Bull. Nov. Comp. Center, Comp. Science, 30 (2010), 53–69
© 2010 NCC Publisher

Parallel implementation of the Ramalingam incremental algorithm for dynamic updating the shortest-paths subgraph

A.S. Nepomniaschaya

Abstract. The paper proposes an efficient implementation of the Ramalingam algorithm for dynamic updating the single-sink shortest-paths subgraph of a directed weighted graph after insertion of an edge using a model of associative (content addressable) parallel systems with vertical processing (the STAR-machine). An associative version of the Ramalingam incremental algorithm is given as the procedure `InsertArc`, whose correctness is proved and the time complexity is evaluated. We compare implementations of the Ramalingam incremental algorithm and its associative version and present the main advantages of the associative version.

1. Introduction

In many applications, graphs are subject to discrete changes, such as insertions and deletions of edges or vertices. The objective of a dynamic algorithm is to efficiently update the solution to a problem after dynamic changes rather than to recompute the entire graph from scratch each time.

The dynamic version of the single source shortest paths problem consists in updating the shortest paths information after any change in a graph. The most general types of updating operations for the single source shortest paths problem include insertions and deletions of edges and updating operations on the edge weights. An algorithm is called *fully dynamic* if arbitrary sequences of the above operations are allowed, and it is called *partially* (*semi-*) *dynamic* if only one type of the updating is allowed. A partially dynamic algorithm is called *incremental* if it supports only insertions of edges, while it is called *decremental* if it supports only deletions of edges.

In the case of arbitrary real edge weights, Ramalingam and Reps [12, 13] devise fully dynamic algorithms for updating the single source shortest paths using the output bounded model. In this model, the run time of an algorithm is analyzed in terms of the output change rather than the input size. The authors assume that the graph has no negative-length cycles before and after the input updating. Frigioni et al. [3] study the semi-dynamic single source shortest paths problem for both directed and undirected graphs with positive real edge weights in terms of the output complexity. Frigioni et al. [4] propose fully dynamic algorithms for updating distances and a single source shortest paths tree (*sp-tree*) in either a directed or an undirected graph with positive real edge weights under arbitrary sequences of the edge updates. The cost of the update operations is given as a function of the number of output updates by using the notion of k-bounded accounting function. Frigioni et al. [5] propose the fully dynamic solution for the problem of updating the shortest paths from a given source in a directed graph with arbitrary edge weights. The authors devise a new algorithm for performing edge deletions and weight increases that explicitly deals with zero–length cycles. Algorithms from [3–5, 12, 13] use a dynamic version of the Dijkstra algorithm [1]. Narváez et al. [6] propose two incremental methods to transform the well-known static algorithms by Dijkstra, Bellman–Ford, and D'Esopo–Pape to the new dynamic algorithms for updating an sp-tree after changing edge weights. In [10], we propose an efficient parallel implementation of the Ramalingam decremental algorithm [12] for the dynamic updating of the shortest-paths subgraph SP(G) that consists of all the shortest paths from every vertex of a given directed graph G to the sink. Our computation model (the STAR-machine) simulates the run of associative (content addressable) parallel systems of the SIMD type with bit-serial (vertical) processing. The associative version of this algorithm is given as the procedure `DeleteArc` whose correctness is proved. We obtain that this procedure takes the time proportional to the number of vertices, whose shortest paths to the sink change after deleting an edge from SP(G). Following [2], it is assumed that each elementary operation of the STAR-machine (its microstep) takes one unit of time.

In this paper, we first propose an extension of the language STAR. Then, using the data structure first proposed in [10], we construct an associative version of the Ramalingam incremental algorithm [12] for the dynamic updating of the shortest-paths subgraph SP(G). The associative version of this algorithm is given as the procedure `InsertArc`, whose correctness is proved. We obtain that this procedure takes $O(hk)$ time, where h is the number of bits required for coding a maximal weight of the shortest paths to the sink in SP(G) and k is the number of vertices, whose shortest paths to the sink change after inserting an edge into the graph G. We also discuss the main advantages of the associative version of the Ramalingam incremental algorithm.

2. An associative parallel machine model

Here, we propose a brief description of our model. It is defined as an abstract STAR-machine of the SIMD type with the vertical data processing [7]. It consists of the following components (Figure 1):

- a sequential control unit (CU), where programs and scalar constants are stored;

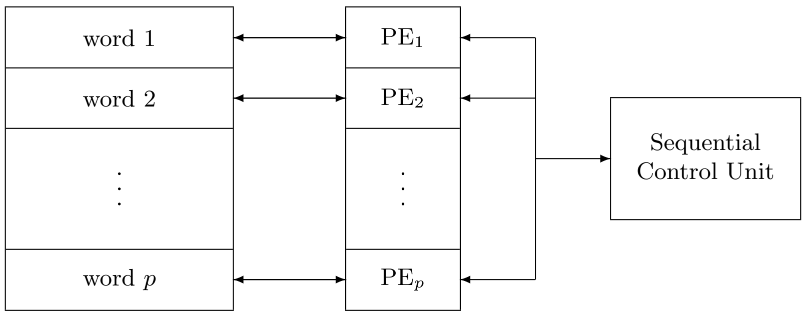

Figure 1. The STAR-machine

- an associative processing unit consisting of p single–bit processing elements (PEs);
- a matrix memory for the associative processing unit.

