Bull. Nov. Comp. Center, Comp. Science, 42 (2018), 41–60

Efficient parallel implementation of the Ramalingam decremental algorithm for updating the all-pairs shortest paths

Anna Nepomniaschaya

Abstract. The paper proposes an efficient parallel implementation of the Ramalingam algorithm for the dynamic update of the all-pairs shortest paths of a directed weighted graph after deleting an edge. To this end, a model of associative parallel systems with vertical data processing (the STAR-machine) is used. With this model, the Ramalingam decremental algorithm for the dynamic update of the all-pairs shortest paths is represented as the main procedure DeleteEdge that uses a group of auxiliary procedures to perform its different parts. We provide the procedure DeleteEdge along with auxiliary procedures, prove the correctness of these procedures and evaluate the time complexity.

Keywords: directed weighted graph, adjacency matrix, decremental algorithm, associative parallel processor, access data by contents, the time complexity.

1. Introduction

The associative processing is a completely different way of storing, manipulating, and retrieving data as compared to conventional computation techniques. Associative (content addressable) parallel processors of the SIMD type belong to fine-grain systems with bit-serial (vertical) processing and simple processing elements (PEs). In [1], such processors are called Vertical Processing Systems (VPS). A distinctive feature of such processors is implementation of a more intelligent memory. Such an architecture performs the data parallelism at the base level, provides a massively parallel search by content, allows one using 2D tables as the basic data structure, and performs the basic operations of searching in constant time [16]. Initially, the vertical processing systems were mainly oriented to nonnumeric processing. By now, the use of VPSs has been extended. The associative processing has been studied since 1960s. But studying the associative processing has never taken off because only a limited amount of memory could be placed on a single die [15]. However, progress in the computer industry and semiconductor technology in recent years has made the associative processing attractive again.

Let us enumerate a few recent results that confirm this assertion. In [20], the associative processor (AP) based on the resistive content addressable

memory is studied. The paper shows that the resistive memory technology potentially allows scaling the AP from a few millions to a few hundred millions of processing units on a single silicon die. Moreover, the resistive AP allows one a much better scalability and a higher carrying out performance compared to the CMOS AP and the popular modern conventional SIMD accelerators Graphics Processing Units (GPUs). In [5], the electronics system used for the massive parallelization performed by the ATLAS Fast Tracker is given. An increase in the center-of-mass energy and luminosity of the Large Hadron Collider (LHC) makes controlling trigger rates with a high efficiency challenge. The LHC is the world largest and most powerful particle accelerator. The LHC is aimed at allowing physicists to test the predictions of different theories of the particle physics. The ATLAS Fast Tracker is a hardware processor built to very quickly reconstruct the charged particle trajectories from the pixel&silicon detectors and makes these tracks available to the algorithms at the software-based processing step. The Fast Tracker exploits the hardware technology with massive parallelism, combining Associative Memory ASICs, FPGAs (Field-Programmable Gate Arrays), and high-speed communication links. In [3], the associative operations of the model MASC (Multiple Associative Computing) were presented. This model is highly efficient when it is used for real time scheduling in control systems for the air traffic. In the current technologies, the architecture of the model MASC has not been built. Therefore the implementation of the MASC associative operations on the GPUs has been proposed.

In [6], we have proposed an abstract model of the SIMD type (the STAR-machine) to simulate running the vertical processing systems at the micro level. This model uses a group of elementary operations that allow one to update the tabular data by contents. On the STAR-machine, associative parallel algorithms are represented as the corresponding procedures written in the language STAR whose correctness has been proved and the time complexity has been evaluated. In [7], we present the basic associative parallel algorithms that are used to design different associative algorithms for different applications. We observe that the architecture of vertical processing systems is best suited for a natural and efficient implementation of graph algorithms. For the STAR-machine, we have built both the new associative parallel graph algorithms and the associative versions of the well-known sequential graph algorithms. First of all, we have constructed a natural straightforward implementation of a group of *classical* graph algorithms. This implementation includes, in particular, associative versions of Dijkstra's algorithm for finding the single source shortest paths [6], Floyd's algorithm for finding the all-pairs shortest paths [8], Kruskal's algorithm and the Prim–Dijkstra one for finding the minimal spanning tree (MST) [9].

Of special interest is the implementation of *dynamic* graph algorithms on the STAR-machine. The objective of a dynamic algorithm is to update

the solution of a problem after dynamic changes faster than to compute the entire graph from a scratch each time with the fastest static algorithm. On the STAR-machine, we have implemented a group of *dynamic* graph algorithms. It includes, in particular, associative parallel algorithms for the dynamic edge update of an MST [10], associative versions of the Italiano algorithms for the dynamic update of the transitive closure of a *directed* graph after inserting and after deleting an edge [11], associative versions of the Ramalingam algorithms for updating the shortest paths subgraph with a sink after inserting and after deleting an edge and their efficient implementations on the STAR-machine [12, 13].

In this paper, we study the dynamic update of the all-pairs shortest paths (APSP) by means of the STAR-machine. The most general types of update operations for the APSP problem include insertions and deletions of edges, update operations on edge weights, finding the shortest distance and finding the shortest path between two vertices, if any. An algorithm is called *fully dynamic* if the update operations include both insertions and deletions of edges. A partially dynamic algorithm is called *incremental* if it supports only insertions, while it is called *decremental* if only deletions are supported.

In [14], we have constructed an associative version of the Ramalingam decremental algorithm for the dynamic update of the all-pairs shortest paths. The associative version is given as a group of associative algorithms that provide execution on the STAR-machine of different parts of the Ramalingam decremental algorithm. In this paper, we present the efficient *parallel* implementation on the STAR-machine of the associative version of the Ramalingam decremental algorithm for updating the all–pairs shortest paths. It consists of the main procedure DeleteEdge along with a group of auxiliary procedures. We prove the correctness of these procedures and evaluate the time complexity.

