees after deleting an edge (v_i, v_j) from the MST.

On the STAR-machine, we first select the *position* of the row in the matrix of tree paths M, where the edge γ is located. Vertices which are marked by $'1'$ in this row constitute a separate subtree of the MST because they are not reachable from v_1 after deleting the edge γ. On the CREW PRAM machine, we first set $F^1(v_i) = v_i$ to delete the edge (v_i, v_j) from the inverted tree. Then we obtain two subtrees, one rooted at v_r and the other at v_i. Further, we compute the matrix F^+. The vertices in each subtree are determined by the corresponding roots in the rightmost column of F^+.

6. Conclusions

In this paper, we have described in detail a new technique for updating tree paths on the STAR-machine being a model of associative parallel systems with vertical processing. We proposed the corresponding procedures, proved their correctness and evaluated time complexity. Moreover, we have compared two techniques for updating tree paths on the STAR-machine and the CREW PRAM machine. From this comparison, we can conclude that the use of associative processors for dynamic edge update of an MST allows one to design simple and natural algorithms.

To improve time complexity, we intend to employ associative systems with bit-parallel processing for solving dynamic graph algorithms.

References

[1] Eppstein D., Galil Z., Italiano G.F., Nissenzweig A. Sparsification – A technique for speeding up dynamic graph algorithms // J. of the ACM. – 1997. – Vol. 44, No. 5. – P. 669–696.

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

[3] Frederickson G.N. Data structures for on-line updating of minimum spanning trees, with applications // SIAM J. Comput. – 1985. – Vol. 14. – P. 781–798.

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

[5] Nepomniaschaya A.S. Comparison of performing the Prim-Dijkstra algorithm and the Kruskal algorithm on associative parallel processors // Cybernetics and System Analysis. – Kiev: Naukova Dumka, 2000. – No. 2. – P. 19–27 (in Russian. English translation by Plenum Press).

[6] Nepomniaschaya A.S. Associative parallel algorithms for dynamic edge update of minimum spanning trees // Proc. 7th Int. Conf. PaCT, 2003 / Lect. Notes in Comp. Sci. – Berlin: Springer–Verlag, 2003. – Vol. 2763. – P. 141–150.

[7] Parhami B. Search and data slection algorithms for associative processors // Associative Processing and Processors / Ed.: A. Krikelis, C.C. Weems. – Los Alamitos, California: IEEE Computer Society, 1997. – P. 10–25.

[8] Pawagi S., Ramakrishnan I.V.: An $O(\log n)$ algorithm for parallel update of minimum spanning trees // Inform. Process. Letters. – 1986. – Vol. 22. – P. 223–229.

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

[10] Tarjan R.E. Applications of path compression on balanced trees // J. of the ACM. – 1979. – Vol. 26, No. 4. – P. 690–715.

[11] Tsin Y., Chin F. Efficient parallel algorithms for a class of graph-theoretic problems // SIAM J. Comput. – 1984. – Vol. 14. – P. 580–599.