The CU passes an instruction to all PEs in one unit of time. All active PEs execute it in parallel, while inactive PEs do not. Activation of a PE depends on data.

The input binary data are given in the form of 2D tables, where each datum occupies an individual row and is updated by a dedicated PE (Figure 2). In any table, rows are numbered from top to bottom and columns — from left to right. Some tables may be loaded into the memory.

An associative processing unit is represented as h vertical registers, each consisting of p bits (Figure 3). Vertical registers can be regarded as a one-column array. The bit columns of the tabular data are stored in the registers that perform the necessary Boolean operations.

The run is described by means of the language STAR being an extension of Pascal. Let us briefly consider the STAR constructions needed here. To simulate data processing in the matrix memory, we use the data types **word,**

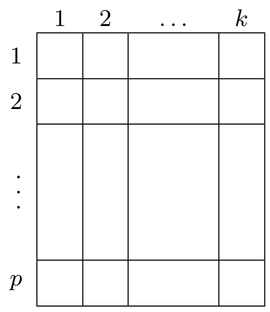

Figure 2. Data array

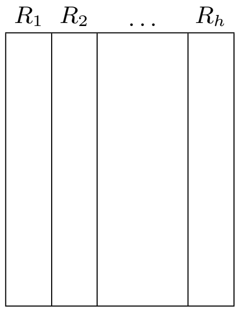

Figure 3. Associative processing unit

slice, and **table**. Constants for the types **slice** and **word** are represented as a sequence of symbols of a set $\{0, 1\}$ enclosed within single quotation marks. The types **slice** and **word** are used for the bit column access and the bit row access, respectively, and the type **table** is used for defining the tabular data. Assume that any variable of the type **slice** consists of p components, which belong to $\{0, 1\}$. For simplicity, let us call *slice* any variable of the type **slice**.

Let us present some elementary operations and a predicate 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)	simultaneously sets all components of Y to $'1'$;
CLR(Y)	simultaneously sets all components of Y to $'0'$;
$Y(i)$	selects the ith component of Y;
FND(Y)	returns the number i of the first (the uppermost) $'1'$ of Y, $i \geq 0$;
STEP(Y)	returns the same result as FND(Y), then resets the first $'1'$ found to $'0'$;
CONVERT(Y)	returns a row, whose every ith bit coincides with $Y(i)$. It is applied when a row of one matrix is used as a slice for another matrix.

The operations FND(Y), STEP(Y), and CONVERT(Y) are used only as the right part of the assignment statement, while the operation $Y(i)$ is used as both the right part and the left part of the assignment statement.

To carry out the data parallelism, we introduce in the usual way the bitwise Boolean operations: $X\ and\ Y$, $X\ or\ Y$, $not\ Y$, $X\ xor\ Y$. We also use a predicate SOME(Y) that results in **true** if there is at least a single bit $'1'$ in the slice Y.[1]

Note that the predicate SOME(Y) and all operations for the type **slice** are also performed for the type **word**.

The extension of the language STAR concerns variables of the type **word**. We will employ the new operation REP(i, j, v, w) that replaces the substring $w(i)w(i+1)\dots w(j)$ of the string w with the string v, where $|v| = j - i + 1$ and $1 \leq i < j < |w|$.

For the two variables v and w of the type **word** having the same length, we will use the new operation ADD(v, w) that performs the addition of the binary strings v and w. Its result is the arithmetical sum of v and w. Note that this operation can be used only in the right part of the assignment statement.

[1]For simplicity, the notation $Y \neq \emptyset$ denotes that the predicate SOME(Y) results in **true**.

We will also utilize the following predicates for the two variables v and w of the type **word** first proposed in [11] by Potter: EQ(v, w), NOTEQ(v, w), LESS(v, w), LESSEQ(v, w), GREAT(v, w), and GREATEQ(v, w).

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

ROW(i, T)	returns the ith row of the matrix T;
COL(i, T)	returns its ith column.

Note that the STAR statements are defined in the same manner as for Pascal.

Now, we recall a group of basic procedures [8, 9] implemented on the STAR-machine which will be used later on. These procedures use the given slice X to indicate with bit $'1'$ the row positions used in the corresponding procedure.

The procedure MATCH(T, X, w, Z) determines positions of the matrix T rows that coincide with a given pattern w. It returns the slice Z, where $Z(i) = '1'$ if and only if ROW$(i, T) = w$ and $X(i) = '1'$.

The procedure MIN(T, X, Z) defines positions of rows in the given matrix T where a minimal entry is located. These positions are marked with $'1'$ in the resulting slice Z.

The procedure SETMIN(T, F, X, Z) defines positions of the matrix T rows that are less than the corresponding rows of the matrix F. It returns the slice Z, where $Z(i) = '1'$ if ROW$(i, T) <$ ROW(i, F) and $X(i) = '1'$.

The procedure TCOPY1(T, j, h, F) writes down h columns from the given matrix T, starting with the $(1 + (j - 1)h)$th column, into the resulting matrix F $(j \geq 1)$.