2. A model of the associative parallel machine

Here, we propose a brief description of the STAR-machine that simulates the run of vertical processing systems. It uses some Staran properties and the Russian associative processor [4]. The current version of the STAR-machine has been presented in [7]. Recently we have started the project of implementing the STAR-machine on GPU NVIDIA GEFORCE 920m [18, 19]. It will allow us to implement the associative algorithms on a real hardware of GPUs.

The STAR-machine consists of the following three components: a sequential control unit, an associative processing unit, and a matrix memory for the associative processing unit. In [7], the run of the STAR-machine is described in considerable detail. Input binary data are loaded into the ma-

trix memory in the form of 2D tables. In any table, the rows are numbered from top to bottom and the columns — from left to right. An associative processing unit is represented as a group of vertical registers. A vertical register can be regarded as a one-column array. To update a matrix, its bit columns are stored in the registers which perform the necessary Boolean operations. To simulate the data processing in the matrix memory the types **word**, **slice**, and **table** are used. 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. For simplicity, a variable of the type **slice** sometimes is called *slice*. In this paper, we will use constants for the types **slice** and **word** without single quotation marks.

Now let us present some operations of the STAR-machine which are used in the paper.

Let X and Y be slices and i be a variable of the type **integer**. The following operations are used for slices:

CLR(Y) simultaneously sets all components of Y to 0;

$Y(i)$ selects the value of the ith component of Y;

FND(Y) returns the ordinal number of the uppermost bit 1 of Y;

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

To carry out the data parallelism, bitwise Boolean operations are introduced in the usual way: X *and* Y, X *or* Y, *not* Y, X *xor* Y.

The predicate SOME(Y) results in **true** if there is at least a single bit 1 in the slice Y. For simplicity, the notation $Y \neq \emptyset$ denotes that the predicate SOME(Y) results in **true**.

Note that the predicate SOME(Y) and all operations for the type **slice** are also performed for the type **word**. In addition, one employs bitwise Boolean operations between a variable w of the type **word** and a variable Y of the type **slice**, where the number of bits in w coincides with the number of bits in Y.

Moreover, for variables v and w of the type **word**, the following two operations are used:

TRIM(i, j, w) returns the substring of w of the form $w(i)w(i+1)\dots w(j)$, where $1 \leq i < j \leq |w|$;

REP(i, j, v, w) 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 a variable T of the type **table** the following two operations are used:

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

COL(i, T) returns its ith column.

The STAR statements are defined in the same manner as for Pascal. They are later used for presenting the procedures.

Now, we briefly enumerate a group of the basic procedures implemented on the STAR-machine [7]. These procedures use a given global slice X to indicate with bit 1 the row positions used in the corresponding procedure.

The procedure MATCH(T, X, v, Z) defines the positions of those rows of the given matrix T which coincide with a given pattern v. It returns the slice Z, where positions of these rows are marked with bit 1.

The procedure MIN(T, X, Z) defines the positions of those rows of a given matrix T, where a minimal element is located. It returns the slice Z, where $Z(i) = 1$ if and only if ROW(i, T) is the minimal element in T and $X(i) = 1$.

The procedure SETMIN(T, F, X, Z) defines the positions of given matrix T rows that are less than the corresponding rows of the matrix F. It returns the slice Z, where positions of such rows of the matrix T are marked with 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 the slice Z, where the positions of identical rows are marked with 1.

The procedure TMERGE(T, X, F) writes the rows of the matrix T, selected with bit 1 in the slice X, in the corresponding rows of the matrix F. Other rows of the matrix F do not change.

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

The procedure ADDV(T, F, X, R) writes into the matrix R the result of the parallel addition of the corresponding rows of the matrices T and F, whose positions are selected with bit 1 in the given slice X.

The procedure ADDC(T, X, v, F) adds the binary word v to the rows of the matrix T selected with bit 1 in the slice X, and writes down the result into the corresponding rows of the matrix F. Other rows of the matrix F are set to zeros.

Following Foster [2], we assume that each elementary operation of our model (its microstep) takes one unit of time. We measure the *time complexity* of an associative algorithm by counting all elementary operations performed in the worst case.

In [7], we have shown that the basic procedures take $O(k)$ time each, where k is the number of bit columns in the corresponding matrix.

3. Preliminaries

Let $G = (V, E)$ be a *directed weighted graph* with n vertices and m directed edges (arcs). We assume that $V = \{1, 2, \ldots, n\}$. Let wt denote a function that assigns a weight to every edge.

An *adjacent matrix* $\texttt{Adj} = [a_{ij}]$ 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 $i = \text{tail}(e)$ and $j = \text{head}(e)$. Also, if $(i, j) \in E$, then j is said to be *adjacent* to i. We assume that all arcs have nonnegative weights and $\text{wt}(u, v) = \infty$ if $(u, v) \notin E$.

The *shortest path* from u to w is the path of a minimal sum of weights of its edges. Let $\text{dist}(u, z)$ denote the *weight* of the shortest path from the vertex u to the distinguished vertex z called *sink*. Let $\texttt{Dist}$ be a matrix whose every entry $\texttt{Dist}[i, j]$ represents the weight of the shortest path from the vertex i to the vertex j.

By analogy with Ramalingam [17], we introduce the following notions.

Let an arc (i, j) be deleted from G. Let $\texttt{AffectedV}$ denote a set of all vertices u in G such that all paths from u to the selected sink z include the deleted arc (i, j).

A predicate $SP(a, b, c)$ is defined as follows:

$$SP(a, b, c) \equiv (\text{dist}(a, c) = \text{wt}(a, b) + \text{dist}(b, c)) \wedge (\text{dist}(a, c) \neq \infty).$$

This predicate verifies whether the arc (a, b) belongs to the shortest path from the vertex a to the selected sink c.

Now we present an example. Let a graph G be given (Figure 1). It can be seen that in G there is a single path to the sink s from vertices 1, 2, 3, 4, and 8, while there are two different paths to the sink s from other vertices.

Let the arc $4 \to 2$ be deleted from G (Figure 2). Then vertices 4, 7, 8, and 10 become *affected* because there is no path from them to the sink s.

Let us consider the execution of the predicate $SP(a, b, c)$ for different parameters in Figure 2. The predicate $SP(6, 5, s)$ returns **true** because

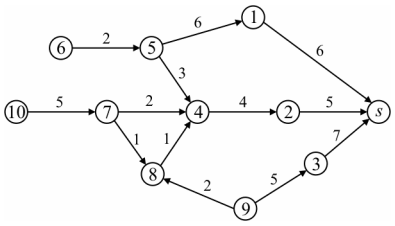

Figure 1. The graph G

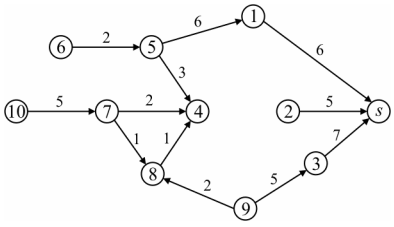

Figure 2. Deletion of the arc $(4, 2)$ from G

there is shortest path $6 \to 5$, $5 \to 1$, and $1 \to s$ starting from the arc $6 \to 5$. The predicate $SP(7, 8, s)$ returns **false** because there is no shortest path from vertex 7 to the sink s.

4. The Ramalingam decremental algorithm for updating the all-pairs shortest paths

In [17], Ramalingam represents the problem of the dynamic update of the all-pairs shortest paths after edge deletion by using the dynamic single sink shortest path problem after edge deletion as a subroutine but with a different sink vertex at each call. Following Ramalingam, a vertex y is called *an affected sink* if there exists such a vertex x that $\mathrm{dist}(x, y)$ changes after edge deletion. As is shown in [17], the set of affected sink vertices after deleting an edge $i \to j$ coincides with the set of all affected vertices for the single source shortest path problem with the source vertex i. Therefore the decremental algorithm for the dynamic update of the all-pairs shortest paths first determines the set of all *affected sink vertices* and then it applies a *modification* of the decremental algorithm for the dynamic update of the single sink shortest paths for every affected sink vertex.

Before presenting a modification of the above mentioned Ramalingam decremental algorithm, we first briefly think back to it. Let a directed graph G, a sink z and the shortest paths subgraph $SP(G)$ be given. Let an arc (i, j) be deleted from G. Then the Ramalingam decremental algorithm for the dynamic update of the single sink shortest paths subgraph constructs a new shortest paths subgraph of G. To this end, first one determines a set of all affected arcs obtained after deleting the arc (i, j) from G. Then the affected arcs are deleted from $SP(G)$, and for every affected vertex one computes a new shortest path to z. We observe that a parallel implementation of this algorithm on the STAR-machine is considered in [12].

Now we present the *modification* of the Ramalingam decremental algorithm for the dynamic update of the single sink shortest paths. Let a directed graph G and a sink z be given. Let an arc (i, j) be deleted from G. The modification consists of the following two stages.

At the *first stage*, one determines the set `AffectedV` of all affected vertices obtained after deleting the arc (i, j) from G. At the *second stage*, for every affected vertex v_i, one computes a new distance to the sink z.

The first stage is performed as follows. At first, `AffectedV` $= \emptyset$. To construct it, an auxiliary set of vertices `WorkSet` is used. Initially, all the arcs (i, x) are analyzed. For every *head* x, one checks whether the predicate $SP(i, x, z)$ returns **true**. If all predicates return **false**, then `WorkSet` $:= \{i\}$. Vertices in `WorkSet` are sequentially updated. The current updated vertex u is deleted from `WorkSet` and is included into the set `AffectedV`. Then all arcs (x, u) are analyzed. For every *tail* x for which the predicate $SP(x, u, z)$

returns **true**, one checks whether there does not exist such a non-affected head y of an arc (x, y) for which $SP(x, y, z)$ returns **true**. In this case, the vertex x is included into `WorkSet`.

To perform the second stage, one uses a heap `PriorityQueue`, whose elements are affected vertices with a key. Initially `PriorityQueue` $= \emptyset$. To build it, for every affected vertex, one first computes a current key. To this end, for every affected vertex k, one defines non-affected heads r of arcs outgoing from the vertex k for which there is a path to z. Then for every vertex r, one computes the sum $\mathrm{wt}(k, r) + \mathrm{dist}(r, z)$. The current key of k in the heap is defined as the *minimal* value of such sums.

After that, one updates the heap `PriorityQueue` as follows. At every iteration, a vertex with the minimum key in the heap (say, a) is deleted from the set `PriorityQueue`. Further all the arcs (c, a) are analyzed. For every tail c, for which $\mathrm{wt}(c, a) + \mathrm{dist}(a, z) < \mathrm{dist}(c, z)$, the current value $\mathrm{dist}(c)$ is equal to $\mathrm{dist}_{\mathrm{new}}(c)$, and this value becomes the new key for the vertex c in `PriorityQueue`. If $c \in$ `PriorityQueue`, the previous key of c receives a new value. Otherwise, the vertex c is included into the heap with the key $\mathrm{dist}_{\mathrm{new}}(c)$. The process is completed after updating all vertices in the heap.