The procedure HIT(T, F, X, Z) defines the positions of the corresponding identical rows of the given matrices T and F. It returns a slice Z, where $Z(i) = '1'$ if and only if ROW$(i, T) =$ ROW(i, F) and $X(i) = '1'$.

The procedure TMERGE(T, X, F) writes down the rows of the matrix T, that correspond to positions $'1'$ in the slice X, into the matrix F. Other rows of the matrix F are not changed.

The procedure CLEAR(h, T) writes down zeros into every ith column of the given matrix T, where $1 \leq i \leq h$.

The procedure ADDV(T, F, X, R) writes into the matrix R a result of the parallel addition of the corresponding rows of matrices T and F, whose positions are selected with $'1'$ in the slice X. This algorithm uses Table 5.1 from [2].

The procedure ADDC(T, X, v, F) adds the binary word v to the rows of the matrix T selected with $'1'$ in X, and writes down the result into the corresponding rows of the matrix F. Other rows of the matrix F are set to zero.

The procedure SUBTC(T, X, v, F) subtracts the binary word v from the rows of the matrix T selected with bits $'1'$ in the slice X, and writes the result into the corresponding rows of the matrix F. Other rows of the matrix F are set to zero.

In [8, 9], we have shown that each basic procedure takes $O(r)$ time, where r is the number of bit columns in the corresponding matrix.

3. Preliminaries

Let $G = (V, E, w)$ be a *directed weighted graph* with a set of vertices $V = \{1, 2, \ldots, n\}$, a set of directed edges (arcs) $E \subseteq V \times V$ and the function w that assigns a weight to every edge. We will consider graphs with a distinguished vertex s called *sink*.

An *adjacency matrix* $\mathrm{Adj} = [a_{ij}]$ of a directed graph G is an $n \times n$ Boolean matrix, where $a_{ij} = 1$ if and only if there is an arc from the vertex i to the vertex j in the set E.

An arc e directed from i to j is denoted by $e = (i, j)$, where the vertex i is the *head* of e (or *father*) and the vertex j is its *tail* (or *son*). We assume that all arcs have a positive weight and $w(u, v) = \infty$ if $(u, v) \notin E$.

The infinity will be implemented with the value $\sum_{i=1}^{n} c_i$, where c_i is a maximal weight of arcs outgoing from the vertex i. Let h be the number of bits for coding this sum.

A *path* from u to s in G is a finite sequence of vertices $u = v_1, v_2, \ldots, v_k = s$, where $(v_i, v_{i+1}) \in E$ for $i = 1, 2, \ldots, k-1$ and $k > 1$. The *shortest path* from u to s is the path of the minimal sum of weights of its arcs.

Let $\mathrm{dist}(u)$ denote the length of the shortest path from u to s and $\mathrm{SP}(G)$ denote a subgraph of the shortest paths from all the vertices of G to the sink.

Like Ramalingam, we introduce the following notations.

We denote by $\mathrm{Outdegree}_{\mathrm{SP}}(v)$ the number of arcs outgoing from the vertex v into $\mathrm{SP}(G)$.

Let $\mathrm{Succ}(u) = \{z : u \to z \in E\}$ and $\mathrm{Pred}(u) = \{x : x \to u \in E\}$.

Let the arc (i, j) be inserted into a subgraph of the shortest paths $\mathrm{SP}(G)$. A vertex u is called *affected* in $\mathrm{SP}(G)$ if and only if $\mathrm{dist}(u, i) + w(i, j) + \mathrm{dist}_{\mathrm{old}}(j) < \mathrm{dist}_{\mathrm{old}}(u)$, where $\mathrm{dist}(u, i)$ is the length of the shortest path from the vertex u to the vertex i in the new graph, $w(i, j)$ is the weight of the arc (i, j) and $\mathrm{dist}_{\mathrm{old}}(j)$ ($\mathrm{dist}_{\mathrm{old}}(u)$, respectively) is the length of the shortest path from the vertex j (u, respectively) to the sink before inserting the edge (i, j).

4. The Ramalingam incremental algorithm for the single-sink shortest paths problem

The Ramalingam algorithm for the dynamic updating of the shortest-paths subgraph after inserting an arc into the graph G uses a heap `PriorityQueue`, whose elements are affected vertices with a key. For any affected vertex u, its key is equal to the distance from u to the vertex i. Initially, `PriorityQueue` $= \emptyset$.

The algorithm runs as follows.

First, the arc (i, j) is inserted into G. If $w(i, j) + \text{dist}(j) = \text{dist}(i)$, then the arc (i, j) is inserted into $\text{SP}(G)$ and $\text{Outdegree}_{\text{SP}}(i)$ is increased by one.

If $w(i, j) + \text{dist}(j) < \text{dist}(i)$, then $\text{dist}(i) := w(i, j) + \text{dist}(j)$ and the vertex i is inserted into the heap `PriorityQueue` with the key that is equal to zero.

The affected vertices are updated as follows.

At each iteration, an affected vertex (say u) with the minimal key is deleted from the current heap. Then all the arcs outgoing from the vertex u are deleted from $\text{SP}(G)$ and $\text{Outdegree}_{\text{SP}}(u) := 0$.