A modification of the decremental algorithm for updating distances to the sink z is given as a function DeleteUpdate that returns the set of affected vertices.

Finally, the Ramalingam decremental algorithm for updating the all-pairs shortest paths runs as follows. At first, the arc $i \to j$ is deleted from G. Then by means of the function DeleteUpdate, one defines the set of affected sink vertices. After that the function DeleteUpdate is applied for every affected sink vertex.

In [14], to represent this modification on the STAR-machine, at first, a special data structure is proposed. Then one constructs the following three algorithms:

Algorithm A: an associative parallel algorithm for selecting a set of affected vertices obtained after deleting the arc (i, j);

Algorithm B: an associative parallel algorithm for finding initial distances from all affected vertices to the sink z;

Algorithm C: an associative parallel algorithm for finding a *new* distance from every affected vertex to z.

On the STAR-machine, the modification of the decremental algorithm for updating the single sink shortest paths is given as procedure DeleteUpdate that performs in succession Algorithms A, B, and C.

5. The data structure

Before presenting the data structure, we recall that the Ramalingam decremental algorithm uses a given directed weighted graph G, distances between all-pairs of vertices, and a given edge (i, j). Let w be a maximal weight of edges in G. The maximal distance between all-pairs of vertices is less than the value $t = w \times n$, where n is the number of graph vertices. In this case, $h = \lceil \log t \rceil$, that is, h bits will be used for representing any distance between any pairs of vertices in the matrix `Dist`. We observe that initially the distances between all-pairs of vertices are defined by means of the classical Floyd's algorithm.

We will use the following data structure:

- an $n \times n$ adjacency matrix `Adj`, whose every ith column saves with bits 1 the heads of arcs outgoing from the vertex i;
- an $n \times n$ adjacency matrix `Adj1`, whose every ith column saves with bits 1 the vertices for which there is the shortest path from the vertex i;
- an $n \times hn$ matrix `Weight` that consists of n fields having h bits each. The weight of an arc (i, j) is written in the jth row of the ith field;
- an $n \times hn$ matrix `Cost` that consists of n fields having h bits each. The weight of an arc (i, j) is written in the ith row of the jth field;
- an $n \times hn$ matrix `Dist` that consists of n fields having h bits each. The distance from the vertex i to the vertex j is written in the jth row of the ith field;
- an $n \times hn$ matrix `Dist1` that consists of n fields having h bits each. The distance from the vertex i to the vertex j is written in the ith row of the jth field;
- a slice `AffectedV` that saves with bits 1 positions of all affected vertices.

We notice 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. Moreover, every jth row of the matrix `Adj` saves with bits 1 tails of the arcs entering the vertex j, while every jth row of the matrix `Adj1` saves the vertices from which there is a shortest path entering the vertex j.

Remark 1. We should mention that Algorithm A uses, in particular, the slices `WS` and `AffectedV`. Algorithm B uses, in particular, the slice `AffectedV` and the matrices `Weight` and `Dist1`. This algorithm constructs a matrix `Queue`, whose every ith row saves the initial distance from the

affected vertex i to the sink z. Algorithm C uses, in particular, the slice `AffectedV` and the matrices `Queue`, `Cost`, `Dist`, and `Dist1`.

6. Implementation of the associative algorithm for finding affected vertices

In this section, we first provide two auxiliary procedures ComputePred1 and ComputePred2. Then we propose the procedure FindAffectedVert for finding affected vertices after deleting an arc from the graph and justify its correctness.

The procedure ComputePred1 simultaneously defines the non-affected heads of arcs outgoing from a given vertex u for which the predicates $SP(u, x, z)$ are true. It uses, in particular, the matrices `Adj`, `Adj1`, `Weight`, `Dist`, and `Dist1`. It returns a slice to save by bits 1 the above mentioned heads of arcs.

Let us briefly explain the *main idea* of this procedure. We first define positions of arcs (u, x) that belong to different paths from u to z. Then we define the weights of these paths. After selecting a distance from u to z, we save the positions of those arcs that belong to the shortest path from u to z.

```
procedure ComputePred1(h,u,z: integer; AffectedV: slice(Adj);
          Adj,Adj1: table; Weight, Dist, Dist1: table;
          var Y: slice(Adj));
var l1,l2: integer; X,X1,X2: slice(Adj); R1,R2,R3: table;
    v: word(Dist); v1: word(Adj); w1: word(R1);
1. Begin X1:=COL(u,Adj);
   /* The slice X1 saves the heads of arcs outgoing from u. */
2.   v1:=ROW(z,Adj1); X2:=CONVERT(v1);
   /* The slice X2 saves the vertices from which there is the shortest path to z. */
3.   X:=X1 and X2;
4.   X:=X and (not AffectedV);
   /* The slice X saves non–affected heads of arcs outgoing from u that belong to
      different paths from u to z. */
5.   TCOPY1(Weight,u,h,R1);
6.   TCOPY1(Dist1,z,h,R2);
7.   ADDV(R1,R2,X,R3);
   /* The matrix R3 saves the weights of different paths from u to z. */
8.   v1:=ROW(z,Dist1);
9.   l1:=1+(u-1)*h; l2:=u*h;
10.  w1:=TRIM(l1,l2,v1);
   /* The row w1 saves the distance from u to z. */
11.  MATCH(R3,w1,X,Y);
```

```
/* The slice Y saves positions of the matrix R3 rows that coincide with
   the distance from u to z. */
12. End;
```

Lemma 1. *Let a slice* `AffectedV` *and the matrices* `Adj`, `Adj1`, `Weight`, `Dist`, *and* `Dist1` *be given. Let the parameters* h, u, *and* z *be given as well. Then the procedure ComputePred1 returns a slice* Y *that saves non-affected heads of arcs* (u, s) *for which the predicates* $SP(u, s, z)$ *are true.*

Proof. We prove the lemma by contradiction. Let the predicate $SP(u, r, z)$ be true for a non-affected head of the arc (u, r). However, after performing the procedure ComputePred1, $Y(r) = 0$. We can see that this contradicts the execution of the procedure ComputePred1.