For every vertex $x \in \text{Succ}(u)$, we check whether $w(u, x) + \text{dist}(x) = \text{dist}(u)$. If this is true, the arc (u, x) is inserted into $\text{SP}(G)$ and $\text{Outdegree}_{\text{SP}}(u)$ is increased by one.

For every vertex $y \in \text{Pred}(u)$, we check whether $w(y, u) + \text{dist}(u) = \text{dist}(y)$. If it is true, the arc (y, u) is inserted into $\text{SP}(G)$ and $\text{Outdegree}_{\text{SP}}(y)$ is increased by one. Otherwise, we check whether $w(y, u) + \text{dist}(u) < \text{dist}(y)$. If this is true, then $\text{dist}(y) := w(y, u) + \text{dist}(u)$, and the vertex y is inserted into the heap `PriorityQueue` with the key that is equal to the distance from the vertex y to the vertex i. If $y \in$ `PriorityQueue`, then its previous key is replaced with a smaller value.

We observe that the Ramalingam incremental algorithm builds a tree with the root i consisting of affected vertices and every edge is directed from the son to the father. In addition, in this tree, there is a single path from every affected vertex to the vertex i. Initially, the tree consists of a single vertex i. At every iteration, a new affected vertex l is inserted into the current tree if and only if $\text{dist}(l) - \text{dist}(i)$ has the minimal value.

Let T_i denote a tree that is built in the process of updating the shortest-paths subgraph after insertion of the arc (i, j) into the graph G.

5. An associative version of the Ramalingam incremental algorithm

To design an associative version of the Ramalingam incremental algorithm, we employ the following data structure first proposed in [10]:

- an $n \times n$ adjacency matrix G whose every ith column saves with $'1'$ the tails of arcs outgoing from the vertex i;
- an $n \times n$ adjacency matrix SP whose every ith column saves with $'1'$ the tails of arcs outgoing from the vertex i that belong to the shortest-paths subgraph;

- an $n \times hn$ matrix `Weight` that contains the arc weights as entries. It consists of n fields having h bits each. The weight of an arc (i, j) is written down in the jth row of the ith field;
- an $n \times hn$ matrix `Cost` that contains the arc weights as entries. It consists of n fields having h bits each. The weight of an arc (i, j) is written down in the i-th row of the j-th field;
- an $n \times h$ matrix `Dist`, whose every ith row saves the shortest distance from the vertex i to the sink s.

We observe that the ith field of the matrix `Weight` saves the weights of arcs *outgoing* from the vertex i, while the ith field of the matrix `Cost` saves the weights of arcs *entering* the vertex i.

We will use the following property of the matrices G and SP:

In every ith row of the matrices G and SP, the heads of arcs entering the vertex i are marked with $'1'$.

Algorithm A (associative parallel algorithm for updating the arcs outgoing from an affected vertex k):

1. With a slice (say Z), save positions of arcs outgoing from the vertex k in G.
2. Compute *in parallel* the distances from the vertex k to s for every path in the matrix G that includes the arc marked with $'1'$ in the slice Z.
3. With a slice (say Y), save positions of those arcs (k, l), for which $\text{dist}(k) = w(k, l) + \text{dist}(l)$.
4. Include the positions of arcs marked with bits $'1'$ in the slice Y into the matrix SP.

On the STAR-machine, this algorithm is implemented as the procedure `UpdateOutgoingArcs`.

Algorithm B (associative parallel algorithm for updating the arcs entering an affected vertex k):

1. With a slice (say Z), save the heads of arcs entering the vertex k in the graph G.
2. For all the vertices l marked with bits $'1'$ in the slice Z, compute *in parallel* the distances from l to the sink s in every path in the matrix SP that starts from the arc (l, k).
3. Include into the matrix SP the positions of arcs (r, k) for which $\text{dist}(r) = w(r, k) + \text{dist}(k)$.

4. With a slice (say Y), save the positions of those vertices r marked with bits $'1'$ in the slice Z for which $\mathrm{dist}_{\mathrm{new}}(r) < dist_{\mathrm{old}}(r)$. After that, write $\mathrm{dist}_{\mathrm{new}}(r)$ in the corresponding rows of the matrix `Dist`.

On the STAR-machine, this algorithm is implemented as the procedure `UpdateIncomingArcs`.

Now let us proceed to an associative parallel algorithm for the dynamic updating of the shortest-paths subgraph after insertion of a new arc (i, j) with the weight v_0 into the graph G. Let the variable v_2 save the shortest distance from the vertex i to the sink in the source matrix SP. The algorithm makes use of a slice Z to save positions of the current affected vertices and an $n \times h$ matrix `Queue`, whose every lth row saves the distance from the vertex l to the vertex i:

1. Insert the position of the arc (i, j) into the matrix G and insert its weight v_0 into the matrices `Weight` and `Cost`.
2. Determine the distance (say v_3) from i to the sink s of the path that starts from the arc (i, j).
3. If $v_2 = v_3$, insert the position of the arc (i, j) into SP and go to exit.
4. If $v_3 < v_2$, write down the value v_3 in the ith row of the matrix `Dist`, then assign the maximal priority in the matrix *Queue* for the vertex i and finally save the vertex i in a slice (say Z_1).
5. While $Z_1 \neq \emptyset$, update the affected vertices with allowance for their new distances to the sink as follows:
    - select the vertex (say k) with the maximal priority in the matrix `Queue` and remove it from the slice Z_1;
    - delete in parallel the arcs outgoing from the vertex k in the matrix SP;
    - with Algorithm A, determine in parallel the positions of those arcs (k, l) for which $\mathrm{dist}(k) = w(k, l) + \mathrm{dist}(l)$ and insert them into the matrix SP;
    - with Algorithm B, determine in parallel the positions of those arcs (r, k), for which $\mathrm{dist}(r) = w(r, k) + \mathrm{dist}(k)$ and insert them into the matrix SP. Then determine in parallel the positions of the new affected vertices l and write down $\mathrm{dist}_{\mathrm{new}}(l)$ in the corresponding rows of the matrix `Dist`;
    - for the new affected vertices l determine in parallel the values $\mathrm{dist}(l) - \mathrm{dist}(i)$ and write them down in the corresponding rows of the matrix `Queue`;
    - include the new affected vertices into the slice Z_1.

On the STAR-machine, this algorithm is implemented as the procedure `InsertArc`.

6. Representation of the associative version of the Ramalingam incremental algorithm on the STAR-machine

Here, we first provide two auxiliary procedures for implementing Algorithms A and B. Then we propose the main procedure for the dynamic updating of the shortest-paths subgraph after insertion of a new arc into the source graph.

Let us consider the procedure `UpdateOutgoingArcs`. Knowing the current updated vertex k, the number of bits h for coding the infinity, and the current matrices G, SP, `Weight`, and `Dist`, the procedure returns the updated matrix SP.

```
procedure UpdateOutgoingArcs(h,k: integer; G: table;
          Weight: table; Dist: table; var SP: table);
/* Here $h$ is the number of bits for coding the infinity and $k$ is
   an updated vertex. */
var W1,W2: table; v: word(Dist); Y,Z: slice(G);
1. Begin Z:=COL(k,G);
2.   TCOPY1(Weight,h,k,W1);
/* The matrix $W_2$ saves different distances from the vertex $k$ to the sink. */
3.   ADDV(W1,Dist,Z,W2);
/* The row $v$ saves a distance from the vertex $k$ to $s$. */
4.   v:=ROW(k,Dist);
/* In the slice $Y$, we save those vertices $l$ of $G$ for which
   $\mathrm{dist}(k) = w(k,l) + \mathrm{dist}(l)$. */
5.   MATCH(W2,Z,v,Y);
/* Positions of arcs $(k,l)$ are inserted into $SP$. */
6.   COL(k,SP):=Y;
7. End;
```

Claim 1. *Let h be the number of bits for coding the infinity and k be the current updated vertex. Let the current matrices G, `Weight`, `Dist`, and SP be given. Then after performing the procedure `UpdateOutgoingArcs`, the positions of arcs (k,l) for which* $\mathrm{dist}(k) = w(k,l) + \mathrm{dist}(l)$ *are included into the matrix SP.*

This claim is proved by contradiction. Let an arc (k,r) belong to G and $\mathrm{dist}(k) = w(k,r) + \mathrm{dist}(r)$. However, after performing the procedure `UpdateOutgoingArcs`, the arc (k,r) does not belong to SP. We prove that it contradicts the execution of this procedure.

Now, we propose the procedure `UpdateIncomingArcs`. Knowing the current updated vertex k, the number of bits h for coding the infinity, and the current matrices G, SP, `Cost`, and `Dist`, the procedure returns the slice Y and the updated matrices `Dist` and SP.

```
procedure UpdateIncomingArcs(h,k: integer; G: table;
          Cost: table; var Y: slice(G); var Dist: table;
          var SP: table);
var X, Z: slice(G); v: word(G); v1: word(Dist); W, W1: table;
1.  Begin v:=ROW(k,G);
    /* The slice $Z$ saves the heads of the arcs entering $k$. */
2.    Z:=CONVERT(v);
    /* The row $v_1$ saves a distance from the vertex $k$ to $s$. */
3.    v1:=ROW(k,Dist);
    /* The matrix $W_1$ saves the $k$th field of the matrix Cost. */
4.    TCOPY1(Cost,k,h,W1);
    /* Every $l$th row of the matrix $W$ marked with $'1'$ in $Z$ saves a new distance
       from $l$ to $s$. */
5.    ADDC(W1,Z,v1,W);
    /* The slice $X$ saves the positions of the matrix Dist rows where
       $\mathrm{dist}(l) = w(l,k) + \mathrm{dist}(k)$. */
6.    HIT(W,Dist,Z,X);
    /* Positions of the arcs $(r,k)$ for which $\mathrm{dist}(r) = w(r,k) + \mathrm{dist}(k)$
       are included into the matrix $SP$. */
7.    v:=CONVERT(X);
8.    ROW(k,SP):=v;
    /* With the slice $Y$, we save the positions of arcs whose new distances
       to $s$ are decreased. */
9.    SETMIN(W,Dist,Z,Y);
    /* We write the new value $\mathrm{dist}(l)$ in the $l$th row of the matrix Dist
       if and only if $Y(l) = '1'$. */
10.   TMERGE(W,Y,Dist);
11. End;
```