Really, after performing line 1, $X1(r) = 1$ because the arc (u, r) belongs to G. After performing line 2, $X2(r) = 1$ because by assumption the predicate $SP(u, r, z)$ is true. After fulfilling lines 3–4, $X(r) = 1$ since r is a non-affected vertex from which there is the shortest path to the sink z. After performing lines 5–6, the rth row of the matrix $R1$ saves $\mathrm{wt}(u, r)$ and the rth row of the matrix $R2$ saves $\mathrm{dist}(r, z)$. After fulfilling line 7, the matrix $R3$ saves the weights of different paths from u to z. In particular, its rth row saves the weight of the path from u to z that starts from the arc (u, r). Now, we have to select the positions of the paths from u to z having a minimal weight. Since $\mathrm{dist}(u, z)$ is written in the zth row of the matrix `Dist1`, we perform lines 8–10 and select the substring $w1$ that corresponds to $\mathrm{dist}(u, z)$. Now, after performing the procedure $\mathrm{MATCH}(R3, w1, X, Y)$ (line 11) we obtain the slice Y that saves the positions of the matrix $R3$ rows that coincide with $\mathrm{dist}(u, z)$. Since $X(r) = 1$ and the predicate $SP(u, r, z)$ is true, we obtain that $Y(r) = 1$ that contradicts the assumption. □

The procedure ComputePred2 simultaneously defines the tails of arcs entering a given vertex u for which the predicates $SP(y, u, z)$ are true. It uses, in particular, the matrices `Adj`, `Weight`, `Dist`, and `Dist1`. It returns a slice to save by bits 1 the above mentioned tails of arcs.

Let us briefly explain the main idea of this procedure. We first define the weights of arcs entering the vertex u. After selecting a distance from u to z, we define the weights of different paths to z that start from the tails of arcs entering u. Then we select different distances to z starting from the tails of arcs entering u. Finally, we compare these distances with the weights of the corresponding paths to z.

```
procedure ComputePred2(h,u,z: integer; Adj: table;
          Cost,Dist,Dist1: table; var Y: slice(Adj));
var l1,l2: integer; X: slice(Adj); R1,R2,R3: table;
    v: word(Dist); v1: word(Adj); v2: word(R1);
```

```
1. Begin v1:=ROW(u,Adj); X:=CONVERT(v1);
   /* The slice X saves the tails of arcs entering u. */
2.   TCOPY1(Cost,u,h,R1);
   /* The matrix R1 saves the weights of arcs entering u. */
3.   v:=ROW(z,Dist);
4.   l1:=1+(u-1)h; l2:=uh;
5.   v2:=TRIM(l1,l2,v);
   /* The row v2 saves the distance from u to z. */
6.   ADDC(R1,v2,X,R2);
   /* The matrix R2 saves the weights of different paths to z starting from the tails
      of arcs entering u. */
7.   TCOPY1(Dist1,z,h,R3);
   /* The matrix R3 saves different distances to z. */
8.   HIT(R2,R3,X,Y);
   /* The slice Y saves positions of the matrix R2 rows that coincide with
      the corresponding distances to z. */
9. End;
```

Lemma 2. *Let the matrices* `Adj`, `Cost`, `Dist`, *and* `Dist1` *be given. Let the parameters h, u, and z be also given. Then the procedure ComputePred2 returns the slice Y that saves the tails of arcs entering the vertex u from which there are the shortest paths to the sink z.*

This lemma is proved by analogy with Lemma 1.

Remark 2. Let us mention that the auxiliary procedures ComputePred1 and ComputePred2 take $O(h)$ time each because of applying the basic procedures [7].

Now we propose the procedure FindAffectedVert. It uses the matrices `Adj`, `Adj1`, `Weight`, `Cost`, `Dist`, and `Dist1` and an auxiliary slice WS. It returns the slice `AffectedV`, where the positions of all affected vertices are marked with bits 1.

Let us briefly explain the main idea of this procedure. We first include the vertex i into the slice WS if there is no shortest path from i to z after deleting the edge (i, j). While $WS \neq \emptyset$, we delete the first bit 1 (say k) from the slice WS and insert it into the slice `AffectedV`. Then by means of a slice, say $X2$, we save the tails of arcs entering the vertex k. Every vertex u from the slice $X2$ is included into the slice WS if there is no shortest path from u to z after deleting the edge (i, j).

```
procedure FindAffectedVert(h,i,z: integer; Adj,Adj1: table;
          Weight,Cost,Dist,Dist1: table;
```

```
          var AffectedV: slice(Adj));
  /* The arc (i, j) has been deleted from the graph G. */
  var k,r,u: integer; X,X1,X2,Y,Y1,Y2,WS: slice(Adj);
      v: word(Adj);
1. Begin CLR(WS); CLR(AffectedV);
2.   Y1:=COL(i,Adj);
  /* The slice Y1 saves the heads of arcs outgoing from the vertex i. */
3.   v:=ROW(z,Adj1); Y2:=CONVERT(v);
  /* The slice Y2 saves the vertices from which there is the shortest path
     to the sink z. */
4.   Y:=Y1 and Y2;
  /* The slice Y saves the heads of arcs outgoing from i that belong to
     different paths from i to z. */
5.   ComputePred1(h,i,z,AffectedV,Adj,Adj1,Weight,Dist,Dist1,X1);
  /* The slice X1 saves the result of the procedure ComputePred1. */
6.   if not SOME(X1) then WS(i):=1;
7.   while SOME(WS) do
  /* The cycle for selecting affected vertices. */
8.   begin k:=STEP(WS);
9.     AffectedV(k):=1;
  /* The vertex k is saved in the slice AffectedV. */
10.    v:=ROW(k,Adj); X2:=CONVERT(v);
  /* The slice X2 saves the tails of arcs entering k. */
11.    while SOME(X2) do
12.    begin u:=STEP(X2);
13.      ComputePred1(h,u,z,AffectedV,Adj,Adj1,Weight,
                      Dist,Dist1,X);
14.      if not SOME(X) then WS(u):=1;
15.    end;
16.  end;
17. End;
```

Theorem. *Let G be a directed weighted graph. Let an arc (i, j) be deleted from G. Let the matrices* `Adj`*,* `Adj1`*,* `Weight`*,* `Cost`*,* `Dist`*, and* `Dist1` *be given. Then the procedure FindAffectedVert returns the slice* `AffectedV`*, where the positions of affected vertices are marked with bits 1.*

Proof. We prove this by induction in terms of the number of vertices l to be included into the slice `AffectedV`.

The basis is checked for $l = 1$, that is, only the vertex i is an affected one after deleting the edge (i, j) from G.

After performing lines 1–4, the slices WS and `AffectedV` consist of zeros, and the slice Y saves the heads of arcs outgoing from the vertex i that belong to different paths from i to z. After performing line 5, the slice $X1$ saves non-affected heads of the arcs (i, r) that belong to the corresponding shortest paths from i to z. By the assumption, the vertex i is an affected one. Therefore the slice $X1$ has to consist of zeros. After performing line 6, the vertex i is included into the slice WS. After execution of lines 7–9, the slice WS consists of zeros and the vertex i is saved in the slice `AffectedV`. After performing line 10, the slice $X2$ saves the tails of arcs entering the vertex i that belong to the corresponding shortest paths to z. If $X2$ consists of zeros, we go to the procedure end. Otherwise, for each such a tail u, we execute the procedure ComputePred1 (line 13). Since only the vertex i is an affected one, the procedure ComputePred1 has to return the slice $X \neq \emptyset$ to each tail u. Therefore after performing the cycle `while SOME(X2) do` (lines 12–16), we go to the procedure end.

Step of induction. Let the assertion be true for $l \geq 1$ vertices included into the slice `AffectedV`. We prove this for $l + 1$ vertices. By the inductive assumption, after including the first l vertices into the slice `AffectedV`, they should be deleted from the slice WS. Moreover, the slice WS saves the tails of arcs entering an affected vertex that do not start any shortest path to the sink z. After including the lth vertex into the slice `AffectedV`, the slice WS saves the position of the $(l + 1)$th affected vertex. Therefore the cycle `while SOME(WS) do` (line 7) is performed. After performing lines 8–9, we first delete a single vertex from the slice WS and $WS = \emptyset$. Then we include this vertex into the slice `AffectedV`. After performing line 10, the slice $X2$ saves the tails of arcs entering the $(l + 1)$th affected vertex. Then we fulfil the cycle `while SOME(X2) do` (lines 12–16). Since there are only $l + 1$ affected vertices after deleting the edge (i, j) from G, no new vertex is included into the slice WS. Therefore the cycle `while SOME(WS) do` (lines 8–17) is finished, and we go to the procedure end. □

Let us evaluate the time complexity of the FindAffectedVert procedure. Let h be the parameter defined in Section 5 (that is, the size of the field) and k be the number of affected vertices that appear after deleting the arc (i, j) from G. The procedure FindAffectedVert takes $O(hk)$ time because the cycle `while SOME(WS) do` (line 7) is performed k times and the auxiliary procedures ComputePred1 and ComputePred2 take $O(h)$ time each.

7. Updating the all-pairs shortest paths after edge deletion on the STAR-machine

In this section, we first propose two procedures for implementing the algorithms B and C presented in [14]. Then we represent the *modification* of

the decremental algorithm for updating the single sink shortest paths on the STAR-machine.

Let us consider the procedure FindCurrentDist. With the matrices `Adj`, `Adj1`, `Weight`, and `Dist1`, it constructs a matrix `Queue` whose every ith row saves the initial distance from the affected vertex i to the sink z.

```
procedure FindCurrentDist(h,z: integer; Adj,Adj1: table;
          AffectedV: slice(Adj); Weight: table; Dist1: table;
          var Queue: table);
var X,X1,X2,Y,Y1: slice(Adj); k,l: integer; R1,R2,F: table;
    v: word(Queue); w: word(Adj);
1. Begin Y:=AffectedV;
2.   while SOME(Y) do
3.   begin k:=STEP(Y);
4.     X:=COL(k,Adj);
5.     X:=X and (not AffectedV);
   /* The slice X saves non-affected vertices being the heads of arcs
      outgoing from the vertex k. */
6.     w:=ROW(z,Adj1); X1:=CONVERT(w);
   /* The slice X1 saves the vertices from which there is the shortest path
      to the sink z. */
7.     X2:=X1 and X;
   /* The slice X2 saves non-affected vertices for which there is a path
      from the vertex k to the sink z. */
8.     TCOPY1(Weight,k,h,R1);
   /* The matrix R1 saves the kth field of the matrix Weight. */
9.     TCOPY1(Dist1,z,h,R2);
   /* The matrix R2 saves the zth field of the matrix Dist1. */
10.    ADDV(R1,R2,X2,F);
   /* The matrix F saves the weights of different paths from k to z. */
11.    MIN(F,X2,Y1); l:=FND(Y1);
   /* The lth row of the matrix F saves the current minimal distance from k to z. */
12.    v:=ROW(l,F); ROW(k,Queue):=v;
   /* In every kth row of the matrix Queue a new current distance
      from k to z is written. */
13.  end;
14. End;
```

It is easy to check that the procedure FindCurrentDist takes $O(hk)$ time.