Claim 2. *Let h be the number of bits for coding the infinity and k be a current updated vertex. Let the current matrices G, SP, `Cost`, and `Dist` be given. Then the procedure `UpdateIncomingArcs` returns the slice Y and the updated matrices `Dist` and SP. In addition, the positions of the arcs (l,k), for which $\mathrm{dist}(l) = w(l,k) + \mathrm{dist}(k)$, are included into the matrix SP, the slice Y saves the heads of the arcs (r,k), for which $\mathrm{dist}_{\mathrm{new}}(r) < \mathrm{dist}_{\mathrm{old}}(r)$, and the new distances $\mathrm{dist}_{\mathrm{new}}(r)$ are written in the corresponding rows of the matrix `Dist`.*

This claim is proved by contradiction. We first assume that an arc (l,k) belong to the graph G and $\mathrm{dist}(l) = w(l,k) + \mathrm{dist}(k)$. However, the arc

(l, k) does not belong to the matrix SP after performing the procedure `UpdateIncomingArcs`. We prove that this contradicts the execution of our procedure. Then we assume that an arc (r, k) belong to G and $\text{dist}_{\text{new}}(r) < \text{dist}_{\text{old}}(r)$. However, after performing the procedure `UpdateIncomingArcs`, the rth row in the matrix `Dist` does not change. We also prove that this contradicts the execution of this procedure.

Now we proceed to the procedure `InsertArc`. Let an arc (i, j) with the weight $v0$ be inserted into the graph G.

```
    procedure InsertArc(h,i,j: integer; v0: word(Dist);
              var Weight,Cost: table; var G,SP: table;
              var Dist: table);
    /* Here $v_0$ is the weight of the arc $(i,j)$ and $h$ is the number of bits
       for coding the infinity. */
    var v1,v2,v3: word(Dist); v4: word(Weight); k: integer;
        X,Y,Y1,Z1,Z2: slice(G); W1,W2, Queue: table;
 1. Begin CLR(Z1); CLR(Y1);
 2.   CLEAR(h,Queue);
    /* Insert position of the arc $(i,j)$ into $G$. */
 3.   X:=COL(i,G); X(j):='1'; COL(i,G):=X;
    /* Insert the weight of the arc $(i,j)$ into the matrix Weight. */
 4.   v4:=ROW(j,Weight);
 5.   REP(1+(i-1)h,ih,v0,v4);
 6.   ROW(j,Weight):=v4;
    /* Insert the weight of the arc $(i,j)$ into the matrix Cost. */
 7.   v4:=ROW(i,Cost);
 8.   REP(1+(j-1)h,jh,v0,v4);
 9.   ROW(i,Cost):=v4;
    /* Save $\text{dist}_{\text{old}}(j)$ and $\text{dist}_{\text{old}}(i)$ in $v_1$ and $v_2$. */
10.   v1:=ROW(j,Dist); v2:=ROW(i,Dist);
    /* Calculate in $v_3$ a distance from the vertex $i$ to $s$ when the path
       starts from the arc $(i,j)$. */
11.   v3:=ADD(v0,v1);
12.   if EQ(v3,v2) then
    /* Insert the arc $(i,j)$ into the matrix $SP$. */
13.     begin
14.       X:=COL(i,SP); X(j):='1'; COL(i,SP):=X;
15.     end;
    /* Set a maximal priority in the matrix Queue for the vertex $i$. */
16.   if LESS(v3,v2) then
17.     begin ROW(i,Dist):=v3; Z1(i):='1';
```

```
18.     end;
    /* Cycle for updating vertices. */
19.   while SOME(Z1) do
20.     begin MIN(Queue,Z1,Z2);
    /* Delete in $SP$ arcs outgoing from the vertex $k$. */
21.       k:=FND(Z2); Z1(k):='0';
22.       COL(k,SP):=Y1;
    /* Insert positions of the arcs $(k,l)$ for which $\text{dist}(k) = w(k,l) + \text{dist}(l)$
       into the matrix $SP$. */
23.       UpdateOutgoingArcs(h,k,G,Weight,Dist,SP);
24.       UpdateIncomingArcs(h,k,G,Cost,Y,Dist,SP);
    /* Save in $v_2$ the current distance from $i$ to $s$. */
25.       v2:=ROW(i,Dist);
    /* Save the value $\text{dist}(l) - \text{dist}(i)$ in the matrix $W_2$ for every $l$th row
       marked with $'1'$ in the slice $Y$. */
26.       SUBTC(Dist,Y,v2,W2);
    /* Write priorities for the new affected vertices to the matrix Queue. */
27.       TMERGE(W2,Y,Queue);
    /* Mark the new affected vertices in $Z_1$ with bits $'1'$. */
28.       Z1:=Z1 or Y;
29.     end;
30. End;
```

Theorem. *Let a directed weighted graph be given as an adjacency matrix G and a matrix `Weight`. Let the matrices `Cost`, SP, and `Dist` be given. Let h be the number of bits for coding the infinity. Let an arc (i,j) be inserted into the graph. Then after performing the procedure `InsertArc` the arc (i,j) is inserted into the matrix SP and its weight is inserted into the matrices `Weight` and `Cost`. Moreover, the matrices SP and `Dist` are updated according to Algorithms A and B.*

Proof. (Sketch.) We prove this by induction in terms of the number q of the affected vertices that appear when a tree with the root i is constructed.

Basis is proved for $q \leq 1$, that is, there is no arc entering the vertex i in G. One can immediately check that after performing lines 1–9, the arc (i,j) is inserted into the graph G and its weight is included into the matrices $Weight$ and `Cost`. After performing lines 10–11, the variable v_3 saves a new distance from the vertex i to s when the path starts from the arc (i,j). Now we compare the new distance to the previous distance v_2 from i to s. If $v_3 > v_2$, nothing needs to be done. Therefore, we have to consider the following two cases.

Case 1. Let $v_3 = v_2$. Then there is no current tree. After performing lines 13–15, the position of the arc (i, j) is inserted into the matrix SP. Then we execute line 19. Since after performing line 1, slice Z_1 consists of zeros and does not change before line 19, we do not carry out the vertices updating cycle (lines 19–29). Therefore, we go to the procedure end.

Case 2. Let $v_3 < v_2$. Then the tree consists of a single vertex i. After performing lines 17–18, the new distance from i to s is written in the ith row of the matrix `Dist` and $Z_1(i) = {'1'}$. After that, we fulfil the cycle for updating vertices (lines 19–29).

One can immediately verify that, after performing lines 20–21, we obtain $k = i$, and slice Z_1 consists of zeros. After performing line 22, the ith column of the matrix SP consists of zeros due to execution of the operation CLR($Y1$) in line 1. This means that the positions of arcs outgoing from the vertex i are deleted from the matrix SP. In view of Claim 1, after performing the auxiliary procedure `UpdateOutgoingArcs` (line 23), positions of arcs (i, l) for which $\text{dist}(i) = w(i, l) + \text{dist}(l)$ are inserted into the matrix SP. In particular, the arc (i, j) is inserted into SP because a new distance v_3 coincides with the current distance from i to s that was previously written in the ith row of the matrix `Dist`. By assumption, there is only a single affected vertex. Therefore, after performing the auxiliary procedure `UpdateIncomingArcs` (line 24), the resulting slice Y consists of zeros. Therefore we do not execute lines 27–28. Since after performing line 29, slice Z_1 consists of zeros, and we go to the exit of the procedure `InsertArc`.

Hence, if $l = 1$ and $v_3 < v_2$, we first write a new distance v_3 from i to s into the ith row of the matrix `Dist`. Then we delete the positions of arcs outgoing from the vertex i in the matrix SP. After that, we insert into the matrix SP the positions of arcs (i, p) for which $\text{dist}(i) = w(i, p) + \text{dist}(p)$.

Step of induction. Let the assertion be true when no more than $q \geq 1$ affected vertices are deleted from the current tree T. We will prove the assertion for $q + 1$ affected vertices.

By analogy with the proof of Case 2, after performing lines 1–11 and 16–18, the position of the arc (i, j) is inserted into the matrix SP, its weight being inserted into the matrices `Weight` and `Cost`, a new distance v_3 from i to s is written in the ith row of the matrix `Dist`, and a maximal priority in the matrix `Queue` is set for the vertex i.

After performing lines 20–22, we obtain $k = i$, $Z_1 = \emptyset$, and the arcs outgoing from the vertex i are deleted from the matrix SP.

In view of Claim 1, after performing the procedure `UpdateOutgoingArcs` (line 23), the positions of arcs (i, p) for which $\text{dist}(i) = w(i, p) + \text{dist}(p)$ are inserted into the matrix SP.

In view of Claim 2, after performing the procedure `UpdateIncomingArcs` (line 24), the slice Y saves the heads of the arcs (r, i), for which $\text{dist}_{\text{new}}(r) < \text{dist}_{\text{old}}(r)$, new distances $\text{dist}_{\text{new}}(r)$ are written in the corresponding rows of the matrix `Dist` and the positions of arcs (r, i), for which $\text{dist}(r) = w(r, i) + \text{dist}(i)$, are inserted into the matrix SP.

By the inductive assumption, after updating the first q affected vertices in the tree T, for every affected vertex r, the positions of arcs (p, r), for which $\text{dist}(p) = w(p, r) + \text{dist}(r)$, and positions of arcs (r, l) for which $\text{dist}(r) = w(r, l) + \text{dist}(l)$ are inserted into the matrix SP, and a new distance from r to s is written in the rth row of the matrix `Dist`.