Now we propose the procedure FindNewDist. With the matrices `Queue`, `Cost`, `Dist`, and `Dist1`, it defines a *new* distance from every affected vertex to z.

```
   procedure FindNewDist(h,z: integer; Adj,Adj1: table;
             AffectedV: slice(Adj); Cost: table; var Queue: table;
             var Dist,Dist1: table);
   var X,X1,X2,Y,Y1,Y2: slice(Adj);
       i,j,k,l: integer; R1,R2,F: table;
       v: word(Queue); v1: word(Adj); w: word(Dist);
 1. Begin Y:=AffectedV;
 2.   while SOME(Y) do
 3.   begin MIN(Queue,Y,X);
 4.     k:=FND(X); Y(k):=0;
 5.     v:=ROW(k,Queue);
   /* The new distance from the vertex k to z is written in the kth row
      of the matrix Queue. */
 6.     w:=ROW(z,Dist);
 7.     i:=1+(k-1)*h; j:=k*h;
 8.     Rep(i,j,v,w); ROW(z,Dist):=w;
   /* The new distance from k to z is written in the matrix Dist. */
 9.     w:=ROW(k,Dist1);
10.     i:=1+(z-1)*h; j:=z*h;
11.     Rep(i,j,v,w); ROW(k,Dist1):=w;
   /* The new distance from k to z is written in the matrix Dist1. */
12.     v1:=ROW(k,Adj); X:=CONVERT(v1);
   /* The slice X saves the tails of arcs entering the vertex k. */
13.     TCOPY1(Cost,k,h,F);
   /* The matrix F saves the kth field of the matrix Cost. */
14.     ADDC(F,X,v,R1);
   /* In every lth row of the matrix R1 marked with 1 in the slice X,
      a weight of a path to z starting with the arc (l,k) is written. */
15.     SETMIN(R1,Queue,Y,Y1);
   /* The slice Y1 saves positions of the matrix R1 rows where
      ROW(i,R1) < ROW(i,Queue). */
16.     TMERGE(R1,Y,Queue);
   /* In every ith row of the matrix Queue, a current distance to z is written
      if Y(i) = 1. */
17.   end;
18. End;
```

The correctness of the procedures FindCurrentDist and FindNewDist is proved by contradiction.

Let us represent a *modification* of the decremental algorithm for updating the single sink shortest paths on the STAR-machine. It performs in succession of Algorithms A, B, and C. On the STAR-machine, it is given as

procedure DeleteUpdate that returns the slice `AffectedV` and the current matrices `Dist` and `Dist1`.

```
procedure DeleteUpdate(i,h,z: integer; Adj,Adj1: table;
          Weight,Cost: table; var Dist,Dist1: table;
          var AffectedV: slice(Adj));
/* Here h is the size of the field, z is the sink, and (i,j) is the arc
   to be deleted from G. */
1. Begin
2.   FindAffectedVert(h,i,z,Adj,Adj1,Weight,Cost,Dist,Dist1,
                      AffectedV);
   /* The procedure returns the slice AffectedV that saves affected vertices
      obtained after deleting (i,j) from G. */
3.   FindCurrentDist(h,z,Adj,Adj1,AffectedV,Weight,Dist1,Queue);
   /* The procedure returns the matrix Queue whose every kth row saves
      a current distance from the affected vertex k to z. */
4.   FindNewDist(h,z,Adj,Adj1,AffectedV,Cost,Queue,Dist,Dist1);
   /* The procedure computes a new distance from every affected vertex to z and
      saves it in matrices Dist and Dist1. */
5. End;
```

Let us evaluate the time complexity of the procedure DeleteUpdate. It takes $O(hk)$ time because the procedures FindAffectedVert, FindCurrentDist, and FindNewDist take $O(hk)$ time each.

The associative version of the Ramalingam decremental algorithm for updating the all-pairs shortest paths runs as follows. At first, the arc $i \to j$ is deleted from G. Then by means of Algorithm A, one defines a set of *affected sink* vertices. After that the procedure DeleteUpdate is applied to every affected sink vertex.

```
procedure DeleteEdge(i,j,h: integer; Weight,Cost: table;
          var Adj,Adj1: table; var Dist,Dist1: table);
/* The arc (i,j) will be deleted from the graph G, h is the size of the field. */
var X,AffectedSinks: slice(Adj); z: integer;
1. Begin X:=COL(i,Adj); X(j):=0; COL(i,Adj):=X;
   /* The arc (i,j) is deleted from G. */
2.   FindAffectedVert(h,i,z,Adj,Adj1,Weight,Cost,Dist,Dist1,
                      AffectedV);
3.   AffectedSinks:=AffectedV;
4.   while SOME(AffectedSinks) do
5.   begin z:=STEP(AffectedSinks);
6.     DeleteUpdate(i,h,z,Adj,Adj1,Weight,Cost,Dist,Dist1,
                    AffectedV);
```

```
7.   end;
8. End;
```

Let us evaluate the time complexity of the procedure DeleteEdge. To this end, we first enumerate the set $R = \{r_1, \ldots, r_l\}$ of the sink affected vertices obtained after performing the procedure FindAffectedVert. Let q_k denote the number of affected vertices when the sink vertex $r_k \in R$ is used. Then the total number of all affected vertices for different sinks from the set R is defined as $q = \sum_{i=1}^{l} q_i$. Hence, the procedure DeleteEdge takes $O(hq)$ time. We can see that h depends on the input, while the number of affected vertices is the output.

Now, let us present the main advantages of the associative version of the Ramalingam decremental algorithm for updating the all-pairs shortest paths:

1. For every affected vertex u and the sink z, the associative version simultaneously computes the predicates $SP(u, x, z)$ for all the heads of arcs outgoing from the vertex u and the predicates $SP(y, u, z)$ for all the tails of arcs entering the vertex u.
2. For every affected vertex u and the sink z, the associative version simultaneously determines the weights of paths from u to z and writes a minimum value of these weights in the corresponding row of the matrix `Queue`.
3. After selecting a new distance from the current affected vertex k to z, the associative version *simultaneously* computes the weights of paths to z, each starting from an arc entering the vertex k and saves them in a matrix, say $R1$. After that one *simultaneously* replaces the matrix `Queue` rows, where $\mathrm{ROW}(i, R1) < \mathrm{ROW}(i, Queue)$ with the corresponding rows of the matrix $R1$.

8. Conclusion

We have proposed an efficient parallel implementation of the Ramalingam decremental algorithm for updating the all-pairs shortest paths on the STAR-machine having no less than n PEs. The associative version of this Ramalingam decremental algorithm is represented as procedure DeleteEdge that includes a group of auxiliary procedures for performing different parts of this algorithm. We have proved the correctness of the auxiliary procedures and the procedure DeleteEdge and evaluated the time complexity. We have obtained that the procedure DeleteEdge takes $O(hq)$ time per deletion, where h is the size of the field, q is the total sum of affected vertices for different sink vertices. It is assumed that each microstep of the STAR-machine

takes one unit of time. The proposed data structure and the proposed technique for updating the all-pairs shortest paths after deleting an arc from G can be used to solve other tasks.

We are planning to implement on GPU NVIDIA GEFORCE 920m the most important associative graph algorithms presented on the STAR-machine.

References

[1] Fet Y.I. Vertical processing systems: A survey. — IEEE, Micro, 1995. — P. 65–75.

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

[3] Jin M. Associative operations from MASC to GPU // The 21th Int. Conf. on Parallel and Distributed Processing Techniques and Applications, PDPTA'15. — 2015. — P. 388–393.

[4] Mirenkov N. The Siberian approach for an open-system high-performance computing architecture // J. Computing and Control Engineering. — 1992. — Vol. 3, No. 3. — P. 137–142.

[5] Nedaa A. A hardware fast tracker for the ATLAS trigger // Physics of Particles and Nuclei Letters. — 20116. — Vol. 13, No. 5. — P. 527–531.

[6] 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.

[7] Nepomniaschaya A.S. Basic associative parallel algorithms for vertical processing systems // Bull. the Novosibirsk Computing Center. — NCC Publisher, 2009. — IIS Special Issue 29. — P. 63–77. — http://ncc.bulletin.ru/files/article/nepomniaschaya.pdf.

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

[9] Nepomniaschaya A.S. Comparison of performing the Prim–Dijkstra algorithm and the Kruskal algorithm by means of associative parallel processors // Cybernetics and System Analysis. — 2000. — No. 2. — P. 19–27 (In Russian, English translation by Plenum Press).

[10] Nepomniaschaya A.S. Associative parallel algorithms for dynamic edge update of minimum spanning trees // Proc. 7th Int. Conf. PaCT 2003. — Springer, 2003. — P. 141–150. — (Lect. Notes Comp. Sci.; 2763).

[11] Nepomniaschaya A.S. Efficient implementation of the Italiano algorithms for updating the transitive closure on associative parallel processors // Fundamenta Informaticae. — IOS Press, 2008. — Vol. 89, No. 2–3. — P. 313–329.

[12] Nepomniaschaya A.S. Efficient parallel implementation of the Ramalingam decremental algorithm for updating the shortest paths subgraph // Computing and Informatics. — 2013. — Vol. 32. — P. 331–354.

[13] Nepomniaschaya A.S. Associative version of the Ramalingam algorithm for the dynamic update of the shortest paths subgraph after inserting a new edge // Cybernetics and System Analysis. Kiev: Naukova Dumka, 2012. — No. 3. — P. 45–57 (In Russian, English translation by Springer).

[14] Nepomniaschaya A.S. Associative version of the Ramalingam decremental algorithm for the dynamic all-pairs shortest-path problem // Bull. of the Novosibirsk Computing Center, Series: Computer Science. — NCC Publisher, 2016. No. 39. — P. 37–50.

[15] Pagiamtzis K., Sheikholeslami A. Content-addressable memory (CAM) circuits and architectures: a tutorial and survey // IEEE J. Solid-State Circuits. — March 2006. — Vol. 41, No. 3. — P. 712–727.

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

[17] Ramalingam G. Bounded Incremental Computation. — Berlin: Springer, 1996. — (Lect. Notes Comp. Sci.; 1089).

[18] Snytnikova T.V., Snytnikov A.V. Implementation of the STAR-machine on GPU // Bull. of the Novosibirsk Computing Center, Series: Computer Science. — NCC Publisher, 2016. — No. 39. — P. 51–60.

[19] Snytnikova T.V., Nepomniaschaya A.S. Solution of graph problems by means of the STAR-machine being implemented on GPUs // Applied Discrete Mathematics. — The Tomsk State University Publisher, 2016. — Vol. 33, No. 3. — P. 98–115 (In Russian).

[20] Yavits L., Kvatinsky S., Morad A., Ginosar R. Resistive associative processor // IEEE Computer Architecture Letters. — 2015. — Vol. 14, No. 2. — P. 148–151.