After updating the qth vertex in the matrix `Queue`, there is a single bit $'1'$ in slice Z_1 that corresponds to the $(q+1)$th affected vertex. Since $Z_1 \neq \emptyset$, we update the $(q + 1)$th affected vertex in the same manner as in the basis. Now slice Z_1 consists of zeros. Therefore we go to the exit of the procedure `InsertArc`. $\square$

Let us evaluate the time complexity of the procedure `InsertArc`. Let h be the number of bits required for coding the maximal weight of the shortest paths to the sink in SP and k be the number of affected vertices that appear after inserting the arc (i, j) into the graph G. One can immediately verify that the auxiliary procedures `UpdateOutgoingEdges` and `UpdateIncomingEdges` take $O(h)$ time each. As shown in [8, 9], the basic procedures used in the procedure `InsertArc`, also take $O(h)$ time each. Since the main cycle (lines 19–29) is performed k times, we obtain that the procedure `InsertArc` takes $O(kh)$ time.

Now we compare implementations of the Ramalingam incremental algorithm and its associative version:

- the Ramalingam incremental algorithm uses a *heap* whose elements are affected vertices with their keys. An associative version employs the matrix `Queue`, whose every rth row saves the current distance from the affected vertex r to the vertex i;
- for every affected vertex u, the Ramalingam incremental algorithm determines successively every arc outgoing from the vertex u and deletes it from $\text{SP}(G)$. The associative version simultaneously determines the positions of all arcs outgoing from u and simultaneously deletes them from $\text{SP}(G)$;
- for every vertex $x \in \text{Succ}(u)$, the Ramalingam incremental algorithm includes the arc (u, x) into $\text{SP}(G)$ if and only if $\text{dist}(u) = w(u, x) + \text{dist}(x)$. The associative version simultaneously inserts the positions of such arcs into the matrix SP;
- for every vertex $y \in \text{Pred}(u)$, the Ramalingam incremental algorithm includes the vertex y into the heap `PriorityQueue` if and only if $w(y, u) + \text{dist}(u) < \text{dist}(y)$. The associative version simultaneously determines the heads of arcs entering the vertex u that satisfy this

inequality. Such vertices are simultaneously included into a set of affected vertices and their priorities are simultaneously written in the corresponding rows of the matrix `Queue`;

- for every vertex $y \in \text{Pred}(u)$, the Ramalingam incremental algorithm inserts the arc (y, u) into $\text{SP}(G)$ if and only if $w(y, u) + \text{dist}(u) = \text{dist}(y)$. The associative version simultaneously determines the positions of arcs entering the vertex u that satisfy this property and simultaneously includes the positions of such arcs into $\text{SP}(G)$.

7. Conclusion

We have proposed an efficient implementation of the Ramalingam incremental algorithm on the STAR-machine having no less than n PEs. The associative version of the Ramalingam incremental algorithm is represented as the procedure `InsertArc`, whose correctness is proved. We have obtained that this procedure takes $O(kh)$ time per an insertion, where h is the number of bits for coding the infinity and k is the number of affected vertices that appear in $\text{SP}(G)$ after inserting an arc. It is assumed that each microstep of the STAR-machine takes one unit of time. We have also compared implementations of the Ramalingam incremental algorithm and its associative version and presented the main advantages of the associative version.

Acknowledgements. I would like to thank referee for his helpful comments.

References

[1] Dijkstra E.W. A note on two problems in connection with graphs // Numerische Mathematik. — 1959. — No. 1. — P. 269–271.

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

[3] Frigioni D., Marchetti–Spaccamela A., Nanni U. Semi-dynamic algorithms for maintaining single source shortest paths trees // Algorithmica. — 1998. — Vol. 25. — P. 250–274.

[4] Frigioni D., Marchetti–Spaccamela A., Nanni U. Fully dynamic algorithms for maintaining shortest paths trees // J. of Algorithms. — Academic Press, 2000. — Vol. 34. — P. 351–381.

[5] Frigioni D., Marchetti–Spaccamela A., Nanni U. Fully dynamic shortest paths in digraphs with arbitrary arc weights // J. of Algorithms. — Elsevier Science, 2003. — Vol. 49. — P. 86–113.

[6] Narváez P., Siu K.-Y., Tzeng H.-Y. New dynamic algorithms for shortest paths tree computation // IEEE/ACM Trans. Networking. — 2000. — Vol. 8. — P. 734–746.

[7] Nepomniaschaya A.S. Language STAR for associative and parallel computation with vertical data processing // Proc. Intern. Conf. Parallel Computing Technologies. — Singapore: World Scientific, 1991. — P. 258–265.

[8] Nepomniaschaya A.S. Solution of path problems using associative parallel processors // Intern. Conf. on Parallel and Distributed Systems, ICPADS'97. — New York: IEEE Press, 1997. — P. 610–617.

[9] 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.

[10] Nepomniaschaya A.S. Associative version of the Ramalingam decremental algorithm for dynamic updating the single-sink shortest-paths subgraph // Proc. 10th Intern. Conf. "Parallel Computing Technologies" (PaCT-09), Novosibirsk, Russia, 2009. — Heidelberg: Springer, 2009. — P. 257–268. — (LNCS; 5698).

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

[12] Ramalingam G. Bounded Incremental Computation. — Heidelberg: Springer, 1996. — (LNCS; 1089).

[13] Ramalingam G., Reps T. An incremental algorithm for a generalization of the shortest paths problem // J. of Algorithms. — Academic Press, 1996. — Vol. 21. — P. 267–305.